A Model Based Approach For Regression Testing Utilizing Distributed Architecture

Vipin Kumar K S
Dept of CSE, RIT, Kottayam

Sheena Mathew
Reader, SOE, CUSAT

ABSTRACT

The explosive growths in the usage of object oriented programming in the development of large applications have put extensive pressure in testing and maintenance of these systems. A graphical representation for these programs has the advantage of lending to efficient analysis compared to code based textual analysis. The Class Dependence Graph (ClDG) is insufficient to capture the features of real time safety critical object oriented program. We extend the basic ClDG to incorporate features like control flow and exception handling, timing, criticality, method sequences and sate information. The model that we have developed can easily be subjected to automated analysis for establishing points within a program that needs to be tested when the program is subjected to changes.

General Terms

ClDG, Regression Testing, Distributed Systems, Model-Based Regression Testing

Keywords

Class, class dependence graph, control dependence, control flow, data dependence, object, object oriented program, state, regression testing.

1. INTRODUCTION

The proliferation and development of complex systems has emphasized the need for an efficient design, development and maintenance of these systems. As a result there has been as increase in the usage of object oriented programming for the development of these systems. The usage of OOPS concept in development also requires the testing of the system to follow a distinctive approach compared to conventional testing. We extend the model that was proposed in [1] to be helpful to regression testing. A graph model provides a convenient way of representing and analysis of programs. Analysis of a graphical model is more efficient compared to a textual code based analysis. The model that we propose, extends the ClDG to create a representation for object oriented real time safety critical systems which is targeted to be helpful in testing and debugging. Exclusive representation of inheritance, polymorphism, method sequences, data dependence, control dependence and control flow relationships between different elements of a program makes automated analysis of the model more efficient when compared to code based analysis. The representation of the state information of the different classes in the program within the model helps in estimating the complexity of state based testing of classes. Another advantage of such model is the capability in assessing the quality of the program. Some serious work has been done in developing test strategies and test cases based on UML diagrams. Most of the work done is based on analysis of design documents and UML diagrams. In our approach we use the model, representing the developed program augmented with additional information from the UML diagrams for performing test case selection during regression testing.

ClDG represents the data and control dependencies between different program elements and is used extensively in representing object-oriented programs. ClDG helps to determine the parts of a program that affect a value computed at a particular point or parts of program affected by a statement. ClDG is augmented with the control flow information so as to be helpful in determining the statements involved in each execution trace.

In order to represent the timing and priority information, we extend the basic ClDG with control flow and method sequence information. We represent control flow in the ClDG by introducing control flow edges to specify the ordering of statements within a given method. The methods invoking other methods along with the messages used for the invocation are represented in the method sequences. The method invocation edges connect the method entry nodes of calling method to that of the called method, which form the individual edges of a method sequence. These method sequences are used to represent threads to which the timing and priority information is attributed. The priority information is stored at the start node of each thread while timing information is attributed to each method. The events and operations on object often lead to state transitions. The state information is stored in each of the class entry nodes to assist in testing. The control flow edges helps to represent how exceptions affect the normal flow of control in a program. As exceptions may transfer control from one method to its calling method in search of handlers, we have introduced control flow along with the additional nodes to represent exceptions in the ClDG. We have named the basic ClDG extended with these information as Extended ClDG or EClDG for short.

This paper is organized as follows:

- Development of the model by augmenting the basic ClDG.
- Use of this model for identifying test cases to be re-run and distribution across a distributed architecture.

2. MODEL FOR OBJECT ORIENTED PROGRAMS

The criticality and timing information are associated with threads in a program. We therefore need to represent a thread in ClDG. To be able to represent a thread, we need to represent the sequence of statements making up the thread. This requires representing control flow information. Control flow information is easy to determine from the source code. The control flow information can be represented in the ClDG by using edges that explicitly specify the ordering of the statements and method calls. The representation of control flow helps us to capture vital information required during testing. The number of independent control flow paths in the graph can be treated as measure of the number of test cases required for testing the program. The programs may be tested with the criteria of selecting the test cases in such a way that all the independent paths are executed atleast once.

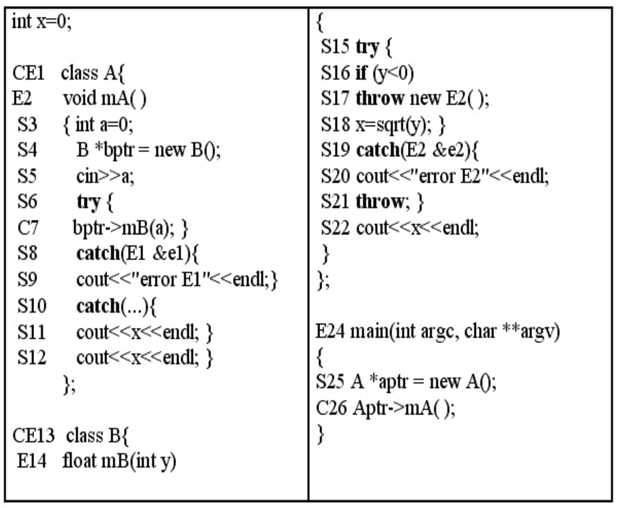
With the incorporation of control flow, the control flow paths help us to represent exceptions. The exception facility allows programmers to define, throw and catch exceptional objects. Here we have assumed the syntax of a C++ language but supporting a strong exception handling facility like Java with built in exception objects. In Java where there is an Exception class. Moreover users can raise any object as an exception by using a throw statement. A try {. . .} catch {. . .} structure attaches handlers led by the catch construct to a guarded block of code led by the try construct. Corresponding to each try, there is one (or more) catch statement(s) that can handle the exception as and when it occurs in the code enclosed within a try block. The exact catch block to be executed is selected based on matching the object raised by the exception and the one used in the catch statement (both must be of same type). If the handler for a raised exception cannot be found in the catch statements available locally, runtime unwinds the call stack of the try block and propagates the exception upwards. This propagation continues until a suitable handler is found. If no suitable handler could be found, the default handler is called which aborts the program execution. After a handler is found and executed, the execution of the try block is terminated. After the catch block completes, execution continues from the first statement after the try block.

Execution of a throw may change the dependence relationships of some statements. In sequential programs, the control dependencies mainly arise due to control condition statements, function call statements and exceptions that alter the sequential execution, while data dependencies arise due to accessing of variables, parameter-transfers and exceptions. Exceptions do affect data dependence as they may alter the definition-use chains of some variables. Since a throw statement and a catch statement may also affect control flow, it is necessary to represent these different paths possible during the execution of the program in a ClDG. The throw and catch nodes in the dependence graph act as head nodes of the control dependence edges representing alternating paths when control dependence is analyzed and constructed. As the evaluation of the conditional expression of a conditional statement can lead to alternate execution paths, the execution of the throw statement can lead to execution of different statements depending on the specific exception event. The set of statements corresponding to a catch statement is executed only when the object used in raising of the exception is of the same type as the one used in the catch statement.

We now illustrate the representation of execution paths through an example.

Example 1: Consider the sample program shown in Figure 1

The numbers have been assigned sequentially to each statement in the order they appear in the source code for identifying them in the ClDG. The prefixes S, E, CE and C denote statements, method entry, class entry and call nodes respectively. ClDG is augmented with the control flow information in Figure 2.



```
int x=0;

CE1  class A{
E2     void mA( )
 S3    { int a=0;
 S4       B *bptr = new B();
 S5       cin>>a;
 S6       try {
 C7      bptr->mB(a); }
 S8       catch(E1 &e1){
 S9       cout<<"error E1"<<endl;}
 S10      catch(...){
 S11      cout<<x<<endl; }
 S12      cout<<x<<endl; }
       };

CE13  class B{
 E14   float mB(int y)
```

```
{
 S15 try {
 S16 if (y<0)
 S17 throw new E2( );
 S18 x=sqrt(y); }
 S19 catch(E2 &e2){
 S20 cout<<"error E2"<<endl;
 S21 throw; }
 S22 cout<<x<<endl;
 }
};

E24 main(int argc, char **argv)
{
S25 A *aptr = new A();
C26 Aptr->mA( );
}
```

Fig. 1. An Example Program.

Exceptions affect the normal flow of control in a program. Our approach to represent the same in the ClDG is based on that reported in [2]. The additional nodes that we introduce in the ClDG to capture the notion of exception handling are try-node, catch-node, throw-node, normal exit-node, exception exit-node, exceptional return-node and normal return-node. The throw and catch nodes function similar to predicate nodes. That is, a throw statement affects flow of control, changing the definition-use chains of some variables, and also changing the dependence relationship of some statements.

The statements associated with a catch statement may or may not be executed depending on whether the exception object matches with that of a catch statement at run-time. So, throw-nodes and catch-nodes have alternate paths for control to flow, as an exception may or may not be raised and matching of exception object with the one used in catch statement.

The Normal Exit and Exceptional Exit nodes are used in the called function to differentiate between a normal return and a return occurring due to an exception in that function. A normal return (represented by Normal Exit node) requires that the values of the shared variables as well as those of the formal out parameters (if any) are copied back to the calling function. When an exception occurs, the exception is handled in the called function (currently executing function) if the appropriate handler is available. If appropriate handlers are not available, the handlers are searched in the calling function and this is repeated until an appropriate handler is found. If no suitable handler could be found, then the default handler is invoked. In either case, no return of out parameters occurs from the called function to the calling function.

An exception (represented by Exception Exit node) requires that only the shared variables are copied back to the calling function, as there is no return of formal out parameters in case of an exception. Two different nodes, normal return and exceptional return, are used in the calling function to determine whether the return from a called function was normal or due to some exception respectively. The normal return node signifies that the values of the formal out parameters as well as that of the shared variables are copied back into the calling function, while the exceptional return node specifies that only the values of the shared variables are copied back to the calling function. The start of the guarded block is shown using a try node.

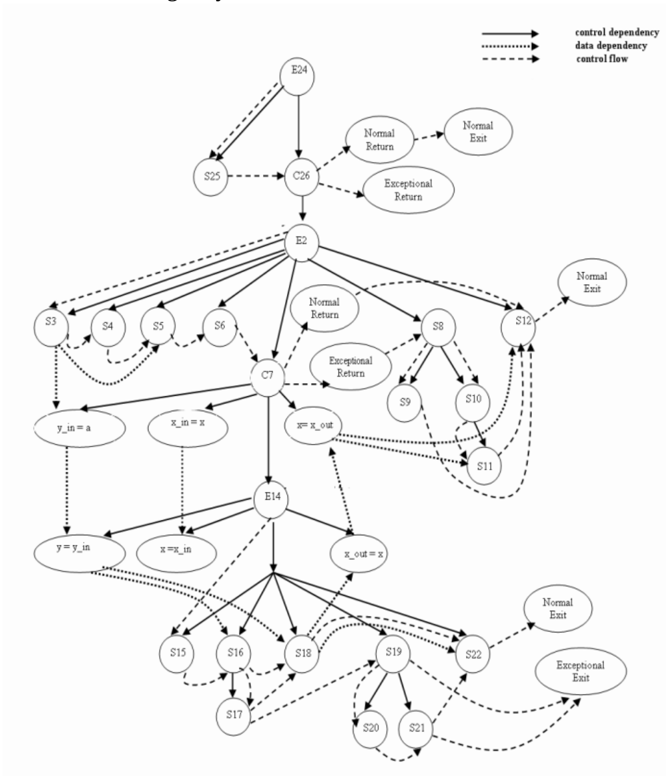


Fig. 2. ClDG augmented with control flow and exception handling for the program in Figure 1.

Figure 2 shows the ClDG augmented with the control flow and exception handling information for the example program shown in Figure 1.

2.1 Representing Priority and Timing Details

An MM-Path (*Method¡MessageP ath*) proposed in [2], [3] represents the sequence in which methods are executed and the corresponding messages invoking these methods. MM-paths capture the order in which different methods are invoked during execution. The information required to identify the various MM-paths in an object-oriented program can be extracted from the UML sequence diagrams. MM-path originates at a method corresponding to user input or other internal event and terminate at methods at which method quiescence (no more method is invoked) occurs.

The specific methods and messages of MM-path for the example program of Figure 1 is given in Figure 3. The message or method information along with the line number of the code is shown. In Figure 3 the scope resolution operator is used to specify the method that is being invoked for each message. In our discussion of representation of timing and priority we consider only the method sequences and not the message sequences. The method sequences can be identified from UML sequence diagrams and by code analysis. In order to identify each method sequence uniquely, a unique identifier is assigned to each method sequence. This identifier is used in labeling the individual edges of the method sequence. Each of the individual edges forming the method sequence is labeled with method sequence identifier. Thus an edge from method *a*() to method *b*() representing an invocation of method *b*() from method *a*() may be labeled with multiple method sequence identifier as there may be several method sequences that has this edge in common. That is several method sequences may follow the same sub paths.

main()	{method,E24}
mA()	{message, C26}
A::mA()	{method,E2}
mB(a)	{message, C7}
B::mB()	{method,E14}
	{return to mA()}
A::mA()	
	{return to main()}
main()	

Fig. 3. The MM-path for the sample program in Fig 1.

We now augment the ClDG with the method sequence information. A new edge 'method sequence edge'(shown in Figure 4) is introduced for representing method sequences. The corresponding nodes of the EClDG and the edges forming the method sequence are shown in Figure 4. The 'method sequence edge' represent the individual edges constituting a method sequence. Each of these edges is labeled with the method sequence identifier of each of the method sequences that has this edge in common.

3. TEST CASE SELECTION FOR REGRESSION TESTING

Regression testing is used when components of the systems evolve or when new components (and functionality) are added to the system. It aims at asserting both that changes are correct and that no regression bugs appear in the system due to the recent evolution. Generally, previous test sequences are launched to guarantee that the system has not regressed in terms of testing quality.

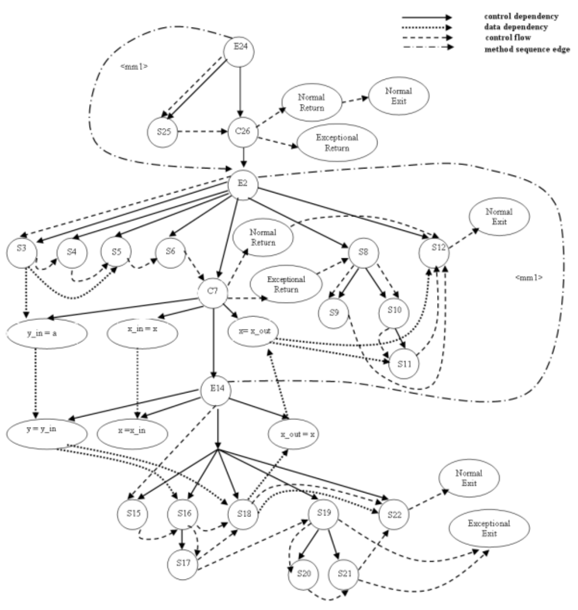


Fig. 4 The method sequence for the sample program in Fig 1.

In order to be sure that no bugs are introduced following any changes to the program, test cases have to be rerun with some new test cases added to a subset of the earlier set. But problem lies in identifying exactly which are the test cases that need to be rerun. Incase some new test cases need to be introduced what would be the criteria based on which these test cases need to be formulated. Running the entire set of test cases in not viable all the time, since the running of all the test cases may consume a lot of time. Often what is seen in the software industry is that often changes are introduced during the final stages of the software release and rerunning the entire test suite is not viable.

The model helps by lending itself to automated analysis and there by determining exactly which all test cases need to be rerun. More over the data dependence and control dependence edges provide us vital information in determining the new test cases that need to be added to the set of test cases. For this we maintain the trace information of each test case when it is run. The trace of a test case provides information concerning the set of statements that are executed during the execution of a test case. Analysis of graphical models of the system preceding and following the regression helps in determining the nodes that are affected following the changes. The test cases can now be selected based on the statements covered by its trace. All the test cases that have at least one of these affected statements in their trace are rerun in a traditional approach to regression testing. The model helps in optimizing the test cases by considering the changes happening in the data dependence relation of the nodes following a regression.

In situation where we need to introduce new test cases to cover newly introduced nodes, the test case generation is based on the data dependence of these nodes with the other nodes of the EClDG. The test cases are generated such that all the data dependence edges are tested.

Consider an arbitrary EClDG having arbitrary nodes N1, N2, N3…Nm. The nodes represented as Nei represent some of the exit or termination nodes in the EClDG. It must be noted that the exit nodes could be exit nodes corresponding to exception exits. Each of the traces corresponding to each test case is represented by T1, T2, T3…Tn. The model helps us identify the test cases to be rerun after a regression. The model created for the regressed program is analyzed for changes with the model created for the program prior to the change. The analysis identifies the nodes in the earlier program that has undergone changes. Now the trace information is analyzed to precisely identify the test cases that have execution trace involving these nodes.

Following are three traces corresponding to three different test cases. The nodes executed by each of the test cases are also listed.

T1: N1 N4 N3 N4 . . . Ni . . .Nk . . . Nj . . . Ne_1

T2: N1 N4 N7 N10 . . .Nk . . . Ne_2

T3: N1 N4 N9 N5 . . . Nj . . .Ni . . . Ne_3

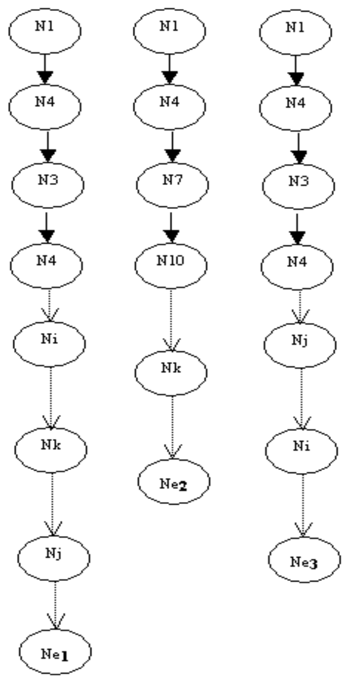


Fig. 5. Trace corresponding to each test case.

Suppose the analysis following the regression identifies the nodes Ni and Nj as being the nodes being affected. This readily provides us with the information that the test cases that need to be rerun are T1 and T2. It is also worth noting that even though a set of nodes may be affected by the regression, it may be possible to identify a subset or a smaller set, which is the dominating set. This is because all the execution paths that pass through some node in the original set may also all the time execute some node in this subset or smaller set. In

Figure 6 if the original set is {Nj, Nk} it is sufficient to analyze the dominating set {Ni}.

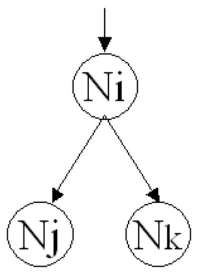


Fig. 6. Execution path.

We now define two relations $\boldsymbol{R}_{pre}$ *(Pre-Dominating)* and $\boldsymbol{R}_{post}$ *(Post-Dominating)* between any two nodes in the graph. The same is extended to be a relation between two sets of nodes. For two nodes *Ni* and *Nj*, *Ni* $\boldsymbol{R}_{pre}$ *Nj* if all the execution paths passing through *Nj* also executes *Ni* before executing *Nj*. For two nodes *Ni* and *Nj*, *Ni* $\boldsymbol{R}_{post}$ *Nj* if all the execution paths passing through *Nj* also executes *Ni* after executing *Nj*. It is possible to define a similar relation $\boldsymbol{R}_{d}$ *(Dominating)* between two set of nodes *S1* and *S2* such that *S1* $\boldsymbol{R}_{d}$ *S2* if all the execution paths executing some node in *S2* also executes some node in *S1*. So for any two set of nodes *R* and *M*, where set *R* represents the nodes that have been affected by the regression and *M* a the minimal set of nodes such that *M* $\boldsymbol{R}_{d}$ *R*. Now it is possible to identify the optimal set of test cases by analyzing just the set M instead of the set R.

Regressions most of the time force a large portion of the test cases to be rerun. Also the fact that regressions occur at later stage of an application development cycle makes it necessary to employ fast means of performing regression testing. It is here that we try to explore the usability of distributed computation to regression testing. It is possible to employ distributed computation at potentially two phases in regression testing that we have discussed. That is in the identification of test cases to be rerun and in the execution of these test cases. Once the minimal set of nodes have been identified the set may be distributed among different nodes which use this set to analyze the test cases exhaustively and exclusively to find the set of test cases to be rerun.

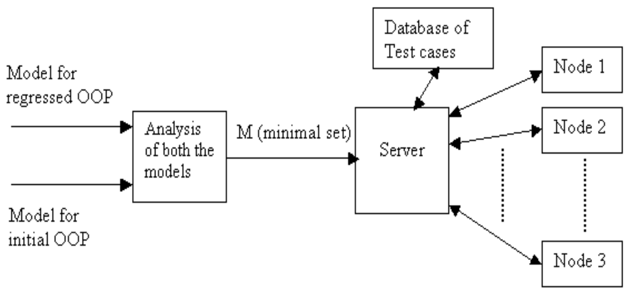


Fig. 7. Identification of test cases

The test cases are then run independently on each nodes and the result is analyzed.

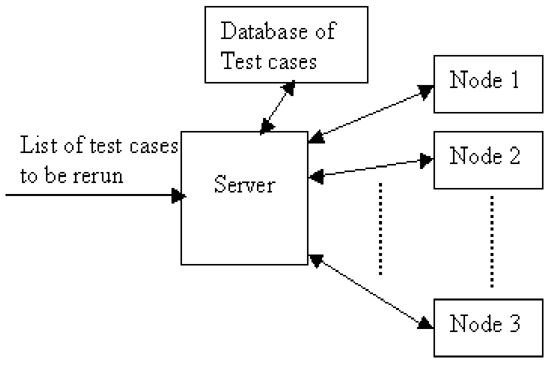


Fig. 8. Execution of test cases

4. RELATED WORK

The representation of intra-functional dependencies in [2], and the concepts of MM-path described in [3], [4] has been incorporated to represent exception handling as well as message sequence information in our model. The Object Oriented testing techniques mostly focuses on the dynamic aspects of the systems viewed at a higher hierarchical level and uses object state modeling or its equivalent which is very in sufficient considering the intricacies involved in determining test cases. Regression testing strategies discussed in [11] are based on component view of the program. We have addressed the same issues at statement level using the model. In [12] the class message diagram is used for test case selection. Though the model helps in test case selection it serves very little purpose is determining the new test cases that need to be added. The representation of data dependence in our model helps in test case generation in case of newly added nodes.

5. CONCLUSION

The proposed method of determining test cases to be rerun uses the proposed model as primary artifact for analysis instead of the source code. The paper demonstrates the building up of model EClDG based on ClDG which incorporates vital information required for identification of test cases. Since regressions force the system to be tested in quick time this is an area where distributed computations need to be used effectively. In this paper we have outlined how the model is used in the identifying the test cases to be rerun and how the distributed architecture can be used in identification of test cases and then to test the system.

6. REFERENCES

[1] 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. 2026, 2009.

[2] Shujuan Jiang, Shengwu Zhou, Yuqin Shi, and Yuanpeng Jiang,"Improving the Preciseness of Dependence Analysis using Exception Analysis", Proceedings of the 15th International Conference on Computing IEEE,pp. 277282,2006.

[3] Ruilian Zhao, Ling Lin, "An UML State chart Diagram Based MMPath Generation Approach for Object Oriented Integration Testing", International Journal of Applied Mathematics and Computer Sciences Volume 3 Number 1.

[4] Paul C. Jorgensen, Carl Erickson, "Object Oriented Integration Testing, Communications of ACM", Vol.37,No. 9, pp. 3038,1994.

[5] Frank Tsui, Orlando Karam and Stanley Iriele, "A Test Complexity Metric Based on Dataflow Testing Technique", Internal Report July, 2008.

[6] L. Larsen and M. J. Harrold, "Slicing object oriented software," in Proc.of the 18th International Conference On Software Engineering, March1996, pp. 495–505.

[7] M. Xenos, D. Stavrinoudis, K. Zikouli and D. Christodoulakis, "Object Oriented Metrics A Survey", Proceedings of the FESMA 2000, Federation of European Software Measurement Associations, Madrid, Spain, 2000

[8] T.J. McCabe, L. A. Dreyer et al., "Testing an object oriented application", Journal of the Quality Assurance Institute, October 1994, pp 2127

[9] Chen, K. and Rajlich, V., "Case study of feature location using dependence graph", Proceedings of IEEE International Workshop on Program Comprehension, Los Alamitos, CA, 2000, pp. 241249

[10] Suresh Nageswaran, "Test Effort Estimation Using Use Case Points", Quality Week 2001, San Francisco, California, USA, June 2001

[11] Yves Le Traon, Thierry Jron, Jean Marc Jzquel, and Pierre Morel, "Efficient Object Oriented Integration and Regression Testing", IEEE Transactions On Reliability, Vol. 49, March 2000, Pg 1225