

2nd International Conference on Computer Science and Computational Intelligence 2017, ICCSCI  
2017, 13-14 October 2017, Bali, Indonesia

## Automated Test Case Generation from UML Activity Diagram and Sequence Diagram using Depth First Search Algorithm

<sup>ab</sup>Meiliana\*, <sup>a</sup>Irwandhi Septian, <sup>a</sup>Ricky Setiawan Alianto, <sup>a</sup>Daniel, <sup>b</sup>Ford Lumban Gaol

<sup>a</sup>*Computer Science Department, School of Computer Science - Bina Nusantara University, Jl. K. H. Syahdan No. 9, DKI Jakarta, 11480, Indonesia*

<sup>b</sup>*Doctor of Computer Science – Bina Nusantara University, Jl. Kebon Jeruk Raya No.27, DKI Jakarta, 11530, Indonesia*

---

### Abstract

Software testing is an important and critical activity in software development that deals with software quality. However, the testing process is consuming activities that need to be automated to save a lot of resources. Towards automated testing, automating test cases generation as the first testing process is being highlighted. This research aims to generate test case automatically from UML diagram since model based testing that conducted on early phase of software development process show higher efficiency. UML diagrams used in this research are activity diagram, sequence diagram and SYTG as the combination graph. These three diagrams have been proved as the most compatible diagram to generate test case from previous research. Method proposed in this paper is Depth First Search algorithm that is modified to generate expected test cases. This paper proves that modified DFS algorithm applied to generate test case is provide accurate result, every node presented on the test case, include any condition (alt and opt). Comparison result from three different test cases generated shows that test cases from combined UML may not necessarily result in better test cases, due to the possibility of redundant test cases for some test cases. This paper also presenting an experiment result that proving sequence diagrams can produce better test cases.

© 2017 The Authors. Published by Elsevier B.V.

Peer-review under responsibility of the scientific committee of the 2nd International Conference on Computer Science and Computational Intelligence 2017.

**Keywords:** test cases, depth first search algorithm, UML diagram, software testing, test cases generator

---

---

\* Corresponding author. Tel.: +6221-534-5830 ext. 2188.  
E-mail address: [meiliana@binus.edu](mailto:meiliana@binus.edu)

## 1. Introduction

Software testing is an important and critical phase that deals with software quality. However, software testing that consists of three phases (test case generation, test execution and test evaluation) <sup>1</sup> is time consuming activity that requires a lot of resource. Therefore, automated testing is strived to save resource spent in the terms of time, cost and effort and to give more accurate result than manual testing that vulnerable to human error. Towards automated testing, automating test cases generation as the first testing process is being highlighted.

Test cases can be generated automatically from source code or visual software model such as Unified Modeling Language (UML), Data Flow Diagram (DFD), or Entity Relationship Diagram (ERD). Research example about code based testing conducted by Srivastaval, et al. used genetic algorithm to optimize test case generation by applying conditional coverage on source code <sup>2</sup>. Another research from Alazzam et al. used information retrieval techniques for the automatic extraction of source code concepts for the purpose of test case reduction <sup>3</sup>. Compare to code based testing, model based testing where test cases are generated from model of the software showed higher efficiency of time and effort. Furthermore, generating test cases in the early phase of software development life cycle provide control management on construction and testing phase. Thus, this research will focus on generating test cases from several UML diagrams that are widely used on software modeling process.

Some previous researches have conducted test cases generation from UML diagrams. However, various methods used and different case provided by previous researchers lead to unclear comparison and evaluation about this field. This research provides one scenario case in different UML diagrams to be used in test cases generation process. As a preliminary work, two UML behavior diagrams which are activity diagram and sequence diagram will be used. An activity diagram can figure the sequential flow of activities of a use-case or business process from the start to the end activity and it can also be used to model logic with system. On the other hand, a sequence diagram can show more detail process about how processes interact with one another and the order of the interaction and indicate the life spans of objects relative to those messages. One additional graph as combination from activity and sequence diagram is formed and used as well for test cases generation in this research.

Modified DFS algorithm is proposed in this research as an enhancement of research from Tripathy et al.<sup>4</sup>. In our experiment, current DFS algorithm that applied for test case generation process generated some redundant node. Thus, a modification is needed to get optimal test cases result. Comparison test cases result for both aforementioned algorithms is provided in the fifth section. The second section will discuss about state of the art of this field. Subsequent section describes our proposed approach in generating test case automatically. Conclusions are given in the last section. This paper aims to provide better evaluation and comparison of test case generated from different UML diagrams with new proposed method; which is modified Deep First Search (DFS) algorithm.

## 2. State of The Art

Test cases are defined as a set of condition or variables which determine the level of correctness and quality of the product. Simple way to present test case is by providing test path to be followed when conducting a testing. The studied literature shows there are various methods described by numerous researchers for generating test cases and comparing test case from different UML diagrams. We have classified the literature according to different aspects of testing from UML design using different UML diagrams.

A.V.K. Shanthi and G. MohanKumar <sup>1</sup> presented an approach to automated generate test path using TABU search algorithm. In this paper, the activity diagram generated from software design, and then all possible information extracting using write parser in java. Based on the extracted information, an Activity Dependency Table (ADT) is generated. Test case is generated with the help of ADT by applying TABU search algorithm. This experiment show that this method has better performance. All possible test cases are generated and validated by prioritization. This approach can reveal all paths for software to be developed and also obtained test cases valid once. Similarly, some approaches <sup>5-9</sup> also used single diagram to generate test cases.

Syed Asad Ali Shah, Raja Khaim Shahzad, Syed Shafique Ali Bukhari and Mamoon Humayun<sup>10</sup> presented automated test case generation using UML class and sequence diagram. UML class and sequence is being converted into XML format using Visual Paradigm. C# code is used to read XML file. This experiment only simple tool that can generate automated test cases using class and sequence diagram without any intermediate form.

Namita Khurana, R.S Chillar<sup>11</sup> in this approach, state chart diagram being converted to State Chart graph and sequence diagram being converted to sequence graph. State chart graph and sequence graph is being converted into a graph called System Testing Graph (SYTG). Genetic algorithm is being applied to generate and optimize the test case based on their criteria. But, in this approach is used UML diagram state chart and sequence diagram and not clearly mentioned the comparison of the result.

Our related work is automated test case from UML diagram and comparing the difference in results - Abinash Triparthy and Anirban Mitra<sup>4</sup> presented an approach to generate test case from UML activity diagram and sequence diagram. In this paper, the activity diagram is being converted into activity graph and the sequence diagram is being converted into sequence graph, and then the combined graphs are integrated to form System Testing Graph (SYTG), and SYTG being traversed to form the test cases by using Depth First Search (DFS) method. But in this approach, uses a SYTG algorithm that combines activity graph with a sequence of graphs that is by checking each node has more than one branch or not, when branches are more than one then the first node of the sequence node included could be the branching is not suitable. Therefore, in this experiment DFS algorithm will be being modified eventually the results will be compared.

### 3. Proposed Approach

In our proposed approach, activity diagram (AD) will be converted into activity diagram graph (ADG) and sequence diagram (SD) to sequence diagram graph (SDG). After that we present the modified DFS to generate the test case from each graph, the test case from each graph is generated. On the next step, a graph called System Testing Graph (SYTG) is formed by combine the activity diagram graph and sequence diagram graph. The necessary information to form the test cases is pre stored into this graph. Then we generate the testing base on that System Testing Graph. The modified DFS works by first traversed the graph to find every last node in the graph, than stored those node in stack for last node. After that the graph is traversed again from the start to every last node that is listed in stack that store last node, every time the algorithm find the node that have more than one branch, that node is stored in stack for multiple branch. When we find the last node than the current path is stored in stack for path and change the current path to next path until every last node is visited, for detail information can be seen below.

#### 3.1. Conversion of AD into ADG

In this sub section, we first define the activity diagram. After that we present the technique to transform the activity diagram to activity diagram graph. An activity diagram figures the sequential flow of activities of a use-case or business process and it can also be used to model logic with system.

We form the graph by getting the information from activity diagram to form the node contain id, activity name, activity target, activity status. ID is a unique number used to identify each activity, activity name is the name of each activity, activity target is the next activity that we would use to make adjacency list, and the activity status is the status after pass the decision component.

#### 3.2. Conversion of SD into SDG

In this section, we first define the sequence diagram. After that we present the technique to transform sequence diagram to sequence diagram graph. A sequence diagram figures how objects communicate with each other in terms of a sequence of messages within a system.

We form the graph by getting the information from sequence diagram to form the node contain id, message name, message target, message number, operand status. ID is a unique number used to identify each message, message name is the name of each message, message target is the next message that we would use to make

adjacency list, message number is the sequence number of each message that we would use to reorder every message, and the operand status is the status of what alternative the message in.

### 3.3. Form the System Testing Graph (SYTG)

After we form the activity diagram graph and sequence diagram graph, the next step is to form the System Testing Graph (SYTG) by combine the activity diagram graph and sequence diagram graph.

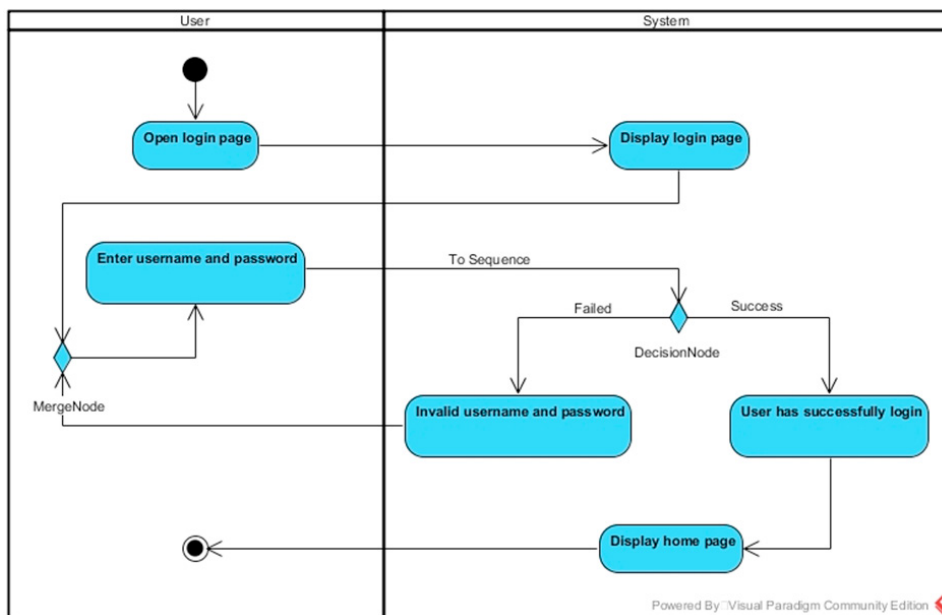


Fig. 1. Activity diagram

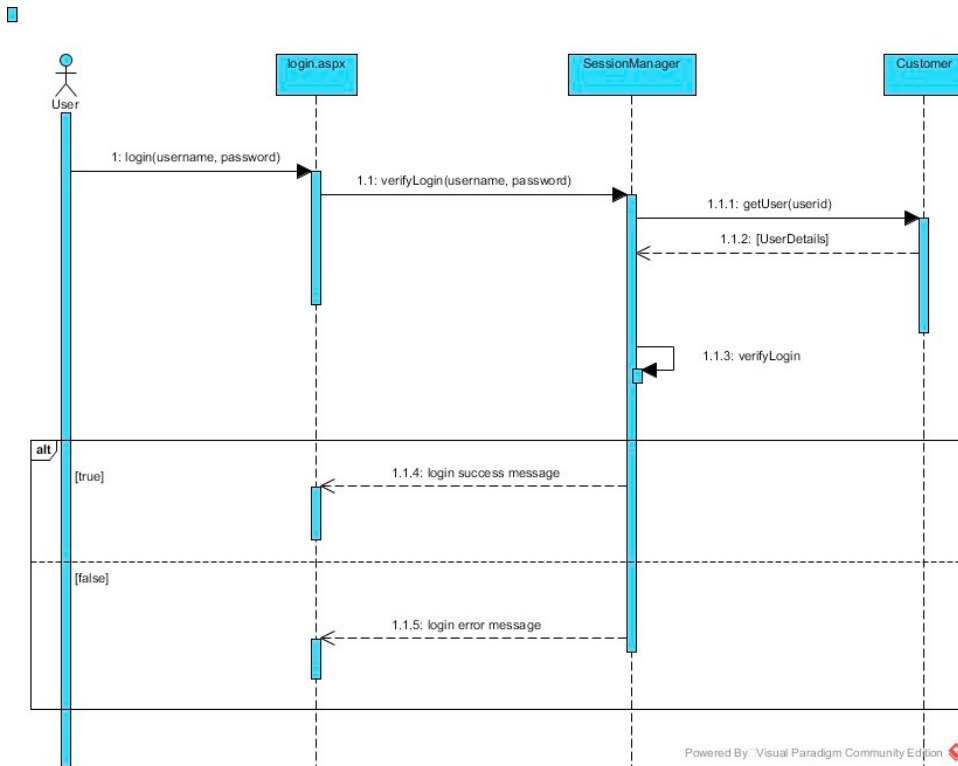


Fig. 2. Sequence diagram

**algorithm** SYTG is

**input:** Graph activity and sequence

**output:** List of combined graphs (LN)

```

LN ← activity_graph
foreach key, value in LN:
    if LN[key].adjacency_list length > 1 and LN[key].status = "To Sequence"
        curr_adj ← value.adjacency_list
        first_SDG ← SDG first key
        LN[key] ← first_SDG
    exit
foreach key, value in SDG:
    LN[key] ← value
foreach key, value in LN:
    if value.adjacency_list length = 0
        if value.status is not null and value.status = true
            add curr_adj[1] to LN[key].adjacency_list
        elseif value.status is not null and value.status = false
            add curr_adj[0] to LN[key].adjacency_list
        end
    end
end
return LN
  
```

### 3.4. Generation test cases

After we form the activity diagram graph, sequence diagram graph and system testing graph, the next step is to generate the test cases. We will generate the test case for each graph.

**algorithm** DFS\_Get\_Last\_Node is

**input:** Activity, sequence and SYTG graph

**output:** List of last node in graph

```

function DFSUtil(node, stack, visited, last_node):
    if visited[node] = 0 and stack length ≠ 0
        visited[node] ← 1
        pop stack
        foreach key, value in Graph[node].adjacency_list:
            if visited[value] = 0
                add value to stack
            else
                add node to last_node
        if Graph[node].adjacency_list length = 0
            add node to last_node
        if stack length ≠ 0
            last_stack ← end stack
            DFSUtil(last_stack, stack, visited, last_node)

node ← start_node
stack ← array
visited ← array
last_node ← array
foreach key, value in Graph
    visited[key] ← 0
add node to stack
DFSUtil(node, stack, visited, last_node)
return last_node

```

**algorithm** DFS\_Mod is

**input:** Activity, sequence and SYTG graph

**output:** Print of path

```

function GetFlowNode(last_node, stack, flow_stack, visited)
    foreach key, value in flow_stack
        print Graph[value].no
        if value ≠ end flow_stack
            print " => "
    first_node ← flow_stack[0]
    length_of_flow ← flow_stack length
    i ← length_of_flow
    while length_of_flow ≠ 0:
        temp_adj = Graph[flow_stack[i]].adjacency_list
        if temp_adj > 1
            foreach value in temp_adj:
                if visited[value] = 0
                    j ← i
                    while j ≠ 0:
                        visited[flow_stack[j]] ← 0
                        j ← j - 1
                    flow_stack ← array
                    DFSUtil(first_node, last_node, stack, flow_stack, visited)
        i ← i - 1

function DFSUtil(node, last_node, stack, flow_stack, visited):
    if visited[node] = 0 and stack length ≠ 0
        visited[node] ← 1
        pop stack
        add node to flow_stack
        if node exists in last_node
            GetFlowNode(last_node, stack, flow_stack, visited)
        foreach key, value in Graph[node].adjacency_list:
            if visited[value] = 0
                add value to stack
        if stack length ≠ 0
            last_stack ← end stack

```

[Ketik di sini]

```

DFSUtil(last_stack, last_node, stack, flow_stack, visited)

node = start
stack = array
flow_stack = array
visited = array
foreach key, value in Graph
    visited[key] ← 0
last_node ← DFS_Get_Last_Node
add node to stack
DFSUtil(node, last_node, stack, flow_stack, visited)

```

#### 4. Case Study

This section provides an experiment result of test case generation from activity graph, sequence graph, and SYTQ from one example case which is login case (fig 1. And fig 2.). Test cases provided in a simple path flow to give simple description of our algorithm implementation and better test cases comparison between graphs. Comparison result between DFS algorithm and the modification clearly show that DFS algorithm will combine all nodes in graph into one test case despite of any branch exist on the graph. A branch should leads to a new test case generation possibility in testing process. DFS algorithm is modified and generates appropriate test cases as show below.

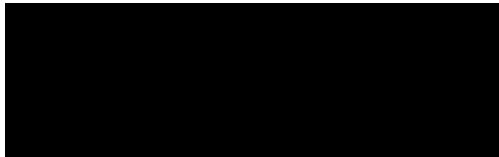


Fig. 3. Activity graph



Fig. 4. Sequence graph

##### 4.1. Activity Graph

Test cases result from DFS algorithm:

A1 => A2 => A3 => A4 => A6 => A7 => A8 => A5

Detail: Start => Open login page => Display login page => Enter username and password => User has successfully login => Display home page => Finish => Invalid username and password.

Test cases result from modified DFS algorithm:

a. Success case:

A1 => A2 => A3 => A4 => A6 => A7 => A8.

Detail: Start => Open login page => Display login page => Enter username and password => User has successfully login => Display home page => Finish.

b. Failed case:

A1 => A2 => A3 => A4 => A5.

Detail: Start => Open login page => Display login page => Enter username and password => Invalid username and password.

This diagram represents the flow of user's activity from the beginning to the end, but the generated test cases can only show the general outline in a system. These test cases used common words which make this easier to read and understand what the system will do by non-expert. However, the test cases can not show the detail process in system and the information obtained from activity diagram is difficult to process because there are no common rules d used in this diagram.

##### 4.2. Sequence Graph

Test cases result from DFS algorithm:

S1 => S2 => S3 => S4 => S5 => S7 => S6

Detail: Login (username, password) => Verify login (username, password) => Get user (userid) => [user details] => Verify login => Login error message => Login success message.

Test cases result from modified DFS algorithm:

a. Success case:

S1 => S2 => S3 => S4 => S5 => S6.

Detail: Login (username, password) => Verify login (username, password) => Get user (userid) => [user details] => Verify login => Login success message.

b. Failed case:

S1 => S2 => S3 => S4 => S5 => S7.

Detail: Login (username, password) => Verify login (username, password) => Get user (userid) => [user details] => Verify login => Login error message.

This diagram represents the detail of methods called and executed on the system, which make the generated test cases more detailed and specific. Based on that information the test cases can show more detailed information such as what the tester need to fill in the application during the testing process.

#### 4.3. SYTG Graph:

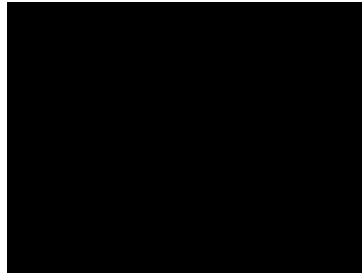


Fig. 5. SYTG graph

Test cases result from DFS algorithm:

A1 => A2 => A3 => A4 => S1=> S2=>S3 => S4 => S5 => S7 => A5=> LOOP

Detail: Start => Open login page => Display login page => Enter username and password => Login (username, password) => Verify login (username, password) => Get user (userid) => [user details] => Verify login => Login error message => Verify login => LOOP

a. Success case:

A1 => A2 => A3 => A4 => S1 => S2 => S3 => S4 => S5 => S6 => A6 => A7 => A8.

Detail: Start => Open login page => Display login page => Enter username and password => Login (username, password) => Verify login (username, password) => Get user (userid) => [user details] => Verify login => Login success message => User has successfully login => Display home page => Finish.

b. Failed case:

A1 => A2 => A3 => A4 => S1 => S2 => S3 => S4 => S5 => S7 => A5.

Detail: Start => Open login page => Display login page => Enter username and password => Login (username, password) => Verify login (username, password) => Get user (userid) => [user details] => Verify login => Login error message => Invalid username and password.

SYTG formed by combine the activity diagram graph and sequence diagram graph, that make the generated test cases more detail and coverage general outline in a system that user do from the beginning to the end however this test cases also contained redundant information, such as in this case it is show 'success/failed notification' two times.

## 5. Conclusion and Future Work

Nowadays software testing is a mandatory process to ensure the created software is corresponding with business process requirements. In this paper we presented modified DFS algorithm to generated automatic test cases using UML activity and sequence diagram. Our study shows DFS algorithm needs to be modified as in our proposed approach to generate appropriated test cases. From the experiment result, the test cases generated based on activity diagram, sequence diagram, and SYTG (combination graph) are provided. Test cases that are generated base on activity diagram coverage all activity from user to system however the test cases can not show the detail process in system such as what the tester need to fill in the application during the testing process and the information obtained



from this diagram is difficult to process because there is no standard words used in this diagram, while test cases that generated base on sequence diagram only coverage the system section but the generated test cases show the details of business process within a system and base on that information the test cases can show more detailed information, such as what the tester need to fill in the application during the testing process and the test cases that generated base on combined graph between activity and sequence diagram show the details of business process within a system from sequence diagram and coverage general outline in a system that user do from the beginning to the end from activity diagram however this test cases also contained some redundant information from combined activity and sequence diagram.

From this preliminary work, we will look up on how to optimize some test case using various kinds of genetic algorithm and implement that algorithm in our application. Another UML diagram will be added to the future experiment to provide extensive evaluation in this study field.

## 6. References

1. V.K.Shanthi a., MohanKumar G. A Novel Approach for Automated Test Path Generation using TABU Search Algorithm. *International Journal of Computer Applications*. 2012;48(13):28–34.
2. Srivastaval J, Dwivedi T. SOFTWARE TESTING STRATEGY APPROACH ON SOURCE CODE APPLYING. 2015;6(3):25–31.
3. Alazzam I, Alsmadi I, Akour M. Test Cases Selection Based on Source Code Features Extraction. 2014;8(1):203–14.
4. Tripathy A, Mitra A. Test case generation using activity diagram and sequence diagram. In: *Proceedings of International Conference on Advances in Computing*. Springer; 2013. p. 121–9.
5. Faria JP, Paiva ACR, Yang Z. Test generation from UML sequence diagrams. *Proceedings - 2012 8th International Conference on the Quality of Information and Communications Technology, QUATIC 2012*. 2012;245–50.
6. Swain RK, Panthi V, Behera PK. Generation of test cases using activity diagram. 2013;
7. Panthi V. Automatic Test Case Generation using Sequence Diagram. 2012;2(4):22–9.
8. Hettab A, Chaoui A, Aldahoud A. Automatic Test Cases Generation From Uml Activity Diagrams Using Graph. 2013;
9. Sumalatha VM, others. Object Oriented Test Case Generation Technique using Genetic Algorithms. *International Journal of Computer Applications*. 2013;61(20).
10. Asad S, Shah A, Shahzad RK, Shafique S, Bukhari A, Humayun M. Automated Test Case Generation Using UML Class & Sequence Diagram. 2016;15(3):1–12.
11. Khurana N, Chillar RS. Test Case Generation and Optimization using UML Models and Genetic Algorithm. *Procedia Computer Science [Internet]*. 2015;57:996–1004. Available from: <http://dx.doi.org/10.1016/j.procs.2015.07.502>