

GATES: Accelerating Solving-based Testing through Adaptive Solving and Guided Execution

Haowei Qiu*, Zhuo Su ^{✉†}, Zehong Yu*, Xinhe Tong*, Dongyi Yu[‡], and Yu Jiang*

*KLISS, BNRist, School of Software, Tsinghua University, Beijing, China

[†]School of Software, Beihang University, Beijing, China

[‡]AVIC International Digital Network Technology Co., Ltd., Beijing, China

Abstract—Simulink models are widely adopted in embedded systems development, where rigorous model testing plays an important role in ensuring system reliability. However, efficiently achieving high model test coverage remains a significant challenge due to vast state spaces and intricate state-dependent logic. Although solving-based test case generation is a powerful method, redundant solving often becomes a performance bottleneck. This leads to significant overhead that limits its practical applicability.

This paper introduces GATES, which employs adaptive solving to address valuable tasks and then incorporates guided execution to cover branches more efficiently. Specifically, GATES uses an asynchronous unreachable branch filter to proactively prune the search space. For solvable tasks, the dynamic priority solving task queue is employed to raise the priority of valuable tasks that are capable of covering more branches. After that, GATES duplicates and assembles solved inputs for guided execution. We evaluate GATES on a suite of industrial benchmark models. Compared with the commercial tool SLDV, GATES achieves average improvements of 89.0%, 93.5%, and 330.3% in Decision Coverage, Condition Coverage, and Modified Condition/Decision Coverage (MCDC), respectively. Furthermore, compared to the recent academic tool HSTCG, GATES achieves higher MCDC while reducing the average time to reach peak coverage by 70.0%.

Index Terms—Simulink, Test Case Generation, Solving-based Testing.

I. INTRODUCTION

Simulink is a widely used model-based development tool for designing complex embedded control systems, offering a unified graphical environment for modeling dynamics. Simulink enables simulation and automatic code generation for embedded controllers, which is essential in domains like automotive and aerospace. As the core of industrial system functionality, model safety is of high importance, and therefore model correctness verification is critical [1], [2], [3], [4]. However, manually constructing a comprehensive test suite to verify correctness is time-consuming and error-prone, highlighting the critical need for automated test case generation. Although existing automated techniques can alleviate this burden to some extent, they still encounter major challenges as system complexity increases, particularly when internal state branch constraints become more intricate [5], [6], [7], [8], [9].

Within the Simulink testing community, test case generation methodologies have largely diverged into two primary paradigms: search-based and solving-based approaches. Representative search-based frameworks, such as SimCoTest [10]

and the recently proposed FCT [11], employ meta-heuristic algorithms to optimize continuous signal features, proving effective at rapidly exploring model dynamics and improving output diversity. Conversely, to systematically achieve deep structural coverage, researchers and practitioners often rely on solving-based techniques. While Simulink Design Verifier (SLDV) [12] serves as the primary commercial standard for this rigorous approach, recent academic research has advanced towards dynamic, state-aware solvers. For instance, tools like STCG [13] and its more efficient successor, HSTCG [14], systematically resolve path constraints one simulation step at a time by capturing intermediate execution states. This state-aware precision has proven advantageous, with empirical results demonstrating that HSTCG outperforms both traditional commercial solvers (SLDV) and standard search-based methods when navigating complex, deeply nested models.

Despite these significant advances, the complexity of constraint-solving for stateful models still presents several key challenges, many of which contribute to the overarching problem of redundant or inefficient solving. First, it is difficult to efficiently identify unreachable branches before invoking the solver, which leads to wasted computational effort on impossible tasks. Second, scheduling the order of solving tasks is a critical performance issue. A solver-based method can become stalled on a single complex branch, preventing the rapid coverage of many simpler ones. Third, fully capitalizing on the value of each successful solve remains a challenge. The generated inputs could be strategically combined to accelerate state space exploration through dynamic execution, thereby reducing the need for subsequent, costly solver calls.

To illustrate these challenges, consider a representative industrial application, a Factory Multi-point Temperature Monitor shown in Figure 1. The complexity of this model stems from its large state space, defined by all possible combinations of sensor states (SensorIdUsed) and stored readings (SensorIdData), and the complex, state-dependent logic driven by opcode inputs. Depending on the opcode, different operations such as connecting, reporting, testing, and disconnecting are executed. Each operation triggers different branches based on the current model state and inputs. For example, in the connecting case, there are different branches such as successful connection, connected, connection failure, and connection limit reached. For the successful disconnection branch to be triggered, the current sensor must be successfully connected and not disconnected. This complexity creates obstacles for

Zhuo Su is the corresponding author.

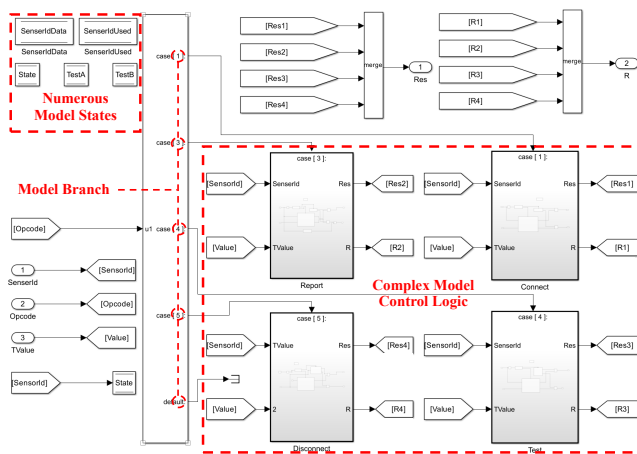


Fig. 1. An example model with complex internal states. This is a Factory Multi-point Temperature Monitor model. It mainly contains the four operations of connecting, reporting, testing and disconnecting.

test case generation. First, the complex interplay between opcodes and state variables may produce unreachable branches. Second, the vast state space makes the solving order critical for efficiency, as exploring from unpromising states creates bottlenecks. Finally, the model's behavior is contextual. The same input might trigger a valid update branch from one state but an error-handling branch from another, where the sensor is disconnected. These issues present difficulties for existing test case generation methods. For a solving-based method, the vast number of state-branch combinations leads to inefficiency. To cover the "disconnect" branch, it might formulate and attempt to solve a constraint for every known state in its set. It would expend significant resources on numerous failed attempts from all states where the sensor is not yet connected. It would also waste cycles proving that certain branches, like reporting on a disconnected sensor, are temporarily unreachable, which results in redundant solving. Meanwhile, a search-based approach struggles with the required logical sequence. The fitness function might guide an input towards a disconnect opcode, but it provides no gradient or reward for first executing the prerequisite connect operation for the same sensor. The probability of a random mutation process discovering this exact, ordered sequence of operations is low, making the targeted discovery of such state-dependent branches difficult.

In fact, redundant solving introduces significant overhead and is hard to avoid. To address these challenges, we introduce GATES. The core of our approach is a state-aware solving engine that maintains a queue of targeted solving tasks. To avoid redundant work, it employs an unreachable branch filter to prune impossible tasks and a dynamic priority task queue to intelligently schedule the remaining tasks based on heuristics. To complement this precision-focused solving, GATES incorporates a guided execution engine. This engine reuses all successfully solved inputs to construct new test sequences, which are then dynamically executed. This process accelerates the coverage of many simple, downstream branches and significantly enhances the ability to explore branches with time-cumulative logic.

We implemented and evaluated the effectiveness of GATES

on a suite of open-source and industrial Simulink benchmarks. Compared with the commercial tool SLDV, GATES achieves average improvements of 89%, 93.5%, and 330.3% in Decision Coverage, Condition Coverage, and MCDC, respectively. Furthermore, compared to the recent academic tool, HSTCG, GATES achieves higher MCDC while reducing the average time to reach peak coverage by 70.0%. In summary, this paper makes the following contributions:

- We designed and implemented GATES that addresses the problem of redundant solving by integrating three key optimization strategies: (1) proactive pruning of the search space with an asynchronous unreachable branch filter; (2) strategic allocation of solver effort using a dynamic priority solving task queue; (3) broad state-space exploration by using a randomized scheduler that reuses successfully solved inputs.
- The experimental results show that GATES significantly outperforms prior methods in both test case generation efficiency and final coverage on industrial models.

II. BACKGROUND AND RELATED WORK

A. Model-Driven Design in Embedded Systems

Model-Driven Design (MDD) is a widely adopted software engineering paradigm that establishes abstract models as primary artifacts in the development lifecycle, shifting the focus away from traditional manual coding. In the context of embedded systems, MDD enables engineers to specify, simulate, and verify system behaviors at an early development stage [15]. By abstracting complex hardware and software interactions into high-level models, MDD helps reduce design defects and development costs.

As a widely used industrial platform for MDD, Simulink provides a graphical block diagram environment equipped with customizable solvers for modeling and simulating dynamic systems [16], [17]. A core feature of Simulink is its ability to explicitly represent both continuous-time dynamics and discrete control logic within a unified framework. This capability enables engineers to execute detailed simulations, validate logic against requirements, and automatically generate C/C++ code for deployment on embedded hardware.

B. Simulink Model Elements

To facilitate automated test case generation and structural analysis, we formalize the core elements of a Simulink model that are directly relevant to logical execution and coverage metrics.

1) Blocks: Blocks are the fundamental computational and functional units within a Simulink model. They process input signals according to predefined mathematical or logical operations to produce output signals.

2) Connections: Connections are the primary data-flow conduits responsible for propagating continuous or discrete signals between blocks. They establish the structural topology of data transmission within the model.

3) Branches: Branches represent the control-flow logic within the model, such as paths within a Switch or If-Action subsystem. A branch is controlled by a specific Boolean

condition, and its corresponding execution path is activated only when this condition is satisfied during simulation. While connections act as the physical pathways, branches operate separately as the conditional logic that determines which specific paths, blocks, or subsystems are actively executed.

4) Internal States: Many embedded control models exhibit state-dependent and time-cumulative behaviors, which are governed by internal states. Blocks such as *Unit Delay* function as memory elements that store values from previous execution steps, making the activation of certain branches dependent not only on current inputs but also on the historical execution trace.

C. Related Test Case Generation Approaches

Automated test case generation for Simulink models is an active research area that aims to improve the efficiency and effectiveness of system testing [13], [14], [18], [9]. Historically, conventional software testing methods have fallen into three main categories: dynamic search-based [19], symbolic execution-based [20], and hybrid approaches that combine both techniques [21]. For instance, a widely used dynamic approach is coverage-guided fuzzing, which executes a program with a high volume of inputs, typically generated through mutations of a seed corpus, and uses feedback such as code coverage to discover new execution paths, as implemented in tools like Libfuzzer [22] and AFL++ [23]. While traditional tools focus on code coverage, model testing requires satisfying structural metrics such as DC, CC, and MCDC. Furthermore, model behavior is often state-dependent, meaning that covering a branch at simulation step i requires a specific sequence of prior inputs. This temporal dependency presents a challenge for traditional methods that are not designed to reason about extended and precise input sequences. Consequently, model-specific testing approaches have been developed, which can be broadly categorized into search-based generation and solving-based generation.

Search-based software testing (SBST) defines test case generation as a search problem, employing meta-heuristic algorithms such as Genetic Algorithms to discover test cases [24], [25]. Instead of logically proving a path is satisfiable, the search is guided by a fitness function, which quantifies how close a given test input is to achieving a test objective. For example, a fitness function might reward inputs whose signal values progressively approach a `switch` block's activation threshold. This makes SBST scalable and capable of exploring the extensive input domains of complex, non-linear, or black-box models where formal analysis is intractable. Building upon this quantitative optimization paradigm, Bartocci et al. recently introduced Signal Feature Coverage and Testing (FCT) [11]. FCT explicitly targets Cyber-Physical System (CPS) dataflow models by employing an optimized Particle Swarm Optimization (PSO) algorithm. Instead of merely tracking discrete structural branches, FCT defines its fitness landscape based on specific signal features such as amplitude variations or temporal patterns, allowing the PSO search to efficiently navigate the input space and optimize these continuous dataflow characteristics. Parallel to meta-heuristic optimization, coverage-guided fuzzing has emerged

as another effective dynamic exploration technique. Tools such as CFTCG [26] adapt fuzzing for Simulink by utilizing code-level instrumentation and mutation-based input generation. By continuously mutating inputs and monitoring execution feedback, CFTCG rapidly explores the model's shallow structural state space without the computational overhead of constraint solving. However, the primary limitation of these search-based and fuzzing methods is their lack of precision when encountering state-dependent discrete logic. While continuous fitness functions such as those in FCT are effective at guiding signal trajectories, they often encounter fitness plateaus when facing exact Boolean transitions, strict equality checks, or deeply nested logical conditions. Similarly, mutation-based fuzzers such as CFTCG have difficulty generating the precise, multi-step input sequences required to traverse prerequisite historical states. Consequently, these methods can become trapped in local optima, restricting their ability to systematically satisfy the time-cumulative path conditions required to uncover structural faults.

In addition, methods focusing on falsification testing have been increasingly applied to Cyber-Physical Systems, as demonstrated in the ARCH competitions [27]. Representative search-based tools such as Breach [28], S-TaLiRo [29], and FalStar [30] utilize optimization algorithms guided by quantitative semantics to find specification violations. Breach performs sensitivity analysis and parameter synthesis by applying gradient-based searches over dense-time continuous trajectories. S-TaLiRo leverages multiple stochastic search engines coupled with Monte Carlo techniques to navigate the continuous dynamics of non-linear hybrid systems. Additionally, FalStar accelerates the falsification process by combining meta-heuristic search with active machine learning and probabilistically adaptive inputs to minimize computationally expensive system simulations. To bridge the gap between requirement-driven testing and structural adequacy, researchers have explored integrating coverage metrics directly into this process. For instance, Dokhanchi et al. proposed a requirements-driven falsification framework that utilizes structural coverage metrics to guide the search in Simulink models [31]. Recent advancements, such as the adaptive testing framework proposed by Bartocci et al., dynamically adjust input parametrizations and search distributions to improve specification coverage [32]. While effective for continuous and hybrid systems with continuous signal behaviors, these optimization-based adaptive methods treat the internal discrete control flow largely as black-box. This abstraction can cause their continuous fitness gradients to provide suboptimal guidance when facing state-dependent and time-cumulative logic. A meta-heuristic algorithm might optimize a signal to maximize an immediate, local objective at the current simulation step. However, this local optimization may structurally violate a prerequisite state accumulated over previous steps, trapping the search trajectory in a local optimum. Solving-based formal methods address these discrete and historical complexities by executing exact logical deductions.

Solving-based methods address these precision limitations by formalizing the test case generation problem as a constraint satisfaction problem [33], [8], [34]. They typically translate

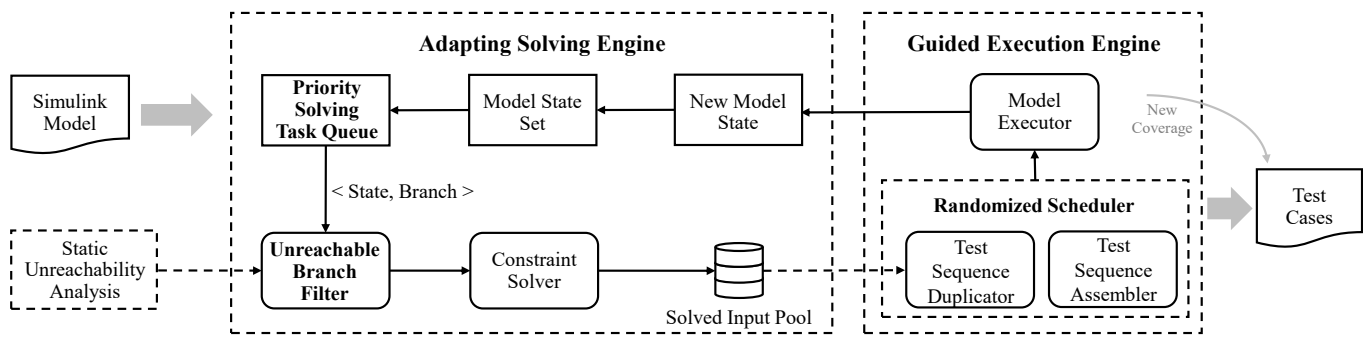


Fig. 2. The architecture of GATES. The system is designed around two collaborating parts to improve efficiency and coverage. The adaptive solving engine optimizes the use of the constraint solver by filtering unreachable branches and prioritizing tasks that are most likely to reach new coverage. The guided execution engine improves the use of every solved input by using a randomized scheduler to strategically replay and combine them, uncovering new test paths without costly solver calls.

the Simulink model into a formal mathematical representation and utilize a formal verification engine, such as a Bounded Model Checker or a Satisfiability Modulo Theories solver, to find input vectors that satisfy a given test objective. A common technique is to unroll the model's control logic for a bounded number of simulation steps, enabling a backend solver to deduce a multi-step input sequence. Simulink Design Verifier (SLDV) [12] is a widely used commercial tool for this paradigm. SLDV operates by translating Simulink semantics into a formal representation and employing formal engines to unroll the model, finding input sequences for structural coverage objectives such as MC/DC. HSTCG is a recent state-aware academic tool designed to mitigate the redundant solving problem in traditional bounded model checkers. Unlike traditional solvers that unroll constraints from the initial state, HSTCG captures, caches, and reuses intermediate model states as starting points for subsequent branch solving, thereby avoiding the repeated resolution of identical historical paths. To navigate the search space, its algorithm traverses the branch dependency tree in a depth-first order. To further control state-space explosion, HSTCG relies on static structural heuristics such as duplicate state filtering. This mechanism prunes the search tree and aborts task scheduling when a newly generated state is identical to a previously explored one.

GATES differs from these baselines in two main aspects. First, while HSTCG uses a static priority scheme, the DP-STQ in GATES dynamically updates the priority of solving tasks based on feedback from previous solving results, such as tracking and prioritizing branches dependent on sensitive variables. Second, GATES employs a guided execution engine to expand the exploration of the state space by recombining and duplicating resolved inputs. In contrast, HSTCG defers its exploration phase until after it attempts to solve from all currently discovered states. This difference enables GATES to process state-dependent branches and reach peak coverage in less time.

III. GATES DESIGN

Figure 2 provides an overview of GATES, a system that takes a Simulink model as input and generates test cases as output. GATES is composed of two interconnected parts that operate in a synergistic cycle: the adaptive solving engine and the guided execution engine. This architecture's objective is to

create a feedback loop in which the solving engine performs precise, targeted analysis to overcome complex branch conditions. Meanwhile, the execution engine uses these successes to rapidly explore the model's state space. For a given model, GATES maintains a model state set containing all previously discovered states. The dynamic priority solving task queue module selects a $\langle State, Branch \rangle$ pair to target next, based on heuristics designed to improve coverage gain. This task is first passed through an unreachable branch filter module, which prunes impossible tasks identified by a static unreachability analysis during runtime. If the task is feasible, a constraint solver is invoked to generate a single input i that covers the target branch from the given state. This newly solved input i is then passed to the guided execution engine. Here, the randomized scheduler takes this input, along with others from the solved input pool, and uses its test sequence duplicator module and test sequence assembler module to construct extended test sequences. These sequences are then run on the model executor. This execution engine facilitates the coverage of subsequent, simpler branches and enables the discovery of new model states. These new states are fed back into the model state set of the solving engine, providing fresh starting points for the next cycle of adaptive solving.

A. Adaptive Solving Engine

The adaptive solving engine is designed to overcome the complex, state-dependent branch conditions that elude random and search-based methods.

While adaptive testing algorithms have been applied in prior work, such as falsification tools and specification refinement frameworks, they are predominantly optimization-based.

These methods continuously adapt input parameterizations based on quantitative robustness metrics derived from Signal Temporal Logic (STL) specifications. Specifically, STL provides a formal robustness semantics that quantifies the continuous degree to which a system's trajectory satisfies desired temporal properties. This quantitative feedback is then utilized to construct fitness landscapes, effectively guiding meta-heuristic searches through continuous or hybrid system dynamics. However, these existing algorithms are not directly applicable to our context. GATES is a solving-based engine that addresses the state-explosion problem in discrete logic, where the core challenge is scheduling state and branch logical

proofs for an underlying SAT or SMT solver. Gradient-driven adaptation is often less effective for state-dependent discrete control logic, such as a sequence of specific opcodes. To address this limitation, we designed a dynamic priority solving task queue (DPSTQ) tailored for formal solving. Instead of continuous fitness feedback, the DPSTQ utilizes white-box structural heuristics, such as state variable sensitivity, to schedule discrete logical tasks.

Its primary objective is not only to generate test inputs but also to do so while improving efficiency by automatically managing solver resources. As shown in Figure 2, the engine operates on a model state set, which contains all of the model's internal states discovered during execution. The engine's task is to select an uncovered branch, pair it with a promising state from this set, and invoke a constraint solver to find an input that covers that branch from that specific state.

To address the limitations of industrial model testing, we designed the engine around two key modules. Specifically, an unreachable branch filter eliminates infeasible tasks, and a dynamic priority solving task queue schedules the remaining tasks to improve generation efficiency.

These modules work in concert to ensure that the expensive constraint solver is deployed on tasks that are both feasible and strategically valuable.

1) *Dynamic Priority Solving Task Queue*: The test case generation process can be defined as an optimization problem that we must continuously allocate solver resources to the task with a high probability of success and potential impact on coverage. We designed a dynamic priority solving task queue to replace the static task list. It constantly calculates the priority of every $\langle State, Branch \rangle$ solving task based on several heuristic strategies. Specifically, these heuristics include branch depth, solving result feedback, and sensitive variable tracking, which are chosen based on the inherent structural data-flow dependencies and dynamic execution characteristics of Simulink models.

The prioritization process is a continuous cycle of initialization, execution, and dynamic adjustment. When a model is first loaded, we perform a depth-first traversal of its branches. The depth of each branch serves as our baseline heuristic for its difficulty because deeper branches often have more complex, cumulative path conditions. The queue is then initialized with a set of solving tasks pairing the model's root state with every branch in the model, with initial priorities inversely proportional to their depth, which means shallower branches are prioritized. In each cycle, the task with the highest priority is dequeued and sent to the constraint solver after passing through the unreachable branch filter module. The outcome of this attempt, whether it is a success (a valid input is found) or a failure (the solver fails to yield any result within a predefined time limit), is used to dynamically update the priorities of related tasks in the queue, as detailed in Algorithm 1. If the solver fails to find an input for a given $\langle StateS, BranchB \rangle$ pair, it suggests the task is difficult. To prevent the system from getting stuck, the algorithm penalizes this path. As shown in the *else* block of Algorithm 1 (lines 13-17), the priority of all tasks involving either the failed branch b_{fail} or originating from the failed state s_{fail} is lowered. A successful solve

provides a wealth of information that is used to guide the next steps, as shown in the *if* block of the algorithm (lines 1-12). The priorities are updated in three ways:

- 1) **Reward Productive States**: The algorithm increases the priority of all remaining tasks originating from the successful state $S_{success}$ (lines 3-5). This state has proven to be a productive starting point, and it is likely that other branches can also be covered from it.
- 2) **Focus on "Sensitive Variables"**: The algorithm identifies which state variables have changed ("sensitive variables") and provides a significant priority boost to all other uncovered branches whose activation conditions depend on them (lines 6-8). This is based on the logic that the change in these variables has fundamentally altered the model's context, making these branches much more likely to be solvable.
- 3) **Incorporate New States**: Finally, the newly discovered model state is added to the global set, paired with all remaining uncovered branches to form a new set of solving tasks, and added to the queue with a depth-based initial priority (lines 9-12).

Algorithm 1: Update Task Queue

Input: task: The task just attempted
Input: outcome: SUCCESS or FAILURE
Input: new_state: The new state if successful
Input: sensitive_vars: Changed variables if successful
Output: Queue is updated

```

1 if outcome == SUCCESS then
    // 1. Reward the productive state
2      $S_{success} \leftarrow task.state$ 
3     for  $t_q$  in Queue do
4         if  $t_q.state == S_{success}$  then
5             Queue.raise( $t_q$ )
    // 2. Boost tasks for sensitive
    // variables
6     for  $t_q$  in Queue do
7         if  $t_q.branch.isDependentOn(sensitive\_vars)$ 
8             then
9             Queue.raise( $t_q$ )
    // 3. Incorporate new state
9     StateSet.add(new_state)
10    for  $b$  in UncoveredBranches do
11        new_task  $\leftarrow CreateTask(new\_state, b)$ 
12        Queue.add(new_task)
13 else
    // Penalize the failed path
14     $s_{fail}, b_{fail} \leftarrow task.state, task.branch$ 
15    for  $t_q$  in Queue do
16        if  $t_q.state == s_{fail}$  or  $t_q.branch == b_{fail}$  then
17            Queue.lower( $t_q$ )
18 return
```

Consider the model in Figure 3, where a set of branches (labeled as B_1 for Branch 1, B_2 for Branch 2, and B_3 for Branch 3) is only solvable from a specific "gateway" state, S_1 , among N total states. A simple scheduler without dynamic priorities would attempt to solve these branches from every state, with a brute-force search. Our priority queue begins by attempting tasks from various states. As it inevitably fails on tasks originating from unproductive states (e.g., $\langle S_i, B_1 \rangle$), it applies a failure penalty, systematically lowering their

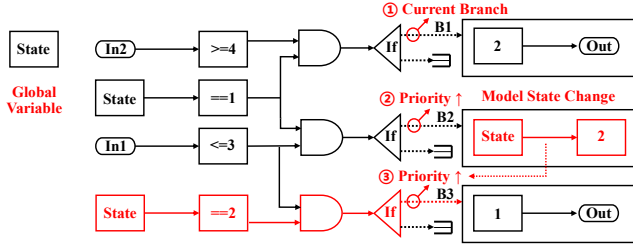


Fig. 3. An example of a dynamic priority queue illustrating the process of solving 3 uncovered branches. $\langle S_1, B_1 \rangle$ is the current task. After solving, the priority of $\langle S_1, B_2 \rangle$ is raised. And $\langle S_2, B_3 \rangle$ will be solved afterward due to the change of state.

priority. This process of elimination naturally promotes the current task, $\langle S_1, B_1 \rangle$, to the top of the queue. Upon its successful solution, the engine applies two heuristics: the priority of all tasks from the now-proven “productive” state S_1 is increased, allowing task $\langle S_1, B_2 \rangle$ to be prioritized for resolution. More critically, if this solution causes the state variable to change, creating a new state S_2 , the sensitive variable heuristic identifies any uncovered branches (Branch 3) dependent on mode and assigns a high priority to the new task $\langle S_2, B_3 \rangle$.

The efficacy of the sensitive variable heuristic is rooted in the structural data-flow dependencies inherent in Simulink models. Formally, let B be the set of uncovered branches and V_{sens} be the set of variables whose values changed in the latest successful execution trace. The heuristic assumes that the probability of a branch $b \in B$ becoming satisfiable is positively correlated with the intersection of its support variables $Supp(b)$ and V_{sens} . This is justified by the fact that V_{sens} represents the dimensions of the state space where the system has recently shown movement. By prioritizing branches where $Supp(b) \cap V_{sens} \neq \emptyset$, GATES focuses on the frontier of the reachable state space.

However, the performance of the sensitive variable heuristic is subject to certain boundary conditions. In high-coupling models, where the dependency graph is nearly complete (i.e., $|Supp(b)|$ is large for most b), the sensitivity feedback becomes less discriminative, potentially leading to an effect where the scheduling reverts to a less informed traversal. Furthermore, in models within branches with time-cumulative logic, the sensitive variable heuristic might not provide timely guidance. In these scenarios, GATES is designed to balance the solving-based heuristics with the guided execution engine to maintain exploration momentum.

2) *Unreachable Branch Filter*: A factor of inefficiency in solving-based test case generation is the time wasted attempting to solve for branches that are, in fact, impossible to cover under any circumstance. As shown in Figure 4, common unreachable branches may be caused by model parameters as in Figure 4a. Since model parameters cannot be changed during execution, the preconditions containing model parameters may never be satisfied. They may also be caused by deep boolean logic as in Figure 4b. In this example, the input is constrained by saturation within the range of value -1 to value 1 . It then passes through a series of complex non-assignment blocks. Finally, the value is evaluated in a deep branch to

determine if it is greater than the value 1 . Our unreachable branch filter is designed to proactively identify and avoid these tasks before they are assigned to the constraint solver. The process is similar to constraint-solving in that it uses CBMC to determine if a branch is satisfiable [35]. However, the crucial difference is that this analysis is independent of states. Instead of starting from a specific model state, the unreachable branch solver treats all global variables and persistent states within the model as symbolic inputs. It then asks the solver to find any valid input sequence that can satisfy the branch’s path condition. If the solver returns unsatisfiable, it proves that the branch is globally unreachable, regardless of the model’s state. These branches are then permanently marked, preventing the constraint solver from ever wasting resources on them.

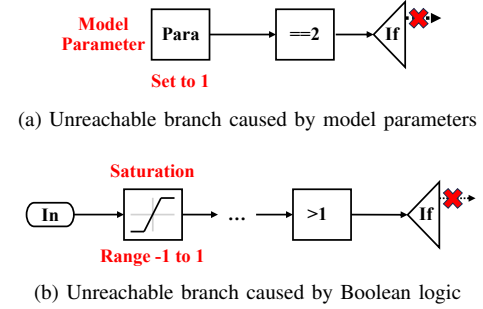


Fig. 4. Two common examples of unreachable branches: (a) illustrates unreachable branches caused by model configuration parameters. (b) illustrates unreachable branches resulting from deep Boolean logic.

To avoid blocking the primary test case generation workflow, we run the unreachable branch solver in a separate, dedicated thread to handle this intensive analysis. This thread works continuously in the background, analyzing the model’s branches one by one. Communication with the solving engine is managed efficiently through a shared file that acts as a dynamic “exclusion list” of unreachable branches. As soon as the unreachable branch solver proves a branch is unreachable, it immediately writes the branch’s unique identifier to this file. Before the adaptive solving engine attempts any new $\langle State, Branch \rangle$ task, it first queries this file. If the target branch is listed, the task is instantly discarded, saving the entire cost of a solver invocation.

In this asynchronous design, the adaptive solving engine and the unreachable branch filter module operate in parallel. To expedite the delivery of unreachable branch data and mitigate redundant resource expenditure, we developed two optimizations to accelerate the filtering process.

Tree-Based Dependency Analysis: Our first optimization exploits the inherent hierarchical relationship between model branches. The reachability of any non-leaf branch is entirely dependent on the reachability of at least one of its children. With this property, we can avoid the need to solve every single branch in the model. Instead, our solver focuses on the leaf branches of the branch tree. When a leaf branch is proven reachable, we recursively propagate a “reachable” status up to all of its ancestor nodes. After all leaf branches have been processed, we perform a final traversal of the entire tree. Any internal branch that has not been marked as reachable is, by logical inference, also unreachable. As shown in the Figure 5,

we give two common branch tree examples to illustrate the two solution processes. We use circles to represent branches, and lines to indicate parent-child relationships between branches. The color green indicates that the branch is reachable, and the color red indicates that it is unreachable. In the Figure 5a, we will solve B_4 first. After obtaining the result, we will recursively mark B_2 and B_1 as reachable. While in the Figure 5b, after B_4 fails to be solved, we try to solve B_5 , which also fails. Since all sub-branches of B_2 are unreachable, B_2 will also be marked as unreachable in the final traversal.

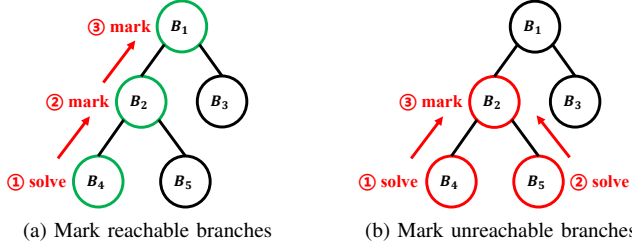


Fig. 5. An example illustrating how tree-based dependency analysis solves unreachable branches. (a) shows multiple reachable branches being marked at once by solving B_4 (green indicates reachable). (b) shows B_2 being marked as unreachable by solving B_4 and B_5 (red indicates unreachable).

Dynamic Coverage Feedback: Our second optimization creates a feedback loop from the main test case generation process back to the unreachable branches filter module. The guided execution engine continually covers new branches as it runs. Before the unreachable branches filter module invokes the expensive solver on any leaf branch, it first checks a shared record of the coverage achieved by the guided execution engine. If the target branch has already been covered, there is no need to prove its reachability. In this case, the unreachable branches filter module skips the solver call entirely and marks the branch and all of its ancestors as reachable. This optimization enables the unreachable branches filter module to utilize the “easy wins” identified by the main engine, thereby freeing up its own resources to concentrate on the ambiguous and challenging-to-analyze branches.

By combining tree-based reduction with dynamic coverage data, the unreachable branch filter module efficiently identifies impossible tasks before they are attempted by the constraint solver. This dynamically focuses the adaptive solving engine’s resources exclusively on meaningful and potential tasks.

Treating state variables as unconstrained symbolic inputs during this filtering phase introduces specific theoretical implications. By unconstraining these variables, the filter explores an over-approximation of the system’s actual reachable state space. If the solver determines a branch is unsatisfiable under these relaxed conditions, the branch is proven to be structurally dead in any valid execution scenario. Consequently, this over-approximation prevents the system from discarding genuinely reachable branches. However, the filter is not complete. Because it does not enforce complex state invariants or multi-step constrained state evolution, the filter may produce false negatives by failing to identify branches that are unreachable due to strict historical state dependencies. This incompleteness represents a pragmatic trade-off. The primary objective of

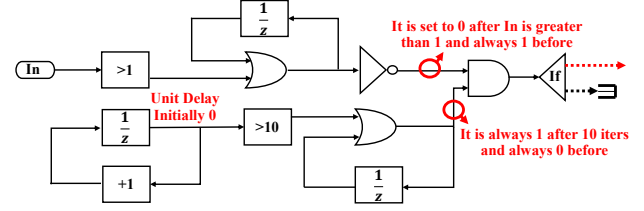


Fig. 6. An AND block with an input threshold and accumulation logic that outputs true if and only if the input is always less than the threshold over multiple iterations.

this filter is the rapid pruning of structurally dead branches often introduced by conservative modeling practices. If an unreachable branch is not identified by this filter, the overall solving process remains stable. The main thread will eventually attempt the branch, encounter a solving timeout, and demote its priority via the DPSTQ, thereby mitigating computational overhead.

B. Guided Execution Engine

While the adaptive solving engine provides depth and precision, the guided execution engine focuses on breadth and speed. It serves as the synergistic counterpart to the solver, with the primary objective of improving the use of every successfully solved input. Instead of relying solely on the solver to find a path to every single branch, this engine uses the solver’s outputs to explore the model’s state space through multi-modal execution. This process is effective at covering the many “easy” branches that follow a difficult-to-reach gateway, and at discovering new model states without requiring expensive solver calls. The core of the guided execution engine is the randomized scheduler, which orchestrates the execution strategy. Upon receiving a new, successfully solved input from the adaptive solving engine, the scheduler initiates a multi-modal execution strategy designed to probe the model’s behavior in different ways. This is accomplished using two key modules: the test sequence duplicator and the test sequence assembler. The scheduler sequentially implements the three distinct execution modes detailed in Table I.

The model in Figure 6 demonstrates the advantages of the randomized scheduler in covering cumulative logic branches. While this example uses an accumulation counter, this mode is broadly applicable to other scenarios necessitating state stabilization, such as timers requiring sustained activation signals, digital filters needing steady inputs to overcome delay buffers, or handshake protocols demanding repeated acknowledgment pulses. In Figure 6, the Unit Delay block acts like a register, storing the value of the current iteration and using it in the next iteration, with an initial value 0. One of the two conditions in the AND logic is always true after the input is greater than 1, and remains false before that. The other condition is always true after 10 iterations, and remains false before that. Therefore, for the AND logic to output true, the input must remain less than or equal to value 1 for at least 10 consecutive times. Such deep temporal conditions are difficult for a bounded solver to solve. However, when solving another shallow branch and obtaining an input less than the value 1, the Repetitive Sequence Execution mode of

the randomized scheduler can quickly duplicate it to create a long, sustained sequence. This constrains the input to remain less than the value 1 consistently over time, efficiently bridging the temporal gap to cover this branch.

During any of these execution modes, if a new branch is covered, the engine records this as a success. It then queries the stored model state tree recursively, tracing the path from the root state to the newly reached state. The complete sequence of inputs along this path is extracted and saved as a test case. Finally, and most importantly for the feedback loop of the framework, the newly discovered model state is passed back to the adaptive solving engine. This new state enriches the model state set, providing a new starting point for the solver to target the next set of branches. This completes the synergistic cycle of precise solving and broad exploration.

The complete integrated workflow of GATES is summarized in Algorithm 2. This algorithm illustrates the cooperative loop between the adaptive solving engine and the guided execution engine. Lines 1-6 initialize the system and populate the priority queue with initial tasks for the model's root state. Lines 7-8 perform asynchronous static analysis of the unreachable branch filter. The main loop then begins. At each iteration, the task with the highest priority in the queue is dequeued. Lines 10-11 query the unreachable branch filter to avoid unreachable branches, and line 12 attempts to generate a single new input. Lines 14-23 dynamically adjust the priority queue based on the solution results, including lowering the priority for failed solutions, raising the priority for successful solutions, and adding new solving tasks. Line 18 uses a randomized scheduler to generate and execute various test sequences to discover new states and coverage. A critical feedback loop is completed when any newly discovered states are used to generate new high-value tasks for the solver's priority queue. This integrated process continuously alternates between precise solution and extensive execution, allowing the system to effectively balance deep analysis and rapid state space exploration.

IV. EVALUATION

A. Tool Implementation

GATES is implemented in 32,150 lines of C++ code as a fully automated, generic framework. Rather than relying on model-specific scripts, our workflow begins with a universal code translator. This translator takes an arbitrary compatible Simulink model (.slx or .mdl) as input, automatically extracts its hierarchical block structures and branch constraints, and

Algorithm 2: Guided Adaptive Test

Input: Model: The model for test case generation
Output: OutTestCase

```

1 StateSet ← ∅           // States already explored
2 Queue ← ∅              // Solving task queue
3 SolvedPool ← ∅         // Solved task results
4 Branches ← AllBranches // Initialization:
   All branches to be covered
5 StateSet.add(model.getInitialState())
6 Queue.add(StateSet, Branches)
7 UBF =
   newThread(RunUnreachableBranchSolver, Model)
8 UBF.start()           // Start threads and detect
   unreachable branches in parallel
9 for task in Queue do
10  if UBF.find(task.branch) then
11    continue           // Skip unreachable branches
12  res = solve(task)
13  if res == NULL then
14    Queue.lower(task)  // Lower the priority
   of tasks
15    continue
16  else
17    SolvedPool.add(res)
18    newStates, Branches, RandomTestCases =
   rdom_scheduler(task.state, res)
19    // Guided execution, updating branch
   and state information
20    StateSet.add(newStates)
21    Queue.add(newStates, Branches) // Add new
   solving tasks
22    Queue.raise(task, newStates) // Raise the
   priority of tasks
23    OutTestCase = res
24    return OutTestCase
25 return NULL

```

generates the necessary C++ testing environment and execution scripts. For all constraint-solving tasks, we translate the logical conditions of a Simulink branch into a Boolean Satisfiability (SAT) formula and employ the lightweight MiniSAT solver to find a satisfying assignment. Regarding input parameterization, GATES currently generates exact, discrete-time input vectors for each targeted simulation step. Consequently, the synthesized test sequences effectively operate as piecewise constant signals over the simulation time. GATES does not currently utilize continuous interpolation functions (e.g., linear or piecewise cubic splines) for signal generation, as our primary focus is satisfying precise, discrete logical

TABLE I
EXECUTION MODES OF THE RANDOMIZED SCHEDULER.

Mode	Description
Direct Application	Validates the solver's output and confirms coverage of the target branch. The newly solved input i is applied directly to the specific model state S for which it was generated. This serves as the foundational step of the execution process.
Repetitive Sequence Execution	Targets stateful logic that requires prolonged, consistent stimuli, such as counters, accumulators, timers, or filter delay buffers. The Test Sequence Duplicator creates a long sequence by repeating the new input i . This sequence is then executed from the context of all known model states.
Long Sequence Assembly and Execution	Designed for broad, opportunistic exploration to discover complex state interactions. The Test Sequence Assembler constructs a long, novel sequence by randomly sampling and combining inputs from the entire solved input pool. This sequence is then executed from a randomly selected state to guide the model into new regions of its state space.

thresholds for structural coverage.

For the unreachable branch filter, this translation is performed in a state-agnostic manner. The model's state variables are treated as free inputs within the generated SAT formula, allowing the solver to determine if a branch is reachable from any possible state. The identifiers of unreachable branches are then written to a shared file, which allows for asynchronous communication with the solving engine. The dynamic priority solving task queue module is implemented using a max heap that minimizes the overhead associated with re-prioritizing tasks after each solve attempt. We provide scripts to manage the execution of experiments with different configurations and to automatically convert the generated test suites into a standard CSV format. As a final verification step, these CSV files are executed by the original Simulink model to validate the final coverage statistics reported in our evaluation.

B. Evaluation Setup

To evaluate our approach, we compared GATES against Simulink's built-in verification toolkit, Simulink Design Verifier (SLDV), Signal Feature Coverage and Testing (FCT), and an academic research tool, HSTCG.

Our evaluation was performed on a diverse benchmark suite of publicly available Simulink models. Crucially, this includes standard benchmarks from the ARCH competition, such as the Engine air-fuel Control (AFC) and the Automatic Transmission System (ATS) models, which are recognized as rigorous testing subjects for hybrid and stateful systems. The remaining models are sourced from SLNET [36], a widely used Simulink benchmark suite, and industry pipelines. The detailed characteristics are provided in Table II.

All experiments were conducted on a server equipped with an Intel i7-13700F CPU and 32 GB of RAM, running Windows 11, SLDV 4.8. To establish a fair comparison, each tool was allocated a fixed time budget of 3600 seconds per model. To account for stochastic elements in the generation process, we performed 10 independent runs for each tool-model pair. For the coverage metrics (Decision, Condition, and MC/DC), we report the average values across these 10 runs. However, to evaluate the peak efficiency of test case generation, we record and report the fastest (minimum) time taken to reach the target coverage among the 10 runs. For specific timing results, see the reproduction package at <https://anonymous.4open.science/r/GATES-85F5/>. GATES produced nearly identical results across each repetition because the solution path remained largely unchanged, and most of the operation time was spent on the deterministic solver. In contrast, the randomized scheduler takes very little time, resulting in minimal variance.

C. Evaluation on Coverage Rate

We use three widely used coverage metrics, including Decision Coverage, Condition Coverage, and Modified Condition/Decision Coverage (MCDC), to evaluate the effectiveness of the generated test cases [37], [38], [39], [40]. A higher coverage percentage signifies a more comprehensive and rigorous examination of the model's internal logic. The goal of

TABLE II
THE DESCRIPTION OF BENCHMARK MODELS

Model	Functionality	UB/TB*	Block
AFC	Engine air-fuel control system	0/35	125
FMTM	Factory Multi-point Temperature Monitor	8/318	713
LANSwitch	LAN Switch controller	2/194	635
LEDLC	LED matrix load control	11/189	403
NICProtocol	Vehicle NIC communication protocol	1/164	394
RTVibration	Vibration signal processing system	10/316	45
SDVProve	SDV-based switching condition verifier	0/771	238
TCP	TCP three-way handshake protocol	2/357	495
TCS	Control model dedicated to testing	3/109	2188
ATS	Automatic Transmission System	0/68	13
DCM	Dynamic Compression Model	51/574	1348

* 'UB' and 'TB' represent the number of unreachable branches and total branches, respectively.

this evaluation is to determine the peak achievable coverage for each tool on every model, representing the upper limit of its exploration capability within the experiment's time limit. Table III shows a direct comparison of the peak coverage values achieved by GATES, SLDV, FCT, and HSTCG for the benchmark models, with all results averaged over 10 independent runs.

The distribution of these results across the benchmark suite is visualized in Figure 7. The results clearly demonstrate the effectiveness of GATES's architecture. Both GATES and HSTCG achieve higher coverage percentages than SLDV and FCT across all models and metrics, establishing the performance of the state-aware solving method. On average, GATES achieved 89.0% higher Decision Coverage, 93.5% higher Condition Coverage, and 330.3% higher MCDC than SLDV. Compared to FCT, GATES outperforms it by 91.0%, 84.8%, and 430.5% across all coverage metrics. When comparing GATES to HSTCG, the performance on Decision and Condition Coverage metrics is comparable, with GATES achieving equal or incrementally higher results. The primary performance difference between the two tools is observed in the MCDC metric, where GATES achieves higher results on a majority of the models. For instance, GATES increased the MCDC over HSTCG on the AFC model from 56% to 78% and on the TCP model from 50% to 75%. Both models contain deep branches whose execution is conditioned on specific state variables. Satisfying MCDC objectives within such logic requires not just reaching the branch, but also demonstrating that each condition can independently alter the decision's outcome. HSTCG, which focuses on branch coverage, struggles to cover the diverse input variations required for all MCDC conditions. In contrast, GATES's guided execution engine can construct the multi-step sequences required to satisfy these intricate MCDC objectives by generating numerous long-sequence test cases through the recombination of successfully solved inputs.

D. Efficiency of GATES

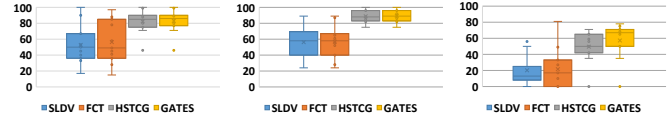
Although peak coverage is an important metric for evaluating a tool's capabilities, how efficiently this coverage is

TABLE III
COMPARISON OF PEAK COVERAGE AND AUC SCORE

Model	Tool	Decision Coverage	Condition Coverage	MCDC	AUC
AFC	SLDV	67%	89%	22%	10.0
	FCT	78%	65%	0%	15.6
	HSTCG	89%	100%	56%	16.6
	GATES	89%	100%	78%	17.8
FMTM	SLDV	50%	58%	13%	357.0
	FCT	28%	40%	0%	224.0
	HSTCG	85%	96%	59%	574.2
	GATES	86%	96%	71%	621.4
LANSwitch	SLDV	53%	63%	13%	70.5
	FCT	58%	61%	18%	86.6
	HSTCG	88%	97%	42%	236.6
	GATES	90%	97%	50%	254.8
LEDLC	SLDV	36%	26%	18%	50.4
	FCT	36%	28%	27%	72.0
	HSTCG	75%	85%	50%	131.9
	GATES	77%	92%	68%	144.4
NICProtocol	SLDV	17%	24%	5%	3.6
	FCT	15%	24%	10%	37.5
	HSTCG	80%	88%	35%	185.6
	GATES	80%	88%	35%	191.8
RTVibration	SLDV	50%	64%	25%	31.0
	FCT	88%	52%	2%	61.6
	HSTCG	90%	82%	50%	62.8
	GATES	90%	82%	50%	62.9
SDVProve	SLDV	33%	27%	12%	58.9
	FCT	49%	87%	49%	49.0
	HSTCG	71%	86%	65%	74.6
	GATES	71%	86%	65%	80.9
TCP	SLDV	40%	53%	8%	250.8
	FCT	41%	55%	17%	328.0
	HSTCG	84%	96%	50%	564.0
	GATES	85%	96%	75%	648.2
TCS	SLDV	90%	82%	56%	335.6
	FCT	97%	89%	81%	388.0
	HSTCG	99%	89%	71%	320.2
	GATES	99%	89%	71%	391.8
ATS	SLDV	100%	75%	50%	8.9
	FCT	84%	67%	33%	42.0
	HSTCG	100%	83%	67%	15.8
	GATES	100%	83%	67%	22.8
DCM	SLDV	45%	58%	0%	72.5
	FCT	45%	58%	0%	90.0
	HSTCG	46%	75%	0%	90.5
	GATES	46%	75%	0%	92.0
Average Improvement	vs SLDV	89.0%	93.5%	330.3%*	575.1%
	vs FCT	91.0%	84.8%	430.5%*	92.7%
	vs HSTCG	0.7%	0.7%	16.5%*	11.6%

* Our comparison with HSTCG focuses primarily on efficiency improvements (see Section IV-D for details). Additionally, we discovered new coverage across multiple models.

* The average MCDC improvement excludes models where the baseline coverage is 0% to prevent mathematical division by zero (e.g., AFC, FMTM, and DCM).



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

Fig. 7. Boxplot comparison of coverage results produced by GATES, SLDV and HSTCG across ten models. Subfigures (a), (b) and (c) show Decision Coverage, Condition Coverage, and MCDC, respectively. Colors denote SLDV (blue), HSTCG (orange) and GATES (green).

cost of regression testing. To evaluate this, we now focus on the rate of coverage gain. For this analysis, we use Decision Coverage as the representative metric, as it clearly illustrates the progress of each tool in exploring the model's fundamental control flow. During each of the 10 experimental runs for each tool-model pair, we recorded both the time taken to generate each test case and the cumulative Decision Coverage achieved at that point. Since the coverage data did not change for each run, we averaged the time-series data points to create a curve showing the coverage over time for each tool on each model.

The results of this efficiency evaluation are presented in the line charts in Figure 8. The figure consists of 11 subplots, each corresponding to a benchmark model. In each subgraph, the X-axis represents execution time (s) and the Y-axis represents the percentage of Decision Coverage (%) achieved. Each turning point in the graph represents a test case. Each plotted line represents the performance of one of the four tools—GATES, SLDV, FCT, and HSTCG. Steeper curves indicate a more efficient test case generation process, achieving higher coverage in a shorter time. The results show that GATES is more efficient in test case generation than the other tools.

To quantitatively capture this dynamic rate of coverage acceleration, we calculated the Area Under the Curve (AUC) for each tool's performance over the designated time budget. The AUC score quantifies the rate of coverage growth; a higher score indicates that a tool achieves higher coverage more rapidly. To establish a fair and standardized comparison across different tools for a given model, we defined a uniform total evaluation time, T_{total} . For each model, T_{total} is set to the time required by the slowest tool to reach its respective peak Decision Coverage. We utilize the trapezoidal rule to numerically integrate the discrete time-coverage data points. Let $(t_0, c_0), (t_1, c_1), \dots, (t_n, c_n)$ be the sequence of recorded data points where t_i is the execution time, c_i is the cumulative Decision Coverage achieved at t_i , $t_0 = 0$, $c_0 = 0$, and $t_n = T_{total}$. If a tool reaches its peak coverage before T_{total} , its coverage is assumed to remain constant until T_{total} . The normalized AUC score (expressed as a percentage) is calculated using the following formula: $AUC = \frac{1}{T_{total} \times 100} \sum_{i=1}^n \frac{(c_{i-1} + c_i)}{2} \times (t_i - t_{i-1})$. As reported previously in Table III, GATES outperforms SLDV, FCT, and HSTCG by 629%, 101.8%, and 12.6%, respectively. GATES achieved the highest AUC scores across the benchmark suite except ATS. Our analysis indicates that this is due to the relatively shallow branch depth in the ATS model and the prevalence of broad Boolean comparisons. Search-based FCT methods perform exceptionally well on this model, rapidly reaching

achieved is equally important. A more efficient tool provides faster feedback to engineers and reduces the computational

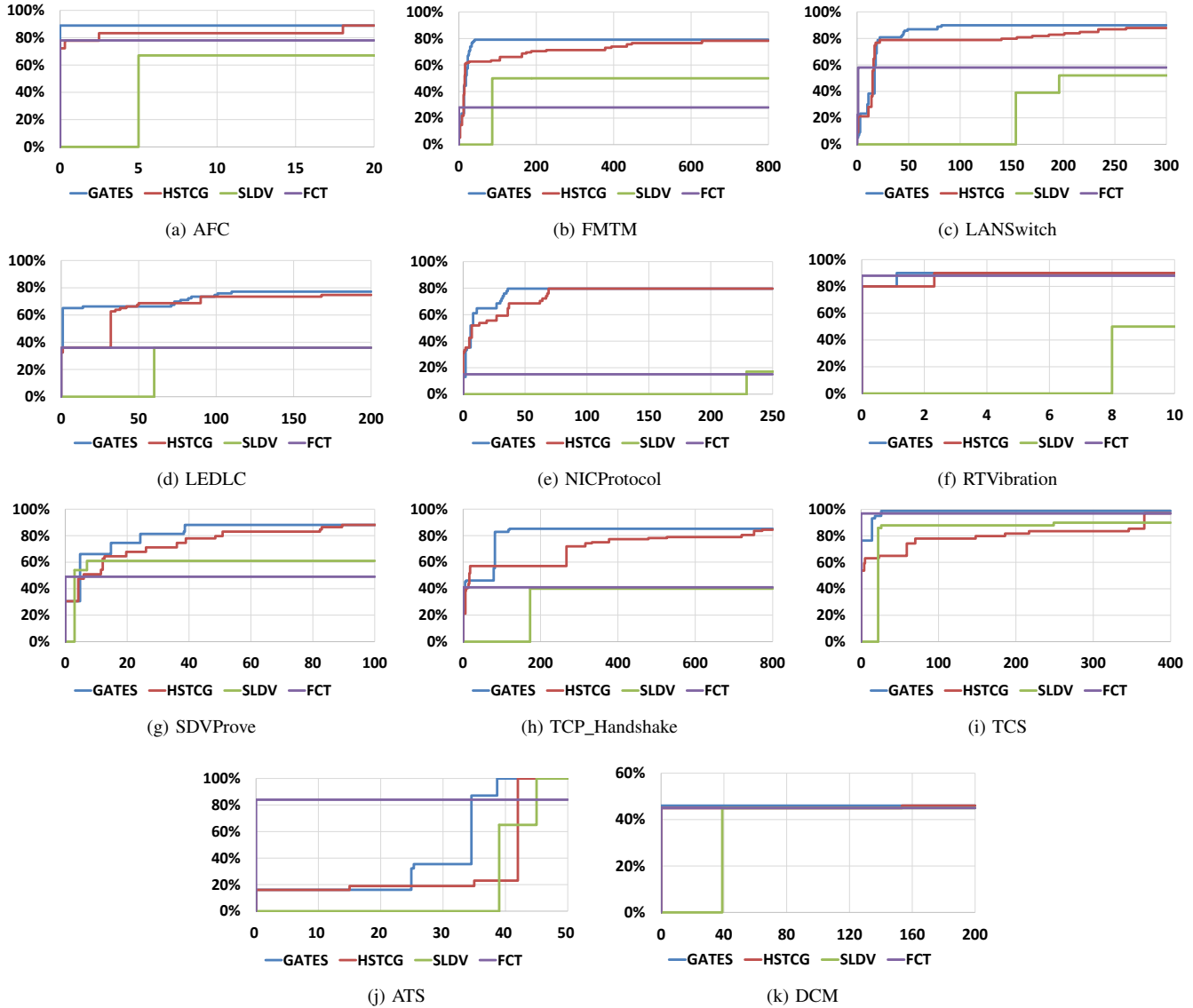


Fig. 8. The folded line plot of the Decision Coverage versus time. The X-axis is time (s), and the Y-axis is Decision Coverage (%).

peak coverage.

Due to the asynchronous nature of this process, a scenario may occur where the main execution thread selects an unreachable branch before the background filter has completely analyzed it. In our benchmark evaluation, this overlap occurred infrequently. Specifically, it occurred zero times in the AFC, LANSwitch, NICProtocol, SDVProve, TCS, and ATS models. It occurred once in FMTM and RTVibration, three times in TCP, and four times in LEDLC. The higher occurrences in LEDLC and TCP arise because their unreachable branches are concentrated in deeper logic levels, which delays the filter's detection time while the DPSTQ explores unattempted branches. In such cases, the main thread invokes the solver. Because the branch is unreachable, the solver eventually reaches the predefined timeout limit for a single invocation and returns a failure. However, this computational overhead is bounded. Upon failure, the dynamic priority solving task queue demotes the priority of this specific state and branch

task, preventing the main thread from repeatedly attempting it. This fallback mechanism protects the overall solving efficiency from degradation, even if the filter's analysis is delayed relative to the main thread.

To accurately measure the efficiency of test case generation, we compared the time it took for GATES, SLDV, and HSTCG, respectively, to reach the same coverage, using the peak coverage achieved by SLDV and HSTCG as the benchmark. It is important to note that while the FCT method was included in our peak coverage evaluation, we excluded it from the time-to-target efficiency comparison. Because FCT utilizes a stochastic search algorithm, it typically hits its peak structural coverage almost instantaneously before stalling on deep, discrete state-dependent logic. Consequently, a direct execution time comparison between such a randomized search heuristic and systematic, rigorous formal solving methods (GATES, SLDV, HSTCG) does not provide a meaningful metric for constraint-solving efficiency. Therefore, our efficiency analysis compares

the performance within the solving-based paradigm. As noted in our setup, the times reported in Table IV represent the fastest execution time recorded across the 10 independent runs for each tool. The results are shown in Table IV. Across all benchmarks, GATES demonstrates a substantial reduction in the time required to reach the target coverage against SLDV, achieving it 84.7% faster on average. The comparison with the academic tool, HSTCG, also shows a consistent and significant efficiency improvement. On average, GATES reached the target coverage 70.0% faster than HSTCG. To validate the efficiency improvements, we performed a Wilcoxon rank-sum test on the time-to-target coverage data gathered from the 10 independent runs. The resulting p -values are reported alongside the efficiency metrics in Table IV. As the data shows, GATES achieves a statistically significant reduction in generation time ($p < 0.05$) across all models compared to both SLDV and HSTCG.

This performance advantage is particularly pronounced on the AFC model. An analysis of this model reveals that its structure contains branches with time-cumulative logic. Covering these branches is a prerequisite for unlocking critical “gateway” states, which in turn enable the solving of numerous other dependent branches. Such cumulative conditions are a known challenge for formal solvers, which are not well-suited to discovering the specific, repetitive input sequences required. For instance, a key branch in AFC is only triggered if the fuel input signal remains within a specific threshold for n consecutive simulation steps. An examination of the test cases generated by GATES confirms this mechanism in action: they contain long, repetitive sequences of the same fuel input value. This is a direct result of the guided execution engine, which can immediately duplicate a successful input to satisfy such temporal conditions, thereby avoiding wasted solver resources. This highlights a key architectural difference between the two tools. GATES’s guided execution engine performs an opportunistic exploration step after every successful solve, allowing it to immediately leverage a successful input to clear these state-dependent roadblocks early in the process. In contrast, HSTCG’s strategy defers its random exploration phase until after attempting to solve from all currently available states. This leads to HSTCG expending numerous, unsuccessful solver invocations on branches that are not yet reachable, while GATES has already unlocked the necessary prerequisites.

Fault-Revealing Capability: While the efficiency and peak coverage metrics demonstrate GATES’s exploratory power, the practical value of a test suite ultimately lies in its ability to reveal faults. To evaluate this, we conducted a fault-seeding experiment. We injected a subtle, state-dependent logic fault into the FMTM and LANSwitch models. The fault was designed to incorrectly latch a previous iteration’s value by combining a Unit Delay block and an OR block into a closed loop without a reset mechanism. Once triggered by a ‘1’, this flawed structure perpetually outputs ‘1’. To simulate a deep, hard-to-reach bug, this faulty branch was placed downstream of a time-cumulative logic condition.

When executing the test suites generated by the three tools, both SLDV and HSTCG failed to trigger the fault. Their underlying constraint solvers struggled to systematically

construct the prolonged, specific input sequences necessary to pass the prerequisite time-cumulative logic. Consequently, the fault remained undetected in their test suites. In contrast, GATES successfully covered the faulty branch and revealed the unresettable latch behavior in both models.

Furthermore, during our evaluation, the test suite generated by GATES successfully detected a naturally occurring fault in the AFC benchmark model that shares a similar time-dependent latching mechanism.

E. Effectiveness of Dynamic Priority Solving Task Queue

For any solver-based test generation approach, the order in which solving tasks are attempted can significantly influence overall efficiency. A change in the search path may lead to a more optimal solution, but it could also explore less productive regions of the state space. Therefore, to evaluate the effect of our heuristic-based scheduling, a dedicated experiment is necessary to determine whether it consistently finds a more efficient solution path than a simpler, non-prioritized strategy.

We compared two configurations of our tool: GATES (with DPSTQ): The full system, using the dynamic priority queue to schedule solving tasks. GATES (without DPSTQ): An ablated version where a standard, non-prioritized first-in, first-out queue replaces the DPSTQ. All other strategies, including the unreachable branch filter and randomized scheduler, remain active in this configuration to maintain a consistent baseline comparison. Since both configurations ultimately achieve the same decision coverage, we recorded the total number of solver calls and the time-to-coverage to evaluate the DPSTQ’s impact on computational effort and overall efficiency. Table V presents the results of this study, comparing the solver invocations required by the DPSTQ-driven version against the first-in, first-out version. The data shows that the configuration with DPSTQ reduced the required solver calls by 36.7% and decreased the time-to-coverage by 40.1%. This improvement correlates with the complexity of the model’s state space. For instance, on models with strong state dependencies like RTVibration and TCS, where a simple search is likely to explore many unproductive paths, the DPSTQ’s scheduling achieved solver call reductions of 56.2% and 82.6%, respectively. Even on models like TWC, where the optimal solution path may be more direct, the DPSTQ still provides a benefit by avoiding unnecessary steps. Furthermore, the ablated GATES configuration without the DPSTQ still outperforms both HSTCG and SLDV in terms of efficiency.

F. Concurrent Hybrid Testing with CFTCG

Modern testing strategies often seek to combine the strengths of different techniques. Coverage-guided fuzzers, like CFTCG, are effective at rapidly exploring a model’s shallow state space and covering branches with simple trigger conditions [41]. However, they often struggle to generate the specific, complex input sequences required to satisfy deep, state-dependent logic. Given that GATES incorporates dynamic methods such as the randomized scheduler, we designed an experiment to evaluate its integration with CFTCG. We

TABLE IV
EFFICIENCY COMPARISON OF GATES AGAINST SLDV AND HSTCG.

Model	GATES vs SLDV (s)				GATES vs HSTCG(s)			
	SLDV	GATES	Reduction	<i>p</i> -value	HSTCG	GATES	Reduction	<i>p</i> -value
AFC	5	0.2	96.0%	< 0.001	18	0.2	98.9%	< 0.001
FMTM	188	18	90.4%	< 0.001	644	41	93.6%	< 0.001
LANSwitch	196	17	91.3%	< 0.001	272	78	72.1%	< 0.001
LEDLC	60	1	98.3%	< 0.001	168	101	39.9%	< 0.001
NICProtocol	229	2	99.1%	< 0.001	95	36	62.1%	< 0.001
RTVibration	8	0.2	97.5%	< 0.001	2.3	1.1	52.1%	0.003
SDVProve	7	4	42.9%	0.002	89	38	57.3%	< 0.001
TCP	173	1	99.4%	< 0.001	366	26	92.9%	< 0.001
TCS	249	14	94.4%	< 0.001	790	120	84.8%	< 0.001
ATS	45	35	22.2%	0.005	42	35	16.7%	0.008
DCM	39	0.1	99.7%	< 0.001	154	0.1	99.9%	< 0.001
Average	-	-	84.7%	-	-	-	70.0%	-

* The 'Reduction' quantifies the percentage decrease in time required by GATES to reach the same peak coverage as the baseline tool.

* The *p*-values are calculated using the Wilcoxon rank-sum test across 10 independent runs. A *p*-value < 0.05 indicates statistical significance.

TABLE V
TOTAL SOLVER CALLS AND TIME REDUCTION REQUIRED TO REACH TARGET COVERAGE, WITH AND WITHOUT THE DYNAMIC PRIORITY SOLVING TASK QUEUE

Model	without DPSTQ	with DPSTQ	Reduction	Time Reduction Rate (Before → After)
AFC	6	5	20.0%	20.0% (5s → 4s)
FMTM	70	39	44.3%	39.2% (79s → 48s)
LANSwitch	154	85	44.8%	40.3% (139s → 83s)
LEDLC	119	105	11.8%	16.4% (134s → 112s)
NICProtocol	63	33	47.6%	44.3% (70s → 39s)
RTVibration	32	14	56.2%	69.8% (63s → 19s)
SDvProve	155	125	19.4%	21.5% (93s → 73s)
TCP	277	248	10.5%	29.9% (421s → 295s)
TCS	195	34	82.6%	81.4% (237s → 44s)
ATS	88	77	12.5%	22.2% (45s → 35s)
DCM	39	18	53.8%	56.1% (98s → 43s)
Average	-	-	36.7%	40.1%

aimed to establish whether GATES's covering capabilities could augment a fuzzer's broad search to improve the coverage and efficiency of a purely dynamic method. This experiment, therefore, assesses the performance of a concurrent configuration in which both tools run simultaneously.

We designed an experiment comparing two distinct testing strategies, both operating under a fixed total time budget of 1 hour. CFTCG(Baseline): The CFTCG tool was run on each benchmark for the full 1-hour duration. Concurrent(CFTCG + GATES): In this configuration, both CFTCG and GATES were run simultaneously for the entire 1-hour period. The tools were configured to collaborate: GATES continuously tracked the test cases and coverage output produced by CFTCG. As CFTCG outputs test cases, GATES executes them to new model states, using these new starting points to generate tasks for its adaptive solving engine. This allows GATES to use the fuzzer's exploration in real time to guide its solving efforts. We compare these two strategies on two primary metrics: the

final Decision Coverage achieved at the end of the hour, and the time required to reach the same coverage (if different, the lower), which evaluates overall efficiency.

The results presented in Table VI demonstrate the performance of the concurrent hybrid system. The data shows that the CFTCG + GATES configuration achieves higher peak decision coverage than CFTCG alone across all models. This is particularly evident in models such as NICProtocol and TCP, in which coverage increased by 233.3% and 50.0%, respectively. Furthermore, the data shows a significant improvement in efficiency. On average, the concurrent hybrid system reached the target coverage 60.5% faster than CFTCG alone. This efficiency gain is a result of the parallel collaboration between the two tools. CFTCG acts as a state-discovery engine, continuously providing GATES with new starting points in real time. This enables GATES to allocate its resource-intensive solver to the most relevant, newly discovered regions of the state space, resulting in much faster convergence.

The near-instantaneous time-to-target for configurations on the TCS, RTVibration, and ATS models warrants analysis. Upon inspection, these models contain branches whose feasibility is tightly coupled to complex calculations or state transitions. CFTCG's exploration of such branches exhibits a binary characteristic. It either finds a path almost immediately through its initial mutations or fails to make further progress. This behavior rapidly populates a set of states that are either identical or have very high functional similarity. For GATES's state-aware engine, this homogenous state corpus offers limited novel starting points from which it can apply its deep-solving capabilities. This result does not indicate a weakness, but rather highlights a key point of concurrent testing that the effectiveness of the solver depends on the quality and diversity of the state space being explored by its fuzzer counterpart. This provides valuable insight for creating tighter feedback loops in future work.

TABLE VI
PERFORMANCE COMPARISON OF STANDALONE CFTCG VS. CONCURRENT (CFTCG + GATES).

Model	Peak Decision Coverage (%)			Time to Target Coverage (s)		
	CFTCG	Concurrent	Improvement	CFTCG	Concurrent	Reduction(%)
AFC	83	89	7.2%	5	4	20.0%
FMTM	75	77	2.7%	644	21	96.7%
LANSwitch	86	90	4.7%	1774	42	97.6%
LEDLC	72	77	6.9%	1986	118	94.1%
NICProtocol	24	80	233.3%	488	2	99.6%
RTVibration	90	90	0%	0.1	0.1	0%
SDVProve	88	88	0%	63	1	98.4%
TCP	58	87	50.0%	2674	40	98.5%
TCS	95	97	2.1%	0.1	0.1	0%
ATS	100	100	0%	0.1	0.1	0%
DCM	46	46	0%	0.1	0.1	0%
Average	-	-	30.7%	-	-	60.5%

* The experiment is over 1 hour. The 'Time to Target Coverage' measures the time required for each method to achieve the peak coverage reached by the CFTCG baseline.

* No efficiency gains were achieved on the RTVibration, TCS, ATS, and DCM model because both configurations reached the target coverage instantaneously.

V. DISCUSSION

The experimental results demonstrate that GATES's design of adaptive solving and guided execution offers significant advantages in test case generation efficiency and effectiveness. This section discusses the implications of our design choices and addresses potential threats to validity.

A. Handling of Time-Dependent Constraints

A persistent challenge in model-based testing is the handling of branches with explicit time-dependent constraints, where a branch's feasibility depends on input values that must correlate with the simulation time (e.g., $\text{input}(t) > f(t)$). Conventional solver-based and dynamic methods often struggle to systematically generate the continuously evolving input signals required to satisfy such logic. The architecture of GATES, however, provides a promising foundation for addressing this challenge. While the current Guided Execution engine composes sequences from a static pool of solved inputs, its ability to analyze and construct sequences can be extended. A potential future direction is to enhance the engine with the ability to analyze temporal patterns within existing, successful test sequences. By applying learning-based techniques, such as linear fitting or integrating parameterized interpolation functions (e.g., linear and piecewise cubic splines), to these observed trajectories, the system could generate a predictive model of the required input behavior. By applying learning-based techniques, such as linear fitting, to these observed trajectories, the system could generate a predictive model of the required input behavior.

This model could then be used to synthesize entirely new, time-aware input signals specifically designed to satisfy these constraints, further broadening the applicability and effectiveness of the GATES approach.

B. Reproducibility and the Role of Randomness

Using a randomized scheduler may raise concerns about the reproducibility of test cases, a critical requirement for debugging and regression testing. It is important to note that

the execution in GATES is pseudo-random. This means the process is driven by a deterministic algorithm initialized with a fixed seed. Providing the same seed to the system will therefore produce the same sequence of "random" decisions, ensuring that any generated test suite can be reproduced precisely. Thus, our approach treats randomness as a tool for opportunistic exploration rather than as a source of non-determinism. The scheduler accomplishes this by combining existing, successfully solved inputs into new, longer sequences. This enables the system to discover novel state interactions and bypass complex local constraints that would be computationally expensive for a formal solver to find systematically. Thus, this controlled use of randomness enhances the system's exploratory capability without sacrificing the deterministic rigor required for industrial applications.

C. Extensibility and Integration with Other Testing Methods

GATES's modular design allows its core strategies to be integrated with other test case generation approaches. For example, the dynamic priority solving task queue and the randomized scheduler are not tied to our solver implementation. These modules could be adapted to improve the performance of other solving-based engines. The priority queue could replace a first-in, first-out queue to provide more intelligent scheduling, directing the solver's effort toward promising states. Similarly, the randomized scheduler could be added to a solving-based tool to introduce a new mechanism using solved inputs for broader exploration. Furthermore, our experiment with CFTCG provides a blueprint for combining GATES with other dynamic testing methods. The concurrent configuration demonstrates a collaborative pattern in which the dynamic tool functions as a high-speed engine for broad state-space discovery. Meanwhile, GATES acts as a deep analysis engine, using the states found by the dynamic tool to solve complex branches that typically act as barriers. This illustrates a generalizable strategy that could augment other coverage-guided fuzzers or random testing tools by combining their strengths for more comprehensive results.

VI. CONCLUSION

In this paper, we addressed the key efficiency challenges in solving-based test case generation for complex, stateful Simulink models. We introduced GATES, an approach that first employs an adaptive solving engine to dynamically allocate computational resources, with an unreachable branch filter and a dynamic priority solving task queue, this engine schedules tasks to avoid unnecessary solving. And then the guided execution engine leverages all successfully solved inputs to generate new test sequences. This process accelerates the coverage of simple branches and enhances the exploration of challenging, time-cumulative logic. Our evaluation on a suite of industrial benchmarks demonstrates significant performance gains. Compared to the commercial tool SLDV, GATES achieved average improvements of 89.0%, 93.5%, and 330.3% in Decision Coverage, Condition Coverage, and MCDC, respectively. Against the academic tool, HSTCG, GATES not only achieved higher MCDC, but also reduced the average time to reach the same coverage by 70.0%. Furthermore, our experiments showed that GATES can be effectively integrated with dynamic methods like the CFTCG, accelerating its time to generate test cases by 60.5%. For future work, we plan to investigate methods for handling time-dependent constraints. We hypothesize that by employing linear fitting or learning-based algorithms on sequences that cover branches related to time, we can calculate new input test sequences to cover time-dependent branches.

VII. DATA AVAILABILITY

The anonymized version for review is available at <https://anonymous.4open.science/r/GATES-85F5/>.

VIII. ACKNOWLEDGMENT

This research is sponsored in part by the NSFC Program under Grant 92582101 and Grant 62372263.

REFERENCES

- [1] E. Estevez and M. Marcos, "Model-based validation of industrial control systems," *IEEE Transactions on Industrial Informatics*, vol. 8, no. 2, pp. 302–310, 2012.
- [2] M. Bennion and I. Habli, "A candid industrial evaluation of formal software verification using model checking," in *Companion Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE Companion 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 175–184. [Online]. Available: <https://doi.org/10.1145/2591062.2591184>
- [3] E. A. Lee, "Cyber physical systems: Design challenges," in *2008 11th IEEE international symposium on object and component-oriented real-time distributed computing (ISORC)*. IEEE, 2008, pp. 363–369.
- [4] R. Alur, *Principles of cyber-physical systems*. MIT press, 2015.
- [5] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Commun. ACM*, vol. 56, no. 2, p. 82–90, Feb. 2013. [Online]. Available: <https://doi.org/10.1145/2408776.2408795>
- [6] C. Aschermann, S. Schumilo, A. Abbasi, and T. Holz, "Ijon: Exploring deep state spaces via fuzzing," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 1597–1612.
- [7] 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.
- [8] R. Matinnejad, S. Nejati, L. C. Briand, and T. Bruckmann, "Automated test suite generation for time-continuous simulink models," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 595–606. [Online]. Available: <https://doi.org/10.1145/2884781.2884797>
- [9] Z. Su, Z. Yu, D. Wang, W. Chang, B. Gu, and Y. Jiang, "Test case generation for simulink models using model fuzzing and state solving," in *2024 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2024, pp. 117–128.
- [10] 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.
- [11] E. Bartocci, L. Mariani, D. Nickovic, and D. Yadav, "Signal feature coverage and testing for cps dataflow models," *ACM Trans. Softw. Eng. Methodol.*, vol. 34, no. 7, Aug. 2025. [Online]. Available: <https://doi.org/10.1145/3714467>
- [12] Simulink and Matlab, "Simulink design verifier," <https://www.mathworks.com/help/sldv/index.html>, 2008, accessed: 2025-01-29.
- [13] Z. Su, Z. Yu, D. Wang, Y. Yang, R. Wang, W. Chang, A. Cui, and Y. Jiang, "Hstcg: State-aware test case generation for simulink models," in *Proceedings of the 60th ACM/IEEE Design Automation Conference (DAC)*, 2023, pp. 1–6.
- [14] Z. Su, Z. Yu, D. Wang, Y. Yang, R. Wang, W. Chang, L. Sha, and Y. Jiang, "Hstcg: State-aware simulink model test case generation with heuristic strategy," *IEEE Transactions on Software Engineering*, vol. 50, no. 12, pp. 3088–3103, 2024.
- [15] 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.
- [16] Simulink and Matlab, *Simulink Documentation*, accessed: 2025-01-29. [Online]. Available: <https://www.mathworks.com/help/simulink/index.html>
- [17] J. Van Noten, K. Gadeyne, and M. Witters, "Model-based systems engineering of discrete production lines using sysml: An experience report," *Procedia CIRP*, vol. 60, pp. 157–162, 2017, complex Systems Engineering and Development Proceedings of the 27th CIRP Design Conference Cranfield University, UK 10th – 12th May 2017. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S2212827117300197>
- [18] E. Bartocci, N. Manjunath, L. Mariani, C. Mateis, and D. Ničković, "Cpsdebug: Automatic failure explanation in cps models," *International Journal on Software Tools for Technology Transfer*, pp. 1–14, 2021.
- [19] N. Tracey, J. Clark, K. Mander, and J. McDermid, "An automated framework for structural test-data generation," in *Proceedings 13th IEEE International Conference on Automated Software Engineering (Cat. No.98EX239)*, 1998, pp. 285–288.
- [20] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2008, pp. 209–224.
- [21] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Chicago, IL: ACM, 2005, pp. 213–223.
- [22] Google, "Libfuzzer tutorial," <https://github.com/google/fuzzing/blob/master/tutorial/libFuzzerTutorial.md>, 2016, accessed: 2025-01-29.
- [23] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "{AFL++}: Combining incremental steps of fuzzing research," in *14th USENIX workshop on offensive technologies (WOOT 20)*, 2020.
- [24] P. McMinn, "Search-based software test data generation: a survey," *Software testing, verification and reliability*, vol. 14, no. 2, pp. 105–156, 2004, cited as a foundational survey for search-based software testing (SBST).
- [25] A. Panichella, "Beyond unit-testing in search-based test case generation: Challenges and opportunities," in *2019 IEEE/ACM 12th International Workshop on Search-Based Software Testing (SBST)*, 2019, pp. 7–8.
- [26] Z. Su, D. Wang, Y. Yang, Y. Jiang, W. Chang, L. Fang, W. Li, and J. Sun, "Code synthesis for dataflow based embedded software design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021.
- [27] G. Ernst, P. Arcaini, G. Fainekos, F. Formica, J. Inoue, T. Khandait, M. Mahboob, C. Menghi, G. Pedrielli, M. Waga, Y. Yamagata, and Z. Zhang, "Arch-comp 2022 category report: Falsification with unbounded resources," *EPiC Series in Computing*, vol. 90, pp. 204–221,

- 2022, publisher Copyright: © 2022, EasyChair. All rights reserved.; 9th International Workshop on Applied Verification of Continuous and Hybrid Systems, ARCH 2022 ; Conference date: 05-09-2022 Through 05-09-2022.
- [28] A. Donzé, "Breach, a toolbox for verification and parameter synthesis of hybrid systems," *Lecture Notes in Computer Science*, pp. 167–170, 2010.
- [29] Y. Annpureddy, C. Liu, G. Fainekos, and S. Sankaranarayanan, "S-taliro: A tool for temporal logic falsification for hybrid systems," in *Tools and Algorithms for the Construction and Analysis of Systems*, P. A. Abdulla and K. R. M. Leino, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 254–257.
- [30] G. Ernst, S. Sedwards, Z. Zhang, and I. Hasuo, "Fast falsification of hybrid systems using probabilistically adaptive input," in *Quantitative Evaluation of Systems: 16th International Conference, QEST 2019, Glasgow, UK, September 10–12, 2019, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2019, p. 165–181. [Online]. Available: https://doi.org/10.1007/978-3-030-30281-8_10
- [31] A. Dokhanchi, A. Zutshi, R. T. Sriniva, S. Sankaranarayanan, and G. Fainekos, "Requirements driven falsification with coverage metrics," in *Proceedings of the 12th International Conference on Embedded Software*, ser. EMSOFT '15. IEEE Press, 2015, p. 31–40.
- [32] E. Bartocci, R. Bloem, B. Maderbacher, N. Manjunath, and D. Ničković, "Adaptive testing for specification coverage and refinement in cps models," *Nonlinear Analysis: Hybrid Systems*, vol. 46, p. 101254, 2022. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S1751570X22000619>
- [33] E. P. Enou, A. Čaušević, T. J. Ostrand, E. J. Weyuker, D. Sundmark, and P. Pettersson, "Automated test generation using model checking: an industrial evaluation," *Int. J. Softw. Tools Technol. Transf.*, vol. 18, no. 3, p. 335–353, Jun. 2016. [Online]. Available: <https://doi.org/10.1007/s10009-014-0355-9>
- [34] L. Zou, N. Zhan, S. Wang, and M. Fränzle, "Formal verification of simulink/stateflow diagrams," in *International Symposium on Automated Technology for Verification and Analysis*. Springer, 2015, pp. 464–481.
- [35] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, K. Jensen and A. Podelski, Eds., vol. 2988. Springer, 2004, pp. 168–176.
- [36] S. L. Shrestha, S. A. Chowdhury, and C. Csallner, "Slnet: a redistributable corpus of 3rd-party simulink models," in *Proceedings of the 19th International Conference on Mining Software Repositories*, ser. MSR '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 237–241. [Online]. Available: <https://doi.org/10.1145/3524842.3528001>
- [37] 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: 4th International Conference, TMPA 2017, Moscow, Russia, March 3–4, 2017, Revised Selected Papers 4*. Springer, 2018, pp. 77–89.
- [38] 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.
- [39] R. Matinnejad, S. Nejati, L. C. Briand, and T. Bruckmann, "Test generation and test prioritization for simulink models with dynamic behavior," *IEEE Transactions on Software Engineering*, vol. 45, no. 9, pp. 919–944, 2018.
- [40] C. D. Nguyen, A. Marchetto, and P. Tonella, "Combining model-based and combinatorial testing for effective test case generation," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, 2012, pp. 100–110.
- [41] Z. Su, Z. Yu, D. Wang, R. Wang, Y. Tao, and Y. Jiang, "CFTCG: test case generation for simulink model through code based fuzzing," in *Proceedings of the 61st ACM/IEEE Design Automation Conference, DAC 2024, San Francisco, CA, USA, June 23–27, 2024*, V. De, Ed. ACM, 2024, pp. 16:1–16:6.