Validate the Correctness of Object Oriented Program with Regression Testing

Tarun Kumar
Assistant Professor
Vidya College of Engineering
Bagphat Road, Meerut (India)

Mayank Singh
Associate Professor
Krishna Engineering College
Mohan Nagar, Ghaziabad

Arun Sharma
Professor & Head, CSED
KIET, Ghaziabad (India)

ABSTRACT

Regression testing is used to validate the correctness of upgrades version of any program or software. The newly introduced features in the system under test are compared with the existing versions which determine the proper implementation of regression testing. The verification is done in a way that the modification made in the SUT does not interfere with the existing features, in this paper two program with new version of each are being put to regression testing with old and new test cases to check the satisfiability of regression testing. Software developers often face the challenge of projecting the difference in behaviour of one version of a program unit as compared to the upgraded on of the same program unit, for such situations, the developers need to generated test cases between the existing and upgraded version, if any exist.

Keywords

SUT, Regression Testing

1. INTRODUCTION

In the software development life cycle, regression testing comes under software testing phase and is an important activity which is used to validate the modification and to ensure that no other parts of the program have been affected by the change. The retesting of a program or software to verify that the change have no lead the software to fail and the product or software still complies or successful run with its specified requirements [1]. Software is said to regress if 1) a new modules or component is added, or 2) a modification done to the existing modules affects other parts of the program. Therefore, it is essential to retest not only the changed code but also to retest the possible affected code due to the change. Due to regression testing activity, extra cost includes in development cost and typically accounts for half of the total cost of software maintenance [2].

The environments in which the changed software is tested frequently, regression test selection techniques [2]. For example, consider an environment in which nightly builds of the software are performed and a test suite is run on the newly built version of the software. Under the regression test selection approach, a small subset of the test suite is selected for testing the new version of the software. This enhance the effectiveness of the testing the time required for testing strategy by reducing the time required for testing due to the utilization of a subset of the test suite.

For instance, considering a development environment that include a regression test selection components, where in the developers often modifying their software can be the regression test selection to select a subset of the test suite for testing. This strategy helps in locating errors during the early stage of development as the developers can frequently test their software while making the changes in the software [2]. The techniques are also effective when the cost of test cases is high. For example, reduction of one test case in the regression testing of avionics software may result in saving one thousands of dollars in the testing resources.

2. HOW TO DO REGRESSION TESTING

Regression testing is done by well defined methodology because it is the final type of testing i.e. normally performed just before release the software. The following steps for regression testing are:

- Initially perform an initial “smoke” or “sanity” test.
- Design the best test case which id based on well defined criteria.
- categorize the test cases
- Methodology for each selected test cases.
- Retest the test cases to ensure the impact the modification.

3. HISTORY

The object oriented concept like classes, object, abstraction, encapsulation, inheritance, dynamic binding, and polymorphism gives the new idea for software development and also give the idea about the relationship between classes and their attributes [2]. They not only introduce new testing problems as recognized in Harrold et al. (1992), Perry and Kaiser (1990), Smith and Robson (1990), and Wild and Huitt (1991), but also raise a new and Challenging question of how to conduct regression Testing for object oriented programs. Regression testing can be found in Leung and White (1989) and Hartman and Robson (1988). The four fundamental problems:

To find out the affected on the function or part of the program automatic is the first problem. Harrold and Soffa (1988) proposed a method to study the change effects within a module. The concept behind the analysis of impact of change in module is to use of a data flow graph. The benefit of this methodology is that to save time and test effort because by retesting only the affected define-use paths and new paths. The technique has been extended so that it can also be used to identify affected procedures at the inter-procedural level (Harrold and Soffa, 1989) [1]. A lot of techniques are proposed by the researchers based on a control flow graph of a procedure/function to identify the affected control paths in a module (Laski and Szermer, 1992; Prather and Meyers, 1987) [3].

The second problem of regression testing comes under cost benefits analysis. And concern with the how we can design effective test case for conducting retests so that test effort and costs are minimized. Testing divided into three categories first top-down in which we start the testing at the root and second bottom-up ,where we start the testing from bottom and third sandwich approaches (Bezier, 1990), where we test parallel form bottom to up and from top to down., These come up to rely on the tester to make the selection. Prather and Myers (1987) proposed an adaptive-path prefix software-testing strategy that used previous test paths as a guide in the selection of subsequent paths. Their method ensures branch coverage and consumes fewer computational resources Harrold et al. (1992) [4] presented an incremental testing methodology based on class inheritance hierarchy. The approach suggests that the base class should be tested before derived classes so that the test cases and relevant information of the base class can be reused in testing the derived classes.

The third problem of regression testing concerns coverage criteria. Fischer [5-6] (1977; Fischer et al. 1981) and Prather and Myers (1987) [7], respectively, described the various retest criteria relating to path coverage of a function/procedure. Leung and White (1990a; White and Leung, 1992) used firewalls as a retesting criteria at module level to ensure that all affected modules and links between modules will be retested.

The selection, reuse, and modification of existing test cases for retesting are the fourth problem. The test case selection problem discussed by Fischer (1977; Fischer et al.; 1981) which is based on the set covering problem. Unit regression testing covers one of the path criteria which are based on the concept of 0-l integer-programming models to find the minimum test cases. And same concept of 0-1 integer-programming model on a test matrix to minimize test efforts in functional regression testing used by Lee and He, (1990) [8] also used the. Leung 1991 and White proposed corrective regression testing in which a retest strategy for performing is used. Two sub problems comes to view regression testing: the test selection problem i.e. a good test case is one which can find out the maximum number of error and the test plan update problem. Thus, the retesting process is divided into two phases: fist phase to classification of test case and second to update the test plan. After the existing test cases are classified into reusable tests, obsolete tests, and retest able tests in the test classification phase, during the regression testing new test cases are considered for testing.

4. REGRESSION TEST SELECTION ALGORITHMS

Although object-oriented languages have been available for some time, only two safe regression-test-selection algorithms that handle features of object-oriented software have been developed [9-10]. However, both approaches are limited in scope and can be imprecise in test selection. Rother- Mel, Harrold, and Dedhia's algorithm [9] was developed for only a subset of C++, and has not been applied to software written in Java. The algorithm does not handle some features that are commonly present in object-oriented languages; in particular, it does not handle programs that contain exception-handling constructs. Furthermore, the algorithm must be applied to either complete programs or classes with attached drivers. For classes that interact with other classes, the called classes must be fully analysed by the algorithm. Thus, the algorithm cannot be applied to applications that call external components, such as libraries, unless the code for the external components is analysed with the applications. Finally, because of its treatment of polymorphism, the algorithm can be very imprecise in its selection of test cases. Thus, the algorithm can select many test cases that do not need to be rerun on the modified software White and Abdullah's approach [10] also does not handle certain object-oriented features, such as exception handling. Their approach assumes that information about that classes that have undergone specification or code changes is available. Using this information, and the relationships between the changed classes and other classes, their approach identifies all other classes that may be affected by the changes; these classes need to be retested. White and Abdullah's approach selects test cases at the class level and, therefore, can select more test cases than necessary

This paper presents the safe regression-test-selection technique for C++ using two programs in C++ that efficiently handles the features of the Java language, such as encapsulation, polymorphism, inherientance, dynamic binding, data abstraction and exception handling. Our technique is an adaptation of Rother Mel and Harold's graph-traversal algorithm [9] [11], which use a control-flow-based representation or algorithms of the original and modified versions of the software to select the test cases to be rerun. Our new algorithm efficiently represents C++ language features, and modified algorithms or LOC in program safely selects all test cases in the original test suite that may reveal faults in the modified software. Thus, unlike previous approaches, our technique can be applied to common commercial software or program written in objected oriented language like C++ or others.

5. REGRESION TESTING

Our analysis assume the existence of an original program P and a changed program P' derived from P. both p and P' are assumed to be syntactically correct and compliable. But we impose no restrictions on the number or the nature of the changes than transform P into P'. We assume that an idea provides information about the files, cases and method that have been edited. Alternatively, one can rely on a utility like different to obtain this information.

The program, P, the modified version of P is P', and a test suite is T.

Leung and White categorise test cases into five classes. The first three classes consist of test cases that already exist in T.

- **Reusable:** Reusable test cases only execute the parts of the program that remain unchanged between two versions, i.e. the parts of the program that are common to P and P'. It is unnecessary to execute these test cases in order to test P'; however, they are called reusable because they may still be retained and reused for the regression testing of the future versions of P.
- **Retest able:** Test cases execute the parts of P that have been changed in P'. Thus retest able test cases should be re-executed in order to test P'.
- **Obsolete:** Test cases can be rendered obsolete because 1) their input/output relation is no longer correct due to changes in specifications, 2) they no longer test what they were designed to test due to modifications to the program, or 3) they are 'structural' test cases that no longer contribute to structural coverage of the program. The remaining two classes consist of test cases that have yet to be generated for the regression testing of P'.
- **New-structural: N**ew-structural test cases test the modified program constructs, providing structural coverage of the modified parts in P'.
- **New-specification:** New-specification test cases test the modified program specifications, testing the new code generated from the modified parts of the specifications of P' [12]

Program 1: sqrt.cpp (P)

```
#include<iostream.h>
#include<stdio.h>
#include<conio.h>
class Square
{ public:
float Square_root(float q);
};
float Square:: Square_root(float q)
{ float i , j;
i = q;
do
{ j = i;
i= (i + q/i) / 2; }
while( i!= j);
printf("%f",a);
return(i);   }
int main()
{ float q;
float q;
Square s;
clrscr();
cout<<" enter any number \n";
cin>>q;
if(q > 0){
cout<<s.Square_root(q);   }
else {
cout<<s.Square_root(-q)<<"i";    }
getch();
return 0;
}
```

Program 2: new version of sqrt.cpp (P')

```
#include<iostream.h>
#include<stdio.h>
#include<conio.h>
class Square
{  public:
float Square_root(float q);
};
float Square:: Square_root(float q)
{float i , j;
i = q;
do
{ j = i;
i = (i + q/i) / 2; }
while( i!= j);
printf("%f",i);
return(i);   }
int main()
{ float q;
float q;
Square s;
clrscr();
cout<<" enter any number \n";
cin>>q;
if(q > 0){
cout<<s.Square_root(q);   }
else if (q<0) {
cout<<s.Square_root(-q)<<"i";    }
else
cout<<" Square root is zero";
getch();
return 0;
}}
```

Program 3: Ackrmen.cpp

```
#include <iostream.h>
#include<conio.h>
int ackerman_number(int l, int m)
{
if (l == 0)
return m+1;
else return ackerman_number(l-1,
ackerman_number(l, m-1));
}
int main()
{int l,m;
cout << "l and  m";
cin>>l;
cin>>m;
cout<<ackerman_number(l,m);
getch();
return 0;
}
```

Program 4: New version of Ackrmen.cpp

```
#include <iostream.h>
#include<conio.h>
int ackerman_number(int l0, int m)
{
if (l == 0)
return m+1;
else if (m == 0)
return ackerman_number(l-1, 1);
else return ackerman_number(l-1,
ackerman_number(l, m-1));
}
int main()
{
int l,m;
cout << "l and  m";
cin>>l;
cin>>m;
cout<<ackerman_number(x,y);
getch();
return 0;
}
```

The initial version of the program 1 sqrt.cpp was tested using the following test suite T;

Test case	**input value**	Actual Output	Expected Output
T1	36	6	6
T2	-36	6i	6i
T3	***0***	***-NaN***	***0***

In case of test case T3, the actual output is different from expected output. So here we need to focus and analyse the module through which T3 is executed. After analyse the source code do the modification in source code and generated the new version of the said program (i.e. program 2).
Now modify the program (as highlighted in program 2) to address the following:

```
1.  else if (x<0) {
2.  cout<<s.Square_root(-x)<<"i";   }
3.  else
4.  cout<<" Square root is zero";
```

The comparison of the old and the new versions of the program sqrt.cpp will produce the following diff output:

Test case	**input value**	Actual Output	Expected Output
T1 T2	No need to execute again		
T3	***0***	***0***	***0***

In this manner test case T3 run successfully with new version of the program sqrt.cpp the initial version of the program **Ackrmen.cpp** was tested using the following set *T* of test cases:

Test case	**input value x**	Actual Output y	Expected Output	
T1	0	2	3	3
T2	***1***	***2***	***Fail***	***4***
T3	***2***	***0***	***Fail***	***3***
T4	*0*	*0*	*1*	*1*

In case of test case T2, T3, the actual output is different from expected output. So here we need to focus and analyse the module or path through which T3 is executed. After analyse the source code do the modification in source code and generated the new version of the said program (i.e. program 3). Now modify the program (as highlighted in program 4) to address the following:

1. ***else if (y == 0)***
2. ***return ackerman_number(x-1, 1);***

The comparison of the old and the new versions of the program **Ackrmen.cpp** will produce the following different output:

Test case	**input value x**	Actual Output y	Expected Output	
T1 T4	No need to execute again			
T2	***1***	***2***	***4***	***4***
T3	***2***	***0***	***3***	***3***

In this manner test case T2, T3 run successfully with new version of the program **Ackrmen.cpp**.

6. AVERAGE PERCENTAGE FAULT DETECTION (APFD) METRIC

Average Percentage of Faults Detected (APFD) metric [1] was used to determine the effectiveness of the new test case orderings, but it considered faults and test cases cost to be uniform. To measure the average rate of fault detection of a regression test suite the average percentage of faults detected (APFD) metric was proposed by Rother Mel et al. [13]. The APFD metric has been used by several researchers [14-15] to evaluate the effectiveness of a test prioritization scheme. The APFD metric for a test suite is calculated by taking the weighted average of the number of faults detected during execution of the program with the test suite. APFD metric values range from 0 to 100 and a higher number indicates a faster rate of fault detection. The APFD metric can be calculated using the following expression. Let T be the original test suite containing n test cases, and let F be a set of m faults revealed by T. Let T0 be an ordering of T. In T0, let TFi be the first test case which reveals a fault i. Then the APFD metric for test suite T0 can be obtained by using the equation n is the number of test case and m is the number of fault [4]:

$$APFD = 1 - \frac{TF1 + TF2 + TF3 + \cdots \ldots \ldots \ldots + TFn}{n * m} - \frac{1}{2n}$$

For Program Sqrt.cpp

APFD =1-[1/ (3*1)] -1/2*3
= 0.55
=55%

For Program **Ackrmen.cpp**

APFD =1-[(1+1)/ (4*2)] -1/2*4
= 0.625
=62.5%

7. IMPACT ON FAULT DETECTION CAPABILITY

The effectiveness of the minimisation itself was calculated as follows [9]:

$$1 - \frac{Number\ of\ Test\ cases\ in\ the\ reduced\ test\ suite}{Number\ of\ test\ cases\ in\ the\ original\ Test\ suite}$$

For program sqrt.cpp (new and old version)

*= (1-1/3)*100*
=66.66%

For program **Ackrmen.cpp**

*= (1-2/4)*100*
=50.00%

The impact of test suite minimisation was measured by calculating the reduction in fault detection effectiveness as follows:

$$1 - \frac{Number\ of\ fault\ detected\ by\ the\ reduced\ test\ suite}{Number\ of\ fault\ detected\ by\ the\ original\ Test\ suite}$$

For program sqrt.cpp (new and old version)

*= (1-0/1)*100%*
=100%

For program **Ackrmen.cpp** (new and old version)

*= (1-0/2)*100%*
=100%

8. TESTING TOOL SUPPORT

In future, regression testing performs by one of the automated testing tool, which is explained below. Every tool has a similar structure of the description. It contains firstly in the header line: Name of the tool, Company name. Then the description begins with one sentence, which explains the main scope of the tool [16-18].

1. CitraTest, Tevron, LLC.
This tool is ideal for latency, functional, and regression testing.
2. Rational Robot, Rational Software Corp, Allows user to create, modify, and run automated Functional, regression, and smoke tests for e-applications.
3. preVue-ASCEE, Rational Software Corporation,
Used in: Assurance, Performance measurement and regression testing.
4. preVue-X, Rational Software Corporation,
Used in: Regression and Performance testing.
5. Teleprocessing Network Simulator,
Used in: Performance, function, & automated regression testing.
6. Silk Pilot, Segue Software, Inc., www.segue.com
Used in: Functional and regression testing of middle-tier servers.
7. CHILL/C/C Pilot, Kvatro Telecom AS,
Used in: Programmable debugger and conformance/ regression tester for CHILL/C/C++ programs.
8. SMARTS, Software Research, Inc.,

Used in: Maintenance and regression testing. STW/Regression for Windows, Software Research, Regression testing tool.
9. AQtest, AutomatedQA Corp.,
Used in: Automated support for functional, unit, and regression testing.

9. CONCLUSION AND FUTURE SCOPE

In this paper two programs (sqrt.cpp & ackerman.cpp) are discussed and regression testing has been performed for both programs with the old test cases as well as new test case. Analysis of impact of fault detection was done and value in percentage is 100% for both the program with some modification where as in previous program value was 66.66% and 50.00 respectively. Average percentage fault detection (APED) for both program is 55.00% and 62.55 respectively. Here test case generates manually but in future test case will be generated through testing tool which is explained in above section VIII. Maintaining a software structure is a high-priced, composite and ever running activity. Among the many activities executed to prevent and improve to harmfully impact the quality of a system, regression testing is the most commonly used technique. When tests reveal a failure, developers have to analyse the execution to understand the causes of the failure to fix the associated fault.

10. REFERENCE

[1] Laski, j., and Szermer, w., “Identification of Program Modifications and Its Applications in Software Maintenance”, in proc. Conf. Software maintenance, 1992, pp. 282-290.

[2] David C. Kung, Jerry GAO, and Pei Hsia, “ On Regression Testing Of Object-Oriented Programs”, j. Systems software 1996; 32:21-40 0 1996 by Elsevier Science inc. 655 avenue of the Americas, New York, NY 10010, ssdi 0164-1212(95)00047-5.

[3] Harold, M. J., McGregor, j. D., and Fitzpatrick, k. J., “Incremental Testing of Object-Oriented Class Structure”, in Proc of 14 international conf. on software engineering, 1992.

[4]] G. Rothermel and m. J. Harrold, “Analysing Regression Test Selection Techniques,” IEEE transactions on software engineering, vol. 22, no. 8, pp. 529–551, 1996.

[5] Fischer, k. F., “A Test Case Selection Method For The Validation Of Software Maintenance Modifications”, in IEEE compsac 77 int. Conf. Procs., 1977, pp. 421-426.

[6] Fischer, K. F., Raji, F., and Chruscicki, A., “A Methodology For Re-Testing Modified Software”, in national telecoms. Conf. Procs., 1981, pp. B6.3.1-6.

[7] Prather, R. E., and Myers, J. P., Jr, “The Path Prefix Software Testing Strategy”, IEEE trans. Software eng. Se-13 (1987).

[8] Lee, j. A. N., and he, xudong, a methodology for test selection, j. **Syst. Software,** 13, 177-185, (1990).

[9] G. Rother Mel, M. J. Harrold, and J. Dedhia., “Regression Test Selection For C++ Software”. Journal of software testing, veri_cation, and reliability, 10(6):77{109}, jun. 2000.

[10] L. J. White and K. Abdullah., “A _Rewall Approach For Regression Testing Of Object-Oriented Software”, In Pro-ceedings of 10th Annual Software Quality Week, May1997.

[11] Scott McMaster and Atif M. Memon, “Fault Detection Probability Analysis For Coverage-Based Test Suite Reduction”, In proceedings of the 23rd international conference on software maintenance, 2007.

[12] Leung Hkn, White l., “Insight Into Regression Testing”, Proceedings of international conference on software maintenance (ICSM 1989), IEEE computer society press, 1989; 60–69.

[13] Rother Mel G, Untch R, Chu C, Harold M, “Prioritizing Test Cases For Regression Testing”. IEEE trans software eng 27(10):929–948, 2001.

[14] Elbaum S, Malishevsky A, Rother Mel G, “Incorporating Varying Test Costs And Fault Severities Into Test Case Prioritization”, In: proceedings of the 23rd international conference on software engineering, pp 329–338, Ontario,2001.

[15] Jeffrey D, Gupta N, “Experiments With Test Case Prioritization Using Relevant Slices” J syst softw 81:196–221.2008.

[16] Fewster, M., Graham, D., “Software Test Automation”. ACM press, New York, 1999.

[17] Tervonen, I., Katselmointi JA Testaus., Lecture notes in University of Oulu, 2000.

[18] Pentti Pohjolainen, “Software Testing Tools”, Department of Computer Science and applied mathematics, University of Kuopio march 2002.

[19] Chhabi Rani Panigrahi, Rajib Mall, “An Approach To Prioritize The Regression Test Cases Of Object-Oriented Programs”, CSI Publications 2013, csit (June 2013) 1(2):159–173 doi 10.1007/s40012-013-0011-7.

[20] S. Yoo, M. Harman, "Regression Testing Minimisation, Selection And Prioritisation: A Survey", software testing, verification and reliability Softw. Test. Verify. Reliab. 2007; 00:1–7 (doi: 10.1002/000) published online in wiley interscience (www.interscience.wiley.com). Doi: 10.1002/000

[21] Kristen r. Walcott et al,” time aware test suite prioritization”, ACM, issta’06, Portland, Maine, USA, July 17–20, 2006.

[22] Bo Qu Changhai Nie et al, “Test Case Prioritization For Multiple Processing Queues”, issue-08, pg. 646-49 IEEE, 2008.

[23] Jonathan Misurda, James A. Clause, Juliya l. Reed, Bruce R. Childers, and Mary Lou Soffa, “ Demand-Driven Structural Testing With Dynamic Instrumentation”. In proceedings of the 27th international conference on software engineering, 2005.

[24] Romain Delamere, Benoit Baudry, Yves Le Traon, “Regression Test Selection When Evolving Software With Aspects”, Proceedings of late workshop in conjunction with aosd'08 (2008).

[25] Swarnendu Biswas, Rajib Mall, "Regression Test Selection Techniques: A Survey", informatics 35 (2011) 289–321.