A Scalable Integration Testing Approach considering Independent Usage Pattern of Global Variables

Muhammad Iqbal Hossain and Woo Jin Lee
School of EECS, Kyungpook National University, Daegu, South Korea
milon_mi7@yahoo.com, Correspondence: woojin@knu.ac.kr

Abstract
Embedded systems are used to design and control variety of complex system. Now-a-days embedded system has been used in every aspect of our daily life such as automobiles, home appliances, cell phones, security system etc. It is also being used in highly safety critical system like aerospace, medical devices, military equipment etc. So it is necessary to test the system before being released. Usually in embedded system there are different modules which have a real time interaction between each other. The best and easiest way to make this relationship is by global variables. But in embedded software global variables potentially causes 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. Scalability in the context of software engineering is the property of reducing or increasing the scope of methods, processes, and management according to the problem size. One way of assessing scalability is with the notion of scalable adequacy; the effectiveness of a software engineering notation or process when used on differently sized problems.

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.

There is a little work on testing approaches considering global variables. Some approaches do consider this issue but they are not scalable. So we introduced an approach to automatically generate scalable test cases for embedded software considering the definition-use of independent global variables. In other word, we are considering the relationship between each global variable along with associated local variables respectively. Our approach 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. Testing usage patterns of global variables

When a source code is given for testing global variables, we parse the source code to find the global variables. A call graph is generated from the source code. But call graph doesn’t represent the number sequence or other information for global variables. We implement an idea to extend the call graph where the relationship between the functions is shown with respect to global variables and callee function. Control flow graph of the function is generated and reduce the graph with different criteria. The reduction technique is used to decrease the number of paths. After reduction, each graph is combined and a Test model is generated.

We have divided the whole procedure in several parts. Figure 1 shows an overview of our approach.

![Figure 1. Overview of whole procedure](image)

Usually global variables are declared outside of a function. This is because of the accessibility of the variable throughout the program. When we parse the source code we look for the variables which are declared outside of the function. According to the global variables we construct a call graph. A call graph allows the user to view the relationship between the subroutines and the main function [3]. Specifically, each node represents a procedure and each edge (f, g) indicates that procedure f calls procedure g. Thus, a cycle in the graph indicates recursive procedure calls. There are two kind of call graph: dynamic and static. Dynamic call graph is a record of a single execution of the program but static call graph represent all possible run of the program. We consider static call graph to represent the relationship between main and subroutine function.
2.1. Extended Call Graph

A static call graph doesn’t show the relationship with respect to variable. It also doesn’t provide the sequence of the subroutines. We extend this static call graph with the definition-use concept of global variables. Global variables can be defined and used several times in the function. We need to find the functions where the global variables are either defined or used and amend this information with the call graph.

A sample source code of producer consumer problem is shown in Figure 2.

![Figure 2. Sample source code](image)

Figure 3 shows a sample call graph of producer-consumer problem, extended by our approach. It describes the relationship between the functions and the definition-use of the global variables [4]. In Figure 3 solid line represent function call and dotted line represent definition-use of global variables. We find that counter and mutex are declared as global variables. Counter is defined in main, insert_item and remove_item functions and used in insert_item and remove_item function. In similar manner mutex is defined in main function and used in producer, consumer, pthread_mutex_lock and pthread_mutex_unlock function. As we can see sem_wait have no relationship with any of the global variables so omission of sem_wait will not affect our process.
2.2. Combined CFG by reduction

We generate control flow graph of all the function which have a relationship in the call graph. A control flow graph is a data structure built on top of the intermediate code representation abstracting the control flow behavior of compiled function [5]. It is an oriented graph where nodes are basic blocks and edges represent possible control flows from one basic block to another.

For testing global variables we combine the entire control flow graph in a single graph which is called a model. The main problem of this approach is the excessive number of paths. When we generate test cases from the model it will be very difficult and complex for covering all paths of the model. We introduce a mechanism that reduces the control flow graph with respect to the global variables and callee function.

A function contains a huge number of conditional, iteration, switch and assignment statements those results the large number of paths. We just consider those statements which are directly or indirectly related to the global variables, others are reduced in our defined criteria.

- **Sequence reduction**: The first criterion is divides the statements into a basic block. Basic block is a maximal sequence of the statement such that if and are two adjacent statements in this sequence and no definition-use of global variables are executed. The execution of is always immediately followed by the execution of .

- **Loop reduction**: Loop in a program has a great effect on the number of paths. We use some optimization technique on the loops where there is no effect of global variables. Loop fusion, loop interchange, loop tiling, loop unrolling are the most common techniques for loop optimization [6].

- **Branch reduction**: Lastly branch optimization technique is used to reduce the number of if-else or switch statements. If a switch or if-else statement is not related with global variables or callee function, we implement branch optimization technique.

![Figure 3. Extended call graph](image-url)
Using these criterions we generate reduced control flow graph for each function. Finally according to the call graph we combine the entire reduced control flow graph as shown in Figure 4. This is our Test model which is used for generating test cases. In Figure 4 the solid line (Bold) represent function call, dotted line represent definition-use of global variables.

3. Test sequence generation from Test Model

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 [7, 8]. 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 3.3 will describe test case generation technique and a case study is presented in Section 4.

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 d1, d2, d3 and used in u1. If a DU pair is <d1, u1> and there is no other define node between d1 and u1 then we can say that it is a valid DU pair. If d2 or d3 is present in between d1 and u1 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 Figure 5. 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 Figure 6. To make the testing procedure more reliable we cover all possible Definition-Use of global variables.

```plaintext
1. main( ) → producer( )
2. main( ) → producer( ) → insert_item( )
3. main( ) → consumer( )
4. main( ) → consumer( ) → remove_item( )
5. main( ) → producer( ) → consumer( )
6. main( ) → producer( ) → consumer( ) → remove_item( )
7. main( ) → producer( ) → insert_item( ) → consumer( ) → remove_item( )
```

**Figure 5. Sequential integration module**

| P1: 55 58(D) 60 61 25 26 28 29 4 5 7(U/D) 33(U) 36 62 64 |
| P2: 55 58(D) 60 61 25 26 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 25 26 28 29 33(U) 36 62 64 |
| P4: 55 58(D) 60 61 25 26 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 25 26 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 63 39 41 43 44 14 15 17(U/D) 48(U) 51 64 |
| P8: 55 58(D) 60 62 63 39 41 43 44 48(U) 51 64 |

**Figure 6. 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. 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.

3.3. 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 unspecified 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 [9] 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) || (I< PT))) && ((I==0) || (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.

4. Empirical Study

Existing approaches use random testing (RA) or path oriented random testing (PRT) approaches to generate test cases but sometimes it generates infeasible test paths. To resolve this issue we select valid test path and then generate feasible test cases. To evaluate our approach we make a comparison with path oriented random testing approach. The experimental results clearly show a dramatic improvement of our approach over path oriented random testing approach. Table 1 represents the comparative data for path oriented random testing and our DU based testing. We choose 3 examples and generate test cases using path oriented random testing and using valid DU pair of global variables. As it is illustrated example 2 contains 70 lines of source code, 8 valid Du pair, 2 global variables and 5 functions. While generating test cases using PRT we get 36 test cases and using our valid DU pairs this number reduced to 14. So the reduction rate is around 61%. For rest of the cases the reduction rate is 41.6% and 25%.

<table>
<thead>
<tr>
<th>Example</th>
<th>Line Number</th>
<th>Valid DU pair</th>
<th>Test cases using PRT</th>
<th>Test cases using DU</th>
<th>Reduction rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>82</td>
<td>5</td>
<td>36</td>
<td>21</td>
<td>41.6%</td>
</tr>
<tr>
<td>2</td>
<td>70</td>
<td>8</td>
<td>36</td>
<td>14</td>
<td>61%</td>
</tr>
<tr>
<td>3</td>
<td>73</td>
<td>1</td>
<td>12</td>
<td>9</td>
<td>25%</td>
</tr>
</tbody>
</table>
We have plotted the number of test cases for each of the examples in Figure 7. Here dashed line represents test case generation for our valid DU pair approach and solid line represents the number of test cases with path oriented random testing approach. After analyzing the graph, it can be found that using Valid DU pair the line always goes under the line of PRT.

![Graphical comparison between PRT and Valid DU approach](image)

**Figure 1. Graphical comparison between PRT and Valid DU approach**

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 Table 2. We have automatically generated different number of test cases according to the number of philosopher. The graph represents in Figure 8 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.

**Table 2. Experimental data from Dinning Philosopher problem**

<table>
<thead>
<tr>
<th>Philosopher number</th>
<th>Number of Global Variables</th>
<th>Total number of states</th>
<th>Test cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>8</td>
<td>70</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>78</td>
<td>12</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>86</td>
<td>15</td>
</tr>
<tr>
<td>6</td>
<td>14</td>
<td>94</td>
<td>18</td>
</tr>
<tr>
<td>7</td>
<td>16</td>
<td>110</td>
<td>21</td>
</tr>
</tbody>
</table>
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.

Acknowledgements

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

References

Authors

Muhammad Iqbal Hossain is a Ph.D student in the department of Computer Science and Engineering at Kyungpook National University, South Korea. He has been a researcher in embedded software engineering lab from 2010. His research interest includes embedded software engineering particularly testing and verification.

Woo Jin Lee is currently an associative professor in the school of Computer Science and Engineering at Kyungpook National University, Korea. He received the Ph.D. degree in Computer Science from Korea Advanced Institute of Science and Technology in 1999. His main research interests include embedded software testing, modeling and verification of embedded software, and component-based software development.