CORE
Metadata, citation and similar papers at core.ac.uk
Provided by Elsevier - Publisher Connector

Available online at www.sciencedirect.com
ScienceDirect
Procedia Computer Science 46 (2015) 859 – 866
Procedia Computer Science

International Conference on Information and Communication Technologies (ICICT 2014)

Model Based Distributed Testing of Object Oriented Programs

Vipin Kumar K S[a,*], Sheena Mathew[b]

[a]*Dept. of CSE, Govt Engineering College, Thrissur,680009,India*
[b]*School of Engineering, Cochin University of Science and Technology,682022,India*

Abstract

In recent times the software systems have evolved in size and complexity. This has resulted in usage of object oriented programming in the development of such systems. Though object oriented programs are helpful in programming large systems, testing of such systems requires much more effort and time. For this the program is analyzed to create a model based on System Dependence Graph(SDG) which is then used to find locations within the program where the state of the program can be freezed and reused while executing other test cases.
© 2015 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).
Peer-review under responsibility of organizing committee of the International Conference on Information and Communication Technologies (ICICT 2014)

Keywords: Distributed Testing; SDG; FCFS; Object Oriented Program Testing;

1. Introduction

In this paper we introduce a novel approach to distributed testing of object oriented programs based on state where intermediate state of the program is saved for reuse. The system brings about efficiency to testing by reusing saved states of the program during execution. The state of a program that is executing can be saved by using snapshotting techniques. In Java, continuation object can be used to capture everything in the java stack of a client (host) node. The continuation objects is then serialized and send to server which reinstates the state on a client node with the help of a client supervisor running on the client node. We propose a hosted model for this system where we have the client supervisor running above the operating system. This simplifies the client supervisor system as it need

* Corresponding author. Tel.: +919288139331; fax: +91487 2334590.
*E-mail address:*vipin.kumar.k.s@gmail.com

1877-0509 © 2015 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).
Peer-review under responsibility of organizing committee of the International Conference on Information and Communication Technologies (ICICT 2014)
doi:10.1016/j.procs.2015.02.155

not keep track of drivers and resources as these responsibilities will be taken care off at the operating system level. Fig.1. depicts the host with the client supervisor running over the host operating system. The client supervisor executes the testcase by reinstating the state of the program at a predetermined point within the program.

Nomenclature

FCFS First Come First Serve
SDG System Dependence Graph
ANTLR Another Tool for Language Recognition

The program to be tested is analyzed and a model is constructed based on SDG. The model helps in identifying potential points within the program where the state of the program in execution can be saved for further reuse. During execution of each testcase, the host saves the state at these points and sends these states to the server along with an identifier which will help in identifying the testcase to be run by reinstating the saved state. The host continues executing the testcase it was executing after sending the saved state to the server.

The analyzer is basically a compiler that builds the model and then analyzes and profiles the code with statements for saving state and sending the state over to the server.

The system apart from executing testcases simultaneously, it also bring about efficiency by reusing intermediate state of the already executed program by carefully analyzing for points within the program for which the subsequent statements have no dependence to preceding statements. It is worth noting that in traditional path coverage based testing the preceding statements are executed more times than statements that appear after a succeeding condition statement. In this approach however succeeding statements are the ones that will be executed more times as the preceding statements are not executed when the state is reinstated. In the discussion that follows we consider condition statements and focus on two paths corresponding to true and false evaluation of the condition statement. The loops can be accommodated into this concept by similar treatment. However for simplicity we focus only on condition statement. The result of analysis of similar approach to regression testing in the work published in[11] is also presented in the Analysis section (section 4).

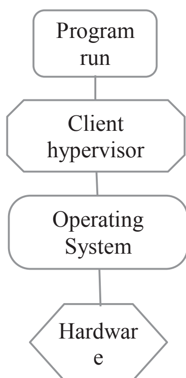


Fig. 1 Client supervisor

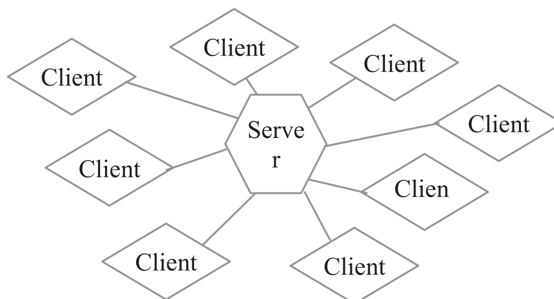


Fig. 2. System Architecture

2. Intermediate representation for the compiled program

The compiler analyses the program and profiles the code to be able to distribute the execution across various client nodes. The Fig.3 depicts the role of the compiler in analyzing the input program. During analysis the compiler

constructs a model based on SDG for the compiled program. This model is built by incorporating control flow and method sequence into the SDG. We augment control flow into the SDG[1,2] by introducing control flow edges which represents the ordering of statements within a given method. Method sequences are used within the model to represent methods invoking other methods along with the messages used for the invocation. An MM-Path (*Method - Message Path*) proposed in[3,4] represents the sequence in which methods are executed and the corresponding messages invoking these methods. Detailed discussion about the model is available in[9,10]. Sample program as well as the augmented SDG is shown in Fig. 4 and Fig. 5.

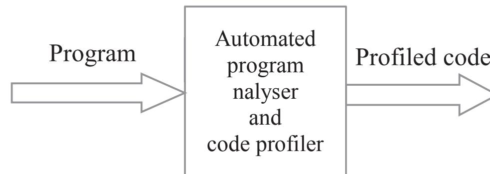


Fig. 3. Compiler analyses the program and profiles the code to be able to save and distribute the execution.

```
public class Fibonacci {
    public static void main(String a[]){
        int Count = 15;
        int i, feb1, feb2, feb3;
        feb1 = 0;
        feb2 = 1;
        i=2;
        System.out.print(feb1);
        System.out.print(feb2);
        while(i < Count){
            feb3 = feb1 + feb2;
            System.out.print(feb3);
            feb1 = feb2;
            feb2 = feb3;
            i++;
        }
    }
}
```

Fig. 4. Sample Program.

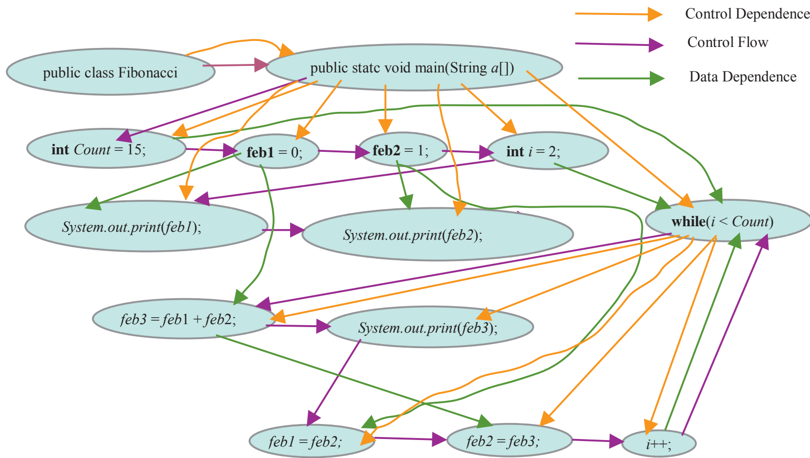


Fig. 5. Augmented SDG for the sample program in Fig. 4 [9,10].

The code is profiled (Fig.3.) to save state and to send the serialized state object to the server. Server then sends it to other client nodes for distributed execution of other testcases. The point at which the state is to be freezed or saved is determined by data dependence analysis. Each of the condition expression allows the execution to proceed along two different paths, doubling the number of execution paths considered till that statement. Each of the segments in the control flow path can be identified uniquely using the critical edge. The critical edge is the edge that connects the condition expression node with the next node along a control flow. In the Fig 6. the edge<si+2, si+3> as well as the edge <si+2,si+5> are critical edges representing two paths corresponding to two different control flow along the condition node si+2. In order to find the point at which the code is to be profiled with respect to the condition node si+2 we follow the data dependence edges originating initially at the node si+2 to preceding node until no more data dependence edges exists from a node. In Fig. 6. there is no further data dependence edge from node si. The code is profiled by inserting appropriate code to save state and send the state along with a condition statement identifier to the server for distributed testing. Critical edge information is particularly useful as the system works under the pretext that path information of each testcase is available before hand.

3. Scheduling of execution of testcases

The server performs the scheduling on a first come first serve basis (FCFS). Each time a state object is received by the server, there are exactly half of the testcases being executed or already scheduled for execution by the server with respect to the number of test cases required with respect to the current condition node. As each condition node has the effect of doubling the number of test cases for achieving path coverage. Hence the server will schedule those many more testcase runs on a first come first serve basis. First come first serve scheduling is an optimal scheduling in this scenario if there are no resource considerations during testing. This is because testcases are scheduled according to decreasing execution times as state is saved for distributed testing. As FCFS leads to least delay and since the testcases are scheduled according to decreasing execution time, it leads to an optimal scheduling.

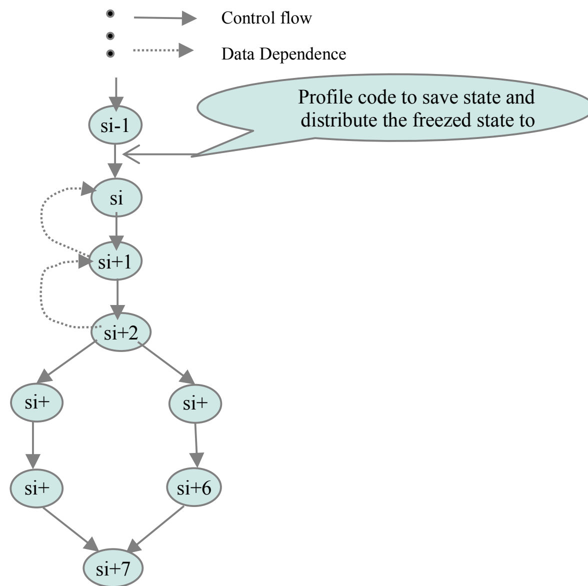
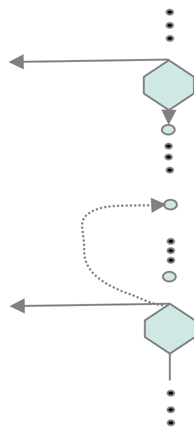


Fig. 6. Code profiling

4. Analysis of the proposed method

For the purpose of analysis an arbitrary program with uniform features is assumed. It is assumed that the program has conditional statements at equal intervals between the first and last conditional statements and all of them are present sequentially. The conditional statements branches the control flow within the program along two different paths creating a multiplication effect of two to the total number of execution paths that exists till that statement.



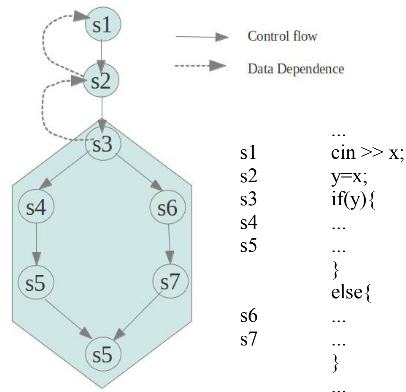


Fig. 7.Data dependence for the condition nodes.

Fig. 8. Code Fragment and part of the model representing the fragment

We also assume that for each of the c_i^{th} (Condition i in Fig.7.)condition statement is data dependent on a statement between the c_{i-1}^{th} condition statement and c_i^{th} condition statement and that this statement is

approximately at the middle of this code segment. A code fragment and the part of the model representing the code fragment are shown in Fig.8.

Each condition statement c_i makes it necessary to double the testcases in order to achieve path coverage Fig.9. Let the arbitrary program discussed above have a total of 'c' conditional statements. Then we require a total of ' 2^c ' testcases Fig.9. Let us assume that each testcases take the same amount of execution time ‘t’ to finish execution. Then the total execution time for executing all the test cases is:

$$= t/c * 2^{c-1} + 2t/c * 2^{c-2} + 3t/c * 2^{c-3} + 4t/c * 2^{c-4} + \ldots + (c-1)t/c * 2^1 + t * 2^0 \quad (1)$$

$$= \sum_{i=0}^{c-1} t/c * (c-i) * 2^i$$

$$= t/c(2^{c+1} - (c+2)) \quad (2)$$

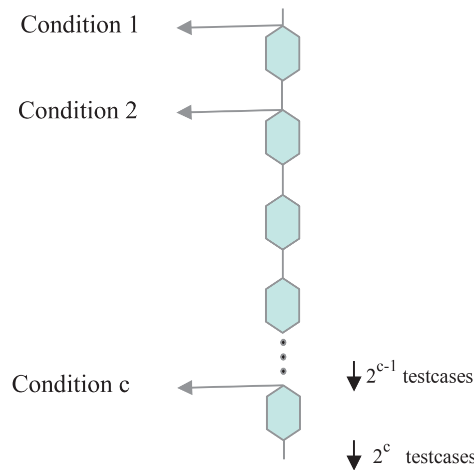


Fig. 9.Testcase count increases two fold with respect to each condition statement.

The execution time for executing 2^c testcases on a single node would be $t * 2^c$ and for executing over 'n' number of nodes is $t * 2^c/n$.

For a program having 10 condition statements and a execution time of 1 millisecond would require 1024 milliseconds for a system with distributed testing without saving of state compared to 202.8 milliseconds for a system with distributed testing with state saving.

The details relating to structural details and object oriented metrics can be used in testcase selection and testcase creation for the system discussed here [5,7,8]. The system works based on the assumption that the internal structural details pertaining to each testcase is available beforehand. So such a system can work in tandem where testcases are generated automatically by code analysis or under the pretext that at least there is path information available for each testcase.

A similar approach was followed in our work cited in[11] which related to regression testing with state saving approach is given below. The system with distributed execution with state saving is compared with single node execution and distributed execution of testcases without state saving in Fig.10 and Fig.11. The system was implemented by constructing a compiler in ANTLR[6] following a similar approach.

4.1. Scenario in which this system is useful is:

- When test cases are generated automatically using testcase generation tools which also provide execution trace information of each testcase.
- Structural testing of software (white box testing).
- Regression testing where trace information of prior run testcases are available.

No of testcases	Single Machine	Distributed testing without state saving	Distributed testing with saved states
5.0	217.9277493	72.64258311	86.47432859
7.0	305.0988491	101.6996164	89.09685871
9.0	392.2699488	130.7566496	93.00884871
11.0	479.4410485	159.8136828	96.10804778
13.0	566.6121483	188.8707161	99.30014751
15.0	653.783248	217.9277493	104.6101478

Fig. 10. Execution times obtained during test runs

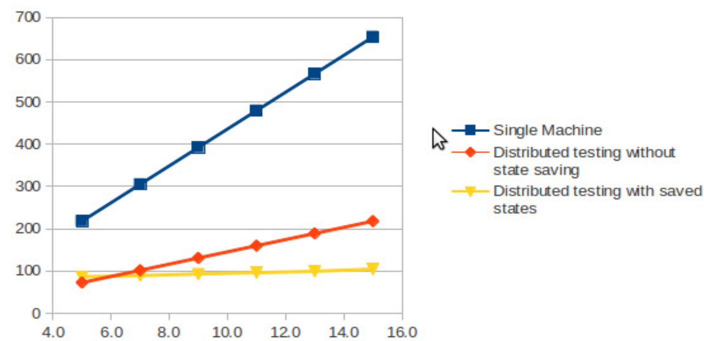


Fig. 11. Test results

5. Conclusion

The system reduces the time required for testing considerably. This approach can be adopted for testing of programs that are tested over a platform that allows snapshotting or for languages that support features like continuation which allow for saving of states during execution.

6. Future Work

The system can be implemented with a concept of k-coverage where the program is split into different segment and each segment can be executed exactly k-times. Each segment is independent with respect to other in terms of data dependence. Such an approach would increase the reliability of testing unlike the case where there is a chance that some segments are executed more often than others.

References

1. L Larsen and MJ Harrold. Slicing object oriented software, *Proc.of the 18th International Conference On Software Engineering*, March1996,p. 495–505.
2. TJ McCabe, LA Dreyer et al. Testing an object oriented application, *Journal of the Quality Assurance Institute*, October 1994, p 2127
3. Ruilian Zhao, Ling Lin. An UML Statechart Diagram-Based MM-Path Generation Approach for Object-Oriented Integration Testing, *International Journal of Applied Mathematics and Computer Sciences*, **3**: 1.
4. Paul C Jorgensen, Carl Erickson. Object-Oriented Integration Testing, *Communications of ACM*, **37:**9, p. 30-38, 1994.
5. Chidamber, S.R.; Kemerer, C.F. A metrics suite for object oriented design, *IEEE Transactions on Software Engineering*,**20:** 6,p. 476 – 493, June 1994
6. Terence Parr. Definitive ANTLR reference, Building domain specific languages, *The Pragmatic Bookshelf*.
7. Fenton N and Neil M. Software metrics: roadmap. *In ICSE – Future of SE Track* (2000), p. 357–370.
8. Sayed Mohsen Jamali, Object Oriented Metrics(A Survey Approach),Tehran, Iran, January 2006
9. Vipin Kumar.K.S, Rajib Mall, A Novel Intermediate Representation for Real Time Safety Critical Object Oriented Program, *Proceedings of National Conference on Computational Science and Engineering NCCSE2009*, pp. 20-26, 2009.
10. Vipin Kumar K S and Sheena Mathew. A Model Based Approach For Regression Testing Utilizing Distributed Architecture. *International Journal of Computer Applications* 16(2):26–31, February 2011.
11. Vipin Kumar K S, Lallu A and Sheena Mathew. An Efficient Approach for Distributed Regression Testing of Object Oriented Programs, *ICONIACC-2014,* ACM Conference (http://dx.doi.org/10.1145/2660859.2660944).