PAPER *Special Section on Mathematical Systems Science and its Applications*

Template-Based Monte-Carlo Test-Suite Generation for Large and Complex Simulink Models*

Takashi TOMITA[†a)], ***Nonmember***, **Daisuke ISHII**[††], ***Member***, **Toru MURAKAMI**[†††], **Shigeki TAKEUCHI**[†††], ***Nonmembers, and*** **Toshiaki AOKI**[†], ***Member***

SUMMARY MATLAB/Simulink is the de facto standard tool for the model-based development (MBD) of control software for automotive systems. A Simulink model developed in MBD for real automotive systems involves complex computation as well as tens of thousands of blocks. In this paper, we focus on decision coverage (DC), condition coverage (CC) and modified condition/decision coverage (MC/DC) criteria, and propose a Monte-Carlo test suite generation method for large and complex Simulink models. In the method, a candidate test case is generated by assigning random values to the parameters of signal templates with specific waveforms. We try to find contributable candidates in a plausible and understandable search space, specified by a set of templates. We implemented the method as a tool, and our experimental evaluation showed that the tool was able to generate test suites for industrial implementation models with higher coverages and shorter execution times than Simulink Design Verifier. Additionally, the tool includes a fast coverage measurement engine, which demonstrated better performance than Simulink Coverage in our experiments.

key words: *model-based development, MATLAB/Simulink, test case generation, Monte-Carlo method, template-based method*

1. Introduction

1.1 Background

Along with the evolution of electronic control technology, various modules have gradually electronized in the automotive, aircraft, and aerospace industries. How to ensure the safety and reliability of *electronic control units* (ECUs) for such modules has been a difficult challenge. IEC 61508 [3], a basic functional safety standard for electronic systems in general industry, and ISO 26262 [4], the specialized standard for the automotive industry, were published as international standards. Over the past decade, many companies have focused on developing hybrid vehicles, electric vehicles, driving assistance systems, and automated driving systems. Hence, effective and efficient methods and techniques are strongly required for developing safe and reliable software for larger and more complex ECUs.

Model-based development (MBD) is widely employed in the development of safe and reliable software for ECUs. Usually, such software (i.e., *controller*) controls a target system (i.e., *plant*) using feedback loops. In MBD, both the controller and plant are *simulatable*, and their behaviors can be checked by *numerical simulation*. A specification model for a controller is initially defined as, e.g., a system of differential equations, and then refined and discretized into an implementation model as a fixed-point and discrete-time system with concrete data definitions and additional features, e.g., overflow measures. This model refinement can be validated and verified because the models are mathematical and simulatable. Finally, such implementation models can be converted into code automatically. The quality of software is guaranteed by traceable model refinement and automatic code generation. *MATLAB*[**]*/Simulink*[***] is widely used as a de facto standard MBD tool, which provides an environment for numerical operation, graphical modeling, simulation, and code generation.

Simulink models contain complex computations. Furthermore, the scale of these models becomes quite large for practical developments. We believe that *testing* has the potential to deal with such models and enables us to ensure the quality of the models. In fact, IEC 61508 and ISO 26262 recommend the use of some coverage criteria for testing. In model testing, a *test case* is a group of input signals (and expected output signals), and a *test suite* is a set of test cases. Model refinement in MBD is justified by back-to-back conformance testing with numerical simulation. Also white-box testing based on *decision coverage* (DC), *condition coverage* (CC), and *modified condition/decision coverage* (MC/DC) criteria is widely employed. Typically, a release version of software must pass the tests of a full (or high, for some criteria) coverage test suite. MATLAB/Simulink has toolboxes to assist efficient testing, which conform to IEC 61508 and ISO 26262. *Simulink Design Verifier*[****] (SLDV) provides functions to detect design errors and generate test cases. *Simulink Coverage*[*****] (SLC) provides coverage measurement functions for various criteria including DC, CC and MC/DC.

Manuscript received April 14, 2019.
Manuscript revised August 30, 2019.

[†]The authors are with Japan Advanced Institute of Science and Technology, Nomi-shi, 923-1292, Japan.
[††]The author is with University of Fukui, Fukui-shi, 910-8507 Japan.
[†††]The authors are with Gaio Technology Co., Tokyo, 140-0002 Japan.

[*]This paper is a merged and revised version of parts of [1] and [2].

a) E-mail: tomita@jaist.ac.jp
DOI: 10.1587/transfun.2019MAP0010

[**]https://www.mathworks.com/products/matlab.html
[***]https://www.mathworks.com/products/simulink.html
[****]https://www.mathworks.com/products/sldesignverifier.html
[*****]https://www.mathworks.com/products/simulink-coverage.html

1.2 Challenges

There are several problems in the testing process of MBD for practical ECU software. First, it is difficult to obtain a full (or even high) coverage test suite for a practical model, which is large and complex. In some cases, it has tens of thousands of operations, which are called blocks in MATLAB/Simulink. Such blocks include (possibly non-linear) mathematical operation blocks, logical operation blocks, switch blocks, delay blocks and (conditional) subsystem blocks. SLDV provides a test case generator, but it cannot fully deal with practical models because SLDV mainly relies on formal analysis, wherein non-linear arithmetic leads to undecidability. Additionally, each test case is a set of time-series data. A large model has complex internal states as well as more than a hundred of inputs. A number of time-steps are sometimes required to observe a series of transitions of such states. So the search space for test cases becomes huge. Formal analysis does not generally scale even in the case that is theoretically decidable. Hence, we need a feasible test generation method. The *Monte-Carlo method* is often used for this purpose. This approach tries to find suitable test cases from many randomly generated candidates. However, we may need a useful guide for finding contributable test cases because of the massive search space we handle.

Second, the coverage measurement function `cvsim` provided by SLC is terribly slow when compared to the naive simulation function `sim` provided by the standard Simulink. In the Monte-Carlo approach, this is critical because we need to generate many candidates and collate their coverage statistics. Even if we build a good test suite, we need to spend an unacceptable amount of time for checking whether the practical model passes the test suite. Coverage measurement consists of the following parts: numerical simulation with signal data logging of a given test case, and identification of time-steps that meet the coverage objectives. It is necessary to accelerate this coverage measurement procedure to improve the efficiency of the overall testing process.

1.3 Contributions

The contributions of this paper are twofold. First, we propose a *template-based Monte-Carlo method* [1] to automatically generate a test suite that achieves high DC, CC and MC/DC in MATLAB/Simulink. Second, we realize the proposed method as a tool [2] with an efficient coverage measurement engine that is critical to its performance and scalability.

The template-based Monte-Carlo method tries to find comprehensible test cases. A candidate test case is generated by assigning random values to *templates* that provide parameterized signals with specific waveforms. Therefore, the candidates are chosen from a plausible and understandable search space specified by a set of templates.

In terms of coverage measurement, we independently implement an engine that exploits fast simulation modes and array operations, which are standard features of MATLAB/Simulink. This implementation is used to accelerate the signal-simulation/coverage-measurement loop, which is essential for Monte-Carlo optimization-based tools that deal with practical models.

Then, we carried out experiments to evaluate the effectiveness of the template-based method and the performance of the tool. We confirmed that our method can produce stimulating signals much more efficiently than naive random signal generation methods, and our tool is able to generate higher-coverage test suites in much less time than SLDV, for real implementation models developed by industry.

1.4 Organization of the Paper

The remainder of the paper is organized as follows: Section 2 introduces related work of the paper. Section 3 provides preliminary information about Simulink models and coverage testing. Section 4 proposes a template-based Monte-Carlo test-suite generation method. Section 5 describes the implementation of our method. Section 6 reports evaluation experiments. Finally, we conclude in Sect. 7.

2. Related Work

Simulink Design Verifier (SLDV) is a first-party MATLAB/Simulink toolbox that generates test cases mainly using a formal analysis. SLDV uses rational approximations of floating-point arithmetic and tries to identify suspicious test cases. Additionally, there are some third-party tools based on formal methods, e.g., *BTC*[†] *Embedded Tester* [5] and *AutoMOTGen* [6]. Embedded Tester converts a model into a C code and generates test cases based on formal analysis of the C code. AutoMOTGen converts a Simulink model into a formal model, and uses model checking techniques to check whether the formal model meets formally described test objectives. However, many practical models cannot be fully dealt with by them due to limitations related to computability and computational complexity. Additionally, a test case produced by them may consist of input signals with any complex waveforms. In contrast, our method generates test cases by template-based Monte-Carlo approach. Although it may fail to output full-coverage test suites even for simple models, our method produces only understandable test cases and deals with any simulatable model even if it has non-linear arithmetic.

Reactis[††] is a third-party tool that consists of simulator, tester, and verifier. The tester tries to find a test case by combining a Monte-Carlo method and a *guided simulation* [7]. Guided simulation generates a new candidate test case from a previous one by changing its suffix after a certain time-step, which is determined based on backward

[†]https://www.btc-es.de/
[††]http://www.reactive-systems.com/

data-flow analysis. *SmartTestGen* [8], [9] is another tool that integrates different techniques: random testing, constraint solving, model checking, and heuristics. An input signal included in a test case generated by them may have any waveform. In contrast, our method generates understandable candidates based on templates that specify the characteristics of the entire waveform of the input signal.

A *randomized directed testing* (REDIRECT) method [10] for state/transition coverage testing, a *model-checking based method* [11] for mutation coverage testing, and an *output-diversity based method* [12], [13] for fault revealing were proposed. These methods aim for a test case generation with criteria different from DC, CC and MC/DC.

Simulink Coverage (SLC) is a first-party toolbox for coverage testing. The details of its implementation are unknown, but the coverage measurement function `cvsim` does not support fast simulation modes and is much slower than the naive simulation function `sim`. In contrast, our tool was implemented from scratch using only standard functions in MATLAB/Simulink, and is much faster than SLC and also supports fast simulation modes.

3. Preliminaries

3.1 Simulink Models

MATLAB is a general-purpose numerical computing environment with an efficient handling of arrays, matrices, images, and signals. There are many first-party toolboxes, e.g., Simulink, SLDV and SLC.

Simulink is a MATLAB toolbox for modeling and simulation purpose. A *Simulink model* is a block diagram that specifies a dynamical system. A *block* is a processing unit in Simulink and varies in various types, e.g., *input/output*, *mathematical operator*, *logical/relational operator*, *(multiport) switch*, *delay*, and *(conditional) subsystem*. Blocks are connected with *lines*, which transfer *Boolean*, *integer*, or *floating/fixed-point* data at each time step. Delay blocks relay data after specified delay times, and thus they brings in internal states within a model. A subsystem represents a hierarchical module in a Simulink model, and is interfaced with their parent system via input/output ports that are implemented with input/output blocks internally. Furthermore, a subsystem is often equipped with control ports, e.g., *enable* and *trigger* ports, for selective activation/inactivation of the subsystem with external signals; also, a subsystem may mask their low-level structure, where the mask is configured with the mask variables. Usually, variables in MATLAB are stored in a base workspace. A Simulink model may refer variables in the base workspace.

The behavior of a Simulink model is characterized in terms of *signals* (i.e., time-series data). Inputs/outputs of blocks, subsystems and the model itself are represented as signals. In a simulation, an input signal is given to each input port of them and processed based on their description. Processed signals are propagated to their successors via lines connected with their output ports.

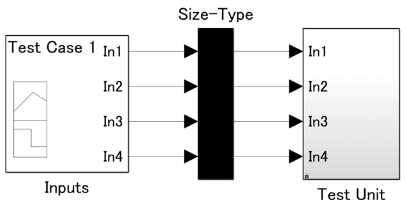


Fig. 1 Example of a harness model. A signal builder block "`Inputs`" provides test cases for "`Test Unit`" subsystem copied from a target model via an adapter subsystem "`Size-Type`."

Simulink has several simulation modes: *normal*, *accelerator*, and *rapid accelerator* (or shortly *rapid*). A normal mode simulator runs within the MATLAB/Simulink environment. An accelerator-mode simulator is generated as an execution engine in memory. A rapid-mode simulator is generated as a stand-alone execution file. Each mode requires to compile a Simulink model into an intermediate representation. For the normal and accelerator modes, we can omit recompilation by enabling the *fast restart* feature. The rapid mode requires generating an excutable file in every simulation. The accelerator and rapid modes require API communications between MATLAB/Simulink and the simulator processes.

3.2 Test Case/Suite

For the ease of testing a target model, we use a *harness model* (Fig. 1). A harness model consists of (i) a subsystem ("`Test Unit`") that contains a copy of the target model, (ii) a *signal builder block* ("`Inputs`") that provides a group of input signals for the target model, and (iii) a subsystem ("`Size-Type`") that configures data types and sampling rates of each input signals. Because a signal builder block generates only `double`-typed signals, this "`Size-Type`" subsystem casts each input into the signal of an appropriate type. SLDV and SLC have functions to construct a harness model of a model under testing.

A *test case* is a group of input signals, and a *test suite* is a set of test cases. In Simulink, a signal block (cf. "`Inputs`" in Fig. 1) can be considered as a test suite; we can add, change, or remove a test case in a signal builder block using the `signalbuilder` function.

3.3 Test Objectives and Coverage Criteria

The objective of a model test is to attain a high *model coverage*, which measures how exhaustively the objects in a model are exercised, given a test case. For a test suite, the coverage of every test case is accumulated. Various coverage criteria are proposed. In the DC, CC and MC/DC criteria, a measured object is a block whose activity affects the logical characteristics (i.e., data flow pattern) of the model; thus, the test objective is broken down to the objectives for those blocks.

(1) Decision coverage (DC)

The DC criterion checks that all the possible decisional out-

Table 1 Relations among templates, its parameters, and instantiated signals.

Template	Template Parameters	Instantiated signal s
`Constant`	$v \in \mathcal{D}$	$s(t) = v.$
`Linear`	$b, a \in \mathcal{D}$	$s(t) = \frac{a-b}{T_{stop}} t + b.$
`NLinear`	$n \in \mathbb{N}_{>0}, v_1, \ldots, v_{n+2} \in \mathcal{D}$, $t'_1, \ldots, t'_n \in (0, T_{stop})$ where $t'_1 < \ldots < t'_n$	$s(t) = \begin{cases} \frac{v_2 - v_1}{t'_1} t + v_1 & \text{if } 0 \le t < t'_1, \\ \frac{v_{i+2} - v_{i+1}}{t'_{i+1} - t'_i} (t - t'_i) + v_{i+1} & \text{if } 1 \le i < n \text{ and } t'_i \le t < t'_{i+1}, \\ \frac{v_{n+2} - v_{n+1}}{T_{stop} - t'_n} (t - t'_n) + v_{n+1} & \text{if } t'_n \le t. \end{cases}$
`Step`	$b, a \in \mathcal{D}, t' \in (0, T_{stop})$	$s(t) = \begin{cases} b & \text{if } t' < t, \\ a & \text{otherwise.} \end{cases}$
`NStep`	$n \in \mathbb{N}_{>1}, v_1, \ldots, v_{n+1} \in \mathcal{D}$, $t'_1, \ldots, t'_n \in (0, T_{stop})$ where $t'_1 < \ldots < t'_n$	$s(t) = \begin{cases} v_1 & \text{if } 0 \le t < t'_1, \\ v_{i+1} & \text{if } 1 \le i < n \text{ and } t'_i \le t < t'_{i+1}, \\ v_{n+1} & \text{if } t'_n \le t. \end{cases}$
`Sine`	$a \in \mathcal{D}_{>0}, f \in [2\pi/T_{stop}, \pi/2T_{smpl}]$, $p \in [0, 2\pi], b \in \mathcal{D}$ where $b + a, b - a \in \mathcal{D}$	$s(t) = a \sin(ft + p) + b.$
`Square`	$a \in \mathcal{D}_{>0}, f \in [2\pi/T_{stop}, \pi/T_{smpl}]$, $p \in [0, 2\pi], b \in \mathcal{D}$ where $b + a \in \mathcal{D}, d \in (0, 1)$	$s(t) = \begin{cases} b + a & \text{if } \frac{ft+p}{2\pi} - \lfloor \frac{ft+p}{2\pi} \rfloor < d, \\ b & \text{otherwise.} \end{cases}$
`Triangle`	$a \in \mathcal{D}_{>0}, f \in [2\pi/T_{stop}, \pi/T_{smpl}]$, $p \in [0, 2\pi], b \in \mathcal{D}$ where $b + a \in \mathcal{D}, d \in [0, 1]$	$s(t) = \begin{cases} \frac{a}{d} \cdot \left(\frac{ft+p}{2\pi} - \lfloor \frac{ft+p}{2\pi} \rfloor \right) + b & \text{if } \frac{ft+p}{2\pi} - \lfloor \frac{ft+p}{2\pi} \rfloor \le d \ne 0, \\ -\frac{a}{1-d} \cdot \left(\frac{ft+p}{2\pi} - \lfloor \frac{ft+p}{2\pi} \rfloor - d \right) + b + a & \text{otherwise.} \end{cases}$

comes of every block occur at some time step in a simulation. The coverage represents the percentage of the number of observed outcomes. Target blocks of the DC criterion are logical operators, (multiport) switches, subsystems with control ports, etc. Their decisional outcomes are Boolean outputs for logical operators, indices of passing inputs for switches, enabled/disabled for subsystems, etc.

(2) Condition coverage (CC)

The CC criterion checks that all the (satisfiable) atomic conditions of every block become `true` at some time step and become `false` at another time step. The coverage represents the percentage of observed conditional outcomes. Target blocks for the CC criterion are relational operators, logical operators, etc.

(3) Modified condition/decision coverage (MC/DC)

The MC/DC criterion checks that, for every block, each atomic condition affects its decisional outcomes independently. The coverage is defined as the percentage of observed witness pairs of atomic conditions. Target blocks for the MC/DC criterion are multi-input logical operators, etc. An example witness for the i-th condition of "AND" operator is the pair of the outcome "all inputs are `true` (and thus the output is `true`) at some time" and the outcome "only the i-th input is `false` (the output is `false`) at some time."

4. Template-Based Monte-Carlo Method

4.1 Signal Templates

In general, a test case is characterized as a group of input signals with arbitrary waveforms. However, in MBD, the input signals to a controller are provided by other electronic modules that are likely to be well-controlled, or by physical objects that are dominated by physical laws. In our experience, it is frequently sufficient to consider input signals with several specific waveforms to achieve high DC, CC, and MC/DC. For instance, when a model contains a band-pass filter circuit, which only passes a wave-shaped signal with a specific frequency, we need to give such signal to exercise the downstream of the circuit. Some circuits are exercised by a signal keeping a specific constant value for a certain time whereas others are activated by a signal that changes its value suddenly and significantly. Therefore, we focus on a collection of such specific signals. For a test case consisting of them, the behavior of a model is relatively easy to understand for a design engineer. Thus it is helpful for refining the given model.

We introduce *templates* for input signals, and search test cases within the subset of instances of the templates. In this paper, we provide templates `Constant`, `Linear`, `NLinear`, `Step`, `NStep`, `Sine`, `Square`, and `Triangle`, each of which represents *constant*, *linear*, *piecewise-linear*, *single-step*, *multiple-step* (in other words, piecewise-constant), *sine-wave*, *square-wave*, or *triangle-wave* signals, respectively (Table 1). Note that we set the upper bound of the number n of pieces for `NLinear` and `NStep` templates in practice.

An individual input signal can be instantiated by choosing a template and assigning concrete values to parameters, based on (i) the range $\mathcal{D}$ of its data type, (ii) user-specified ranges, and (iii) its sampling time T_{smpl}, and (iv) stop time T_{stop} of the target model, and then rounded based on the data type and the sampling time.

Algorithm 1 Template-based Monte-Carlo test-suite generation

```
Inputs: Simulink model mdl, and template setting tmp
Outputs: Test suite ts
 1: ts := ∅;
 2: tsc := null_coverage;
 3: repeat
 4:     cand := generateTestCaseFromTemplates(mdl, tmp);
 5:     sd := simulate(mdl, cand);
 6:     candc := measureCoverage(mdl, sd);
 7:     if a covering area of candc is not subset of that of tsc then
 8:         ts := ts ∪ {cand};
 9:         tsc := accumulateCoverage(tsc, candc);
10:     end if
11: until tsc achieves full coverage
12: return ts;
```

4.2 Algorithm

We propose a template-based algorithm for Monte-Carlo test-suite generation as shown in Algorithm 1. First, we generate *randomly* an entire candidate test case (i.e., a group of input signals) based on a *template setting* specifying a set of available templates and the ranges of their parameters for each input (Line 4). Each input signal is instantiated from one of templates in Table 1. All templates in Table 1 are available and the ranges of their parameters are assumed as wide as possible by default, however, a user can customize the setting. Second, we simulate the target model for the candidate with signal logging (Line 5). Third, we measure its coverage from the simulation log (Line 6). Fourth, if the candidate is contributable to the coverage, we add it to the test suite (Lines 7–10). The steps are repeated until the test suite achieves full coverage.

If, for each DC, CC, or MC/DC objective of the target model, there exists a test case meeting the objective in the set $\mathcal{T}$ of possible test cases instantiated from the available templates, and if each candidate is chosen randomly and uniformly from $\mathcal{T}$, Algorithm 1 returns a full coverage test suite eventually.

However, there is generally no ground for the existence of a suitable test case in $\mathcal{T}$ for each objective. Additionally, if a specific value is required to meet an DC, CC, or MC/DC objective, the probability to choose it by random and uniform sampling is very low. Therefore, we practically set the maximum number of repetitions, i.e., that of candidates. When the size of the test suite reaches the maximum number, the procedure will halt even if the test suite does not achieve full coverage.

4.3 Discussions

The template-based approach has several benefits. First, it is efficient in generating a test case. A general signal is naive time-series data, so it has as many parameters as the time-steps of a simulation. A number of parameters are required to be examined to observe a complex behavior in a simulation with a certain length. On the other hand, a template-based signal is characterized by only a few parameters, which is independent of the length of a simulation. When an appropriate template is selected, test case generation becomes efficient by assigning concrete values for the reduced number of parameters. Second, the intention of a test case becomes understandable. The definition of a template determines the waveform of instantiated signals. Ramp-like, pulse-like and sawtooth signals can be instantiated from the `NLinear`, `NStep` (or `Square`) and `Triangle` templates, respectively. According to the definitions of templates, we can easily classify the signals and understand their characteristics. Additionally, we can extend our method by adding another template as needed.

As a trade-off of the benefits, the template-based approach cannot deal with complex shaped signals, and hence it may be impossible to cover some objective using any template-based test case. Additionally, it is difficult, or rather impossible, to know whether a given objective can be covered with signals instantiated from available templates, in advance. Note that there is another issue in handling a complex-shaped signal with a template because it might be an unrealistic signal.

A general advantage of Monte-Carlo approaches is that we can deal with any model, e.g., that involving non-linear computation. The Monte-Carlo test suite generation does not require to analyze the behavioral characteristics of the target model. We need only an analysis for coverage measurement. A static/dynamic analysis of the target model enables us to predict a hopeful search space (i.e., templates and the ranges of their parameters in the template-based method), but it is optional.

As a general disadvantage of naive Monte-Carlo approaches, we cannot deny the possibility that we fail to solve even a small problem. The difficulty of static formal analysis, e.g., constraint solving, strongly depends on the complexity of operations, i.e., the number/types of signals and blocks. On the other hand, that of our naive Monte-Carlo approach depends on only the size of the search space for test cases, i.e., the product of ranges of parameters of (templates assigned for) top-level input signals. So, if a desired test case can be instantiated only by a combination of values in a very narrow region against the search space, we may not find it within an acceptable length of time. This is because it is almost impossible to randomly choose a specific value from a nearly continuous search space without a user-specific limitation. This kind of difficulty is typically related with a DC/CC objective checking that a floating/fixed-point input value is equivalent to another input or constant. However, such cases are very rare in practical models.

5. Implementation

We implemented the template-based Monte-Carlo method as a tool, which contains an efficient coverage measurement engine as a replacement for the first-party implementation `cvsim` of SLC.

A design overview of our tool is provided in Fig. 2. The tool consists of a *structure analyzer*, *test case generator*,

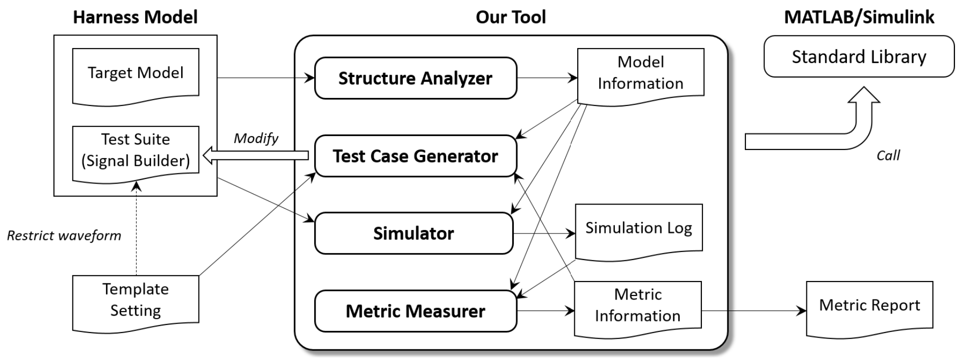


Fig. 2 Overview of the tool.

simulator, and *metric measurer*, and realizes Algorithm 1 by cooperation among them. The structure analyzer performs preprocesses for test generation. The test case generator generates new candidate test cases (Line 4 in Algorithm 1). The simulator configures the logging settings of the harness model and simulates it (Line 5 in Algorithm 1). The metric measurer receives simulation logs of test cases, and evaluates/compares/accumulates the coverages of the test cases (Lines 6, 7, 9 and 10 in Algorithm 1).

These modules are implemented as MATLAB scripts, and call standard MATLAB/Simulink functions for analysis, modification, and simulation of Simulink models.

5.1 Structure Analyzer

Our tool performs a static analysis of the structure of the model as a preprocess for test suite generation, simulation, and metric measurement. The primary functional features of the structure analyzer are: (1) listing of all blocks and connections between them as well as their parameter values, (2) detection of the signal builder block and the adapter subsystem of a harness, and (3) identification of all target blocks for DC, CC and MC/DC and the lines connected to their inputs from the target model.

The analyzer scans all blocks, including lower-level blocks in masked subsystems, recognizes their predecessors, successors and higher/lower-level blocks, and determines their concrete parameter values (e.g., a value of a constant block, a threshold of a switch block) based on mask variables of their higher-level subsystems and variables in the base workspace of the MATLAB environment.

Signals flowing on the lines between blocks may be multiplexed using multiplexer or bus-related blocks. Hence, the analyzer also temporally compiles the model and collects the multiplicity and data-type information for each of the lines.

5.2 Test Case Generator

The test case generator prepares a signal for each top-level input of "`Test Unit`," based on a template. The primary functional features of the test case generator are: (1) parsing of a text file specifying a template setting, (2) random value generation, and (3) test case instantiation based on the templates and the random values.

A template setting specifies available templates and ranges (maximum and minimum values) of the parameters for each top-level input of "`Test Unit`" Note that multiple ranges may be available for each parameter. Each range may be adjusted based on the data type recognized by the analyzer. Currently, our tool supports templates given in Table 1. A user can customize a template setting by giving a text file. An example of lines of such template setting file is given as follows.

```
In1, Step; Time = { [0, 5] }; Before = { [0, 10] }; After = { [10, 20] };
In1, Step; Time = { [5, 10] }; Before = { [10, 20] }; After = { [0, 10] };
In1, Nstep; Value = { [0, 5], [15, 20] }; Switch = { [2, 5] };
```

The example specifies three templates for the top-level input named "`In 1`." The first line specifies a `Step` template with parameter constraints $b \in [0, 10]$, $a \in [10, 20]$ and $t' \in [0, 5]$. The first line specifies another `Step` template with parameter constraints $b \in [10, 20]$, $a \in [0, 10]$ and $t' \in [5, 10]$. The third line specifies a `NStep` template with parameter constraints $n \in [2, 5]$ and $v_1, \ldots, v_{n+1} \in [0, 5] \cup [15, 20]$ and $t'_1, \ldots, t'_n \in [0, T_{stop}]$.

The generator chooses one from the available templates (resp., ranges) for each input (resp., for each parameter of a chosen template) randomly and uniformly, and then instantiates a candidate test case by assigning uniformly-random values to the parameters from the chosen ranges. Note that, we can specifically assign a certain value to a parameter of interest by using a singleton range. Therefore, we can reduce the disadvantage of Monte-Carlo approach, i.e., we can obtain a signal instantiated by a combination of specific values with a certain probability if we know each of them in advance.

Then the candidate is temporarily added to the test suite (Line 8 in Algorithm 1) by editing "`Inputs`," i.e., calling the `signalbuilder` function. If the candidate is not contributable, the generator replaces it by the next candidate.

5.3 Simulator

The simulator is a wrapper for a standard Simulink simulation function `sim`. The primary functional features of the simulator are: (1) making a mapping table between output ports and signal identifiers, (2) configuring the simulation settings, e.g., on signal logging and simulation mode, and (3) simulating the harness model for a test case in an arbitrary simulation mode.

Based on block information provided by the analyzer, the simulator gives unique names to each output port of the predecessors of target blocks of DC, CC, and MC/DC criteria. Hence, we can access the logged signal flowing out of a given output port via the corresponding name.

The simulator activates the newest test case by calling the `signalbuilder` function and then invokes the `sim` function. Here, simulation can be performed with any simulation mode: normal, accelerator, or rapid.

5.4 Metric Measurer

The metric measurer collects signal data of interest from the simulation log and evaluates the metrics for that test case. The primary functional features of the metric measurer are: (1) coverage measurement for a test case, (2) comparing/accumulating coverages of test cases/suites, and (3) reporting the accumulated coverage result for a test suite.

For each target block of the DC, CC, and MC/DC criteria, the measurer reads the signal logs fed into the block, which are identified by the names given by the analyzer.

5.4.1 Coverage Measurement

According to the types of conditions and/or decisions and parameters of a block, the measurer evaluates the outcomes of the conditions and/or decisions at each step. The coverage for the block is derived from the time series of the outcomes. Then, the measurer sums up the coverages for each target block to calculate the coverage for the test case. The coverage for a test suite is obtained by summing the individual coverages for each test case.

5.4.2 Comparison with SLC

In this section, we explain our coverage measurement engine, and contrast it with the coverage measurement function `cvsim` provided by SLC. As SLC is a proprietary software, our observation here is based on our experiments.

(1) Features of SLC coverage measurement

Normal mode is the only simulation mode supported by `cvsim`. Accelerator and rapid modes are not supported. Instead, we can pause simulation and immediately obtain intermediate results. Based on this observational evidence, we inferred that `cvsim` repeats the following steps: (i) advancing the simulation by a step, (ii) checking whether each decision/condition holds, and incrementing a counter for each objective if it is satisfied at that time. This approach processes a time series data in a `for`-loop and is very inefficient.

(2) Features of our coverage measurement

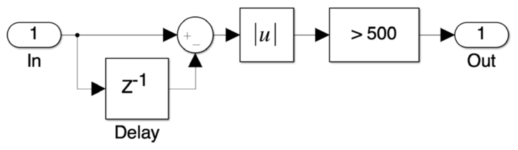
In contrast, our tool employs the following approach: First, based on structure analysis, we enable logging for all output ports, which provide signals feeding in blocks that should be measured. Second, a target harness model is simulated until completion without pause. Finally, we check whether every decision/condition holds for each step and count the number of steps that satisfy decisions/conditions by applying array/matrix operations to the simulation log. In this approach, we can use any simulation mode because we assume that simulations are pauseless. Therefore, we can accelerate coverage measurement by using fast simulation modes and efficient array/matrix operations of MATLAB/Simulink.

6. Evaluation

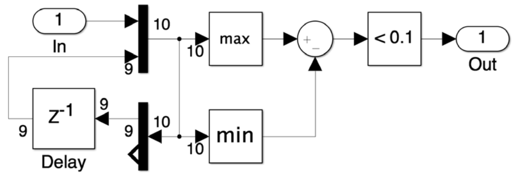
This section shows experimental evaluations. All of the experiments were done with macOS 10.13.4 and MATLAB/Simulink (including SLDV and SLC) R2017b on a PC with a 2.8 GHz Intel Core i7 CPU and 16 GB of RAM.

6.1 Basic Evaluation

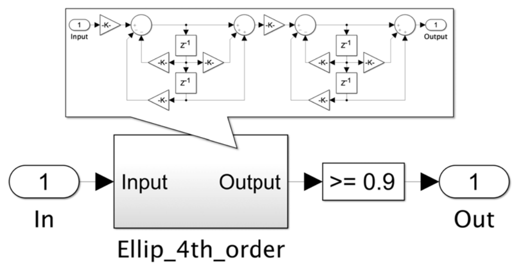
First, we report the results of basic experiments with small models to evaluate the effectiveness of our template-based method proposed in Sect. 4. In the experiments, we used three models m_1, m_2 and m_3 given in Fig. 3. SLDV can generate a full coverage test suite for m_1 and m_2, but cannot for m_3 even with more than an hour.



(i): A sudden change detector m_1



(ii): A steady state detector m_2



(iii): A band-pass filter m_3

Fig. 3 Small models for basic experiments. For m_1, the value range of its input signal is [0, 1000], its detect threshold is 500 for a time step. For m_2, the value range of its input signal is [0, 1000], its detect threshold is 0.1 for 10 time steps. For m_3, the value range of its input signal is $[-1, 1]$, its pass frequency is around 0.63-0.68 rad/s, its detect threshold is 0.9. For all of them, sample time is 1s and stop time is $1,000$s.

(1) Descriptions

We compared the following randomized methods.

1. Uniform Random: For each time step, a value of an input signal is uniformly and randomly chosen from the value range.
2. Wiener-process: The initial value of an input signal is uniformly and randomly chosen from the value range. A change for each time step follows a (truncated) normal distribution $N(0, \sigma^2)$ where the standard deviation σ is uniformly and randomly chosen, once for each test case, from a hundredth of the width of the value range.
3. Our method (Algorithm 1): A test case is generated by our method with the following setting, called "ALL": all templates in Fig. 1 are enabled, and any additional constraints for template parameters are not given. For only m_3, we also use the following setting, called "SS": "`Sine`" template with constraints "amplitude and bias is fixed to 1 and 0, respectively" and "`Square`" template with constraints "amplitude and bias is fixed to 2 and -1, respectively" are enabled, and all other templates are disabled.

We observed 10 trials for each of them and models. For each trial, we repeatedly generated candidate test cases (up to 1,000) and measured its coverage, and scored the number of candidates required to achieve full coverage.

(2) Results

The results are summarized in Table 2. Hyphens mean that the numbers of candidates exceed 1,000. Each trial completes up to few minutes. Random signals are extremely unstable, thus only sudden changes were normally observed. Wiener-process signals with smaller variances are more stable, therefore steady states were observed when its variances were enough small. The results indicate that these naive types of random signals is suitable only in limited cases. Our template-based method assumes various types of signal templates, so got to normally produce stimulating signals. For m_3, our method required a suitable template setting to efficiently generate contributable test cases, because few templates are suitable for band-pass filter and the conditions for parameters to pass the filter are strict. Although, for each input signal, the range of its values and considered types of its waveforms are usually limited and known by modeling engineers. So it is expected to improve the efficiency of our method by giving a suitable template setting as suggestions based on such their knowledge.

6.2 Practical Evaluation

Second, we report two experimental results to practically evaluate the quality of the generated test suites (Sect. 6.2.1) and the coverage measurement performance (Sect. 6.2.2) of our tool introduced in Sect. 5.

In the experiments, we focused on implementation models of realistic applications. To the best of our knowledge, there are no open benchmarks of such models. We have evaluated our tool with several industrial models and obtained good results, however, all of the models are confidential and most of the results cannot be open. In this paper, we report the summarized results for (closed-source) two Simulink models M_1 and M_2 developed as real products in the industry; statistical data are summarized in Table 3. Note that, two and (at least) 56 objectives in M_1 and M_2, respectively, are unsatisfiable. They are detected by SLDV and confirmed manually.

Table 2 Sorted numbers (from the best case to the worst case) of candidates to fully cover objectives in small models m_1, m_2 and m_3.

Trials	Unif. Random			Wiener Proc.			Our Method			
							All			SS
	m_1	m_2	m_3	m_1	m_2	m_3	m_1	m_2	m_3	
Best	1	-	-	-	470	-	1	1	-	1
2nd	1	-	-	-	625	-	2	1	-	12
3rd	1	-	-	-	-	-	3	1	-	17
4th	1	-	-	-	-	-	3	1	-	24
5th	1	-	-	-	-	-	3	2	-	25
6th	1	-	-	-	-	-	3	2	-	34
7th	1	-	-	-	-	-	4	2	-	48
8th	1	-	-	-	-	-	11	2	-	67
9th	1	-	-	-	-	-	12	3	-	78
Worst	1	-	-	-	-	-	16	11	-	79

Table 3 Summary of experimental models M_1 and M_2.

Attributes	M_1	M_2
#Inputs	2	51
#`uint8`-typed signals	0	7
#`int16`-typed signals	1	3
#`uint16`-typed signals	0	1
#`single`-typed signals	0	40
#`double`-typed signals	1	0
#Blocks	476	2374
#RelationalOperator blocks	16	120
#Logic blocks	7	7
#Switch blocks	1	152
#MultiPortSwitch blocks	33	42
#Abs blocks	0	1
#MinMax blocks	0	1
#If blocks	0	11
#Subsystems	29	145
#Enable subsystems	2	4
#Trigger subsystems	5	0
#IfAction subsystems	0	24
#Objectives	200	710
#DC objectives	123	428
#CC objectives	64	268
#MC/DC objectives	13	14
StopTime	3.0	7.0
SampleTime (base)	1/20000	1/1000

6.2.1 Quality of Generated Test Suite

In the first experiment, we compared the coverages of test suites generated by our tool with those generated by SLDV.

Table 4 Comparison between qualities of generated test suites.

Model	Tool		Coverage	Size of test suite	#Candidates	Execution time (sec)
M_1	Our tool	Best	198/200	5	9	29.9
		Average	198/200	6.6	69.5	209.3
		Worst	198/200	10	151	445.5
	SLDV		144/213	7	-	>600
M_2	Our tool	Best	395/710	6	16	>600
		Average	367.9/710	8.5	16	>600
		Worst	335/710	11	16	>600
	SLDV		232/549	8	-	>600

(1) Description

We generated test suites for M_1 and M_2 in two ways, using either our tool or SLDV. Then, we compared the coverages, sizes of test suites (i.e., numbers of contributable test cases), number of candidates (for our tool), and execution times. The timeout was set to 600 seconds.

For our tool with the Monte-Carlo approach, we performed 20 trials for each model. Increasing the number of trials was effective when revealing the characteristics of the target models, but had only a minor benefit to evaluate the quality of a Monte-Carlo based tool. The simulations in all trials were performed in accelerator mode with the fast restart option. We used the following template settings: for M_1, we enabled the templates suggested by industry engineers for both top-level inputs: `Step` for the `int16`-typed input, and `Constant`, `Linear` and `Sine` for the other `double`-typed input. The engineers also provided the ranges of the parameter values. For M_2, we enabled `Step` and `Linear` templates for every top-level input. We used the default ranges of the parameter values (i.e., those of the corresponding data types for the inputs).

In contrast, we only observed one SLDV trial for each model due to its non-randomness.

(2) Results

Our results are summarized in Table 4. Our tool employs a randomized method, so we list the best, average, and worst values among the 20 trials. The execution times for our tool include the time taken for the structure analysis. The differences between the total numbers of DC, CC, and MC/DC objectives on M_1 are due to the differences between the MC/DC objective counting policies. Our tool (resp., SLDV) count 1 (resp., 2) for each witness pair of MC/DC criterion. Additionally, SLDV ignored the unsupported objectives on M_2, e.g., those related to blocks in the downstream of non-linear arithmetic. This is clear because the total number of objectives was less than double the number of relational operator blocks and (multiport) switches.

The experimental results demonstrate that our tool can achieve higher coverage than SLDV, with shorter execution times, against practical implementation models. In the case of the medium-sized model M_1, the coverage achieved by SLDV was only 70%, but all trials of our tool achieved practically full coverage (because 2 objectives are unsatisfiable) with short execution times, thanks partly to suggestive template settings from the engineers. Neither tools achieved full coverage in the case of large model M_2, but all trials of our tool covered more objectives than SLDV even without suggestive template settings.

Table 5 Comparison on performance on coverage measuring.

Model	Tool	Sim. mode	Exec. time (sec)
M_1	Our tool	Normal	1.8
		Accelerator	2.9
		Rapid	10.2
	SLC	Normal	50.6
M_2	Our tool	Normal	54.6
		Accelerator	38.5
		Rapid	30.0
	SLC	Normal	258.4

6.2.2 Performance on Coverage Measurement

In the second experiment, we compared the performances on coverage measurement of our tool to that of SLC.

(1) Description

For M_1 and M_2, we observed execution times of coverage measurement of our tool and SLC under each available simulation mode. For each model, we used a test suite consisting of 20 test cases generated by our tool.

We enable the fast restart option in the normal and accelerator modes, so the execution times did not include the recompilation for simulation. In the case of rapid mode, the fast restart option must be disabled, and the recompilation is unavoidable for each simulation. Hence, in this case, the execution times included every recompilation time for simulation.

(2) Results

We assessed the execution times of coverage measurement of our tool and SLC, and then calculated the average time per test case. Note that the coverage measurement has no randomness, and therefore, for each tool and each model, the execution times for test cases were almost the same. Our results are summarized in Table 5.

The experimental results confirm that our tool performs better than SLC. Our tool evaluated the coverage about 5 times faster than SLC in each simulation mode. In particular, our tool ran more than 25 times faster than SLC in some cases. In the case of M_1, the faster simulation modes were slower than the normal mode. We conjecture that the recompilation and API communication overheads of the faster simulation modes have a more significant effect when testing smaller models. Actually, in our manual measurement, most of execution times in the rapid mode were taken for recompilation. Conversely, we conclude that using faster simulation modes is very effectively when measuring test coverage for large models.

7. Conclusions

In this paper, we proposed a scalable test suite generation method, based on signal templates, for practical Simulink models, and implemented it as a tool. The tool succeeded in dealing with real implementation models used in industry and generated a higher-coverage test suite than the first-party toolbox SLDV. Moreover, the tool evaluates test coverage much faster than the first-party toolbox SLC. This enables it to generate and check more candidates per hour. Thus, we can obtain effective test cases within a reasonable length of time.

In the future, we will improve the sophistication of the tool based on the results of trials in real development teams. Furthermore, we will combine the current method with light-weight formal analysis that can narrow the range of values of template parameters. In contrast to using conservative approximations in purely formal approaches, we may accept loose approximations for this light-weight analysis. It is enough useful to extract hopeful regions from the search space. Heuristic techniques will improve the efficiency of test suite generation.

References

[1] T. Tomita, D. Ishii, T. Murakami, S. Takeuchi, and T. Aoki, "Template-based Monte-Carlo test generation for Simulink models," Cyber Physical Systems. Design, Modeling, and Evaluation, R. Chamberlain, W. Taha, and M. Törngren, eds., LNCS 11267, pp.63–78, Springer International Publishing, 2019.

[2] T. Tomita, D. Ishii, T. Murakami, S. Takeuchi, and T. Aoki, "A scalable Monte-Carlo test-case generation tool for large and complex Simulink models," Proc. 11th Workshop on Modelling in Software Engineering, MiSE'19, pp.39–46, IEEE Press, 2019.

[3] International Electrotechnical Commission, "IEC 61508 functional safety of electrical/electronic/programmable electronic safety-related systems."

[4] International Organization for Standardization, "ISO 26262 road vehicles – functional safety."

[5] P. Schrammel, D. Kroening, M. Brain, R. Martins, T. Teige, and T. Bienmüller, "Incremental bounded model checking for embedded software," Formal Aspects of Computing, vol.29, no.5, pp.911–931, Sept. 2017.

[6] S. Mohalik, A.A. Gadkari, A. Yeolekar, K. Shashidhar, and S. Ramesh, "Automatic test case generation from Simulink/Stateflow models using model checking," Softw. Test. Verif. Reliab., vol.24, no.2, pp.155–180, 2014.

[7] S. Sims and D.C. DuVarney, "Experience report: The Reactis validation tool," Proc. 12th ACM SIGPLAN International Conference on Functional Programming, ICFP'07, pp.137–140, New York, NY, USA, ACM, 2007.

[8] P. Peranandam, S. Raviram, M. Satpathy, A. Yeolekar, A. Gadkari, and S. Ramesh, "An integrated test generation tool for enhanced coverage of Simulink/Stateflow models," Proc. Conference on Design, Automation and Test in Europe, DATE'12, pp.308–311, San Jose, CA, USA, EDA Consortium, 2012.

[9] S. Raviram, P. Peranandam, M. Satpathy, and S. Ramesh, "SmartTestGen+: A test suite booster for enhanced structural coverage," Theoretical Aspects of Computing – ICTAC 2012, A. Roychoudhury and M. D'Souza, eds., pp.164–167, Berlin, Heidelberg, Springer Berlin Heidelberg, 2012.

[10] M. Satpathy, A. Yeolekar, and S. Ramesh, "Randomized directed testing (REDIRECT) for Simulink/Stateflow models," Proc. 8th ACM International Conference on Embedded Software, EMSOFT'08, pp.217–226, New York, NY, USA, ACM, 2008.

[11] A. Brillout, N. He, M. Mazzucchi, D. Kroening, M. Purandare, P. Rümmer, and G. Weissenbacher, "Mutation-based test case generation for Simulink models," Formal Methods for Components and Objects, F.S. de Boer, M.M. Bonsangue, S. Hallerstede, and M. Leuschel, eds., pp.208–227, Berlin, Heidelberg, Springer Berlin Heidelberg, 2010.

[12] R. Matinnejad, S. Nejati, L.C. Briand, and T. Bruckmann, "Automated test suite generation for time-continuous Simulink models," Proc. 38th International Conference on Software Engineering, ICSE'16, pp.595–606, New York, NY, USA, ACM, 2016.

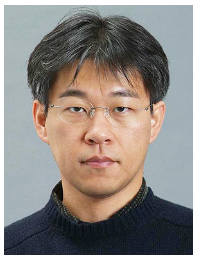
[13] R. Matinnejad, S. Nejati, L.C. Briand, and T. Bruckmann, "SimCoTest: A test suite generation tool for Simulink/Stateflow controllers," Proc. 38th International Conference on Software Engineering Companion, ICSE'16, pp.585–588, New York, NY, USA, ACM, 2016.

Takashi Tomita received the B.S., M.S., and Ph.D. degrees in engineering from Tokyo Institute of Technology in 2007, 2009, and 2013, respectively. During 2013–2015, he was a researcher in the institute. Since 2015, he has been an assistant professor at Japan Advanced Institute of Science and Technology. He is engaged in the researches on formal methods, quantitative verification, probabilistic/statistical model checking, specification verification, automated program synthesis, and systems biology.

Daisuke Ishii received the B.Eng., M.Eng., and Ph.D. degrees in computer science from Waseda University in 2001, 2003, and 2010, respectively. He was a research fellow at INRIA/LINA in France (2010–2011), a research fellow of the JSPS at National Institute of Informatics (2011–2013), and an assistant professor at Tokyo Institute of Technology (2013–2015). He is currently an associate professor at University of Fukui. His research interests include interval analysis and formal methods for hybrid systems.

Toru Murakami was a manager of Engineering Department, Cybernet Systems Corporation (an agent company of MathWorks in Japan), until 2011. Currently, He is a model-based development expert in Technology Development Division, Gaio Technology Corporation. He is engaged as a chief consultant on the model-based development in OEM/supplier companies.

Shigeki Takeuchi was a system architect and manager of in-vehicle infotainment system with the model-based development in Sony Corporation until 2013. Currently, He is a deputy general manager in Technology Development Division, Gaio Technology Corporation. He has examined model-based systems engineering that integrates the model-based design and the model-based test for OEM/supplier companies.



Toshiaki Aoki is a professor in Division of Transdisciplinary Sciences and School of Information Science, JAIST. He received B.S. degree from Science University of Tokyo (1994), M.S. and Ph.D. from Japan Advanced Institute of Science and Technology (1996, 1999). His research interests include formal methods, formal verification, theorem proving, model checking, automotive systems and embedded systems.