Verification of SysML Activity Diagrams Using Hoare Logic and SOFL

Yufei Yin[1(✉)], Shaoying Liu[2], and Yixiang Chen[1]

[1] East China Normal University, Shanghai, China
2571603738@qq.com
[2] Hosei University, Tokyo, Japan

Abstract. During the process of utilizing Model-Based Systems Engineering (MBSE), SysML activity diagrams are often used for designing the 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 and to the best of our knowledge, there are few tools to support the verification of SysML models. Testing-based formal verification (TBFV) is designed for verifying the sequence code. To solve the problem, we creatively apply the existing TBFV approach into the verification of SysML activity diagrams and established a new approach, called TBFV-M. TBFV-M has ability to verify a SysML activity diagrams meet the user' need. We also propose a method to dealing with invocation, because invocation is very common in the model-driven development process. In this paper, 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 design large scale systems, because it can make sure of their reliability and save the cost of modification effectively. The systems modelling language SysML [2, 3] can support effective use of MBSE, for its well-designed mechanism for creating object-oriented models, which can be combined with software, people, material and other physical resources. In MBSE, SysML models are often used as the design for code. It means that whether the SysML model meets the users' requirement in relation to 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. TBFV integrated the specification-based testing approach and Hoare logic to verify the correctness of all the traversed program paths during testing. The advantage of TBFV is

Z. Duan et al. (Eds.): SOFL+MSVL 2018, LNCS 11392, pp. 71–88, 2019.
https://doi.org/10.1007/978-3-030-13651-2_5

its potential and capability of achieving full automation for verification utilizing 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 behavior and describe complex control and parallel activities, 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. And at the same time, test paths are generated from corresponding SysML Activity Diagrams waiting to be verified. Then, test paths are matched with functional scenarios by a given algorithm. 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 with the guard condition of the functional scenario to the derived pre-condition of the path is verified which concerns the accuracy of the activity diagram. And the processing method of dealing with invocation is also be proposed by TBFV-M.

The remainder of the article will detail the TBFV-M method. Section 2 presents related work we have referenced. Section 3 characterizes the definitions of basic terms and concepts. Section 4 introduces TBFV and the derivation of the main idea of TBFV-M. Section 5 describes the principle of TBFV-M, showing the core technology of TBFV-M. Section 6 uses one case study to present the key point of TBFV-M. Finally, the details of the implementation are presented in Sect. 6 and Sect. 7 concludes the paper.

2 Related Work

2.1 Testing-Based Verification

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.

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.

2.2 Test Case Generation

Lasalle [12] 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.

Nayak [13] 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.

Oluwagbemi [14] proposed a new concept called activity flow 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 Related Concept

3.1 Formal Definition of Activity Diagram

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

$$AD = (Node; Edge) \tag{1}$$

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

$$Node = \{InitialNode; FlowFinalNode; ActivityFinalNode; Action{-}Node; ActivityNode; ForkNode; JoinNode; DecisionNode; MergeNode; RecieveSignaNode; SendSignalNode\} \tag{2}$$

InitialNode signifies the beginning of Activity Diagram, while ActivityFinalNode signifies the ending of Activity Diagram. Edges defines the relationship between nodes such that:

$$Edge = \{(x, y)|x, y \in Node\} \tag{3}$$

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.

3.2 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:

$$TC(AD) = (Path; Data) \tag{4}$$

For Activity Diagram, test path 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:

$$path = \left(a'_1, a'_2, \ldots, a'_n\right) \tag{5}$$

$$a'_i = (t_n, a_n), (i = 2, \ldots, n) \tag{6}$$

$$t_n = a_{i-1} \rightarrow a_i, (i = 2, \ldots, n) \tag{7}$$

In these formulas, 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 $t_2 \ldots t_n$.

Test data indicates the input information corresponding to a particular test scenario including various types of data, even user actions and so on.

3.3 Test Coverage Criteria

For software, the adequacy measurement of testing is reflected in the rate of coverage and effectiveness of the test case. 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 [15, 19, 20]:

- 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.

3.4 Hoare Logic

Hoare Logic is a formal system developed by Hoare [21, 22], and it is designed for the proof of partial correctness of a program. In Hoare Logic, the Hoare Triple [23] is best known and is also referenced in our method. The Hoare triple is of this form:

$$\{P\}\, C\, \{Q\} \tag{8}$$

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:

$$\{Q(E\backslash x)\}\, x := E\, \{Q\}, \tag{9}$$

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] are true because x is assigned by representing E after the execution.

3.5 Functional Scenario Form

A functional scenario is a logical expression that tells clearly what condition is used to constrain the output when the input satisfies some condition. S_{pre} and S_{post} denote the pre- and post-conditions of operation S. Let:

$$S_{post} = (G_1 \wedge D_1) \vee (G_2 \wedge D_2) \vee \ldots \vee (G_n \wedge D_n), \tag{10}$$

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 listed below:

$$Functional\, Scenario = S_{pre} \wedge G_i \wedge D_i \tag{11}$$

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.

$$FSF = (S_{pre} \wedge G_1 \wedge D_1) \vee (S_{pre} \wedge G_2 \wedge D_2) \vee \ldots \vee (S_{pre} \wedge G_n \wedge D_n) \tag{12}$$

3.6 Path Triple

The path triple is similar in structure to Hoare triple, but is specialized to a single path rather than the whole program and the definition is below:

$$\{S_{pre} \wedge G_i\}\, P\, \{D_i\}, \tag{13}$$

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.

4 TBFV and TBFV-M

4.1 TBFV

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 [19], 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, and also overcome its weak point by making good use of relevant part of Hoare logic.

4.2 TBFV-M

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 [17]. Because of the graphical notation for defining system design as nodes and edge diagrams, SysML model addresses the ease of adoption amongst engineers [18] (Fig. 1).

During the Model-Driven process, model is an important medium for the Model based system engineering development. 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 according to the specification. 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).

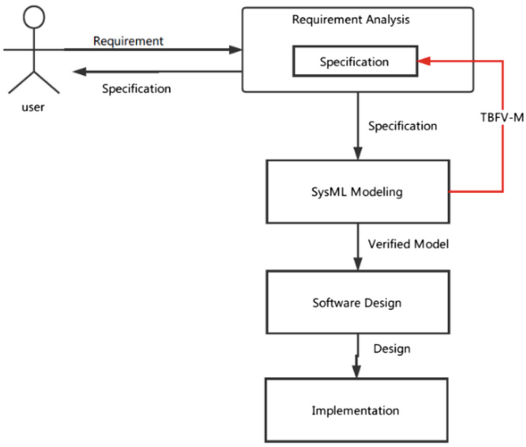

Fig. 1. TBFV-M usage scenario.

5 Principle of TBFV-M

The procedure of TBFV-M is illustrated in Fig. 2. 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

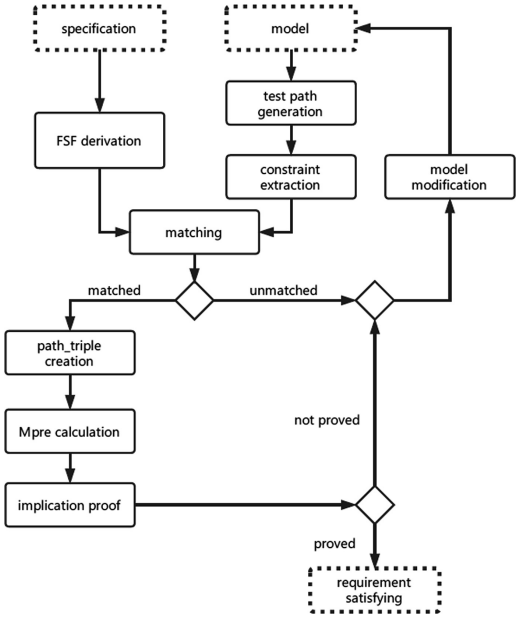

Fig. 2. TBFV-M processing procedure.

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 we conclude that the model is to meet the requirements.

5.1 Unified Formal Expression

Using a unified formal expression can not only reduce the ambiguity during communications, but also give a possibility to automate the entire process, making analysis and verification more accurate and efficient.

We establish the unified formal expression, including specification guide and modeling guide. Specification reflects complete requirements and we chose SOFL to describe formal specification. The SOFL method intergrades formal methods, structured methods and object-oriented methodology, which not only supports requirements analysis and specifications, but also play an import role during design and implementation stages. An example specification written in SOFL is given below. It describes that if a non-negative integer a equals to zero, TRUE will be returned; otherwise return FALSE.

```
process: equal_zero (a: int) equal: bool
pre: a > 0
post: a == 0 AND equal == TURE
OR
a != 0 AND equal == FALSE
```

5.2 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 related concept section. 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 [16].

The below segment of the process "equal_zero", mentioned previously, shows the FSF generated from the specification described in the last one.

```
1.  S_pre: a > 0
    G_1: a == 0
    D_1: equal == TRUE

2.  S_pre: a > 0
    G_2: a != 0
    D_2: equal == FALSE
3.  ~S_pre: a <= 0
```

5.3 Test Path 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, according to test path generation algorithms and coverage criteria chosen by test group members.

The 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 an iteration 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 [24].

We give a motivation case to show the above process. Figure 3 is an unstructured SysML activity diagram model, which contains a concurrency module and a loop module.

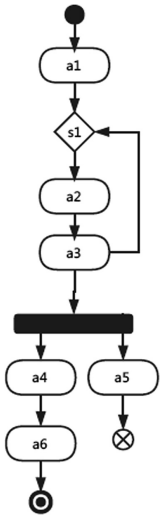

Fig. 3. Motivation example.

Figure 4 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 Fig. 4(a). The compressed black box node is the intermediate representation of the loop shown in the following Fig. 5(a).

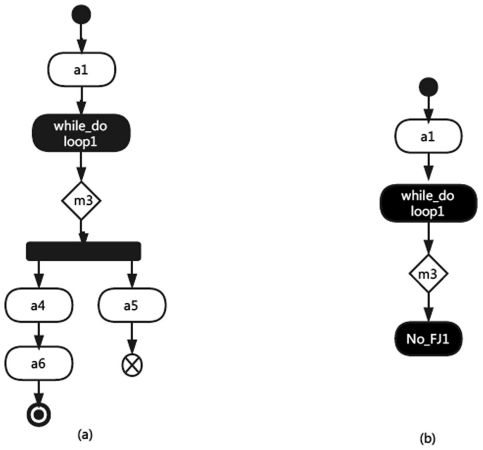

Fig. 4. Transformation process.

The second step is to identify the noJoin concurrency module and compress it into a black box node No FJ1, shown in Fig. 4(b). The compressed black box node is shown in the following Fig. 5(b).

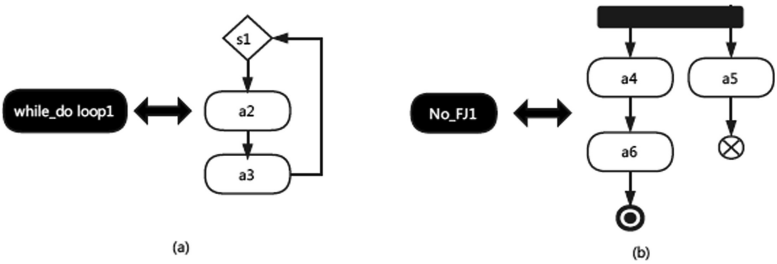

Fig. 5. Motivation example.

Figure 5(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.

5.4 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 exist in this model. And the model needs to be modified. The matching algorithm is given below.

Algorithm 1 Matching

```
Input: Edge_list, FS_list
Output: labelled Edge_list, labelled FS_list
for each edge ∈ Edge_list do
    integration(edge.guard_collection)
    edge.label = unvisited
for each fs ∈ FS_list do
    fs.label = unvisited
for each edge ∈ Edge_list do
    for each fs ∈ FS_list do
        if fs.S_pre ∧ G_i == edge.guard_collection then
            fs.label = edge.ID
            edge.label = fs.ID
    if e thendge.label == unvisited
        return (edge, exist unmatched requirement);
if e thenxist FS.label == unvisited
    return (fs, exist unmodeled requirement);
else
    return Edge_list, FS_list;
```

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 edge 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 $x < 6$.

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 mean that 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.

5.5 Path Triple Establishment

Establish Path Triple and apply each node with the axiom in Hoare Logic. "$(S_{pre} \wedge G_i \wedge D_i)$ $(i = 2, \ldots, n)$" denote one functional scenario and $P = [node_1; node_2; \ldots; node_m]$ be a program path in which each $node_j (j = 2, \ldots, n)$ is called a functional node, which is a DecisionNode, ActionNode, or other activity diagram nodes. 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. Repeatedly apply the axiom for assignment to derive a pre-assertion, denoted by Ppre. Finally, we can form the following expression:

$$\{Spre \wedge Gi\} \rightarrow Ppre, \tag{14}$$

where S_{pre}, P_{pre} and $G_i \wedge D_i$ are a predicate resulting from substituting every decorated input variable $\sim x$ for the corresponding input variable x in the corresponding predicate, respectively. And the correctness of the specific path is transformed into the implication Spre $\wedge$ Gi $\rightarrow$ Ppre. 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.

5.6 Implication

Prove the implication. Finally, the correctness of one path whether it meets the corresponding requirement is changed into the proof of the implication "$S_{pre} \wedge G_i \rightarrow P_{pre}$". 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 "$S_{pre} \wedge G_i \rightarrow P_{pre}$" 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 Spre is true.

For example, if we need to judge the validity of the implication "(price > 0) $\rightarrow$ (price < 100 AND $\sim$ price-5 = $\sim$price^2 - $\sim$price", use the test case (price, 60) and we can easily prove the implication is not correct.

5.7 Invocation

During the process of design, especially for the complex system, modularization is very necessary when modelling, according to users' requirements. Model driven software development process often faces the problem of function or module invocation.

Because the TBFV-M method needs to deal with functional scenario derivation from specification describing users' requirement and test path generation from SysML activity diagrams, we need to take both side into account while dealing with invocation.

For specification, if a function invocation is used as a statement, it can change the current state of a program. So that, the traversed path containing the invoked function should consider in deriving the pre-assertion of the invocated function. Our solution is

utilizing the sub path of the invocated function to substitute the actual traversed path, while deriving the functional scenario form. Also, we need to append the pre-condition of invocated function into the S_{pre} of particular functional scenario and during the above process parameter substitution needs to be considered.

To express the idea, we will give a motivation example.

```
function FareDiscount (age:int, fare:int) FinalPrice: int
pre: age > 0 AND fare > 0
post: age <= 6 AND FinalPrice == 0
     OR
     age >= 70 AND FinalPrice == 0
      OR
     age > 6 AND age<60 AND FinalPrice == fare
      OR
  age >= 60 AND age<70 AND FinalPrice == HalfPrice(fare)

function HalfPrice (price: int) Half_P: int
pre: price > 0
post: Half_P = 0.5 * price
```

While deriving, we can get the below functional scenario. HalfPrice is the invocation function.

S_{pre}:age > 0 AND fare > 0
G_4: age >= 60 AND age<70
D_4: FinalPrice == HalfPrice(fare)

According to the solution we mentioned above, we will substitute the original form with the sub path of invocation function and the actual parameter `price` is replaced by `fare` in the invocation function. The result is shown below.

S_{pre}:age > 0 AND fare > 0
G_4: age >= 60 AND age<70
D_4: FinalPrice == 0.5 * fare

For Activity Diagram, "Activity" is often used to realize the hierarchy design. Our solution is also utilizing the sub path of the invocated activity to substitute the actual traversed test path, while generating test path.

6 Case Study

Now we show a motivation example to detail the process of TBFV-M method. First, we will get a requirement from the user, which consists of inform the description, may like this: "The park will give the tourist fare discount according to their age. If he is

younger than 6 or older than 70, he will be free; Or if he is between 60 and 70, he can enjoy the half price, otherwise he will pay the normal price". This specification is formal and structured, as shown in the last section.

According to the specification, we can construct a set of SysML model and the Activity Diagram is shown below (Fig. 6).

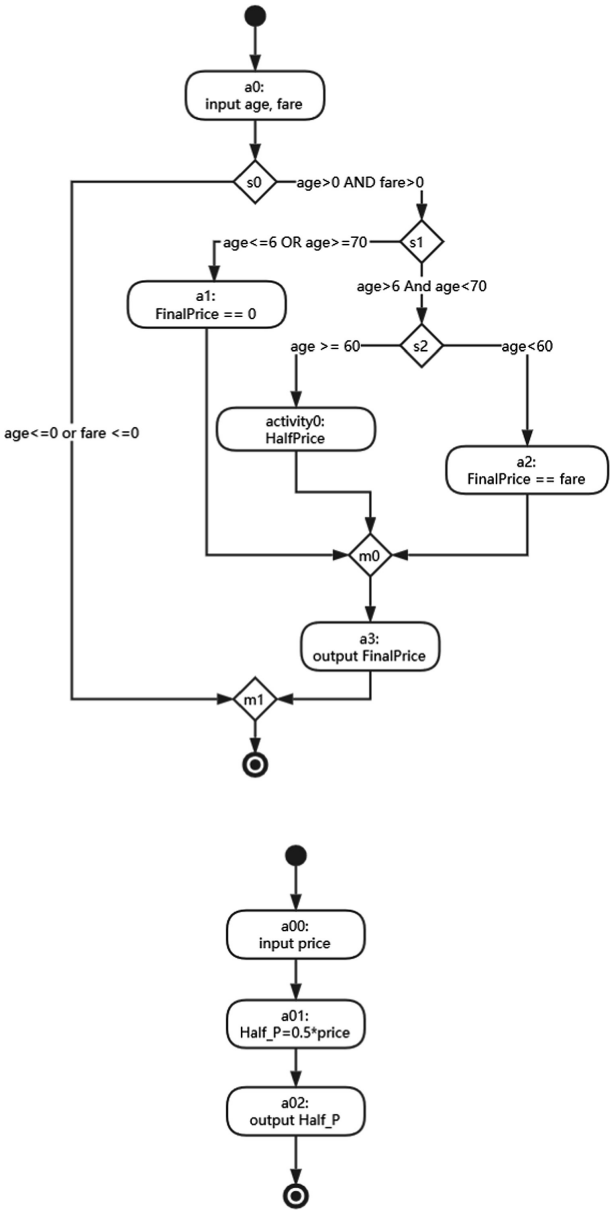

Fig. 6. Case study.

First, we derive Functional Scenarios from specification and generate test paths from Activity Diagram. The result is shown as below.

1. S_{pre}: age > 0 AND fare > 0
   G_1: age <= 6
   D_1: FinalPrice == 0
2. S_{pre}: age > 0 AND fare > 0
   G_2: age >= 70
   D_2: FinalPrice == 0
3. S_{pre}: age > 0 AND fare > 0
   G_3: age > 6 AND age<60
   D_3: FinalPrice == fare
4. S_{pre}: age > 0 AND fare > 0
   G_4: age >= 60 AND age<70
   D_4: FinalPrice == 0.5*fare
5. ~S_{pre}: age<=0 or fare<=0

Because of the invocated activity, we should substitute the original test path, like T4, into the update version, T4′, by substituting activity0 with its sub actions.

```
Test Path:
T1: start →a0 → so → m1 → end
T2: start → a0 → so → s1 → a1 → m0 → a3 → end
T3: start → a0 → so → s1 → s2 → a2 → m0 → a3 → end
T4: start → a0 → so → s1 → s2 → activity0 → m0 → a3 → end

T4´: start → a0 → so → s1 → s2 → start_0 → a00 → a01 →a02
→ end_0 → m0 → a3 → 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.

```
Matching Result:
FSF_1 - T2
FSF_2 - T2
FSF_3 - T3
FSF_4 - T4
FSF_5 - T1
```

The blow segment chose the forth path and matched the first functional scenario as an example and shows the substitution process, from bottom to up.

```
Derivation Process:

{age> 0 AND fare > 0 AND age >= 60 AND age<70}
{0.5 *fare == 0.5*fare}
input age, fare
{0.5 *fare == 0.5*fare}
input fare
{0.5 *fare == 0.5*fare}
FinalPrice =0.5 *fare
{FinalPrice == 0.5*fare}
output FinalPrice
{FinalPrice == 0.5*fare}
output FinalPrice
{FinalPrice == 0.5*fare}
```

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. As the strategy of implication mentioned before, this implication uses simple data structure, so that we use testing to access the procedure of verification. In this case, we prove it is correct.

7 Conclusion

We presented an approach, known as TBFV-M (Testing-Based Formal Verification for Model), for requirement 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 form specification 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 made up the limitation of TBFV, not concerning about models and solved the problem of inconsistent, incomplete, and inaccurate models. We also give a solution to deal with the invocation problem. 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 Defense Industrial Technology Development Program JCKY 2016212B004-2. The authors would like to thank the anonymous referees for their valuable comments and suggestions.

References

1. Wymore, A.W.: Model-Based Systems Engineering: An Introduction to the Mathematical Theory of Discrete Systems and to the Tricotyledon Theory of System Design. CRC Press, Boca Raton (1993)
2. Friedenthal, S., Moore, A., Steiner, R.: A practical guide to sysml. San Francisco Jung Inst. Libr. J. **17**(1), 41–46 (2012)
3. Weilkiens, T.: Systems engineering with SysML/UML. Computer (6), 83 (2006)
4. Shah, M., et al.: Knowledge engineering tools in planning: state-of-the-art and future challenges. Computer (2013)
5. Vaquero, T.S., Silva, J.R., Beck, C.J.: A brief review of tools and methods for knowledge engineering for planning scheduling. Computer 7–14 (2011)
6. Liu, S.: Utilizing hoare logic to strengthen testing for error detection in programs. Computer **50**(6), 1–5 (2014)
7. Liu, S., Nakajima, S.: Combining specification-based testing, correctness proof, and inspection for program verification in practice. In: Liu, S., Duan, Z. (eds.) SOFL+MSVL 2013. LNCS, vol. 8332, pp. 3–16. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-04915-1_1
8. Liu, S.: A tool supported testing method for reducing cost and improving quality. In: IEEE International Conference on Software Quality, Reliability and Security, pp. 448–455 (2016)
9. Liu, S.: Testing-based formal verification for theorems and its application in software specification verification. In: Aichernig, B.K.K., Furia, C.A.A. (eds.) TAP 2016. LNCS, vol. 9762, pp. 112–129. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41135-4_7
10. Liu, S., Ofiutt, A.J., Hostuart, C., Sun, Y., Ohba, M.: So: a formal engineering methodology for industrial applications. IEEE Trans. Softw. Eng. **24**(1), 24–45 (1998)
11. Raimondi, F., Pecheur, C., Brat, G.: PDVer, a tool to verify PDDL planning domains. Computer (2009)
12. Lasalle, J., Bouquet, F., Legeard, B., Peureux, F.: SysML to UML model transformation for test generation purpose. ACM SIGSOFT Softw. Eng. Notes **36**(1), 1–8 (2011)
13. Nayak, A., Samanta, D.: Synthesis of test scenarios using UML activity diagrams. Softw. Syst. Model. **10**(1), 63–89 (2011)
14. Oluwagbemi, O., Asmuni, H.: Automatic generation of test cases from activity diagrams for UML based testing (UBT). Computer **77**(13) 2015
15. Khurshid, S., Marinov, D.: TestEra: specification-based testing of Java programs using SAT. Autom. Softw. Eng. **11**(4), 403–434 (2004)
16. Liu, S., Nakajima, S.: A decompositional approach to automatic test case generation based on formal specifications. In: International Conference on Secure Software Integration Reliability Improvement, pp. 147–155 (2010)
17. Liu, S., Hayashi, T., Takahashi, K., Kimura, K., Nakayama, T., Nakajima, S.: 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 2010, Yokohama City, Japan, 29 September–1 October 2010, pp. 383–397 (2010)
18. Kent, S.: Model driven engineering. In: Butler, M., Petre, L., Sere, K. (eds.) IFM 2002. LNCS, vol. 2335, pp. 286–298. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-47884-1_16
19. Broy, M., Havelund, K., Kumar, R., Steffen, B.: Towards a unified view of modeling and programming (track summary). In: Margaria, T., Steffen, B. (eds.) ISoLA 2016. LNCS, vol. 9953, pp. 3–10. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47169-3_1

20. Joseph, A.K., Radhamani, G., Kallimani, V.: Improving test efficiency through multiple criteria coverage-based test case prioritization using modified heuristic algorithm. In: International Conference on Computer and Information Sciences, pp. 430–435 (2016)
21. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(1), 53–56 (1969)
22. Floyd, R.W.: Assigning meanings to programs. In: Colburn, T.R., Fetzer, J.H., Rankin, T.L. (eds.) Program Verification, pp. 65–81. Springer, Dordrecht (1993). https://doi.org/10.1007/978-94-011-1793-7_4
23. Pratt, V.R.: Semantical considerations on Floyd-Hoare logic. In: Symposium on Foundations of Computer Science, pp. 109–121 (1976)
24. Yin, Y., Xu, Y., Miao, W., Chen, Y.: An automated test case generation approach based on activity diagrams of SysML. Int. J. Perform. Eng. **13**(6), 922–936 (2017)