Integration Testing Approach using Usage Patterns of Global Variables

Muhammad Iqbal Hossain, Youngsul Shin, Woo Jin Lee
School of EECS, Kyungpook National University
E-mail : milon_mi7@yahoo.com

Abstract

Global variables can be read or modified by any part of the program, making it difficult to remember or reason about every possible use. Sometime it has tight couplings between some of other variables, and couplings between variables and functions. The main focus of this paper is to use call graph and the control flow analysis to design a model from where we generate the test cases for testing global variables.

1. Introduction

Global variables are used extensively to pass information between sections of code that do not share a caller/callee relation. It is a common practice to use global variables to simplify the code for the availability of it throughout the code. However, global variables potentially cause many issues such as lack of access control and implicit coupling, which make the program more difficult to test. In data sharing among the modules or parallel computing in distributed system, it needs to be synchronized and track the data dependencies of the global variables.

Asakura, Sugie [1] introduced a practical data sharing method for global variables on the inter process communication in distributed system. It calculates the execution order among functions firstly and then to control concurrently executable functions by synchronization indicator Trigger. White and Leung [2] introduced a testing and regression 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 but further research is needed. Since the test values of global variables are selected by an experienced developer, a sensitization problem still remains. So the approach cannot be automated. On the other hand for large program the complexity increases and makes it difficult to keep track of the dependencies of the global variables among the functions.

In comparison with the above approach, we have unified the control flow testing and data flow testing by combining reduced CFGs. We overcome the sensitization problem by the extended call graph and thus the approach is being made automated. Complexity of a program depends of the number of paths. Large program contains a huge number of paths that makes the testing procedure more complex. We introduced a graph reduction criterion to minimize number of test paths. The reduced graphs are combined in a single graph and generate the model. This Test model ensures all path execution for global variables.

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.

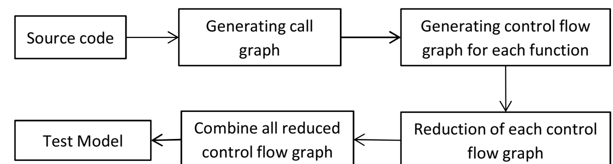

Figure 1: Overview of the whole procedure.

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.

3. 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.

Figure 2 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 2 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.

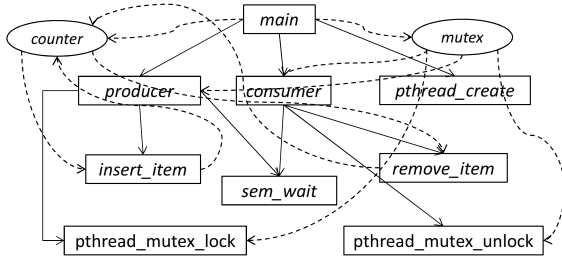

Figure 2: Extended call graph

4. Combined CFG by reduction

We generate control flow graph of all the function which have a relationship in the call graph (as shown in Figure 2). 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 $s_0, s_1, s_2 \dots s_n$ such that if s_j and s_{j+1} are two adjacent statements in this sequence and no definition-use of global variables are executed. The execution of s_j is always immediately followed by the execution of s_{j+1}.
- 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.

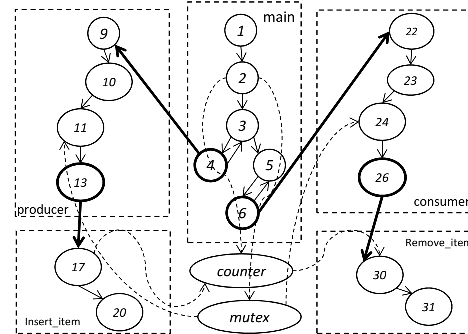

Figure 3: Combination of reduced CFG

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 3. This is our Test model which is used for generating test cases. In Figure 3 the solid line (Bold) represent function call, dotted line represent definition-use of global variables.

5. Conclusion and Future Work

We have discussed the side effect of global variables and an approach to test global variables. In functional testing we identify the global variables and generate an extended call graph with respect to the global variables. We use definition-use relationship for identifying the relationship among the subroutines. Then we generate control flow diagram for each function and with criterion technique reduce the graph. Finally we combine all reduced graph and generate a test model for generating test cases for the global variables.

In the future work we will introduce a technique to generated test cases from this model. The control flow graph reduction technique needs more research to assure the least optimization of the graph.

Acknowledgement

본 연구는 지식경제부 및 정보통신산업진흥원의 IT 융합 고급 인력과정 지원사업의 연구결과로 수행되었음(NIPA-2012-C6150-1202-0011)

Reference

[1] Koichi Asakura, Toyohide Watanabe and Noboru Sugie, “An execution order control method of distributed processes for sharing global variables,” IEEE Region 10's Ninth Annual International Conference on Frontiers of Computer Technology, pp.156–160, August 1994.

[2] Lee j. White and Hareton K.N Leung, “A firewall concept for both control-flow and data-flow in regression integration testing,” IEEE Conference on Software Maintenance, pp. 262–271, November 1992.

[3] Barbara G. Ryder, “Constructing the Call Graph of a Program,” IEEE Transactions on Software Engineering, vol. SE-5, pp. 216–226, May 1979.

[4] Pu Yun-ming and Fan Ming-hong, “The research of a new software testing model,” 2nd International Conference on Anti-counterfeiting, Security and Identification, pp. 67 – 70, August 2008.

[5] Franz Wotawa and Willibald Krenn, “Knowledge Extraction from C-Code,” Fifth Workshop on Intelligent Solutions in Embedded Systems, pp. 49–60, June 2007.

[6] Michael E. Wolf, Dror E. Maydan and Ding-Kai Chen, “Combining loop transformations considering caches and scheduling,” MICRO-29. Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 274–286, December 1996.