Experimental Comparison of Automated Mutation Testing Tools for Java

Shweta Rani[1] Bharti Suri [2] Sunil Kumar Khatri[3]

[1] USICT, GGS Indraprastha University, Delhi
Email: shweta2610@gmail.com
[2] USICT, GGS Indraprastha University, Delhi
Email: bhartisuri@gmail.com
[3] Amity Institute of Information Technology, Amity University, Noida
Email: sunilkkhatri@gmail.com , skkhatri@amity.edu

Abstract -- Mutation testing has been used to evaluate the quality of the test set and provides the confidence in testing activity. Generation as well as running of mutants needs practice. A huge amount of mutants are generated and therefore, it is a very costly activity in terms of time and effort. Automatic mutant generation and execution is essential to support testing. Automation achieves more attention, saves time and effort as developers and testers use automated tools. In the recent years, researchers have found that practitioners have diminutive knowledge about tools and their effectiveness. This paper compares five well-known publicly accessible mutation testing tools: 'MuClipse', 'Judy', 'Jumble', 'Jester' and PIT. This comparison uses a collection of Java classes taken from various easily accessible sources. Tests were designed and generated with the help of test generation techniques. The performance of each mutation tool was analyzed and was compared based on their mutation operators, mutation score and execution time.

Keywords – Mutation Testing, Mutation Score, Automated Tools, Test Suite

I. INTRODUCTION

The principle behind 'Mutation Testing' is to locate and represent the mistakes incorporated by the programmer while implementing the code. These mistakes, called mutations, include syntactic error, variable removal, constant value alteration etc. and the faulty program is called the mutant. A single alteration in the original program, called the first order mutant is executed against the test suites. It is detected and is referred to as a killed mutant when the results of executing tests on them differ from that of the original program. However, some mutants are not caught as their behavior is similar to that of the original program, known as equivalent mutants. The quality of the test-suites depend directly upon the amount of killed mutants and achieving better quality test suites is a major aim of software testing.

978-1-4673-7231-2/15/$31.00 ©2015 IEEE

Mutation testing [1] [2] is a powerful, fault based testing technique which is used to measure the quality of test suites. Mutation testing provides the mutation score or mutation adequacy score as given in equation (1).

$$MAS\ (P, TS) = \frac{\mathrm{Kn}}{M - En} \quad (1)$$

Where, P = Program under test, TS = Test suite, Kn = number of Killed mutants, M = total number of Mutants, En = number of equivalent mutants.

Mutants are created by applying the mutation operators. There are two types of mutation operators: class level mutation operators [3] and method level mutation operators [4]. These operators yield the mutants in bulk. Cost of mutation testing is very high because generation and compilation of such large number of mutants take a lot of time and effort. Offutt et al [5], in his 6 selective mutation strategy, divided the Mothra mutation operators into three categories: statement, operands and expressions. Offutt found that there are five mutation operators that achieved 99.5 mutation score. These traditional mutation operators were ABS, UOI, LCR, AOR and ROR. These selective operators reduced the cost of mutation analysis by reducing the number of mutants and also covered approximately all faults produced by all traditional operators. Jia and Harman introduced higher order mutants [6] [7] for mutant reduction. According to Jia and Harman, a single higher order mutant (HOM) can be used to replace some first order mutants (FOMs), a strong HOM is only killed by a subset of the intersection of test suites that kill each FOM. Ma et al [8] [9] proposed byte code translation to reduce compilation cost in this, mutants are generated using compiled objects instead of the original program. In their proposal, mutant's execution did not require compilation of source code.

Mutation testing is generally not performed to its extreme as it needs a lot of time and effort by the developers or testers. Moreover, developers or testers are not aware of the ways to

design and implement the mutants. Instead of manual generation and execution, automated mutation testing tools can help with the above problems. Tools reduce time and effort required for mutation analysis. There are a number of language dependent tools that are free and easy to use. MILU [10], CSAW [11] are freely available tools for C language. Some tools for Java are MuJava [8] [12], MuClipse [13], Judy [14], Jumble [15], Jester [16], Bacterio [17], PIT [18], Javalanche [19]. Mutation testing has been performed on databases too. SQLMutation [20] [21] was developed for mutating the SQL queries. Selecting an appropriate mutation testing tool is another major issue.

In this study, a few automated tools were selected for evaluation based on the factors: i) tools that were freely available for use, ii) tools, that automatically generate mutants, and iii) tools supporting Java classes.

For this experimental work, five freely accessible and automated mutation testing tools were chosen. The first one was MuClipse [13] which generates traditional and class level mutants separately. Judy [14] was the next appropriate tool, which has been developed in Java with AspectJ extensions. The third tool, Jumble [15] reduces the number of mutation operators for mutant generation. The fourth automated tool was Jester [16] which performs slow mutation analysis and mutates the test code as well. PIT [18] is a byte code mutation testing tool which results the mutation and line coverage. These tools were used to generate and run mutants for a collection of Java classes (discussed in detail in section III).

JUnit [22], a unit testing framework was used for writing and executing the test suites. Test data were created using equivalence and boundary value testing techniques. Finally, test suites were applied to the mutants produced by these selected tools and the results were analyzed. Automated tools are described in detail in section II. Experimental process including Java classes used is then demonstrated in section III. Results and analysis of experimental comparison have been discussed in section IV and concluded in section V.

II MUTATION TESTING TOOLS

Mutation testing is a process of finding the faults seeded in the program intentionally or by mistake. Automated tools were developed to minimize the cost of mutation analysis. These tools rely on the language and works on a set of mutation operators. Mutation operators play a very significant role in mutation analysis. In this study, we worked on the Java based tools that are open source and have simpler operability. Table I summarizes the comparison of the tools based on some chosen factors and are described in detail below**.**

MuClipse [13] is an eclipse based plugin [23] for mutant generation and for running JUnit test suites. It is based on MuJava introduced by Offutt et al [9]. MuClipse supports 15 traditional operators and 28 class level mutation operators. It provides GUI, mutant viewer and test case executor. GUI support helps in selecting the mutation operator and generating the mutants. Mutant viewer is used to view the mutants and test case executor for execution of mutants using test set. MuClipse automatically generates mutants by selecting the class to mutate and run JUnit test suites on each mutant. It accounts the total number of live as well as killed mutants along with the mutation score for that test suite. For mutant creation, it first generates the parse tree of the subject program and then generates the mutants. Mutation operates on the source code, and thus, it is necessary to compile the mutants before running JUnit test suite. This process makes it quite slow. It also ignores the equivalent mutants.

Judy [14], developed by Madeyski and Radyk, is a command line mutation tool which supports the generation, compilation to bytecode and execution of mutants. It is implemented by FAMTA i.e., Fast Aspect- Oriented mutation testing algorithm. It implements both traditional and object oriented mutation operator and supports JUnit testing. It generates mutants from Java source code and avoids multiple compilations of every mutant during their generation. Mutant generation, compilation, testing process and mutation operators for Judy are given in [14].

TABLE I COMAPARISON BETWEEN SELECTED MUTATION TOOLS

Name	Version	Mutant Generation Level	Traditional & object oriented Mutation Operators available?	Interface	Mutant state	Operator selection facility available?	Output Generated
MuClipse	1.3	Java code and byte code	Yes	Eclipse Plug-in	Separate source and class files	Yes	Mutants, mutation score, live and killed mutants
Judy	2.1.0	Java code	Yes	Command line	Not visible but present in memory	Yes	No. of mutant, test methods, test classes, mutation score, time in milliseconds
Jumble	1.0.0	Byte code	Some traditional operators	Eclipse Plug-in	Not visible but present in memory	No	Mutation score, mutation points, period, M fail
Jester	Simple jester	Java code	No	Command-line	Separate source files	No	Mutation score, total mutants, survived mutants, duration
PIT	1.1.4	Byte code	Traditional operators	command line & Eclipse-plugin	Separate file in memory	Yes	No of mutants, Mutation score, killed mutants, line coverage, mutation coverage

Jumble [15] is a mutation tool that mutates Java class at bytecode level and also supports JUnit. It supports mutation operators: AOR, ASR, COR, LOR, ROR and SOR. It considerably reduces the number of mutation operators. It is fast as it works at bytecode level. While executing test data on mutants, it returns mutation score along with the total time taken as well as the mutants for which the test failed. Mutants obtained by Jumble are hidden from user and are not available for analysis.

Jester [16] is the JUnit test tester. It modifies the source code as well as the test code and verifies whether the change is detected by the test. The test is erroneous if it is not modified and also does not fail. It offers a strategy to extend the traditional set of mutation operators. It repeats the process of creating, executing, testing and reporting for each mutant during analysis; thus, very slow to work with. The cost of using 'Jester' is very high in terms of time that includes machine time and developer time to interpret the result. It also doesn't provide the interface to select the mutation operators applied for the programs under test.

PIT [18] is an open source mutation testing tool developed by Coles. It works fast for mutant generation. There are 4 phases: mutant generation, test selection, mutant insertion and mutant detection in PIT. PIT uses mutation operators like conditional boundary, negate conditionals, conditions removal, math mutator and so more. Mutation is performed at byte code level. Initially, mutants are generated and test data are selected to run over mutants. Then mutants are loaded into the JVM and detected by the test set. Along with mutants details, it also results line coverage and mutation coverage, thus requires some overhead.

III EXPERIMENTAL DESIGN

This section explains the conducted experiment in detail. Initially, Java classes used for evaluating the performance of each tool in the experiment are accounted. Next, test data generation, using boundary value and equivalence class testing, is presented. The process used in performing an experiment is then depicted in a diagram.

A. Test Data Generation

The quality and effectiveness of testing depend primarily on test data. We have used boundary value and equivalence class partitioning testing technique to generate the test data for each selected Java class. Boundary value testing says that the error possibly occurs at the boundary or nearby it [24] [25]. For an input range min to max, it checks the boundary values like min, min+, nominal, max-, max without considering the internal logic behind the program. Equivalence class supports two types of test data generation on account of input and output [24] [25]. It supports the logical structure of the problem and generates test case for each logical equivalence class. We have used JUnit version 4 [22] to write the test cases for Java classes. JUnit tests are supported by mutation tools selected in this work. To work with JUnit, download JUnit jar file, hamcrest-core.jar and put them in the class path of the Java project.

B. Java Classes Mutated and Tested

Used Java classes with LOC, methods and a number of tests have been listed in Table II. These subject programs have been widely used in mutation testing [26] [27] [28].

'checkIt', 'TriType', 'Quadratic', 'Cal-days' are taken from [29]. 'TriType' is a triangle classification problem. 'Quadratic' program calculates the roots of given quadratic equation. 'Cal-days' computes the number of days between 2 input dates for the same year. 'Student_division', 'Day_of_week' are the programs selected from [25] [30]. 'Student_division' is a small and simple program that takes 3 inputs as marks and calculates the division of a student. The problem 'Day_of_week' is used to find the day for the input date.

TABLE II SUBJECT PROGRAMS

Java Classes(alias name)	LOC	Methods	Description	Tests
checkIt (J1)	17	1	Check for the boolean value i.e. true or false	2
Student_division (J2)	29	1	Finds the division of a student using marks	21
Quadratic (J3)	34	1	Decides whether a quadratic equation has real roots or not.	11
TriType (J4)	40	1	Return type of triangle	23
Cal-days (J5)	54	1	Return the number of days between input dates	15
Day_of_week (J6)	111	1	Find the day of week for the date	22

C. Experimental Process

Fig. 1 illustrates the process used in the experimental study. The Java classes are represented by leftmost box, Java Program. Each of the five automated tools was used to generate mutants for each subject program. Mutants are generated automatically by selecting the mutation operators. Then, test data are generated using equivalence class and boundary value methods. After generating the test set, it is applied to mutants that were generated by mutation testing tools separately. Each tool results with mutation score and the total number of generated mutants. In order to compare the tools, mutation score was recorded.

D. System Setup

We performed this experiment on Intel core i3 processor of a 1.70 GHz clock with 64 bit windows 8.1 operating system. Automated tools that were used as a part of this experiment are version specific as mentioned in Table I. For each program, separate JUnit tests were created. We have used Java 6 as it is compatible with all selected tools.

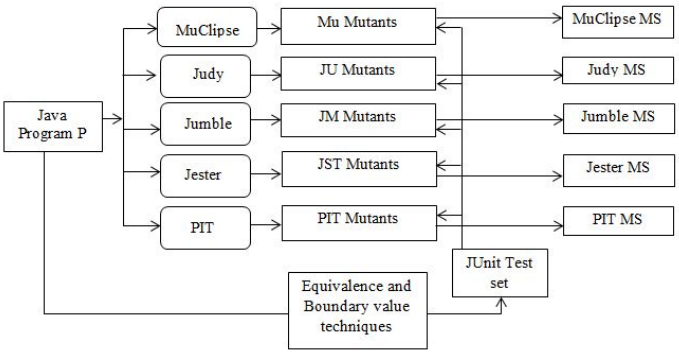

Fig. 1 Experimental Procedure

IV RESULTS AND ANALYSIS

The results of running each tool on the programs selected in this experiment have been shown in Table III. Table IV and Table V demonstrate the statistics of using the automated tools to the subject programs. Line graphs have been used to display the total number of mutants generated and the average mutation score obtained by each mutation tool when executing the test data on mutants respectively (Fig. 2 & Fig. 3).

'MuClipse' supports both traditional and class level mutants, however, we have created traditional mutants only. Traditional operators focus on individual statements in Java program. Results show that program 'Day_of_week' has the highest number of traditional mutants 606 over 111 lines of code. While the smallest program 'checkIt' with LOC 17 has 9 traditional mutants only. Generation of Mutants depends on the arithmetic, logical or shift operators used in the program. 'MuClipse' also provides the mutants to analyze. Mutation operators applied to the programs depends on tester's choice. In this case, we applied all the 15 traditional mutation operators.

As from the Table III, we analyzed that mutants generated by 'Judy' are less than 'MuClipse' however, are greater than other selected tools. Judy applied selective traditional operators like ABS, AOR, ROR, UOI and LCR with some other operators, i.e. UOD, SOR, LOR, COR, ASR, EOA, EOC, JTD, JTI, EAM and EMM while MuClipse included all the traditional operators and therefore, yielded more mutants than Judy. Mutation score of Judy was found to be a little better than that of MuClipse. Judy also runs faster. In this experiment, PIT obtained highest mutation score, yet Judy is faster than PIT as Judy took approximately 1 second to respond while for the same subject program (J6), PIT consumed 3 seconds.

In this work, 'Jumble' returned a moderate number of mutants as it applies very few selected mutation operators. These operators were AOR, ASR, COR, LOR, ROR and SOR. Jumble is easy to use as an eclipse plugin is also available. Jumble works within seconds and provides the list of live mutants.

Jester generated a small number of mutants as it applies mutation to both programs and test suites, however, in our research; mutants generated for Java class have been considered instead of the test suites. As per analysis, average mutation score using Jester is less than other tools, though for some programs it performed better than other tools in terms of mutation score. Jester is very slow as it modifies the code, recompiles it and then runs the test on the compiled object. It applies a simple mutation to the program like increases the integer value by 1, modifies if condition etc. Jester doesn't support traditional mutation operators. As can be seen by the results in Table III, Jester took minutes to run the test while PIT and other tools consumed some seconds or milliseconds. Hence, it was found to be slower because it recompiles the mutant again and again during analysis.

TABLE III TOTAL MUTANTS WTH MUTATION SCORE

Classes		J1	J2	J3	J4	J5	J6	Total
MuClipse	#Mutants	9	158	130	338	289	606	1530
	Mutation Score (%)	55	72	86	74	76	74	71.42
	Execution Time (in sec)	2.40	7.60	6.10	12.55	36.64	22.64	87.93
Judy	#Mutants	15	83	64	139	129	281	711
	Mutation Score (%)	20	86.75	92.19	86.33	69.77	69.75	72
	Execution Time (in sec)	0.040	0.42	0.19	0.61	0.36	1.21	2.83
Jumble	#Mutants	7	36	27	54	82	137	343
	Mutation Score (%)	57	80	88	92	70	64	70.22
	Execution Time (in sec)	2.38	2.19	2.12	2.10	2.50	3.30	14.59
Jester	#Mutants	4	25	31	48	63	162	333
	Mutation Score (%)	100	96	88	94	67	66	67.79
	Execution Time (in sec)	11.0	47.0	57.0	85.0	107.0	276.0	583.0
PIT	#Mutants	4	30	17	39	42	69	201
	Mutation Score (%)	50	83	94	92	86	80	98.52
	Execution Time (in sec)	1.0	2.0	2.0	3.0	2.0	3.0	13

TABLE IV
DESCRIPTIVE STATISTICS FOR EACH TOOL
IN TERMS OF NUMBER OF MUTANTS

Statistics	MuClipse	Judy	Jumble	Jester	PIT
Min	9	15	7	4	4
1st Qu	137	68.75	29.25	26.5	20.25
Mean	255	118.5	57.16	55.5	33.5
Median	223.5	106	45	39.5	34.5
3rd Qu	325.75	136.5	75	59.25	41.25
Max	606	281	137	162	69

TABLE V
MUTANTS COST FOR EACH AUTOMATED
MUTATION TOOL

Statistics	MuClipse	Judy	Jumble	Jester	PIT
Min	55	20	57	66	50
1st Qu	72.5	69.75	65.5	72.25	80.75
Mean	71.42	72	70.22	67.79	98.52
Median	74	78.05	75	91	84.5
3rd Qu	75.5	86.64	86	95.5	90.5
Max	86	92.19	92	100	94

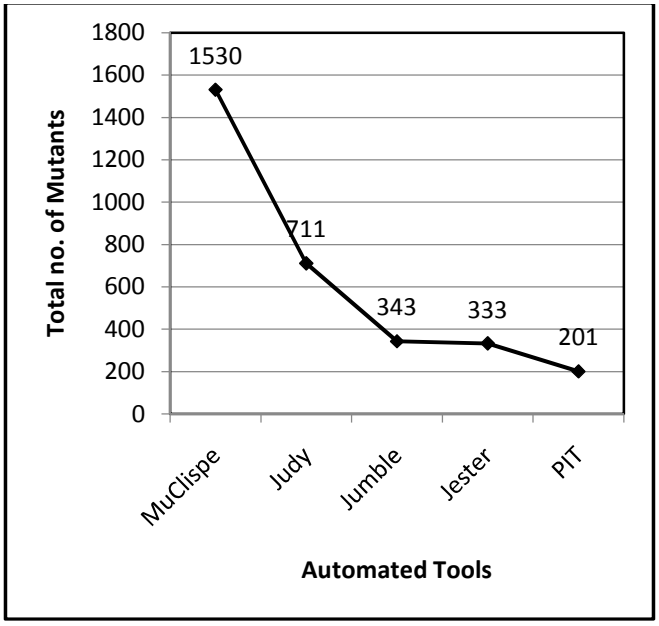

Figure 2 Line graph for Total Mutants Generated by Each Tool

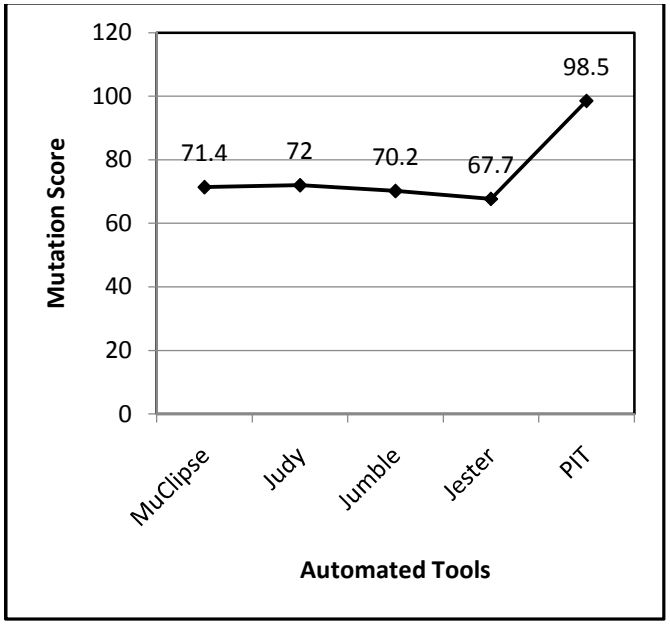

Figure 3 Line graph for Average Percentage of Mutants Killed by Each Tool

Current research used the eclipse plugin of PIT. PIT is very easy to use and also provides the coverage measures like line coverage and mutation coverage. It created a very little amount of mutants, thus, the cost of mutation testing becomes low in terms of storage and test data requirements. PIT supports method level mutation operators and doesn't support object oriented mutation operators. Mutation score of PIT is very high on the data set used in this work in comparison to other tools. Time of the generation and execution of mutants was found to be few seconds, yet was slower than Judy.

Now on the basis of number of mutants, mutation score and execution time, a comparison among these tools was performed. Table IV provides the minimum number of mutants as well as maximum number of mutants for each experimental tool. As seen by the statistics in Table IV and Table V, MuClipse produced the largest number of mutants, therefore, it requires slightly greater storage for keeping all the mutants in the memory. With this downside, MuClipse also displayed all the mutants to the tester for further analysis and also is very easy to use with eclipse IDE. PIT generated a pint sized set of mutants, however slower than Judy. Jester mutated both programs as well as test data, were found to be little slower.

Fig 2 & 3 demonstrate the cost in terms of mutants and mutation score for the tools using boxplot. It is clear that PIT killed maximum number of mutants compared to other tools. PIT also produces a small set of mutants. Judy is very fast and provides the result in milliseconds. All in all, no tool was found to be faster in general. Depending on the criterion for comparison, sometimes a tool was found to be good and sometimes the others.

V. CONCLUSION

Mutation testing is a very powerful technique used by the researchers. It is a useful approach for detecting the pieces of code that are executed by running tests, though not completely tested. It is not widely applied in software engineering because of its confined performance of existing tools [31] [32] [33]. Mutation testing also has a lot of limitations in terms of high mutation cost, requirement of huge test data and great execution cost in terms of time and effort. In this, equivalent mutations also need manual checking, real faults are much different and more complex than seeded faults [34]. Mutation testing also requires a large amount of memory space and incurs the heavy computational cost. Tools for mutation testing reduce the cost of mutation analysis.

This paper compared five publicly accessible mutation testing tools on the basis of their fault finding abilities. Faults were seeded into Java classes using these mutation testing tools and caught using equivalence class and boundary value tests. It was noticed that, when tools were applied to Java classes, PIT generated less mutants than others and also it killed more mutants. All the tools have some special features and operate on a different set of mutation operators. Thus, depending on the requirements, duration and cost of application, an appropriate tool can be chosen using our results.

It can be concluded that all the studied tools are language dependent and each tool works on a different set of mutation operators. In future, the authors wish to propose and develop a new mutation testing tool that will work for more than one language with fast generation and execution of mutants.

REFERENCES

[1] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," *computer*, vol. 11, pp. 34-41, 1978.

[2] R. G. Hamlet, "Testing Programs with the Aid of a Compiler," *IEEE Transaction on Software Engineering*, vol. 3, no. 4, pp. 279-290, 1977.

[3] Y.S. Ma, Y.R. Kwon, and A. J. Offutt, "Inter-class Mutation Operators for Java," in *Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE'02).*, 2002, p. 352.

[4] Y. S. Ma and J. Offutt. (2005) Description of Method-level Mutation Operators for Java. [Online]. https://cs.gmu.edu/~offutt/mujava/mutopsMethod.pdf

[5] A.J. Offutt, A. Lee, G. Rothermel, R.H. Untch, and C. Zapf, "An Experimental Determination of Sufficient Mutant Operators," *ACM Transactions on Software Engineering and Methodology*, vol. 5, no. 2, pp. 99-118, 1996.

[6] Y. Jia and M. Harman, "Constructing Subtle Faults Using Higher Order Mutation Testing," in *Proceedings of the 8th International Working Conference on Source Code Analysis and Manipulation (SCAM'08)*, 2008, pp. 249-258.

[7] Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, vol. 37, no. 5, pp. 649-678, 2010.

[8] Y.S. Ma, A. J. Offutt, and Y.R. Kwon, "MuJava: An Automated Class Mutation System," *Software Testing, Verification & Reliability*, vol. 15, no. 2, pp. 97-133, 2005.

[9] A. J. Offutt, Y.S. Ma, and Y.R. Kwon, "An Experimental Mutation System for Java," *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 4, pp. 1-4, 2004.

[10] Y. Jia and M. Harman, "MILU: A Customizable, Runtime-Optimized Higher Order Mutation Testing Tool for the Full C Language," in *Proceedings of the 3rd Testing: Academic and Industry Conference Practice and Research Techniques (TAIC PART'08)*, 2008, pp. 94-98.

[11] M. Ellims, D Ince, and M. Petre, "The Csaw C Mutation Tool: Initial Results," in *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, 2007, pp. 185-192.

[12] J. Offut and N Li. (2011) µJava Homepage. [Online]. http://cs.gmu.edu/~offutt/mujava/

[13] B. H. Smith and L. Williams, "An Empirical Evaluation of the MuJava Mutation Operators," in *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION,'07*, 2007, pp. 193-202.

[14] L. Madeyski and N. Radyk, "Judy a mutation testing tool for Java," *IET Software*, vol. 4, no. 1, pp. 32-42, 2010.

[15] S. A. Irvine et al., "Jumble Java Byte Code to Measure the effectiveness of Unit Tests," in *Proceeding. Testing: Academic and Industrial Conference Practice and Research Techniques*, Windsor, 2007, pp. 169-175.

[16] I. Moore, "Jester-a Junit test tester," in *2nd International Conference on Extreme Programming and Flexible Processsed in Software Engineering*, Italy, 2001.

[17] P. R. Mateo and M. P. Usaola, "Bacterio: Java Mutation Testing Tool:A framework to evaluate quality of tests cases," in *28th IEEE International Conference on Software Maintenance (ICSM)*, 2012, pp. 646-649.

[18] Henry Coles. (2012) PIT Mutation Testing. [Online]. http://pitest.org/

[19] B. J. M. Grun, D. Schuler, and A. Zeller, "The Impact of Equivalent Mutants," in *IEEE International Conference on Software Testing Verification and Validation Workshops*, 2009, pp. 192-199.

[20] J. Tuya, M. J. Suarez-Cabal, and C. de la Riva, "SQLMutation: A Tool to generate mutants of SQL database queries," in *2nd workshop on Mutation Analysis ((MUTATION'06)*, 2006.

[21] Test4Data - Testing for Database Applications. [Online]. http://in2test.lsi.uniovi.es/sqlmutation/

[22] JUnit. [Online]. http://junit.org/

[23] Eclipse. [Online]. https://www.eclipse.org/

[24] P. C. Jorgensen, *Software Testing, a craftsman's approach*, 2nd ed., grand valley state university. Department of Computer Science and Information Systems, Ed., 2002.

[25] Y. Singh, *Software Testing*. Cambridge, UK: Cambridge University press, 2012.

[26] M. E. Delamaro, J. Offutt, and P. Ammann, "Designing Deletion Mutation Operators," in *IEEE International Conference on Software Testing, Verification and Validation*, 2014.

[27] M. Papadakis and N. Malevris, "Searching and generating test inputs for mutation testing," *SpringerPlus, Knowledge based software engineering*, 2013.

[28] M. Patrick, R. Alexander, M. Oriol, and J. A. Clark, "Using Mutation Analysis to Evolve Subdomains for Random Testing," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, 2013, pp. 53-62.

[29] P. Ammann and J. Offutt, *Introduction to software Testing*. Cambridge, UK: Cambridge University Press.

[30] K.K. Agarwal and Y. Singh, *Software Enginnering*.: New Age International Publishers.

[31] A.J. Offutt and R.H. Untch, "Mutation 2000: uniting the orthogonal," in *Mutation testing for the new century*, 1st ed.: Kluwer Academic Publishers, 2001, pp. 34-44.

[32] A.J. Offutt, Y.S. Ma, and Y.R. Kwon, "The Class-level Mutants of MuJava," in *Proceedings of the 2006 international workshop on Automation of software test(AST'06)*, 2006, pp. 78-84.

[33] Y.S. Ma, M.J. Harrold, and Y.R. Kwon, "Evaluation of Mutation Testing for Object-Oriented Programs," in *Proceedings of the 28th international conference on Software engineering(ICSE'06)*, 2006, pp. 869-872.

[34] B. Schwarz, D. Schuler, and A. Zeller, "Breeding High-Impact Mutations," in *Fourth International Conference on Software Testing, Verification and Validation Workshops*, 2011, pp. 382-387.