Available online at www.ijpe-online.com

International Journal of Performability Engineering

Vol. 13, No. 6, October 2017, pp. 922-936
DOI: 10.23940/ijpe.17.06.p13.922936

An Automated Test Case Generation Approach based on Activity Diagrams of SysML

Yufei Yin, Yiqun Xu, Weikai Miao*, Yixiang Chen

Shanghai Key Lab for Trustworthy Computing, East China Normal University, Shanghai 200062, China

Abstract

Model based software testing is one of the most popular software quality assurance techniques adopted by industrial practitioners. The SysML Activity Diagram (AD) can describe dynamic behaviors of a software system under testing in an intuitive way. That is, the AD is a promising foundation for generating test case to test the target software system. Unfortunately, there are few effective AD based testing approaches for industrial practitioners due to the lack of automated generation technique and powerful tool support, especially for whose shape is out of structure. To tackle this problem, we propose an automated generation approach with a supporting tool. For a specific AD, we first transform it into an intermediate representation form — Intermediate Black box Model (IBM). Then the IBM is used to generate test cases automatically. The approach presented in this paper can make up the deficiency of automated test case generation with the unstructured SysML AD.

Keywords: SysML activity diagram; test case generation; model based software testing

(Submitted on July 25, 2017; Revised on August 30, 2017; Accepted on September 15, 2017)

(This paper was presented at the Third International Symposium on System and Software Reliability.)

1. Introduction

With the increasing scale of the software system, people prefer to use the Model-Based Systems Engineering (MBSE) instead of the traditional software engineering. MBSE [15,20] refers to an introduction to applying the modeling method into the system engineering and using the model to support activities such as system requirements, designs, analysis, verifications and confirmations. A MBSE methodology [3] is used to support a series of related processes, methods, and tools for complying with the laws of MBSE. Systems Modeling Language (SysML) [1,5] is a support language to using MBSE. In MBSE methodology, SysML can describe the system model and bring more intuitive management of the development process. SysML is a convenient graphical language as the primary product of the system development and can express both static structure and dynamic behavior for systems.

Software reliability [13] engineering refers to a series of activities to meet the reliability requirements of the software, such as design, analysis, testing and other works. In complicated software systems, reliability is the most significant aspect of software quality and software faults may make heavy losses. It can even cause casualties, such as Ariane 5 flight failure. You can effectively find the program defects affecting the reliability of the software, achieving the growth of reliability through testing [17].

Generally, the previous research of model-based software testing mainly focused on UML models and there is not a lot of research achievements on SysML models. However, SysML can be applied to modeling for complex systems, especially for embedded systems. Therefore, we chose the automatic generation of test cases based on SysML model as our research priority.

* Corresponding author.
E-mail address: wkmiao@sei.ecnu.edu.cn.

We selected the Activity Diagram (AD) from 9 kinds of SysML models and studied it. We discovered that the existing test case automatic generation algorithms based on SysML activity diagram had some limitations to unstructured AD. They could only deal with structured activity diagram model and usually had restrictions on its shape. However, in the industrial world, the SysML AD is constructed according to the specific requirement and it is usually unstructured. Especially in the model of embedded system, message sending and receiving actions, circulation and concurrent modules often exists. Moreover, on the modeling stage, the model cannot convert to structured, otherwise it will change its proper semantics and violate the requirements. To solve this limitation, we propose an automated test case generation algorithm based on unstructured SysML AD.

The paper is organized as follows: section 2 presents related works we surveyed. Section 3 characterizes the definitions of basic terms and concepts used in the proposed technique. Section 4 introduces the framework of our approach. Section 5 describes the process of transformation from SysML unstructured activity diagram to IBM. Section 6 presents the automated test case generation algorithm according to IBM. The details of the implementation of the algorithm are presented in Section 7. Finally, Section 8 shows a case study and section 9 concludes the paper.

2. Related Work

A lot of research work has been done on model-based test technology and automated test case generation method. And the UML model based automated test case generation algorithm and implementation has always been the hot research topic in academia. However, there are not many related researches based on SysML. Because this article focuses on SysML activity diagram, only the related work based on activity diagrams is listed. There are two types of related work: (1) studies on current popular automated test case generation algorithms discussed in Section 2.1. and (2) implemented tools discussed in Section 2.2.

2.1. Generation Algorithm

These following articles applied the idea of transformation, transforming activity diagram into intermediate representation, similar to the way we propose. Although these algorithms are more powerful in terms of processing, there are still deficiencies.

Ashalatha Nayak [14] introduced an approach to transform the particular AD 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 AD, because the unstructured ADs shape is out of structure and cannot be reduced into a single chain of nodes. And a single chain of nodes is the final intermediate expression, which they used to generate test cases.

Oluwatolani Oluwagbemi [16] constructed 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 works done by the predecessors. They found that they could generate test cases that met the required number of tests completely. However, they still did not address the limitations on unstructured AD.

Debasish Kundu [10] used a more abstract perspective to treat AD than ever before and they took it as corresponding use case. They first transformed the activity diagram into an activity graph and design an algorithm generating test cases from the activity graph automatically. In their article, they claimed that the disadvantage of the current algorithm is that their method can only handle one use case at a time.

Puneet E. Patel [18] implemented algorithms proposed in two articles. The first approach [10] has been introduced before and the second's core idea [2] is to generate by aid of activity diagram path. By comparing the implementation of the two algorithms, he found that they all existed a limitation on only activity diagram of a single use case processing at a time.

In Ajay Kumar Jenas approach [8], two intermediate expressions were used: Activity Flow Table (AFT) and Activity Flow Graph (AFG). First, he designed an algorithm to transform the activity diagram into AFT, then transformed it into a AFG, and used AFG for test case generation. By comparing their work with others, they found that their test case generation algorithm had better quality. Because there are no redundant test cases to be produced, they can generate test cases that meet more requirements of the integration and system level.

2.2. Implemented Tool

Jonathan Lasalle [12] established VETESS, which is a tool chain that can generate test case for embedded systems automatically. By leveraging existing test case generation and test path extraction tools, VETESS can extract valid information from UML or SysML models and execute functional test case automatedly. It is composed of modeling, abstract test case generation, concretization and analysis. The generic process of automated test cases generation is instructive.

Jonathan Lasalle [11] 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. Although this paper focused on the rewriting rules of State Machine diagrams and Block Diagram, it still a new ideas of automated test case generation based on SysML.

Wang Linzhang [19] designed a prototype tool named UMLTGF. It used the idea of gray box testing to generate test cases directly from UML activity diagram. The advantage of this tool is that it leverages both white-box testing to cover all the internal path of the system under test, and also makes full advantage of black box testing to analyze the expected output of the software. However, the tool did not concern about unstructured diagram, just deal with the simple activity diagram.

Chen Mingsong [4] proposed an approach to instrument a JAVA program for the specific UML activity diagram and run the instrumented JAVA program to generate test cases according to the algorithm he designed. Finally, he used the program execution result to analyze the corresponding UML activity diagram. Chen also developed a tool named UMLTGF to support the above process. Chen's approach provides a new train of thought and method for the practice of test case set reduction.

3. Basic Concepts and Definitions

This section will introduce some preparatory knowledge, including the formal definition of activity diagram, test case and test coverage criteria.

3.1. Formal Definition of Activity Diagram

Activity Diagram Formal Definition [6] can be represented as:

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

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

$$\begin{aligned} Node = \{&InitialNode; FlowFinalNode; ActivityFinalNode; ActionNode; ActivityNode; \\ &ForkNode; JoinNode; DecisinNode; MergeNode; RecieveSignalNode; SendSignalNode\} \end{aligned} \tag{2}$$

Here, we emphasize on InitialNode and ActivityFinalNode, such that there is a path from InitialNode to any other node, signifying the real beginning of AD, and there is a path from any other node to ActivityFinalNode, signifying the real ending of AD, respectively.

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. Edges defines the relationship between nodes such that:

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

3.2. Formal Definition of Test Case

From a global view, test case based on the SysML dynamic behavior model, whether based on the activity diagram, the sequential diagram or the state machine diagram, consists of test scenario and test data. And the definition is as followed [21]:

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

For activity diagram, test scenario consists of a series of actions and edges in the diagram, in other words, an execution test path in the activity diagram is from the initial node to the final node. Based on the formal definition of the activity diagram given above, the test path is defined as follow:

$$\text{path} = (a'_1 , a'_2 , \dots , a'_n) \tag{5}$$

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

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

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 $t_2 \dots t_n$. 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.

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. Traditional software testing is divided into white box testing and black box testing. The test coverage criteria in white box tests includes statement coverage, branch coverage, conditional coverage and so on. These coverage criteria ensures 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 [7,9,16]:

- **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.

The TS (Test Suite) based on the SysML activity diagram model is a collection of test cases that meet certain particular test coverage criteria. According to those definitions above, the test scenario, defined by us, is a set of paths that meet these four test coverage criteria from the initial node to the final node.

4. Framework of the Approach

The framework of our approach is shown in the Figure 1. We use SysML AD as input and it may contain unstructured parts. Then, we use transformation algorithm to compress the input AD. 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. Basic module only contains simple actions and black boxes.

The third phase of our approach is test case generation based on IBM. In this phase, two problems should be solved, including basic module test case generation and black box test case generation. And the process of black box test case generation contains loop module and concurrent module.

After these three steps of our approach, we obtain the test cases and we can go to the final phase — testing.

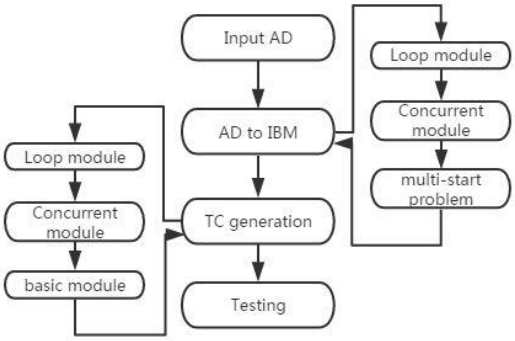

Figure 1. Framework of the approach

5. Transformation from AD to IBM

This chapter focuses on algorithms transforming unstructured SysML activity diagrams into IBM.

5.1. Unstructured Module

Before introducing transformation algorithm from the SysML activity diagram to IBM, we first introduce the unstructured module, including the loop module, concurrent module and the problem of multiple starting nodes in the unstructured SysML activity diagram.

5.1.1. 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 Figure 2, 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. When the loop executes, the do-while loop needs to be executed at least once and the while-do loop execution is not limited.

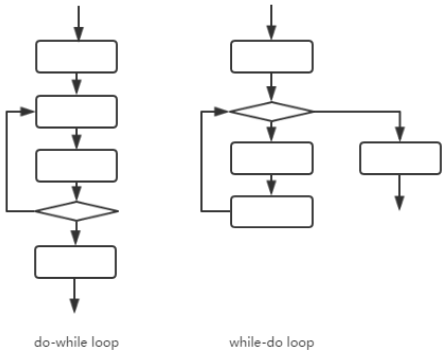

Figure 2. Classification of loop modules based on the position of judgment

5.1.2. 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 Figure 3(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”. That is, you need to wait for all the parallel flows to complete the task before the stream flows to the next action node.

However, the synchronization stream can also be the logical relationship “OR”, as shown in Figure 3(b). The merge node indicates that the control flow out of the merge node can flow to the next action node as long as there is one control flow flowing to the merge node.

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 Figure 3(c) and (d) below, respectively. It means synchronizing only a subset of parallel flows and no parallelism required.

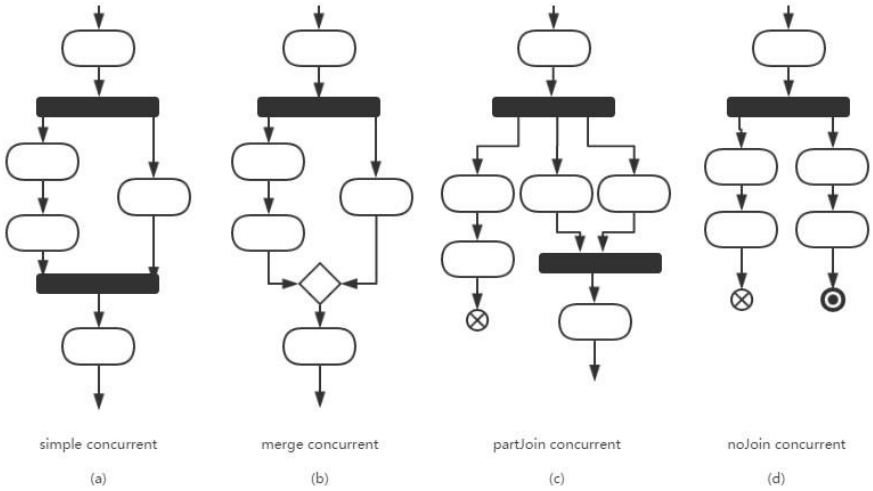

Figure 3. Classification of concurrent modules

5.1.3. The Problem of Multiple Starting Nodes

In unstructured SysML activity diagrams, as Figure 4 shown, there are multiple nodes with zero in-degree. This multi-starting phenomenon can lead to incomplete coverage of the test path if we use the ordinary SysML activity diagram test algorithm. This is because these algorithms only consider the starting point from the initial node and do not consider the other nodes with zero in-degree. These zero-in-degree nodes can also be the starting point of the diagram.

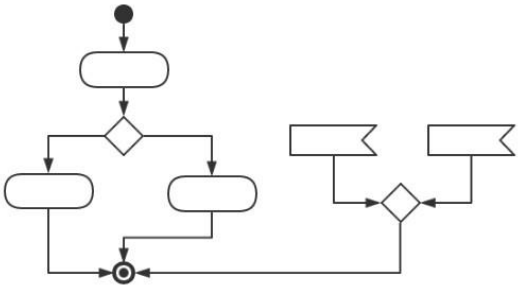

Figure 4. Multiple starting nodes problem

5.2. Transformation of Unstructured Modules

This section focuses on the transformation process from unstructured modules to IBM.

5.2.1. Transformed IBM

Based on the unstructured modules defined above, our algorithm plans to black-box these unstructured modules. The transformed black box node, with a unique incoming edge and an outgoing migration edge, can be embedded into the original activity diagram.

The compressed IBM, which includes only the simple selection structure and the sequential structure, is called the Basic Module. In other words, IBM has two parts, one is the basic module diagram and the other is a collection of nodes represented by the black box.

5.2.2. Loop Module

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. This operation refers

to the loop recognition algorithm in directed graphs in graph theory. Because of the limited space, there is no redundant presentation here. Figure 5 shows the process.

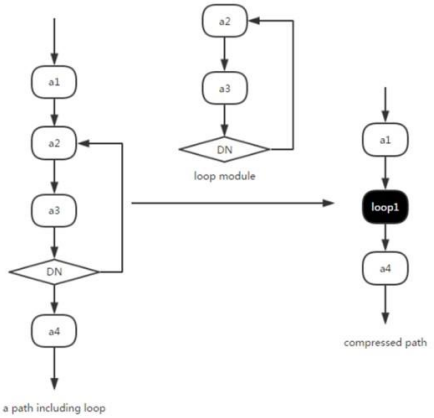

Figure 5. The transformation of loop module

5.2.3. Concurrent Module

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 Figure 6.

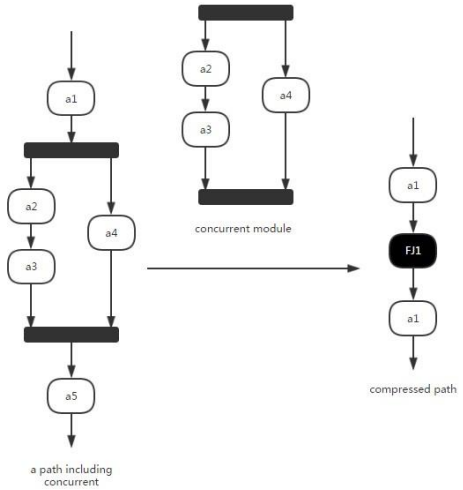

Figure 6. The transformation of concurrent module

Because concurrent modules have the possibility of nested concurrency, and the nested concurrent modules may be recognized by the program as serial concurrent modules during the testing phase. Therefore, we recommend that modelers use matching names to name the fork nodes and the join nodes in modeling phase, which helps improving the accuracy of matching fork nodes and join nodes. For example, the nested concurrent modules can be set to Fork0 (Fork1 Join1) Join0, so that the program can accurately identify the Fork0/Join0 concurrency module nested in the Fork1/Join1 concurrency module, during the test phase.

5.2.4. The Problem of Multiple Starting Nodes

For the multistarting node problem, this paper proposes a transformation algorithm for the problem, which treats n initial nodes as n-way concurrent paths. That is, to add a public parent fork node for these multiple starting nodes and create an initial node to point to the fork node.

Algorithm 1 Multi starting Transformation

Input: *AD* after compressed
Output: Well-transformed *AD*
Data: *N*, a set of nodes without in-degree, initially empty
n , the number of nodes without in-degree, initially 0

1. **for** each $node_i \in AD$ **do**
2. **if** $node_i$. *incoming* = *NULL* **then**
3. $N = N \cup \{ node_i \}$;
4. $n = n + 1$;
5. **if** *n==1* **then**
6. **return** *AD* ;
7. **else**
8. **new** a fork node;
9. **for** each $node_j \in N$ **do**
10. **new** an edge from fork node to $node_j$;
11. **new** an initial node;
12. **new** an edge from initial to fork;
13. **return** *AD* ;
14. **end**

The transformation algorithm is descripted in Algorithm1. This transformation algorithm can transform the activity diagram of the multiple-starting node into the activity diagram of the same structural semantics of the only one initial node. Figure 7 shows the result of activity diagram descripted in Figure 4 after transformation.

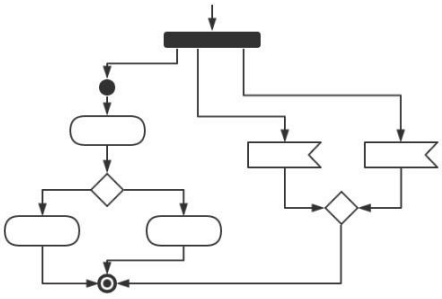

Figure 7. Transformed multiple starting nodes diagram

6. Automated Test Case Generation based on IBM

The test case generation based on IBM needs to process three types of modules: basic modules, concurrent modules and loop modules. The basic module is the activity diagram where it 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 contains the unique incoming edge and the unique outgoing edge.

6.1. Basic Module

The problem of generating a basic test path can be transformed into a problem of finding all paths between two points in the graph theory. The basic module test path generation algorithm is descripted in Algorithm 2.

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. All the test paths from the starting node to the end of the terminal node can be recorded without further leakage. This algorithm uses the stack, as the middle of the storage structure of test path. The test path is from the top of the stack to the bottom of the stack.

Algorithm 2 Multi starting Transformation

Input: *AD* after compressed
Output: A set of set scenarios
Data: *Scenarios*, a set of paths
Path, a stack of nodes

1. *Path*. push(*initial*);
2. make |*Path. top ( ). outgoingedge*| copies of path;
3. **for** each $node_i$ **do**
4. **if** edge (*Path. top( )*, $node_i$) **then**
5. fetch a *pathCopy*;
6. *pathCopy*. push($node_i$);
7. **if** $node_i$ = *final* **then**
8. *Scenarios* = *Scenarios* ∪ $node_i$;
9. **delete** *pathCopy*;
10. **if** |*pathCopy*| ≠ 0 **then**
11. jump to 2, dealing with each pathCopy separately
12. **else**
13. **return** *Scenarios*
14. **End**

6.2. Loop Module

For the loop module, we can use the loop module path generation algorithm and generate the test path automatically. For the compressed base path, the test path generation algorithm of the basic module can be applied. After the basic path is generated, replace the loop black box with the test path generated by the loop module.

Loops are an important and common part of the test case generation algorithm, which can usually be divided into simple loop, nested loop, and unstructured loop, as shown in Figure 8.

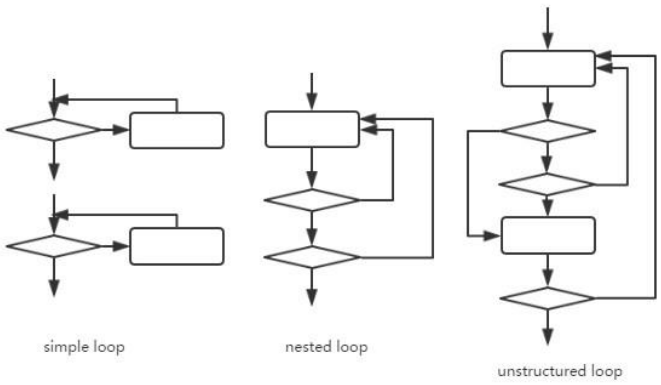

Figure 8. Classification of loop modules

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
- only once through the loop
- twice through the loop
- m times through the loop
- n-1, n, n + 1 times through the loop

For nested loop, if a simple loop test method is to be used for nested loops, the number of possible tests increases as the number of nesting layers increases. It results in an unrealistic number of tests. Here's a way to reduce the number of tests:

- Step 1. Set the cycles to the minimum from the innermost cycle.
- Step 2. Use a simple cycle test for the innermost cycle, leaving the outer cycle of the number of cycles to a minimum.
- Step 3. The test of the next loop is constructed from the inside to the outside, but the other outer layer loops to the minimum value, and the other nested loops are the typical values.
- Step 4. Continue until all the cycles have been tested.

Finally, for unconstructed loop, we require this type of loop redesigned as a structured loop structure as much as possible.

6.3. Concurrent Module

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.

In the previous chapter, we categorized the concurrent modules. For different kinds of concurrent modules, we propose different generation algorithm.

For simple concurrent, this is the classic concurrency module, which can be used fully arranged algorithm or other optimization algorithm for automatic test case generation.

For merge concurrent, which represents the semantics of the logical operation “OR”, an algorithm for generating the merge concurrent module test case is presented below:

- Step 1. Build a test path using a full arranged algorithm or other optimization algorithm.
- Step 2. When the algorithm copies all the nodes in one parallel stream to the test path being produced completely, this means that the control flow can flow to the merge node, and we insert the merge node into the test path.
- Step 3. Discard the active nodes in the other parallel streams.
- Step 4. Complete one test path generated from concurrency module.

For partJoin concurrent, a partial parallel flow is synchronized, and an algorithm for generating partJoin concurrent module test cases is presented below:

- Step 1. Build a test path using a full arranged algorithm or other optimization algorithm.
- Step 2. When the algorithm copies all the nodes in one parallel stream that need to be synchronized to the test path being produced, which means that the control flow can flow to the join node, we insert the join node into the test path.
- Step 3. If all nodes in the parallel stream that do not require synchronization are not fully copied to the test path being produced, the algorithm continues to apply until all nodes are copied.
- Step 4. Complete one test path generated from concurrency module.

For noJoin concurrent, the difficulty lies in the processing flow termination node and the activity termination, the following proposed an algorithm for generating the noJoin concurrent module test case:

- Step 1. Build a test path using a full arranged algorithm or other optimization algorithm.
- Step 2. If the algorithm copies all nodes in a parallel stream completely, including the active termination node, to the test path being produced, which means that the entire activity has been completed, the active nodes in the other parallel streams are discarded.
- Step 3. If the algorithm will complete all nodes in a parallel stream, only the stream termination node, which is copied to the test path being produced, means that the parallel stream is complete, but it does not represent the completion of the entire activity, then continue to apply the algorithm until the situation described in Step2 appears.
- Step 4. Complete one test path generated from concurrency module.

7. Implementation of Test Case Generation

The design goal of the system described in this chapter is to use the SysML activity diagram model as input, to process it, to generate the test case script according to the test coverage criterion and the test case generation algorithm, and output it to the software test group members. Testers can test the system automatically through test case scripts. The implementation framework is shown in Figure 9.

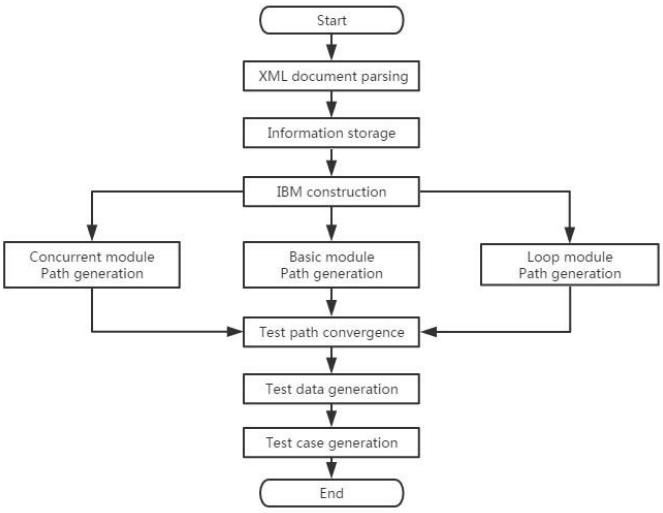

Figure 9. Implementation of test case generation

We use 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 EA and exported into XML format.

The second step is Information storage. We store the SysML activity diagram as a directed graph. Then we construct IBM. This is a process interacting with the user. Users can arrange the order to compress the black boxes. This is to improve the accuracy of black box.

The forth step includes three parts: concurrent module path generation, loop module path generation, basic module path generation. The implementation uses the above algorithm.

Then we combine these test paths. In this step, we replace black boxes with particular generated path. Finally, we generate test data and merge test path and test data into test case. We also output test case into XML format.

8. Case Study

There are two types of cases: (1) a motivating example of the process of the algorithm described in this article discussed in Section 8.1. and (2) an application case discussed in Section 8.2. The motivating example is a startup case. In this case, we're trying to show you the whole process of the project we're putting forward. And the application case is an example of a particular scene and it shows the process of generating video outputs of an in-car entertainment.

8.1. Motivating Example

The following Figure 10 is an unstructured SysML activity diagram model, which contains a concurrency module, a loop module and has a multiple start node problem, as defined earlier in this article.

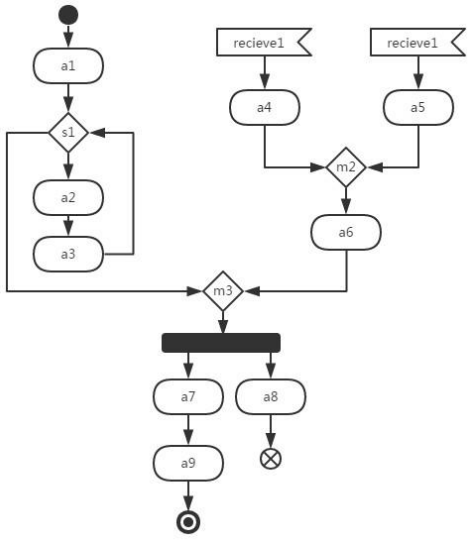

Figure 10. Motivating case

The Figure 11 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 IBM.

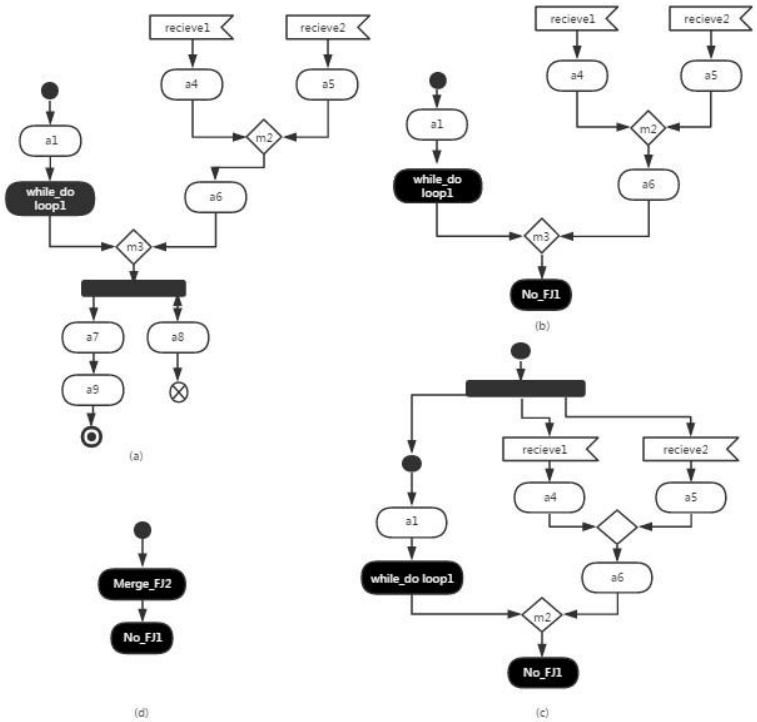

Figure 11. The process of transformation

The first step is to identify the loop module and compress it into a black box node while-do loop1, shown in Figure 11(a). The compressed black box node is the intermediate representation of the loop shown in the following Figure 12(a).

The second step is to identify the noJoin concurrency module and compress it into a black box node No FJ1, shown in Figure 11(b). The compressed black box node is shown in the following Figure 12(b).

The third step is to solve the multiple starting nodes. Call the Algorithm1 to reconstruct the activity graph, shown in Figure 11(c).

The fourth step, identify the merge concurrency module and compress it into a black box node Merge FJ2, shown in Figure 11(d). Wherein, the compressed black box node is the intermediate representation of the concurrency shown in the following Figure 12(c).

Figure10(d) is a compressed and structured SysML activity diagram that can be used to automatically generate test cases using Algorithm 2. For black box modules, we can use the algorithms described in the previous section to generate them. Finally, the black box module can be replaced.

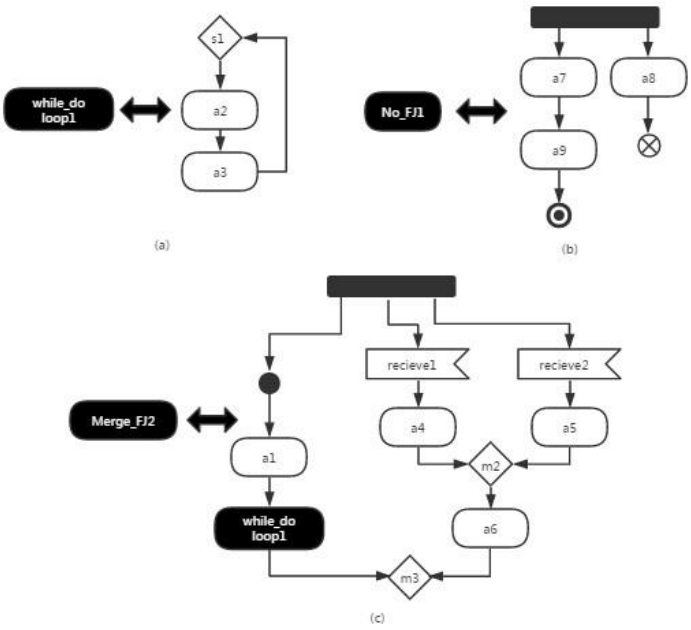

Figure 12. The map of black boxes

8.2. Application Case

Figure 13. shows the process of generating video outputs of an in-car entertainment. When the activity begins, it starts the action of Produce Test Signal and waits to receive the input signal. Then, it goes to action Process Frame. Finally, Encode MPEG and Convert to composite happens synchronously.

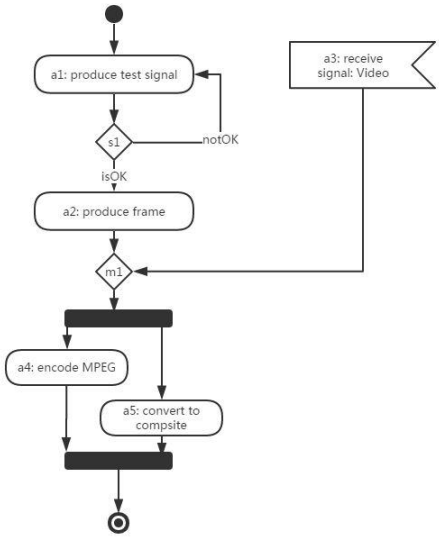

Figure 13. Application case

The generated test cases of Figure 13 are listed in Table 1. Table 1 has two columns: TestID and Test path. TestID

consists of the name of the diagram and the number of sequences that are generated automatically. Test path is a sequence ofactions from start to end. We set the cycle times of the loop to 1, 2, 3, respectively. Due to the space, we omit a part of test cases and the total number is 36.

Table 1. TEST CASES OF BASIC MODULE

TestID	Test path
Video 1	start, a1, s1, a2, m1, a4, a5, end
Video 2	start, a1, s1, a2, m1, a5, a4, end
Video 3	start, a3, m1, a4, a5, end
Video 4	start, a1, a3, m1, a4, a5, end
Video 5	start, a1, s1, a3, m1, a4, a5, end
Video 6	start, a3, m1, a5, a4, end
Video 7	start, a1, a3, m1, a5, a4, end
Video 8	start, a1, s1, a3, m1, a5, a4, end
Video 9	start, a1, s1, a1, s1, a2, m1, a4, a5, end
Video 10	start, a1, a1, s1, s1, a2, m1, a5, a4, end
:	:
:	:
Video 36	start, a1, s1, a1, s1, a1, s1, a2, m1, a5, a4, end

In our approach, we can transform these unstructured parts into an intermediate representation — IBM without losing its proper semantics. And we can generate test case with intermediate representation automatically. However, the process of transformation is not automated. Presently, our approach can provide the transformation service only when interacting with testing group members. It means that the identification of black box should be finished with interaction.

Another limitation is that our approach does not consider the test data too much. We generate the test path using IBM. For every generated test path, we need to use the test data generator to generate the input data and the expected output automatically. This allows the tester to analyze the potential errors in the program by comparing the actual output with the expected output. The design and implementation of the test data generator is also our next research focus.

9. Conclusions

In this article, we present a solution to generate test cases from unstructured activity diagrams. The core idea is to transform unstructured activities into the intermediate expression — IBM and use IBM for the generation of test cases. The algorithm we designed brings the possibility of automatic generation of test cases to solve unstructured activity graphs, and also improves the efficiency of test case generation.

For future work, we plan to design and implement a fully automated tool for test cases generation based on SysML activity diagrams. The first step is to design an automatic recognition algorithm for unstructured modules, instead of the existing semi-automatic identification process. Then, we will transfer our research to the generation of test data to make test cases more complete.

Acknowledgements

This work is supported by the National Natural Science Foundation of China (Grant No. 61370100, Grant No. 61321064 and Grant no.61402178), and Shanghai Knowledge Service Platform for Trustworthy Internet of Things (Grant No. ZF1213). The authors would like to thank the anonymous referees for their valuable comments and suggestions.

References

1. A. Albers and C. Zingel, “Challenges of Model-based Systems Engineering: A Study towards Unified Term Understanding and the State of Usage of Sysml,” in *CIRP Design Conference*, pp. 83–92, 2013.
2. Y. D. S. Almulham, “Automatic Test Case Generation from UML Activity Diagram using Activity Path,” 2010.
3. S. authors, “A Survey of Model-based Systems Engineering (MBSE) Methodologies,” 2013.
4. M. Chen, X. Qiu, and X. Li, “Automatic Test Case Generation for UML Activity Diagrams,” in *International Workshop on Automation of Software Test*, pp. 2–8, 2006.
5. Dori and Dov, “Model-Based Systems Engineering with OPM and SysML. Springer Publishing Company,” Incorporated, 2016.
6. 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.
7. G. G., “Generating Effective Test Suites by Combining Coverage Criteria,” in *Proceedings of the Symposium on Search-Based*

Software Engineering, 2017
8. A. K. Jena, S. K. Swain, and D. P. Mohapatra, "A Novel Approach for Test Case Generation from UML Activity Diagram," in *International Conference on Issues and Challenges in Intelligent Computing Techniques*, pp. 621–629, 2014.
9. 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*, pp. 430–435, 2016.
10. D. Kundu and D. Samanta, "A Novel Approach to Generate Test Cases from UML Activity Diagrams," *Journal of Object Technology*, vol. 8, no. 3, pp. 65–83, 2009.
11. 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.
12. J. Lasalle, F. Peureux, and F. Fondement, "Development of an Automated MBT Toolchain from UML/Sysml Models," *Innovations in Systems Software Engineering*, vol. 7, no. 4, pp. 247–256, 2011.
13. M. R. Lyu, "Handbook of Software Reliability Engineering." McGraw-Hill, Inc., 1996.
14. A. Nayak and D. Samanta, "Synthesis of Test Scenarios using UML Activity Diagrams," Springer-Verlag New York, Inc., 2011.
15. C. B. Nielsen, P. G. Larsen, J. Fitzgerald, J. Woodcock, and J. Peleska, "Systems of Systems Engineering: Basic Concepts Model-based Techniques and Research Directions," *Acm Computing Surveys*, vol. 48, no. 2, p. 18, 2015.
16. O. Oluwagbemi and H. Asmuni, "Automatic Generation of Test Cases from Activity Diagrams for UML based Testing (UBT)," vol. 77, no. 13, 2015.
17. J. I. Park and J. T. Choi, "Test Framework Development for Software Reliability Test using Formal Method," in *Information Technology and Computer Science*, pp. 116–119, 2016.
18. P. E. Patel and N. N. Patil, "Testcases Formation using UML Activity Diagram," in *International Conference on Communication Systems and Network Technologies*, pp. 884–889, 2013.
19. L. Wang, J. Yuan, X. Yu, J. Hu, X. Li, and G. Zheng, "Generating Test Cases from UML Activity Diagram based on Gray-box Method," in *Software Engineering Conference*, Asia-Pacific, pp. 284–291, 2004.
20. 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.
21. Z. Xinyu, "Research on Test Case Generation Method based on UML Activity Diagram Model," Ph.D. dissertation, Shan Dong University, 2007.