# A Model-based Test Cases Generation Method for Spacecraft Software

Ye Tian
*School of Automation Science and Electrical Engineering*
*Beihang University*
Beijing, China
tianye0039@buaa.edu.cn

Chenglong Li
*School of Automation Science and Electrical Engineering*
*Beihang University*
Beijing, China
li_chenglong@buaa.edu.cn

Beibei Yin *
*School of Automation Science and Electrical Engineering*
*Beihang University*
Beijing, China
yinbeibei@buaa.edu.cn

***Abstract*—In order to systematically generate sufficient test scenes for the spacecraft's controller software, an automatic model-based test cases generation method is proposed in this article. More specifically, the test requirement is modelled as a statechart diagram using Unified Modeling Language, which can be converted into an intermediate model. After that, test scenes that consider the entire natural scenes are automatically generated from this intermediate model. In this paper, a method to generate test scenes in a particular distribution is proposed, which may reveal more mistakes in the software. The testing framework is a model-based test cases generation method combined with Combinatorial Testing and Adaptive Random Testing, which improves the deficiencies of the designing test scenes manually. The test scenes generated by this method can be directly used for the test of spacecraft.**

***Keywords: spacecraft software, model-based testing, test cases generation, Combinatorial Testing, Adaptive Random Test-ing***

## I. Introduction

With the continuous evolution of spacecraft designing technology, the autonomy and intelligence of spacecraft are getting higher and higher. Some spacecraft can complete the corresponding tasks according to the ground command and adjust the tasks according to the changes of the space environment. The increasing autonomy and intelligence of spacecraft mean a higher demand for its ground testing. The focus of the testing has changed from verifying the function and performance of hardware to examining its ability of mission completion in the real space scene.

It takes the source code as the testing object in traditional code-based testing, which is a necessary and critical process for spacecraft software development. However, this kind of testing cannot directly verify whether the desired function of the software has been achieved correctly. Therefore, the code-based testing alone can not satisfy the requirements of reliability and quality of spacecraft systems with increasing functions and complexity.

A typical example is the controller software of the spacecraft. After a spacecraft enters orbit, it needs to complete a series of actions such as despinning and unfolding the solar panels before it is ready to perform tasks. These actions are completed under the ground command and the space environment. The controller software that completes this series of actions is the test object of the paper.

In order to thoroughly test the spacecraft before its launching, testers need to treat the control system as a black box and test its function in a simulation environment. In the past, testers manually designed test cases to carry out on the software, which is time-consuming and laborious. However, spacecraft has plenty of working states and transitions among these states are complex. Moreover, the logic conditions of the transition also may be complicated. So the adequacy and comprehensiveness of such tests, highly dependent on personal experience, are suspicious. In this regard, research on how to automatically generate better test cases for controller software is conducted.

Concerning automatic testing, there are many test generation methods. Model-based Testing (MBT) provides a technique for test cases generation using models extracted from software under test [1]. A formal model describing the software or system behaviour is needed in an MBT method. Moreover, the survey showed that MBT was mostly applied to systematic testing [2]. The Unified Modeling Language (UML) statechart is a better model to be used in model-based test cases generation due to its capabilities to capture the changes in the lifecycle of an object [3]. In fact, differentiating the MBT approaches by the model used, the UML statechart based method was mostly used [2]. The model used in our paper is the UML statechart diagram. Combinatorial testing techniques deal with the problem of reducing the considered input combinations while ensuring the adequacy of the resulting test suite [4].

* Beibei Yin is the corresponding author.

2767-6684/21/$31.00 ©2021 IEEE
DOI 10.1109/DSA52907.2021.00057

Cu D. Nguyen proposed a method to combine model-based and combinatorial testing approaches. The idea of that is used in our paper because the combination of parameters of spacecraft is huge. According to Adaptive Random Testing (ART) [5], a program input far away from non-failure-causing inputs may have a higher probability of causing failure than the neighbouring test inputs, which may guide the generation of concrete inputs for the spacecraft.

We divide the problems to be solved into the following parts:

1.How can testing requirements of spacecraft controller software be modelled with all test information preserved and without any ambiguity?

2.How to find all the paths that can satisfy the test requirements from the statechart diagram?

3.How can a test path of spacecraft be divided into disjoint equivalence classes considering the complexity of conditions and their combinations on the path?

4.How to generate concrete test scenes from each equivalence class and ensure the coverage of test scenes in the input domain?

An automatic model-based test case generation method was proposed in this paper, and the method used will be introduced from the following aspects in the next section:

Subsection A will illustrate the testing requirements of controller software and the modelling rules.

Subsection B will introduce the method to transform the statechart diagram into a directed graph, an intermediate model.

Subsection C will show how to find test paths from the directed graph to satisfy the test requirements.

For each test path found, subsection D will illustrate how to divide it into equivalence classes.

Subsection E will show the technology to generate adequate test scenes from each equivalence class automatically. In the end, a metric for the difference between test scenes and a method to generate test cases in a particular distribution will be proposed.

## II. Related Work

The MBT depends on three key technologies: (i) the model used for the software behaviour description, (ii) the test-generation algorithm, and (iii) tools that generate supporting infrastructure for the tests [1].

The UML statechart is a better model to be used in model-based test cases generation due to its capabilities to capture the changes in the lifecycle of an object [3]. Differentiating the MBT approaches by the model used, the UML statechart based method was mostly used [2]. In the UML statechart diagram, each state represents a working state of the system, and the whole diagram describes the test requirements of the system.

Test requirements are specific things that must be satisfied or covered during model-based testing. Guided by the testing criteria, test cases generated should be adequate to make test requirements be satisfied. According to Jeff [6], The adequacy criteria can be divided into specification-based criteria or program-based ones. In particular, A specification-based criterion specifies the required testing in terms of identified features of the specifications of the software. It can be used for black-box functional testing of the system. A specification plays two main roles in software testing [7]. One is to provide the necessary information to check whether the program's output is correct, known as the oracle problem [8]. The other is to provide information to select test cases and measure test adequacy [9].

The results of the paper [10] revealed that most of the work in test case generation using UML statecharts translated the UML statechart diagram into an intermediate model. The intermediate models include extended finite-state machine [11], directed graph [12], testing flow graph [13], state graph [14], and so on. While converting the statechart diagram into the intermediate model, the model always needs to be flattened due to states' hierarchical and concurrent structure [11]. Then the test cases are generated later from the intermediate model by applying various algorithms, such as genetic algorithm (GA) [15], Depth First Search (DFS) [16], Breadth-First Search (BFS) [17] or other customized algorithms.

The testing criteria, including Transition Coverage, Full Predicate Coverage, Transition-Pair Coverage and Complete Sequence were described in the first formal testing technique based on UML [6]. In addition to these criteria, State Coverage [13] is another commonly used testing criteria. It should be noted that the Basic Path Coverage criterion was proposed while generating test cases from activity diagrams using gray-box method [18]. This conception was used in testing generation from the UML statechart diagram [19].

Functional testing and specification-based testing aim at defining test cases that can exercise the functional requirements of a system, by systematically selecting representative combinations of its parameter values [20]. Combinatorial testing techniques deal with the problem of reducing the considered input combinations while ensuring the adequacy of the resulting test suite [4]. Cu D. Nguyen proposed a method to combine model-based and combinatorial testing approaches [20] by deriving test sequences from models and by completing them with selective test input combinations. Considering each path as a root of a tree, the leaves of the tree define equivalence classes of input data, and the combinatorial heuristics define the combinations to use during testing. However, the level of automation of the method proposed is not particularly high as the equivalence classes of the events need to be manually configured. Moreover, concrete test inputs were created based on the classification specifications, but the distribution of the inputs was not taken into consideration.

Considering the distribution of the inputs, program inputs that trigger failures (failure-causing inputs) tend to cluster into contiguous regions (failure regions) [21]. So, a program input that is far away from non-failure-causing inputs may have a higher probability of causing failure than the neighbouring test inputs, which is the idea of ART [5]. Based on this, ART

aims to achieve an even spread of (random) test cases over the input domain. ART generally involves two processes: one for the random generation of test inputs; and another to ensure an even-spreading of the inputs throughout the input domain [22].

## III. Method

In this section, the method of generating test cases from test requirements will be illustrated in order. Before we begin, the entire flowchart of automatic test scenes generation is attached below for clarity of presentation, shown in Fig. 1.

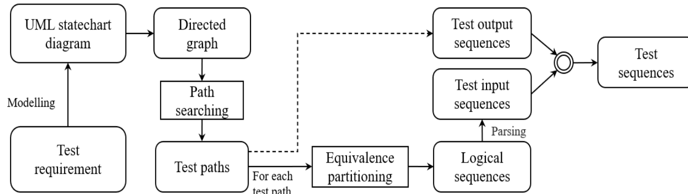


Fig. 1. Flowchart of automatic test scenes generation

### A. Modelling for test requirements

In the introduction, we mentioned that after the spacecraft enters orbit, the spacecraft needs to complete a series of actions such as despinning and unfolding the solar panels before it is ready to perform tasks, as shown in Fig. 2.

So let us see how the testing goes: in the simulation environment, we need to determine whether, during the simulation, the spacecraft can respond correctly to the commands from the ground considering the simulated space environment where it is.

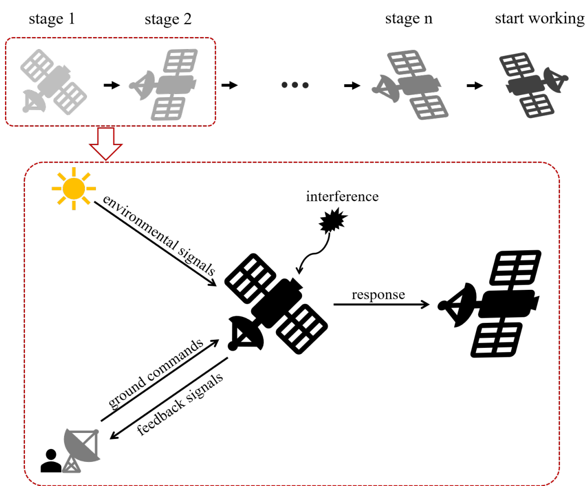


Fig. 2. The control process of the spacecraft

Here is a simple example:

After the spacecraft enters orbit, the ground gives the spacecraft an instruction, telling the spacecraft to "begin despinning". During the process of despinning, spacecraft needs to generate two signals, telling the ground, "I am despinning". When the angular velocity sensors display that rotation speed has been reduced to a certain level, the spacecraft automatically turns to the next stage, where it starts unfolding the solar panels. Moreover, during this process, the spacecraft generates another two signals to tell the ground, "I am unfolding the solar panel". The process continues until the sensors show that the solar panels have been unfolded.

That is what developers expect the software to do. If there is any inconformity between behaviours expected and those of the software during testing, the test fails.

The above is a small part of the whole control process. There may be complex transitions between the behaviour modes of the spacecraft, so we built the dynamic behaviour of the spacecraft into a UML statechart diagram to describe that.

We combed through the information needed while testing the spacecraft controller software and divided it into four categories:

- Current working states of the spacecraft (CSOS):
  Such as "despinning mode" and "solar panels are unfolding mode", and so on. The spacecraft can only be in a specific working state at any given moment.
- Ground commands and interferences received by the spacecraft (GCAI):
  Ground commands are easy to understand, and interferences refer to some unexpected conditions divided into two parts: external interferences and internal ones.
  External interferences: For example, an external impact (a rock hits the spacecraft) can be detected by the spacecraft's torque sensors, and an associated signal can be transmitted to the controller software.
  Internal interferences: For example, if one momentum wheel breaks down, it is also an interference for the controller.
  These signals may change the working state of the spacecraft.
- Sensors' values which represent the state of the spacecraft (SV):
  For instance, the angular velocity sensor's value can directly reflect whether the spacecraft is still spinning.
- Feedback signals from the spacecraft (FS):
  The spacecraft keeps the ground informed of its current condition by sending feedback signals to ground station.

TABLE I
MODELLING RULES

| Information needed while testing | UML state diagram element |
|---|---|
| CSOS | State |
| GCAI | Signal event |
| SV | Change event |
| FS | Action |

The testing requirements of spacecraft controller software were modelled according to the rules above,as shown in Table I. A UML statechart diagram was built without ambiguity, making it more visible for testers to view and modify the

testing requirements. The UML statechart diagram built is the model from which we generate the test scenes for spacecraft.

***Definition 1*:** The UML statechart diagram modelled is defined as a tuple $D = \langle S, T, \delta, s_0, F \rangle$, Where

$S$ is the set of all states. A state could contain actions;

$T$ is the set of transitions. A transition can contain events with predicate expressions;

$\delta$: $S \times T \rightarrow S$ is a function describing the transitions between states;

$s_0 \in S$ is the initial state;

$F \in S$ is the final state.

It should be noted that there may be cycles in the state diagram. For example, when the spacecraft is in a particular working mode, the torque sensor captures a change due to the impact of a meteorite. According to the design requirements, the spacecraft should not be influenced and continue working in this mode.

Assume that state $A$ is the current state of the spacecraft in the state diagram. When the torque sensor gets a value, a transition starting from state $A$ to state $A$ will be triggered. This kind of cycle in the statechart diagram is shown in the Fig. 3.

In addition, the transitions in the established UML statechart diagram for spacecraft controller software have no guard conditions, which is determined by the features of spacecraft controller software. When the spacecraft is in a specific state, once an event that satisfies the trigger condition occurs, the transition occurs.

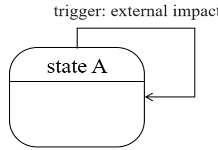


Fig. 3. A cycle in diagram

### *B. Model preprocessing*

This section will show how to convert the statechart diagram $D$ into a directed graph $G$.

The established UML statechart diagram contains all the information in the testing requirements, which can generate the corresponding XML (Extensible Markup Language) document through XMI (XML based Metadata Interchange).

The transformation process ensures the consistency of the information in the UML statechart diagram and the information in the XML document. In an XML document, the states and transitions of the diagram are stored as elements with attributes and other elements. DOM (Document Object Model) method was used to extract information from the XML document [23]. All the elements of the XML document were stored in a tree structure by the DOM method, and there are functions to access or modify specific elements.

The model built using a statechart diagram may contain hierarchical relationships. For example, a state may contain several substates, which is called a composite state. We modelled some states of the system as composite states. For example, one of its states is 'state $A$', which contains two different substates, '$A1$' and '$A2$' (As shown in the figure, an empty state is added according to the specification for UML modelling). Maybe two different controlling methods can be used when the spacecraft was controlled to complete an action. As shown in Fig. 4,the spacecraft in the state '$A1$' or '$A2$' is also in state '$A1$'.

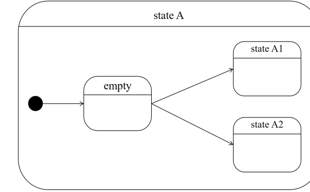


Fig. 4. A composite state

Fig. 5 shows a part of a whole statechart diagram. Although this kind of hierarchical relationship can reduce the complexity of the model, it is unobvious to find the relationship among states from the diagram containing such composite states. So the model needs to be flattened to make it easy to understand the semantics of the model and convenient to generate test cases from it. The flattening rule is shown in this section.

We assume that for a composite state $s_c$, if the initial state inside the composite state is $s_{c-I}$, there are $n$ substates $s_{ci}(1 \leq i \leq n)$, and the substate connected to the state $s_{c-I}$ is $s_{c1}$. The following is the strategy of flattening for spacecraft:

1) The target state of a transition $t$ goes into a composite state $s_c$ may be the composite state $s_c$ or its substate $s_{ci}$, and $t$ goes from $s_1$.
   If the target state of $t$ is $s_c$, generate a new transition $t'$ from the state $s_1$ to the substate $s_{c1}$ connected to the initial state $s_{c-I}$;
   If the target state of $t$ is $s_{ci}$, generate a new transition $t'$ from the state $s_1$ to the substate $s_{ci}$.
   Then delete the original transition $t$, and add the entry actions of the composite state $s_c$ to the effects of the new transition $t'$.
2) The source state of a transition $t$ goes from a composite state $s_c$ may be the composite state $s_c$ or its substate $s_{ci}$, and $t$ goes into $s_2$.
   If the source state of $t$ is $s_c$, generate $n$ new transitions $t^i$ $(1 \leq i \leq n)$ from the substates $s_{ci}$ $(1 \leq i \leq n)$ to the state $s_2$;
   If the source state of $t$ is $s_{ci}$, generate a new transition $t'$ from the state $s_{ci}$ to the state $s_2$.
   Then delete the original transition $t$, and add the exit actions of the composite state $s_c$ to the effects of the new transition $t^i$ or $t'$.
3) Delete the initial state in the composite state $s_c$.
4) Add the do actions of the composite state $s_c$ to those of every substate $s_{ci}$ of $s_c$.

According to the rules above, a statechart diagram $D$ (for example, shown in Fig. 5) was transformed to a flattened one

$D'$ (shown in Fig. 6).

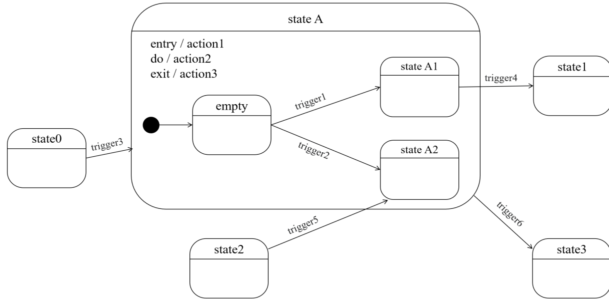


Fig. 5. A diagram with a composite state

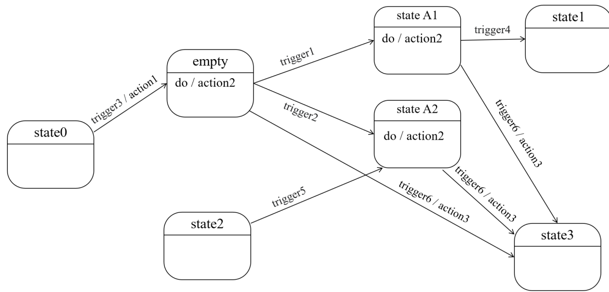


Fig. 6. Flattened diagram

Since there is no guard condition for the triggering event according to the modelling method in this paper, there is no infeasible path after flattening. (If the trigger condition has a guard condition and the diagram is flattened, there may be a path whose guard condition may have a conflict with other conditions of it in the flattened diagram, in which case conflicts-resolving method is needed [24]).

***Definition 2*:** A directed graph $G$ can be defined as a tuple: $G = \langle V, E \rangle$, Where

$V$ is a non-empty set of vertices, used to represent states information;

$E$ is a set of directed edges, used to represent the information of transitions between states.

Each $e$ in $E$ corresponds to a tuple $\langle u, v \rangle$, where $u$ and $v$ are two elements in $V$ and vertex $u$ and $v$ are the head and tail of the edge $e$, respectively. To get the directed graph $G$ from the flattened statechart diagram $D'$, the information of states and transitions in the XML document of $D'$ was stored in the instantiated vertex class and edge class. In particular, the initial state of $D'$ involves the starting vertex, and the final state involves the ending vertex. As an attribute of the vertex class, the adjacency list was used to store the information that by which edge this vertex is connected to other vertices. So far, a directed $G$ graph class was built.

### C. Test paths searching

The flattened directed graph $G$ records all the states of the spacecraft and transitions of them. The path from the beginning vertex to the ending vertex represents the spacecraft's various states and their transitions, from entering the orbit to the end of control.

***Definition 3*:** A **test path** ($TP$) is an abstract path that represents a sequence of states and transitions from the initial state of the spacecraft to the final state, which can be defined as a list $TP = [v_0, e_1, v_1, e_2, v_2, \ldots, e_n, v_n]$, where

$v_i(0 \leq i \leq n)$ is a vertex in the directed graph $G$ representing a state of spacecraft. In particular, $v_0$ and $v_n$ represent the initial state and the final state of the spacecraft, respectively;

$e_i(0 \leq i \leq n)$ is an edge from $v_{i-1}$ to $v_i$.

The first step for test case generation is to find the set of such paths from the directed graph.

Testers can choose different levels of test coverage according to their requirements, and different levels of coverage imply different sets of paths. The selection of test coverage criteria should be based on the adequacy and the cost of the test.

Coverage criteria based on the statechart diagram include state coverage, transition coverage, basic path coverage, full ZOT path coverage [25] and so on.

Here are the conceptions of these coverage criteria:

1) State coverage: the set of test paths can cover every state in the state machine diagram.
2) Transition coverage: the set of test paths can cover every transition in the state machine diagram.
3) Basic path coverage: the set of test paths can cover every independent path of the statechart diagram. If there is a loop, the loop is executed for 0 and 1 time. A independent path has at least one transition which is not covered by any other paths in the set.
4) Full ZOT path coverage: the set of test paths can cover every independent path outside the loop, and each loop of the statechart diagram is executed for 0, 1 and 2 times, respectively.

The following are the practical significances of these coverage criteria for the spacecraft software:

1) The set of test paths satisfying the state coverage will cover every possible working state of the spacecraft in space.
2) The set of test paths satisfying the transition coverage will cover every event which can cause the transition between working states of spacecraft.
3) The set of test paths satisfying the basic path coverage will cover every independent path of the spacecraft.
4) The set of test paths satisfying the full ZOT path coverage will cover every independent path outside the cycles of the spacecraft, and the cycles of it will be covered 0, 1 and 2 times.

The first three coverage criteria are related. The paths set satisfying transition coverage satisfies state coverage, and paths set satisfying basic path coverage satisfies transition coverage. The cycles in the statechart diagram of spacecraft need to be executed at most once according to the basic path coverage. If the cycles in the states need to be executed twice

while testing, the full ZOT coverage criterion needs to be considered.

Test paths searching methods based on the above coverage criteria were improved on graph search methods, such as DFS and BFS.

For the state coverage criterion, Breadth-First Search is used to generate a breadth-first tree without repeated states. The tree has $n$ leaf nodes, node For each leaf node in the tree, a path from the starting node to it can be obtained. However, since the test path obtained may not contain the ending node, the generated test path needs to be extended to be complete.

Particularly in the completing path process, the shortest path from a leaf node to the ending node is jointed to the path for each leaf node if it is not the ending node. Algorithm 1 was used to generate the test paths set, and the flowchart of the algorithm is shown in Fig. 7:

**Algorithm 1** Completing path process based BFS

**Input:** the directed graph $G$.
**Output:** the set of test path $PS$.

```
1: obtain the breadth-first tree for G, and define the set of leaf nodes as SOF
2: define the set of test paths as PS = {}
3: for ln ∈ SOL do
4:     obtain a path p (from starting node to ln)
5:     PS ← p
6: end for
7: repeat
8:     for ln ∈ SOL do
9:         if ln is not the ending node then
10:            DP = {}
11:            for cn ∈ the set of direct successors of ln do
12:                DP ← path(from cn to the ending node)
13:            end for
14:            find the shortest path sdp ∈ DP
15:            joint sdp to the path p(from the starting node to ln)
16:            update PS
17:        end if
18:    end for
19: until PS cannot be updated anymore
20: return PS
```

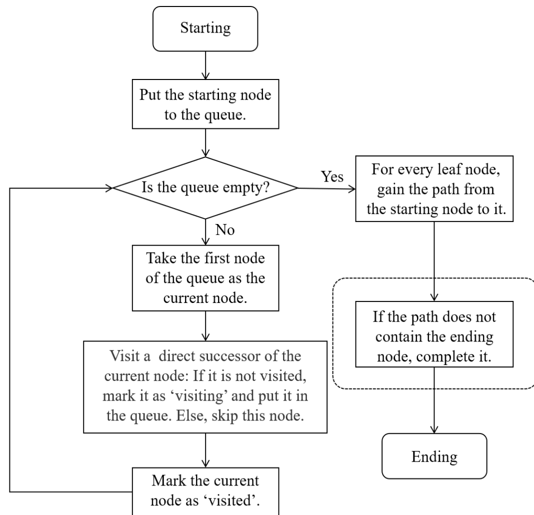


Fig. 7. Flow chart of breadth-first searching

For example, suppose that node $a$ is the starting point, and the node $e$ is the ending point in the directed graph transformed from the statechart diagram of the spacecraft, as shown in Fig. 8. When using the graph search algorithm, the node $c$ cannot be extended because its child node $d$ is already in the extended list, so node $c$ is a leaf node. In the end, two paths are obtained: $\{\langle a \rightarrow b \rightarrow c\rangle, \langle a \rightarrow b \rightarrow d \rightarrow e\rangle\}$ . This set covers all nodes (the state of the spacecraft), but its element does not conform to our definition of test path. It is necessary to complete the path $\langle a \rightarrow b \rightarrow c\rangle$ to $\langle a \rightarrow b \rightarrow c \rightarrow d \rightarrow e\rangle$ to get the desired paths set $\{\langle \rightarrow b \rightarrow c \rightarrow d \rightarrow e\rangle, \langle a \rightarrow b \rightarrow d \rightarrow e\rangle\}$.

The path search algorithm for the transition coverage criterion is similar to the above one.An algorithm was proposed which can generate base paths set with character that length of each path is the smallest by visiting control flow graph according to Depth-First Search method and flag set of edges and nodes [26], which was used in this article to generate basic paths set for statechart diagram.

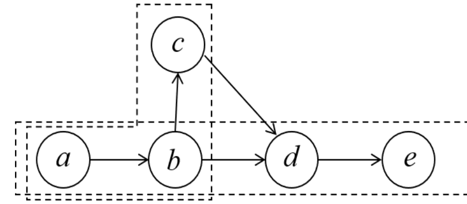


Fig. 8. A example for path searching

### D. Logical sequences generating

For the next step of the work, we first give the definition of a test case.

***Definition 4*:** A **Test Case** is defined as a tuple $tc = \langle input, output\rangle$. When a test input is given to the spacecraft under a specific working state for execution, the spacecraft should output accordingly.

From the definition of a test case, it is not difficult to know that for a test path of the system, one test case can be generated from each transition and the successor state of the transition. There are many such states and transitions on one test path, so plenty of test cases can be generated from each test path, as shown in Fig. 9.

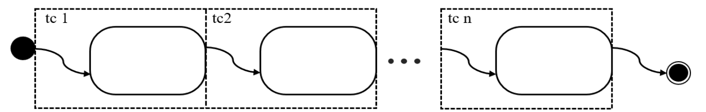


Fig. 9. Test cases on one test path

In the statechart diagram, events are the conditions that trigger the transition, but they are not equivalent to the inputs of the test. According to the modelling rules in this article, events consist of signal events and change events. Signal events come from outside the system, and they can be used as the system's input while testing. However, change events are

some of the conditions inside the system that can trigger the transferring, and these events should occur with the simulation goes but not be input by users or the environment. So whether change events occur can be used as criteria to judge whether the system is working correctly, that is to say, the test outputs.

The effects in transitions, the actions of states, and the change events constitute the spacecraft's test outputs, as shown in Fig. 10.

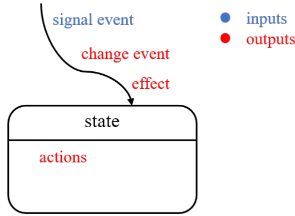


Fig. 10. The components of the inputs and the outputs

According to the definition, a test case of the spacecraft is dependent on its current state of it. A test case can be generated for each transition with the successor state of the transition. Many such states and transitions are on a test path, so many test cases can be generated on each test path. Test cases and test sequences are inextricably linked.

***Definition 5***: A **Test Sequence** is a sequence of the various test cases generated from each test path in order. A test sequence is defined as a list $ts = [tc_1, tc_2, tc_3, \ldots, tc_n]$, reviewing the example in Fig. 9.

***Definition 6***: A **Logical Sequence** is a sequence of logical conditions according to which the inputs of a test sequence are generated. A logical sequence is defined as a list $ls = [c_1, c_2, c_3, \ldots, c_n]$,where $c_i$ $(1 \leq i \leq n)$ is a logical condition referring to the parameters associated with the $i$-th transition of the test path.

A logical sequence is an abstract form of a test inputs sequence. In other words, a sequence of test inputs is the embodiment of a logical sequence. For example, assuming that a test path has two transitions $t_1$ and $t_2$, the signal event of $t_1$ is '$a > 1$', and that of $t_2$ is '$b < 20$'. $[a > 1, b < 20]$ is a logical sequence while $[a = 3, b = 12]$ is a test inputs sequence of it.

However, things are not as simple as the above example. The logical condition of a transition may involve more than one parameter. In order to cover the input domain adequately while testing, equivalence partitioning is needed for the input domain and then design different test inputs to cover them separately.

Considering the logical condition of the signal event on each transition, we divided it into disjoint logical conditions, which are also called sub-conditions in the following text.

The logical condition of the signal event on each transition is a predicate formula $F$. According to the Full Predicate Coverage Criterion [6] [27], each major predicate $P$ of $F$ is set to be $True$ and $False$, respectively (if the truth value of the predicate $P$ can determine the truth value of the predicate formula $F$, $P$ is called the major predicate of $F$).

This paper implemented this method of condition decomposition to decompose the logical condition into disjoint sub-conditions automatically, but this paper only saved the sub-conditions of which the truth values are True. For example, a logical condition is a predicate formula $(X \vee Y) \wedge Z$ and it can be automatically decomposed into disjoint sub-conditions (i) $X \wedge Y \wedge Z$, (ii) $\neg X \wedge Y \wedge Z$, (iii) $X \wedge \neg Y \wedge Z$.

Further, the combination among sub-conditions of every transition could be huge when generating a logical sequence along the test path. If there are $n$ transitions $[t_1, t_2, t_3, \ldots, t_n]$ along a test path and the signal event of transition $t_i$ can be decomposed $V_i$ sub-conditions $\langle c_1, c_2, c_3, \ldots, c_{V_i} \rangle$, there would be $V_1 \times V_2 \times V_3 \times \ldots \times V_n$ logical sequences generated by one test path. That is the combination explosion. The tree in Fig. 11 shows the logical relationship among test path, transitions and sub-conditions.

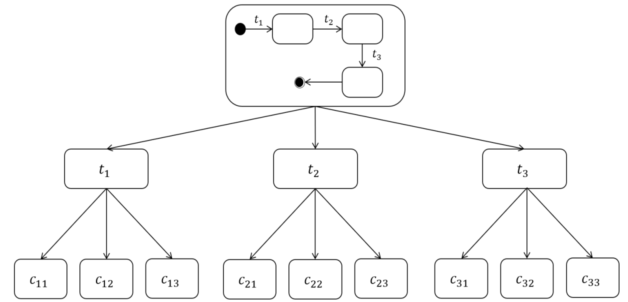


Fig. 11. Logical relationship among test path, transitions and sub-conditions

So the idea of combinatorial testing was used in this paper. What is different is that the regular combinatorial testing covers the combination of different values of different parameters in the system. However, the combinatorial test in this article is a combination of the logical conditions that trigger a series of transitions. This article used the 2-way coverage of the combinatorial test, which means that for any two signal events of the test path, the generated logical sequences can cover the combination of all the sub-conditions of them two.

For combinatorial testing, the most critical part is to generate the covering array. In this paper, we used the AETG algorithm, a 2-way combinatorial testing method. For example, shown in Fig. 11, the test path has 3 transitions, and the condition of trigger event on each transition is decomposed into 3 sub-conditions. The 2-way covering array of this example is shown in Table II. 10 combinations of sub-conditions can satisfy the 2-way coverage for 3 transitions, while $3 \times 3 \times 3 = 27$ combinations are needed if all combinations of sub-conditions are covered. The sequence of sub-conditions in each row of the table is a logical sequence.

The AETG algorithm is a heuristic method for generating a covering array [28]. Assume that we have a test path with $k$ transitions and that the $i$-th transition has $c_i$ different sub-conditions. Assume that we have already selected $r$ logical sequences. We select the $(r + 1)$-th logical sequence by first generating $M$ different candidate logical sequences and then

TABLE II
THE COVERING ARRAY OF THE EXAMPLE SHOWN IN FIG.11

| $t_1$ | $t_2$ | $t_3$ |
|---|---|---|
| $c_{11}$ | $c_{21}$ | $c_{32}$ |
| $c_{11}$ | $c_{22}$ | $c_{32}$ |
| $c_{11}$ | $c_{23}$ | $c_{31}$ |
| $c_{12}$ | $c_{21}$ | $c_{33}$ |
| $c_{12}$ | $c_{22}$ | $c_{31}$ |
| $c_{12}$ | $c_{23}$ | $c_{32}$ |
| $c_{13}$ | $c_{21}$ | $c_{31}$ |
| $c_{13}$ | $c_{22}$ | $c_{33}$ |
| $c_{13}$ | $c_{23}$ | $c_{32}$ |
| $c_{11}$ | $c_{23}$ | $c_{33}$ |

choosing one that covers the most new pairs. Each candidate logical sequence is selected by the following greedy algorithm:

1) Choose a transition $f$ and a sub-condition $l$ for $f$ such that $l$ appears in the greatest number of uncovered pairs.

2) Let $f_1 = f$. Then choose a random order for the remaining transitions. Then, we have an order for all $k$ transitions $f_1, f_2, \ldots, f_k$.

3) Assume that sub-conditions have been selected for transitions $f_1, \ldots, f_j$. For $1 \leq i \leq j$, let the selected sub-condition for $f_i$ be called $v_i$. Then, choose a sub-condition $v_{j+1}$ for $f_{j+1}$ as follows.

For each possible sub-condition $v$ for $f_{j+1}$, find the number of new pairs in the set of pairs $\{f_{j+1} = v$ and $f_i = v_i$ for $1 \leq i \leq j\}$. Then, let $v_{j+1}$ be one of the values that appeared in the greatest number of new pairs.

The logical sequences generated can cover all the combination of different logical sub-conditions for the signal events on any two transitions of the spacecraft.

### *E. Logical sequences generating*

Subsections $A$-$B$ described the methods of modelling spacecraft and preprocessing of the model, subsection $C$ illustrated the method of generating test paths from the model, and subsection $D$ described how to generate logical sequences from one test path. This section will show the method to generate test sequences from a logical sequence, the concretization method.

Lex is a lexical analyzer generator [29] and is commonly used with the yacc [30] parser generator. Ply is a parsing tool written purely in Python. PLY tool was used to analyze each condition on the logical sequence in this work. Combining the type and value range of the parameter in each sub-condition, we generated a specific value for each variable automatically using the program.

Then a test inputs sequence was generated from the logical sequence by generating specific values for every parameter in the logical sequence.A test input sequence is also called a **test scene** for the spacecraft.

Plenty of test scenes for the spacecraft can be generated from a logical sequence, and how these test scenes are distributed is also important.

According to ART, the generated test cases should be distributed evenly in the high-dimensional input domain to improve the coverage of defects as much as possible. For the spacecraft, the generated test scenes should be distributed as evenly as possible in the space formed by logical conditions.

Fixed Sized Candidate Set Art (FSCS-ART) algorithm is a classical distance-based ART (D-ART) algorithm [31].

The steps of the algorithm are as follows: Firstly, randomly generate a test case in the input domain, put it into the test case set $E$.

And then randomly generate a candidate set with a fixed number test case $C = \{c_1, c_2, \ldots, c_k\}$,

Finally, select a test case from candidate set $C$ which has the largest distance from set $E$, and add this test case into $E = \{e_1, e_2, \ldots, e_q\}$.

Repeat the above process for generating the following test case until enough test cases are generated in $E$.

The distance between the candidate test case $c_i (1 \leq i \leq k)$ and the set $E$ is defined as the minimum of distances between test case $e_j (1 \leq j \leq q)$ and $c_i$.

For spacecraft, each test scene can be viewed as a vector. Each component of the scene vector is a value of a variable. The distance vector of two test scenes is the difference between two scene vectors. Each component of the distance vector is the difference of two values of the variable in two test scenes. Because some variables of spacecraft are discrete values, while others are continuous, the definition of the difference of values was given.

For a discrete variable $a$, $\Delta a$ is the difference of its two values $a_1$ and $a_2$,

$$\Delta a = \begin{cases} 1, & a_1 \neq a_2 \\ 0, & a_1 = a_2 \end{cases}$$

A continuous variable $b$ has two values $b_1$ and $b_2$, and they need normalization in value before calculating distances.

$$b'_1 = \frac{b_1 - b_{min}}{b_{max} - b_{min}}, b'_2 = \frac{b_2 - b_{min}}{b_{max} - b_{min}}$$

where $b_{min}$ and $b_{max}$ are the minimum and maximum of variable $b$. $\Delta b$ is the difference of its two values $b_1$ and $b_2$.

$$\Delta b = b'_1 - b'_2$$

However, when testing spacecraft, empirical values of variables are sometimes important. For a given logical condition, an experienced tester could give a classical value $b_e$ that satisfies the logical condition. Often the values of the variables that trigger errors are more concentrated around the classical value.

So the goal is to generate test scenes that are as evenly distributed as possible, and for variables that have experience values provided by experts, we want the values to be more concentrated around the experience values. This variable was mapped again after normalization. Considering the sigmoid function

$$f(x) = \frac{1}{1 + e^{-Kx}}, K \geq 1$$

The $f(x)$ is a nonlinear function that maps the real numbers to the range $(0, 1)$. When $f(x_1), f(x_2), \ldots, f(x_n)$ are evenly

distributed in the range $(0,1)$, $x_1, x_2, \ldots, x_n$ are distributed more around the value 0. This phenomenon will be more obvious when $K$ increases.

For the spacecraft, the normalization of the empirical value is that

$$b'_e = \frac{b_e - b_{min}}{b_{max} - b_{min}}$$

Considering $g(x) = f(x - b'_e)$, the difference of its two values $b_1$ and $b_2$ was modified to $\Delta b = g(b'_2) - g(b'_1)$.

According to the above definition of the difference, the difference of different test scenes under the same logical sequence can be measured.

In a word, we non-linearly mapped the vectors representing the test scenes of the spacecraft to a 'twisted' space, and the distance between the test scenes on this twisted space was used to represent the difference of the test scenes. We used the ART algorithm based on this distance. Under the premise of satisfying the logical sequence, the set of generated test scenes can be more 'evenly' distributed in space, making it easier to find the errors of spacecraft software.

## IV. Case Study

Automatic tests generator that apply the above methods was implemented using a Python program. This section will show the practical result of the work and conduct a simple case study.

The results we generated were stored hierarchically in a XML document according to a certain structure. The first level is different test paths of the spacecraft. Under each test path, multiple logical sequences were stored, and under each logical sequence, multiple test scenes were stored. Each test scene consists of a series of test cases, and a test case contains an input and the expected output. Fig.12 shows a part of the XML document automatically generated by program.

Although Fig.12 only shows a part of the result, it contains a complete test sequence with 13 test cases.

Test cases need to be executed in order. If the input of a test case is empty, it means that there is no need to give any input for the spacecraft at this time, and the spacecraft will complete some actions autonomously. When the test case's input is fed to the spacecraft controller software, we should compare the actual output of the spacecraft with the expected output automatically generated. If the two are consistent, the testing passes on this test case, and the next test case will be executed. In a test sequence, if there is any inconsistency between the expected output and the actual output, the test fails.


```
<Route1 type="dict">
 - <Logical_sequence1 type="dict">
   - <test_sequence1 type="dict">
     + <init_vars type="list">
     - <test_case1 type="dict">
         <Input type="str"/>
         <Output type="str">AOCCforYG.AOCC.mode_feedback=1; AOCCforYG.AOCC.method_
       </test_case1>
     - <test_case2 type="dict">
         <Input type="str">after t=0.71 s: AOCCforYG.AOCC.mode=1 </Input>
         <Output type="str">AOCCforYG.AOCC.mode_feedback=2; AOCCforYG.AOCC.method_
       </test_case2>
     - <test_case3 type="dict">
         <Input type="str"/>
         <Output type="str">before t=100 s: when (delt t=10 s: Satllite_linux64.attitude_ang
           AOCCforYG.AOCC.mode_feedback=2; AOCCforYG.AOCC.method_feedback=21</(
       </test_case3>
     - <test_case4 type="dict">
         <Input type="str"/>
         <Output type="str">before t=100 s: when (delt t=10 s: AOCCforYG.AOCC.SA_unfold_
       </test_case4>
     - <test_case5 type="dict">
         <Input type="str"/>
         <Output type="str">before t=100 s: when (delt t=10 s: Satllite_linux64.attitude_ang
       </test_case5>
     - <test_case6 type="dict">
         <Input type="str"/>
         <Output type="str">before t=100 s: when (DSS1_appear=1 | DSS2_appear=1); mod
       </test_case6>
     - <test_case7 type="dict">
         <Input type="str"/>
         <Output type="str">before t=100 s: when (delt t=10 s: (AOCCforYG.AOCC.DSS1_m<(
           AOCCforYG.AOCC.method_feedback=40</Output>
       </test_case7>
     - <test_case8 type="dict">
         <Input type="str">after t=6.624 s: AOCCforYG.AOCC.method=31 </Input>
         <Output type="str">AOCCforYG.AOCC.ES_switch=1; AOCCforYG.AOCC.mode_feedbac
       </test_case8>
     - <test_case9 type="dict">
         <Input type="str"/>
         <Output type="str">before t=100 s: when (UserDef∈[-0.008, 0.008] rad/s & Satllite_
           AOCCforYG.AOCC.method_feedback=41</Output>
       </test_case9>
     - <test_case10 type="dict">
         <Input type="str"/>
         <Output type="str">before t=100 s: when (AOCCforYG.AOCC.ES_appear=1); AOCCfor
       </test_case10>
     - <test_case11 type="dict">
         <Input type="str">after t=0.295 s: AOCCforYG.AOCC.mode=5 </Input>
         <Output type="str">AOCCforYG.AOCC.mode_feedback=5</Output>
       </test_case11>
     - <test_case12 type="dict">
         <Input type="str">after t=4.266 s: AOCCforYG.AOCC.method=51 </Input>
         <Output type="str">AOCCforYG.AOCC.method_feedback=50; AOCCforYG.AOCC.mode_
       </test_case12>
     - <test_case13 type="dict">
         <Input type="str"/>
         <Output type="str"/>
       </test_case13>
   </test_sequence1>
```


Fig. 12. A part of the result

## V. Contributions

The work of this paper is based on industrial practice, and the paper presents an automatic process to generate test cases for the controller software of spacecraft. The test cases are generated from the model of testing requirements.

The contributions of this paper are as follows:

1) An automatic process to generate test cases for spacecraft is proposed and implemented. It is a model-based method combining the combinatorial testing and adaptive random testing.
2) A metric for the difference of test scenes is proposed. According to the metric, an FSCS-ART method based on non-uniform distance is proposed to generate test scenes which are easier to cover errors of the spacecraft.

## VI. Future Work

The main purpose of the work in this paper is to generate test scenes for spacecraft controller software that fully cover the test requirements, that is, to generate test cases for the spacecraft controller software offline when only the test requirements are known.

The disadvantage is that test scenes have not been run to verify their validity. Soon, we will continue to study the

validity of test cases and use the dynamic random testing [32] method to select and execute test cases online.

## VII. Acknowledgement

This work was supported by the National Natural Science Foundation of China under Grant 61772055 and Grant 61872169.

## References

[1] S. R. Dalal, A. Jain, N. Karunanithi, J. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz, "Model-based testing in practice," in *Proceedings of the 21st international conference on Software engineering*, 1999, pp. 285–294.

[2] A. C. Dias Neto, R. Subramanyan, M. Vieira, and G. H. Travassos, "A survey on model-based testing approaches: a systematic review," in *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*, 2007, pp. 31–36.

[3] S. K. Swain, D. P. Mohapatra, and R. Mall, "Test case generation based on state and activity models." *J. Object Technol.*, vol. 9, no. 5, pp. 1–27, 2010.

[4] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Computing Surveys (CSUR)*, vol. 43, no. 2, pp. 1–29, 2011.

[5] I. K. Mak, "On the effectiveness of random testing," Ph.D. dissertation, University of Melbourne, Faculty of Science, 1997.

[6] J. Offutt and A. Abdurazik, "Generating tests from uml specifications," in *International Conference on the Unified Modeling Language*. Springer, 1999, pp. 416–429.

[7] D. Richardson, O. O'Malley, and C. Tittle, "Approaches to specification-based testing," in *Proceedings of the ACM SIGSOFT'89 third symposium on Software testing, analysis, and verification*, 1989, pp. 86–96.

[8] A. Podgurski and L. A. Clarke, "A formal model of program dependences and its implications for software testing, debugging, and maintenance," *IEEE Transactions on software Engineering*, vol. 16, no. 9, pp. 965–979, 1990.

[9] H. Zhu, P. A. Hall, and J. H. May, "Software unit test coverage and adequacy," *Acm computing surveys (csur)*, vol. 29, no. 4, pp. 366–427, 1997.

[10] N. L. Hashim and Y. S. Dawood, "A review on test case generation methods using uml statechart," in *2019 4th International Conference and Workshops on Recent Advances and Innovations in Engineering (ICRAIE)*. IEEE, 2019, pp. 1–5.

[11] Y. G. Kim, H. S. Hong, D.-H. Bae, and S. D. Cha, "Test cases generation from uml state diagrams," *IEE Proceedings-Software*, vol. 146, no. 4, pp. 187–192, 1999.

[12] J. Hartmann, C. Imoberdorf, and M. Meisinger, "Uml-based integration testing," in *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, 2000, pp. 60–70.

[13] S. Kansomkeat and W. Rivepiboon, "Automated-generating test case using uml statechart diagrams," in *Proceedings of the 2003 annual research conference of the South African institute of computer scientists and information technologists on Enablement through technology*, 2003, pp. 296–300.

[14] R. K. Swain, P. K. Behera, and D. P. Mohapatra, "Minimal testcase generation for object-oriented software with state charts," *International Journal of Software Engineering & Applications (IJSEA)*, vol. 3, no. 4, 2012.

[15] M. Shirole, A. Suthar, and R. Kumar, "Generation of improved test cases from uml state diagram using genetic algorithm," in *Proceedings of the 4th India Software Engineering Conference*, 2011, pp. 125–134.

[16] R. Swain, V. Panthi, P. K. Behera, and D. P. Mohapatra, "Automatic test case generation from uml state chart diagram," *International Journal of Computer Applications*, vol. 42, no. 7, pp. 26–36, 2012.

[17] M. A. Ali, K. Shaik, and S. Kumar, "Test case generation using uml state diagram and ocl expression," *International Journal of Computer Applications*, vol. 95, no. 12, 2014.

[18] W. Linzhang, Y. Jiesong, Y. Xiaofeng, H. Jun, L. Xuandong, and Z. Guoliang, "Generating test cases from uml activity diagram based on gray-box method," in *11th Asia-Pacific software engineering conference*. IEEE, 2004, pp. 284–291.

[19] Y. D. Salman and N. L. Hashim, "An improved method of obtaining basic path testing for test case based on uml state chart," *Science International*, vol. 26, no. 4, pp. 1–8, 2014.

[20] 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.

[21] L. J. White and E. I. Cohen, "A domain strategy for computer program testing," *IEEE transactions on software engineering*, no. 3, pp. 247–257, 1980.

[22] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. Tse, "Adaptive random testing: The art of test case diversity," *Journal of Systems and Software*, vol. 83, no. 1, pp. 60–66, 2010.

[23] L. Wood, A. Le Hors, V. Apparao, S. Byrne, M. Champion, S. Isaacs, I. Jacobs, G. Nicol, J. Robie, R. Sutor *et al.*, "Document object model (dom) level 1 specification," *W3C recommendation*, vol. 1, 1998.

[24] Y. Chen, N. Sang, and H. Lei, "An improved efsm model for class test," *Journal of Computer Applications*, vol. 25, no. 8, pp. 1890–1892, 2005.

[25] X.-d. ZHAN and H.-k. MIAO, "Test framework of object-oriented software based on uml statecharts [j]," *Journal of Applied Sciences*, vol. 5, 2006.

[26] Q.-j. WU, X.-h. YANG, J.-c. LU, and T.-l. YU, "An optimized algorithm of auto-generate base paths set based on depth-first search [j]," *Journal of University of South China (Science and Technology)*, vol. 3, 2012.

[27] M. Huai-kou and F. Li-zhi, "Axiomatic assessment of uml statecharts-based test adequacy criteria [j]," *Journal of Shanghai University (Natural Science Edition)*, vol. 5, 2007.

[28] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The aetg system: An approach to testing based on combinatorial design," *IEEE Transactions on Software Engineering*, vol. 23, no. 7, pp. 437–444, 1997.

[29] M. E. Lesk and E. Schmidt, "Lex: A lexical analyzer generator," 1975.

[30] S. C. Johnson *et al.*, *Yacc: Yet another compiler-compiler*. Bell Laboratories Murray Hill, NJ, 1975, vol. 32.

[31] T. Y. Chen, T. Tse, and Y.-T. Yu, "Proportional sampling strategy: A compendium and some insights," *Journal of Systems and Software*, vol. 58, no. 1, pp. 65–81, 2001.

[32] K.-Y. Cai, H. Hu, C.-H. Jiang, and F. Ye, "Random testing with dynamically updated test profile," in *Proceedings of the 20th International Symposium On Software Reliability Engineering (ISSRE 2009)*, 2009, pp. 1–2.