TBFV-M: TESTING-BASED FORMAL VERIFICATION FOR SYSML ACTIVITY DIAGRAMS

Yufei Yin[*,+] Shaoying Liu[*] and Yixiang Chen[+]

[*]Hosei University, Tokyo, Japan
[+]East China Normal University, Shanghai, China

ABSTRACT

SysML activity diagrams are often used as models for software systems and its correctness is likely to significantly affect the reliability of the implementation. However, how to effectively verify the correctness of SysML diagrams still remains a challenge. In this paper, we propose a testing-based formal verification (TBFV) approach to the verification of SysML diagrams, called TBFV-M, by creatively applying the existing TBFV approach for code verification. We describe the principle of TBFV-M and present a case study to demonstrate its feasibility and usability. Finally, we conclude the paper and point out future research directions.

KEYWORDS

SysML activity diagrams, TBFV, test path generation, formal verification of SysML diagram

1. INTRODUCTION

Model-Based Systems Engineering (MBSE) [1] is often applied to develop large scale software systems in order to effectively ensure their reliability and to reduce the cost for testing and verification. The systems modelling language SysML [2, 3] can support effective use of MBSE due to its well-designed mechanism for creating object-oriented models that incorporate not only software, but also people, material and other physical resources. In MBSE, SysML models are often used as the design for code. Therefore, its correctness in terms of meeting the users' requirements becomes critical to ensure the high reliability of the code. Unfortunately, to the best of our knowledge from the literature, there are few tools to support the verification of SysML models [4, 5] in particular rigorous ways of verification.

Testing-Based Formal Verification (TBFV) proposed by Liu [6-8] shows a rigorous, systematic, and effective technique for the verification and validation of code. Its primary characteristic is the integration of the specification-based testing approach and Hoare logic for correctness proof of code to guarantee the correctness of all the traversed program paths during testing. The advantage of TBFV is its potential and capability of achieving full automation for verification through testing. However, the current TBFV is mainly designed for sequential code in which all of the details are formally expressed, and there is no research on applying it to verify SysML models yet. In this paper, we discuss how the existing TBFV can be applied to SysML models for their verification and we use TBFV-M (testing-based formal verification for models) to represent the newly developed approach. Since SysML Activity Diagrams can model the systems dynamic behaviour and describe complex control and parallel activities, which are similar to code but with additional constructs such as parallel execution, our discussion in this paper focuses on the activity diagrams.

The essential idea of TBFV-M is as follows. All of the functional scenarios are first extracted from a given formal specification defining the users' requirements where each functional scenario defines a meaningful functional behaviour of the system. Meanwhile, test paths are generated from corresponding SysML Activity Diagrams waiting to be verified. Then, test paths are matched with functional scenarios by comparing the collection of decision condition of each test path and the guard condition of the functional scenario. After this, the pre-condition of the test path is automatically derived by applying the assignment axiom in Hoare logic based on the functional scenario. Finally, the implication of the pre-condition of the specification in conjunction with the guard condition of the functional scenario to the derived pre-condition of the path is verified through automatic proof or testing to determine whether the path contains bugs. The details of this approach will be discussed from Section 5.

The remainder of the article will detail the TBFV-M method. Section 2 presents related work we have referenced. Section 3 introduces the Testing-Based Formal Verification technique for the verification and validation of code. Section 4 mainly details the whole development process of using Model-Based Systems Engineering and the application scenarios of TBFV-M method. Section 5 characterizes the definitions of basic terms and concepts and section 6 describes the principle of TBFV-M, showing the core technology of TBFV-M. Section 7 uses one case study to present the key point of TBFV-M. Finally. The details of the implementation of the algorithm are presented in Section 8. Section 9 concludes the paper.

2. RELATED WORK

In this section, we briefly review the existing work related to our study. For the sake of space, we focus on those we have referenced during our research. We divide the related work into four different parts, including testing-based verification, requirements verification, verification using Hoare Logic and test case generation.

Considering the shortcoming of formal verification based on Hoare logic being hard to automate, Liu proposed the TBFV (Testing-Based Formal Verification) method by combining specification-based testing with formal verification [6]. This method not only take the advantage of full automation for testing, but also the efficiency of error detection with formal verification. Liu also designed a group of algorithms [9] for test cases generation from formal specification written in SOFL [10]. A supporting tool [8] is also developed. These efforts have significantly improved the applicability of formal verification in industrial settings.

Franco Raimondi [11] addressed the problem of verifying planning domains written in the Planning Domain Definition Language (PDDL). First, he translated test cases into planning goals, then verified planning domains using the planner. A tool PDVer is also generated. In this paper, testing is also used during verification and the effectiveness and the usability is improved.

Stefano Marrone [12, 13] designed a Model-Driven Engineering approach, in which formal models are constructed and test cases are generated from UML model, utilizing UML profiles and model transformation algorithms, automatically. As they claimed, formal models can be used for quantitative analysis of non-functional properties, while test cases can be used for model checking. A railway signalling example is shown to introduce its integration, usability and reduction of manual activities.

Feng Liang [14] proposed a vVDR (Virtual Verification of Designs against Requirements) approach for verifying a system with its requirement. In his research, the system is modeled in Modelica, and requirement verification scenarios are specified in ModelicaML, an UML profile and a language extension for Modelica. vVDR approach guarantees that all requirements can be verified by running this scenario automatically. However, the deficiency appears when the number of requirements and scenarios increase.

Ralf Sasse [15] designed a tool called Java+ITP to verify a subset of the Java language. During the verification process, Maude-based continuation passing style (CPS) is used to rewrite the logical semantics of Java, and they also developed CPSbased Hoare Logic rules to justify the correctness of the rewritten fragment. Ralf Sassi's tool provides an extensibility of Hoare logics, but exceptions and objects should be considered in the future research.

Magnus O. Myreen [16] used Hoare Logic to deal with machine code. They designed a mechanized Hoare-style programming logic framework to accommodate the restrictions and features present in real machine-code, such as finite memory, data and code in the same memory space. ARM machine-code now can be verified using the proposed logic.

Jonathan Lasalle [17] utilized the existing UML/OCL Model-Based Test generation tool, Smartesting Test DesignetTM. He designed rewriting rules to translate a SysML model into an equivalent UML model. The advantage of this process is that we can use the existing UML tools to handle the SysML model.

Ashalatha Nayak [18] introduced an approach to transform the particular Activity Diagram into a model that can be used for testing, called ITM, based on its structure characteristics. The advantage of using ITM is that it can simplify the process of extracting and analyzing test scenarios based on the coverage criteria. However, it also has limitations on processing unstructured Activity Diagram because the unstructured Activity Diagrams shape is out of structure.

Oluwatolani Oluwagbemi [19] proposed a new concept called activity ow tree (AFT) and it can store the information obtained by traversing the activity diagram. Then, AFT is used as an intermediate expression to generate test cases automatically. They designed the transformation and generation algorithm and compared their achievement with the work done by the predecessors.

Inspired by Liu's work, we apply and extend the TBFV approach to models and propose the TBFV-M. A model is more intuitive than a formal specification because it requires less relevant background knowledge and is easier to communicate with customers. TBFV approach shows the treatment of code, while TBFV-M approach deals with SysML Activity Diagrams. And different with Feng Liang's work, TBFV-M approach do not use other supporting tools, like Modelica, we merely use Hoare Logic to do the verification. Referring to test case generation, TBFV- M approach can deal with unstructured diagrams, which may have stronger processing power than existing approaches.

3. INTRODUCTION OF TBFV FOR CODE

TBFV is a novel technique that makes good use of Hoare logic to strengthen testing. The essential idea is first to use specification-based testing to discover all traversed program paths and then to use Hoare logic to prove their correctness. During the proof process, all errors on the paths can be detected.

Testing is a practical technique for detecting program errors. A strong point of testing superior to formal correctness verification is that it is much easier to be performed automatically if formal specifications are adopted [20], but a weak point is that existing errors on a program path may still not be uncovered even if it has been traversed using a test case. TBFV takes advantage of testing, realized full automation for error detection efficiency.

TBFV is a specific specification-based testing approach that takes both the precondition and post-condition into account in test case generation [21]. It treats a specification as a disjunction of functional scenarios (FS), and to generate test sets and analyse test results based on the functional scenarios. A functional scenario is a logical expression that tells clearly what condition is used to constrain the output when the input satisfies some condition. To precisely describe this strategy, we first need to introduce functional scenario. S_{pre} and S_{post} denote the pre- and post-conditions of operation S. Let:

$$S_{post} = (G_1 \wedge D_1) \vee (G_2 \wedge D_2) \vee \ldots \vee (G_n \wedge D_n) \tag{1}$$

G_i and D_i ($i \in 1, \ldots, n$) are two predicates, called guard condition and defining condition, respectively. The definition of functional scenarios and FSF (functional scenario form) are list below:

$$\text{Functional Scenario} = S_{pre} \wedge G_i \wedge D_i \tag{2}$$

In the definition of functional scenario, $S_{pre} \wedge G_i \wedge D_i$ is treated as a scenario: when $S_{pre} \wedge G_i$ is satisfied by the initial state (or intuitively by the input variables), the final state (or the output variables) is defined by the defining condition D_i. The conjunction $S_{pre} \wedge G_i$ is known as the test condition of the scenario, which serves as the basis for test case generation from this scenario.

$$FSF = (S_{pre} \wedge G_1 \wedge D_1) \vee (S_{pre} \wedge G_2 \wedge D_2) \vee \ldots \vee (S_{pre} \wedge G_n \wedge D_n) \tag{3}$$

A systematic transformation procedure, algorithm, and software tool support for deriving an FSF from a pre-post style specification written in SOFL have been developed in our previous work [22]. In the case study section, we will show an example to detail FSF generation. Test cases can be generated from FSF. TBFV has three main techniques. First, generate test cases from specification. Second, form path triple and the definition are below:

$$\{S_{pre} \wedge G_i\} P \{ D_i \} \tag{4}$$

P is called a program segment, which consists of decision (i.e., a predicate), an assignment, a return statement, or a printing statement. It means that if the pre-condition S_{pre} and the guard condition G_i of the program are both true before path P is executed, the post-condition D_i of path P will be true on its termination.

Finally, repeatedly apply the axiom for assignment to derive a pre-assertion, denoted by P_{pre}. And the correctness of the specific path is transformed into the implication $S_{pre} \wedge G_i \rightarrow S_{pre}$. If the implication can be proved, it means that no error exists on the path; otherwise, it indicates the existence of some error on the path.

4. TBFV-M IN MBSE

Model-Based Systems Engineering (MBSE) combines process and analysis with architecture. In the last decade, the model-driven approach for software development has gained a growing interest of both industry and research communities as it promises easy automation and reduced time to market [23]. Because of the graphical notation for defining system design as nodes and edge diagrams, SysML model addresses the ease of adoption amongst engineers [24].

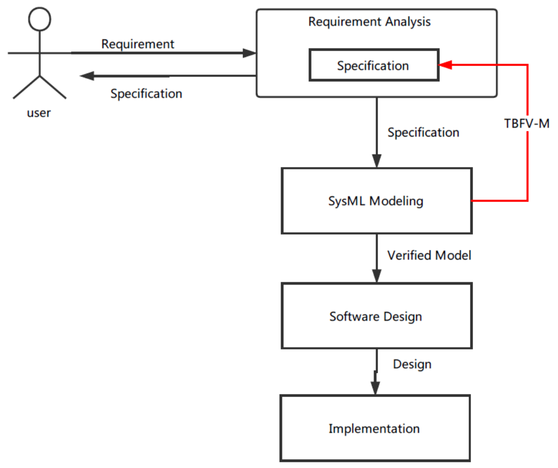


Figure 1. TBFV-M usage scenario

In the development process using MBSE, as shown below, the users' requirements are obtained first and the requirement document is usually written in natural language. This document is not only difficult for developers to understand, but also may contain ambiguities and other problems.

To obtain requirements without ambiguities, we may generate a SysML Model. During the model-driven development process, we use the SysML Model Diagram to communicate with the user, because it does not contain many mathematical symbols and syntax. In the SysML modelling phase, we will refine the SysML diagram, involved in the SysML model and specification.

During the Model-Driven process, model is an important medium for the Model based system engineering development. The TBFV-M method is mainly used to verify whether SysML Activity Diagram model meets the user's requirements written in SOFL (Structured-Object-oriented-Formal Language).

5. Basic Concept

5.1. Formal Definition of Activity Diagram

Activity Diagram Formal Definition [2] can be represented as:

$$AD = (Node; Edge) \tag{5}$$

Node is a set of nodes of which definition as follow:

$$\begin{gathered}Node = \{InitialNode; FlowFinalNode; ActivityFinalNode; ActionNode; \\ ActivityNode; \quad ForkNode; JoinNode; DecisionNode; MergeNode; \\ RecieveSignalNode; SendSignalNode\}\end{gathered} \tag{6}$$

Edges defines the relationship between nodes such that:

$$Edge = \{(x, y) | x, y \in Node\} \tag{7}$$

There are two types of edges: control flow and object flow. Control flow edges represent the process of executing token passing in AD and object flow edges are used to show the flow of data between the activities in AD.

5.2. Test Case

From a global view, test case based on the SysML activity diagram consists of test path and test data. And the definition is as followed:

$$\mathrm{TC(AD)} = (\mathrm{Path}; \mathrm{Data}) \tag{8}$$

For activity diagram, test scenario consists of a series of actions and edges in the diagram. Based on the formal definition of the activity diagram given above, the test path is defined as follow:

$$\mathrm{path} = (a'_1, a'_2, \ldots, a'_n) \tag{9}$$

$$a'_i = (t_n, a_n), (i = 2, \ldots, n) \tag{11}$$

$$t_n = a_{i-1} \rightarrow a_i, (i = 2, \ldots, n) \tag{12}$$

In this formula, a_i means node, t_i means edge.In this case, a test path is a set of nodes, starting from node a_1 and ending with node a_n through the transition edges $t_2 \ldots t_n$. For activity diagram, a_1 and a_n represent the initial node and final node, respectively. Test data indicates the input information corresponding to a particular test scenario including various types of data, even user actions and so on.

5.3. Test Coverage Criteria

For software, the adequacy measurement of testing is reflected in the rate of coverage and effectiveness of the test case. The test coverage criteria in white box tests includes statement coverage, branch coverage, conditional coverage and so on. These coverage criteria ensure the sufficiency of testing and provide implications for the test case generation algorithm. Here are four test coverage criteria used in our design, for test case generation of SysML activity diagram [19,25,26]:

- **Action coverage criteria:** In software testing process, testers are often required to generate test cases to execute every action in the program at least once.
- **Edge coverage criteria:** In software testing process, testers are often required to generate test cases to pass every edge in the program at least once.
- **Path coverage criteria:** These coverage criteria require that all the execution paths from the programs entry to its exit are executed during testing.
- **Branch coverage criteria:** These coverage criteria generate test cases from each reachable decision made true by some actions and false by others.

5.4. Hoare Logic

Hoare Logic is a formal system developed by C. A. R. Hoare [27, 28], and it is designed for the proof of partial correctness of a program. In Hoare Logic, the Hoare Triple [29] is best known and is also referenced in our method. The Hoare triple is of this form:

$$\{P\}\ C\ \{Q\} \tag{13}$$

, where P and Q are assertions and C is a command. P is named the pre-condition, which is a predicate expression describing the initial states and Q the post-condition, which is also a predicate expression describing the final states.

Hoare also established necessary axioms to define the semantics of each program construct, including axiom of assignment, rules of consequence, axioms of composition, axioms of alternation, iteration and block. Axiom of assignment is used in our work, so we will briefly introduce it:

$$\{Q(E\backslash x)\}\ x{:=}E\ \{Q\} \tag{14}$$

,where x is a variable identifier, E is an expression of a programming language without side effects, but possibly containing x, Q[E\x] is a predicate resulting from Q by substituting E for all occurrences of x in Q. This axiom means that to verify the correctness of the assignment, the postcondition Q should be satisfied. This equals to Q[E\x] is true because x is assigned by representing E after the execution.

6. Procedure of TBFV-M

The TBFV-M method takes the specification describing the users' requirements and the SysML Activity Diagram model as input and verifies the correctness of the SysML model with respect to the specification. The procedure of TBFV-M is illustrated in Figure2.

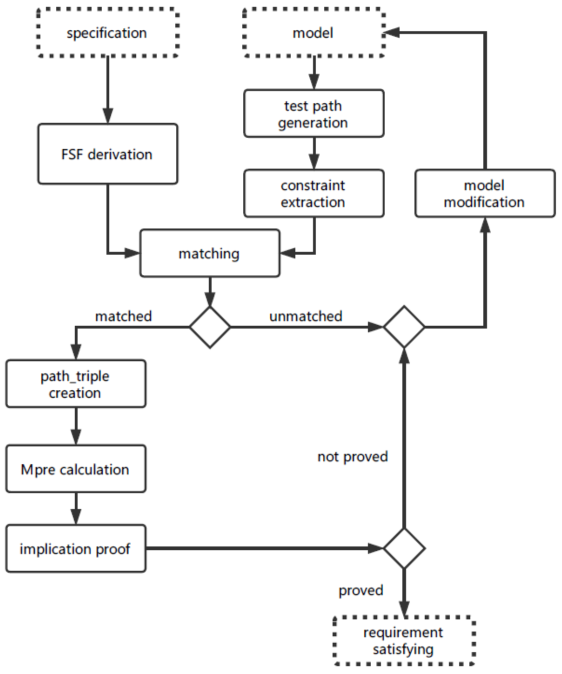


Figure 2. TBFV-M processing procedure

From this figure, we find that functional scenarios are derived from the specification written in the pre- / post-condition style, while test paths are generated from the Activity Diagram and the data constraints can be extracted from each test path. Then, the extracted data constraints are used to match with functional scenarios. A matching algorithm is defined by us. We will verify the successful matched the test path according to the requirements represented in specification. The verification part can be separated into three parts: first, create a path triple, and then use the axiom of Hoare Logic to derive pre-assertion for each test path. Finally, prove the implication of the pre-condition in the specification and pre-assertion. If we can prove all the implication of pre-assertion of all the test paths of the model and the matching pre-condition, then the model is to meet the requirements.

These critical steps in the TBFV-M method, including functional scenarios derivation, test path generation, matching algorithm, pre-assertion derivation and implication will be discussed next and comprehensive details will be described in the case study section.

6.1. Unified Formal Expression

Using a unified formal expression can reduce the ambiguity between communications, We establish the unified formal expression, including specification guide and modeling guide. Specification reflects complete requirements and we chose SOFL to describe formal specification. An example specification written in SOFL is given below. It describes that if a person is smaller than 6, he will be free; otherwise, he should buy the normal price for $10.

```
process buy_ticket (age: int) price: int
pre: age>0
post: age<=6 and price = 0
      or
      age>6 and price = 10
end_process
```

6.2. Functional Scenarios Derivation

The overall goal of functional scenario derivation is to extract all functional scenarios completely in "$S_{pre} \wedge G_i \wedge D_i$" form (FSF), as mentioned above in TBFV section. Because this part is not our main topic and has been researched before. In our work, we assume that an FSF of the specification has been available somehow. The below segment of the process buy ticket, mentioned previously, shows the FSF generated from the specification described in the last one.

```
  1  buy_ticket_pre: age>0
G1: age<=6
D1: price = 0

  2  buy_ticket_pre: age>0
G2: age>6
D2: price = 10

   3   ^buy_ticket_pre: age<=0
```

6.3. Test Paths Generation

A test path auto-generation tool based on the SysML Activity Diagram model takes the model as input and generates test cases as outputs automatically. Our SysML Activity Diagram test path generation includes three parts. First, we use transformation algorithm to compress the input Activity Diagram, which may contain unstructured module. The transformation is a cyclic process, dealing with loop module, concurrent module and the problem of multiple starting nodes separately. After compressing, we transform this unstructured activity diagram into an intermediate representation form Intermediate Black box Model (IBM). IBM consists of one basic module and a map from black box to the corresponding original actions. The third phase of our approach is test path generation based on IBM. In this phase, two problems should be solved, which are basic module test path generation and black box test path generation. Details of automated test paths generation algorithm and implementation of unstructured SysML Activity Diagram has been developed in our previous work [30].

6.3.1. Loop Module

The Loop module in the SysML activity diagram can be considered as a node collection, and these nodes in the collection can be cycled multiple times. As shown in Figure3, according to the location of cyclic judgment condition located at the end of the loop module or the front, we can divide the loop module into do-while loop and while-do loop.

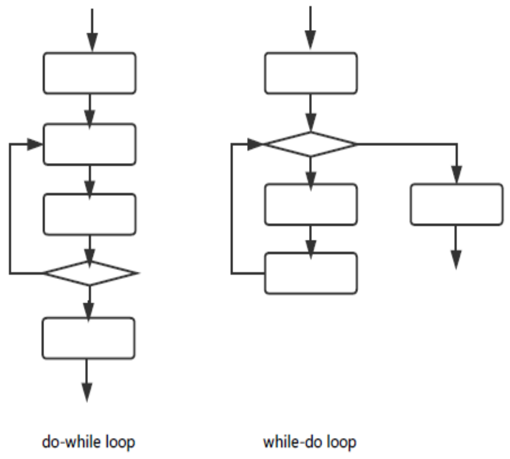


Figure 3. Classification of loop modules

The first step in the transformation algorithm of the Loop module is to identify the loop module, the second step is to compress it into a black box node loop, and finally reinsert it into the original SysML activity diagram. Figure4 shows the process.

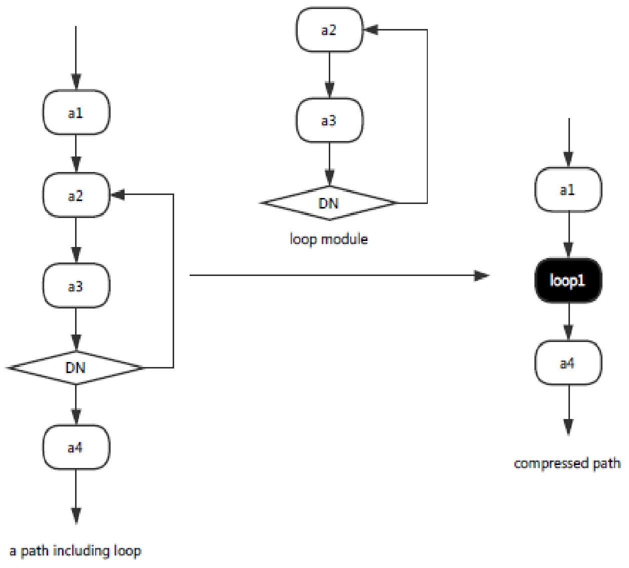


Figure 4. The transformation of loop module

Since the infinite traversal loop is not possible, it is possible to propose a different expansion algorithm for different types of loops when processing the loop module. For simple loop, you can take the following test case sets (where n is the maximum number of passes allowed):

- skip the entire loop
- go through the loop once
- go through the loop twice
- go through the loop m times
- go through the loop n-1, n, n + 1 times

6.3.2. Concurrent Module

In the SysML activity diagram, the most common form of a concurrent module is a pair of fork node and join node and all actions between these two nodes, as shown in Figure5 (a). The fork node can be represented as simultaneous start of multiple parallel streams and the join node represents the possible synchronization of multiple parallel streams, inflowing into next action. The logical representation is AND. However, the synchronization stream can also be the logical relationship OR, as shown in Figure5 (b).

Depending on how many concurrent streams can be synchronized by the join node, the parallel modules can be divided into partJoin concurrent and noJoin concurrent, as shown in Figure5 (c) and (d) below, respectively.

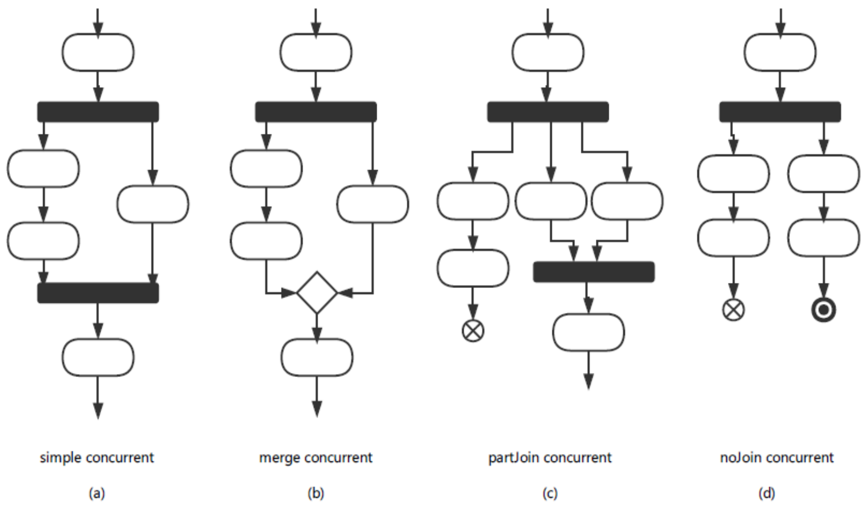


Figure 5. Classification of concurrent modules

On the test path generation algorithm for concurrent modules, the first step is to identify the concurrency module, the second step is to compress it into a black box node FJ (Fork-Join), and finally reinsert it into the original SysML activity diagram, as shown in the following Figure6.

For concurrent modules, we can use the Concurrent module path generation algorithm and generate the test path automatically. For the compressed basic path, the test path generation algorithm of the basic module can be applied. Once the basic path is generated, replace the FJ black box with the test path generated from the concurrency module.

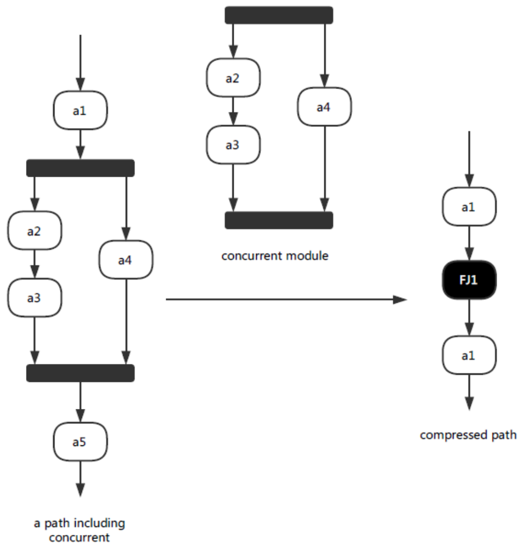


Figure 6. The process of transformation of concurrent modules

6.3.3. Test path generation with IBM

The test case generation with IBM needs to deal with three types of modules, which are basic modules, concurrent modules and loop modules. The basic module is the activity diagram that compresses the concurrent module and the loop module into the black box node respectively. The concurrent module and the Loop module are the transformed black box module which contain the unique incoming edge and the unique outgoing edge.

For basic module, without considering the concurrent module and the loop module, we can transform the SysML activity diagram model into a directed acyclic graph, using the idea of DFS (Depth First Search) algorithm. While for unconstructed module, we can use corresponding generation algorithm and generate the test path automatically. After the basic path is created, the black box can be replaced with the test path generated by the unconstructed module.

6.3.4. Motivation Example

Figure7 is an unstructured SysML activity diagram model, which contains a concurrency module and a loop module.

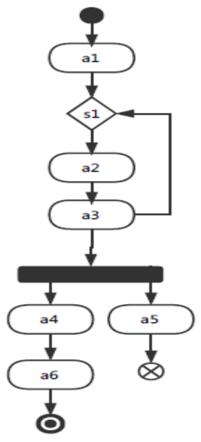


Figure 7. Motivating case

Figure8 shows how to compress an unstructured activity diagram and transform the unstructured module into a black box node. Eventually the unstructured activity diagram converts into an intermediate representation of IBM. The first step is to identify the loop module and compress it into a black box node while-do loop1, shown in Figure8(a). The compressed black box node is the intermediate representation of the loop shown in the following Figure9(a). The second step is to identify the noJoin concurrency module and compress it into a black box node No FJ1, shown in Figure8(b). The compressed black box node is shown in the following Figure9(b).

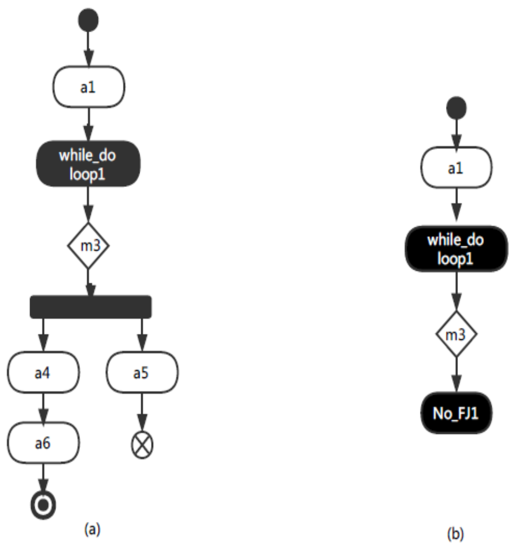


Figure 8. The process of transformation

Figure8(b) is a compressed and structured SysML activity diagram that can be used to automatically generate test cases. Finally, the black box module can be replaced.

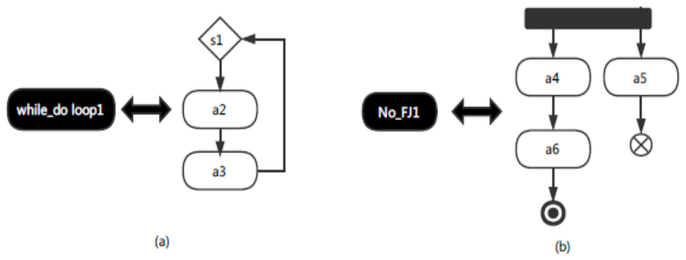


Figure 9. The map of black boxes

6.4. Matching Algorithm

Matching the test path with functional scenario is very important for verification. In order to verify the correctness of one path in Activity Diagram, we need to match it with corresponding functional scenario. The constraints of test path can be extracted from edges of each path, which are used to compare with $S_{pre} \wedge G_i$ part of functional scenario. If unmatched test paths or functional scenarios appears, it means some errors may be existed in this model. And the model needs to be modified. The matching algorithm is given below.

Matching algorithm takes the edge list and FS_list as input. Edge list is the collection of guard conditions saved from test path and FS_list is extracted functional scenario form from specification. First, the algorithm sets the label of the two lists unvisited. And for each in edge list do data integration. Data integration is like data intersection. For example, if we contain two guard conditions x < 6 and x < 60, the integration of it is x < 6.

Algorithm 1 Matching

```
Input: Edge_list, FS_list
Output: labelled Edge_list, labelled FS_list
1: for each edge ∈ Edge_list do
2:     integration(edge.guard_collection)
3:     edge.label = unvisited
4: for each fs ∈ FS_list do
5:     fs.label = unvisited
6: for each edge ∈ Edge_list do
7:     for each fs ∈ FS_list do
8:         if fs.$S_{pre} \wedge G_i$ == edge.guard_collection then
9:             fs.label = edge.ID
10:            edge.label = fs.ID
11:    if e thendge.label == unvisited
12:        return (edge, exist unmatched requirement);
13: if e thenxist FS.label == unvisited
14:    return (fs, exist unmodeled requirement);
15: else
16:    return Edge_list, FS_list;
```

After completing the initialization step, find a matching functional scenario for each element in edge list. The specific operation is: the edge after the integration compares with $S_{pre} \wedge G_i$ in the functional scenario, if exactly the same, then we find the edge with the matched functional scenario. If there is no exact matched functional scenario, then there is an inaccurate modeling problem and needs to be refined. Therefore, immediately terminate the program, the problem of the edge will also be returned. After traversing all the edge_list, we also need to check whether each in FS_list has been visited. If there is an unvisited functional scenario, then it means that there is a requirement that the model fails to be represented in the specification, and the model needs to be refined.

6.5. Path Triple Establishment

Establish Path Triple and apply each node with the axiom in Hoare Logic. "($S_{pre} \wedge G_i \wedge D_i$) (i = 2, … ,n)" denote one functional scenario and P = [$node_1$; $node_2$; … ;$node_m$] be a program path in which each $node_j$(i = 2, … , n) is called a functional node, which is a DecisionNode, ActionNode, or others. Assume each path P has its own target functional scenario, which is decided utilizing matching algorithm. To verify the correctness of P with respect to the functional scenario, we need to construct Path Triple: $\{S_{pre}\}$ P $\{G_i \wedge D_i\}$.

The path triple is similar in structure to Hoare triple, but is specialized to a single path rather than the whole program. It means that if the pre-condition S_{pre} of the program is true before path P is executed, the post-condition $G_i \wedge D_i$ of path P will be true on its termination. By applying the axiom of assignment in Hoare Logic repeatedly, we can get pre-assertion, P_{pre}. Each node has different processing approach, and the details are listed in the form below.

Table 1. Processing approach of AD node

Node Type	**Approach**
ActionNode(assignment)	The axiom for assignment
ActionNode(input/output)	SKIP
Others node	SKIP

Finally, we can form the following expression:

$$\{S_{pre} \wedge G_i\}\ P\ \{D_i\} \tag{15}$$

$$\{S_{pre} \wedge G_i\} \rightarrow P_{pre} \tag{16}$$

, where $S_{pre}(\sim x/x)$, $P_{pre}(\sim x/x)$ and $G_i \wedge D_i(\sim x/x)$ are a predicate resulting from substituting every decorated input variable ~x for the corresponding input variable x in the corresponding predicate, respectively.

If we get a path [start -> action01: input c -> selection01 -> action02: price: =10 -> merge01 -> end], which represents a path generating according to the above specification. Using the above table, we can form path triple:

```
{age>0 AND age>6}
start
action01: input c {price=10}
selection01
action02: price:=10
merge01
{price=10}
```

Then, we can apply for assignment to this path triple, like this:

```
{age>0 AND age>6}
{10=10}
start
{10=10}
action01: input c {price=10}
{10=10}
selection01
{10=10}
action02: price:=10
{price=10}
merge01
{price=10}
```

, where {10=10} is pre-assertion.

6.6. Implication

Prove the implication. Finally, the correctness of one path whether it meets the corresponding requirement is changed into the proof of the implication "$S_{pre} \wedge G_i \rightarrow S_{pre}$". If the implication can be proved, it means that the path can model one part of the requirement; otherwise, it indicates the existence of some error on the path.

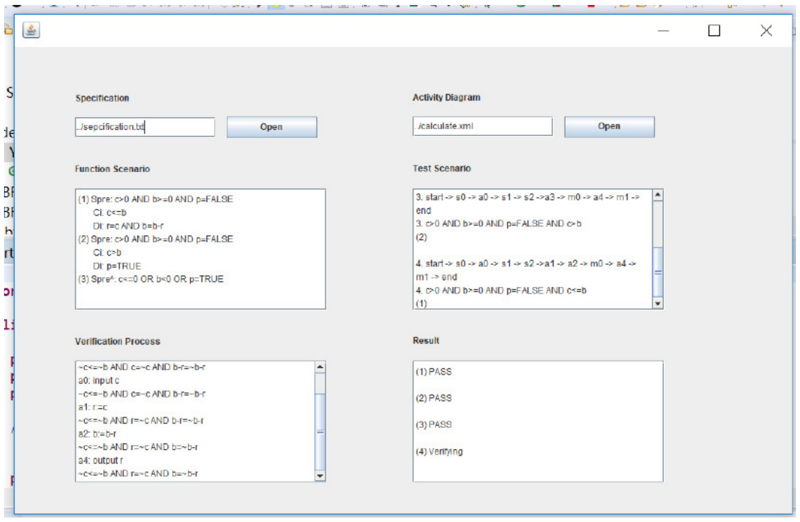
Formally proving the implication "$S_{pre} \wedge G_i \rightarrow S_{pre}$" may not be done automatically, even with the help of a theorem prover such as PVS, depending on the complexity of S_{pre} and P_{pre}. Our strategy is as follows: if the complexity of data structure is not high, we will transform the problem into solver, which can achieve full automation. Otherwise, if achieving a full automation is regarded as the highest priority, as taken in our approach, the formal proof of this implication can be "replaced" by a test. That is, we first generate sample values for variables in S_{pre} and P_{pre}, and then evaluate both of them to see whether P_{pre} is false when S_{pre} is true. If this is true, it tells that the path under examination contains an error.

For example, if we need to judge the validity of the implication "(age > 0 AND normal: = 10) → (a < 12 AND normal * 0.5 =~ normal^2 - ~normal)", use the test case (age, 6), (normal, 10) and we can easily prove the implication is not correct.

7. Supporting Tool

We have developed a prototype software tool to support the TBFV-M method. Specifically, it provides five major functions, which are functional scenario generation, test path generation, matching function scenarios to test paths, pre-condition derivation, verification of test paths, and output of verification result.

The tool interface is shown in Figure10. We can load specification and Activity Diagram. We simply use .txt file to store specification and notice that the specification file should guarantee the unified formal expression. We choose Enterprise Architect modeling tool to establish the system model. The Enterprise Architect modeling tool is powerful and supports the SysML model that will be created in Enterprise Architect and exported into XML format, so the Activity Diagram is described in XML file.



```
../sepcification.txt
```

```
../calculate.xml
```

```
(1) Spre: c>0 AND b>=0 AND p=FALSE
    Ci: c<=b
    Di: r=c AND b=b-r
(2) Spre: c>0 AND b>=0 AND p=FALSE
    Ci: c>b
    Di: p=TRUE
(3) Spre^: c<=0 OR b<0 OR p=TRUE
```

```
3. start -> s0 -> a0 -> s1 -> s2 ->a3 -> m0 -> a4 -> m1 ->
end
3. c>0 AND b>=0 AND p=FALSE AND c>b
(2)

4. start -> s0 -> a0 -> s1 -> s2 ->a1 -> a2 -> m0 -> a4 ->
m1 -> end
4. c>0 AND b>=0 AND p=FALSE AND c<=b
(1)
```

```
~c<=~b AND c=~c AND b-r=~b-r
a0: input c
~c<=~b AND c=~c AND b-r=~b-r
a1: r:=c
~c<=~b AND r=~c AND b-r=~b-r
a2: b:=b-r
~c<=~b AND r=~c AND b=~b-r
a4: output r
~c<=~b AND r=~c AND b=~b-r
```

```
(1) PASS

(2) PASS

(3) PASS

(4) Verifying
```

Figure 10. Tool interface

The second step is deriving functional scenarios and generating test paths. And the tools interface is shown in Fig.10. The below two windows are used to display the intermediate outcome. When the user clicks on "match", the match result will refresh into the test paths window. And if it exists unmatched part, a popup will remind user to refine the model.

8. Case Study

Now we show a motivation example to detail the process of MBSE and TBFV-M method described in the article above. First, we will get a requirement from the user, which consists of inform the description: “In a banking system, a money-withdrawing function needs to be realized. User input the required cash(c), if the cash is less than or equal to the balance(b), then the amount of money received(r), do not print information(p), the balance deducted the corresponding cash. Otherwise, user will not get money and the screen will print "insufficient balance"." This specification is formal and structured, as shown:

```
Specification:
process: cash_withdraw ( c: int ) r: int
ext: wt b: int, wt p: bool
pre: c>0 and b>=0 and p=FALSE
post: c<=b and r=c and b=b-r
      or
      c>b and p=TRUE
end_process
```

According to the specification, we can construct a set of SysML model and the Activity Diagram is shown below.

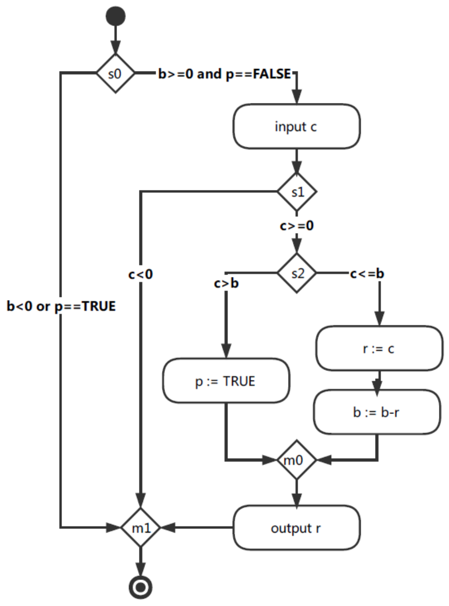


Figure 11. Activity Diagram

We can find the expression is described with SOLF. After getting ready with all the input, specification and Activity Diagram, we will start the TBFV-M method process. First, derive Functional Scenarios from specification and generate test paths from Activity Diagram. The result is shown as below.

```
FSF:
   1  Spre: c>0 AND b>=0 AND p=FALSE
Gi: c<=b
Di: r=c AND b=b-r
   2  Spre: c>0 AND b>=0 AND p=FALSE
Gi: c>b
Di: p=TRUE
   3  Spre^: c<=0 OR b<0 OR p=TRUE
```

```
Test Path:
1. start -> s0 -> m1 -> end
2. start -> s0 -> a0 -> s1 -> m1 -> end
3. start -> s0 -> a0 -> s1 -> s2 ->a3
->m0 -> a4 -> m1 -> end
4. start -> s0 -> a0 -> s1 -> s2 ->a1
-> a2 ->m0 -> a4 -> m1 -> end
```

At the same time, we can extract data constraints from each test scenario, which is used for matching with functional scenario. Then, the matching process is shown below. If it does not exist a matched functional scenario, then it means that it exists a problem in the model, exactly in this unmatched test path. This path is not established accurately according to the requirements described in specification in the activity diagram model. If the match succeeds, it indicates that the test path is designed for the matched test scenario.

```
FSF Match:
1. b<0 OR p=TRUE
(3)
2. c<=0
(3)
3. c>0 AND b>=0 AND p=FALSE AND c>b
(2)
4. c>0 AND b>=0 AND p=FALSE AND c<=b
(1)
```

We will do the verification of test scenario according to the successfully matched functional scenario. First, we establish Path Triple and then apply the axiom of Hoare Logic to derive P_{pre}, pre-assertion of one path for the corresponding test path. The blow figure chose the forth path and matched the first functional scenario as an example and shows the substitution process, from bottom to up. So, the top one "~c≤ ~b AND c =~c AND b-r =~b-r" is the P_{pre}.

```
start -> s0 -> a0 -> s1 -> s2 ->a1 -> a2
->m0 -> a4 -> m1 -> end

~c<=~b AND c=~c AND b-r=~b-r (Ppre)
a0: input c
~c<=~b AND c=~c AND b-r=~b-r
a1: r:=c
~c<=~b AND r=~c AND b-r=~b-r
a2: b:=b-r
~c<=~b AND r=~c AND b=~b-r
a4: output r
~c<=~b AND r=~c AND b=~b-r
```

Finally, we turn this verification problem into proving whether the pre-condition of specification can imply P_{pre}. If it can be proved, means that the path satisfies the requirement. If not, there is a problem existing in the model, exactly in this unmatched test path. If the matched pre-condition can imply the corresponding P_{pre} of all the test paths in the model, then the model is satisfied with the user's requirements.

From the above segment, we can see the implication (~c > 0 AND b ≥ 0 AND ~p = FALSE AND ~c ≤ ~b) → (~c ≤~ b AND c =~c AND b - r =~b - r) is true. This it means that the test path is satisfied with the corresponding functional scenario. We have proved all the test paths, due to the space limit, we omit further details.

9. CONCLUSION

We have presented an approach, known as TBFV-M (Testing-Based Formal Verification for Model), for requirement design error detection in SysML Activity Diagrams by integrating test cases generation and Hoare Logic. The principle underlying TBFV-M is first to derive functional scenarios from specifications and generate test scenarios from Activity Diagrams. Then match them and verify each test scenario according to the corresponding functional scenario. Hoare logic is used during the verification process. TBFV-M method solve the limitation of TBFV, not concerning about models and solved the problem of inconsistent, incomplete, and inaccurate models. It has advantage in reducing the probability of system error and shortening the developing time.

ACKNOWLEDGEMENTS

This work was supported by JSPS KAKENHI Grant Number 26240008, and Defence Industrial Technology Development Program JCKY 2016212B004-2.

REFERENCES

[1] A. W. Wymore, Model-based systems engineering: an introduction to the mathematical theory of discrete systems and to the tricotyledon theory of system design. CRC Press, 1993.

[2] S. Friedenthal, A. Moore, and R. Steiner, “A practical guide to sysml,” San Francisco Jung Institute Library Journal, vol. 17, no. 1, pp. 41-46, 2012.

[3] T. Weilkiens,”Systems engineering with sysml/uml,” Computer, no. 6, p. 83, 2006.

[4] M. Shah, L. Chrpa, F. Jimoh, D. Kitchin, T. Mccluskey, S. Parkinson, and M. Vallati, “Knowledge engineering tools in planning: State-of-the-art and future challenges,” Computer, 01 2013.

[5] T. S. Vaquero, J. R. Silva, and C. J. Beck, \A brief review of tools and methods for knowledge engineering for planning scheduling," Computer, pp. 7-14, 2011.

[6] S. Liu, “Utilizing hoare logic to strengthen testing for error detection in programs,” Computer, vol. 50, no. 6, pp. 1-5, 2014.

[7] S. Liu and S. Nakajima, Combining Specification-Based Testing, Correctness Proof, and Inspection for Program Verification in Practice. Springer International Publishing, 2013.

[8] S. Liu, “A tool supported testing method for reducing cost and improving quality,” in IEEE International Conference on Software Quality, Reliability and Security, 2016, pp. 448-455.

[9] S. Liu, Testing-Based Formal Verification for Theorems and Its Application in Software Specification Verification. Springer International Publishing, 2016.

[10] S. Liu, A. J. Ofiutt, C. Hostuart, Y. Sun, and M. Ohba, “So: A formal engineering methodology for industrial applications,” IEEE Transactions on Software Engineering, vol. 24, no. 1,pp. 24-45, 1998.

[11] F. Raimondi, C. Pecheur, and G. Brat, “Pdver, a tool to verify pddl planning domains,” Computer, 2009.

[12] S. Marrone, F. Flammini, N. Mazzocca, R. Nardone, and V. Vittorini, "Towards model-driven v&v assessment of railway control systems," International Journal on Software Tools for Technology Transfer, vol. 16, no. 6, pp. 669-683, 2014.

[13] F. Flammini, S. Marrone, N. Mazzocca, R. Nardone, and V. Vittorini, "Model-driven v&v processes for computer-based control systems: A unifying perspective," Computer, vol. 7610, pp. 190-204, 2012.

[14] F. Liang, W. Schamai, O. Rogovchenko, S. Sadeghi, M. Nyberg, and P. Fritzson, "Model-based requirement veri_cation : A case study," in International Modelica Conference, Munich,Germany, 2012.

[15] R. Sasse and J. Meseguer, "Java+itp: A veri_cation tool based on hoare logic and algebraic semantics 1," Electronic Notes in Theoretical Computer Science, vol. 176, no. 4, pp. 29-46, 2007.

[16] M. O. Myreen and M. J. C. Gordon, "Hoare logic for realistically modelled machine code." In TOOLS and Algorithms for the Construction and Analysis of Systems, International Conference, Tacas 2007, Held As, 2007, pp. 568-582.

[17] J. Lasalle, F. Bouquet, B. Legeard, and F. Peureux, "Sysml to uml model transformation for test generation purpose," Acm Sigsoft Software Engineering Notes, vol. 36, no. 1, pp. 1-8, 2011.

[18] A. Nayak and D. Samanta, "Synthesis of test scenarios using uml activity diagrams," Software & Systems Modeling, vol. 10, no. 1, pp. 63-89, 2011.

[19] O. Oluwagbemi and H. Asmuni, "Automatic generation of test cases from activity diagrams for uml based testing (ubt)," Computer, vol. 77, no. 13, 2015.

[20] S. Khurshid and D. Marinov, "Testera: Speci_cation-based testing of java programs using sat, "Automated Software Engineering, vol. 11, no. 4, pp. 403-434, 2004.

[21] S. Liu and S. Nakajima, "A decompositional approach to automatic test case generation based on formal specifications," in International Conference on Secure Software Integration Reliability Improvement, 2010, pp. 147-155.

[22] S. Liu, T. Hayashi, K. Takahashi, K. Kimura, T. Nakayama, and S. Nakajima, "Automatic transformation from formal specifications to functional scenario forms for automatic test case generation," in New Trends in Software Methodologies, TOOLS and Techniques Proceedings of the Somet 10, September 29 October 1, 2010, Yokohama City, Japan, 2010, pp. 383-397.

[23] Kent and Stuart, Model Driven Engineering. Springer Berlin Heidelberg, 2002.

[24] M. Broy, K. Havelund, R. Kumar, and B. Steffen, Towards a Unified View of Modelling and Programming (Track Summary). Springer International Publishing, 2016.

[25] G. Gay, "Generating effective test suites by combining coverage criteria," in International Symposium on Search Based Software Engineering, 2017, pp. 65-82.

[26] A. K. Joseph, G. Radhamani, and V. Kallimani, \Improving test efficiency through multiple criteria coverage-based test case prioritization using modified heuristic algorithm," in International Conference on Computer and Information Sciences, 2016, pp. 430-435.

[27] C. A. R. Hoare, "An axiomatic basis for computer programming," Communications of the Acm, vol. 12, no. 1, pp. 53-56, 1969.

[28] R. W. Floyd, Assigning Meanings to Programs. Springer Netherlands, 1993.

[29] V. R. Pratt, “Semantical consideration on oyo-hoare logic," in Symposium on Foundations of Computer Science, 1976, pp. 109-121.

[30] Y. Yin, Y. Xu, W. Miao, and Y. Chen, \An automated test case generation approach based on activity diagrams of sysml," International Journal of Performability Engineering, vol. 13, no. 6, pp. 922-936, 2017.

AUTHORS

Yufei Yin
Master Student of East China Normal University
Exchange student in Hosei University

Prof. Dr. Shaoying Liu
High-Quality Software Engineering Lab
Department Faculty of Computer and Information Sciences
Hosei University

Prof. Dr. Yixiang Chen
Software and hardware co design technology and application, director of Engineering Research Centre
East China Normal University