STCG: State-Aware Test Case Generation for Simulink Models

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 been widely used in system design, which supports the efficient modeling and synthesis of embedded controllers, with automatic test case generation to simulate and validate the correctness of the constructed Simulink model. However, the increasing complexity of the model, especially the internal states, brings extra challenges to existing model testing techniques such as constraint solving and random search, which results in difficulties when trying to reach the deeper logic of the model effectively.**

In this paper, we propose `STCG`, a state-aware test case generation method for Simulink models. `STCG` 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 novel 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. We implemented `STCG` and evaluated it on several benchmark Simulink models. Compared to the built-in Simulink Design Verifier and state-of-the-art academic work SimCoTest, `STCG` achieves an average improvement of 58% and 132% on Decision Coverage, 52% and 70% on Condition Coverage and 239% and 237% on Modified Condition Decision Coverage, respectively.

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

I. Introduction

Simulink [1] is one of the most widely used model-driven design tools and is increasingly used in embedded scenarios [2], [3], [4]. It supports efficient modeling, fast simulation, and high-quality code generation for embedded control model [5], [6]. For ensuring the security and stability of the model, it is necessary to test the model sufficiently [7]. However, manually constructing test cases not only consumes a lot of effort but also has a difficulty in comprehensively testing the model elements. Automatic test case generation can save significant efforts and cover a lot of logic that is difficult to detect manually [8].

Currently, there are a lot of works on test case generation for Simulink models [9], [10], [11], [12], [13]. These works can be generally classified into two types. One is based on the constraint solving method, such as the Simulink built-in toolkit Simulink Design Verifier(SLDV) [14]. The other is based on the random search method, such as SimCoTest [15]. The former usually transforms the model into a specific formal representation and then uses a formal solver to solve the constraints on the various branch logic in the model. It finally obtains model inputs that satisfy all the constraints. The latter usually generates the input data randomly for the model and obtains the feedback coverage information by executing the model to further optimize the test case generation.

Although the existing works mentioned above have achieved great progress in the Simulink model testing, they are difficult to generate high-coverage test cases for models that contain complex internal states. Since there are many control conditions that require the model in a specific state to be triggered, traditional constraint solving methods are faced with solving for more complex model states, resulting in difficulty to work out feasible solutions in a short time. 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 real industry example 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. Which means that the corresponding task item must exist in the task queue before performing these three operations. For constraint solving methods, it is difficult to obtain an input test case like "add data first and then modify data" directly. Because it is difficult to solve directly for two consecutive operations on an array. Solving for arrays is already very difficult, let alone twice, which makes the problem exponentially more complex [16]. For random search methods, 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".

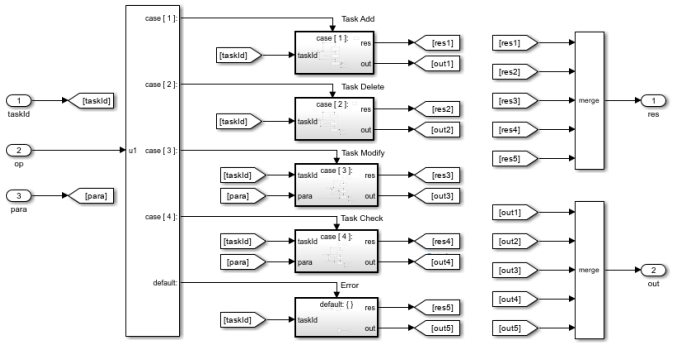


Fig. 1. An example of model with complex internal states. This is an AutoSAR CPU task dispatch model. It mainly contains the four operations of adding, deleting, modifying and checking CPU task in the queue.

To address the above problem, we propose a state-aware test case generation method. 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 the new state space when all the tree nodes are unable to solve for a new coverage out. Based on this 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.

* Yu Jiang is the corresponding author.

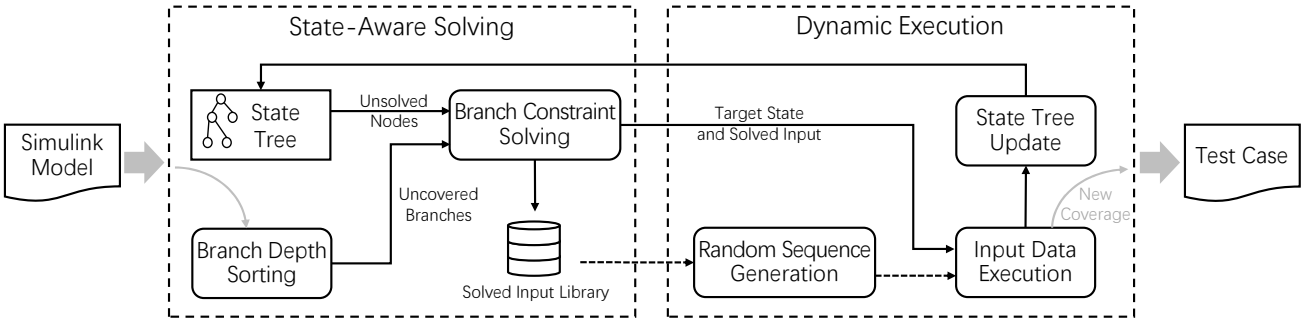


Fig. 2. Overview of STCG. Two parts are executed cyclically to obtain test cases. The State-Aware Solving part focuses on obtaining the one-step input data by constraint solving on one iteration of the model. The Dynamic Execution part focuses on obtaining the specific state of the model by executing the solved input data and outputting test cases.

We implemented and evaluated STCG on several benchmark Simulink models. Compared to the built-in Simulink Design Verifier and the academic work SimCoTest, STCG achieves an average improvement of 58% and 132% on Decision Coverage, 52% and 70% on Condition Coverage and 239% and 237% on Modified Condition Decision Coverage(MCDC), respectively.

II. Related work

Constraint solving based test case generation. It usually uses formal techniques to obtain input cases that satisfy the property requirements. Simulink Design Verifier(SLDV) [14], the built-in validation toolkit of Simulink, leverages symbolic execution to automatically generate test cases for model coverage criteria, including Decision Coverage, Condition Coverage, Modified Condition Decision Coverage, and derive customize test objectives. The work in [9] adopts the model-checking approach to explore the structure of the target model and obtain the subset of nodes that maximizes the observation of mutants. Then, it will generate a small set of test cases to achieve high coverage based on this information. AutoMOTGen [10] describes the Simulink model by a formal language named SAL [17], encodes the coverage specifications in the formal model, and utilizes the built-in model checking tools to perform test case generation.

Due to the lack of perception of internal states, they have difficulty in deriving test cases for models that use internal states as conditions. Unlike these works, STCG records internal states obtained by the dynamic execution and solves the state-aware branch conditions iteratively. Consequently, STCG can explore more state space and achieve higher coverage at a faster speed.

Random search based test case generation. Random search is widely used for testing large models [11], [12], [15], [13]. This approach typically utilizes dynamic simulation to obtain test feedback. Reactis [12] uses Monte Carlo methods to generate test cases for random simulation. It also adopts the guided simulation technique to evaluate output values for selecting test cases for exploring uncovered blocks. REDIRECT [11] focuses on analyzing the feedback from the simulation of the generated test cases and uses a set of heuristics for non-linear blocks. SimCoTest [15] generates test cases for both continuous-time and discrete-time Simulink models.

For deep blocks and internal states inside models, the random search approaches are hard to generate test cases to trigger them and unable to satisfy high-standard coverage criteria like MCDC. Different from them, STCG uses the constraint solving method with internal states to derive precise requirements for satisfying coverage criteria and generate corresponding test cases.

III. STCG Design

Figure 2 shows an overview of STCG. STCG 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 solved on a state for a branch, 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. 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.

A. State-aware Solving

Before we introduce the detailed steps of state-aware solving of branch constraints, 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 Simulink model that has conditional judgment 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 also the number of its all ancestor 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 fact, 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.

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 turn to state S based on the parent state P. SB is a set of the model branch, it records all the model branches solved in this state. And 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 `STCG`. The state tree contains a root node N_0 by default. This root node represents the initial state of the model. Other nodes will be added through the dynamic execution process. Each path in the tree represents an execution trace of the model and also represents one test case.

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: $TargetState$: The selected state node in $StateTree$
$TargetBranch$: 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          curStateData = Node.getState()
9          Model.setState(curStateData) // Switch model state
10         SolvedInput = solve(Model, Branch) // Constraint
             solving for a branch on the current model state
11         Node.setSolved(Branch)
12         if SolvedInput != NULL then
13             // The current state is solvable for target branch
14             TargetState = Node
15             TargetBranch = Branch
16             break
17     if SolvedInput != NULL then
18         break
19 if SolvedInput == NULL then
20     TargetState = NULL
21     TargetBranch = NULL
22 return TargetState, TargetBranch, SolvedInput
```

When the state-aware solving part 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 select shallow model branches in preference to performing constraint solving. It is usually easier to perform solving for the shallow branches because the constraint solver will formalize for less computing logic so that the solution can be obtained in less time. Not only that, the solved inputs that can cover shallow branches will sometimes cover deeper branches in the same execution. In this way, it can avoid solving for those deeper branches, which further reduces the solution time. Then, we traverse all model states in the state tree and all branches of the model to perform state-aware constraint solving.

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 and line 9, the model state value will be taken from the state node and the model state need to be switched. We just bring the model state value as constants rather than variables into the model for solving. In line 10, 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 12-18. As for 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 there is no state in the state tree that can be solved for new branch coverage, the algorithm will return NULL. And then, the dynamic execution part will use the previously solved inputs to construct a random sequence to expand the state space.

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

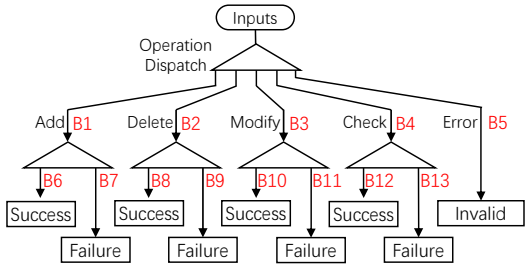
Input: $Model$: The Simulink model for test case generation
$StateTree$: The state tree during test case generation
$TargetState$: The selected state node of Algorithm 1
$SolvedInput$: The solved input of Algorithm 1
$InputLibrary$: All solved inputs from the solver
Output: $TestCase$: The test case that result in new coverage

```
1  TestCase = ∅
2  inputSequence = ∅ // Used for dynamic execution
3  curState = NULL
4  newCover = false
5  if SolvedInput != NULL then
6      // When state-aware solving has a solution
7      curState = TargetState
8      inputSequence.append(SolvedInput)
9  else
10     // When state-aware solving has no solution
11     curState = StateTree.getRandomNode()
12     for N times do
13         curInput = InputLibrary.getRandomInput()
14         inputSequence.append(SolvedInput)
15 for input in inputSequence do
16     curStateData = curState.getState()
17     Model.setState(curStateData) // Switch model state
18     newCover, newState = Model.run(SolvedInput)
19     curState.addChild(newState) // Update state tree
20     curState = newState
21 if newCover then
22     // Get the complete input sequence
23     while curState != StateTree.root do
24         TestCase.addData(curState.getInput())
25         curState = curState.getParentNode()
26 return TestCase
```

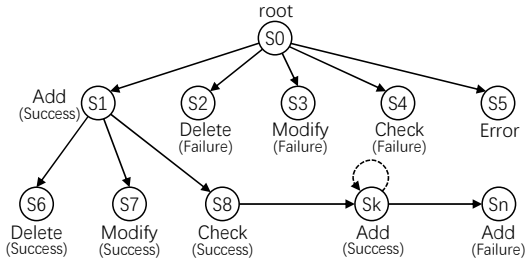
The detailed dynamic execution process is described in Algorithm 2. If the $SolvedInput$ from Algorithm 1 is not NULL, it will be used for one-step execution along with the selected state ($TargetState$). As shown in lines 5-8, this single input will be regarded as an input sequence. When state-aware solving has no solution, we construct a random sequence using the input data that has been previously solved, in lines 9-14. 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 7. And when $SolvedInput$ is invalid, we randomly select a state in the state tree, in line 11. The dynamic execution of the input sequence is shown in lines 15-20. 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 21-25.

C. An running example of STCG workflow

In order to illustrate the working process of STCG more clearly, we use the CPU Task model in Figure 1 as an example for analysis. Figure 3.(a) shows the branch structure of the model after simplification. 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 matched task in the queue. For this model, a possible state tree constructed by STCG is shown in Figure 3.(b). The corresponding construction process is shown in Table I.



(a) Simplified Simulink model branches



(b) State tree constructed by STCG

Fig. 3. An example of STCG corresponding to the model in Figure 1. (a) shows the simplified model branches, which contain a total of 13 branches. (b) illustrates the state tree with the full coverage explored.

The execution conditions of B1-B5 are simple and only require the opcodes of the input data to be the target values. Therefore, based on the root state S0, it is easy to obtain the corresponding input data from the constraint solver. Besides, some branches can be covered by the dynamic execution, like B6, B9, B11 and B13, as shown in steps 1-5 in Table I. Next, we try to solve B7, but we cannot get a valid solution from state S0 to S5 because the CPU task queue needs to be added more tasks to fill it up. Then, B8 fails to obtain a valid solution from state S0 in step 7, but obtains a valid solution from state S1 in step 8 because the state S1 has a CPU task in the queue, so that a task can be successfully deleted. Similarly, B10 and B12 can also obtain valid solutions on state S1. When only B7 is left unsolvable in 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 step 17. During the execution process of STCG, steps 1-5, 8, 11, 14 and 17 will output the test cases.

TABLE I
THE MAIN PROCESS OF CONSTRUCTING THE STATE TREE

Step	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-S5, but failed.				IIIIII..I.I.I
7	Try to solve B8 on state S0, but failed.				IIIIII..I.I.I
8	B8	S1	B8	S6	IIIIII.II.I.I
9	Try to solve B7 on state S6, but failed.				IIIIII.II.I.I
10	Try to solve B10 on state S0, but failed.				IIIIII.II.I.I
11	B10	S1	B10	S7	IIIIII.IIII.I
12	Try to solve B7 on state S7, but failed.				IIIIII.IIII.I
13	Try to solve B12 on state S0, but failed.				IIIIII.IIII.I
14	B12	S1	B12	S8	IIIIII.IIIIII
15	Try to solve B7 on state S8, but failed.				IIIIII.IIIIII
16	Failed to solve B7 on all state tree nodes.				IIIIII.IIIIII
17	Random execution on S8.		B7	S9 - Sn	IIIIIIIIIIII

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

IV. Evaluation

Tool Implementation. STCG[1] is implemented in C++, with 28,288 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). And 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 linearly arranged in memory. 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

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

mapping the "key-value" structure. Similarly, the inputs for each iteration of the model are stored in the state tree nodes by linear memory. 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 STCG, so that a fair coverage comparison can be performed by using the Simulink test block named "Signal Builder".

Experiment Setup. To evaluate the effectiveness of STCG, 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 STCG 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) with the same duration (1 hour). Since both SimCoTest and STCG include random strategies, we repeat the experiment 10 times to obtain the average coverage result for a fair comparison. All benchmark models are derived from the industry and deployed in embedded scenarios. Table II shows the detailed description of these models, including model functionality, number of branches, and number of blocks.

TABLE II
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
UTPC	Underwater thruster power control	92	214
LANSwitch	LAN Switch controller	131	570
LEDLC	LED matrix load control	94	270
TCP	TCP three-way handshake protocol	146	330

Detailed Results. 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. Decision Coverage is concerned with whether different branches of a block with branching logic can be executed. Condition Coverage is concerned with whether conditions affecting boolean logic and branch changes are triggered. MCDC is concerned with the effect of a single condition change on the entire determination.

Table III shows the coverage of the test cases generated by the different tools for benchmark models. Compared to SLDV and SimCoTest, STCG improves the Decision Coverage for 12%-127% (avg. 58%) and 15%-513% (avg. 132%), Conditional Coverage for 18%-144% (avg. 52%) and 16%-144% (avg. 70%), and MCDC for 100%-900% (avg. 239%) and 100%-400% (avg. 237%), respectively. It can be seen that our method achieves good results on these control models with complex internal states. For example, STCG achieves 100% Decision Coverage and 100% Condition Coverage on the CPUTask 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 and NICProtocol, STCG 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 beside the corresponding four ports.

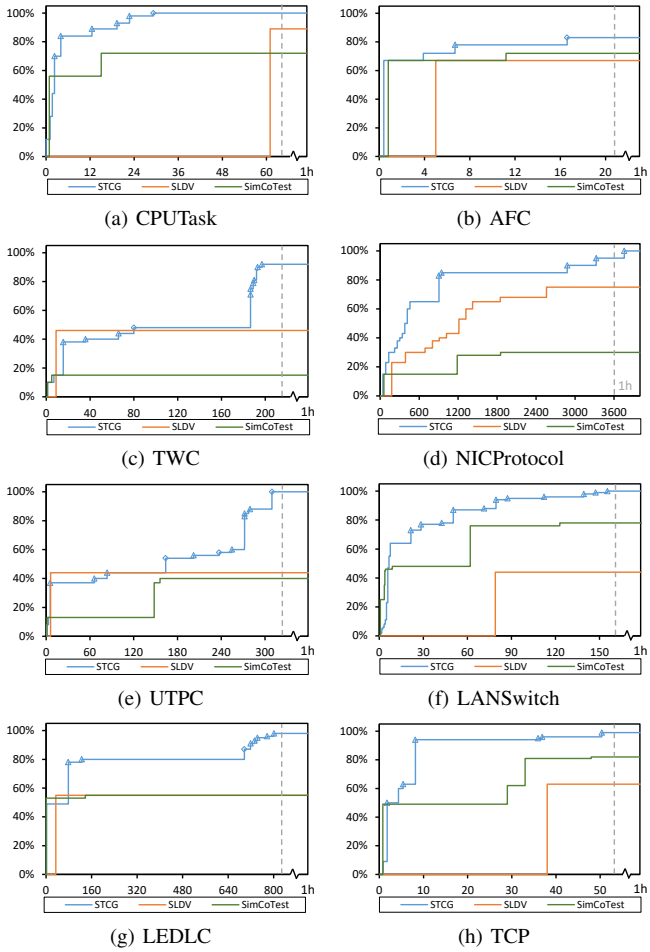
TABLE III
COMPARISON OF THE TEST COVERAGE OF DIFFERENT TOOLS

Model	Tool	Decision Coverage	Condition Coverage	MCDC
CPUTask	SLDV	89%	72%	42%
	SimCoTest	72%	56%	21%
	STCG	**100%**	**100%**	**100%**
AFC	SLDV	67%	64%	11%
	SimCoTest	72%	68%	11%
	STCG	**83%**	**79%**	**22%**
TWC	SLDV	46%	68%	40%
	SimCoTest	15%	57%	20%
	STCG	**92%**	**97%**	**100%**
NICProtocal	SLDV	75%	83%	10%
	SimCoTest	30%	43%	33%
	STCG	**95%**	**98%**	**100%**
UTPC	SLDV	44%	59%	44%
	SimCoTest	40%	58%	44%
	STCG	**100%**	**100%**	**100%**
LANSwitch	SLDV	72%	76%	15%
	SimCoTest	78%	81%	15%
	STCG	**100%**	**98%**	**55%**
LEDLC	SLDV	55%	41%	43%
	SimCoTest	55%	41%	43%
	STCG	**98%**	**100%**	**100%**
TCP	SLDV	63%	64%	33%
	SimCoTest	82%	74%	17%
	STCG	**99%**	**100%**	**67%**
Average Improvement	vs SLDV	↑ 58%	↑ 52%	↑ 239%
	vs SimCoTest	↑ 132%	↑ 70%	↑ 237%

Effectiveness of State-Aware Solving. We also recorded the timestamp of each generated test case of three tools, STCG, SLDV and SimCoTest. Figure 4 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 4, we can see that in most cases, STCG is able to achieve higher coverage at a faster speed, and obtain new 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 STCG, the higher coverage fraction is almost always obtained by our state-aware branch solving. For example, on the TCP model, STCG 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.

Discussion. In our experiments, we found that some branches of the model, such as the TWC and LEDLC model, could not be triggered even after a long solving time and random execution. After analyzing the models manually, we found that this situation is caused by the branch conditions being perpetually false. Due to the fact that



(a) CPUTask (b) AFC

(c) TWC (d) NICProtocol

(e) UTPC (f) LANSwitch

(g) LEDLC (h) TCP

Fig. 4. The folded line plot of the Decision Coverage versus time. The X-axis is time (s) and the Y-axis is Decision Coverage (%). "△" indicates test cases generated by constraint solving based on internal model states, and "◇" indicates test cases generated by executing the random input sequence.

these branches will be solved in every known state in the state tree, STCG performs multiple solving for this type of branch, resulting in a lot of wasted time. For this reason, we should probably verify the unreachable branches using the formal method to improve efficiency.

By comparing with SimCoTest, we found that the random search method is usually able to explore some coverage earlier than STCG, as shown in Figure 4. This is due to the random generation of test cases is faster than constraint solving. If the random method can be introduced into STCG to perform the random generation process first and then use the constraint solving method to solve the remaining uncovered branches, the efficiency of STCG can be further improved. In addition, constructing a random input sequence using only previously solved inputs may not reach some branches, which can be compensated by attaching random methods.

V. Conclusion

In this paper, STCG is proposed to optimize the test case generation of Simulink models with state-aware solving, especially for the control models which 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. Experiments show that STCG can perform well on benchmark Simulink models. Compared to SLDV and SimCoTest, the Decision Coverage from STCG can be improved by 58% and 132%, the Condition Coverage can be improved by 52% and 70%, and the MCDC can be improved by 239% and 237%, respectively. Our future work includes incorporating more constraint solvers to facilitate the efficiency of STCG.

VI. Acknowledgment

We would like to express our deep gratitude to Hongyu Fan for his help in formal verification. This research is sponsored in part by the National Key Research and Development Project (No. 2022YFB3104000, No2021QY0604) and NSFC Program (No. 62022046, 92167101, U1911401, 62021002).

References

[1] Simulink and Matlab, *Simulink Documentation*. [Online]. Available: https://www.mathworks.com/help/simulink/index.html

[2] F. Pasic, "Model-driven development of condition monitoring software," in *Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. ACM, 2018, pp. 162–167.

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

[4] 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.

[5] J. Krizan, L. Ertl, M. Bradac, M. Jasansky, and A. Andreev, "Automatic code generation from matlab/simulink for critical applications," in *2014 IEEE 27th Canadian Conference on Electrical and Computer Engineering (CCECE)*. IEEE, 2014, pp. 1–6.

[6] 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 Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 11, pp. 4504–4515, 2022.

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

[8] 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.

[9] N. He, P. Rümmer, and D. Kroening, "Test-case generation for embedded simulink via formal concept analysis," in *Proceedings of the 48th Design Automation Conference*, 2011, pp. 224–229.

[10] S. Mohalik, A. A. Gadkari, A. Yeolekar, K. Shashidhar, and S. Ramesh, "Automatic test case generation from simulink/stateflow models using model checking," *Software Testing, Verification and Reliability*, vol. 24, no. 2, pp. 155–180, 2014.

[11] M. Satpathy, A. Yeolekar, and S. Ramesh, "Randomized directed testing (redirect) for simulink/stateflow models," in *Proceedings of the 8th ACM international conference on Embedded software*, 2008, pp. 217–226.

[12] R. Cleaveland, S. A. Smolka, and S. T. Sims, "An anstrumentation-based approach to controller model validation," in *Automotive Software Workshop*. Springer, 2006, pp. 84–97.

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

[14] G. Hamon, B. Dutertre, L. Erkok, J. Matthews, D. Sheridan, D. Cok, J. Rushby, P. Bokor, S. Shukla, A. Pataricza *et al.*, "Simulink design verifier-applying automated formal methods to simulink and stateflow," in *Third Workshop on Automated Formal Methods*, 2008.

[15] R. Matinnejad, S. Nejati, L. C. Briand, and T. Bruckmann, "Simcotest: A test suite generation tool for simulink/stateflow controllers," in *Proceedings of the 38th International Conference on Software Engineering Companion*, 2016, pp. 585–588.

[16] V. D'silva, D. Kroening, and G. Weissenbacher, "A survey of automated techniques for formal software verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 7, pp. 1165–1178, 2008.

[17] S. International, *Symbolic Analysis Laboratory*. [Online]. Available: https://sal.csl.sri.com/