INTERNATIONAL JOURNAL OF INNOVATIVE RESEARCH & DEVELOPMENT

ISSN 2278 – 0211 (Online)

A Practical Approach to Perform Software Testing Using Equivalent Mutants

Simrandeep Kaur
Department of CSE, Chandigarh University, CGC, Gharuan, Punjab, India
Rupinder Singh
Assistant Professor, Department of CSE
Chandigarh University, CGC, Gharuan, Punjab, India

Abstract:
In software testing, different testings are found. Software Testing using mutants is called as mutation testing. The impact of software testing on software maintenance is problematic and is of great weightage. So the need for identification of mutants different operator is necessary and this process is called u-java testing. Various techniques are proposed, that are quite efficient in providing and finding mutants in different viewer .The approaches mainly followed for clone detection are comparison based on text and token based approach. These techniques take a huge amount of time and the process of comparison is very tedious and expensive. Moreover tree based techniques are very complex. The present study addresses a proposed approach which can be applied as byte code to reduce the complexity encountered with the previous techniques .Test sets are designed in order to run mutants and analyse uses byte code to calculate killed mutants. The reason of using byte code is that it is platform independent and represents the unified structure of the code. So the proposed approach also detected semantic mutants up to some extent. This approach can be used independently and it can be combined with other approaches to detect mutants

Keywords*: software testing, mutants, regression testing, equivalent ,test set, operator*

1. Introduction

The accuracy and quality of a software product depends largely testing i.e how thoroughly it is tested [6]. As the complexity and size of software products grow with every passing year, the time and effort required for adequate testing is growing at a very rapid rate. Test case design constitutes a large part of testing cost .The testing phase basically testing has to performed in order increase the reliability of software product[12].

It is generally come to term that manual testing is becoming a congestion and is a frequent cause of project delays especially for large programs [7].Therefore, automatic test case design has become important to assure the quality of present day large software products and to contain the rapidly growing testing costs.

Test case generation from code is inefficient, especially for large programs. Some aspects of program behaviour e.g. state behaviour are very difficult to test, based on code alone. Also different approach is to generate test cases [23, 12, 9] from UML models constructed during the design process in order to perform testing at early level before development part accomplished. By using this type of approach it ensures great success rate to testing because it is time saving and speedy. Use of UML models to generate test cases holds out several advantages. Test case generation from design help in evaluating the rate of progress is high.

There are huge numbers of different target languages and purposes. For example, one back-end could generate Java byte code, while another could generate C++[4]. Furthermore, even within the same target language, different back-ends can generate different kinds of code. For example, one C language back-end might generate a parser optimized for speed, while another might generate code optimized for power-efficiency on a mobile device. Even with the same target language, an optimization on one hardware architecture may be a upgrade on a different one, suggesting that different code be generated for each architecture [10].

Similarly to how different compiler front-ends can generate the same intermediary language from different source languages (like the architecture for GCC), we can also front-ends for different schema languages, as shown in Figure 1. Our current work focuses on the XML Schema language.

Each back-end provides a specific set of actions, predicates, and variables that together define an abstract processor.

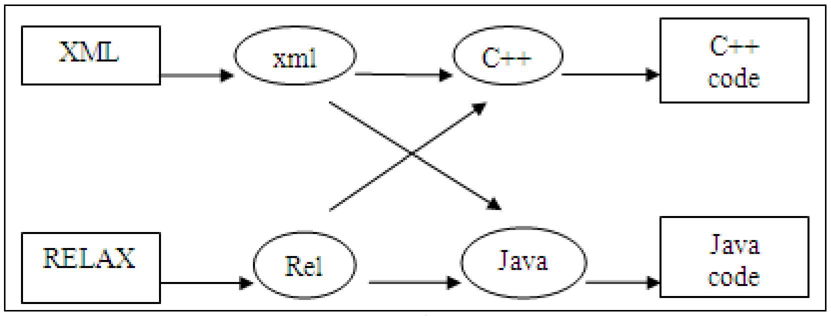


Figure 1: Our architecture divides the compilation process into a front-end and back-end, with an intermediate representation used in between. This allows different front-ends to work with different back-ends. Round shapes represent processes, while rectangular shapes represent data [8].

Software maintenance animation on an average, account for as much as two-thirds of the overall software life cycle development cost. Maintenance of a software product is frequently necessitated to fix defects, to add, enhance or adapt existing functionalities, or to port it to different environments.
Whenever an application program is modified for carrying out any maintenance activity and test cases are designed which are highly decisionable.
The execution test set to check that the modified parts of the code work properly. *Regression* testing (also referred to as *program revalidation*) is carried out to ensure that no new errors (called regression errors) have been introduced into previously validated code (i.e., the unmodified parts of the program) [13] and aim to check change in behaviour while executing the program.
Test case selection **-** This involves identification of a subset of test cases from the initial test suite *T* which can *effectively* test the unmodified parts of the program .The aim is to be able to select the subset of test cases from the initial test suite that has the potential to detect errors induced on account of the change[2].

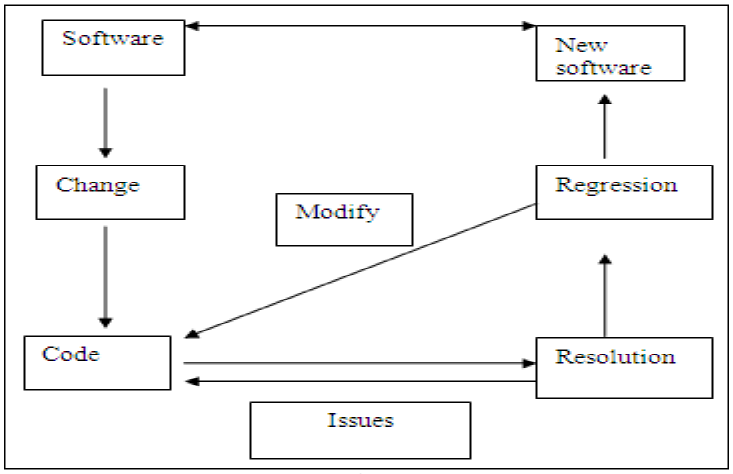


Figure 2: Activities that take place during software maintenance and regression testing [12]

The approaches that are most often used in the industry for analyse of appropriate regression test cases are either based on expert judgment, or based on some form of manual and automatically program analysis. However, selection of test cases based on expert judgment tends to become ineffective and unreliable for large software products.
Software made with collaboration of number of complex program using predicate, variable and operator. Some program code written such way that it ultimately generate equivalent mutants or non-equivalent mutants. Example:
Sample Program A
if(x == 2 && y == 2)
z = x + y;
Mutant A'
if(x == 2 && y == 2)
z = x * y;
The value of Z will be equal to 4; any test set will be unable to determine any faults with this program because the value will always be equal to 4.
The clone technique has been used in order to retrieve the number of different operator. In order to find out clone by comparing different program together specially the text in program and evaluation has to made .the basically involve the analysis of code such lexical and checking local dependence of code. Various class levels metric on the basis of analysis. Similarly the function level metric

calculated are also compared to find the similarity. And after analyzing the results potential clones are detected. And this approach is very efficient in providing potential clones [1].

2. Proposed Work

A mutation tool that is based on byte code or on nay translation that may result faster than any other manual testing tools. In today scenario, Automation testing performs an important role in software industries. Today automation is need of hour. Testing not only increased the accuracy of software but also represent support with reliability and the quality of software. Mutant testing works by seeding faults in the software program. Various mutation operators are used to create these faulty programs. These programs are called mutants. The mutants depict software faults that may be caused by programmers while writing the software. Test cases are then executed on these mutants to determine if they have been *killed* or not. Test sets that kill all the mutants are considered to be good as they successfully detect all the possible program faults. Class level and operator level mutation operators are applied to generate a large number of mutants and are the most critical factor as the time required for mutation testing is totally dependent on these two factors. Equivalent mutation detection technique improves the efficiency of mutation testing and reduces the time required to perform it. The program under test is Original program. A mutant operator is rule for changing mutant operator into a create mutants. Following are the, methodology steps which are involved during software testing:

2.1. Select a program i.e. source code and run the program

2.1.1. The source code must save by using *.java* extension

2.1.2 .Before run the source code files should be compiled in order to remove errors sand .class file easily generated

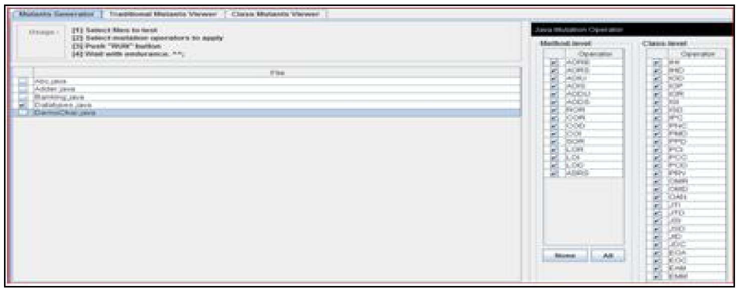
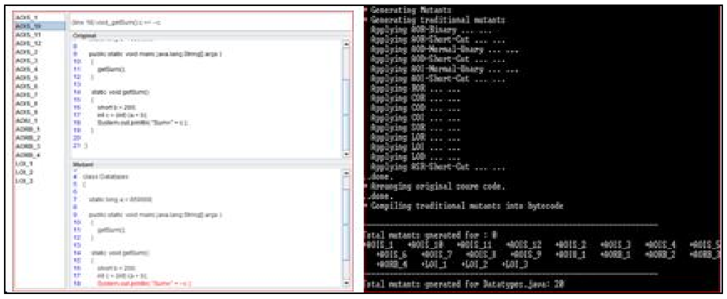


Figure 3: Select .Java File

2.2. After class file has been generated execution, start executing the command in order to perform software testing

2.3. Select the .java source file and run source file in order to retrieve the different operator

2.3.1. The operator like inheritance, encapsulation, java specified operator and other operator also like AOIS, JID, IHD and LIO.

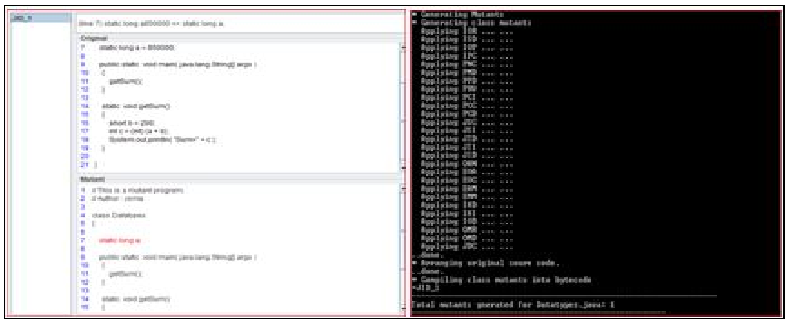
.


```
    public static void main( java.lang.String[] args )
    {
        getSum();
    }

    static void getSum()
    {
        short b = 200;
        int c = (int) (a + b);
        System.out.println( "Sum=" + c );
    }

}
```

```
class Datatypes
{

    static long a = 450000;

    public static void main( java.lang.String[] args )
    {
        getSum();
    }

    static void getSum()
    {
        short b = 200;
        int c = (int) (a + b);
        System.out.println( "Sum=" + -c );
```

```
* Generating Mutants
* Generating traditional mutants
  Applying AOR-Binary ... ...
  Applying AOR-Short-Cut ... ...
  Applying AOD-Normal-Unary ... ...
  Applying AOD-Short-Cut ... ...
  Applying AOI-Normal-Unary ... ...
  Applying AOI-Short-Cut ... ...
  Applying ROR ... ...
  Applying COR ... ...
  Applying COD ... ...
  Applying COI ... ...
  Applying SOR ... ...
  Applying LOR ... ...
  Applying LOI ... ...
  Applying LOD ... ...
  Applying ASR-Short-Cut ... ...
..done.
* Arranging original soure code.
..done.
* Compiling traditional mutants into bytecode

------------------------------------------------------------
Total mutants generated for : 0
*AOIS_1   *AOIS_10   *AOIS_11   *AOIS_12   *AOIS_2   *AOIS_3   *AOIS_4   *AOIS_5
  *AOIS_6   *AOIS_7   *AOIS_8   *AOIS_9   *AOIU_1   *AORB_1   *AORB_2   *AORB_3
  *AORB_4   *LOI_1   *LOI_2   *LOI_3
------------------------------------------------------------
Total mutants generated for Datatypes.java: 20
```

Figure 4: Traditional mutants Generation



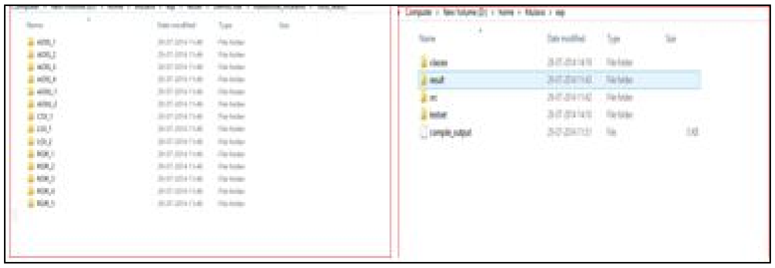
```
(line 7) static long a800000 => static long a
Original
     static long a = 800000;

     public static void main( java.lang.String[] args )
     {
         getSum();
     }

     static void getSum()
     {
         short b = 290;
         int c = (int) (a + b);
         System.out.println( "Sum=" + c );
     }

 }
Mutant
 // This is a mutant program.
 // Author : ysma

 class Datatypes
 {

     static long a;

     public static void main( java.lang.String[] args )
     {
         getSum();
     }

     static void getSum()
     {
```

```
* Generating Mutants
* Generating class mutants
  Applying IOR ... ...
  Applying ISI ... ...
  Applying IOP ... ...
  Applying IPC ... ...
  Applying PNC ... ...
  Applying PMD ... ...
  Applying PPD ... ...
  Applying PRV ... ...
  Applying PCI ... ...
  Applying PCC ... ...
  Applying PCD ... ...
  Applying JSC ... ...
  Applying JSI ... ...
  Applying JTD ... ...
  Applying JTI ... ...
  Applying JID ... ...
  Applying OAN ... ...
  Applying EOA ... ...
  Applying EOC ... ...
  Applying EAM ... ...
  Applying EMM ... ...
  Applying IHD ... ...
  Applying IHI ... ...
  Applying IOD ... ...
  Applying OMR ... ...
  Applying OMD ... ...
  Applying JDC ... ...
..done.
* Arranging original soure code.
..done.
* Compiling class mutants into bytecode
<JID_1
------------------------------------------------------------
Total mutants generated for Datatypes.java: 1
```

Figure 5: Generation of Class Mutants

2.4. MUTANTS are generated such as traditional mutants and class mutant Also method are there to imply on the both mutants(traditional and class)

<u>2.4.1. The .java source file has been saved and followed the path */exp/src/.javafile.*</u>
And .class file must have followed the path */exp/classes/.class file.*



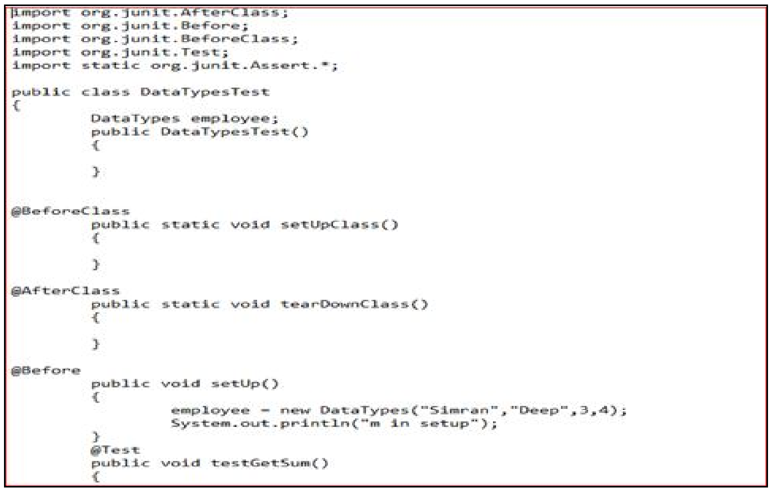
```
Computer › New Volume (D:) › home › Mutants › exp

Name              Date modified      Type         Size
classes           25-07-2014 14:19   File folder
result            25-07-2014 11:42   File folder
src               25-07-2014 11:42   File folder
testset           25-07-2014 14:19   File folder
compile_output    25-07-2014 11:51   File         1 KB
```

Figurer 6: Resultant Mutant

<u>2.4.2. The mutants have also been stored in /exp/ result /.with object-oriented mutants in class mutants and traditional mutants in a separate directory</u>

2.5. Now, Run the mutants from another GUI.

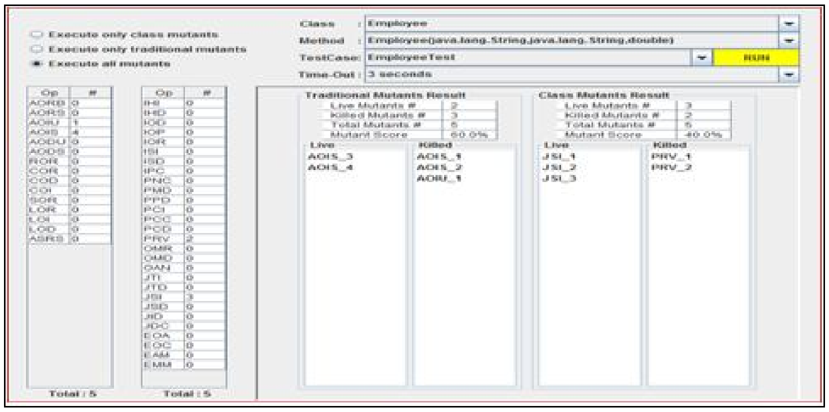
<u>2.5.1 Test set has to design in such way that what input kills it i.e it help in analyse the source code program and finding the live mutants.</u>



```
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;

public class DataTypesTest
{
        DataTypes employee;
        public DataTypesTest()
        {

        }

@BeforeClass
        public static void setUpClass()
        {

        }

@AfterClass
        public static void tearDownClass()
        {

        }

@Before
        public void setUp()
        {
                employee = new DataTypes("Simran","Deep",3,4);
                System.out.println("m in setup");
        }
        @Test
        public void testGetSum()
        {
```

Figure 6: Design Test set

NOTE: test set has to design (not generated). We have to carefully design the test set in order to find out live and killed mutants



( ) Execute only class mutants
( ) Execute only traditional mutants
(●) Execute all mutants

Class : Employee
Method : Employee(java.lang.String,java.lang.String,double)
TestCase: EmployeeTest
Time-Out : 3 seconds

Op	#
AORB	0
AORS	0
AOIU	1
AOIS	4
AODU	0
AODS	0
ROR	0
COR	0
COD	0
COI	0
SOR	0
LOR	0
LOI	0
LOD	0
ASRS	0

Total : 5

Op	#
IHI	0
IHD	0
IOD	0
IOP	0
IOR	0
ISI	0
ISD	0
IPC	0
PNC	0
PMD	0
PPD	0
PCI	0
PCC	0
PCD	0
PRV	2
OMR	0
OMD	0
OAN	0
JTI	0
JTD	0
JSI	3
JSD	0
JID	0
JDC	0
EOA	0
EOC	0
EAM	0
EMM	0

Total : 5

Traditional Mutants Result

Live Mutants #	2
Killed Mutants #	3
Total Mutants #	5
Mutant Score	60.0%

Live	Killed
AOIS_3	AOIS_1
AOIS_4	AOIS_2
	AOIU_1

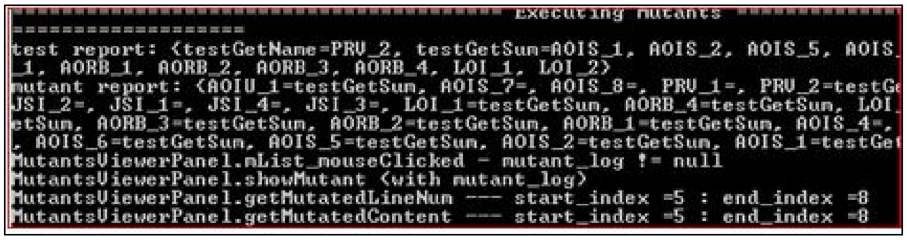
Class Mutants Result

Live Mutants #	3
Killed Mutants #	2
Total Mutants #	5
Mutant Score	40.0%

Live	Killed
JSI_1	PRV_1
JSI_2	PRV_2
JSI_3	

Figure 8: Run Test Set

While run test set which are manual design in order to run mutant and analyse the behaviour of program. If there is change in behaviour of program, some mutants are killed. Mutant killing not show behaviour change but also in analysing the performance of program while change in input of program.



```
================================== Executing mutants ================
===================
test report: (testGetName=PRV_2, testGetSum=AOIS_1, AOIS_2, AOIS_5, AOIS
_1, AORB_1, AORB_2, AORB_3, AORB_4, LOI_1, LOI_2)
mutant report: (AOIU_1=testGetSum, AOIS_7=, AOIS_8=, PRV_1=, PRV_2=testGe
JSI_2=, JSI_1=, JSI_4=, JSI_3=, LOI_1=testGetSum, AORB_4=testGetSum, LOI
etSum, AORB_3=testGetSum, AORB_2=testGetSum, AORB_1=testGetSum, AOIS_4=,
, AOIS_6=testGetSum, AOIS_5=testGetSum, AOIS_2=testGetSum, AOIS_1=testGet
MutantsViewerPanel.mList_mouseClicked - mutant_log != null
MutantsViewerPanel.showMutant (with mutant_log)
MutantsViewerPanel.getMutatedLineNum --- start_index =5 : end_index =8
MutantsViewerPanel.getMutatedContent --- start_index =5 : end_index =8
```

Figure 9: Executing Mutants

Above mention fig8 identify the mutants report how all mutant executing and checking each operator by using manual design test set. One main advantage is that by design test set, we can analysis the executing mutant and check mutants score .equivalent mutant can be calculated by subtract killed mutant from total mutant. Live mutants score also can be calculated.
Mutants testing can play an active role while executing program .net bean and eclipse which are current follow in software industry. By using this testing can be easily performed and rapidly results are generated by design various design test.

3. Conclusion

This paper proposes and outlines approach of performing software testing using equivalent mutants has show the performance in such way that generation of mutant The novel aspect of the algorithm is the exploitation of both textual and dependence information. Approach of performing software testing using equivalent mutants has show the performance in such way that generation of mutant types and performance. For example, Design the test set of program carefully that the value should kill the mutants and retrieve the valid result. By implementing this proposed approach software testing easily and quickly in order to remove bugs. MUTANTS can be used to build the short length programming code.

4. Future Work

The tool developed works only for the Java language and is easy to use. In the presented a proposed approach is used to perform testing lead to the generation mutants which do work directly on source code and class file. Since the byte code has been taken as input to generated mutants in different viewer, so up to some extent it is able to generated mutants. Moreover byte code is platform independent which makes this tool more efficient than already existing tools. As an application of abstract parse tree based approach and program dependence graph approach on all code is tedious work so the proposed tool have reduced the work by identifying mutants(equivalent or non-equivalent). In future this approach can be integrated with other approaches like abstract parse tree based approach and the program dependence graph approach to make this a approach to efficiently find some semantic code for byte code those are short in length. With the help of proposed approach, by integrating *junit* which help generation test set to make these techniques could be applied to make it more efficient and cost effective.

5. References

1. Kanika Raheja Rajkumar Tekchandan "An Emerging Approach towards: Metric Based Approach on Byte Code "published in ijarcsseVolume 3, Issue 5, May 2013 ISSN: 2277 128X
2. 24Monalisa Sarma and Rajib Mall. Automatic generation of test specifications for coverage of system state transitions. Information & Software Technology,51(2):418–432, 2009.

3. M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 2000..
4. Nitin Bhide, Code Analysis and Visualization Tool. [Online] Available:http://thinkingcraftsman.in/articles/codeanalysistools.html,2009.
5. J. H. Andrews, L. C. Briand & Y. Labiche (2005): Is Mutation an Appropriate Tool for Testing Experiments?In: Proceedings of the 27th International Conference on Software Engineering (ICSE’05), St Louis, Missouri,pp. 402–411, doi:10.1145/1062455.1062530.
6. [19] R. Mall. Fundamentals of Software Engineering. Prentice Hall, Springer-VerlagGmbH, 2nd edition, 2004
7. 18 P.C Jorgensen. Software Testing: a Craftsman’s approach. CRC PRESS, 2ndedition, 2002.
8. 75[] R. Pressman. Software Engineering: A Practitioner’s Approach. McGraw-Hill, New York, 2002.
9. [20] Robert V. Binder. Testing Object-oriented Systems: Models, Patterns, AndTools. Addison-Wesley Object Technology Series, 1999.
10. Jean-Francois Patenaude, Bruno Lagu¨e, “Extending Software Quality Assessment Techniques to Java Systems”, Seventh International Workshop on Digital Object Identifier, pp. 45- 56, 1999.
11. IEEE Standard 829-1998, “IEEE Standard for Software Test Documentation”,pp.1-52, IEEE Computer Society, 1998.
12. IEEE Standard 1059-1993, “IEEE Guide for Software Verification and ValidationPlans”, pp.1-87, Computer Society, 1993.
13. H. Leung and L. White. Insights into regression testing. In Proceedings of the Conference on Software Maintenance, pages 60–69, 1989.
14. R. D. Craig, S. P. Jaskiel, “Systematic Software Testing”, Artech House Publishers, Boston-London, 2002.
15. Quality assurance and testing [Online]. Availableat:http://www.hanusoftware.com/services/quality-assurance-testing [As accessed on September 2012]