Test Case generation considering Definition-Use of Global Variables for Integration Testing

Muhammad Iqbal Hossain, Woo Jin Lee

School of EECS, Kyungpook National University
Daegu, South Korea
milonmi7@yahoo.com

Abstract. In embedded software global variables potentially cause many issues such as lack of access control, implicit coupling and dependencies with different module of the source code. Large number of integration module and its associated global variables introduce the problem of scalability. This paper propose an automated test case generation approach to solve dependency problem considering the definition-use of global variable and generate scalable test cases according to feasible test sequences.

Keywords: Integration testing, Global Variable, Definition-Use, Scalability.

1 Introduction

In embedded software global variables are used for memory synchronization, interrupt services and monitoring system behavior to achieve portability. Global variable does not share caller/callee relationship and available throughout the source code. However, global variables potentially cause many issues such as lack of access control and implicit coupling with different module of the source code. When number of module increases along with global variables there exist a lack of scalability. To solve these problems we introduced an approach to automatically generate test cases for embedded software which includes a test model; generated by different graph analysis technique and then sequential integration module sequences are generated from test model. Using data flow analysis we generate the valid definition-use of global variables to validate test sequences. Finally test cases are generated from test sequences. Here we assume that the global variables are independent from other global variables. This approach is automated, cost effective and the case study indicates high scalability on generating test cases.

2 Related works

White and Leung [1] introduced a testing approach for global variables based upon firewall concept for data flow module dependences. In this approach they attempted to unify firewall concept for both control flow testing and data flow testing. Because

of sensitization problem, the approach cannot be automated. Basically data flow testing refers to the use of data flow information to select test sequences. Definition-use pairs of the global variables are generated from the source file to determine the sequences. Definition-Use has been used in testing object oriented programming [2]. The DU-pair criterion is used to monitor the behavior of the objects through the execution period and keeping track of the DU of objects. Inter procedural data flow analysis methodology is used and to identify intra class definition-use pairs an object data flow graph is constructed for the program. But effectiveness of object flow based testing approach is low and it doesn't concern about scalability issue because usually OOP contains small amount of code with less number of global variables. But embedded system contains enormous number of codes with large number of global variables which implies the issue of scalability.

3 Test sequence generation from Test Model

In our previous work [3] we have generated test model as shown in Fig. 1(b) from source code of producer-consumer problem in Fig. 1(a). At first a call graph is generated from the source code, and then extends the call graph where the relationship between the functions is shown with respect to global variables and caller/callee relation. Control flow graph of each the functions are generated and reduce these graphs with different reduction criteria. After reduction, we combine the entire reduced graph and a test model is generated. This test model contains the Definition-Use of global variables and also the caller/callee relationship among the functions.

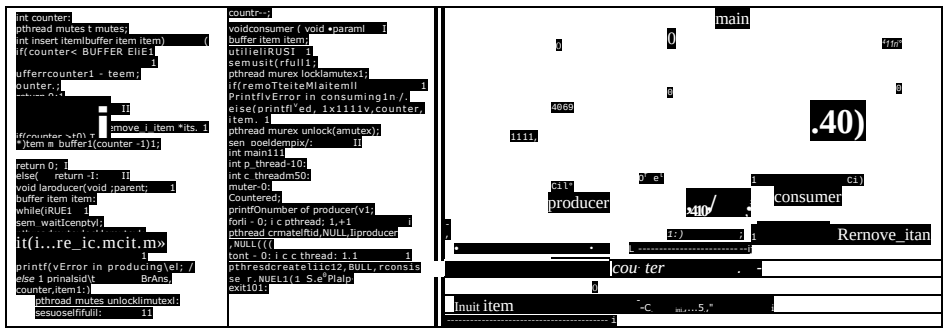


Fig. 1. (a) Sample source code, (b) Test model using graph analysis technique.

The entire workflow of this process is divided into who parts. In first part we describe how we generate test sequences using valid Du pair and then generate test cases. To generate test sequence from the test model we use DFS algorithm and determine sequential integration module sequences [4] [5]. On the other hand we generate valid DU pair and use this information to select the possible valid test sequences. After that test cases are generated according to test sequences. In this section we describe how

to generate test sequences and in section 4 will describe test case generation technique and a case study is presented in section 5.

3.1 Determination of valid DU pair

In our approach the coverage criterions are cover all Definition-Use of global variables and function coverage. Using the usages pattern of global variables, we generate DU pair and sequential integration module will cover the all function coverage. For global variable *counter* Definition set contains node 58, 7, 17 and Use set contains node 33, 7, 48, 17. DU pair is the Cartesian product of the elements of Definition and Use set. Not all Du pairs are valid. We define some rules or constrain for the DU pair to be valid. First rule (R1) is if there exists a define node between the elements of DU pair is not a valid pair. This definition may present in another node or it can be defined inside another function. Suppose global variable is defined in dl, d2, d3 and used in ul. If a DU pair is <dl, ul> and there is no other define node between dl and ul then we can say that it is a valid DU pair. If d2 or d3 is present in between dl and ul then the DU pair in invalid. Second rule (R2) is if the DU pair is not present in paths i.e. if we cannot make any relationship between the DU pair is also an invalid pair. If R1 or R2 is true then we can wind up that the pair is invalid and for other cases it is valid. After considering the rules we get valid DU pair as follows,

<58, 7>, <58, 17>, <7, 33>, <7, 17>, <17, 48>

These DU pairs will be used when validating the test paths.

3.2 Sequential Integration module Path with valid DU pair

In this section test sequences are generated according to the Definition-Use pairs of global variables. Here we use an idea of sequential integration module as represented in Fig. 2(a). As an integration testing approach, we consider the caller/callee relationship of the functions. It contains all possible combination of function that can be called from our source code. For each scenario of sequential integration module we generate sequential integration module sequences as generated in Fig. 2(b). To make the testing procedure more reliable we cover all possible Definition-Use of global variables.

1. main( )—.producer( )
2. main( )—Troducer( )—>insert_item( )
3. main( )—>consumer( )
4. main( )—>consumer( )—n·emove_item( )
5. main( )—>producer( )—>consumer( )
6. main( )—Troducer( )—.consumer( )—remove_item( )
7. main( )—>producer( )—>insert_item( )—.consumer(remove_item( )

(a)

PI: 55 58(D) 60 61 2526 28 29 4 5 7(U/D) 33(U) 36 62 64

P2: 55 58(D) 60 61 2526 28 29 4 5 7(U/D) 33(U) 36 62 63 39 41 43 44 48(U) 51 64

P3: 55 58(D) 60 61 2526 28 29 33(U) 36 62 64

P4: 55 58(D) 60 61 2526 28 29 33(U) 36 62 63 39 41 43 44 14 15 17(U/D) 48(U) 51 64

P5: 55 58(D) 60 61 2526 28 29 33(U) 36 62 63 39 41 43 44 48(U) 51 64

P6: 55 **58(D)** 60 62 64

P7: 55 **58(D)** 60 62 6339 41 43 44 14 15 **17(U/D) 48(U)** 51 64

P8: 55 58(D) 60 62 6339 41 43 44 48(U) 51 64

(b)

Fig. 2. (a) Sequential integration module, **(b)** Sequential integration module path with Definition-Use notation

Using the test model we put some symbol (D= Definition, U=Use, U/D=Use and Definition) to make the notation of Definition-Use of global variables. After identifying the DU of global variable inside the path we use the valid DU pair to verify whether the sequences are valid or not as illustrated in Fig. 2(b). The validation procedure is, we make a pair from definition-use node from integration module path and match it with previously generated valid DU pair. For example in path P1 we find a DU pair <58,7> and <7,33>. This pairs are present in the valid DU pair so we can say the sequence is feasible or valid. On the other hand in path P3 contain a DU pair, <58,33>. But it is not present in valid DU pair. So we can say that path P4 is invalid. Here the path contained Bolded DU pair is the valid test sequences and rests of them are invalid.

4 Test case generation from Test sequences

There are different techniques for generating test cases. Random testing is the simplest method and it can be used to generate input values for any type of program if data types of variables are known. But it doesn't perform well in terms of coverage because it relies on probability. Goal oriented test data generation technique generate input that traverses an unspecific path so it is sufficient for the generator to find input for any path. Since this method uses the find-any-path concept it is hard to predict the coverage given a set of goals. We use symbolic execution, an automatic technique, to generate test cases according to the test paths. The main idea behind symbolic execution [6] is to use symbolic values, instead of actual data, as input values, and to represent the values of program variables as symbolic expressions. This expression in interpreted as Path Condition. It accumulates constraints which the inputs must satisfy in order for an execution to follow the particular associated path, P7. Consider a test path that we generated before as:

55—>58—>60—>62—>63—>39—>41—>43—>44—>14—>15—>17—>48—>51—>64

The path condition using symbolic execution is:

$$(!\ ((I^{-0})11(I\langle\ PT))\ \&\&\ ((i==^{0})11\ ^{(I<}\ CT))\ \&\&\ !(CNT > 0).$$

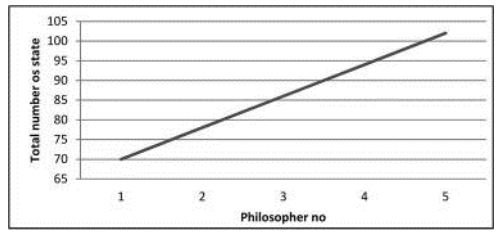
We randomly generate values for the symbolic inputs and check whether it satisfies the Path condition. If a set of input values satisfies the Path condition, is considered as a test case of the system.

5 Empirical Study

In our experiment we examine the source code of Dinning-Philosopher problem. For different number of philosopher we calculate the total number of states generated by our approach. While increasing the number of philosopher we also calculate the total number of global variable and total number of test cases as shown in Fig. 3(a). We have automatically generated different number of test cases according to the number of philosopher. The graph represent in Fig. 3(b), shows the relationship between total number of states and number of philosopher. The graph shows the total number of states increases linearly with the philosopher's number. So we can conclude that our approach is highly scalable.

Phi no	Number of Global variables	Total number , of state	Test cases
3	8	70	9
4	10	78	12
5	12	86	15
6	14	94	18
7	16	110	21

(a)



(b)

Fig. 3 (a) Experimental data from Dinning Philosopher problem, (b) Graphical representation of total number of state and number of philosopher

5 Conclusion

In this paper we introduced an automated approach for generating scalable test cases for integration testing. In particular the testing approach generates test paths according to the Definition-Use of global variables from where the test cases are generated. Using DU of global variables the number of test sequences are reduced compare to other testing approaches like random testing or path oriented random testing. A study of Dinning-Philosopher problem shows high scalability of our approach. A limitation in the presented approach is that for large program we have some error regarding memory allocation. In future work we like to improve the algorithms for generating test model, solve memory allocation problem and consider the dependency between global variables.

Acknowledgement

This work was supported by the IT R&D program of MKE/KEIT. [10041145, Self-Organized Software-platform (SOS) for welfare devices].

References

1. Lee, J.W., Hareton, K.N.L.: A firewall concept for both control-flow and data-flow in regression integration testing. IEEE Conference on Soft. Maintenance, pp. 262-271 (1992)
2. Chen, M.H., Kao, H.M.: Testing object-oriented programs - an integrated approach. 10th International Symposium on Software Reliability Engineering, pp. 73-82 (1999)
3. Hossain, M.I., Shin, Y.S., Lee, W.J.: Integration Testing Approach using usage patterns of global variables. The 37th KIPS Spring Conference, pp. 1213-1214 (2012)
4. Pu Y., Wan W.: A Balancing Model between Structural Testing and Functional Testing. Intl. Conference on Computer Science and Software Engineering. vol. 2, pp. 716-718 (2008)
5. Mary J. H., Mary L. S.: Efficient Computation of Interprocedural Definition-Use Chains. ACM Transactions on Programming Lang. & Systems, vol. 16, pp. 175 — 204 (1994)
6. James C. K.: Symbolic execution and program testing. Communications of the ACM, vol.19, Issue 7 (1976)
7. Ramamoorthy, C.V., Chen W. T..: On the automated generation of program test data. IEEE Transactions on Software Engineering, vol. SE-2, issue. 4, pp.2935-300 (1976)