CAPITAL UNIVERSITY OF SCIENCE AND TECHNOLOGY, ISLAMABAD

Improving test adequacy assessment by novel JavaScript mutation operators

by

Muneeb Muzamal

A thesis submitted in partial fulfillment for the degree of Master of Science

in the

Faculty of Computing
Department of Computer Science

2018

I dedicate my dissertation work to my family, teachers and friends. A special feeling of gratitude is for my loving parents for their love, endless support and encouragement.

CAPITAL UNIVERSITY OF SCIENCE & TECHNOLOGY
ISLAMABAD

CERTIFICATE OF APPROVAL

Improving test adequacy assessment by novel JavaScript mutation operators

by

Muneeb Muzamal
MCS153006

THESIS EXAMINING COMMITTEE

S. No.	Examiner	Name	Organization
(a)	External Examiner	Dr. Muhammad Uzair Khan	FAST, Islamabad
(b)	Internal Examiner	Dr. Azhar Iqbal	CUST, Islamabad
(c)	Supervisor	Dr. Aamer Nadeem	CUST, Islamabad

Dr. Aamer Nadeem
Thesis Supervisor
May, 2018

Dr. Nayyer Masood
Head
Dept. of Computer Science
May, 2018

Dr. Muhammad Abdul Qadir
Dean
Faculty of Computing
May, 2018

Author's Declaration

I, **Muneeb Muzamal** hereby state that my MS thesis titled "**Improving test adequacy assessment by novel JavaScript mutation operators**" is my own work and has not been submitted previously by me for taking any degree from Capital University of Science and Technology, Islamabad or anywhere else in the country/abroad.

At any time if my statement is found to be incorrect even after my graduation, the University has the right to withdraw my MS Degree.

(Muneeb Muzamal)

Registration No: MCS153006

Plagiarism Undertaking

I solemnly declare that research work presented in this thesis titled "*Improving test adequacy assessment by novel JavaScript mutation operators*" is solely my research work with no significant contribution from any other person. Small contribution/help wherever taken has been dully acknowledged and that complete thesis has been written by me.

I understand the zero tolerance policy of the HEC and Capital University of Science and Technology towards plagiarism. Therefore, I as an author of the above titled thesis declare that no portion of my thesis has been plagiarized and any material used as reference is properly referred/cited.

I undertake that if I am found guilty of any formal plagiarism in the above titled thesis even after award of MS Degree, the University reserves the right to withdraw/revoke my MS degree and that HEC and the University have the right to publish my name on the HEC/University website on which names of students are placed who submitted plagiarized work.

(Muneeb Muzamal)

Registration No: MCS153006

Acknowledgements

All worship and praise is for ALLAH (S.W.T), the creator of whole worlds. First and leading, I would like to say thanks to Him for providing me the strength, knowledge and blessings to complete this research work. Secondly, special thanks to my respected supervisor Dr. Aamer Nadeem for his assistance, valuable time and guidance. I sincerely thank him for his support, encouragement and advice in the research area. He enabled me to develop an understanding of the subject. He has taught me, both consciously and unconsciously, how good experimental work is carried out. Sir you will always be remembered in my prayers. I would also like to thank all members of CSD research group for their comments and feedback on my research work. I am highly beholden to my parents, for their assistance, support (moral as well as financial) and encouragement throughout the completion of this Master of Science degree. This all is due to love that they shower on me in every moment of my life. No words can ever be sufficient for the gratitude I have for my parents. I hope I have met my parents high expectations. I pray to ALLAH (S.W.T) that may He bestow me with true success in all fields in both worlds and shower His blessed knowledge upon me for the betterment of all Muslims and whole Mankind.

Aameen

(Muneeb Muzamal)

Registration No: MCS153006

Abstract

Software testing is an essential process to verify that software meets its specifications and to detect the faults and works as intended. Mutation testing is an effective software testing technique to assess the adequacy of the test suite. In mutation testing technique, we take the original program as input and make variants of it by introducing change in each variant by using defined mutation operators. These variants are called mutants. After creating mutants of the original program, we execute test cases on each mutant with the objective that these test cases will detect changes. To identify a change in a mutant, we require executing each mutant with each test case. If a change is detected in a mutant, then it is called a killed mutant otherwise it is considered alive. The effectiveness of a test suite depends on the number of mutants killed by it. If frequent number of mutants is killed then the test suite is deemed as adequate enough to detect changes in mutants. We measure the adequacy of the test suite by using mutation testing technique as the ratio of the number of killed mutants to the total number of mutants. In literature, a lot of research has been done on mutation testing and number of mutation operators are proposed for Java and other programming languages. However, mutation operators for JavaScript language are few in numbers as compared to mutation operators for other programming languages. The focus of our research is on JavaScript mutation operators. Nowadays, JavaScript is regressively used in the front end of the web applications. To check the adequacy of the test suite of JavaScript application, mutation testing is an appropriate approach but the mutation operators for JavaScript are few in number. These JavaScript mutation operators are used to seed faults in JavaScript source program and cover some of the specific JavaScript features but there are some features that are not addressed and require more mutation operators. In this thesis, we propose a set of new JavaScript mutation operators to address the features not covered by existing operators. Using our proposed operators, introduced faults are diverse faults that existing operators faults does not subsume proposed operators faults. We have implemented our proposed operators in our tool, Mutant Tracer, which is used to

generate mutants of JavaScript source program with existing and proposed operators, executes test cases on these generated mutants and then generates a status report of each mutant. For evaluation of our proposed mutation operators, we used mutation score approach wherein we measure the percentage of the number of the killed mutants by the total number of mutants. Based on the mutation score, it is concluded that seed faults of our proposed operator are not subsumed in faults of existing operators. We have performed the experiments on three different case studies and have attained 10 to 25 percent mutation score which indicates that proposed operators introduce different faults that are not detected by test suite and they require additional test cases.

Contents

List of Figures

List of Tables

Abbreviations

STI	String to Integer
ITS	Integer to String
CTI	Character to Integer
VRL	Var Replace Let
LRV	Let Replace Var
ILK	Insert Let Keyword
DLK	Delete Let Keyword
MS	Mutation Score
T	Test Suite
MSn	Mutation Score of proposed operators generated mutants
MSe	Mutation Score of existing operators generated mutants

Symbols

M_e	Set of non equivalent mutants generated by applying existing operators
M_n	Set of non equivalent mutants generated by applying new operators
M_{ek}	Set of existing mutants killed by T
M_{nk}	Set of new mutants killed by T
MS_e	Mutation Score of existing operators generated mutants
MS_n	Mutation Score of new operators generated mutants

Chapter 1

Introduction

1.1 Overview

In the current technological era, increased human dependence on computer systems has also increased the demand of the software development. Humans are becoming more dependent on different types of software to perform certain tasks. For instance, humans are examined by utilizing some critical types of software systems such as, health care systems wherein the patients are automatically monitored with the help of machines controlled by software. Therefore, before using such machines there should be a proper testing of controlling software as they have a major role to accurately guide the doctors to treat the patient accordingly. Similarly, with help of the internet everyone can access their bank accounts through the mobile devices and computers to perform the transactions. In such systems, proper security testing of the software is required to protect the confidential data of users. Software testing is performed to ensure the performance of a software that whether it works according to the requirements or not Offutt (1994), Chauhan (2010). A lot of time and resources are consumed during the software testing phase Srivastava & Kim (2009), Mantere (2003), Rajappa et al. (2008). Almost half of the resources of the software development are consumed in software testing Doungsa-ard et al. (2007), Ribeiro et al. (2008). Due to necessity

of testing a software after its development, various studies have been conducted to make the testing process more cost effective. Two basic testing techniques include Black box testing and White Box testing. Black box testing is known as functional testing and white box testing is known as structural testing. Functionality requirements are checked in the functional testing, whereas the code testing is performed in the structural testing Last et al. (2005) Sthamer (1995) Sharma et al. (2014). The type of testing in which the white and black box testing are combined is known as Grey Box testing DeMillo et al. (1978). However, by using the above-mentioned testing techniques, the effectiveness or adequacy of the test cases cannot be checked as the testing techniques are the coverage based testing techniques.

1.2 Mutation Testing

Mutation testing (Budd & Angluin 1982) is a code-based testing technique in which faults are introduced to measure the effectiveness of a test suite. In this technique, faults are introduced into the program by creating a set of Faulty Programs (FP) such that FP=FP1,FP2,,FPn (where FPi represents a faulty program in FP) of original program P. These faulty versions are created from the original program by applying different mutation operators. Mutation operators: are those operators that are used to introduce one change in each version of the original program. Mutant: when we introduce a change or seed fault in the original program then this faulty version of the original program is called mutant. Test cases are used to execute mutants with the goal that every mutant should produce different output from the original program. The same test case is used to execute both the original program P and the mutant FPi, then it compares the FPi output with the P output. Killed mutant: if the output of FPi is different from P then the mutant is said to be killed. Alive mutant: if the mutant FPi is not killed by existing test cases then the mutant FPi is said to be alive. Equivalent mutant: if the mutant FPi is not killable by any of the test cases and it remains always alive then it is called the equivalent mutant. These mutants are syntactically different

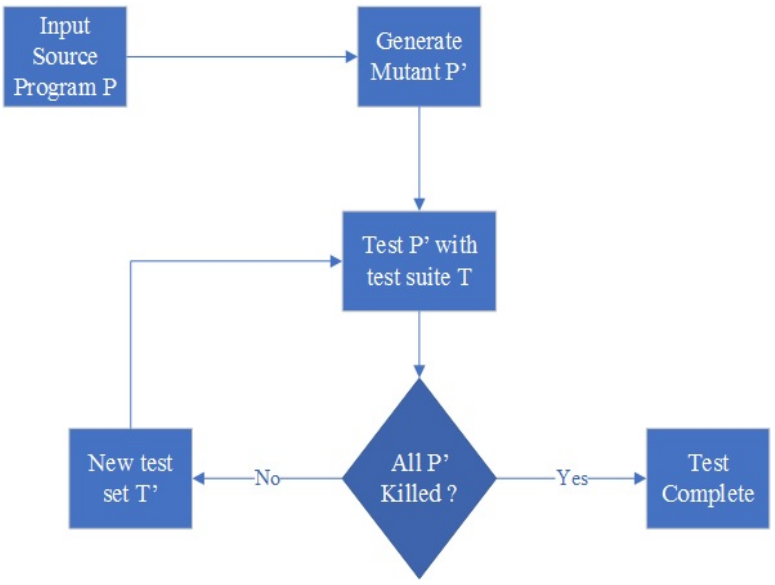


FIGURE 1.1: General Process of Mutation Testing.

but semantically equivalent to the original program. Automatic detection of these equivalent mutants is impossible (Offutt & Pan 1997), (Jia & Harman 2011) as it is an undecidable problem. After executing all mutants with test cases, mutation score is calculated, which indicates the adequacy or quality of a test suite. Mutation Score: is ratio of a number of killed mutant by a total number of mutants except for equivalent mutants (Lipton 1971). Purpose of mutation analysis is to raise the mutation score indicating that the Test Suite T is sufficient to detect faults. Figure 1.1 shows the general process of mutation testing.

Mutation testing is used to design new test suites and evaluate the quality of existing test suites. It can be used to test software at unit level, the integration level and the system level and it is a form of White box testing. It helps the tester develop effective test cases. Mutation testing does not test the software directly; it tests the test suite of the software and helps to improve them. The assumption is that test suite that detects more faults will also detect more potential faults. Therefore, they help to improve the quality of the software. Mutation testing has very high computational cost in terms of mutants compilation and execution with test suite. Equivalent mutants also involves very high computational cost because they are syntactically different but semantically equal to the original program. We execute test cases again and again on these mutants and they always produce

same output as original program which takes a very high computational cost.

History of a mutation testing can be traced back to 1971 in the study conducted by Lipton (1971). It has widely being studied since 1971 and first survey related to mutation testing was conducted in 1989 by DeMillo (1989). Woodward conducted a survey of very specific sub-area of Weak, Strong and Firm mutation testing approaches (Agrawal et al. 1989*a*). Different authors presented introductory chapters on mutation testing in their books (Woodward 1993, Ammann & Offutt 2016). Offutt and Untch (2001) summarized the history of mutation testing and briefly described the existing optimization techniques in the survey.

We can use mutation testing for testing software at a unit level, specification level and integration level (Mathur 2013, Offutt & Untch 2001). It has been applied to different programming languages like C language (Delamaro et al. 2001, Delamaro & Maldonado 1999), C# language (Agrawal et al. 1989*b*, Shahriar & Zulkernine 2008), Fotran language (Derezinska & Szustek 2008), SQL code (Derezinska 2003, Budd & Sayward 1977) and Java language (Chan et al. 2005), (Chevalley 2001, Chevalley & Thevenod-Fosse 2003) etc. Mutation testing can also be applied at the design level to the specification of a program. Examples of design level mutation testing are Statecharts (Ma et al. 2006), (Trakhtenbrot 2007), (Fraser & Wotawa 2007), Network protocols (Yoon et al. 1998), (Yoon et al. 1998), Finite State Machines (Sidhu & Leung 1988), (Jing et al. 2008), (Bombieri et al. 2008) and Web Services (Batth et al. 2007), (Fabbri et al. 1994). We use mutation testing for two purposes: (1) it is most commonly used to assess the adequacy of a test suite, and (2) it is also used to generate test cases. Most of the time, mutation testing is used for adequacy assessment but sometimes it is also used to generate test cases. Although it is an effective technique but still it has certain issues such as, very high computational cost of executing the enormous number of mutants against a test suite. The other issues include the equivalent mutant problem (Lee et al. 2008) and human oracle problem (Xu et al. 2005) which involved a lot of human effort. Human oracle problem is defined as a process of comparing output of the original program with the output of a mutant while executing each test

case. This is the most expensive part of a testing activity. Although it is impossible to completely overcome such issues, however, various studies have focused on reducing the computational cost (Weyuker 1982).

1.2.1 JavaScript Mutation Testing

Web applications are client-server software in which user interfaces run in the web browser and the client can access these interfaces through web browsers. These web applications are typically developed using diverse frameworks and web components (Praphamontripong & Offutt 2010), (Offutt 2002). Web components are developed in different programming languages, including Java Servlets, Active Server Pages (ASPs), Java Server Pages (JSPs), JavaScripts, AJAX, and PHP. These web components are integrated dynamically and they may reside in different locations. In literature, a lot of research has been done on mutation testing and number of mutation operators are proposed for Java and other programming languages but not much mutation operators available for scripting languages like JavaScript. The focus of our research is on JavaScript mutation operators. Nowadays, JavaScript is regressively used in the front end of the web applications. To check the adequacy of the test suite of JavaScript application, a number of JavaScript mutation operators have been proposed for generating mutants. Nishiura K. et al. (2013) proposed mutation operators according to JavaScript features that are event driven, asynchronous communication, and DOM manipulation. Shabnam, and Karthik (2013) proposed mutation operators that cover the variable, branch statements, and some JavaScript specific mutation operators. Purpose of these mutation operators is to introduce changes in JavaScript program and then observe the behavior of web application by running the test cases.

1.3 Problem Statement of Thesis

Mutation testing technique is widely being used to measure the effectiveness of a test suite by applying different mutation operators. However, the existing

JavaScript mutation operators are not enough to measure the effectiveness of a test suite. JavaScript specific features like variable type, scope and let keyword insertion deletion are not fully covered by the existing JavaScript mutation operators due to which the generated mutants are not adequate. Therefore, the test suite adequacy is not accurate. There should be some new JavaScript mutation operators that can play a role in enhancing the adequacy assessment via computing the adequacy of test suite in a more comprehensive way.

1.4 Research Questions

In this research work, we will implement a new set of JavaScript mutation operators that will generate the mutants of the original source program. However, the following questions must be taken into account:

RQ. 1: To what extent the JavaScript specific features are covered by existing JavaScript mutation operators? To answer this research question, a detailed literature survey is conducted through which we have identified the existing JavaScript mutation operators and JavaScript specific features that have not been covered in the existing JavaScript mutation operators related studies.

RQ. 2: How effective are the proposed JavaScript mutation operators in assessing adequacy of a test suite? To answer this research question, a number of experiments are performed on different case studies with their respective test suites and Mutation Score ratio is calculated for proposed and existing operators to perform the comparison. Focus of this research is to address the aforementioned research questions.

1.5 Research Objectives

The objective of this thesis is to propose a new set of JavaScript mutation operators that seed diverse faults in JavaScript source programs that are not seeded by the existing JavaScript mutation operators.

1.6 Research Methodology

1. First of all, we have comprehensively reviewed a literature to identify the existing mutation operators in JavaScript scripting language. After studying various mutation operators, we have concluded that few JavaScript features are covered by the existing mutation operators while some of the JavaScript features are ignored by the mutation operators that are of significance.

2. To overcome the gap of the mutation testing for the JavaScript scripting language, we have proposed a new set of mutation operators for JavaScript that will increase the test suite effectiveness for the features not covered by the current JavaScript mutation operators.

3. The work on the mutation operator implementation is performed as following steps:

    - We have implemented a set of mutation operators for the JavaScript features that are not covered by the existing JavaScript mutation operators.
    - In the next phase, all the data including JavaScript source program are collected. After collecting the source programs, a test suite of each source program is created by applying the worst-case boundary value analysis technique. After data collection, mutants of the source program are created by applying existing and proposed JavaScript mutation operators.
    - After creating the mutants of the source program, mutation testing is performed in which each mutant is executed along with an original source program against all the test cases. Then the output of both original and source program is compared. Afterwards, we have counted all the killed mutants along with the number of test cases that have killed the mutants of existing and proposed JavaScript mutation operators separately.

4. After executing all mutants with test cases, we have evaluated our proposed operators. We have measured the Mutation Score ratio for existing and proposed operators generated mutants based upon the data that we maintain during execution.

1.7 Research Contribution

Research contribution of this thesis is to introduce a set of new JavaScript mutation operators that could be used to introduce faults in JavaScript programs. In mutation testing, faults are seeded in original source program by using different mutation operators, thus creating number of mutants of original source program. Our proposed JavaScript mutation operators have introduced diverse faults that are currently not seeded by existing JavaScript mutation operators.

1.8 Thesis Organization

Rest of the thesis is organized as follows: second chapter presents the existing JavaScript mutation operators. Proposed solution is described in third chapter. Fourth chapter presents the implementation details. Fifth chapter is about results and discussion and sixth chapter about conclusion and future directions of a conducted study.

Chapter 2

Literature Review

Mutation testing is considered as one of the very effective testing technique in software testing for assessing the effectiveness of test suite. In this technique, multiple copies of the original program are made by introducing a fault in each copy. Faults are the minor syntactic change in the original program and these changes are introduced by applying different mutation operators. This faulty copy of an original program is called mutant. Now, these faults are identified by executing test cases. Each mutant executes against each test case until the different output is getting from the original program. This chapter contains an overview of existing mutation operators which are already proposed by different researchers. Purpose of these mutation operators is to introduce changes in JavaScript code and then observe the behavior of web application.

2.1 JavaScript Feature based mutation operators

Nishiura et al. (2013) defines mutation operators by focusing JavaScript features. They first conducted feature analysis on JavaScript and then defined mutation operators based on the result of the analysis. They describe three characteristics that

TABLE 2.1: User event feature mutation operators

JavaScript feature	Operator name	Original code	Mutated code
User event registration	Event target replacement	buyButton.click(requestBuy)	cancelButton.click(requestBuy)
	Event type replacement	button.click(showDetail)	button.**mouseover**(showDetail)
	Event callback replacement	cancelButton.click(closeModal)	cancelButton.click(requestBuy)

TABLE 2.2: Description of user event mutation operators

JavaScript feature	Operator Name	Description
User event registration	Event target replacement	Replaces the target DOM element
	Event type replacement	Replaces the event type
	Event callback replacement	Replaces the callback function

distinguish JavaScript web applications from traditional web applications: event-driven model, DOM manipulation, and asynchronous communication. Based upon the features of JavaScript applications, different mutation operators are proposed by the researchers.

2.1.1 Event driven model

Implementing events in JavaScript programs, developers determine the event, target, and callback function. The target corresponds to the DOM elements such as buttons and the callback function is the response of event type. Table 2.1. shows the mutation operators for user event feature with examples and Table 2.2. describe these operators.

2.1.2 Asynchronous communication

Enables web applications to continuously accept user request while waiting for server responses. In asynchronous communication, two main constituents are request and response. A request contains a destination URL and a request method (GET, POST). The server processes the request and sends the response to the application. Callback functions are invoked according to the status of the response.

TABLE 2.3: Asynchronous communication feature mutation operators.

JavaScript feature	Operator name	Original code	Mutated code
Asynchronous communication	Request target replacement	$.get('item.php', showItem)	$.get('item_list.php', showItem)
	Request onsuccess callback replacement	$.get('item.php', showItem)	$.get('item.php', buyItem)

TABLE 2.4: Description of Asynchronous communication mutation operators

JavaScript feature	Operator Name	Description
Asynchronous communication	Request target replacement	Replaces the Request target
	Request onsuccess callback replacement	Replaces the callback function

TABLE 2.5: DOM manipulation feature mutation operators

JavaScript feature	Operator name	Original code	Mutated code
DOM manipulation	Nearby DOM element	$(" #items"). append(newItem)	**$(**" #items"). parent (). append(newItem))
	Attribute assignment target replacement	element.id = "cancelButton"	element.textContent = "cancelButton"
	Attribute assignment value replacement	element.id =" cancelButton"	element.id = "buyButton"

TABLE 2.6: Description of DOM manipulation mutation operators

JavaScript feature	Operator Name	Description
DOM manipulation	Nearby DOM element	Select a DOM element that is near a proper one
	Attribute assignment target replacement	Replaces the target attribute
	Attribute assignment value replacement	Replaces the value of target attribute

Table 2.3 shows the mutation operators for asynchronous communication with example and Table 2.4 describe these operators.

2.1.3 DOM manipulation

DOM manipulation consists of target DOM elements and these elements are selected by their relative position to another element. Table 2.5 shows the mutation operators for DOM manipulation with example and Table 2.6 describe these operators.

2.2 JavaScript specific mutation operators

Mirshokraie et al. (2013) also proposed mutation operators for JavaScript and they also discuss two main issues due to which mutation testing suffers. First, there is a high computational cost in executing test suite against a large set of generated mutants. Secondly, this requires a significant amount of effort in distinguishing equivalent mutants, which are syntactically different but semantically identical to the original program (Budd & Angluin 1982). Equivalent mutants have no observable effect on the applications behavior, so cannot be killed by any test case. They propose a generic mutation testing approach that guides mutation generation process towards effective mutations, mutations that have a clear impact on applications behavior and as such are potentially non-equivalent. They only select, and mutate critical behavior affecting portions of the application code in their approach. They implement their approach in a tool called MUTANDIS. They show that on average, 93% of the mutants generated by MUTANDIS are non-equivalent. In the first part of their approach, they extracted the JavaScript code from a given web application, executed the extracted code with the existing test suite, and gathered detailed execution traces of the application under test. Then they extracted the variable usage frequency, dynamic invariants and dynamic call graph from execution traces. Using dynamic call graph, rank the programs function that has an impact on applications behavior. Within the highly ranked functions, they identified the variables that have a significant impact on functions outcome, and selectively mutated only those to reduce the likelihood of equivalent mutants. Based on the proposed approach, they also proposed specific JavaScript mutation operators that are implemented in MUTANDIS. They targeted variables, branch statements, and specific JavaScript operators to perform mutation steps. Table 2.7 describe the specific JavaScript mutation operators. Variables and branch statement generic mutation operator types are applied using the ranking technique, while JavaScript specific mutation operator type is applied regardless of the ranking. These JavaScript specific operators are known to be error-prone.

TABLE 2.7: JavaScript-Specific Mutation Operators

Type	Operator Name	Description
JavaScript Specific Mutation Operator	Adding/Removing the **var** keyword	Preventing overwriting of global variables, use var keyword in function declares the variable local scope
	Removing the global search flag from **replace**	Assuming that the string replace method affects all possible matches. It only changes the first occurrence
	Removing the integer base argument from **parseInt**	Assuming that parseInt returns the integer value to base 10, however the second argument of parseInt, which is the base, varies from 2 to 36
	Changing setTimeout function	setTimeout (f, 3000) be the function that should be executed after 3000ms. Addition of parentheses "()" with function name, i.e. setTimeout(f(), 3000) invokes the function immediately
	Replacing**(function()!==false) by (function())**	If a function does not return a value, while developer expects a boolean outcome, then the returned value is undefined. Undefined is considered as false, the condition statement will not be satisfied

Mirshokraie et al. (2015) describe their previous work in more detail and evaluates their technique using eight JavaScript applications. In this paper, they discuss their technique called FunctionRank that they developed for reducing the number of equivalent mutants. This technique selects only those functions that have a significant impact on applications behavior and then mutation operators are applied for the mutant generation. They implement their technique in a tool called MUTANDIS. They apply their technique to different JavaScript applications and evaluates their technique. They conclude that on average, only 7% of equivalent were generated by MUTANDIS and more than 70% of equivalent mutants originated from the branch mutation category. Table 2.8 provide list of all existing JavaScript mutation operators.

2.3 Critical Analysis

Although, in literature, many JavaScript mutation operators are defined that covers some JavaScript features, some generic mutation operators and some specific to the JavaScript. But all of these operators are not sufficient to test the adequacy of a test suite. In the next section, we will define some parameters for comparison between existing approaches.

TABLE 2.8: List of all existing JavaScript Mutation operators

Category	Name of JavaScript Mutation operator
User event registration	Event target replacement
	Event type replacement
	Event callback replacement
Asynchronous communication	Request target replacement
	Request onsuccess callback replacement
DOM manipulation	Nearby DOM element
	Attribute assignment target replacement
	Attribute assignment value replacement
JavaScript Specific Mutation Operator	Adding/Removing the **var** keyword
	Removing the global search flag from **replace**
	Removing the integer base argument from **parseInt**
	Changing setTimeout function
	Replacing**(function()!==false) by (function())**

2.3.1 Define evaluation criteria

In this section, we will define some parameters for gap analysis in existing approaches. We divide parameters into three categories, all of these define below in detail.

2.3.1.1 Tool

Purpose of this category is to check, whether the existing approaches implemented have their tools and performs all the required steps that are essential for mutation testing approach. Parameters of this category define below: I. General Mutation Operators There are so many general purpose mutation operators like ROR, LOR, and AOR etc. that can be used for many programming languages. Purpose of this parameter is to check whether all of these general purpose mutation operators used in existing approaches tool or not. II. Mutation Generation Mutation generation is an essential part of mutation testing approach. Purpose of this parameter is to check whether the existing approaches tool performs this essential part or not.

III. Test case execution Test case execution is another essential part of mutation testing approach. Test cases are executed on generated mutants and original program with the goal that each mutant generate different output. Purpose of

this parameter is to check whether existing approaches tool performs this essential part or not.

IV. Mutation score Mutation score is another essential part of mutation testing approach. Mutation score is a percentage of no. of killed mutants by no. of total mutants. This parameter defines with the purpose that whether existing approaches tool performs this essential part. V. Equivalent mutants We define this parameter with the purpose to check that the existing approaches tool avoid generating equivalent mutant or not.

VI. Bug Severity Bug severity is that how much injected faults are severe. Whether the injected faults cause the major data loss, major loss of functionality or some minor loss of functionality. This parameter defines to check existing approaches tool provide this information about faults.

2.3.1.2 JavaScript Features

We define this category to check whether the existing approaches cover JavaScript special features. Parameters of this category define below: I. DOM manipulation We define this parameter to check whether the existing tools handle DOM manipulation. Whether a developer incorrectly selects a DOM element or insert/delete DOM element at an improper position to identify faults.

II. Event driven We define this parameter to check whether the existing tools handle the user events or not. III. Asynchronous communication We define this parameter to check whether the existing tools handle asynchronous communication. The Client sent a request to the server and after processing request, the server responded correctly or not.

TABLE 2.9: Comparison of existing JavaScript approaches

Category	Parameter	JavaScript Feature Based Mutation Operators	JavaScript Specific Mutation Operators
Tool	General Mutation Operator	×	✓
	Mutant Generation	✓	✓
	Test case execution	✓	✓
	Mutation Score	✓	✓
	Bug Severity	×	✓
	Equivalent Mutants	×	✓
JavaScript Features	Asynchronous communication	✓	×
	DOM manipulation	✓	×
	Event driven	✓	×
JavaScript Specific	Common mistakes	×	✓
	Variable scope	×	×
	Variable datatype	×	×
	Window object	×	×

2.3.1.3 JavaScript Specific

Purpose of this category is to check whether the existing approaches cover the common mistakes in JavaScript program that a programmer can do like adding/removing var keyword with a variable in local scope. Variable declaration is done by using two different keywords and these keywords have different scope in JavaScript program. JavaScript is a loosely typed programming language in which datatype of a variable is assigned at the time of initialization of value and wrong datatype may cause an error in results. We define this category parameters to check whether existing approaches check scope and datatype of variables.Comparison of existing approaches are shown in Table 2.9.

2.4 Gap Analysis

A number of JavaScript mutation operators are proposed that are used to seed the faults in JavaScript program. There are some faults that can occur in a program that is not currently handled by the existing JavaScript mutation operators. For example, in JavaScript, `let` and `var` are two different keywords that are used to declare a variable. Variable declare with `var` keyword has scope to entire function regardless of block scope whereas variable declares with `let` keyword has the scope

limited to the current block in which it declares. The programmer may declare variable assuming that this variable has the scope limited to current block but this may happen variable has scope outside the current block if it declares with var keyword. There are no such JavaScript mutation operators that can handle the scope of a variable. As we know, JavaScript is a loosely typed programming language in which datatype of a variable depends on what type of value is assigned to that variable. A variable declares and assigns a specific datatype value and later on, in the program, this may happen the same variable can be assigned another type of a value by the programmer that can cause an error in the result. There are no such JavaScript mutation operators that can handle these kinds of faults in the program. There are also some JavaScript specific window objects that are not currently handled by the existing JavaScript mutation operators. This requires a new set of JavaScript mutation operators that will seed these kinds of faults in the program and improve the assessment of test suite.

Chapter 3

Proposed Solution

Mutation Testing is an effective testing technique in software testing domain to assess the effectiveness of test suite. Many researchers have presented the mutation operators for the JavaScript that have been discussed in chapter 2 in detail and Table 2-8 provides a list of all existing JavaScript mutation operators. We have seen in the previous chapter that there are some faults that can arise in JavaScript program that are currently not handled by the existing JavaScript mutation operators. Some of the problems have been discussed in section 2.4. Therefore existing JavaScript mutation operators are not sufficient to assess the effectiveness of test suite. We have proposed a new set of JavaScript mutation operators to improve assessment of test cases by considering the faults associated with the JavaScript program. An overall overview of our approach is depicted in Figure 3.1.

1. In mutant generator process, our tool takes JavaScript program as input then generates mutants of JavaScript program with the selected JavaScript mutation operators.

2. In mutant executor process, our tool executes mutants with test cases to check if the test cases detect faults that are seeded in the mutants by using existing and proposed mutation operators. After executing all mutants with test cases, in the next phase, for comparison, we will measure the Mutation

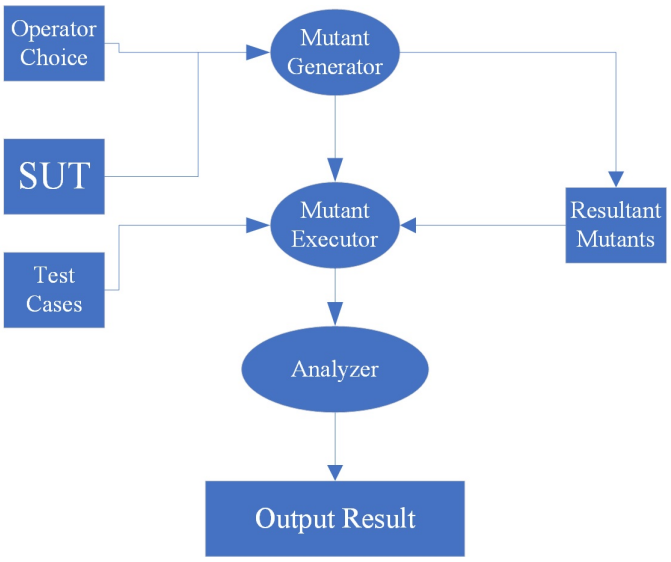


FIGURE 3.1: Flow diagram of proposed approach.

Score ratio for existing and proposed operators generated mutants based upon the data that we maintain during execution.

3. In result analyzer process, measure the Mutation Score ratio based on the results that we collect in the previous phase. Our tool generates a status report of mutants that which mutants remain alive and which mutants killed by whom test cases.

For expose faults in JavaScript program, in next section, we proposed a set of JavaScript mutation operators that will introduce diverse faults that existing operators unable to introduce such faults. In the next section, we provide detail of our proposed JavaScript operators.

3.1 Proposed Mutation Operators

The new type of mutants generated from the original program code with the proposed mutation operators are explained below with the examples.

TABLE 3.1: Changing Datatype String to Integer Datatype (STI)

Original Program	Mutated Program

if(a < b){
value = "string";
document.write("value: "+ value);
}
	if (a < b){
	value = Math.random();
	document. write ("value: "+ value);
	}

3.1.1 Changing Variable Datatype

The data type of a variable is defined by the value assigned to that variable, a common mistake can be done by assigning different data type value to the same variable. The mutation operator can further be categorized based on the 3 types of cases:

3.1.1.1 Case 1: Changing Datatype String to Integer Datatype (STI)

The mutant of the original program is created by finding the variable of the String Data type and changing it to the integer data type. This type of error can arise if the programmer uses integer literal instead of a string literal. If the programmer uses string value the output will be accurate as desired but if the programmer uses integer literal then the outcome may be incorrect and faulty. Table 3.1 demonstrated the above scenario with example. Therefore, to handle such faults we need this operator for seeding such faults.

3.1.1.2 Case 2: Changing Datatype Integer to String Datatype (ITS)

The mutant is created for the original program by changing the Integer Datatype to the String Datatype. This type of error can arise if the programmer uses string literal instead of an integer literal. If the programmer uses integer literal the output will be accurate as desired but if the programmer uses string value then the outcome may be incorrect and faulty. Table 3.2 demonstrated the above scenario with example. Therefore, to handle such faults we need this operator for seeding such faults.

TABLE 3.2: Changing Datatype Integer to String Datatype (ITS)

Original Program	Mutated Program

if(a < b){
value = 75;
document.write("value: "+ value);
}
	if(a < b){
	value = String(75);
	document.write("value: "+ value);
	}

TABLE 3.3: Changing Datatype Character to Integer Datatype (CTI)

Original Program	Mutated Program

if(a < b){
value = 's';
document.write ("value: "+ value);
}
	if(a < b){
	value = 115;
	document.write ("value: "+ value);
	}

3.1.1.3 Case 3: Changing Datatype Character to Integer Datatype (CTI)

Character datatype in the program will be changed with the integer data type to produce a mutant from the original program. This type of error can arise if the programmer uses integer literal instead of a character literal. If the programmer uses a character literal the output will be accurate as desired but if the programmer uses integer literal then the outcome may be incorrect and faulty. Table 3.3 demonstrated the above scenario with example. Therefore, to handle such faults we need this operator for seeding such faults.

3.1.2 Replacing Keyword var with let (VRL)

The scope of the variable that declares with var keyword is entire function regardless of block scope. A common mistake is assuming that var keyword declares a variable with the scope limited to the current block in which it declares but it has scope to the complete function and outside this block same variable value will be used. By replacing var with let keyword, to limit the scope of a variable to the current block and outside this block global variable value will be used. Table 3.4 demonstrated the above scenario with example.

TABLE 3.4: Replacing Keyword var with let (VRL)

Original Program	Mutated Program

a=20;
if(a < b){
var a = 10;
document.write (“value: ”+ a);
}
document.write (“value: ”+ a);
	a=20;
	if(a < b){
	let a = 10;
	document.write (“value: ”+ a);
	}
	document.write (“value: ”+ a);

TABLE 3.5: Replacing Keyword let with var (LRV)

Original Program	Mutated Program

a = 15;
if(a < b){
let a = 10;
document.write (“value: ”+ a);
}
document.write (“value: ”+ a);
	a = 15;
	if(a < b){
	var a = 10;
	document.write (“value: ”+ a);
	}
	document.write (“value: ”+ a);

3.1.3 Replacing Keyword let with var (LRV)

The scope of a variable that declares with let keyword is specific to the current block in which in declare. A programmer declares a variable in a specific block with let keyword to limit the scope of this variable and outside the block use the global variable value. By replacing let with var which extends the variable scope to complete function instead of the current code block which may cause an error in the result. Table 3.5 demonstrated the above scenario with example.

3.1.4 Insert let keyword(ILK)

If developer assumes to declare a variable inside a function as a local scope to prevent overriding of a global variable. A common mistake is to forget to add let keyword with variable inside a function which overrides global scope variable value and that value can cause an error in the result if this global variable used outside the block. By adding let keyword with the variable in block scope can limit the scope as well as prevent overriding the global scope variable value. Table 3.6 demonstrated the above scenario with example. By adding let keyword with variable in block scope can limit the scope as well as prevent overriding the global scope variable value.

TABLE 3.6: Insert let keyword(ILK)

Original Program	Mutated Program

a = 15;
if(true){
  a = 10;
  document.write ("value: "+ a);
}
document.write ("value: "+ a);
	a = 15;
	if(true){
	  let a = 10;
	  document.write ("value: "+ a);
	}
	document.write ("value: "+ a);

3.1.5 Delete let keyword (DLK)

A variable declares in global scope and uses this variable inside a function. A common mistake can be done by adding let keyword with the variable name that declares a new variable in block scope. This will prevent overwriting global variable in block scope and outside this block original value of a global variable will be used that can cause an error in the result. By deleting let keyword from variable name inside block scope to prevent the declaration of a new variable as well as allow overriding global scope variable in a block. Table 3.7 demonstrated the above scenario with example.

By defining the above JavaScript mutation operators we can able to handle JavaScript specific faults that a programmer can be done while programming and these faults cannot handle the existing mutation operators. With the proposed mutation operators, a different type of faults will be seeded in the program which includes variable scope and type faults. As we know variable declaration can be done in JavaScript with two different keywords (let, var) and both of them have different scope. With our proposed operators, we can seed faults that change the scope of a variable by replacing var keyword with let and vice versa. Similarly by deleting/inserting these keywords can also cause the change in scope of a variable. Change in scope of a variable can cause an error in the result. As we know, JavaScript is a loosely typed programming language and this may happen by mistake programmer assign another type of the value to same variable which may cause an error in the result. With our proposed mutation operators, we can also seed these kinds of faults that can change the type of a variable by assigning a different type of value

TABLE 3.7: Delete let keyword (DLK)

Original Program	**Mutated Program**

a = 15;
if(true){
  let a = 10;
  document.write (“value: ”+ a);
}
document.write (“value: ”+ a);
	a = 15;
	if(true){
	  a = 10;
	  document.write (“value: ”+ a);
	}
	document.write (“value: ”+ a);

to the same variable. By combining these proposed mutation operators with the existing mutation operators, the assessment of test cases will be improved.

Chapter 4

Implementation

4.1 Overview

This chapter contains implementation details of the proposed technique, we have developed a GUI based desktop application and implementation is done using the Object Oriented Paradigm using the JAVA language. IDE used to develop the software is NetBeans. Generic Software Architecture of tool is shown in Figure 4.1.

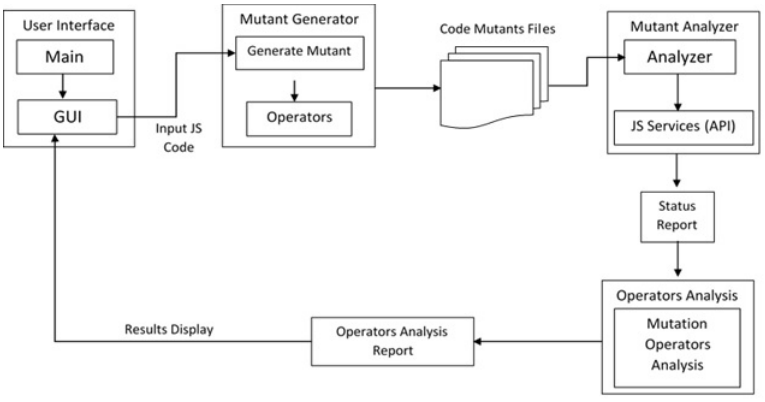
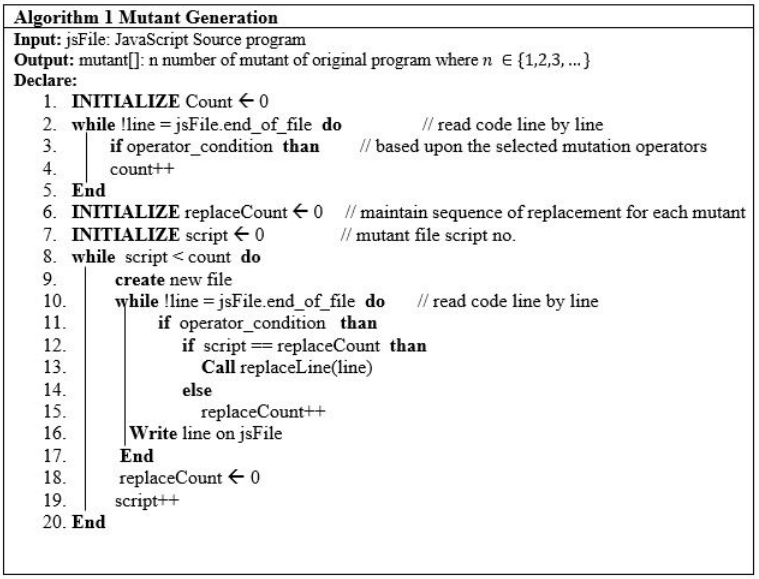


FIGURE 4.1: Tool Architecture



Algorithm 1 Mutant Generation

Input: jsFile: JavaScript Source program
Output: mutant[]: n number of mutant of original program where $n \in \{1,2,3,...\}$
Declare:

```
1.  INITIALIZE Count ← 0
2.  while !line = jsFile.end_of_file do          // read code line by line
3.      if operator_condition than               // based upon the selected mutation operators
4.      count++
5.  End
6.  INITIALIZE replaceCount ← 0   // maintain sequence of replacement for each mutant
7.  INITIALIZE script ← 0         // mutant file script no.
8.  while script < count do
9.      create new file
10.     while !line = jsFile.end_of_file do      // read code line by line
11.         if operator_condition than
12.             if script == replaceCount than
13.                 Call replaceLine(line)
14.             else
15.                 replaceCount++
16.     Write line on jsFile
17.     End
18.     replaceCount ← 0
19.     script++
20. End
```

FIGURE 4.2: Algorithm 1 Mutant Generation

4.2 Mutant generation process

The working of the tool starts with the main menu presented as a GUI to the user which asks for the JavaScript code as an input for mutation testing. Then this input JavaScript program file pass to Mutant Generator component. Multiple mutants are generated for the JavaScript code based upon selected operators. The algorithm used for the mutant generation is explained in Figure 4.2

4.2.1 Algorithm 1 description

To generate mutants of given program, first, count how many time selected mutation operator(s) will apply for the mutant generation (lines 1-5). While loop continuously read file line by line until the end of file found and in each iteration of while loop, if statement, check whether the current line contains selected operator condition or not (lines 2-3). If the condition true then increment count value by 1 (line 4). After that mutants generation process begins and a number of mutants will be created as many as the value of count variable (lines 6-20). The outer while loop executes until script value is less than the count variable value and in each iteration of this loop, every time new file create with incremented

script number (lines 8-9). Inner while loop of our algorithm, read original code from the file line by line for replacement (line 10). If statement (line 11), check whether the current line contains selected operator condition or not. If the condition true then check the sequence in which the replacement will have to apply (lines 12-13). For example, if tester select var replacing let mutation operator and var keyword found more than once in the program. Then in each mutant, replace var keyword that already not have been replaced for the creation of new mutant. If the condition is false then found the next appearance of operator condition that not already mutated (line 15). In line 16, write each line on mutant file to make a copy of the original program (line 16). At the end of the outer loop, n number of mutants will be generated.

4.3 Analyzer Executor process

After generating mutants of the source program, next phase is to execute these generated mutants with test cases that are developed by the tester for test the source program. Our tool takes test cases as input in a specific format and then execute each mutant with all test cases. Same test case executes both the original program and the mutant with the goal that each mutant should produce different output from the original program. The algorithm used for the mutant generation is explained in Figure 4.3

For the execution of JavaScript program using Java, we use Java ScriptEngine, ScriptEngineManager, and Invocable APIs. ScriptEngineManager provides the mechanism for ScriptEngine to initiate engine. ScriptEngine provides scripting functionality and it includes methods that execute scripts. Invocable interface implemented by ScriptEngine and invocableFunction of the Invocable interface is used to call functions that are defined in the script. Our algorithm read one test case in each iteration, then execute all mutants along with the original program and compare the result of both. Based upon the result, calculate the assessment of proposed operators.

Algorithm 2 Mutant Execution

Input: tcFile: Test cases file
Output: statusReport: mutant status report
oprAssessment: proposed operator assessment
Declaration:
ScriptEngineManager manager, ScriptEngine engine, File folder,
File[] listOfMutantName; int scriptNo, String tstCase;
FileReader origCode, mutCode;
killedTClist ArrayList;

```
1.  INITIALIZE manager ← new ScriptEngineManager ()
2.  INITIALIZE engine ← manager.getEngineByName ("JAVASCRIPT")
3.  INITIALIZE folder ← new File (MutantsFolderPath)
4.  INITIALIZE listOfMtName ← folder.listFiles ()
5.  INITIALIZE scriptNo ← 0
6.  INITIALIZE mutCode ← new FileReader (originalCode)
7.  while !tstCase=tcFile.end_of_file() do
8.      engine.eval (origCode)
9.      while scriptNo < TotalMutant do
10.         mutCode ← new FileReader (listOfMtName [scriptNo++])
11.         engine.eval (mutCode)
12.         origResult ← invocable.invokeFunction ( funcName, parameter(s) value)
13.         mutResult ← invocable1.invokeFunction ( funcName, parameter(s) value)
14.         if origResult != mutResult than
15.                 killedArraylist.add (mtName)
16.                 killed = 1
17.                 no_killed++
18.                 testCaseCount++
19.             Write mtName on statusReport
20.     End
21. End
22. while scriptNo < TotalMutant do
23.     if !killedArraylist(mtName) than
24.         liveArrayList.add (mtName)
25.         Write live status on statusReport
26. End
27. oprAssessment = Call NewOprAssessmentCalc ()
```

FIGURE 4.3: Algorithm 2 Mutant Execution

4.3.1 Algorithm 2 description

For the execution of mutants with test cases, we first initialize engine object by calling ScriptEngineManager API function (getEngineByName) and this function will create script engine for JavaScript, in our case (line 1 and 2). Next, initialize folder object with mutants folder path, then get all code flies from this folder and place them in file array (line 3 and 4). Next, we initialize FileReader object by assigning original program code and then this object will be used to execute the

original program (line 6). Outer loop read test cases one by one from test case file and execute all mutants with each test case (line 8). The inner loop will execute until the scriptNo value is less than the total number of generated mutants (line 9). In this loop, we first place the mutant code in FileReader object one by one in each iteration of an inner loop from File array (line 10). Then execute mutant code by using eval method of ScriptEngine API. This eval method is used to execute the specified script (JavaScript in our case) by passing script as an argument to that method (line 11). Our algorithm uses invocakeFunction method of Invocable API that invokes a function from the original code and mutated code script and execute invoked function with test case values (line 12, 13). After evaluating both script codes, compare the results that returned. If the results are different (mutant killed) then add the mutant name to the killed status list and increment the killed mutant count as well as the killed test case count. After that write this mutant name on status report along with test case which killed mutant (line 14, 19). All the remaining mutants that are not in the killed status list, add in the alive status list and write alive status of all these mutants on a status report (line 22, 26). After executing and maintaining the status of each mutant, calculate the proposed operator assignment based on the record that we collect while executing all mutants with test data (line 27).

4.4 Tool Usage

This section includes all the user interfaces of our tool.

4.4.1 Mutant Generation Interface

To start creating the mutants for the JavaScript program, our tool will take JavaScript program as input and then tester select mutation operators that will apply on JavaScript source code to generate mutants of a source program. Figure 4.4 shows the mutant generation interface. In the first step, the tester will provide

JavaScript source code and then select mutation operators for which he/she want to generate mutants. In our tool, we also provide existing JavaScript mutation operators along with our proposed JavaScript mutation operators. A tester can also select existing JavaScript mutation operators for the mutant generation. After selection of operators tester will press Mutant Generator button to generate mutants. A pop-up message will be shown on successful generation of mutants as shown in Figure 4.5. After generation of mutation, next step is to execute test cases on these generated mutants and original program.

4.4.2 Analyzer Executor process

After generating mutants of a source program, next phase is to execute these generated mutants with test cases that are developed by the tester for test the source program. Our tool takes test cases as input in a specific format and then by clicking Execute Mutants button, execute each mutant with all test cases. Mutant execution interface is shown in 4.6. List of all killed and live mutants are also shown on the interface.Our tool also keeps the record that one test case killed how many test cases and based upon this record at the end of the execution process, generate a status report that one test case killed which mutant(s) and also mention the mutant name that remains alive as shown in Figure 4.7.After executing all mutants with test data set, list of all killed and live mutants are also shown on executor/analyzer interface. Based on the data of killed and live mutants, assessment of proposed operators in terms of Mutation Score ratio calculate and shown on executor/analyzer interface. Names of alive and killed mutants, as well as the assessment in terms of Mutation Score, are shown in Figure 4.8

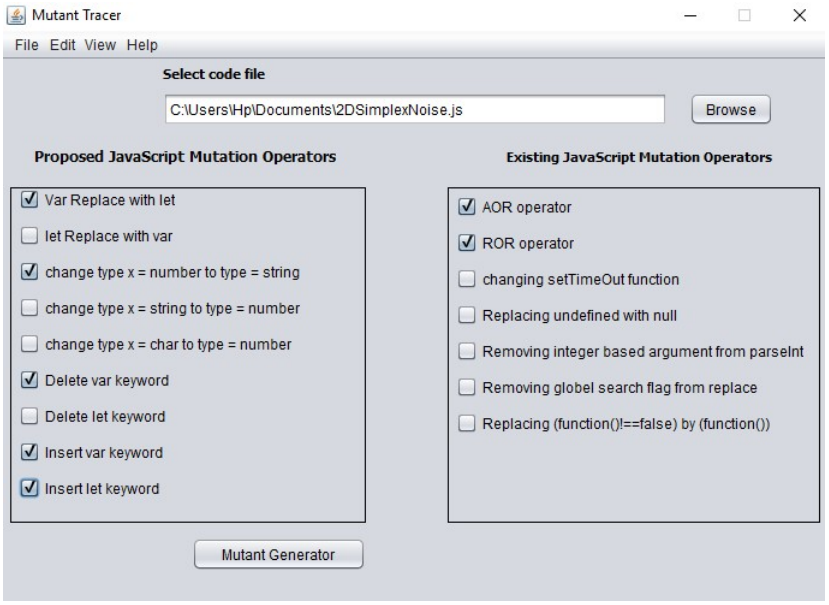


FIGURE 4.4: Mutant Generator Interface

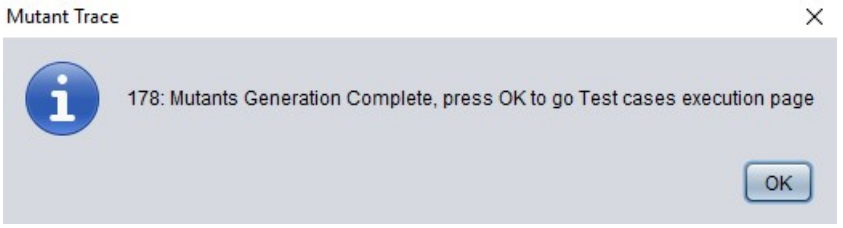


FIGURE 4.5: Pop-Up message of successfully generation of mutants

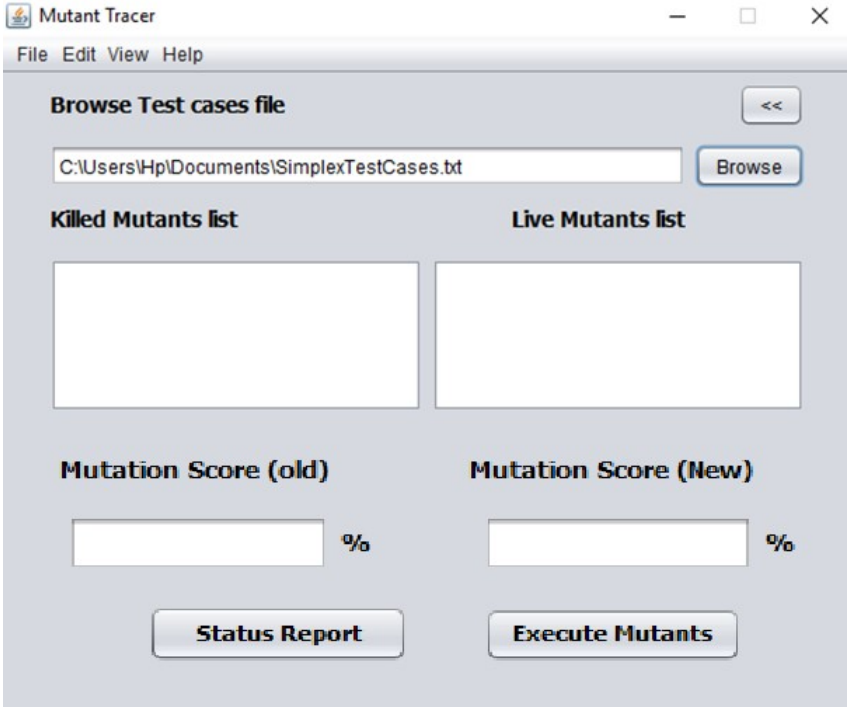


FIGURE 4.6: Mutant Analyzer Interfacee

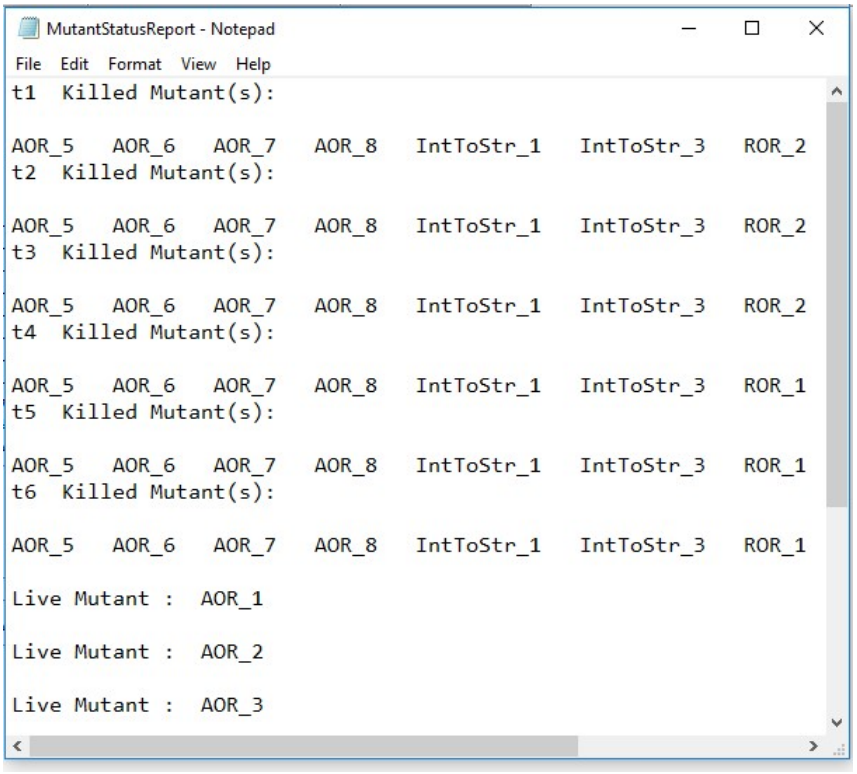


FIGURE 4.7: Mutant Status Report

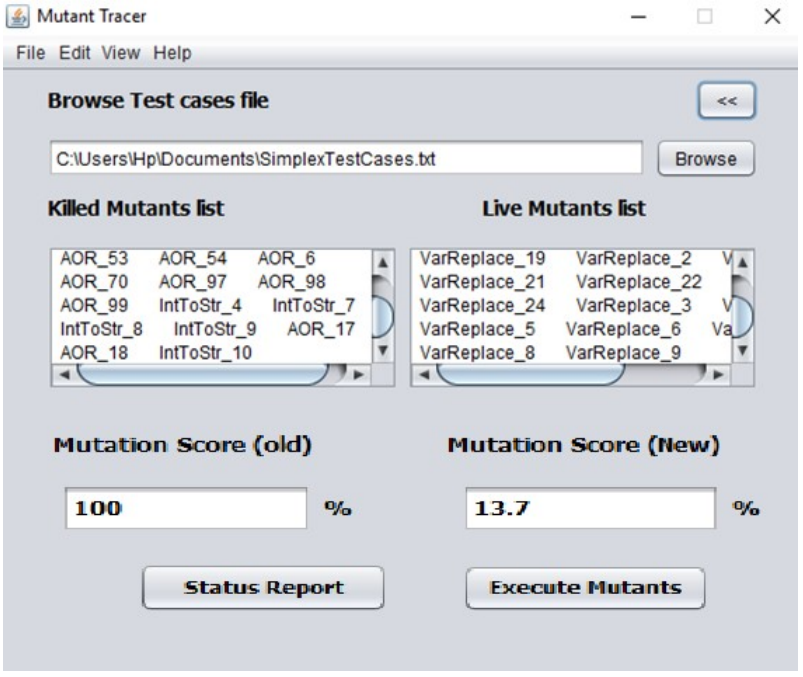


FIGURE 4.8: Result of Existing and Proposed Operators

Chapter 5

Results and Discussion

In this section, we have discussed experiments result, which we have performed on different JavaScript source code. Using existing and proposed JavaScript mutation operators, we generated mutants of original JavaScript source code and then using dataset we execute mutants along with original source code. During execution we maintain a record of each mutant (killed or alive) and based upon this record, we compare our proposed JavaScript mutation operators with existing operators that our proposed operators does not generate mutant or seed such faults which are redundant.

5.1 Evaluation Criteria

For evaluation of our proposed JavaScript mutation operators, we used mutation score based approach. We execute existing and proposed operators generated mutants with the test suite and then we calculate mutation score for both existing and proposed operators mutants separately. After that, we calculate usefulness of proposed operators that confirms how effective or useful our proposed operators in introducing diverse faults. The following formulas are used to calculate mutation score and usefulness of mutants: T=Test Suite

Me =Set of non equivalent mutants generated by applying existing operators

Mn =Set of non equivalent mutants generated by applying new operators

Mek=Set of existing mutants killed by T

Mnk= Set of new mutants killed by T

MSe=Mutation Score of existing operators generated mutants

MSn=Mutation Score of new operators generated mutants

We calculate Mutation Score for both existing and proposed operators generated mutants by following formula:

5.1.1 Redundancy approach

Redundancy is define as, the faults that seeded by our proposed operators, they are similar kind of faults that seeded by existing operators then we say proposed operators faults are redundant. To check the redundancy, we execute both set of mutants (generated with existing and proposed operators) with existing test suite and then measure the redundancy ratio. For calculation of redundancy we used following formula:

$$MS_e = \frac{M_{ek}}{M_e}, MS_n = \frac{M_{nk}}{M_n} \tag{5.1}$$

$$Createtsuchthat M_{ek} = M_e therefore MS_e = 1 \tag{5.2}$$

We have created test suite T when we execute the existing operators generated mutants with this test suite, the MSe is equal to the probability 1 which means that all the mutants that are in set Me have been killed with T. This implies that if set Me and Mek are both equal then MSeis 1. To find the usefulness of new mutants, we calculate the mutant score with new mutants and same test suite by following formula:

$$Usefulness = 1 - MS_n \tag{5.3}$$

If the Usefulness value closed to 0 then this intimates that these mutants are easy to kill. These mutants have similar kind of faults with existing mutants faults. If the Usefulness value closed to 1 then this intimates that these mutants are hard to kill. These mutants have diversity in faults and are useful to measures the

adequacy of a test suite. The faults seeded by our proposed operators are not subsumed in existing operators faults and they require new test cases to kill.

5.2 Case Studies

For evaluating our proposed JavaScript mutation operators, we have used four different JavaScript source programs. For the selection of source programs, different sources were searched including open source project repositories, and SIR (Software Infrastructure Repository). From these, repositories, we have found a lot of applications developed in JavaScript language. From these we select two applications named ECharts and Tech interview handbook. ECharts is a powerful and visualization library that provides an easy way to add interactive and highly customizable charts to your products. Tech interview handbook has practical content related algorithms that can help for technical interviews. This handbook pretty new and only have content related sorting algorithms with their code and explanation. This handbook also covers contents beyond the typical algorithmic coding questions. Mutation testing is very complex if we consider whole application due to a large number of mutants. So our focus in this thesis on method level mutation testing and we looked for reasonably sized methods in these applications. We select methods from these applications that perform some computation. Methods that have some input parameters and after performing computation returned some output. The source codes of these programs are given as an input to the tool described in Chapter 4 to generate mutants using selected (existing and proposed) JavaScript mutation operators and for execution of original and mutant programs using the test suite generated by Worst case boundary value analysis. Based on the data collected after execution, compare the proposed operators with existing operators using Mutation Score approach. We have used Simplex Noise (yangshun 2017), Linear Map (ecomfe 2016*a*) and Merge Sort (ecomfe 2016*b*) as example programs for evaluating our approach. A brief description of each program is given below:

TABLE 5.1: Line of code information

App Name	LOC
Simplex Noise	60
Linear Map	40
Merge Sort	30
ATM Machine	85

1. **Simplex Noise**: Simplex Noise algorithm is used for constructing n-dimensional noise functions and this algorithm is the extension of Perlin Noise algorithm. Perlin Noise algorithm is used to produce natural appearing textures on computer generated surfaces for visual effect on motion pictures. Purpose of Simplex Noise algorithm is same as Perlin Noise but it uses a simpler space filling and alleviates some problems with Perlin Noise. Simplex Noise method takes coordinate (x and y in our example) as input and returned final noise value.

2. **Merge Sort**:Merge Sort is one of the divide and conquer technique for sorting an unsorted array. This algorithm is one of the most popular sorting algorithm and most commonly used for sorting arrays. It first divides the array into sub-arrays and then combines these sub-arrays in a sorted manner. Merge Sort method take an unsorted array as input and return a sorted array.

3. **ATM Machine**:In this method allow the customer to complete the basic transactions like balance query, deposit amount and withdraw amount etc.

4. **Linear Map**: Linear Map method map a value from domain to range. This method takes domain values in an array, range values in an array, clamp value as Boolean and a number as input. Based upon the following conditions map domain value to the range value:

   C1: clamp == true

   C1a: number <= domain [1]

   C1b: number >= domain [0]

   C2: clamp == false

   C2a: number == domain [1]

TABLE 5.2: Information of generated mutants

Sr#	Operator Names	Simplex Noise	Linear Map	Merge Sort	ATM
1	ROR	28	56	42	42
2	AOR	144	28	8	8
3	DVK	33	5	5	4
4	IVK	9	1	2	12
5	ITS	31	5	5	9
6	VRL	33	5	5	4
7	ILK	9	-	2	12

C2b: number == domain [0]

Based on the above conditions returned respective range value to domain. Table 5.1 contain information about the line of code of each application. We have generated mutants of all these programs by applying existing and proposed mutation operators. Table 5.2 gives complete information about the generated mutants that each operator generate how many mutants.

After generating mutants, we identify non-equivalent mutants for execution. For Simplex Noise Problem, we identify existing operators generated 169 non-equivalent mutants out of 214 and proposed operators generated 51 non-equivalent mutants out of 73. Similarly, we identify non-equivalent mutants for other three problems. A detailed analysis of generated mutants with existing JavaScript mutation operators and proposed JavaScript mutation operators are describe in Table 5.3 and Table 5.4. Next step is to execute these non-equivalent mutants along with original program with the test suite. We generated test suites for Simplex Noise and Linear Map programs by using Worst case boundary value analysis. We have found test suite for Linear Map problem in ecomfe (2016*c*), so we also used this test suite along with our generated test suite for experiments. We randomly generate test cases for Merge Sort program because this program has one parameter that is unsorted array, so we randomly generate different array values. We execute mutants of each program with their respective test suite and collect the data after executing mutants. Table 5.5 gives complete information about how many non-equivalent mutants generated in total with existing and proposed operators

TABLE 5.3: Detailed analysis of existing operator generated mutants

App Name	Total Mutants	Equivalent Mutants	Non-Equivalent Mutants
Simplex Noise	214	45	169
Merge Sort	57	4	53
Linear Map	90	3	87
ATM Machine	66	19	47

TABLE 5.4: Detailed analysis of proposed operator generated mutants

Case Study Name	Total Mutants	Equivalent Mutants	Non-Equivalent Mutants	Syntax Error
Simplex Noise	73	13	46	14
Merge Sort	12	2	8	2
Linear Map	10	2	5	3
ATM Machine	28	9	18	2

and how many existing and proposed operators non-equivalent generated mutants killed.

5.3 Comparison

After executing mutants with test suites compares the existing mutation operators with proposed mutation operators based upon the result that we extract from execution phase. We calculate MS of proposed operators generated mutants and then calculate the effectiveness of our proposed operators. Complete results of all four case studies with MS percentage and effectiveness of our proposed operators in percentage are shown in Table 5.6 and the progressive results are shown in Table 5.8.

We also create test cases for reverse analysis that kill all proposed operators generated mutants to get 100% MS with these test cases. Then we execute existing operators generated mutants with these test cases to analyze how many faults are detected with these test cases. Table 5.7 shows the reverse analysis with the existing operators generated mutants.

TABLE 5.5: Results of execution of mutants

Case Study Name	M_e	M_n	M_{nk}	M_{ek}	T
Simplex Noise	169	51	7	169	31
Linear Map (T1)	87	5	1	87	254
Linear Map (T2)	87	5	1	87	46
Merge Sort	53	8	2	53	60
ATM Machine	41	18	9	41	30

TABLE 5.6: Summary of proposed operators Effectiveness in percentage

Case Study Name	Mutation Score (Proposed)	$Effectiveness$ in %
Simplex Noise	13.7 %	86.3 %
Linear Map (T1)	20 %	80 %
Linear Map (T2)	20 %	80 %
Merge Sort	25 %	75 %
ATM Machine	50%	50%

TABLE 5.7: Summary of Reverse analysis of mutants

Case Study Name	Mutation Score (Existing)	$Effectiveness$ in %
Simplex Noise	71 %	29 %
Linear Map (T1)	55.8 %	44.2 %
Linear Map (T2)	55.8 %	44.0 %
Merge Sort	60.3 %	39.7 %
ATM Machine	72.3 %	27.7%

Our experiments show that the existing test suite killed all non-equivalent mutants that generated with existing mutation operators and we get 100% mutation score for all three case studies. When we execute the same test suite on proposed mutation operators generated non-equivalent mutants, mutation score percentage not much high which indicates that these mutants have different faults than existing mutation operator seeded faults. These mutants require additional test cases to detect these seeded faults. Here one thing is very important if existing operators all non-equivalent mutants are not killed with the test suite, then we cannot conclude that proposed operators seeded diverse faults. So it is necessary, we have such test cases that must kill all existing operators mutants and if this test suite detects few faults that seeded with proposed operators then we can say that the rest of

TABLE 5.8: Progressive results of three case studies

Application Name	No. of TC's	MS_e %	MS_n %
Simplex Noise	6	32.5	5.8
	12	62.1	11.7
	18	87.5	13.7
	24	100	13.7
	30	100	13.7
Linear Map With T1	50	43.6	0
	100	67.8	20
	150	94.2	20
	200	100	20
	250	100	20
Linear Map With T2	8	44.80	0
	18	55.10	0
	28	85.05	20
	38	93.10	20
	46	44.80	20
Merge Sort	12	33.90	13
	24	49.05	13
	36	79.20	25
	48	100	25
	60	100	25

faults are not redundant. They are of different type and very useful faults. In our experiments, Simplex Noise problem only have 13.7% MS of proposed operators which indicates that the maximum number of faults that seeded are diverse faults and require additional test cases to kill. Other two case studies also show similar kind of percentage which endorses the usefulness of our proposed operators. The graphical representation of effectiveness in percentage of all four case studies is shown in Figure 5.1.

From reverse analysis we get high percentage of MS with existing operators generated mutants which indicate that the faults seeded by the proposed operators subsume some of the existing operators seeded faults. For Simplex Noise problem, 29% effectiveness of existing JavaScript mutation operators indicate that most of the seeded faults are redundant because test cases that detect faults in proposed mutation operators generated mutants, also detect faults in existing mutation operators generated mutants. This shows the redundancy of faults with existing mutation operators. All other cases studies show similar results which ensures the partial subsumption of existing operators' faults by proposed operators' faults. Graphical representation of reverse analysis of mutants is shown in Figure 5-2.

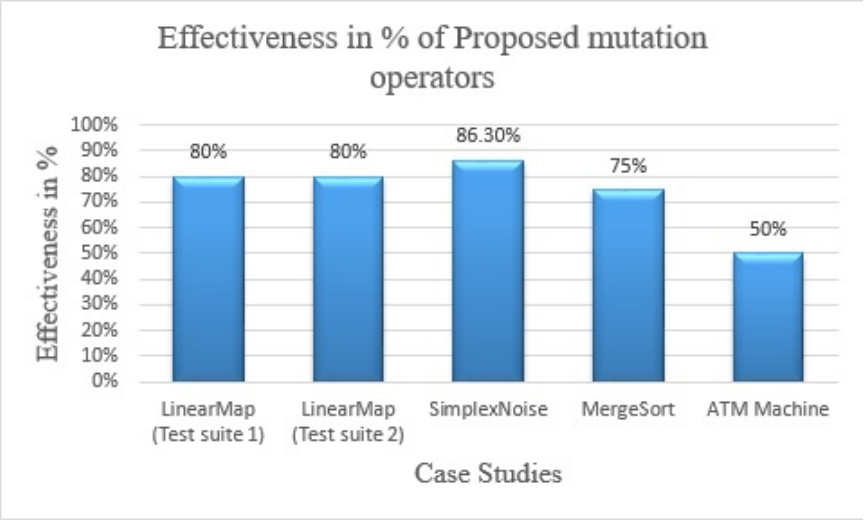


FIGURE 5.1: Graphical representation of Non Redundant faults % of all four cases studies

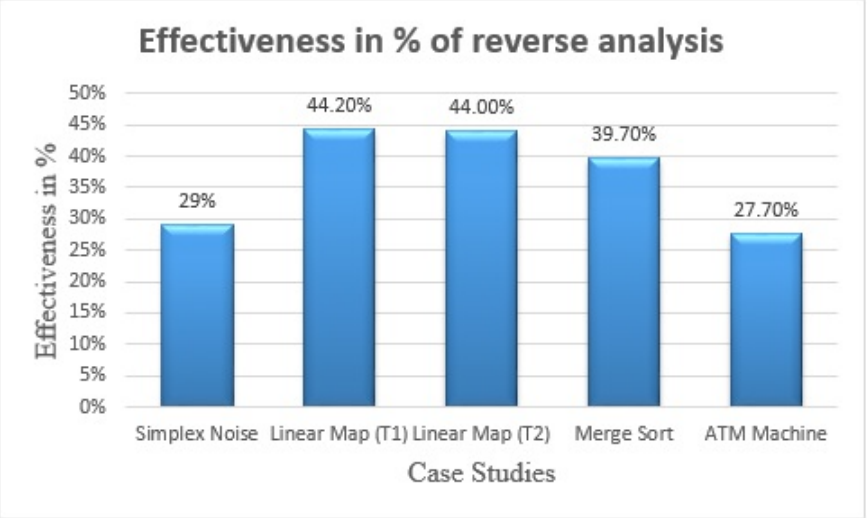


FIGURE 5.2: Graphical representation of reverse analysis of all four cases studies

In Simplex Noise problem, when we gradually increase the number of test cases, the MS percentage of existing operators also increases. After executing 24 test cases, all mutants killed and we get 100% MS percentage of all non-equivalent existing operators mutants whereas the proposed operators mutation score is only 13.6%, 6 test cases only killed 7 out of 51 non-equivalent mutants. This indicates the diversity in faults that seeded by our proposed operators. The graphical representation of Simplex Noise effectiveness of proposed operators is shown in Figure **??**. In Linear Map problem, when we gradually increase number of test cases, the existing operators MS percentage increases and we get 100% MS percentage after

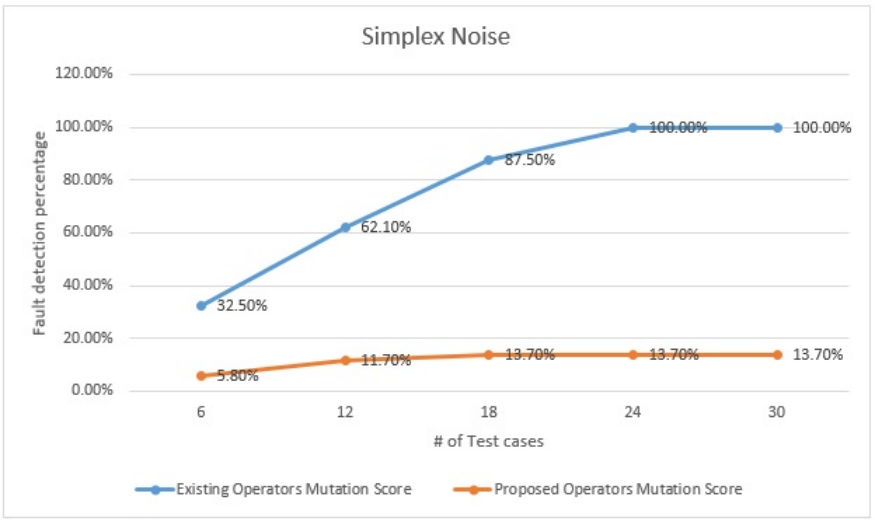


FIGURE 5.3: Graphical representation of Simplex Noise effectiveness of proposed operators

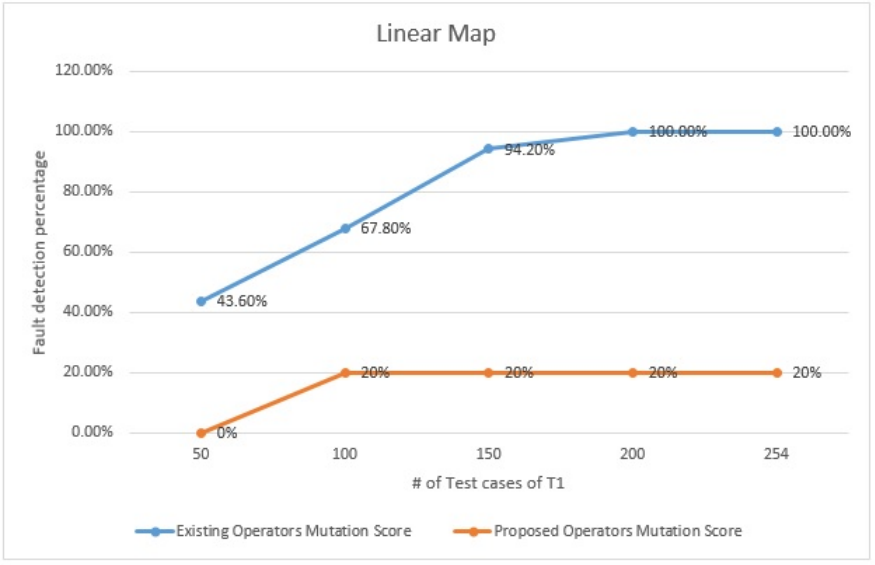


FIGURE 5.4: Graphical representation of Linear Map effectiveness of proposed operators with T1

executing 200 test cases. Whereas proposed operators MS percentage is only 20%, only two mutants killed that is generated with proposed operators when we execute first 100 test cases, after that no mutant killed with any test case. This also shows the diversity in faults that seeded by our proposed operators. The graphical representation of Linear Map effectiveness of proposed operators with test suite 1 is shown in Figure 5.4. When we evaluating Linear Map problem with test suite 2, the existing operators MS percentage rapidly increasing as we increase the number of test cases and we get maximum MS percentage that is 100% when we

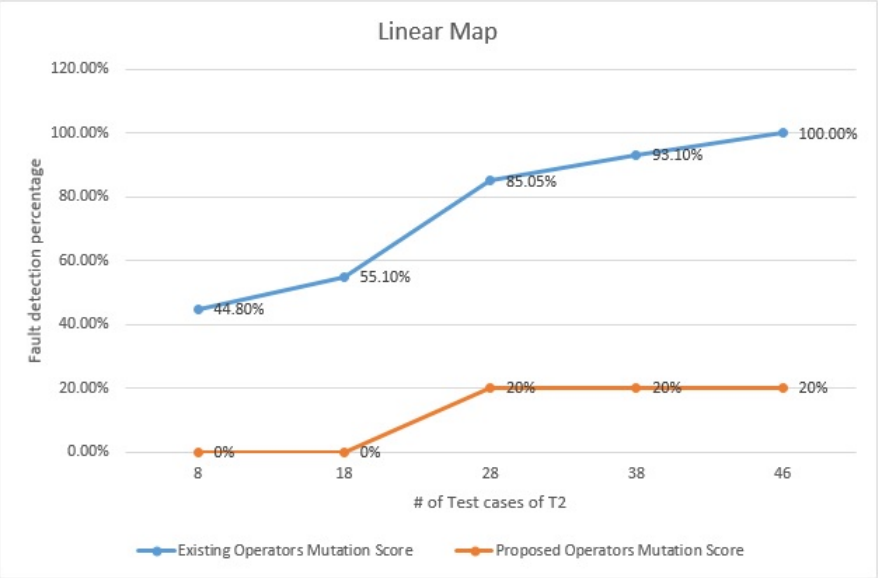


FIGURE 5.5: Graphical representation of Linear Map effectiveness of proposed operators with T2

execute all test cases of this test suite. Whereas we get 0% of proposed operators Mutation Score ratio at the execution of 18 test cases. After that only one mutant is killed by one test case and get maximum MS percentage that is 20%. This endorses that existing operators seeded faults does not subsume proposed operators seeded faults, so additional test cases require to detect these faults. The graphical representation of Linear Map effectiveness of proposed operators with test suite 2 is shown in Figure 5.5. In Merge Sort problem, when we gradually increase the number of test cases and execute first 48 test cases out of 60, the MS percentage of existing operators rapidly increasing and we get maximum MS that is 100%. Whereas the proposed operators MS ratio is only 25%, only 2 test cases killed 3 mutants and hence additional test cases require to detect faults. The graphical representation of Merge Sort effectiveness of proposed operators is shown in Figure 5.6.

By comparing MSn with MSe, it can be concluded that proposed mutation operators do not introduce faults that existing mutation operators introduce. Our proposed operators are useful that they introduce such faults that currently not introducing existing JavaScript mutation operators. For detection of such faults,

we require additional test cases. By combining our proposed operators with existing operators, assessment of test cases can be measured in a better way. Through the detailed literature survey and experimentation of different case studies, we are able to answer our research questions described in Chapter 1 as follows:
RQ. 1: To what extent the JavaScript specific features are covered by existing JavaScript mutation operators?

A number of JavaScript mutation operators have been proposed in literature that are used to introduce faults in JavaScript source program. We have seen some of the existing mutation operators cover JavaScript specific features like event, asynchronous communication, and DOM manipulation etc. we have also seen some of the operators that proposed by Mirshokraie S. et al. (2013), they cover JavaScript specific faults like removing the global search flag, removing integer based argument and set time out function etc. after detailed study we have found some faults that are not seeded by existing JavaScript mutation operators. In JavaScript, we can declare variable with two different keywords and both keywords have different scope. There are no such JavaScript mutation operator that can handle the scope of variable. JavaScript is a loosely typed programming language and datatype of a variable depends on what type of value is assigned to that variable. A variable declares and assign a specific datatype value and later on in the program this may happen the same variable can be assigned another type of value by the programmer that can cause error in the result. There are no such JavaScript mutation operators that can handle these kind of faults in the program. There are also some JavaScript specific window objects that are not currently handled by the existing JavaScript mutation operators.

RQ. 2: How effective are the proposed JavaScript mutation operators in assessing adequacy of a test suite?

In this research work, we proposed JavaScript mutation operators that are used to seed faults that are not currently seeded by existing mutation operators. By using three different case studies, we generate mutants with existing and proposed

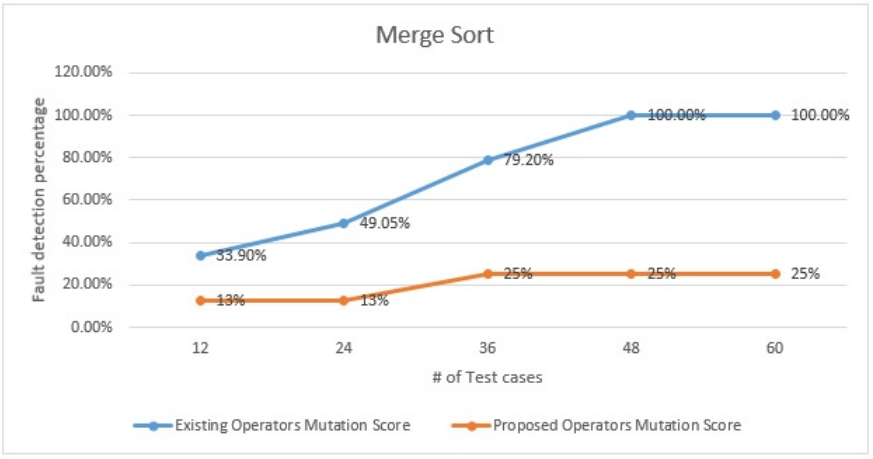


FIGURE 5.6: Graphical representation of Merge Sort Mutation Score

mutation operators. Then execute test cases on both existing and proposed operator mutants. These test cases killed all non-equivalent existing operator mutants and we have seen that killed few proposed operators mutants. The main objective of our proposed operators was to seed diverse faults and through experiments we have seen that proposed operators mutant killed ratio is very less which indicate the diversity of faults than existing operators faults. Only 10 to 25% killed ratio of proposed operators mutants indicate that additional test cases require to kill remaining mutants.

By comparing Redundancy Ratio proposed JavaScript mutation operators with existing JavaScript mutation operators, it can be concluded that proposed mutation operators does not introduces faults that existing mutation operators introduces. Our proposed operators are useful that they introduces such faults that currently not introducing by existing JavaScript mutation operators. By combining our proposed operators with existing operators, assessment of test cases can be measuring in a better way.

Chapter 6

Conclusion and Future Work

After reviewing literature, we conclude that existing JavaScript mutation operators are less effective in terms they do not cover some JavaScript specific features. To deal with this challenge, we proposed a new set of JavaScript mutation operators that cover the features that are not yet covered by existing JavaScript mutation operators. Our proposed operators introduced diverse faults that existing operators faults does not subsume proposed operators faults. We perform experiments on different case studies and the results indicate that the faults seeded by proposed mutation operators they are not redundant. Proposed operators MSn is very less, only 10 to 25% MSn of proposed operators mutants indicate that these mutants have diversity in faults than existing operators introduced faults.Through experiments on different case studies, we concluded that proposed operators are useful operators, they introduce faults that are not seeded by existing JavaScript mutation operators. By combining these operators with existing operators, the assessment of test cases will be improved.

After successful experimentation of the proposed JavaScript mutation operators, we plan to find out more advanced features of JavaScript that are not yet covered by more detailed literature survey. We also have the plan to investigate a new algorithm that reduces the computational cost (kill mutant efficiently) of proposed operators.

Bibliography

Agrawal, H., DeMillo, R., Hathaway, R., Hsu, W., Hsu, W., Krauser, E., Martin, R. J., Mathur, A. & Spafford, E. (1989*a*), Design of mutant operators for the c programming language, Technical report, Technical Report SERC-TR-41-P, Software Engineering Research Center, Department of Computer Science, Purdue University, Indiana.

Agrawal, H., DeMillo, R., Hathaway, R., Hsu, W., Hsu, W., Krauser, E., Martin, R. J., Mathur, A. & Spafford, E. (1989*b*), Design of mutant operators for the c programming language, Technical report, Technical Report SERC-TR-41-P, Software Engineering Research Center, Department of Computer Science, Purdue University, Indiana.

Ammann, P. & Offutt, J. (2016), *Introduction to software testing*, Cambridge University Press.

Batth, S. S., Vieira, E. R., Cavalli, A. & Uyar, M. Ü. (2007), Specification of timed efsm fault models in sdl, *in* 'International Conference on Formal Techniques for Networked and Distributed Systems', Springer, pp. 50–65.

Bombieri, N., Fummi, F. & Pravadelli, G. (2008), A mutation model for the systemc tlm 2.0 communication interfaces, *in* 'Proceedings of the conference on Design, automation and test in Europe', ACM, pp. 396–401.

Budd, T. A. & Angluin, D. (1982), 'Two notions of correctness and their relation to testing', *Acta Informatica* **18**(1), 31–45.

Budd, T. & Sayward, F. (1977), 'Users guide to the pilot mutation system', *Yale University, New Haven, Connecticut, Technique Report* **114**.

Chan, W., Cheung, S. & Tse, T. (2005), Fault-based testing of database application programs with conceptual data model, *in* 'Quality Software, 2005.(QSIC 2005). Fifth International Conference on', IEEE, pp. 187–196.

Chauhan, N. (2010), *Software Testing: Principles and Practices*, Oxford university press.

Chevalley, P. (2001), Applying mutation analysis for object-oriented programs using a reflective approach, *in* 'Software Engineering Conference, 2001. APSEC 2001. Eighth Asia-Pacific', IEEE, pp. 267–270.

Chevalley, P. & Thevenod-Fosse, P. (2003), 'A mutation analysis tool for java programs', *International journal on software tools for technology transfer* **5**(1), 90–103.

Delamaro, M. E., Maidonado, J. & Mathur, A. P. (2001), 'Interface mutation: An approach for integration testing', *IEEE transactions on software engineering* **27**(3), 228–247.

Delamaro, M. & Maldonado, J. C. (1999), Interface mutation: Assessing testing quality at interprocedural level, *in* 'Computer Science Society, 1999. Proceedings. SCCC'99. XIX International Conference of the Chilean', IEEE, pp. 78–86.

DeMillo, R. A., Lipton, R. J. & Sayward, F. G. (1978), 'Hints on test data selection: Help for the practicing programmer', *Computer* **11**(4), 34–41.

Derezinska, A. (2003), Object-oriented mutation to asses the quality of tests, *in* 'null', IEEE, p. 417.

Derezinska, A. & Szustek, A. (2008), Tool-supported advanced mutation approach for verification of c# programs, *in* 'Dependability of Computer Systems, 2008. DepCos-RELCOMEX'08. Third International Conference on', IEEE, pp. 261–268.

Doungsa-ard, C., Dahal, K. P., Hossain, M. A. & Suwannasart, T. (2007), 'An automatic test data generation from uml state diagram using genetic algorithm.'.

ecomfe (2016*a*), 'echart'.

URL: *https://github.com/ecomfe/echarts/blob/master/test/lib/perlin.js*

ecomfe (2016*b*), 'echart'.

URL: *https://github.com/ecomfe/echarts/blob/master/src/util/number.js*

ecomfe (2016*c*), 'echart'.

URL: *https://github.com/ecomfe/echarts/blob/master/test/ut/spec/util/number.js*

Fabbri, S. P. F., Delamaro, M. E., Maldonado, J. C. & Masiero, P. C. (1994), Mutation analysis testing for finite state machines, *in* 'Software Reliability Engineering, 1994. Proceedings., 5th International Symposium on', IEEE, pp. 220–229.

Fraser, G. & Wotawa, F. (2007), Mutant minimization for model-checker based test-case generation, *in* 'Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007', IEEE, pp. 161–168.

Jia, Y. & Harman, M. (2011), 'An analysis and survey of the development of mutation testing', *IEEE transactions on software engineering* **37**(5), 649–678.

Jing, C., Wang, Z., Shi, X., Yin, X. & Wu, J. (2008), Mutation testing of protocol messages based on extended ttcn-3, *in* 'Advanced Information Networking and Applications, 2008. AINA 2008. 22nd International Conference on', IEEE, pp. 667–674.

Last, M., Eyal, S. & Kandel, A. (2005), Effective black-box testing with genetic algorithms, *in* 'Haifa Verification Conference', Springer, pp. 134–148.

Lee, S., Bai, X. & Chen, Y. (2008), Automatic mutation testing and simulation on owl-s specified web services, *in* 'Simulation Symposium, 2008. ANSS 2008. 41st Annual', IEEE, pp. 149–156.

Lipton, R. J. (1971), 'Fault diagnosis of computer programs', *Student Report, Carnegie Mellon University* .

Ma, Y.-S., Offutt, J. & Kwon, Y.-R. (2006), Mujava: a mutation system for java, *in* 'Proceedings of the 28th international conference on Software engineering', ACM, pp. 827–830.

Mantere, T. (2003), *Automatic software testing by genetic algorithms*, Universitas Wasaensis.

Mathur, A. P. (2013), *Foundations of software testing, 2/e*, Pearson Education India.

Mirshokraie, S., Mesbah, A. & Pattabiraman, K. (2013), Efficient javascript mutation testing, *in* 'Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on', IEEE, pp. 74–83.

Mirshokraie, S., Mesbah, A. & Pattabiraman, K. (2015), 'Guided mutation testing for javascript web applications', *IEEE Transactions on Software Engineering* **41**(5), 429–444.

Nishiura, K., Maezawa, Y., Washizaki, H. & Honiden, S. (2013), Mutation analysis for javascriptweb application testing., *in* 'SEKE', pp. 159–165.

Offutt, A. J. (1994), A practical system for mutation testing: help for the common programmer, *in* 'Test Conference, 1994. Proceedings., International', IEEE, pp. 824–830.

Offutt, A. J. & Pan, J. (1997), 'Automatically detecting equivalent mutants and infeasible paths', *Software testing, verification and reliability* **7**(3), 165–192.

Offutt, A. J. & Untch, R. H. (2001), Mutation 2000: Uniting the orthogonal, *in* 'Mutation testing for the new century', Springer, pp. 34–44.

Offutt, J. (2002), 'Quality attributes of web software applications', *IEEE software* **19**(2), 25–32.

Praphamontripong, U. & Offutt, J. (2010), Applying mutation testing to web applications, *in* 'Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on', IEEE, pp. 132–141.

Rajappa, V., Biradar, A. & Panda, S. (2008), Efficient software test case generation using genetic algorithm based graph theory, *in* 'Emerging Trends in Engineering and Technology, 2008. ICETET'08. First International Conference on', IEEE, pp. 298–303.

Ribeiro, J. C. B., Rela, M. Z. & de Vega, F. F. (2008), A strategy for evaluating feasible and unfeasible test cases for the evolutionary testing of object-oriented software, *in* 'Proceedings of the 3rd international workshop on Automation of software test', ACM, pp. 85–92.

Shahriar, H. & Zulkernine, M. (2008), Mutation-based testing of format string bugs, *in* 'High Assurance Systems Engineering Symposium, 2008. HASE 2008. 11th IEEE', IEEE, pp. 229–238.

Sharma, C., Sabharwal, S. & Sibal, R. (2014), 'A survey on software testing techniques using genetic algorithm', *arXiv preprint arXiv:1411.1154* .

Sidhu, D. & Leung, T.-K. (1988), Fault coverage of protocol test methods, *in* 'INFOCOM'88. Networks: Evolution or Revolution, Proceedings. Seventh Annual Joint Conference of the IEEE Computer and Communcations Societies, IEEE', IEEE, pp. 80–85.

Srivastava, P. R. & Kim, T.-h. (2009), 'Application of genetic algorithm in software testing', *International Journal of software Engineering and its Applications* **3**(4), 87–96.

Sthamer, H.-H. (1995), The automatic generation of software test data using genetic algorithms, PhD thesis, University of Glamorgan.

Trakhtenbrot, M. (2007), New mutations for evaluation of specification and implementation levels of adequacy in testing of statecharts models, *in* 'Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007', IEEE, pp. 151–160.

Weyuker, E. J. (1982), 'On testing non-testable programs', *The Computer Journal* **25**(4), 465–470.

Woodward, M. R. (1993), 'Mutation testingits origin and evolution', *Information and Software Technology* **35**(3), 163–169.

Xu, W., Offutt, J. & Luo, J. (2005), Testing web services by xml perturbation, *in* 'Software Reliability Engineering, 2005. ISSRE 2005. 16th IEEE International Symposium on', IEEE, pp. 10–pp.

yangshun (2017), 'tech-interview-handbook'.
URL: *https://github.com/yangshun/tech-interview-handbook/blob/master/utilities/javascript/mergeSort.js*

Yoon, H., Choi, B. & Jeon, J.-O. (1998), Mutation-based inter-class testing, *in* 'Software Engineering Conference, 1998. Proceedings. 1998 Asia Pacific', IEEE, pp. 174–181.