An Automatic Test Case Generation Method based on SysML Activity Diagram

Yiqun Xu, Linbo Wu

1 MoE Engineering Center for Software/Hardware Co-Design Technology and Application, East China Normal University, Shanghai 200062, China
2 Institute of Computer Application Research, Chinese Academy of Engineering Physics, Mianyang 621000, China
*Corresponding author e-mail address: xu_yiqun@163.com

Abstract. Model-Based Systems Engineering (MBSE) is a new method to engineer the modern large, complex systems. MBSE has a good guarantee for system reliability and it can improve the efficiency of system development. So, MBSE has received extensive attention from academia and industry. SysML is an auxiliary language for MBSE. SysML activity diagram can model the behavior of the system. In this paper, we use SysML activity diagram to generate test case. There are two main things in our work. One is that we have designed an automated SysML activity diagram test case generation method. Second, we develop a test case generation tool to support related applications in the industry.

1. Introduction
People prefer to use Model-Based Systems Engineering (MBSE) instead of traditional software engineering to engineer the modern large, complex systems. MBSE [1…3] applies the modeling method into the system engineering and uses the model to support activities such as system requirements, designs, analysis, verifications and confirmations. A MBSE methodology is used to support a series of related processes, methods and tools for complying with the laws of MBSE. Systems Modeling Language (SysML) is a support language to using MBSE. SysML [4, 5] is a graphical language defines semantics for the structure, behavior, requirements and parametric models of the system.

Model-Based Testing [6…8] is an important part of quality assurance in MBSE. As the complexity of system grow, more and more time and manpower are required for testing. Manual testing is so labor-intensive and error-prone that it is necessary to develop automatic testing techniques in some circumstance. In Model-Based Testing, test case derived from a model can detect bugs in deep paths and guarantee test case coverage requirements. So, Model-Based Testing is an effective way to improve test efficiency and reduce test costs.

A lot of researches have been done on Model-Based Testing and test case generation method. Chen [9] using a modified DFS (depth-first search) algorithm to generate a set of paths from the Activity Diagram according to the simple path coverage criteria. Then they obtained the minimum set of test cases by comparing the set of paths obtained in the previous step with the execution traces obtained by running the source program with random test cases. However, this method may not meet the test sufficiency requirement. Wang [10] proposed an approach based on Gray-Box method to generate test cases from UML Activity Diagrams. However, they gave no method to automatically generate test
cases from their extracted scenarios. Ashalatha [11] proposed a model called ITM. Activity diagrams can be transformed into ITM for testing. 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 activity diagrams with complex concurrent structures. Yin [12] proposed an algorithm to transform unstructured activity diagram into a model called IBM. Compared IBM with unstructured activity diagrams, IBM generates test case more convenient. However, in the process of model transformation, the semantics of the activity diagram changed, some scenarios have changed and may cause other problems.

In this paper, we design a method to generate test case automatically from SysML activity diagram. The previous research of Model-Based testing mainly focused on UML models and there is not a lot of research achievement on SysML models. However, SysML can be applied to modeling for complex systems, especially embedded systems. Therefore, we chose SysML as research field.

By summarizing the existing research results, we found that existing test case generation algorithms base on SysML or UML have some limitations. When the structure of the activity diagram is complex, these algorithms are not feasible, the generated test case set does not meet the sufficient requirement. Moreover, the existing research results are difficult to apply to practice, resulting in the lack of automated tools in the industry. In this paper, we aim at the automated test case generation method base on SysML activity diagram. This method automatically analyses the structure of the activity diagram and generate test case. The generated test case set guarantee the sufficient requirement. Based on this method, we have also developed a tool called SysML Activity Diagram Test Case Generation (SATCG) tool to support related applications in industry that can improve test efficiency.

The paper is organized as follow: In next section, we introduce SysML activity diagrams and present four test adequacy criteria for SysML activity diagram. In section 3, we design an algorithm to analyse the structure of activity diagrams automatically. In section 4, we introduce the test case generation method and two test path generation algorithms used in the method. In section 5, we introduce the tool SATCG and run a case to show the test case generation process. The related works and some conclusions are given in the last section.

2. Basic concepts and definitions
In this section we introduce the activity diagram formal definition and present three test adequacy criteria.

2.1. Activity diagram formal definition
Activity Diagram Formal Definition [4] can be represented as:

\[ AD = (\text{Node}, \text{Edge}) \]

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

\[ \text{Node} = \{ \text{InitialNode}; \text{FlowFinalNode}; \text{ActivityFinalNode}; \text{ActionNode}; \text{ActivityNode}; \text{ForkNode}; \text{JoinNode}; \text{DecisionNode}; \text{MergeNode}; \text{RecieveSignalNode}; \text{SendSignalNode} \} \]

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:

\[ \text{Edge} = \{ (x, y) | x, y \in \text{Node} \} \]
2.2. Test case formal definition

Test cases derived from activity diagrams are considered as an abstract test suite. A system under test can not executed abstract test suite directly. In activity diagram, the test case is a sequence of nodes form the InitialNode to the ActivityFinalNode, indicating the running process form the beginning to the end of the system. The test case is defined as follow:

\[ TC = (a'_1, a'_2, \ldots, a'_n) \]

\[ a'_i = (t_n, a_n), (i = 2, \ldots, n) \]

\[ t_n = a_j \rightarrow a_i, (j < i, i = 2, \ldots, n) \]

In this formula, \( a_i \in \text{Node} \), \( t_i \in \text{Edge} \).

2.3. Test adequacy criteria

As an essential part of any testing method, test adequacy criterion specifies the requirement of a particular software testing. The adequacy criteria of activity diagrams are based on the matching between the paths of activity diagrams and the program execution traces of the implementation codes. They mainly deal with the coverage of elements in activity diagrams.

The definition of test adequacy is given in [13] as a measurement function. To generate test cases for an activity diagram, we consider the following three test adequacy criteria:

- **Node Coverage** requires all nodes in the activity diagram be covered.
- **Edge Coverage** requires all edges in the activity diagram be covered.
- **Basic Path Coverage** requires all basic paths in the activity diagram be covered.

3. Activity diagram structure analyse

Due to the variety types of activity node types, we can model complex scenes, which leads to the complexity activity diagram structure. As shown in Figure 1, the activity diagram in Figure 1 has a complex structure. The conventional method treats the activity diagram as a directed graph. The test path generation algorithm is based on the Depth-First Search algorithm. Such a path generation algorithm has certain limitations: it can’t generate concurrent structure’s test case well. Our goal is to propose a generalized path generation method that can handle activity diagrams of many complex structures. Based on this goal, we must first fully analyse the structures of the activity diagram.

Figure 1. A SysML activity diagram.
3.1. Decision module
As shown in Figure 2, it is a decision module. There are two branches after DecisionNode, one branch satisfies the condition in DecisionNode and the other branch does not satisfy.

![Figure 2. Decision module.](image)

3.2. Loop module
As shown in Figure 3, the loop module is like a while loop statement in the programming language. The nodes in the loop module will execute multiple times.

![Figure 3. Loop module.](image)

3.3. Concurrent module
In SysML activity diagrams, there are four different concurrent structures. There is a fork-join structure in Figure 4(a). Fork-join means all concurrent threads must be executed and summarized before proceeding to the next activity node. The fork-merge structure is shown in Figure 4(b). For these concurrent threads in fork-merge structure, if one of the concurrent thread finishes executing, others thread will be interrupted. As long as one concurrent thread finishes executing, the activity diagram will continue to execute.

The third type is fork-flowfinal structure. As shown in Figure 4(c), one concurrent thread has a FlowFinalNode, this concurrent thread only has a concurrency relationship with other concurrent threads. The execution of the activity diagram is not affected by this thread.

The last type is multi-starting structure. As shown in Figure 4(d), there are two nodes with zero in-degree. These zero in-degree nodes are usually ReceiveSignalNode. These zero in-degree nodes have a concurrent relationship with other nodes in the activity diagram.
3.4. Activity diagram modular algorithm

We found that the basic structure in the activity diagram are decision module and loop module. We can use the basic structure to decompose the activity diagram. The decomposed activity diagram consists of several basic structural modules. To analyse the structure of activity diagram automatically, we designed an activity diagram module algorithm. The algorithm takes activity diagrams as input and output as a branch set.

We use the SysML model as input, then convert the model into a directed graph and we use adjacency lists to store the directed graph.

**Algorithm 1:** Modular SysML activity diagram

**Input:** AD

**Output:** BranchSet

1. for each nodeᵢ ∈ AD do
2.    nodeᵢ. IsModule = false;
3.    new branch, branch. father = AD, branch. level = 0;
4.    add all nodes to the branch that the InitialNode can reach, nodeᵢ. IsModule = true;
5.    BranchSet = BranchSet ∪ {branch};
6. for each nodeᵢ ∈ AD do
7.    if (nodeᵢ. IsModule == false && nodeᵢ. pre == null) then
8.        new branch1, branch1. father = AD, branch1. level = 0;
9.        branch1.add(nodeᵢ); nodeᵢ. IsModule = true;
10.     for each nodeᵢ ∈ AD do
11.        if (nodeᵢ can reach nodeᵢ && nodeᵢ. IsModule = false) then
12.            branch1.add(nodeᵢ); nodeᵢ. IsModule = true;
13.            BranchSet = BranchSet ∪ {branch1};
14.     for each branchᵢ ∈ BranchSet do
15.        FindSubBranch(branchᵢ);
16.    End;

**Algorithm 2:** Find sub-branches in lists

**Input:** branch

**Output:** BranchSet

1. Procedure: FindSubBranch(branch)
2. for each nodeᵢ ∈ branch do
3.    if (nodeᵢ. type = ForkNode) then
4.        new n subBranch, add n concurrent threads in n subBranch;
5.        subBranch. father = branch, subBranch. level = branch. level + 1;
6.        for each subBranchᵢ do
7.            BranchSet = BranchSet ∪ {subBranchᵢ};
8. FindSubBranch(subBranch_i);
9. End;

We use Algorithm 1 and Algorithm 2 to analyse the AD’s structure. Algorithm 1 puts the nodes that InitialNode and the InitialNode can reach into a branch, then searches for other zero in-degree nodes. Next, Algorithm 1 puts a zero in-degree node into a new branch until all zero in-degree nodes in activity diagram are added to the branches. But the BranchSet we get from Algorithm 1 may not be the final result, due to the branch may contain concurrent structures. Algorithm 2 will be called to find the sub-branch until the branch cannot be modular, so Algorithm 2 is an algorithm of recursive structure. When we create a new branch, we will record the father branch and branch level of the branch. If the branch levels of the two branches are equal and the parent branches are the same, then two branches have a concurrent relationship.

We use Algorithm 1 and Algorithm to analyse the activity diagram in Figure 1. The results is shown in Table 1.

| Branch name | Father branch | Branch level | Activities in branch |
|-------------|---------------|--------------|----------------------|
| branch1     | AD            | 0            | A1, A2, A3, A4, A5, A6 |
| branch2     | AD            | 0            | A7                   |
| branch3     | branch1       | 1            | A1, A2, A3, A4       |
| branch4     | branch1       | 1            | A5, A6               |
| branch5     | branch3       | 2            | A2, A3               |
| branch6     | branch3       | 2            | A4                   |

We can get 6 branches, through the father branch and branch level, we can find branches with concurrent relationship. If the parent branches of the two branches are the same and the branch levels are equal, then the two branches have concurrent relationship. In this activity diagram, there are three pairs of branches with concurrent relationships: branch1 and branch2, branch3 and branch4, branch5 and branch6. The concurrency types of these three pairs are multi-starting, fork-flowfinal, fork-join.

4. Test case generation
In this section, we will detail the steps to generate the activity diagram test path, then introduce the test path algorithm of basic structure and concurrent structure.

4.1. Test case generation method
We proposed an automatic test case generation method. Test case set we get from activity diagram meet the test sufficiency requirements. This method describes the specific process of test case generation. The process can be divided into the following steps:

- **Step 1.** We use Algorithm 1 and Algorithm 2 to analysis the structure of input activity diagram, we can get several modularized branches.
- **Step 2.** Save modularized branches to two sets, set 1 store branches without child branches and set 2 store branches with child branches.
- **Step 3.** Use basic structure test path generation algorithm to generate test paths for branches in set 1.
- **Step 3.** Use concurrent structure test path generation algorithm to generate test paths for branches in set 1.
- **Step 4.** For a branch in set 2, if the test path for all child branches of a branch has been generated, then generate test path for this branch.
- **Step 5.** In set 2, generate test path for branches with concurrent relationship.
• **Step 6.** Jump to **Step 4** until the test path for all branches is generated.
• **Step 7.** Get the test case for input activity diagram.

The basic structure test path generation algorithm and the concurrent structure test path generation algorithm are used in the process of generating test case. Through the result of the modularization of the activity diagram, we can know the relationship between the branches and use the corresponding test path generation algorithm to generate the test path of the branch. Next content we will detail the two test path generation algorithms.

4.2. Basic structure test path generation
Algorithm 3 generates a basic structure branch test path. It is based on Depth-first search algorithm. This algorithm can generate test paths for a branch, which can be a branch containing one or more decision modules and loop modules.

**Algorithm 3 Basic structure test path generation**

**Input:** branch, loopCount

**Output:** PathSet

1. `new` path, path.add(branch.get(0));
2. `for` each nodeᵢ ∈ branch do
3.   `if` (nodeᵢ.next ∉ branch) then
4.     branchEnd = nodeᵢ;
5.     BasicTestPathGeneration(branch.get(0));
6. `Procedure:` BasicTestPathGeneration(node)
7.   `if` (node == branchEnd) then
8.     PathSet = PathSet U {path};
9.     `return`;
10. `while`(node.next != null) do
11.   `if` (node.count < C(node)) then
12.     node.count ++;
13.     path.add(node.next);
14.     BasicTestPathGeneration(node.next);
15.     path.remove(node.next);
16. `endif`;
17.   node = node.next;
18. `End`;

In Algorithm 3, loopCount is a list that stores the number of node traversal in the loop module. When the number of traversals of a node is specified, the number of entries of the loop module in which the node is located is limited. Usually, you only need to specify the number of traversals of one node in the loop module. If there are multiple loop modules in a branch, each loop module needs to specify the number of loop module entries. C(node) is a function that finds the number of traversals that the node is specified for. If the number of traversals for node is not specified, the function returns a value of ∞.

4.3. Concurrent structure test path generation
Algorithm 4 is used for generating test paths test path in concurrent structure. The algorithm is based on the idea of permutation, it can generate test paths for the four concurrent structures summarized above. The input to the algorithm consists of two paths, path1 and path2 and a concurrent type cType.

**Algorithm 4 Concurrent structure test path generation**

**Input:** path1, path2, cType
Output: concurrentPathSet
1. nodeA = path1.get(0), nodeB = path2.get(0);
2. P = ∅;
3. concurrentPathGeneration(P);

4. Procedure: concurrentPathGeneration(P)
5. if (nodeA == null) then
6. if (cType == fork − join
   || cType == fork − flowfinal && path1Final == FlowFinalNode) then
7. concurrentAction(nodeB);
8. concurrentPathSet = concurrentPathSet ∪ {P} ;
9. if (nodeB == null) then
10. if (cType == fork − join
    || cType == fork − flowfinal && path2Final == FlowFinalNode) then
11. concurrentAction(nodeA);
12. concurrentPathSet = concurrentPathSet ∪ {P} ;
13. path.add(nodeA), nodeA = nodeA.next;
14. concurrentPathGeneration(P);
15. path.add(nodeB), nodeB = nodeB.next;
16. concurrentPathGeneration(P);

17. Procedure: concurrentAction(node)
18. while(node != null) do
19. path.add(node);
20. node = node.next;
21. End;

In Algorithm 4, the generated case paths are saved in concurrentPathSet. nodeA.next represents the next node of nodeA in path1. path1Final is the last node in path1, path2Final is the last node in path2. Procedure concurrentPathGeneration is a program that generates concurrent test paths for path1 and path2. Lines 5 and 9 in the algorithm are used to determine if path1 and path2 have been traversed. If the concurrent type is fork-merge, then the algorithm only needs to complete the traversal of one of path1 and path2. If the concurrent type is fork-join, both path1 and path2 need to complete the traversal. If the concurrent type is fork-flowfinal and the algorithm has completed the traversal of one path, if the path contains the FlowFinalNode, then the algorithm needs to complete the traversal of another path. If the concurrent type is multi-starting, then the algorithm only needs to complete the traversal of one of path1 and path2.

5. An automatic tool for test case generation
In the previous section, we proposed a test case generation method based on SysML model. In this section, we introduce the tool SATCG and run a case to show the test case generation process.

5.1. Tool introduction
SATCG is developed in Java. It takes SysML activity diagrams as input. The tool consists of the following modules: activity diagram pretreatment, test path generation, test path reduction, test data generation and test case generation.

   Tool interface is shown in Figure 5.
5.1.1. **Activity diagram pretreatment.** The input to the tool is the SYSML activity diagram. The activity diagram model is saved in a file in XML format. The information of the activity diagram can be obtained by parsing the XML file. Then, the activity diagram is converted into a directed graph and saved in adjacency lists. Finally, use activity diagram modular algorithm to analyse the structure of the activity diagram.

5.1.2. **Test case generation.** This module implements the test case generation method we proposed in Section 4. Before generating the test case, we need to limit the number of entries of the activity diagram loop module to ensure that the program runs correctly.

5.1.3. **Test case reduction.** When generate test cases for activity diagrams that contain concurrent structures. Due to the complexity of the scenes in the concurrent structure, we get a lot of test cases. This is a challenge for tester who use all test cases for testing. So, we provide three test case reduction methods to help tester select the test case in a specific scenario for testing.

- **Must execution node:** Specifies that one or more nodes must exist in the test case.
- **Adjacent nodes:** Specifies that one or more pairs of nodes are adjacent in the test case.
- **Priority nodes:** Specifies the priority relationship of one or more pairs of nodes. In the test case, the node with the higher priority is executed before the node with the lower priority.

Through these three methods, the tester can select test cases in a specific scenario to test, the obtained test cases can meet the test sufficiency requirements in the scenario.

5.1.4. **Test data generation.** After getting the test case, we generate the test data for each test case. In the activity diagram, only the data in the DecisionNode can affect the execution of the activity diagram. Therefore, the generated test data is the data in the DecisionNode that satisfies the test case.

5.1.5. **Test script generation.** Based on the test case generated, this module can generate executable test script in a specific format. It improves the automated test rate, testers do not need to write test scripts, which improves the test efficiency of testers.
5.2. Case study

We use the activity diagram shown in Figure 1 to generate test cases. The first step is to pretreatment the model. We can get several branches. The information on each branch is shown in Table 1.

When we generate the test path for branch4 in Table 1, we need to specify the number of entries for the loop module in branch4, in this case we set this number to 1. Continue the steps based on the test case generation method. We can get 264 test cases shown in Table 2. These 264 test cases contain all the test scenarios for the activity diagram. This also shows that the tool can run normally and generate test cases.

| ID | Test case         |
|----|-------------------|
| 1  | A7                |
| 2  | A1,A7             |
| 3  | A1,A2,A7          |
| ...|                   |
| 63 | A1,A5,A2,A6,A6,A7 |
| 64 | A1,A5,A2,A6,A6,A3,A7 |
| 65 | A1,A5,A2,A6,A6,A3,A4 |
| ...|                   |
| 260| A5,A6,A1,A6,A4,A2,A7 |
| 261| A5,A6,A1,A6,A4,A2,A3 |
| 262| A5,A6,A1,A4,A7    |
| 263| A5,A6,A1,A4,A2,A7 |
| 264| A5,A6,A1,A4,A2,A3 |

6. Conclusion

In this paper, we propose an automatic test case generation method based on SysML activity diagram. This method can generate test case for activity diagram with complex structure, the generated test case set meet the test sufficiency requirements. Based on our proposed method, we developed a tool called SATCG, it can generate test cases for activity diagrams.

For future work, we plan to study the reduction method of test case set due to the number of test cases generated by concurrent structures is very large. A large number of test cases will increase test costs, so a rigorous test case reduction method is necessary.

Acknowledgements

This work is supported by the Defense Industrial Technology Development Program JCKY 2016212B004-2. Thanks to Professor Chen and Professor Bu for their help in the paper. The authors would like to thank the anonymous referees for their valuable comments and suggestions.

References

[1] Wymore, Wayne A 1993 Model-based systems engineering Proceedings of the Spie 9150(1) 101-111
[2] Selic Bran 2008 Personal reflections on automation, programming culture, and model-based software engineering Automated Software Engineering 15(3-4) 379-391
[3] Woodcock J 2015 Systems of systems engineering: basic concepts, model-based techniques, and research directions Acm Computing Surveys 48(2) 1-41
[4] Friedenthal S, Moore A, Steiner R 2011 A practical guide to sysml San Francisco Jung Institute Library Journal 17(1) 41-46
[5] Carneiro E, Maciel P, Callou G, Tavares E and Nogueira B 2008 Mapping SysML State Machine Diagram to Time Petri Net for Analysis and Verification of Embedded Real-Time Systems with Energy Constraints International Conference on Advances in Electronics & Micro-electronics
[6] Dalal S R, Jain A, Karunanithi N, Leaton J M, Lott C M and Patton G C 1999 Model-based testing in practice International Conference on Software Engineering
[7] Heckel R, Lohmann M 2003 Towards model-driven testing Electronic Notes in Theoretical Computer Science 82(6) 33-43
[8] Pretschner A, Prenninger W, Wagner S, Kjanel C, Baumgartner M and Sostawa B 2005 One evaluation of model-based testing and its automation International Conference on Software Engineering
[9] Mingsong C, Xiaokang Q, Xuandong L 2006 Automatic test case generation for UML activity diagrams International Workshop on Automation of Software Test
[10] Linzhang W, Jiesong Y, Xiaofeng Y 2004 Generating test cases from UML activity diagram based on Gray-box method 11th Asia-Pacific Software Engineering Conference
[11] Nayak A, Samanta D 2011 Synthesis of test scenarios using uml activity diagrams Software and Systems Modeling 10(1) 63-89
[12] Yin Y, Xu Y, Miao W and Chen Y 2017 An automated test case generation approach based on activity diagrams of sysml International Journal of Performability Engineering 13(6) 922-936
[13] Zhu H, Hall P A V, May J H R 1997 Software unit test coverage and adequacy ACM Computing Surveys 29(4) 366-427