International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:10, No:1, 2016

The Effect of Program Type on Mutation Testing: Comparative Study

B. Falah, N. E. Abakouy

***Abstract*—**Due to its high computational cost, mutation testing has been neglected by researchers. Recently, many cost and mutants' reduction techniques have been developed, improved, and experimented, but few of them has relied the possibility of reducing the cost of mutation testing on the program type of the application under test. This paper is a comparative study between four operators' selection techniques (mutants sampling, class level operators, method level operators, and all operators' selection) based on the program code type of each application under test. It aims at finding an alternative approach to reveal the effect of code type on mutation testing score. The result of our experiment shows that the program code type can affect the mutation score and that the programs using polymorphism are best suited to be tested with mutation testing.

***Keywords*—**Equivalent mutant, killed mutant, mutation score, mutation testing, program code type.

I. Introduction

SOFTWARE testing is the process of determining if a program behaves as expected. It is an intellectually challenging activity aimed at evaluating the ability of a program or system and determining whether or not it meets its requirements [1].

With increased expectations for software component quality, software developers are expected to perform effective and efficient testing. In today's scenario, mutation testing has been used as a fault injection technique to measure test adequacy. It is done by seeding artificial faults into the implementation (original source code) and checking whether the test suite is able to find the errors.

Mutation testing is a powerful fault-based testing technique used to ensure software quality and evaluate the effectiveness of test cases [1]-[3]. It is based on the assumption that a program will be well tested if all simple faults are detected and removed.

Early studies suggest that testing can comprise up to 50% of the software development budget [4], and a recent survey revealed that billions of dollars are routinely wasted on large software projects due to inadequate testing [5].

Mutation testing is a structural testing method that is meant to approximate faults in the program being tested, while improving the test cases and test suites. The main goal of mutation testing is to evaluate and develop the test cases and test suites by injecting faults in the original code. There are mainly three types of mutation testing:

1) Value mutations: involves changing the values of constants or parameters;
2) Decision mutations: involves modifying conditions to detect potential slips and errors in the coding of conditions in programs;
3) Statement mutations: might involve deleting certain lines to detect omissions in coding or swapping the order of lines of code.

Although powerful, mutation is computationally a very expensive testing technique. In fact, its three main stages (mutant generation, mutant execution and result analysis) require many resources to be successfully accomplished [6]. The high cost of mutation testing is due to high computational cost of executing the huge number of mutants against a test set, more there is the human oracle problem which refers to the process of checking the original program's output with each test case [7].

While taking into consideration the program code type, this paper is a comparative study between four operators' selection techniques: (1) mutant sampling, which is a simple approach that randomly chooses a small subset of mutants from the entire set; (2) class level operators; (3) method level operators; and (4) all operators' selection.

II. Related Work

To reduce the cost, many mutation strategies have been developed and adopted. These strategies have been done mainly in two directions which are addressing the common problems of mutation testing in order to reduce cost and time spent to execute mutants, or in doing comparative studies between the different techniques of mutation testing. However, the relationship between the program type and the cost of mutation testing has been ignored.

P. R. Mateo and M. P. Usaola [8] proposed a technique that improves existing mutation cost reduction technique. This technique gives the ability to identify situations where a mutant must not be executed, therefore reducing the number of total required executions in a mutation analysis. The experiment shows that the execution order of the tests has an influence on the number of executions, and therefore the execution order influences the efficiency of MUSIC.

B. H. Smith and L. Williams [9] tried to answer the following question: should software testers use mutation analysis to augment a test set? They showed that the choice of mutation tool and operator set can play an important role in determining how efficient mutation analysis is, for producing new test cases.

Bouchaib Falah, Nour El Houda Abakouy are with the Al Akhawayn University, Ifrane, Morocco (phone: 212-535862194; fax: 212-53862030; e-mail: B.falah@aui.ma, n.abakouy@aui.ma).

References [10], [11] suggested that 10% was the optimal percentage of mutants to be selected from the set of mutants generated; executing 10% of mutants is only 16% less effective than testing the entire set of mutants in mutation score.

Other studies focused on different approaches to determine the optimal subset of mutants. Sahinoglu and Spafford [12] suggested a sampling approach that proposed to calculate the ratio of the mutant based on the Bayesian sequential probability ratio test. This new approach suggested that the subset of the mutants to be tested is randomly selected until a statistically appropriate sample size is reached.

III. Mutation Testing Building Blocs

Using mutation is very important in evaluating, comparing, and improving the quality of a test suite. However, the value of mutation testing depends on the set of mutants used in the evaluation. These mutants are formed from the original program through the use of predefined mutation operators.

Since mutation is always based on mutation operators, researchers have designed and developed mutation operators to support various programming languages including Java. The quality of mutation operators is the key to mutation testing. One mutation testing used with Java code is MuJava [13], [14]. MuJava includes two general types of mutation operators: class level operators [15] and method level operators [13], [16].

The efficiency of mutation testing relies heavily on mutation operators used. A mutation operator consists of a set of "predefined program transformation rules" used to substitute a section of the program in order to introduce faults in the source code [17]. So their main role is to produce different version of the original program. These versions are called mutants. A mutant is created by applying mutation operator on the original source code. Fig. 1 illustrates the creation process of mutants.

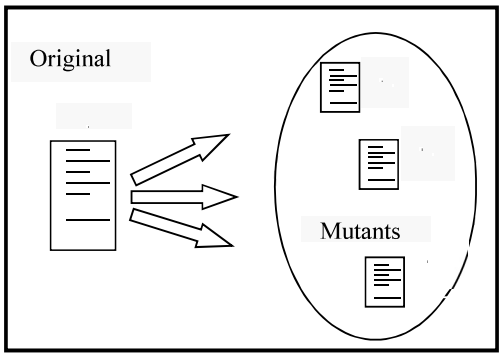

Fig. 1 Mutants Creation Process

A. Method Level Operators

The method level operators, also called traditional operators, handle the primitive features of programming languages. They modify a subsection of an original program by replacing, deleting or inserting primitive operators (arithmetic operator, relational operator, conditional operator, shift operator, logical operator, and assignment) [18].

Those operators were influenced by the procedural language that dominated software engineering at its early stages.

MuJava provides 6 types of method operators [1]:

1. Arithmetic operator,
2. Relational operator,
3. Conditional operator,
4. Shift operator,
5. Logical operator,
6. Assignment.

According to the number and type of operands, some of the method level operators are divided into two or three operators. Table I describes the method operators defined by [18] for Java language.

TABLE I
Method-Level Mutation Operators [18]

Operator	Operator	Description
Arithmetic	AOR	**A**rithmetic **O**perator **R**eplacement
	AOI	**A**rithmetic **O**perator **I**nsertion
	AOD	**A**rithmetic **O**perator **D**eletion
Relational	ROR	**R**elational **O**perator **R**eplacement
Conditional	COR	**C**onditional **O**perator **R**eplacement
	COI	**C**onditional **O**perator **I**nsertion
	COD	**C**onditional **O**perator **D**eletion
Shift	SOR	**S**hift **O**perator **R**eplacement
Logical	LOR	**L**ogical **O**perator **R**eplacement
	LOI	**L**ogical **O**perator **I**nsertion
	LOD	**L**ogical **O**perator **D**eletion
Assignment	ASR	**A**ssignment **O**perator **R**eplacement

B. Class Level Operators

Object Oriented Programming brought many functionalities and properties like polymorphism, encapsulation and inheritance. As a result of those new properties, new bugs and faults were revealed in the programs that traditional operators were unable to detect. Thus, class-level operators were introduced in the late 90's to be used with those object-oriented programs.

The class mutation operators are classified into four groups:

- Encapsulation,
- Inheritance,
- Polymorphism,
- Java-Specific Features.

These four groups are based on the language features that are affected [19]. The first three categories depend on programming language features that are common in all OO programming languages. The fourth category depends solely on Java.

Table II lists the current set of class mutation operators of MuJava as well as their descriptions.

IV. Experiment

To conduct our experiment, we have used four java open source code applications of different lengths and complexity, which are downloaded from the internet. These applications are automatically injected by faults using an Eclipse Plug-in, MuClipse [20]. MuClipse is a popular automatic mutation testing tool that provides a bridge between the existing MuJava mutation engine and the Eclipse IDE.

TABLE II
CLASS-LEVEL MUTATION OPERATORS [18]

Language Feature	Operator	Description
Encapsulation	AMC	**A**ccess **m**odifier **c**hange
Inheritance	IHI	**H**iding variable **i**nsertion
	IHD	**H**iding variable **d**eletion
	IOD	**O**verriding method **d**eletion
	IOP	**O**verriding method calling **p**osition
	IOR	**O**verriding method **r**ename
	ISI	**S**uper keyword **i**nsertion
	ISD	**S**uper keyword **d**eletion
	IPC	Explicit call of a **p**arent's **c**onstructor
Polymorphism	PNC	**n**ew method **c**all with **c**hild class type
	PMD	**M**ember variable **d**eclaration with parent class type
	PPD	**P**arameter variable **d**eclaration with child class type
	PCI	Type **c**ast **o**perator **i**nsertion
	PCD	Type **c**ast **o**perator **d**eletion
	PCC	**C**ast type **c**hange
	PRV	**R**eference assignment with other comparable **v**ariable
Java-Specific Features	OMR	**O**verloading **m**ethod contents **r**eplace
	OMD	**O**verloading **m**ethod **d**eletion
	OAC	**A**rguments of **o**verloading method call **c**hange
	JTI	**T**his keyword **i**nsertion
	JTD	This keyword **d**eletion
	JSI	Static modifier **i**nsertion
	JSD	Static modifier **d**eletion
	JID	Member variable **i**nitialization **d**eletion
	JDC	**J**ava-supported **d**efault constructor **c**reation
	EOA	Reference assignment and content **a**ssignment replacement
	EOC	Reference comparison and content **c**omparison replacement
	EAM	**A**ccessor **m**ethod change
	EMM	**M**odifier **m**ethod change

The combined class level and method level mutation operators, including their subdivisions, that are currently used in MuJava are illustrated in Fig. 2.

These four programs are, then, analyzed by computing the score of each program for each of the four selection testing techniques, All operators, Class level, Method level, and Mutation sampling.

A. Programs under Test

1. Cruise Control Program

- An open source program with pre-built mutants.
- Composed of eight classes.
- The cruise control system simulates a car engine and its cruising controller
- Type of code: a logical code that uses polymorphism.

2. Elevator Program

- An open source program with pre-built mutants.
- Composed of eight classes.
- The Elevator system consists of a number of elevators servicing a number of floors.
- Type of code: a logical code that uses inheritance.

3. Black Jack

- Composed of eight classes program.
- Logical program that uses some computational assignment.
- Type of code: a logical code that uses polymorphism.

4. Calculator

- A basic program that contains standard operations.
- No inheritance, polymorphism or special features are used.
- This program was an in-class practice for JUnit.

B. Mutation Testing Tool Used

The creation and the execution of mutants are long and resource consuming tasks. Several researchers have worked on developing mutation software that automates the process. As part of our research, we have used MuClipse, which adopts architecture similar to the architecture adopted by MuJava that uses "Mutant Schemata Generation" (MSG) approach [20].

MuClipse provides functionality to easily create mutants as well as to run unit tests against those mutants, in order to generate a mutation score that will indicates the quality of those test cases used. A mutation score is the percentage of all the generated mutants that have been killed divided by the total number of all the mutants created in the first runtime configuration. If the mutation score is low then test cases should be improved, in order to kill more mutants.

C. Metric Used

The metric used in this research project is the standard scoring metric for mutation testing, which is defined as: Mutation Score = 100 * D / (N - E), where

- D = Dead mutants that were killed by testing them against the test cases.
- N = Number of total generated mutants depending on the operator selection method used.
- E = Number of equivalent mutants. Equivalent mutants are kind of program mutations that leave the program's overall semantics unchanged, and therefore cannot be caught by any test suite [21].

This score allows us to have an idea about the quality of test cases tested against the generated mutants. The higher the score will be, the better the test cases are.

D. Our Approach

The aim of our approach is to do a comparative study between the four operators' selection techniques, mutant sampling, class level operators, method level operators and all operators' selection, while taking into consideration the program code type.

On the mutant sampling technique, 15 operators were randomly selected from both class-level and method-level operators. These operators are: AOR, AOI, AOD, ROR, COR, COI, IHI, IHD, IOD, IOP, IOR, ISI, ISD, IPC, and PNC.

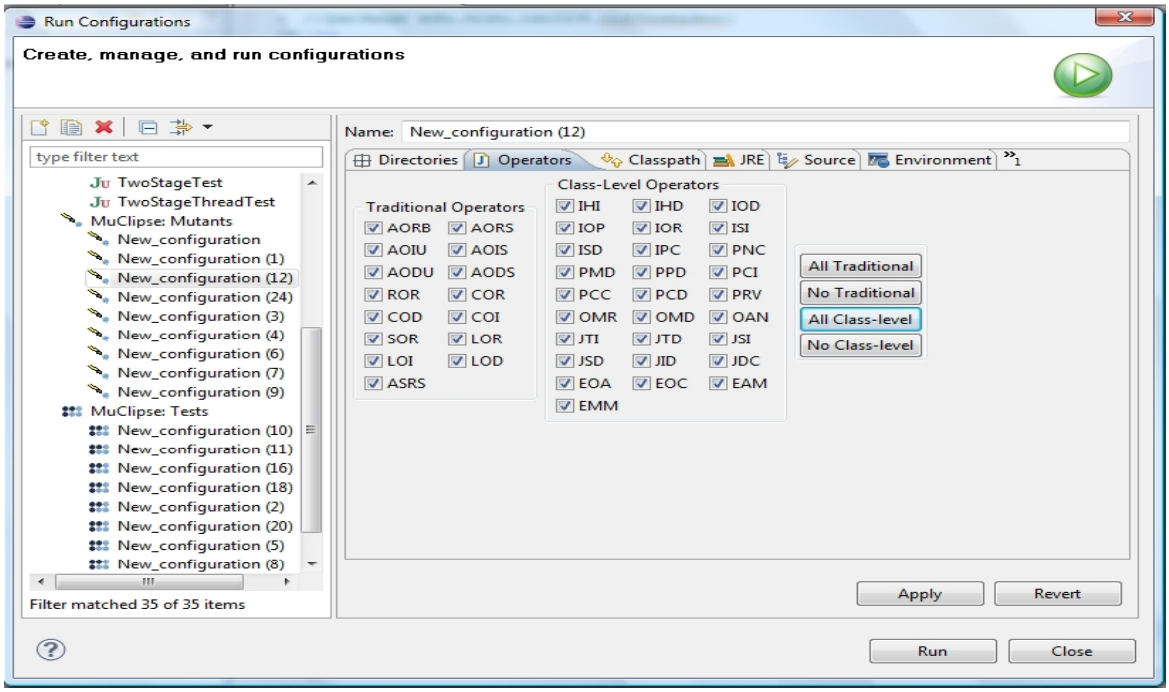

Fig. 2 MuJava Mutation Operators

E. Results and Analysis

Tables III-VI provide results generated from running mutation testing for every program under test. On each test, different operators' selection techniques were chosen. For that reason, on each operator selection technique, we generate a new result file of mutants separately. Moreover, for every operator selection method, the test was run for every class from the program and score values were calculated.

1. Cruise Control

TABLE III
CRUISE CONTROL PROGRAM RESULTS

Selection Method	Total mutants	Alive mutants	Dead mutants	Score %
All Operators	206	0	206	100%
Class Level	40	0	40	100%
Method Level	40	0	40	100%
Mutant Sampling	251	206	45	18%

Cruise Control program testing score was a full score for the first operators' selection techniques, while it was very low for the mutant sampling which is quite logic since random selection may result some random operators that may not be good representatives.

2. Elevator

The elevator program testing score ranges from 7% to 61%; meaning it varies from low to very low for all operators' selection technique except for class level method which is quite decent.

TABLE IV
ELEVATOR PROGRAM RESULTS

Selection Method	Total mutants	Alive mutants	Dead mutants	Score %
All Operators	404	261	149	36,9%
Class Level	95	37	58	61%
Method Level	342	251	91	27%
Mutant Sampling	459	427	32	6.9%

3. Black Jack

TABLE V
BLACK JACK PROGRAM RESULTS

Selection Method	Total mutants	Alive mutants	Dead mutants	Score %
All Operators	202	23	179	88.6%
Class Level	25	0	25	100%
Method Level	177	27	150	84.7%
Mutant Sampling	67	17	50	74.7%

The Black Jack program testing scores were pretty high, even if they are not totally full for the mutant sampling technique, method level technique, and All Operators technique.

1. Calculator

TABLE VI
CALCULATOR PROGRAM RESULTS

Selection Method	Total mutants	Alive mutants	Dead mutants	Score %
All Operators	45	32	13	71.0%
Class Level	NA	NA	N A	NA
Method Level	45	32	13	71.0%
Mutant Sampling	42	26	13	50%

From Table VI, we notice that the class level operators' selection method was not applicable. This is actually due to the nature of the program since the features that those operators' addresses do not exist in the calculator code.

2. Result Graph

Based on the results of the all programs obtained in our empirical study, we summarize the mutation testing score of each of the four programs tested in Fig. 3, which illustrates a bar graph of scores using the four different proposed techniques.

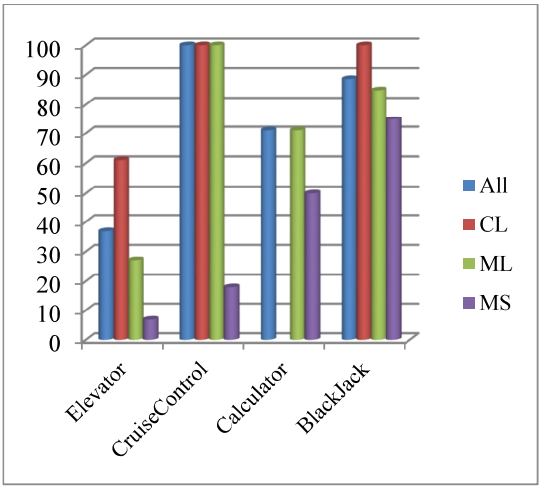

Fig. 3 Mutation Testing Scores Bar Graph

In this bar graph, the x-axis represents the different programs used, and the y-axis represents the testing score (percentage %) of the program.

- The blue color, ALL, represents the score using "All operators" selection technique.
- The red color, CL, represents the score using "Class-Level" selection technique.
- The green color, ML, represents the score using "Method-Level" selection technique.
- And the purple color, MS, represents the score using "Mutant Sampling" selection technique.

A quick analysis of the graph shows that, using mutant sampling operator selection method, the mutation score of CruiseControl and BlackJack applications for all four techniques are quite higher than the score of the other two applications except for the Method Sampling technique of the CruiseControl application. These two applications have logical codes that use polymorphism. In addition, the Elevator application whose code uses inheritance has the lowest score for all four techniques. So we can conclude that the code type has a big effect on the mutation score and that polymorphism is the best suited type for the mutation score.

VIII. Conclusion and Future Work

Nowadays, mutation testing has been used as a fault injection technique to measure test adequacy. It is done by seeding artificial faults into the original source code and checking whether the test suite is able to find the errors.

In this paper, four java open source code applications of different length and complexity have automatically injected by faults and, then, analyzed by computing the score of each program for each of the four selection testing techniques, All operators, Class level, Method level, and Mutation sampling.

The results of our experiment have proved that reducing mutants do not necessary mean that the testing score will decrease. The research has suggested that the choice of the program code type, especially polymorphism, can have an impact on the effectiveness of the mutation testing score and mutant.

This research opens the opportunity to consider other possible program code types and check if they affect the mutation score or not. It provides a field for more research on the future; for example, on the mutant sampling selection technique other combinations can be used and compared to the ones used in this paper. Moreover, bigger, diverse and additional programs can be tested following the same pattern in order to confirm our idea, or may be come up with new one.

References

[1] B. Falah. "An Approach to Regression Test Selection Based on Complexity Metrics", Scholar's Press, ISBN-10: 3639518683, ISBN-13: 978-3639518689, Pages: 136, October 28, 2013.
[2] B. Falah, K. Magel. "Test Case Selection Based on a Spectrum of Complexity Metrics". Ph.D. Dissertation. North Dakota State University, Fargo, ND, USA. Advisor(s) Kenneth Magel. 978-1-124-62588-1.
[3] B. Falah, K. Magel, O. El Ariss. "A Complex Based Regression Test Selection Strategy", Computer Science & Engineering: An International Journal (CSEIJ), Vol.2, No.5, October 2012.
[4] B. Beizer. Software testing techniques (2nd ed.). ISBN:0-442-20672-0, Van Nostrand Reinhold Co.,New York, NY, USA, 1990.
[5] R. N. Charette. Why software fails, Spectrum IEEE, Volume 42, Issue 9, Pages 42-49, September, 2005.
[6] M. Plol, M. Piattini, and I. G. Rogriguez, "Decreasing the Cost of Mutaion Teting with the Second Order Mutants," Software Testing, Verification and Reliability, vol. 19, pp. 111-131, June 2009.
[7] Y. Jia and M. Harman, An analysis and Survey of the Development of Mutation Testing, IEEE transactions on Software Engineering, Volume 37, Issue 5, pages 649-678, 2011.
[8] P. R. Mateo, M. P. Usaola, Mutant Execution Cost Reduction through MUSIC (MUtant Schema Improved with extra Code), ICST '12 Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, pages 664-672.
[9] B. H. Smith, L. Williams. Should Software Testers Use Mutation Analysis to Augment a Test Set? Journal of Systems and Software, Volume 82 Issue 11, pages 1819-1832, Elsevier Science Inc. New York, NY, USA, November, 2009.
[10] A. P. Mathur and W. E. Wong, "An Empirical Comparison of Mutation and Data Flow Based Test Adequacy Criteria," Purdue University, West Lafayette, Indiana, Technique Report, 1993.
[11] W. E. Wong, "On Mutation and Data Flow," PhD Thesis, Purdue University, West Lafayette, Indiana, 1993.
[12] M. Sahinoglu and E. H. Spafford, "A Bayes Sequential Statistical Procedure for Approving Software Products," in Proceedings of the IFIP Conference on Approving Software Products (ASP'90). Garmisch Partenkirchen, Germany: Elsevier Science, pp. 43–56, September 1990.
[13] Y-S. Ma, J. Offutt, and Y. R. Kwon, MuJava: An Automated Class Mutation System, Journal of Software Testing, Verification and Reliability, 2005, pages. 97 - 133.
[14] A. J. Offutt, Y. S. Ma, and Y. R. Kwon, "An Experimental Mutation System for Java," ACM SIGSOFT Software Engineering Notes (SECTION: Workshop on empirical research in software testing papers), 29(5), Sep. 2004.
[15] Y. S. Ma, Y. R. Kwon, and J. Offutt, "Inter-Class Mutation Operators for Java," In Proc. of ISSRE 2002, pages 352–363, IEEE Computer Society Press, Nov. 2002.
[16] A. Jefferson Offutt and K. N. King, "A Fortran Language System for Mutation-Based Software Testing," Software Practice and Experience, 21(7):686–718, July 1991.
[17] E. S. Mresa, L. Bottaci. (1999). Efficiency of mutation operators and selective mutation strategies: An empirical study, Journal of Software

Testing, Verification and Reliability, Volume 9, Issue 4, pages 205–232, December 1999.
[18] Y-S. Ma and J. Offut, Description of Method Level Mutation Operators for Java, December 2005, http://cs.gmu.edu/~offutt/mujava/mutopsMethod.pdf
[19] Y-S. Ma and J. Offutt, Description of Class Mutation Operators for Java, December 2005, http://cs.gmu.edu/~offutt/mujava/mutopsClass.pdf
[20] Muclipse: An Open Source Mutation Testing Plug-in for Eclipse, July 2008, http://muclipse.sourceforge.net
[21] Bernhard J. M. Grun, David Schuler, and Andreas Zeller. The Impact of Equivalent Mutants. Proc. 2nd International Workshop on Multi-Core Software Engineering (IWMSE), Pages 49-55, May 2009.

Dr. Bouchaib Falah is born in Casablanca, Morocco. He received his PH.D in Software Engineering from North Dakota State University, U.S.A, in 2011, a master degree in Computer Science from Shippensburg University, Pennsylvania, U.S.A, in 2001, and a bachelor degree/teaching certificate from Ecole Normale Superieure, Casablanca, Morocco in 1992.

Offering more than 20 years of combined experience developing and implementing computer science and technical math curriculum for different colleges and universities as well as web designer for multimillion-dollar organizations in USA and researcher in different projects, Dr. Falah is currently an Assistant Professor at Al Akhawayn University, teaching graduate and undergraduate software engineering courses, School of Science and Engineering. Beside teaching high school level math in Morocco and college mathematics and computer science at Harrisburg Area Community College in Pennsylvania, Suny Orange Community College in New York, Pennsylvania State University in Pennsylvania, Central Pennsylvania College in Pennsylvania, Concordia College in Minnesota, and North Dakota State University in North Dakota, he has an extensive industrial experience with Agri-ImaGIS, Synertich, and Commonwealth of Pennsylvania Department of Environmental Protection. His current research interests include Complexity Metrics, Security Testing, Agile Methodology and Extreme Programming, Mutation Testing, Regression Testing, Software Engineering Processes, Web Application testing, and Design and Architecture. He published a book titled: An Approach to Regression Test Selection Based on Complexity Metrics, Scholar's Press as well as many journal papers on software testing.

In 2014, Dr. Falah became a member of APCBEES and in 2013, he was awarded certificates from International Association of CS and IT as well as ARPN journal of systems and software.