An Approach for Test Case Generation from a Static Call Graph for Object-Oriented Programming

Sitdhibong Laokok and Taratip Suwannasart

Abstract— **In software development, a software testing is a mandatory process to indicate the quality level of the software and to verify that all components have been working properly. For integration testing, it is a testing process used to verify the efficiency and to uncover errors occurring between class interfaces. This error indicating method may be expensive due to the reason that each class might have numbers of interfaces that need to be considered in source code. This paper aims at proposing an approach to generate test cases in order to cover all class interfaces, including of branch coverage, by collecting data from source code and generating a static call graph, which will represent all class interfaces found in source code. Moreover, our can gather appropriate data to support the generated test cases.**

Index Terms— **Control Flow Graph, Static Call Graph, Test Case Generation**

I. Introduction

SOFTWARE testing is an important process to indicate the confidence level of Software Under Testing (SUT) by verifying conformance to Software Requirements Specification (SRS) and uncovering errors that still remain in source code [1]. In order to perform a software testing, a software tester is required to read between the lines of code to generate a set of test case, test suite, and test data. This is to cover all interested software components. The software tester should have numbers of approaches to determine coverage criteria that are to be achieved.

While the test case generation is performed, the software tester has to read between the lines of code in order to understand the source code structure of SUT. There are several approaches that a software tester can use to represent the source code structure. Normally, Control Flow Graph (CFG) is widely used, as it makes source code more understandable even when the software tester is not familiar with the language used by developers. In addition, the software tester will also have to be able to derive test cases from CFG by picking up interested paths to be the test paths. Then, the software tester has to generate test data by considering the test path in order to assure that each test path is working well along with the generated test cases. Besides, the software tester has to set the goal before selecting the test paths. In path-oriented, the software tester must generate test cases to cover all branches in the source code structure (Branch-coverage) or to cover all predicate nodes [2].

For object-oriented programming, software is composed of classes that work together by sending signal to one another. The approach to the test case generation cannot focus on only an individual class, but it has to focus on the connection between them during the integration testing process, as errors can occur anytime when objects are connected.

During the integration testing process, the test case should cover all components that are found in source code in order to find errors occurring in each path that is placed between classes. Test case generation to cover all the existing paths of the source code structure is an expensive process, because the software tester has to seek for all paths one by one. The Static Call Graph (SCG), which is a graph that represents the connections between classes, will assist the software tester to generate the test case in order to validate all connected paths by gathering data from source code.

In this paper, we aim to propose test cases, which is generated from a call graph retrieved from source code, in order to represent all of the connections between objects. Moreover, we also propose test data generation, which complies with the test paths that the software tester has picked up.

The rest of the paper is organized as follows. Section II discusses existing works done based on the path-oriented method. Section III introduces the background of the program graph, SCG, CFG, and automated test case generation techniques. For Section IV presents the approach to test case generation and test data retrieved from SCG and CFG. Lastly, Section V concludes all the contents provided in this paper, altogether with future work.

II. Related Works

Unit test is a testing process to locate errors in SUT by focusing only on the interested parts and eliminate interaction occurred between software components by creating a drive or stub [1]. In contrast, integration testing is a process to uncover errors that may occur even though all the components have been working properly together [6]; however, the test case for this process that has to cover all of the class interfaces is expensive, as there are a large number of interfaces between classes that must be considered by the software tester. V. Panthini and D. Prasad [7] has proposed a generated test case based on a sequence diagram in order to identify interactions between objects. However, the sequence diagram may not reflect the current state of source

S. Laokok and T. Suwannasart are with the Department of Computer Engineering, Faculty of Engineering, Chulalongkorn University, Bangkok Thailand e-mail: Sitdhibong.L@student.chula.ac.th, Taratip.S@chula.ac.th

code, due to the reason that source code might be changed to another appropriate development methodology or techniques. S. Z. Waheed and U. Qamar [8] has proposed that the test case generation for the integration testing is based on the flow diagram data and selected DU paths that are used to be the test path. However, software is a result of class communication and the test path must be as long as possible to traverse each component that used to work together.

According to the references given above, we found that there is not any approaches that generate a test case that can be traversed through selected test paths between objects which have been working together based on object-oriented development in order to cover all branch interfaces found in source code. In addition, we are confident that our proposal will come up with an appropriate data set for the future test case generations.

III. Background

A. Program Graph

Software has been developed with multiple purposes and made source code become more complicated. The designing diagram is used here to illustrate how source code works. However, source code is always changeable to fit with the language of developers. Therefore, a program graph is a graph that is used to represent the structure of source code and to reflect the current version of it.

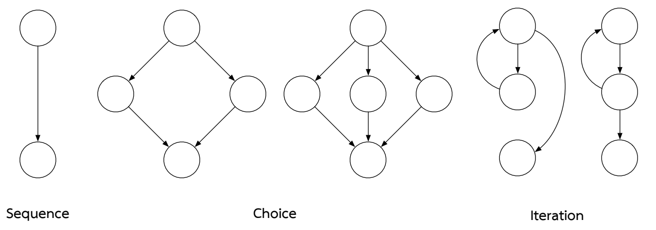

Fig. 1. The Primitive Operations of Structured Programing

1) *Control Flow Graph*

With difference purposes of software to be developed, there are several platforms and languages that software developer use for develop software. Therefore, it might be difficult for the software tester, as he or she may not be familiar with all of these differences. CFG is able to represent the structure of source code in the form of graph. CFG is Directed Acyclic Graph (DAG) that could represent the structure of source code and relationship between the lines of code with nodes and edges. It starts from the source node to sink node through the sequences of nodes, which are connected by edges. CFG with its primitive structure was defined by McCabe [3] as shown in Fig. 1. The software tester should analyze CFG and select the test path, which traverses from the source node through nodes in the graph with a purpose to cover each branch and exercise all predicate nodes in the graph.

2) *Static Call Graph*

Software is composed of classes, which work together by calling their own methods or other methods from other classes. SCG, which is a Directed Multiple Graph, represents the relationships between classes in SUT, in which each node represents classes and edge represents calling of method. In general, there can be several outdegree in a single node. SCG is formed by collecting calling statements found in source code, in which the called method has been called from the calling method for the other classes. Only when SCG illustrates the current source code structure, the software tester will be able to analyze the interfaces between classes to generate a test case that will cover all the interfaces in SUT.

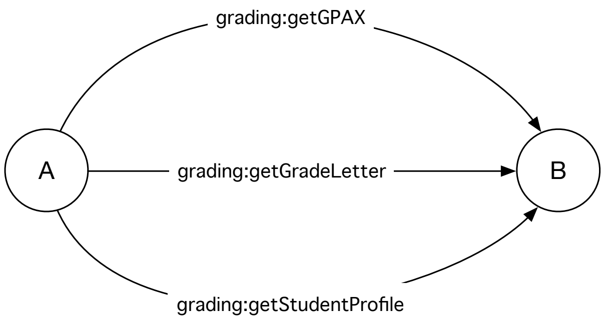

Fig. 2. Static Call Graph of Class A and Class B

In Fig. 2 shows an example of SCG that represents the relationships between class A and class B, Where class *A* has 3 calling statements in calling method, grading, that calls to called method *getGPAX*, *getGradeLetter*, and *getStudentProfile* in class *B*. Calling and called method are separated with ":" and used for label the edge between class *A* and *B* such as "*grading:getGPAX*". This relationship should be formed into *G = (V, E, l, p)*, where *G* is a Multiple Call Graph, *V* is a set of node, *E* is a set of edge, *l* is a set of label, *p* is a function that maps each edge in *E* to label in *l*. For this pair of nodes, *A* is the head node and *B* is the tail node [4].

IV. Proposed Approach

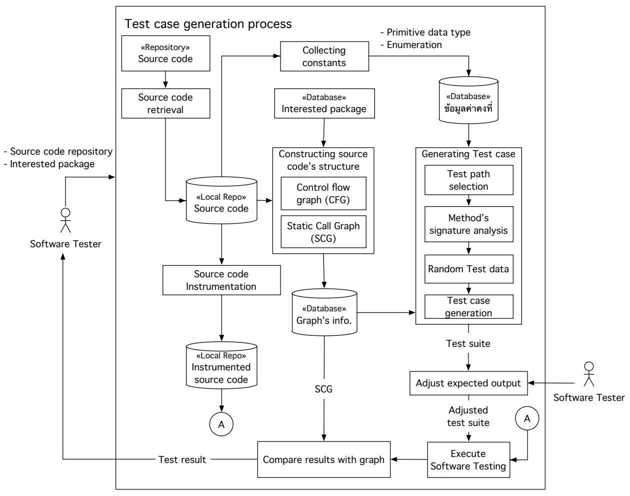

Fig. 3. Methodology of Test Case Generation based on Static Data

In this section, we present our approach to test case generation based on SCG. A set of static data is gathered from source code and formed into SCG and CFG. We assure that the generated test case conforms to the selected test path. At the last step, we execute a test case that comes with instrumentation source code and display the result to the software tester. Each step of our approach is explained in Fig 3.

A. Source Code Retrieval

Source code repository is a place to store source code, determined by the software tester. At this step, we should retrieve source code to access the local repository.

B. Constant Collection

An object is composed of a primitive data set (booleans, numbers, and strings); however, an object which is composed of random data will normally fail to satisfy the conditions and is not possible to activate a specific path. Therefore, we need to find an appropriate value that possibly satisfies the criteria. We could say that a predefined primitive value in source code can be used to construct an object more potentially than the others [5].

This process aims to analyze each class's files stored within a certain package designated by the software tester to collect constant values. The collected primitive values should also lead to random satisfying values in order to cover the test paths [5].

C. Constructing Source Code's Structure

In order to form a satisfying condition for source code, the software tester has to understand its structure. Therefore, this process is to create graphs, CFG and SCG, which represent the structure of source code and eliminate infeasible paths. When the process ends, source code structure, the graphs, and all possible paths will be stored in the database. Soon after, the generated test case will retrieve data from the database to generate another test case that conforms to the selected test paths. For more detail, each step is clearly discussed as follows:

1) Control Flow Graph Construction

Source code within the assigned package is retrieved from the repository given by a software tester, and parses each statement into nodes is assigned. Then, it will assign a relationship of the source node and the other nodes, which work after the previous node, until nodes are all connected. Finally, we must keep all the feasible paths and source node to sync all of them in the form of graph in the database. For example, if C_1, C_2, and C_3 are classes within the interested package which contains method sets: $\{m_{11}, m_{12}\}$, $\{m_{21}, m_{22}, m_{23}\}$, and $\{m_{31}, m_{32}\}$ respectively, CFG will be created from each method in a certain class.

From this process, we create CFG from source code to represent the source code structure. Furthermore, we also gather data from the steps of creating an object. These steps should be used as a reference in the test case generation process.

2) Static Call Graph Construction

This process collects the static data from source code for SCG construction. To perform this process, source code should be retrieved from the local repository. Then, calling statements that call to another class should be collected. A method that contains calling statements should be assigned to calling method, and a method is called in calling statement should be assigned to called method. To create SCG, in which each node represents classes in SUT and each edge represents the relationship between nodes. At the final phase, each edge will be labeled by the calling and called methods as shown in Fig. 4.

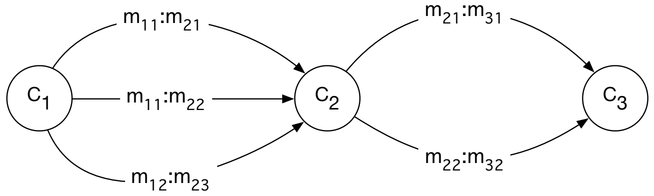

Fig. 4. Relationship between Classes C_1, C_2 and C_3

D. Source code instrumentation

This process adds instrumentation message that displays a specific message when the test case has exercised through the lines. To make sure that the generated test cases have already covered all SCG's branches (Branch coverage), at least once. Source code should be instrumented for monitoring if all branches are covered.

1) Method Entry and Existence

In statement instrumentation, the instrumentation message is added right after method declaration statement and before method ending or returning values. Message from this statement shows that test cases are able to exercise through methods between classes on test paths.

2) Predicate Execution

Selected test paths can be consisted of several predicate nodes. For this statement type, instrumentation message is added right after predicate statements.

3) Calling Method

Displaying message while methods between classed are invoked is the main goal of this approach. Instrumentation message is added right before and after finding these statements in order to display the message before and after invoking for method, respectively. For the message analysis, we analyze message from the entry method and explain the execution path for both invoking and invoke method.

After this process, instrumentation source code will be archived in the local repository.

E. Test Case Generation

In our approach, test cases are generated based on static data collected from source code. Previously, the constant and source code structures collected. This process analyze the data and generates test data and test cases in order to exercise each SCG's branch at least one time. The process is shown as in Fig. 5.

1) Test Path Selection

This process starts with retrieving all test paths of SCG and CFG structure formed in the database. Then, we should consider each pair of the SCG nodes. After that, the calling and called methods should be extracted from each edge label, one by one. For the next step, we should select CFG for calling node in order to analyze the test paths by finding the shortest path that contains calling nodes which represent to calling statements. This means that the first node pair of the test path is being selected. Next, the previous tail node is set to be the head node. Calling method is set to previously called method. Then, we repeat the test path selection steps in order to find all of the test paths. This is to achieve the branch coverage.

From SCG of class C_1, C_2, and C_3 as shown in Fig. 4, we can get started from the nodes of C_1 and C_2 (C_1 is the head node, while C_2 is the tail node). After that, we have to extract the name of the calling and called methods from each of the edge labels. Therefore, m_{11} and m_{21} retrieved from the above edge should be calling and called methods respectively. After that, we have to find calling statements and called method, which are m_{21}, in m_{11} from the structure of m_{11} as shown in Fig. 6, given that node 16 is invoking node (invoking statement). Now, we have already generated test paths, *11-12-14-16-18-19*, from m_{11} to cover all the branches (C_1, C_2, m_{11}:m_{12}). Then, the head node will be changed from C_1 to C_2, and the invoking method will be changed from m_{11} to m_{21}. When it comes to this final step, we need to repeat all over again if there are any edge labels that start with the invoking method. If there are no any edge labels with the invoking method left, we have to set the invoking method on another invoking method that is left on the previous head node.

According to the steps explained above, the test cases can be generated as shown in Table I.

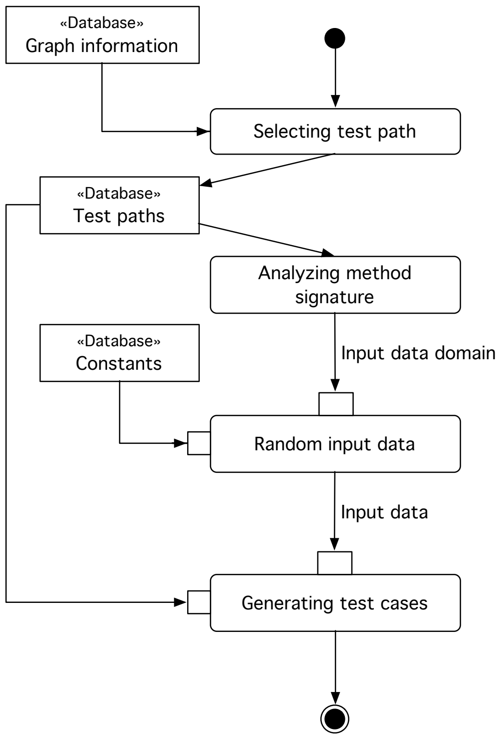

Fig. 5. Activities in Test case generation process

TABLE I
GENERATED TEST CASE FROM STATIC CALL GRAPH

ID	Test Path
1	(C_1, C_2, m_{11}:m_{21}) - (C_2, C_3, m_{21}:m_{31})
2	(C_1, C_2, m_{11}:m_{22}) - (C_2, C_3, m_{22}:m_{32})
3	(C_1, C_2, m_{12}:m_{23})

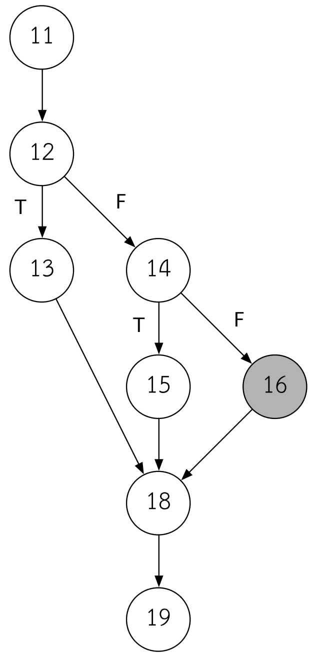

Fig. 6. Control Flow Graph of method m_{11} of class C_1

For the test paths retrieved from SCG in Table I we have to consider CFG of method m_{11} within class C_1, method m_{21} within class C_2, and finally method m_{31} within class C_3. CFG shown in Fig. 7 is the CFG of m_{11}, m_{21} and m_{31} respectively. The generated test paths should be tuple of ((*11, 12, 14, 16*)m_{11}, (*10, 11, 17*)m_{21}, (*10, 11, 12, 14, 18, 19*)m_{31}, (*17, 18, 19*)m_{21}, (*16, 18, 19*)m_{11}). Finally, all test paths are sent to the signature analysis method for the next process.

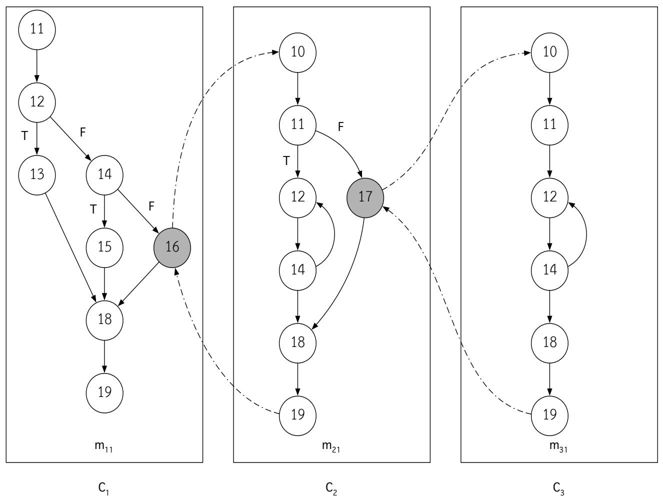

Fig. 7 Calling statement between method m_{11} of C_1, method m_{11} of class C_2, and method m_{31} of class C_3

2) Signature Analysis Methods

This process is performed in order to guide an input domain for random input process by signature analysis method, calling and called methods and test path that are transferred from the previous process, including the predicate nodes that are found on the test path [5]. For example, a selected path (Fig. 7) has 3 predicate nodes i.e. node 12, node 14 of method m11, and node 11 of method m_{21}. The random input data are guided by conditions found in predicate node. Thus, input data should conform to each predicate node and each predication node must be in line with the conditions provided in Fig. 8.

```
m11:12 studentScore > 80
m11:14 studentScore > 50
m21:11 true == hasQuizScore
```

Fig. 8. Conditions in Predicate Nodes

This is where the test data generation has considered each predicate node provided above. Test data should be generated to conform with predicated found in node m_{11}:12, m_{11}:14 and m_{21}:11 in order to activate the test case.

```
m11(String studentID, float studentScore) : String
m21(String studentID) : float
m31(String studentID) : float
```

Fig. 9. Method Signatures of m_{11}, m_{21} and m_{31}

With the signature method m_{11}, m_{21} and m_{31} as given in Fig. 9, when the test data has considered each predicate above, it is to say that, there is only studentScore that we have to consider and ignore for hasQuizScore. Because hasQuizScore does not exists in method signature. Finally, studentScore should conform predicate node must be lower than or equal to 80 and 50.

3) Random Input Data

Previously, random data has already randomized the value, but the method could have been more than one parameter. That is, each parameter of the method has to be considered. For the parameter found in the predicate nodes, it already has a guiding value from the previous process. On the other hand, parameters that are not found in the predicate nodes must be randomized by using constant value gathered from the constant collection process [5].

4) Test Case Generation

For this process, test path should be converted to a set of test cases. Test case generation must conform to the steps of object creation that can be found in the test path, including of the test data formed in the previous process.

F. *Expected Output Adjustment*

At this process, test cases are generated; however, they are not actually executed, because our approach has gathered only the static data and regardless the behavior method such as methods of returning values or input values that do not appear on the test paths. The software tester must adjust these values to satisfy the test paths, altogether with considering the behavior method.

G. *Software Testing Execution*

After expected output adjusted, the software tester has achieved to adjust expected outputs of the test cases, the test case must be executed with the instrumented source code retrieved from the source code instrumentation process in the local repository. During the execution process of the test, we should collect instrumentation messages that are displayed when the test cases traverse through nodes of test path.

H. *Result Comparison to Graph*

In order to verify a generated test case, we should create a traversing path from the instrumentation message that is collected from the previous process and display the execution result to the software tester.

V. Conclusion and Future Work

This paper introduces an approach for test case generation based on integration testing by considering the static call graph. The generated test cases exercise all branches in the static call graph at least one time. Moreover, the test data must be generated for the test case by collecting static data from source code to make the test cases exercise through the selected test paths.

For future work, we aim to adopt this approach in creating a tool that is to be used for the future test case generations in an integration testing.

References

[1] P. Jorgensen, Softare Testing: A Craftman's Approach. Auerbach Publications, 2013.
[2] W. Luanghirun and T. Suwannasart, "Test cases generation tool for JavaScript based on statement coverage criteria," Lect. Notes Eng. Comput. Sci., vol. 1, pp. 479–482, 2016.
[3] T. J. McCabe, "A Complexity Measure," IEEE Trans. Softw. Eng., vol. SE-2, no. 4, pp. 308–320, 1976.
[4] J. Bang-Jensen and G. Z. Gutin, Digraphs: Theory, Algorithms and Applications, 2nd ed. Springer Publishing Company, Incorporated, 2008.
[5] L. Ma, C. Artho, C. Zhang, H. Sato, J. Gmeiner, and R. Ramler, "GRT: Program-analysis-guided random testing," Proc. - 2015 30th IEEE/ACM Int. Conf. Autom. Softw. Eng. ASE 2015, pp. 212–223, 2016.
[6] P. Jalote, An Integrated Approach to Software Engineering, 2nd ed. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1997.
[7] V. Panthi and D. P. Mohapatra, "Automatic Test Case Generation Using Sequence Diagram," in Proceedings of International Conference on Advances in Computing, A. Kumar M., S. R., and T. V. S. Kumar, Eds. New Delhi: Springer India, 2012, pp. 277–284.
[8] S. Z. Waheed and U. Qamar, "Data flow based test case generation algorithm for object oriented integration testing," in 2015 6th IEEE International Conference on Software Engineering and Service Science (ICSESS), 2015, pp. 423–427.