CAPITAL UNIVERSITY OF SCIENCE AND TECHNOLOGY, ISLAMABAD

Mutation Testing for Unity 3D

by

Omaid Ghayyur

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

Mutation Testing for Unity 3D

by

Omaid Ghayyur

MCS163026

THESIS EXAMINING COMMITTEE

S. No.	Examiner	Name	Organization
(a)	External Examiner	Dr. Muhammad Uzair Khan	FAST, Islamabad
(b)	Internal Examiner	Dr. Arshad Islam	CUST, Islamabad
(c)	Supervisor	Dr. Aamir Nadeem	CUST, Islamabad

Dr. Aamir Nadeem
Thesis Supervisor
October, 2018

Dr. Nayyer Masood	Dr. Muhammad Abdul Qadir
Head	Dean
Dept. of Computer Science	Faculty of Computing
October, 2018	October, 2018

Author's Declaration

I, **Omaid Ghayyur** hereby state that my MS thesis titled "**Mutation Testing for Unity 3D**" 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.

Omaid Ghayyur

Registration No: MCS163026

Plagiarism Undertaking

I solemnly declare that research work presented in this thesis titled "**Mutation Testing for Unity 3D**" 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.

Omaid Ghayyur

Registration No: MCS163026

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

Omaid Ghayyur

Registration No: MCS163026

Abstract

To verify the correctness of a software and to check the requirements fulfilled, software testing is performed. The effective technique use to check the adequacy of test suite is known as Mutation testing. In mutation testing, multiple variants known as mutants of programs are created using a set of defined mutation operators from the original program. Test cases are run on mutants to verify if the changes are detected or not. Each mutant is executed for each test case to identify a change in the mutant. Mutant is said to be killed if the change is detected by test case otherwise it is considered as alive. The effectiveness of test suite is calculated from the number of mutants killed. The mutation score of test suite is measured as the ratio of number of the mutants killed to the total number of mutants. Research has been done on the mutation testing of the JAVA and other programming languages. A very little work is being done for the mutation testing of the mobile applications and programming languages used for mobile application development. Recently, work is done on the mutation testing of the ANDROID programming language. The focus of our research is on Unity 3D mutation testing by proposing mutation operators of the Unity 3D C# programming language used for the mobile game development. Nowadays, Unity 3D C# programming language is most commonly used for mobile games development. Unity 3D C# mutation operators will be used to seed the faults in Unity 3D C# games source scripts containing the special programming features that are not covered by traditional C# mutation operators.

In our work, we have proposed a set of Unity 3D C# mutation operators to address special programming features of Unity 3D C# programming language used for the mobile game development. We have implemented a simple JAVA tool, which is used to generate the mutants of Unity 3D C# source scripts with the proposed mutation operators. Test cases are executed for each mutant during experimentation. Evaluation of the proposed Unity 3D C# mutation operators is performed with traditional C# programming mutation operators using mutation score. Based on mutation score, it is concluded that the faults seeded using new proposed Unity 3D C# mutation operators are not detected by the traditional mutation operators as strong mutants are generated with proposed mutation operators. New additional test cases with strong coverage criterion are required to detect the Unity 3D specific faults seeded with proposed mutation operators.

Contents

List of Figures

List of Tables

List of Abbreviations

3D	3 Dimensional
GUI	Graphical user interface
I/O	Input Output
API	Application programming interface
iOS	iPhone operating system
2D	2 Dimensional
A/I	Activities and intents
A/P	Android Programming
BES	Back end service
C	Connectivity
D	Data/Object parsing and Format
DB	Database
GP	General programming
NFR	Non-functional requirements
VR	Virtual reality
AR	Augmented reality
XML	Extensible Markup Language
UI	User interface
AI	Artificial intelligence
VFX	Visual effects
ED	Unity editor
UP	Unity programming
IDE	Integrated development environment
HP	Hewlett-Packard

DDR	Double data rate
AMD	Advanced micro devices

Chapter 1

Introduction

1.1 Overview

In this current era of technology, demand of the software development has increased due to the large dependency on the computers to perform the multiple tasks. Dependency on the different types of software to perform critical tasks has increased which include the systems like health care in which the monitoring of the patients is performed automatically using the machines controlled by the software. The use of machines for such critical and important tasks requires proper testing of the software as the task performed is crucial to guide the doctors about the patient condition at a certain time. Similarly, with the excessive use of the mobile devices and internet everyone can access their bank accounts and can perform multiple activities on their bank account which majorly include the transactions of multiple types. Such systems require proper testing of the software including the security testing to protect the confidential data of the users.

1.2 Software Testing

The process of system evaluation to check if its meets its original specified requirements or not is known as software testing. The process of testing mainly consists of validation and verification of the software system. The process of the finding the bugs, errors or missing requirements of the developed system and software is known as software testing. Software testing is used to know about the quality of the product [1]. Software testing or software quality assurance importance can be considered from the life critical systems like human kidney dialysis system, flight engine control system [2].

Using software testing the reliability of the software is ensured as the work is being performed by the software according to its requirements or not [3]. During the software testing large amount of time and a lot of resources are consumed to ensure the quality of the software [4]. Over 60% of the budget and half of the resources are consumed in the software testing phase [5]. The testing of the software after the development is an important phase, and work is being done to make the process of the software testing cost effective. Black box and white box testing are the basic testing techniques. Black box testing is also referred as functional testing while white box testing is referred as structural testing. In the black box or functional testing, the software functionality testing is performed while in the white box or structural testing, the code testing of the software is performed [6]. Gray box testing is referred as the combination of the black and white box testing. The above testing techniques do not check the test cases effectiveness and adequacy as these techniques are coverage based testing techniques [7].

Testing of the software can be performed at the unit level, integration level or system level. The testing process is performed by the quality assurance engineer or software tester [8]. The process of testing each developed module of the software independently is known as unit testing. While integrating the single modules of the system integration testing is performed as the errors or bugs can arise during the integration process. Testing of the complete software after development is known as system level testing [9].

1.3 Mutation Testing

Mutation testing is a code-based testing technique in which faults are introduced in the original program to measure the test suite effectiveness to detect those faults [10]. Using the technique of the mutation testing, faults are introduced in the original program to create the mutants using a set of defined mutation operators. **Mutation operators** are defined as the operators used to introduce a single change in each version of the original program. **Mutant** is defined as the program which is generated as a result of the fault introduced in the original program. Test cases are executed on the original and mutated programs with the goal that different output will be observed on the test case execution for the mutant as compared to the original program. Mutant is referred to be **killed** if the output for the mutant is different from the original program. The mutant is said to be **live** if it is not killed from the existing test cases. If the mutant is not killable by any test case then it is called an **equivalent** mutant. Equivalent mutants are syntactically different but semantically equivalent to the original program and for this reason are not killed by any test case. Equivalent mutants cannot be detected automatically as it is an undecidable problem [11]. Some of the mutants do not compile and are known as **still born**. In the still born mutants the change performed by the mutation operator makes the mutant syntactically incorrect. Still born mutant can be avoided by properly defining the mutation operator [12]. Mutation score is calculated by executing all the mutants with the test cases which indicates the quality of the test suite. The ratio of the number of mutants killed by the total number of mutants except for the equivalent mutants is referred as **mutation score** [13]. Main purpose of the mutation testing is to raise the score to indicate the test suite efficiency which makes it sufficient to detect faults.

Mutation testing is used for two purposes:

1. It is used to either check the fault detection effectiveness of a test suite by calculating the mutation score by the mutants killed

2. Or to guide test case generation to kill the mutants

Commonly it is used to check the adequacy but can also be used to generate test cases to improve the software quality. Figure 1.1 shows the general process of mutation testing.

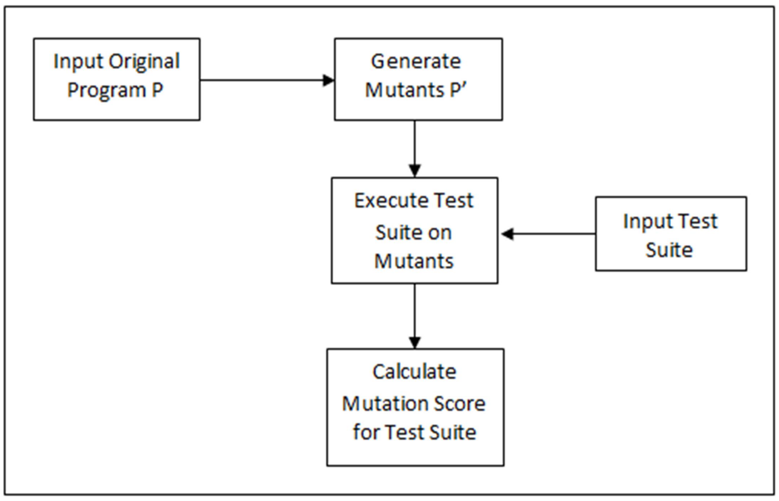

Input Original Program P
Generate Mutants P'
Execute Test Suite on Mutants
Input Test Suite
Calculate Mutation Score for Test Suite

Figure 1.1: General Process of Mutation Testing

New test suites are designed and the quality of the existing test suites is evaluated using mutation testing. Mutation testing can be applied to test the software at the unit level and integration level which makes it a white box testing technique [14]. Mutation testing helps the testing resource to generate effective test suites. Mutation testing is not used to test the software directly, rather tests the test suite of the software and helps to improve test cases to increase effectiveness of the test suite. It is assumed that the test suite which detects more faults will also detects potential faults of the software which in turns helps in the improvement of software quality. The cost of mutation testing is high in terms of the mutants generation, mutants compilation and execution with the test suite and identification of the equivalent mutants and mutants analysis [15].

History of the mutation testing comes from way back as its being studied and used from long time ago from 1970s. Many surveys related to mutation testing have been conducted, and the first survey of mutation testing was performed in 1979 [16]. Survey on very specific sub area of weak, strong and firm mutation testing approaches is performed in 1988 [17]. Introductory chapters on mutation testing are included in the books by multiple authors [18] [19].

We can use mutation testing for testing software at a unit level and integration level [20] [21]. Mutation testing has been applied to many languages, including Fortran 77 [22] [23], C [24], Java [25] [26], Javascript [27], AspectJ [28], and web applications [29]. Several papers also extend mutation analysis to model-level, such as Finite State Machines [30] [31], statecharts [32], Petri nets [33], timed automata [34], and Aspect-oriented models [35]. For GUI-based applications, a specific set of mutation operators [36] are also proposed.

Mutation testing has certain drawbacks even though it is a very effective technique; main drawback of the technique is that it has a high computational cost for executing enormous number of mutants against a test suite. Issue of equivalent mutants also exists in mutation testing which cannot be killed [35]. Mutation testing also involves a lot of human effort which refers to the human oracle problem [36]. Oracle problem refer to a process of comparing outputs of the original and mutated program during test case execution. It is most expensive part of the testing activity. To overcome these issues of the mutation testing is impossible but various studies have been done minimize and reduce the computation cost of mutation testing [37].

1.4 Mutation Testing of Mobile Applications

So far, little work has been done by researchers in the area of mobile applications mutation testing which included only the Android applications mutation testing in recent years. Android is the most popular mobile operating system. Android mobile applications consist of specific characteristics which are not available for simple JAVA or other language applications. The specific characteristics of android applications are GUI-centric design and interaction, event driven programming, inter processes communication, interaction with backend and local services, permission mechanism, software development kit version compatibility. These special Android programming features can lead to application failure. Deng et al., (2017) proposed eleven mutation operators for Android applications by applying on new Android programming features. Mutation operators are proposed for Android apps elements comprising of intent, activity lifecycle, event handler and XML [38]. Following the work of Deng et al., (2017) Linares-Vásquez et al., (2017) worked on the Android applications mutation operators and proposed 38 new mutation operators according to the faults taxonomy defined by them for Android applications [12]. The faults taxonomy consists of Activities and Intents, Back-end Services, Collections and Strings, Data/Objects Parsing and Format, Threading, Android Programming, Non-Functional Requirements, GUI, Input Output (I/O), Device or Emulator, API and Libraries, Connectivity, Database and General Programming.

As Unity 3D supports cross platform development and games developed can easily packaged for deployment on the devices including Android and iOS. Games development in Unity game engine is different as compared to the application development. In Unity game development Unity Editor is used for scene creation which contains the game environment, game objects and GUI while the scripting is done on Mono Developer which is a Unity scripting tool. The scripts for the game are attached with the scenes, game objects, GUI from Unity editor. The game development in Unity uses special programming features and implementation characteristics as games are actually event based programs with GUI design and interaction with multiple back end processes to handle event system and input is given by the users. The main features of the game development are scenes creation in Unity editor, game objects initialization, physics implementation, animations, handling input touch events, game logic, scenes rendering, GUI rendering, coroutines, lifecycle methods handling, decommissioning, backend and local services, permissions, software development kit version compatibility. Failure may occur because of incorrect use of these special features used for game development. Unity 3D supports the game scripting in 3 different programming languages which are C#, JavaScript and Boo. The most commonly used language is C# used by the developers for Unity 3D. The traditional mutation operators are not sufficient to cover these special programming features of Unity 3D game development. Some examples of the Unity 3D features and faults that are not covered by the existing mutation operators are as follows:

Example 1: Unity 3D Physics, Life Cycle Event use Code Snippet

The code is used to move a rigid body game object in an environment in forward direction.

```
public class ExampleClass : MonoBehaviour
{
  public Vector3 point;
  public Rigidbody body;
  void Start()
  {
     body = GetComponent<Rigidbody>();
  }
  void FixedUpdate()
  {
     body.MovePosition(transform.position + transform.forward *
     Time.deltaTime);
  }
}
```

The above Unity code uses two different Unity 3D lifecycle functions that are Start() and FixedUpdate(), if we replace these lifecycle functions with other life-cycle functions like replace Start() with Awake() and FixedUpdate() with Update() and LateUpdate() functions, the behavior of the game will be affected due to difference in the execution sequence of the lifecycle functions.

Example 2: Unity 3D Coroutine Use for Time Delay Code Snippet

Coroutines are used to create time delay in the code execution. In the example below coroutines are used in multiple ways to display the text on Unity editor after the delay of few seconds.

```
public class ExampleClass : MonoBehaviour
{
  private IEnumerator coroutine;
  void Start()
  {
     print("Starting " + Time.time + " seconds");
     coroutine = WaitAndPrint(5.0f);
     StartCoroutine(coroutine);
     print("Coroutine started");
  }
  private IEnumerator WaitAndPrint(float waitTime)
  {
     yield return new WaitForSeconds(waitTime);
         print("Coroutine ended: " + Time.time + "seconds");
  }
}
```

The above Unity code uses Coroutine programming feature of Unity 3D to display the text, if the parameter value of StartCorotuine() method is changed, the behavior of game will be affected.

Example 3: Unity 3D Game Object Collision Detection Code Snippet

Code initiate the explosion prefab when the game object collide the surface and after explosion game object get destroyed.

```
public class ExampleClass : MonoBehaviour
{
  public Transform explosionPrefab;
  void OnCollisionEnter(Collision collision)
  {
     ContactPoint contact = collision.contacts[0];
     Quaternion rot = Quaternion.FromToRotation(Vector3.up, contact.normal
     Vector3 pos = contact.point;

     Instantiate(explosionPrefab, pos, rot);
     Destroy(gameObject);
  }
}
```

The above Unity code uses OnCollisionEnter() to check the collision of game object with other objects in game environment, by replacing OnCollisionEnter() function with OnTriggerEnter(), collision of the objects will not be detected which will affect the game behavior and may cause the run time error.

1.5 Problem Statement of Thesis

Unity 3D games which are developed for the mobile devices involve several new programming features and a very little information is available to test them which results in an ineffective testing. Unity 3D platform supports 3D and 2D games development for the cross platforms and special programming features which are used are game objects, physics implementation, handling input touch events, game logic, scenes rendering, GUI rendering, coroutines, lifecycle methods, decommissioning, animations, backend and local services, permissions, software development kit version compatibility. These special programming features require the mutation operators for the evaluation of the test suite adequacy as these features are not covered by the existing mutation operators. No mutation operators are defined for the Unity 3D special programming features which can measure the test suite effectiveness. So, to assess the effectiveness of the test cases, a need of novel and effective Unity 3D mutation operators with the guide of the test case generation to kill mutants is required.

1.6 Research Questions

In this research work, we will propose the mutation operators for the Unity 3D C# programming language special features to increase the effectiveness of the test suite. However, the following questions must be taken into account:

RQ. 1: *Can we use the traditional C# programming language and Android mutation operators for Unity 3D?*

To answer this question, the special programming features used for the Unity 3D C# are studied and explained in the proposed solution which are not covered by the traditional C# and Android mutation operators.

RQ. 2.1: *What are the special features of the Unity 3D C# programming language which are not covered by the traditional C# and Android mutation operators?*

To answer this research question, the game development process is defined in the proposed solution with the special programming features and Unity implementation characteristics used for game development using C# language.

RQ. 2.2: *How to design new mutation operators for the Unity 3D C# special programming features?*

To answer this research question, in the proposed solution the special programming features of the Unity 3D C# are explained. Due to incorrect use of those explained special programming features faults may occur. Base on the faults that can be introduced, the new mutation operators for Unity 3D C# special programming features are designed.

RQ. 3: *How effective are the new proposed mutation operators?*

To answer this question, a number of experiments are performed on different case studies with their respective test suites and mutation score ratio is calculated for proposed operators and comparison is done with the traditional C# mutation operators.

Focus of this research is to address the aforementioned research questions.

1.7 Research Objectives

The objective of this thesis is to propose a novel set of Unity 3D C# programming language mutation operators that can allow developers to find faults in Unity 3D C# games before release, especially in the parts of code that use new programming features.

1.8 Research Contribution

Research contribution of this thesis is to propose set of Unity 3D C# special programming features mutation operators that could be used to introduce faults in Unity 3D 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. Unity 3D mutation operators are proposed for the first time that introduces the diverse faults that are currently not seeded by the traditional existing mutation operators of the C# language.

1.9 Thesis Organization

Rest of the thesis is organized as follows: second chapter presents the literature review and the related work. 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

In the software testing, mutation testing is considered as the most effective technique to check the effectiveness of the test suite. Mutants are generated from the original program by introducing a single fault in each mutant. The faults are minor syntactic change in the original program which is introduced using the different mutation operators. Test cases are executed on the mutants to identify the faults introduced. Mutants are executed against each test case to get the different output from the original program. This chapter contains an overview of mutation testing done for Android applications and the traditional C# mutation operators. Purpose of the work is to identify the mutation operators for the special programming features used for the development of mobile applications.

2.1 Mobile Applications Mutation Testing

As mobile application development is a new emerging area due to which very little mutation testing work is done by the researchers for mobile development platforms. Work is done recently for mutation testing for the Android applications only by defining mutation operators for the new programming features used for the Android JAVA programming [12] [39]. As mobile application development programming languages uses new and unique programming features due to different operating system which run over the mobile hardware, so not much work on the mutation operators to target those special features is yet done and the domain is free to be explored. C# programming language is basically used for web application development but it is also being used extensively for the Unity 3D game development with the special new features for game programming. Mutation operators for the C# programming languages are proposed which are same as the traditional mutation operators of JAVA and with some mutation operators for advance features of C# programming language [40].

2.1.1 Android Applications Mutation Testing

Deng et al., (2017) proposed the 11 mutation operators for Android applications which are not tested using existing mutation operators of the JAVA languague due to which the test results are weak and ineffective. At first, the unique technical Android programming features are analyzed and later novel mutation operators are designed for those features. They proposed the novel mutation operators specific for the Android applications and implemented the proof-of-concept mutation analysis tool for the implementation of the new Android mutation operators along with the traditional known mutation operators. Later, they evaluated the mutation operators on the eight Android applications. Empirical studies are done for the application for results collection and thorough analysis. Comparison was done with the mutation operators of the muJava which generates the mutants with the traditional mutation operators.

Table 2.1 shows the mutation operators proposed for the Android special programming features with description.

Table 2.1: Android programming features mutation operators

Android Features	**Cat**	**Mutation Operator**	**Mutation Operator Description**
Intent Mutation Operators	A/I	Intent Payload Replacement (IPR)	Replace the second parameter of the intent function to its default value
	A/I	Intent Target Replacement (ITR)	Replace the target of each intent with all possible classes
Activity Lifecycle Mutation Operator	A/I	Lifecycle Method Deletion (MDL)	Deletes each overriding method to force Android to call the version in super class
Event Handler Mutation Operators	GUI	OnClick Event Replacement (ECR)	Replace each handler with every other compatible handler
	GUI	OnTouch Event Replacement (ETR)	Replaces the event handler for each OnTouch event
XML Mutation Operators	AP	Activity Permission Deletion (APD)	Deletes and app's permission for its Android Manifest.xml file one at a time
	AP	Button Widget Deletion (BWD)	BWD deletes one button at a time from the XML: layout file of the UI
	AP	EditText Widget Deletion (TWD)	TWD mutation operators remove each EditText widget at a time

XML Mutation Operators	AP	Button Widget Switch (BWS)	BWS switches the location of the two buttons on the same screen
Mutation Operators Based on Common Faults	AP	Fail on Null (FON)	FON add a “Fail on Null” statement before each object is referenced
	AP	Orientation Lock (ORL)	ORL mutants freeze the orientation of any activity by inserting special locking statement in source code

Linares-Vásquez et al., (2017) extended the work of Deng et al., (2017) by defining the more mutation operators for the Android application after the brief study of the bugs that occurs during the Android development. They collected the large data of the Android applications from the GitHub repository and used Stack Overflow for the faults and bugs study during Android development. By analyzing the total of 1,623 documents they made the faults taxonomy of JAVA and Android bugs for the categorization of mutation operators which were later proposed. In their work, they proposed total of 38 mutation operators for the Android programming features. For the proposed mutation operators MDroid+ tool was developed to implement mutation operators for analysis with the basic JAVA mutation testing tools. The tool was executed for the total 68 applications. Mutation analysis results of MDroid+ tool was compared with the traditional Java mutation tools.

Table 2.2 shows the proposed mutation operators along with the defined category and description for the Android applications.

Table 2.2: Android programming features mutation operators with categories

Mutation Operators	**Cat**	**Description**
Activity Not Defined	A/I	Delete an activity <Android:name= “Activity”/>entry in the Manifest file

Different Activity Intent Definition	A/I	Replace the Activityċlass argument in an Intent instantiation
Invalid Activity Name	A/I	Randomly insert typos in the path of an activity defined in the Manifest file
Invalid Key Intent Put Extra	A/I	Randomly generate a different key in an Intentṗut Extra(key, value) call
Invalid Label	A/I	Replace the attribute “Android:label” in the Manifest file with a random string
Null Intent	A/I	Replace an Intent instantiation with null
Null Value Intent Put Extra	A/I	Replace the value argument in an Intent.putExtra(key, value) call with new Parcelable[0]
Wrong Main Activity	A/I	Randomly replace the main activity definition with a different activity
Missing Permission Manifest	AP	Select and remove an <uses-permission /> entry in the Manifest file
Not Parcelable	AP	Select a parcelable class, remove “ implements Parcelable” and the @override annotations
Null GPS Location	AP	Inject a Null GPS location in the location services
SDK Version	AP	Randomly mutate the integer values in the SdkVersion-related attributes
Wrong String Resource	AP	Select a <string />entry in /res/values/strings.xml file and mutate the string value
Null Back End Service Return	BES	Assign null to a response variable from a back-end service
Bluetooth Adapter Always Enabled	C	Replace a BluetoothAdapter.isEnabled() call with“true”

Null Bluetooth Adapter	C	Replace a BluetoothAdapter instance with null
Invalid URI	D	If URIs are used internally, randomly mutate the URIs
Closing Null Cursor	DB	Assign a cursor to null before it is closed
Invalid Index Query Parameter	DB	Randomly modify indexes/order of query parameters
Invalid SQL Query	DB	Randomly mutate a SQL query
Invalid Date	GP	Set a random Date to a date object
Invalid Method Call Argument	GP	Randomly mutate a method call argument of a basic type
Not Serializable	GP	Select a serializable class, remove “implements Serializable”
Null Method Call Argument*	GP	Randomly set null to a method call argument
Buggy GUI Listener	GUI	Delete action implemented in a GUI listener
Find View By Id Returns Null	GUI	Assign a variable (returned by Activity.findViewById) to null
Invalid Color	GUI	Randomly change colors in layout files
Invalid ID Find View	GUI	Replace the id argument in an Activitity.findViewById call
Invalid View Focus	GUI	Randomly focus a GUI component
View Component Not Visible	GUI	Set visible attribute (from a View) to false
Invalid File Path	I/O	Randomly mutate paths to files
Null Input Stream	I/O	Assign an input stream (For Example reader) to null before it is closed
Null Output Stream	I/O	Assign an output stream (For Example, writer) to null before it is closed
Lengthy Back End	NFR	Inject large delay right-after

Service		a call to a back-end service
Lengthy GUI Creation	NFR	Insert a long delay (that is, Thread.sleep(..)) in the GUI creation thread
Lengthy GUI Listener	NFR	Insert a long delay (that is, Thread.sleep(..)) in the GUI listener thread
Long Connection Time Out	NFR	Increase the time-out of connections to back-end services
OOM Large Image	NFR	Increase the size of bitmaps by explicitly setting large dimensions

2.2 C# Mutation Advance Mutation Operators

Derezinska A., (2006) proposed mutation operators for C# programming language, the advance mutation operators applicable for the C# programs are proposed in their work along with the traditional mutation operators for the C# programming languages. In their work the relation for the JAVA and C# mutation operators were also defined and indicated. The object-oriented operators adopted for the C# and the advance operators with the new programming features were studied.

Table 2.3 shows the traditional and advance C# mutation operators along with the description used for C# mutation testing.

Table 2.3: C# Traditional and Advance Mutation Operators

Mutation Operators	**Description**
AMC	Access modifier change
IHD	Hiding variable deletion
IHI	Hiding variable insertion
IOD	Overriding method deletion
IOP	Overridden method calling position change
IOR	Overridden method rename

ISK	Base keyword deletion
IPC	Explicit call of a parent's constructor deletion
PNC	New method call with child class type
PMD	Member variable declaration with parent class type
PPD	Parameter variable declaration with child class type
PRV	Reference assignment with other compatible type
OMR	Overloading method contents change
OMD	Overloading method deletion
OAO	Argument order change
OAN	Argument number change
JTD	This keyword deletion
JSC	Static modifier change
JID	Member variable initialization deletion
JDC	C#-supported default constructor create
EOA	Reference assignment and content assignment replacement
EOC	Reference comparison and content comparison replacement
EAM	Accessor method change
EMM	Modifier method change
MNC	Method name change
MBC	Member changed
MCO	Member call from another object
MCI	Member call from another inherited class
RFI	Referencing fault insertion
EHR	Exception handler removal
EHC	Exception handling change
DMC	Delegated method change
DMO	Delegated method order change
DEH	Method delegated for event handling change
PRM	Property replacement with member field
IOK	Override keyword substitution

OPD	Overriding property deletion
OID	Overriding indexer deletion
NDC	Namespace declaration change

2.3 Critical Analysis

Although, in literature, many C# mutation operators are defined that covers some tradition and advance features, some generic mutation operators and some specific to the C# language features. But all of these operators are not sufficient to test the adequacy of a test suite for Unity 3D C# special programming features like the one used for the Android JAVA for which the new mutation operators are proposed in the recent work by the researchers. The C# mutation operators defined are for C#. NET framework and do not covers the special programming features of Unity 3D game development engine using C# programming language.

2.4 Gap Analysis

With the emerging field of the mobile application development along with the increase in the users of mobile devices, the need of mutation testing is important to check the effectiveness of mobile applications which will in turn improve the quality of mobile softwares.

Mobile games are the video games played on feature phone, smartphones, tablets or smart watch. Multiple game engines are used for the development of 3D and 2D mobile games. Unity 3D is an ultimate cross platform game engine developed by Unity Technologies which is primarily used to develop both 3D and 2D games deployed on mobile (Android, iOS, Tizen, Microsoft), desktop, VR/AR, consoles or web. Unity 3D game engine is widely used in the mobile games development now a day for Android and iOS platforms. Unity games are built differently from the traditional software and use new structures, new control, and data connections.

Unity 3D game development involves several new programming features and framework for game designing and development. Very little knowledge is available to test them as a result of this weak and ineffective testing is done. Unity 3D supports the game programming in 3 different languages which are C#, JavaScript and Boo. The most commonly used language is C# used by the developers for Unity 3D. So, there is need of new Unity 3D C# mutation operators to generate mutants to seed the faults using the mutation operators in the original source code to generate more effective test suite to kill the mutants for Unity 3D C# special programming features.

Chapter 3

Proposed Solution

In the software testing domain mutation testing is an effective testing technique which either (1) check the fault detection effectiveness of a test suite by calculating the mutation score by the mutants killed or (2) to guide test case generation to kill mutants. Researchers have done work on the Android mobile applications mutation testing in recent years, the proposed mutation testing of Android special programming features have been discussed in the chapter 2 in detail and tables 2.1, 2.2 provide the list of all the Android mutation operators. Work done by researchers for the C# programming language is also mentioned in chapter 2 and the traditional along with the advance mutation operators list of C# programming language are mentioned in the table 2.3. We have seen in the previous chapter that so far no mutation operators exists to cover the Unity 3D C# special programming features as compared to the Android programming features. There existing C# mutation operators are not sufficient to assess the effectiveness of the test suite for the Unity 3D C# games. We have proposed new Unity 3D C# mutation operators to improve the assessment of test cases by considering the faults associated with the Unity 3D C# scripts.

Detailed methodology steps of the proposed solution are as follows:

1. In the initial step we have studied and discussed the game development process for the 2D and 3D games with Unity 3D game engine.

2. Special programming features of the Unity 3D C# language used for the game development are identified.

3. The Unity 3D features are categorized for selection to propose the mutation operators.

4. Special programming features are selected from multiple categories of Unity 3D to generate the mutation operators.

5. To cover the special programming features of Unity 3D mutation operators for mutation testing are proposed.

6. The operators are defined such that rate of the still born mutants is minimum.

Methodology steps our proposed approach is depicted in Figure 3.1.

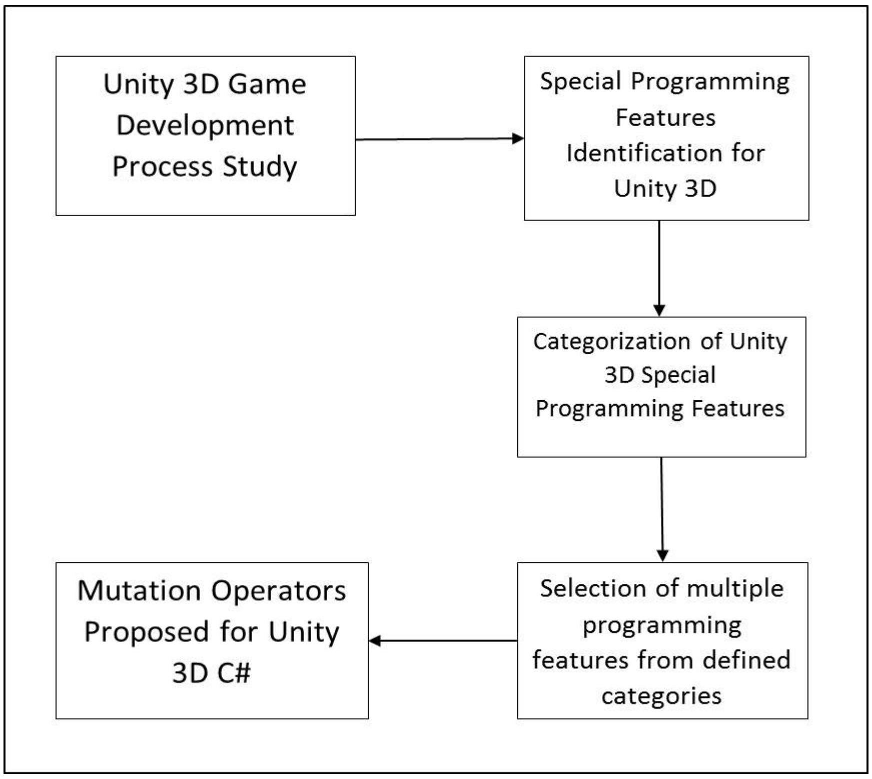

Unity 3D Game Development Process Study
Special Programming Features Identification for Unity 3D
Categorization of Unity 3D Special Programming Features
Selection of multiple programming features from defined categories
Mutation Operators Proposed for Unity 3D C#

FIGURE 3.1: Proposed Solution Detailed Methodology

In the next sections, we have discussed the general process of the 2D and 3D game development using Unity 3D game engine and later have proposed a set of

Unity 3D C# mutation operators that will introduce diverse faults that existing tradition/advance C# mutation operator are unable to introduce. In the later sections, we have identified Unity 3D special programming features, defined features category and have proposed Unity 3D C# mutation operators.

3.1 Game Development Process

Unity 3D game engine is used for the development of both 2D and 3D games. Two main essential components of the Unity 3D used for the game development are Unity Editor and Unity scripting tool. The game development process in Unity consists of several steps. The main life cycle of the game development consists of following steps:

1. Game Conceptualization is the initial step before the game development. In the process of game conceptualization following main tasks are performed:
   (a) Game idea preparation
   (b) Game play strategy designing
   (c) Assets collection or preparation including characters, environment
   (d) UI theme preparation to be used in the game
   (e) Story board preparation for the game to interact with the user for maximum time
2. Game design step is followed by game conceptualization in which the main components of game are designed and prepared for user interaction visually on mobile devices. For game designing Unity 3D Editor is used. In the game design process following tasks are performed:
   (a) Game user interface and user interface components designing
   (b) Game characters designing with features for user interaction
   (c) Game levels designing

   (d) Game 3D environment or 2D platform designing
   (e) User camera and camera projection setup for game along with game lightening
   (f) Preparing game extra assets including sounds and music
   (g) Graphics and visual effects preparation

3. Game programming is performed after the game designing process in which the core functionality of game is coded and linked with the game objects including environment, characters, UI. Game programming is done using Unity 3D scripting. The main steps of game programming process are:
   (a) Game main functionality and logic scripting
   (b) Enemy and Player AI scripting
   (c) Data storage and Data Management
   (d) Game Controller and Game Manager scripting
   (e) UI handling
   (f) Third party modules integration

4. Game testing is most important and essential process of game development life cycle which ensures the game quality and checks the game requirements. The main testing of the games consists of following steps:
   (a) Test cases generation
   (b) Functionality and Compliance testing
   (c) Compatibility and Performance testing
   (d) Game Memory testing
   (e) Bug Reports Management

5. After complete game development and testing, game publishing is performed on multiple stores after the game executable file is prepared.

6. Sales and marketing of the game is performed after the game release which include game advertisement, short videos, social media marketing.

Figure 3.2 illustrates the game development process:

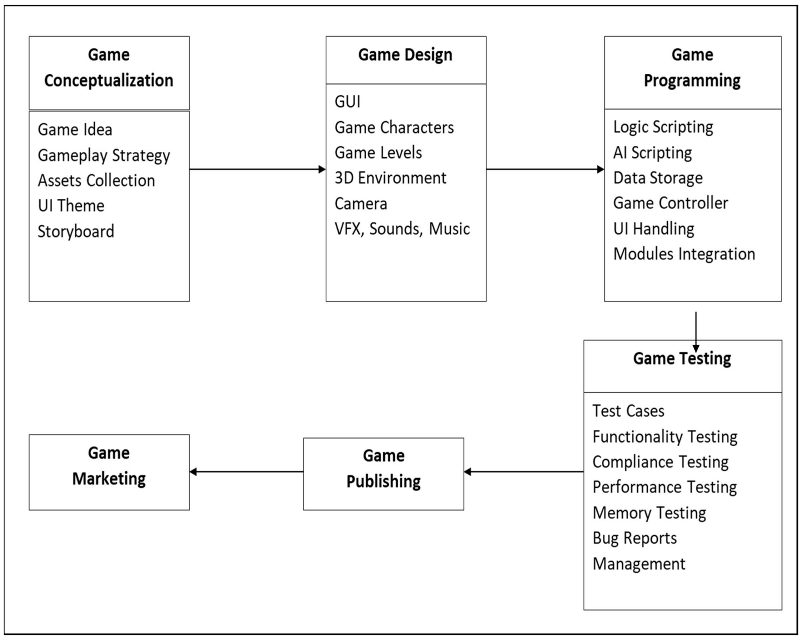


FIGURE 3.2: Game Development Process

3.2 Unity 3D Special Programming Features

Unity 3D game engine have special characteristics that are used for 3D and 2D mobile game development along with the special features to code multiple games functionalities. List of major special features used for Unity 3D programming along with the description are as follows:

3.2.1 Rigid Bodies

Used to apply Unity 3D physics behavior on any game object. Rigid body class is used to set force and controls of game object. Example of adding force to a game object using a rigid body is as follows:

```
public class ExampleClass : MonoBehaviour
{
    public float thrust;
    public Rigidbody body;
    void Start()
    {
        body = GetComponent<Rigidbody>();
    }
    void Move()
    {
        body.AddForce(transform.forward * 5.0f);
    }
}
```

3.2.2 Coroutines

It’s a special feature of Unity 3D coding which can pause the execution of program for certain time frame until instructions of yield are executed completely and resume the normal execution from where normal execution was paused.

```
public class ExampleClass : MonoBehaviour
{
    IEnumerator WaitAndPrint()
    {
        yield return new WaitForSeconds(5);
        print("WaitAndPrint " + Time.time);
    }
    IEnumerator Start()
    {
        print("Starting " + Time.time);
        yield return StartCoroutine("WaitAndPrint");
        print("Done " + Time.time);
    }
}
```

3.2.3 Player Preferences

Used to store and access player game data in device memory such as player scores, game levels locked and unlocked information. Example code of saving the player score is as follows:

```
public class SetUpPlayerPrefsExample : MonoBehaviour
{
    void AddScore()
    {
        PlayerPrefs.SetInt("Scores", 50);
    }
}
```

3.2.4 Game Objects

The fundamental objects in Unity game engine that represent characters, props and scenery are known as game objects. In the example below a game object is created and components are added to the game objects.

```
public class ExampleScript : MonoBehaviour
{
    void SetGameObject()
    {
        GameObject player; player = new GameObject("Player");
        player.AddComponent<Rigidbody>();
        player.AddComponent<BoxCollider>();
    }
}
```

3.2.5 Tags

Tags are the keywords assigned to the game objects. Tags are used to access and identify the game objects in the code to perform main functionality. In the example game object are found using tag.

```
public class Example : MonoBehaviour
{
    public GameObject regenerate;
    void CreateEnemy()
    {
        if (regenerate == null)
            regenerate = GameObject.FindWithTag("regenerate");
    }
}
```

3.2.6 Transform

Every object in the game scene can be transformed. Transform is used to set the position, rotation and scale (size) of game objects. Example to set the position of game object in upward direction is explained below.

```
public class ExampleClass : MonoBehaviour
{
    void Move()
    {
            transform.position += Vector3.up * 5.0F;
    }
}
```

3.2.7 Game Scenes

Environment, obstacles, decoration, game objects, game play designing and building is performed in the scenes in Unity editor.

```
public class LoadScenesA : MonoBehaviour
{
    SceneManager.LoadScene("SceneA");
}
```

3.2.8 Game Cameras

Game objects and environment of game is visualized by the player on the device using the cameras. Example to set orthographic camera for 2D game environment and objects display to the player is as follows.

```
public void Start()
{
    Camera.main.orthographic = true;
}
```

3.2.9 Canvas

Canvas is a specified area in the game editor used to set the user interface elements. All the UI elements are set inside the canvas in Unity editor.

3.2.10 Invoke

Invoke function are used to call method at a later time as per schedule in the code scripting. Game object is created using the invoke function in the code below example.

```
public class InvokeScript : MonoBehaviour
{
    void CallEnemy()
    {
        Invoke ("Enemy", 2);
    }
}
```

3.2.11 Raycast

A ray is cast from a point of origin in a specific direction of length set by the programmer of maximum value which is used for collision detection of the objects in the scenes.

```
public class ExampleClass : MonoBehaviour
{
    public Collider coll;
    void Start()
    {
        coll = GetComponent<Collider>();
    }
    void Update()
    {
        Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
        transform.position = ray.GetPoint(100.0F);
    }
}
```

A large number of other special programming features also exist for Unity 3D used for game development but the main and basic programming features which are used in the development of 2D and 3D games are explained above with appropriate code example. Game play failure at the run time may occur because of incorrect use of these special programming features during game development.

3.3 Unity 3D Features Categorization

We have categorized the Unity 3D features which are shown in the Figure 3.3. 7 high level categories are defined with basic features used in Unity 3D for developing 3D and 2D games. Specific programming feature are group together that could affect the game development if faults are introduced in them during the development process. From the multiple categorizes few of the basic special programming features of the Unity 3D are selected for defining the mutation operators. To avoid generation of the still born mutants such Unity 3D features are selected which do not results compilation error if faults are seeded in them. The features categorizes of the Unity 3D is as follows:

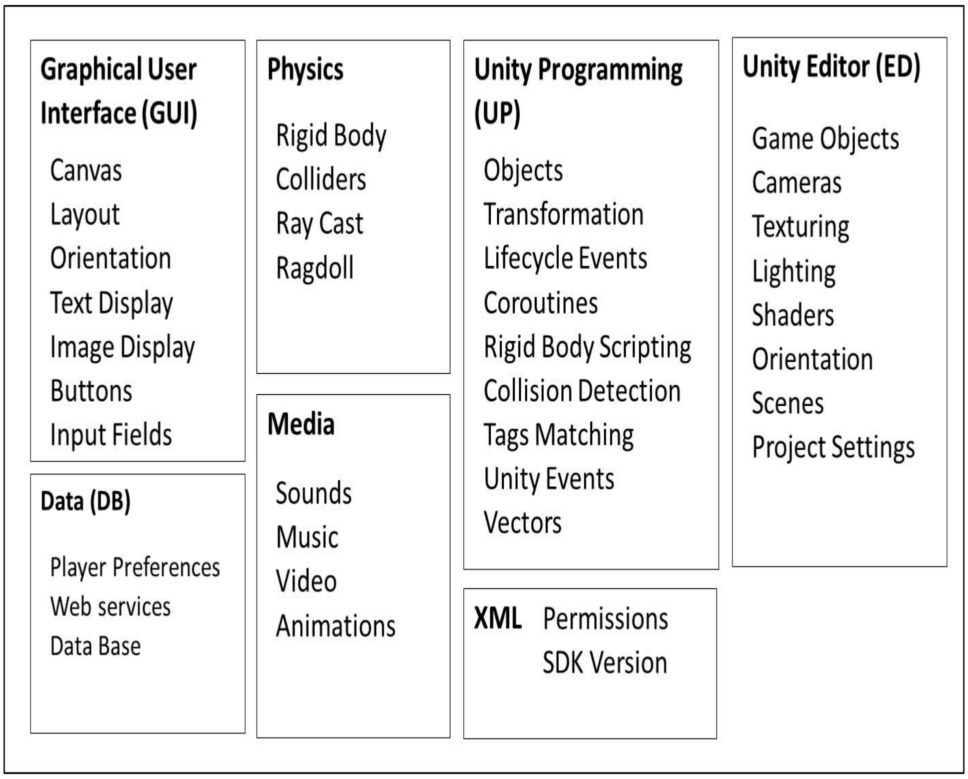


FIGURE 3.3: Identified Taxonomy of Unity 3D Faults

3.4 Proposed Mutation Operators

The proposed mutation operators to cover the Unity 3D C# special programming features along with the example code with the output or behavior difference of mutants as compared to the original code. Special programming features for designing the mutation operators are selected from multiple categories in such a way that no still born mutants are generated by seeding the mutation. Basic programming features from each category are selected depending on its use and importance in the Unity 3D C# game development programming.

The list of proposed mutation operators selected from the major categories along with the name and short description is provided in the Table 3-1.

Table 3.1: List of Proposed Unity 3D C# Mutation Operators

Mutation Operators	**Cat**	**Description**
Changing Parameter Name of PlayerPrefs (PPC)	DB	Randomly replace the parameter values in the PlayerPref Get function
Removing Parameter of PlayerPrefs (PPR)	DB	Search and Remove the parameter values in the PlayerPref Get function
Invalid Parameter of PlayerPref (PPI)	DB	Pass the invalid random value in the PlayerPrefs Set function parameter
Disabling Game Object (DGO)	ED	Find and replace the SetActive value False for game
Invalid Scene Loading (ISL)	ED	Load the invalid scene by replacing the parameter for the function randomly SceneManager.LoadScene(1)
Changing Main Game Camera Type (CGC)	ED	Set the false value for the Camera.main.orthographic = true function
OnClick Event Replacement (OCR)	GUI	Replace the function call for the OnClick event
Game Orientation Lock (LGO)	GUI	Replace the value of the ScreenOrientation function

Disabling Canvas Panel View (DCV)	GUI	Set False value for SetActive(true) function
Changing Parameter Values of Invoke Function (IPC)	UP	Find and Replace the parameter values of the Invoke function randomly
Null Pointer Exception (NPE)	UP	Call the reference of the object not available in the script
Game Objects Tags Matching (MGOT)	UP	Replace the parameter value randomly for col.collider.tag == ”Enemy”
Finding Game Object with Tags (FGOT)	UP	Replace the parameter value randomly for the function GameObject.FindGameObjectWithTag()
Invalid Function Call for Coroutine (CIC)	UP	Find and call the invalid function in the Parameter of Coroutine function
Life Cycle Method Replacement (LCR)	UP	Find and Replace the life cycle function in the script
XML Manifest Activity Permission Deletion (APD)	XML	Delete the <uses permission />in the XML manifest file

The 16 mutation operators are defined based on several unique features of Unity 3D C# games from different categories. These mutation operators are proposed to cover multiple game features. The detail of each mutation operator is explained for generating the mutant along with example code.

3.4.1 Changing Parameter Name of PlayerPrefs (PPC)

PlayerPrefs or player preferences is the most commonly used for storing and accessing the data of the player game progress in the local Unity 3D database, a common mistake can be done by assigning the different/wrong variable or leaving the parameter empty.

The mutant of the original program is created by finding the PlayerPrefs statement and changing the parameter used in the function. This type of error can arise if the programmer uses multiple variables for the data saving in the PlayerPrefs to store or fetch user game data. If wrong parameter is used, the appropriate value will not be fetched and the game behavior will be affected linked with that parameter value fetched.

Original Program	Mutated Program

`public void addscore()`
`{`
`  int tempscore = PlayerPrefs.GetInt("Score");`
`  tempscore += 5;`
`}`
	`public void addscore()`
	`{`
	`  int tempscore = PlayerPrefs.GetInt("scor");`
	`  tempscore += 5;`
	`}`

Output/Behavior of Original Program:	**Output/Behavior of Mutated Program:**
The score variable value fetched from the Score Player Preference and is incremented by 5	Incorrect value of scor is fetched from the Player Preference, thus incorrect value will be incremented which will later effect the game behavior as every time wrong value will be accessed

3.4.2 Removing Parameter of PlayerPrefs (PPR)

The mutant of the original program is created by finding the PlayerPrefs statement removing the parameter used. This type of error can arise if the programmer leaves the field without parameter to be updated later. If no parameter is used, the appropriate value will not be fetched and the behavior of the game and player will be affected linked with that parameter value fetched.

Original Program	**Mutated Program**
`public void addscore()`	`public void addscore()`
`{`	`{`
`  int tempscore = PlayerPrefs.GetInt("Score");`	`  int tempscore = PlayerPrefs.GetInt("");`
`  tempscore += 5;`	`  tempscore += 5;`
`}`	`}`
Output/Behavior of Original Program:	**Output/Behavior of Mutated Program:**
The score variable value fetched from the Score Player Preference and is incremented by 5	0 integer value is fetched from the Player Preference, thus incorrect value will be incremented which will later effect the game behavior as every time wrong value will be accessed

3.4.3 Invalid Parameter of PlayerPrefs (PPI)

The mutant of the original program is created by finding the PlayerPrefs statement and using the invalid or static value of function parameter. This type of error can arise if the programmer use invalid variable or static value for the data saving in the PlayerPrefs to store user game data. If invalid or static parameter is used, the appropriate value will not be stored and the behavior of the game will be affected linked with that parameter value saved to be used for game functionality.

Original Program	**Mutated Program**
`public void addcoin()`	`public void addcoin()`
`{`	`{`
`  PlayerPrefs.SetInt("Coins",  noofcoins)`	`  PlayerPrefs.SetInt("Coins",  0)`
`}`	`}`
Output/Behavior of Original Program:	**Output/Behavior of Mutated Program:**
The integer value of noofcoins variable is stored in the Coins Player Preference	The incorrect 0 integer value is stored in the Coins Player Preference which will affect the game behavior when the value of the Coins will be accessed later

Original Program	Mutated Program

`public void addcoins()`
`{`
`  PlayerPrefs.SetInt("Coins",  noofcoins)`
`}`
	`public void addcoins()`
	`{`
	`  PlayerPrefs.SetInt("Silver", noofcoins)`
	`}`

Output/Behavior of Original Program:	**Output/Behavior of Mutated Program:**
The integer value of noofcoins variable is stored in the Coins Player Preference	The integer value of noofcoins variable is stored in the Silver Player Preference due to which the value of coins will not increase while value of Silver will increase that can affect the game play

3.4.4 Changing Parameter Values of Invoke Function (IPC)

Invoke Function is an important feature of Unity 3D C# game programming which is used to call the methods, or scenes of the game at the run time after the specific time. Change in function name called or time of call may result in the change in game behavior which result errors in game. By replacing the name of function in the parameter of the Invoke function or by changing the time value of Invoke function may cause error or change of game behavior.

Original Program	Mutated Program

`public class TimedObjectDestructor : MonoBehaviour`
`{`
`   private float m_TimeOut = 1.0f;`
`   private void Awake()`
`   {`
`     Invoke("DestroyNow",  m_TimeOut);`
`    }`
`   void DestroyNow ()`
`   {`
`     Destroy(gameobject);`
`   }`
`}`
	`public class TimedObjectDestructor : MonoBehaviour`
	`{`
	`   private float m_TimeOut = 1.0f;`
	`   private void Awake()`
	`   {`
	`     Invoke("destroy",  m_TimeOut);`
	`    }`
	`   void DestroyNow ()`
	`   {`
	`     Destroy(gameobject);`
	`   }`
	`}`

Output/Behavior of Original Program:	**Output/Behavior of Mutated Program:**
The DestroyNow function is called for the game object after the time of 1 second	The function destroy is called which do not exists thus the game object remain in the game play which can change the game play behavior

Original Program

```
public class TimedObjectDestructor : MonoBehaviour
{
   private float m_TimeOut = 1.0f;
   private void Awake()
   {
     Invoke("DestroyNow", m_TimeOut);
    }
   void DestroyNow ()
   {
     Destroy(gameobject);
   }
}
```

Mutated Program

```
public class TimedObjectDestructor : MonoBehaviour
{
   private float m_TimeOut = 1.0f;
   private void Awake()
   {
     Invoke("", m_TimeOut);
    }
   void DestroyNow ()
   {
     Destroy(gameobject);
   }
}
```

Output/Behavior of Original Program:

The DestroyNow function is called for the game object after the time of 1 second

Output/Behavior of Mutated Program:

No function is called for the game object after the 1 second time which can result in the run time error which will affect the game functionality

Original Program

```
public class TimedObjectDestructor : MonoBehaviour
{
   private float m_TimeOut = 1.0f;
   private void Awake()
   {
     Invoke("DestroyNow", m_TimeOut);
    }
   void DestroyNow ()
   {
     Destroy(gameobject);
   }
}
```

Mutated Program

```
public class TimedObjectDestructor : MonoBehaviour
{
   private float m_TimeOut = 1.0f;
   private void Awake()
   {
     Invoke("DestroyNow ", 5f);
    }
   void DestroyNow ()
   {
     Destroy(gameobject);
   }
}
```

Output/Behavior of Original Program:

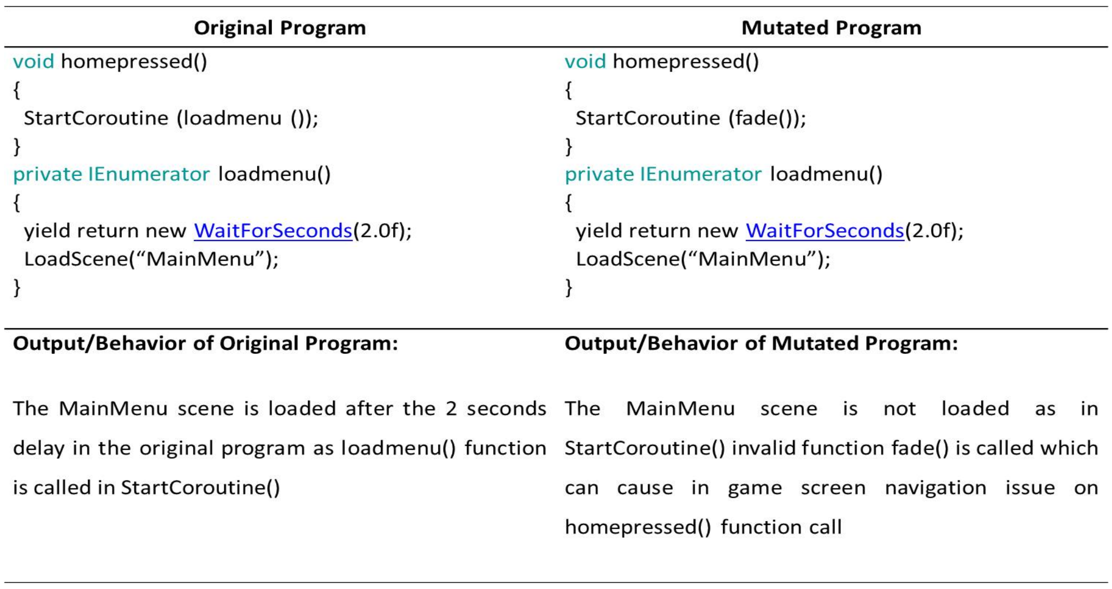
The DestroyNow function is called for the game object after the time of 1 second

Output/Behavior of Mutated Program:

The DestroyNow function is called for the game object after the time of 5 seconds which can change game play behavior as the game object will take more time to destroy as compared to the normal destroy time

3.4.5 Invalid Function Call for Coroutine (CIC)

A coroutine is like a function that has the ability to pause execution and return control to Unity but then to continue where it left off on the following frame. By changing the function name called in the Coroutine will generate a mutant that will cause the error or change in the game behavior.



Original Program	**Mutated Program**
void homepressed()	void homepressed()
{	{
StartCoroutine (loadmenu ());	StartCoroutine (fade());
}	}
private IEnumerator loadmenu()	private IEnumerator loadmenu()
{	{
yield return new WaitForSeconds(2.0f);	yield return new WaitForSeconds(2.0f);
LoadScene("MainMenu");	LoadScene("MainMenu");
}	}
Output/Behavior of Original Program:	**Output/Behavior of Mutated Program:**
The MainMenu scene is loaded after the 2 seconds delay in the original program as loadmenu() function is called in StartCoroutine()	The MainMenu scene is not loaded as in StartCoroutine() invalid function fade() is called which can cause in game screen navigation issue on homepressed() function call

3.4.6 Disabling Game Object (DGO)

Any object of the game used in the environment can be accessed and controlled using scripts, mainly objects visibility is controlled using scripts to set specific points to enable and disable game objects during game play as per player interactions. A mutant can be generated by disabling the enabled game object by changing Boolean value for the object. The change in the Boolean value will result in hiding game object thus game behavior will be affected with the disabled game object.

Original Program	**Mutated Program**
`void Start()`	`void Start()`
`{`	`{`
`    GameObject tempobj =   GameObject.Find`	`    GameObject tempobj =   GameObject.Find`
`    ("Cube2");`	`    ("Cube2");`
`    tempobj.SetActive (true);`	`    tempobj.SetActive (false);`
`}`	`}`
Output/Behavior of Original Program:	**Output/Behavior of Mutated Program:**
In the game environment Cube2 game object is found and it is set active which will be viewed by the user during the game play	In the game environment Cube2 game object is found and the value set active is set false hiding the game object from the game play which affects the game behavior and functionality.

3.4.7 Game Object Tag Name Mutation Operators

A tag is a reference word which you can assign to one or more game objects. Tags help programmer identify game objects for scripting purposes. Tags are useful for triggers in Collider control scripts; they need to work out whether the player is interacting with an enemy, a prop, or a collectable.

3.4.7.1 Game Objects Tags Matching (MGOT)

The complete game behavior can be disturbed in a game during the run time if the wrong tag is being used for any game object which cause bugs in the game.

Original Program	**Mutated Program**

`void OnCollisionEnter(Collision col)`
`{`
` if (col.collider.tag == "Enemy") {`
`  GameGUI.instance.addscore ();`
` }`
`}`
	`void OnCollisionEnter(Collision col)`
	`{`
	` if (col.collider.tag == "Player") {`
	`  GameGUI.instance.addscore ();`
	` }`
	`}`

Output/Behavior of Original Program: In the game play the Enemy tag is checked to add the score of user on hitting the enemy game object	**Output/Behavior of Mutated Program:** In the game play the incorrect tag value which is player is checked to add the score of the user on hitting thus wrong scoring will be performed in the game play which will affect the game play.

Original Program	**Mutated Program**

`void OnCollisionEnter(Collision col)`
`{`
` if (col.collider.tag == "Enemy") {`
`  GameGUI.instance.addscore ();`
` }`
`}`
	`void OnCollisionEnter(Collision col)`
	`{`
	` if (col.collider.tag == "") {`
	`  GameGUI.instance.addscore ();`
	` }`
	`}`

Output/Behavior of Original Program: In the game play the Enemy tag is checked to add the score of the user on hitting the enemy game object	**Output/Behavior of Mutated Program:** In the game play no tag value is matched to add the score of the user on hitting which results in no score value increment during the game play which will affect the game play.

3.4.7.2 Finding Game Object with Tags (FGOT)

Tags are majorly used to find the game objects in the Unity 3D scripting to perform multiple functionalities on them. If wrong tag is used for finding game object the wrong result will affect the game behavior and cause multiple issues during the game play at run time.

Original Program	Mutated Program

void destroyenemy()
{
GameObject enemy = GameObject.FindGameObjectWithTag
("Enemy");
 	
  if (enemy) {
    Destroy (enemy);
  }
}
	void destroyenemy()
	{
	GameObject enemy = GameObject.FindGameObjectWithTag
	("Player");
	
	  if (enemy) {
	    Destroy (enemy);
	  }
	}

Output/Behavior of Original Program: Game object with the enemy tag is searched and if found then the enemy is destroyed.	**Output/Behavior of Mutated Program:** Game object with the player tag is searched instead of enemy and as a result the real enemy object in game play will not be destroyed which will cause the behavior change of the game play.

Original Program	Mutated Program

void destroyenemy()
{
GameObject enemy = GameObject.FindGameObjectWithTag
("Enemy");
 	
  if (enemy) {
    Destroy (enemy);
  }
}
	void destroyenemy()
	{
	GameObject enemy = GameObject.FindGameObjectWithTag
	("");
	
	  if (enemy) {
	    Destroy (enemy);
	  }
	}

Output/Behavior of Original Program: The game object with enemy tag is searched and if found then the enemy is destroyed.	**Output/Behavior of Mutated Program:** No game object will be searched because of invalid tag value and as a result the enemy object in the game play will not be destroyed which will cause the behavior change of the game play.

3.4.8 Life Cycle Method Replacement (LCR)

Life cycle methods are used by developers to override them to define functionalities among those life cycle states. Life cycle method replacement change the overriding life cycle method to force calls different life cycle method which results in different functionality performance in calling methods which may cause issues at the run time.

Original Program	Mutated Program

`void Start()`
`{`
`GameObject enemy = GameObject.FindGameObjectWithTag`
`("Enemy");`
`}`
	`void Awake()`
	`{`
	`GameObject enemy = GameObject.FindGameObjectWithTag`
	`("Enemy");`
	`}`

Output/Behavior of Original Program:	**Output/Behavior of Mutated Program:**
Game object is found in the Start method of the Unity 3D life cycle to perform the functionality on the object later.	In the Unity 3D life cycle the Awake function is called before the Start function at the very beginning so run time error can be caused as if object is searched before it is being initialized.

3.4.9 Invalid Scene Loading (ISL)

Scenes contain the environments and menus of the game. Levels are designed in multiple scenes which are called from the scripts to load and start scene to ensure the normal game play behavior. Scenes of games can be loaded either by calling scene name or index of scene depending who the scenes are created and naming convention used for scenes saving. Calling invalid scene name can cause the error in game play.

Original Program	Mutated Program

`public void nextscene()`
`{`
` SceneManager.LoadScene(1);`
`}`
	`public void nextscene()`
	`{`
	` SceneManager.LoadScene(0);`
	`}`

Output/Behavior of Original Program:	**Output/Behavior of Mutated Program:**
The Scene 1 of the game is loaded and played after the input event given by the user.	The Scene 0 of the game is loaded which can cause the game navigation issue or even a run time crash if no scene of index 0 is available in the game.

Original Program	Mutated Program

public void nextscene()
{
 SceneManager.LoadScene("Gameplay");
}
	public void nextscene()
	{
	 SceneManager.LoadScene("Menu");
	}

Output/Behavior of Original Program: The Gameplay Scene of the game is loaded after the input event given by the user.	**Output/Behavior of Mutated Program:** The Menu Scene is loaded which can cause game navigation issue making it impossible to start the game play for the user.

3.4.10 Changing Main Game Camera Type (CGC)

Cameras are the devices that capture and display game world to the player. Unity 3D support 2 types of main camera that is, Orthographic and Prospective. Usually perspective camera type is used for 3D games while orthographic is used for 2D games. Changing camera type may result in complete game behavior change also affecting the visibility of game objects.

Original Program	Mutated Program

public void Start()
{
 Camera.main.orthographic = true;
}
	public void Start()
	{
	 Camera.main.orthographic = false;
	}

Output/Behavior of Original Program: In the game start the Orthographic camera type is used.	**Output/Behavior of Mutated Program:** In the game start the camera type other than orthographic is set that is prespective camera is set which is used for the 2D games, thus it will completely change the game play and game GUI view for the user hiding the GUI and game play objects.

3.4.11 OnClick Event Replacement (OCR)

Behind OnClick event the appropriate function set for the functionality is called. By replacing the function call with other compatible function for OnClick event may result in the behavior change or error during the game play at the run time.

Original Program	Mutated Program
`public void Start()`	`public void Start()`
`{`	`{`
` homebtn.onClick.AddListener (homepressed);`	` homebtn.onClick.AddListener (restartpressed);`
`}`	`}`
`public void homepressed()`	`public void homepressed()`
`{`	`{`
` SceneManager.LoadScene("Menu");`	` SceneManager.LoadScene("Menu");`
`}`	`}`
Output/Behavior of Original Program:	**Output/Behavior of Mutated Program:**
On button pressed from the GUI, the button event of home is registered by the OnClick function listener and Menu scene is loaded.	On button pressed from the GUI, the button event of restart is registered by the OnClick function listener and instead of calling the homepressed function restart function is called which will affect the game screens navigation.

3.4.12 Disabling Canvas Panel View (DCV)

The Canvas is the area where all UI elements placed. Disabling the enabled canvas will hide UI components for the player interaction to perform the game functionality causing multiple game play issues at run time.

Original Program	Mutated Program
`public void gameover()`	`public void gameover()`
`{`	`{`
`  gameoverpanel.SetActive(true);`	`  gameoverpanel.SetActive(false);`
`}`	`}`
Output/Behavior of Original Program:	**Output/Behavior of Mutated Program:**
On the gameover game over panel is displayed to the user with appropriate buttons to navigate in the game.	On the gameover the game over panel is not displayed to the user which restricts user interaction after game over causing navigation issues in the game.

Original Program	**Mutated Program**
`public void gameplay()`	`public void gameplay ()`
`{`	`{`
`  gameplaypanel.SetActive(false);`	`  gameplaypanel.SetActive(true);`
`}`	`}`
Output/Behavior of Original Program:	**Output/Behavior of Mutated Program:**
During the game play the game over panel is not displayed and is turned off.	During the game play the game over panel is displayed which restricts user to view the game play scene and play and interact completely with the game play objects.

3.4.13 Game Orientation Lock (LGO)

Many apps change the layout of GUI when the orientation changes but this is not the case for the mobile games as the single orientation is set for the game and on changing the orientation GUI of the game and game play is changed. Thus, switching the orientation in turn leads to many faults in the game.

Original Program	**Mutated Program**
`public void Start()`	`public void Start()`
`{`	`{`
`  Screen.orientation = ScreenOrientation.LandscapeLeft;`	`  Screen.orientation = ScreenOrientation.Potrait;`
`}`	`}`
Output/Behavior of Original Program:	**Output/Behavior of Mutated Program:**
At the game start the device orientation is set as landscape as per the game requirements.	At the game start the game orientation is changed to portrait which will change the game display for user as the game GUI & game play is set for the landscape screen mode of the device.

3.4.14 Null Pointer Exception (NPE)

The most common error faced is the Null Pointer Exception which occurs when object reference does not exist while it is being used in the game. The Null Pointer

Exception causes the run time errors and bugs in the game also affecting the game play behavior.

Original Program	Mutated Program
`void destroyenemy() { GameObject enemy = GameObject.FindGameObjectWithTag ("Enemy"); if (enemy) { Destroy (enemy); } }`	`void destroyenemy() { GameObject enemy = GameObject.FindGameObjectWithTag ("Friend"); if (enemy) { Destroy (enemy); } }`
Output/Behavior of Original Program:	**Output/Behavior of Mutated Program:**
The game object with the enemy tag is searched and if found then the enemy is destroyed.	The game object with the friend tag is searched instead of enemy and if no object of the Friend tag will be found the Null Pointer Exception will be thrown at the run time which will change the game behavior.

3.4.15 XML Manifest Activity Permission Deletion (APD)

For each Unity 3D game permissions are requested as per main functionalities performed in the game which can be location permission, device data storage access permission etc. These permissions are required and requested from the user when an application is installed first or after app is successfully installed on the device. By deleting the permission required for game from Manifest.xml file functionality of the game can be affected during the game play.

Original Program	Mutated Program
`<uses-permission Android:name="Android.permission.WRITE_EXTERNAL_STORAGE“ />`	`<- - <uses-permission Android:name="Android.permission.WRITE_EXTERNAL_STORAGE" /> - - >`
Output/Behavior of Original Program:	**Output/Behavior of Mutated Program:**
The permission to access and store game data in the device storage is used.	If the permission is commented the game data may not be stored thus on restarting the game all the user data will be lost.

By defining the above Unity 3D C# special programming features mutation operators we can be able to handle Unity 3D C# specific programming features faults that a programmer can be do while game programming. With the proposed mutation operators, different type of faults will be seeded in the program which can relate to game programming, editor, GUI. As we know, Unity 3D C# has special programming features for game scripting and by mistake of the programmer in assigning the parameters, parameters values, function calling errors or faults may occur. By combining these proposed mutation operators with the existing mutation operators, the assessment of test cases will be improved.

Chapter 4

Implementation

4.1 Implementation

4.1.1 Overview

Mutation analysis for the mobile applications cannot be done as it is done for the traditional languages including JAVA, Java Script programs. Mobile applications require additional processing before they are being executed and deployed. Mobile applications are compiled, packaged, installed and executed on mobile devices and emulators which is different from the JAVA or other basic languages programs.

This implementation details of the proposed technique is described, we have developed a GUI based desktop application for generating mutants for Unity 3D C# using defined and existing mutation operators which can be applied on Unity 3D C# program. Implementation is done on JAVA language using Object Oriented Paradigm. NetBeans IDE is used to develop tool for mutants generation.

Generic Software Architecture of tool used to generate the mutants of Unity 3D C# is shown in Figure 4.1.

1. Our tool takes the Unity 3D C# or Manifest XML file as an input and traditional or proposed mutation operators can be applied on the selected program files.

2. Mutants of the files are generated based on the selected operators in the mutant generator process.

3. The mutant codes generated are stored separately in the respective folder created for the each mutation operator to run the test cases to check if test cases detect faults that are seeded in the mutants by mutation operators.

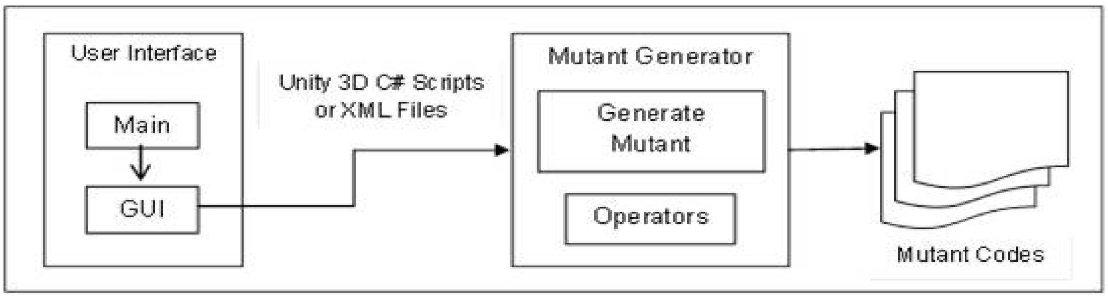


FIGURE 4.1: Generic tool architecture for mutants generation

The detailed process of mutation generation for Unity 3D C# games is explained in the Figure 4.2.

1. At first the mutation operators are selected that should be used for both C# scripts and XML file from the list of traditional and proposed mutation operators.

2. The C# scripts mutation operators are applied directly on the source code of the Unity 3D to be complied thus creating a new copy.

3. Similarly, XML mutation operators are applied directly to the XML file, creating a new copy of the file.

4. The mutants are compiled to be executed on the device using the Unity 3D editor using the Unity Remote 5 platform.

5. During the process of compilation some mutants might cause the compilation errors which are referred as the still born mutants which are discarded immediately. But no such mutation operators are designed which can cause the program syntax error leading to still born mutants generation.

6. The original program and each mutant are executed with all test cases and output of the original programs and mutants are recorded for the analysis of test suite effectiveness.

7. The results comparison of test cases is performed for the killed mutants by the test cases. Mutation score is calculated according to the mutants killed by the test cases.

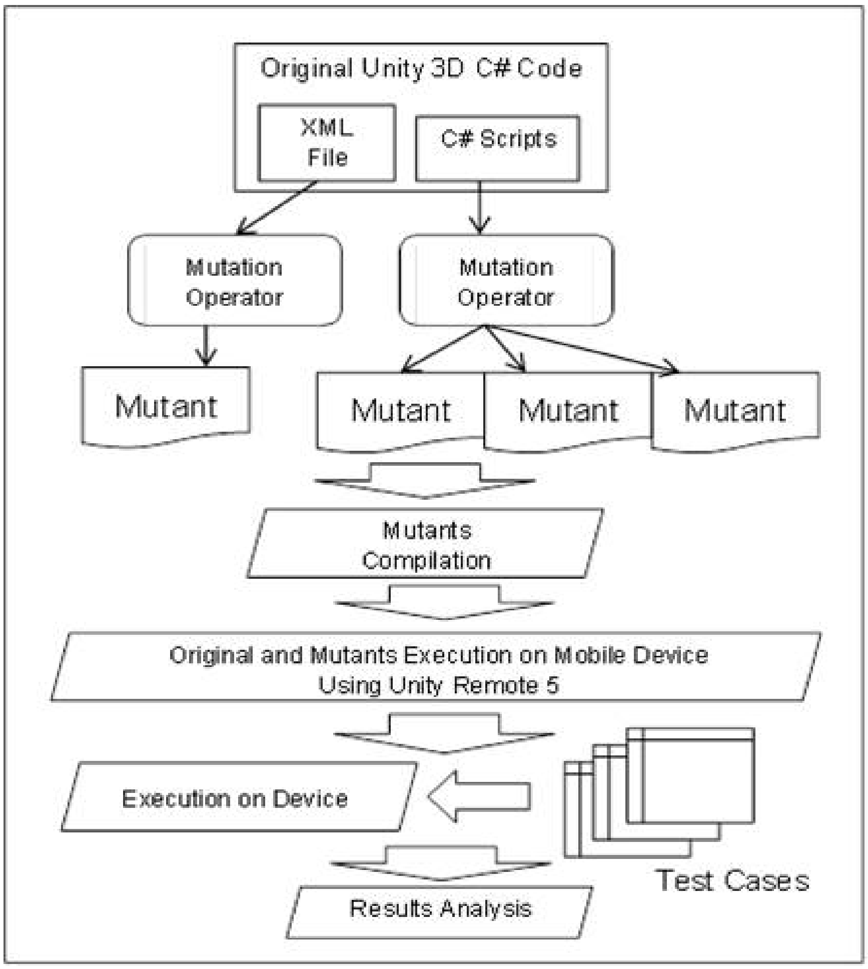


FIGURE 4.2: Mutation Analysis Process for Unity 3D C# Games

4.2 Mutant Generation Process

The working of tool starts with the main menu presented as a GUI to the user which asks for the Unity 3D C# Scripts code as an input for mutation testing. Then Unity 3D C# scripts files are passed to "Mutant Generator" component. Mutation operators are selected for mutants generation. Mutants are generated based on the operators selected for mutation. The algorithm used for mutant generation is explained as follows:

Algorithm Mutant Generation

Input: Unity 3D C# Scripts

Output: n number of mutant of original program where $n \in \{1,2,3,\dots\}$

Declare:

```
1.  INITIALIZE
2.  COPY C# Scripts for mutations
3.  while !line = C#.end_of_file do        // read code line by line
4.      if operator_condition than         // check the code for mutation
5.      count++
6.  End
7.  DISPLAY no. of mutants against each operator in scripts
8.  INITIALIZE script ← 0                   // mutant file script no.
9.  while script do
10.     create new file
11.     while !line = c#.end_of_file do         // read code line by line
12.         if operator_condition than
13.             if script == replaceCount than
14.                 Call replaceLine(line)
15.             else
16.                 replaceCount++
17.         Write line on C#
18.     End
19.     replaceCount ← 0
20.     script++
21. End
```

4.2.1 Algorithm Description

To generate mutants of given program, first, count how many time selected mutation operator(s) will apply for mutant generation (lines 1-5). While loop continuously read file line by line until 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 is true then increment count value by 1 (line 4). The number of mutants to be generated against each mutation operator in each script is displayed to the user (line 6). After mutants generation process begins and number of mutants will be created as many as the value of count variable (lines 9-20). The outer while loop executes until script value is available for mutation and in each iteration of this loop, every time a new file is created with incremented script number. Inner while loop of algorithm read original code from the file line by line for replacement (line 14). If statement (line 12), check whether current line contains selected operator condition or not. If the is condition true then check the sequence in which the replacement will have to apply (lines 13-14). For example, if tester selects “PlayerPref.SetInt” mutation operator and “PlayerPref.SetInt” keyword found more than once in the program. Then in each mutant, replace “PlayerPref.SetInt” parameter that already not has been replaced for the creation of new mutant. If the condition is false then found the next appearance of operator that has not already mutated (line 16). In line 17, write each line on mutant file to make a copy of the original program. At the end of the outer loop, n number of mutants will be generated.

4.3 Analysis Process

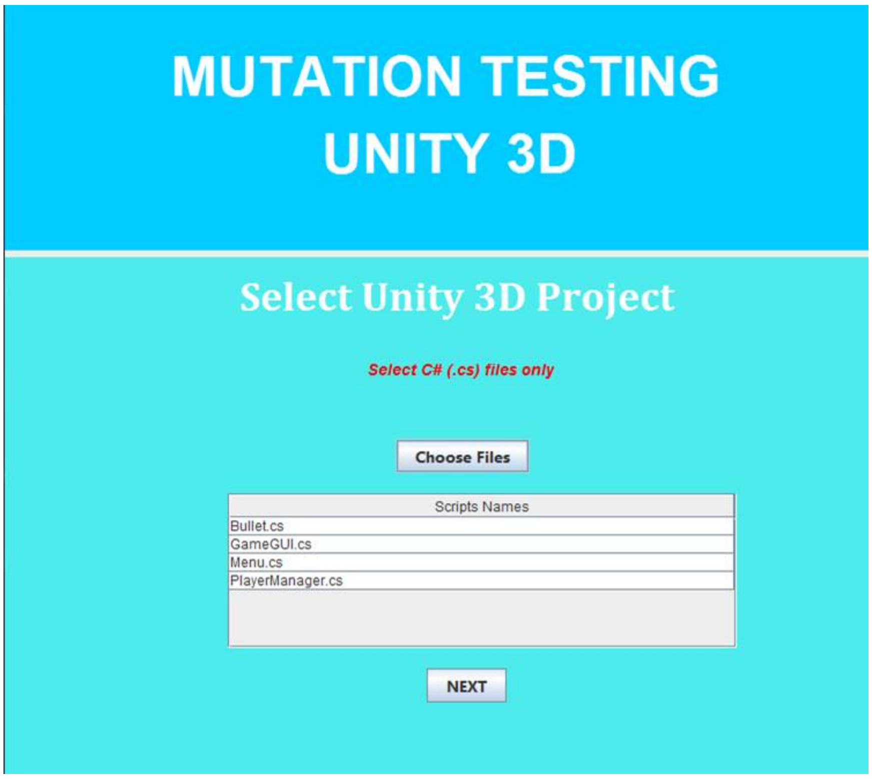
After generating mutants of the source program, next phase is to execute these generated mutants with test cases that are developed by the tester to test the source program. Test case executes both the original program and the mutant with the goal that each mutant should produce different output from original program. The detailed process of analysis along with the results is explained in the next chapter

4.4 Tool Usage

This section includes all the user interfaces of our tool developed to generate the mutants.

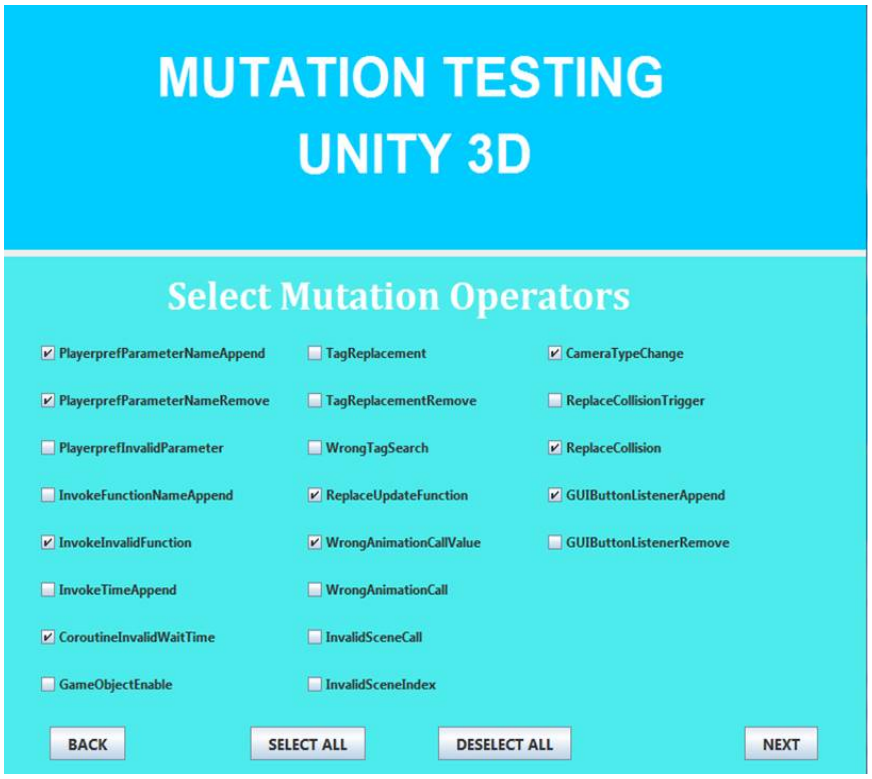
4.4.1 Mutant Generation Interface

To start creating the mutants for the Unity 3D C# scripts and XML file, our tool will take C# and XML program as input and then tester select mutation operators that will apply on scripts source code to generate mutants of source program. Figure 4.2 shows the files selected for applying mutation operators. In the first step, the tester will browse the C# source code or XML file and then select mutation operators for which he/she want to generate mutants. After selection of operators tester will press "Next' button to check mutants stats. The mutants count for each script will be displayed to the user in table form. On "Generate Mutant" button click the mutants will be generated against each mutation operators. After generation of mutants for the selected mutation operators, next step is to execute test cases on these generated mutants and original program.



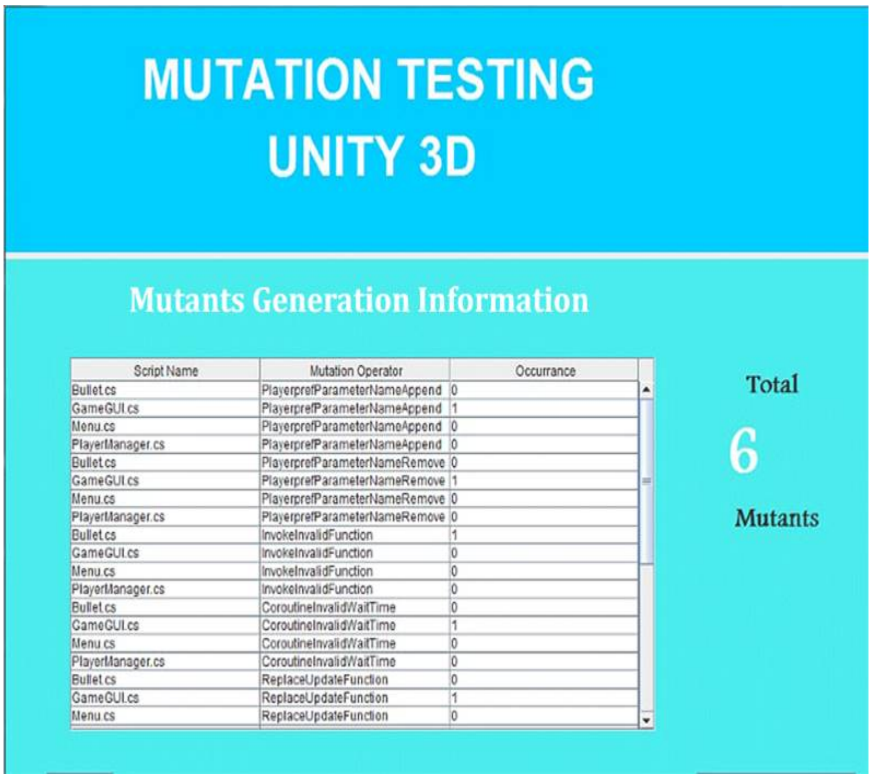
Scripts Names
Bullet.cs
GameGUI.cs
Menu.cs
PlayerManager.cs

FIGURE 4.3: Source Files Selection for Mutation Analysis



- [✓] PlayerprefParameterNameAppend
- [✓] PlayerprefParameterNameRemove
- [ ] PlayerprefInvalidParameter
- [ ] InvokeFunctionNameAppend
- [✓] InvokeInvalidFunction
- [ ] InvokeTimeAppend
- [✓] CoroutineInvalidWaitTime
- [ ] GameObjectEnable
- [ ] TagReplacement
- [ ] TagReplacementRemove
- [ ] WrongTagSearch
- [✓] ReplaceUpdateFunction
- [✓] WrongAnimationCallValue
- [ ] WrongAnimationCall
- [ ] InvalidSceneCall
- [ ] InvalidSceneIndex
- [✓] CameraTypeChange
- [ ] ReplaceCollisionTrigger
- [✓] ReplaceCollision
- [✓] GUIButtonListenerAppend
- [ ] GUIButtonListenerRemove

FIGURE 4.4: Mutation Operators Selection for Mutants Generation



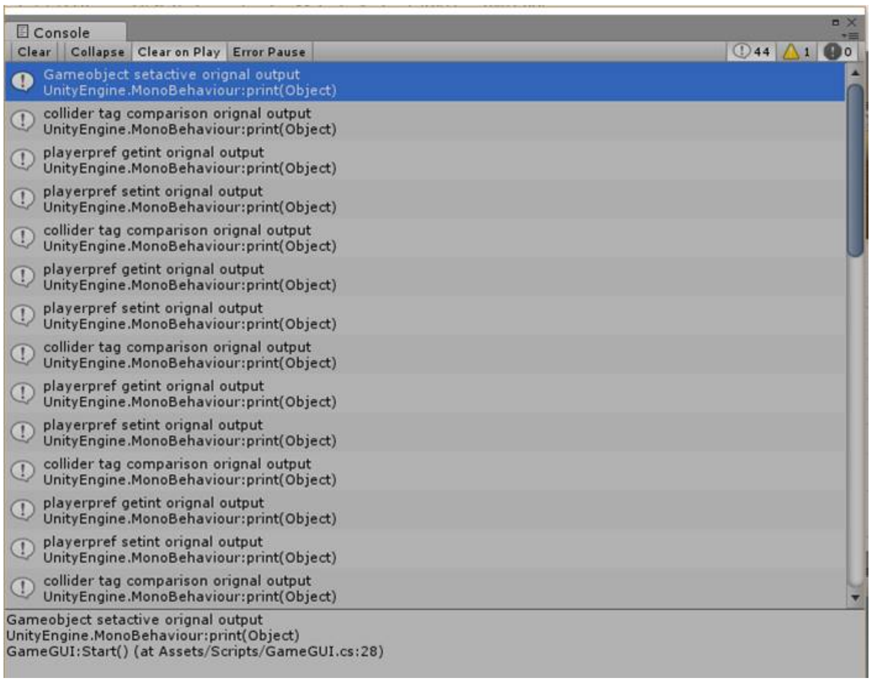
Script Name	Mutation Operator	Occurrance
Bullet.cs	PlayerprefParameterNameAppend	0
GameGUI.cs	PlayerprefParameterNameAppend	1
Menu.cs	PlayerprefParameterNameAppend	0
PlayerManager.cs	PlayerprefParameterNameAppend	0
Bullet.cs	PlayerprefParameterNameRemove	0
GameGUI.cs	PlayerprefParameterNameRemove	1
Menu.cs	PlayerprefParameterNameRemove	0
PlayerManager.cs	PlayerprefParameterNameRemove	0
Bullet.cs	InvokeInvalidFunction	1
GameGUI.cs	InvokeInvalidFunction	0
Menu.cs	InvokeInvalidFunction	0
PlayerManager.cs	InvokeInvalidFunction	0
Bullet.cs	CoroutineInvalidWaitTime	0
GameGUI.cs	CoroutineInvalidWaitTime	1
Menu.cs	CoroutineInvalidWaitTime	0
PlayerManager.cs	CoroutineInvalidWaitTime	0
Bullet.cs	ReplaceUpdateFunction	0
GameGUI.cs	ReplaceUpdateFunction	1
Menu.cs	ReplaceUpdateFunction	0

Total
6
Mutants

FIGURE 4.5: Stats Display before Mutants Generation

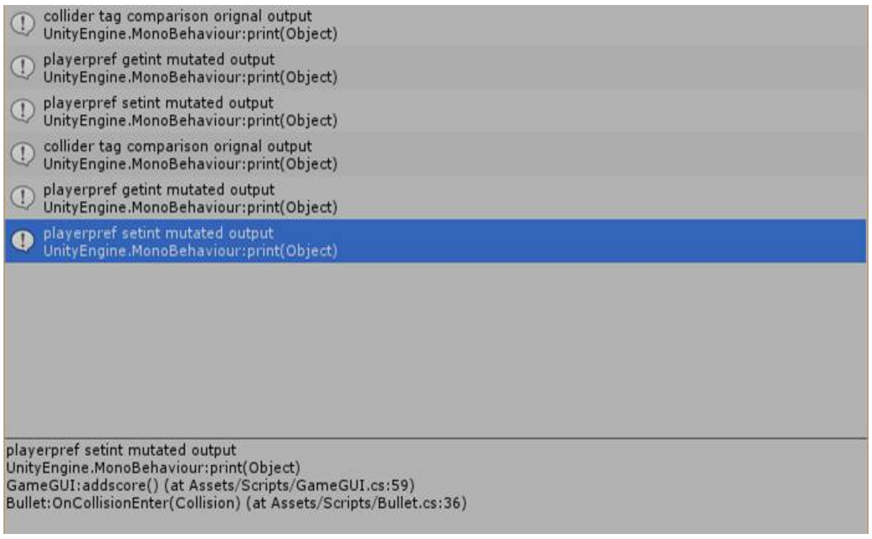
4.5 Test Case Execution Process

The mutants created are executed with the test cases designed by the testers to record the output. Each test case is executed using the Unity Remote 5 platform for running the mutants on the mobile device. The output is recorded using the Unity 3D console for comparison.



```
Gameobject setactive orignal output
UnityEngine.MonoBehaviour:print(Object)
collider tag comparison orignal output
UnityEngine.MonoBehaviour:print(Object)
playerpref getint orignal output
UnityEngine.MonoBehaviour:print(Object)
playerpref setint orignal output
UnityEngine.MonoBehaviour:print(Object)
collider tag comparison orignal output
UnityEngine.MonoBehaviour:print(Object)
playerpref getint orignal output
UnityEngine.MonoBehaviour:print(Object)
playerpref setint orignal output
UnityEngine.MonoBehaviour:print(Object)
collider tag comparison orignal output
UnityEngine.MonoBehaviour:print(Object)
playerpref getint orignal output
UnityEngine.MonoBehaviour:print(Object)
playerpref setint orignal output
UnityEngine.MonoBehaviour:print(Object)
collider tag comparison orignal output
UnityEngine.MonoBehaviour:print(Object)
playerpref getint orignal output
UnityEngine.MonoBehaviour:print(Object)
playerpref setint orignal output
UnityEngine.MonoBehaviour:print(Object)
collider tag comparison orignal output
UnityEngine.MonoBehaviour:print(Object)
Gameobject setactive orignal output
UnityEngine.MonoBehaviour:print(Object)
GameGUI:Start() (at Assets/Scripts/GameGUI.cs:28)
```

FIGURE 4.6: Output of Original Program for Test Cases Execution



```
collider tag comparison orignal output
UnityEngine.MonoBehaviour:print(Object)
playerpref getint mutated output
UnityEngine.MonoBehaviour:print(Object)
playerpref setint mutated output
UnityEngine.MonoBehaviour:print(Object)
collider tag comparison orignal output
UnityEngine.MonoBehaviour:print(Object)
playerpref getint mutated output
UnityEngine.MonoBehaviour:print(Object)
playerpref setint mutated output
UnityEngine.MonoBehaviour:print(Object)
playerpref setint mutated output
UnityEngine.MonoBehaviour:print(Object)
GameGUI:addscore() (at Assets/Scripts/GameGUI.cs:59)
Bullet:OnCollisionEnter(Collision) (at Assets/Scripts/Bullet.cs:36)
```

FIGURE 4.7: Output of Mutated Program for Test Case Execution

After executing all mutants with test data, list of all killed and live mutants is compared. Based on the data of killed and live mutants, assessment of proposed operators in terms of mutation score ratio is performed.

Chapter 5

Results and Discussion

In this section, we have discussed process used for analysis of experiments result, which we have performed on different Unity 3D C# games both 3D and 2D source code. Using existing and proposed Unity 3D C# mutation operators, we generated mutants of original source codes and then using dataset we executed 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 mutation operators with existing operators.

5.1 Analysis 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 to test the source program. Test cases are executed for both original program and mutants with the goal that each mutant should produce different output from the original program.

Test cases for the traditional C# mutation operators are executed for the full statement coverage. We expected that mutation testing for Unity 3D will be stronger than statement coverage so we used statement coverage for comparison. The test inputs are designed manually for each game to achieve the 100% statement coverage. All test cases for C# traditional mutation operator for each game were executed against all the mutants generated with traditional and proposed mutation operators. Later, newly designed test cases were executed to detected the faults introduced using newly proposed Unity 3D mutation operators. The input for test cases are provided using the test device with the game running on both device and the Unity editor communicating through the Unity 3D Remote 5 platform. Test cases output is recorded on the Unity 3D editor console.

The process for analysis is as follows:

1. Against each mutation operator selected, mutants are created.
2. After the compilation of code, Unity Remote 5 platform installed on the mobile device is run with the Unity 3D editor to execute test cases on mutants to record output on Unity Console for test cases that are executed.
3. Existing and new test cases are executed for original and mutants by executing on device and output of test cases is recorded on the Unity 3D Editor for comparison.
4. After collecting results for the test cases executed for mutants and original program, results comparison is being performed.
5. If result of the mutant for test case is different from the result of original program, mutant is said to be killed by that specific test case.
6. Analysis of live and killed mutants with the existing and new test cases is done.
7. Mutation score is calculated for the existing and new test cases.
8. Mutation score comparison is done to check the effectiveness of new and existing test cases

The detailed analysis process is explained in the Figure 5.1

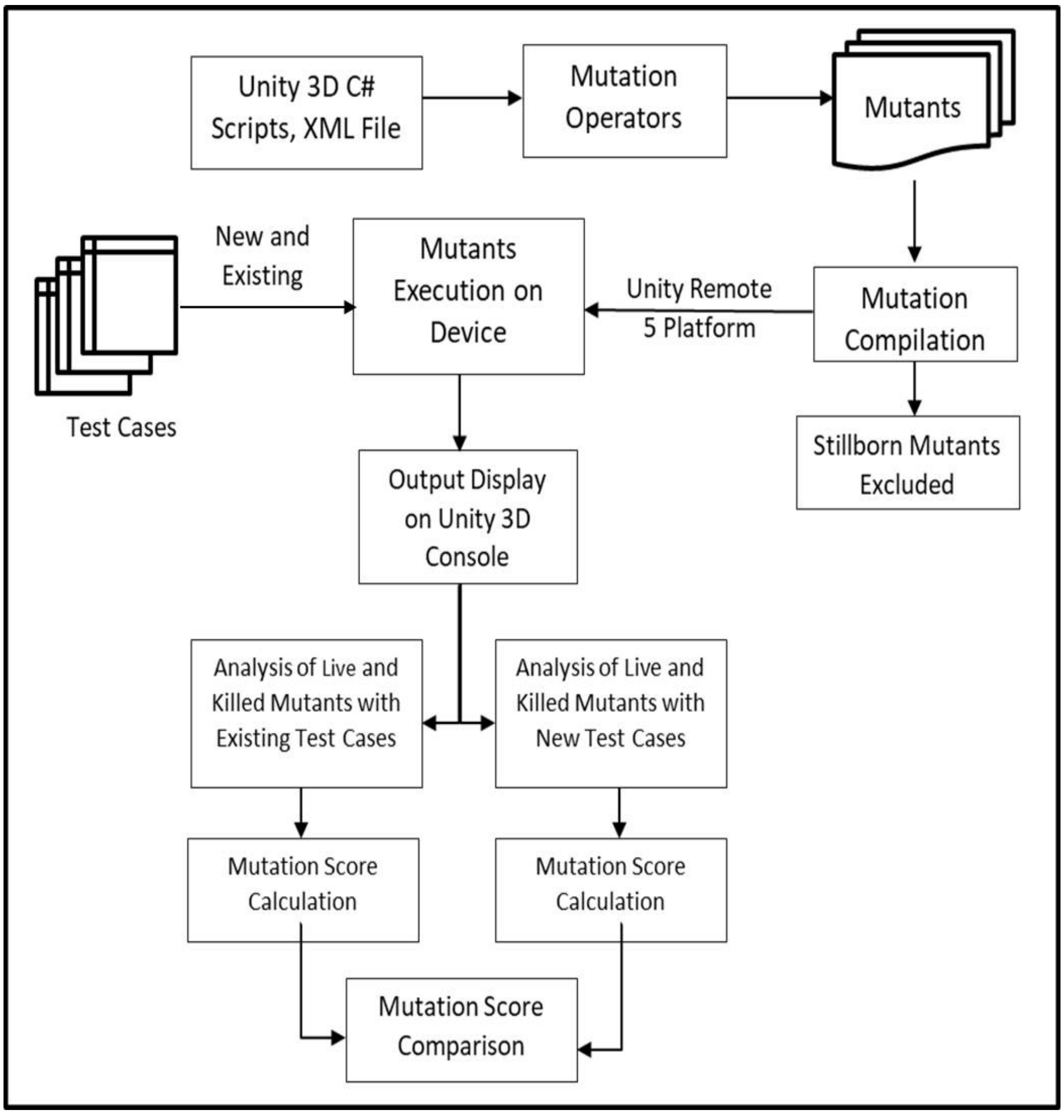

Unity 3D C# Scripts, XML File
Mutation Operators
Mutants
New and Existing
Mutants Execution on Device
Unity Remote 5 Platform
Mutation Compilation
Test Cases
Stillborn Mutants Excluded
Output Display on Unity 3D Console
Analysis of Live and Killed Mutants with Existing Test Cases
Analysis of Live and Killed Mutants with New Test Cases
Mutation Score Calculation
Mutation Score Calculation
Mutation Score Comparison

FIGURE 5.1: Results Analysis Process

5.2 Case Studies

For the evaluation of our proposed Unity 3D C# proposed mutation operators, the developed tool with the new and existing mutation operators was used to generate the mutants, which were compiled, and executed on the real Android device. Test cases were executed against all the mutants and results were recorded.

Total of 4 different Unity 3D games are used for the evaluation of Unity 3D C# proposed mutation operators. Games were selected based on the programming practices and features used along with the availability of source code. All the games selected covers the major games genre of mobile devices. The detail of case studies is as follows:

- Sample Unity 3D C# project was developed with all the C# features to check the mutant generations using the proposed and existing mutation operators. As the sample game project was prepared so it is not available on the Google Store and was only used for experiment. The game was prepared with the 2 different scenes including the 4 main C# scripts with all Unity 3D special programming features for which the mutation operators were proposed.

- Archery is a 2D Unity 3D C# game with the landscape play mode with only 1 game scene. The game is freely available on the Google Play store. According to Google Play game version 3.0.1 is available for the users under the Arcade category which was updated on the store on December 2017 with more than 10,000,000+ installs and with the 3.8 game rating provided by the 127,822 users [41]. The game is developed by the Innovative games developers [42] that are specialized for the 2D games. The game code is freely available from the Unity asset store with the name of Bow and Arrow [43]. Game consists of main 1 scene with the bowAndArrow main script with 473 line of code.

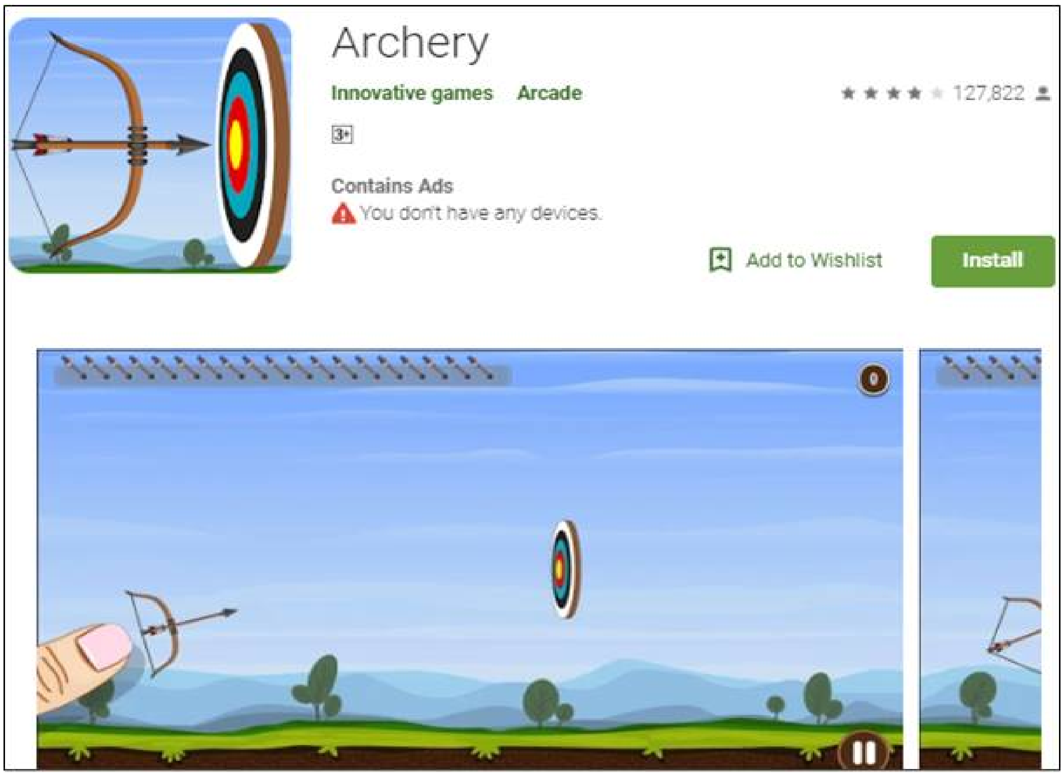
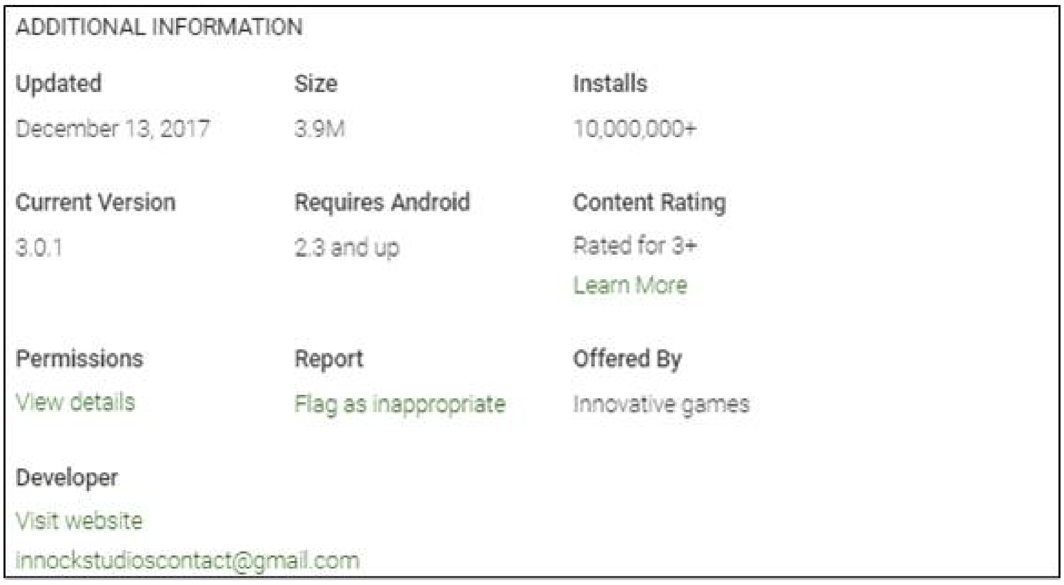


FIGURE 5.2: Archery 2D Game Google Play Store



ADDITIONAL INFORMATION

Updated	Size	Installs
December 13, 2017	3.9M	10,000,000+
Current Version	Requires Android	Content Rating
3.0.1	2.3 and up	Rated for 3+ Learn More
Permissions	Report	Offered By
View details	Flag as inappropriate	Innovative games
Developer		
Visit website innockstudioscontact@gmail.com		

FIGURE 5.3: Archery Game 2D Google Play Store Information

- Tanks is a 3D top down Unity 3D C# single and multiplayer game with the landscape game play mode. Game is freely available on the Google Play store [44]. According to Google Play game version 1.0 is available for the users under the Arcade category which was updated on the store on October

2016 with more than 10,000+ installs and with ranking of 4.1 stars given by 295 users. The game is developed and released by the Unity Technology Aps which is an official Unity 3D Google Play store [45]. The game code is freely available on the Unity asset store provided for the developers as a tutorial code [46]. Game consists of one main scene along with Android Manifest file and multiple game C# scripts to perform the game main functionalities. The main script of game with 180 line of code is GameManager.cs script of the game.

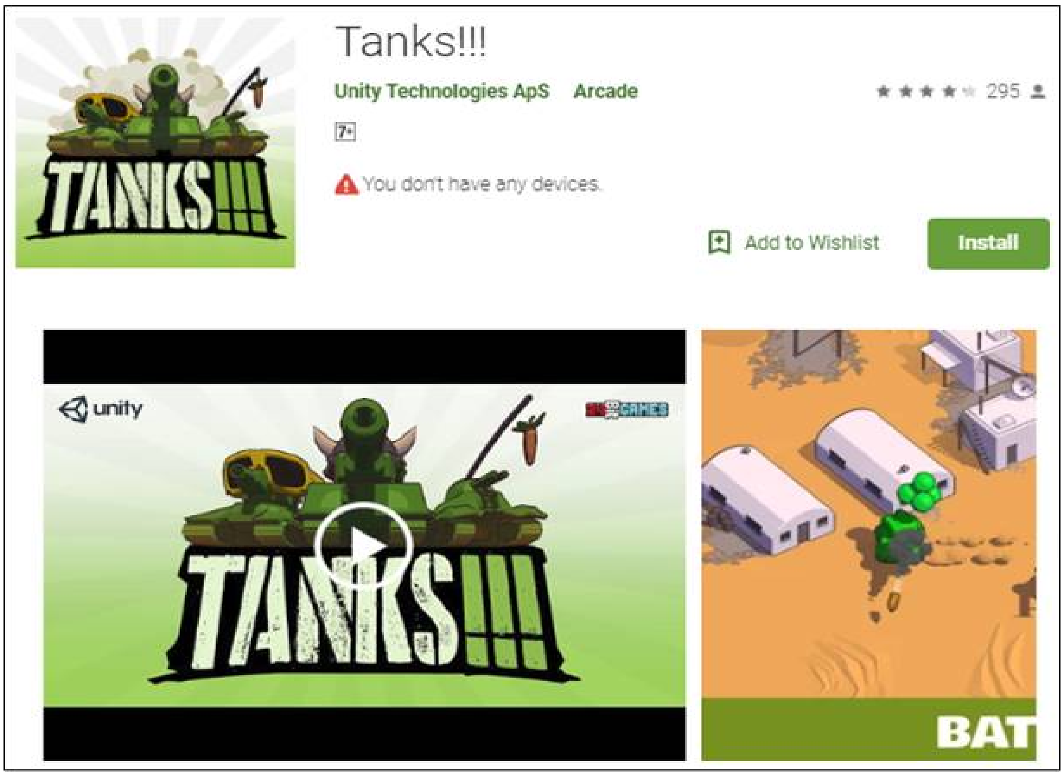
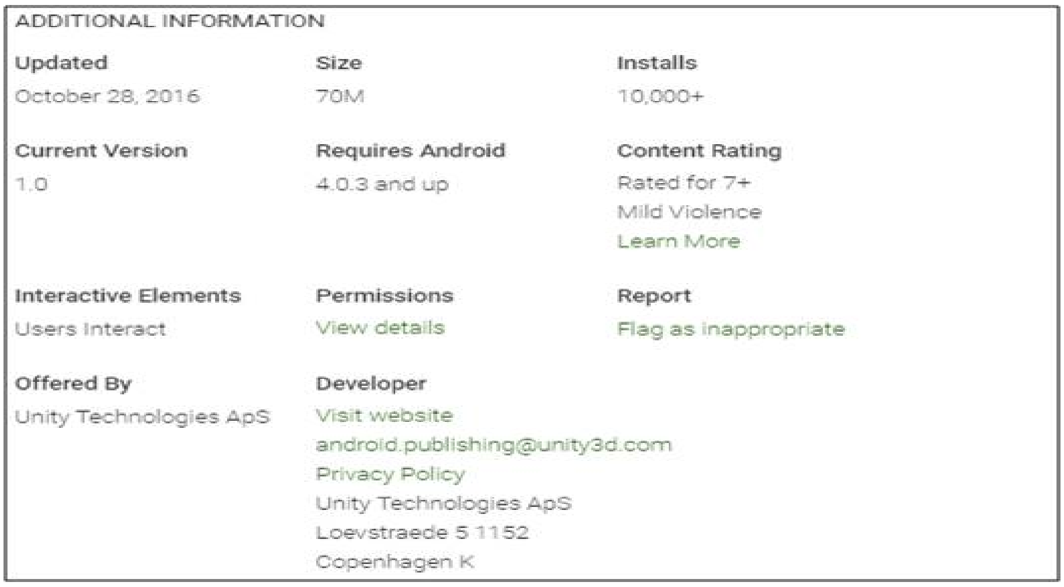


FIGURE 5.4: Tanks 3D Game Google Play Store


ADDITIONAL INFORMATION

Updated	**Size**	**Installs**
October 28, 2016	70M	10,000+
Current Version	**Requires Android**	**Content Rating**
1.0	4.0.3 and up	Rated for 7+ Mild Violence Learn More
Interactive Elements	**Permissions**	**Report**
Users Interact	View details	Flag as inappropriate
Offered By	**Developer**	
Unity Technologies ApS	Visit website android.publishing@unity3d.com Privacy Policy Unity Technologies ApS Loevstraede 5 1152 Copenhagen K	

FIGURE 5.5: Tanks 3D Game Google Play Store Information

- Monster Kill Shooting Adventure is a 3D top down game with Tower Defense game idea developed in Unity 3D with the C# scripting with the landscape game play mode. Game is available on the Google Play store freely for the users. Game is categorized as an Action game on the Google Play store with total of 28 MB apk size [47]. According to Google play game version 1 is available for the users which was last updated by the developers on May, 2018.Game have more than 5000+ installs and with the raking of 3.8 stars on the Google Play store. Game is developed and released by the HalfBrain Games [48]. Game code is provided by the developers for the research work purpose to conduct the experiments. Game main C# file consists of 316 line of code with the name of GameGUI.cs along with other main functionality scripts and an Android Manifest file.

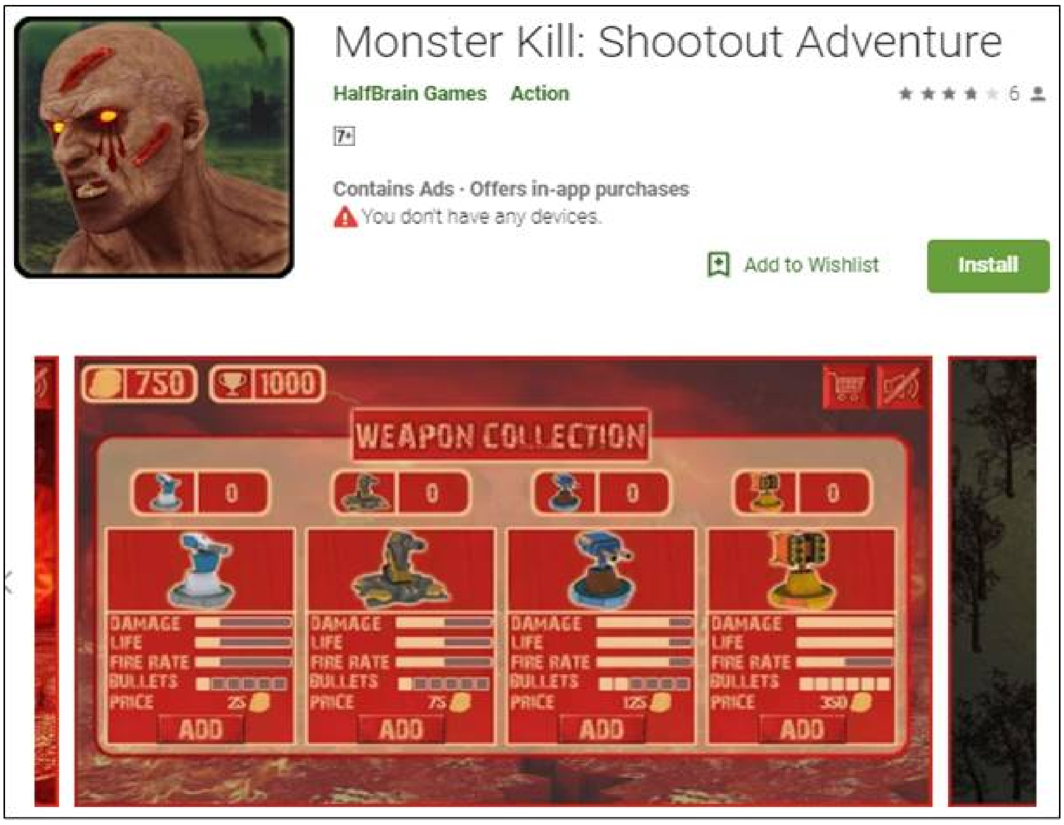
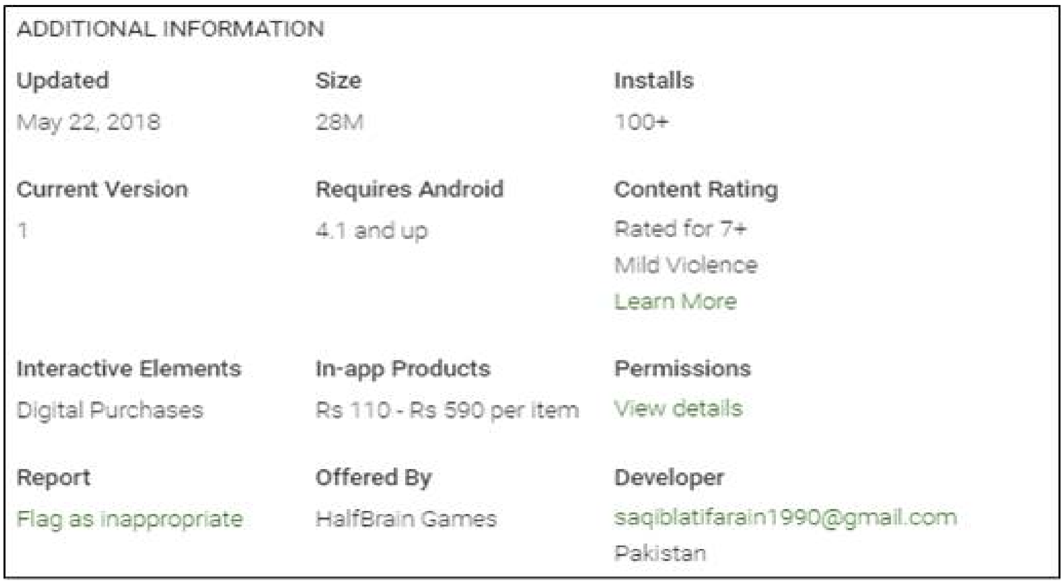


FIGURE 5.6: Monster Kill Shooting Adventure Google Play Store



ADDITIONAL INFORMATION

Updated	Size	Installs
May 22, 2018	28M	100+
Current Version	Requires Android	Content Rating
1	4.1 and up	Rated for 7+ Mild Violence Learn More
Interactive Elements	In-app Products	Permissions
Digital Purchases	Rs 110 - Rs 590 per item	View details
Report	Offered By	Developer
Flag as inappropriate	HalfBrain Games	saqiblatifarain1990@gmail.com Pakistan

FIGURE 5.7: Monster Kill Shooting Game Google Play Store Information

Detail about case studies along with source line of code (SLOC) for C# scripts with main functionality of game is provided in the Table 5.1.

Table 5.1: Details of Unity 3D Games

Game	**File Name**	**Source Lines of Code (SLOC)**
Sample Game	Bullet.cs	49
	GameGUI.cs	115
	Menu.cs	17
	PlayerManager.cs	47
	Game Scenes	2
Archery Game	bowAndArrow.cs	473
	camMovement.cs	31
	AndroidManifest.xml	15
	Game Scene	1
Tanks Game	AndroidManifest.xml	95
	TankMovement.cs	79
	TankHealh.cs	54
	ShellExplosion	31
	CameraControl.cs	104
	GameManager.cs	180
	Game Scene	1
Monster Kill	Player.cs	76
	Enemy.cs	117
	EnemyManager.cs	120
	GameGUI.cs	316
	Turret.cs	147
	TurretManager.cs	80
	AndroidManifest.xml	25
	Game Scenes	4

5.3 Mutants Generation

According to the process of applying mutation analysis for Unity 3D C# games, we used a set traditional mutation operators of C# that are Access Modifier Change (AMC), This Keyword Deletion (JTD), Member Variable initialization Deletion (JID), Method Name Change (MNC) along with the sixteen Unity 3D C# mutation operators proposed in our research to generate the mutants and compile them to be executed on the device using Unity 5 platform.

Traditional mutation operators were selected based on major C# programming features used by the developers in game development. 39 C# traditional and advanced mutation operators were proposed by Dereziska [40]. On mapping the traditional and advanced mutation operators of C# with the programming features used in game development, few mapped C# programming features mutation operators were selected for comparison.

Generating a mutant and compiling them on Unity 3D took the time of almost 5 minutes on HP Windows PC with the 3.30 GHz Intel i5 Processor, 8 GB memory with the dedicated 1 GB DDR AMD Radeon R5 graphics card to support easy and smooth execution of Unity 3D codes.

The totals of 675 mutants were generated using the traditional and proposed mutation operators with the maximum of 230 mutants of the Monster Kill Shooting Adventure 3D game. Mutants were generated for both C# scripts and Android Manifest XML files. A mutant that cannot be compiled into an APK is called the still born mutant and is not counted in the results. No such mutants for any of the case study were made.

Some mutants are not killed by any of the test case which are known as equivalent mutants Out of total 181 mutants generated using the existing mutation operators 53 equivalent mutants are generated while using the proposed mutation operators 55 equivalent mutants were generation from the total of 494 mutants. The detailed analysis of the generated mutants with the existing C# mutation operators and proposed Unity 3D C# mutation operators are described in Table 5.2.

TABLE 5.2: Mutants Generated with Existing and Proposed Mutation Operators

Game	**File**	**C# Mutants**	**Unity 3D Mutants**
Sample	C# Scripts	15	40
	Manifest File	0	0
Archery Game	C# Scripts	48	76
	Manifest File	0	5
Tanks Game	C# Scripts	55	131
	Manifest File	0	7
Monster Kill	C# Scripts	63	230
	Manifest File	0	5
	TOTAL	**181**	**494**

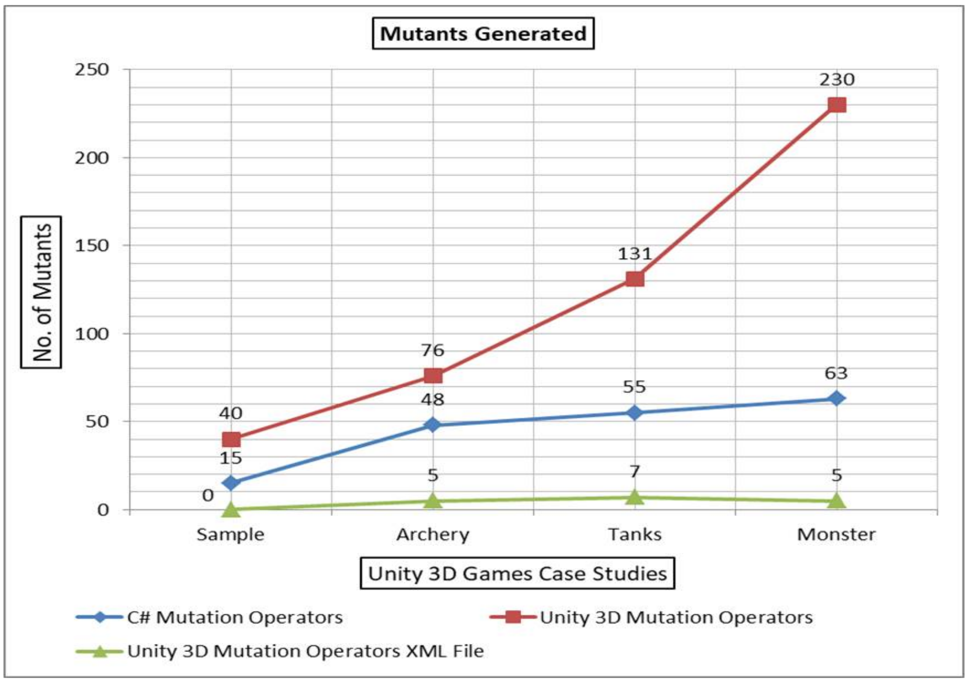


FIGURE 5.8: Mutants Generated Comparison with Traditional and Proposed Mutation Operators

5.4 Comparison and Analysis

We used test cases for the Unity 3D C# games which were generated by hand. Test cases were made manually to achieve the full statement coverage for all the traditional mutation operators. Test inputs to achieve the 100% statement coverage for the designed test cases were provided manually from the touch events while executing the game on real mobile device attached with the Unity 3D editor using the Unity Remote 5 platform to record the output of test case with the provided input.

For all case studies used for the experiment, 128 of 181 C# traditional mutants and 128 of 494 Unity 3D C# mutants were killed by the traditional mutation operators test cases. Equivalent mutants were identified by manual analysis.

After executing mutants with the test cases for the mutants generated with the existing mutation operators and with the proposed mutation operators the results are recorded and evaluated. Mutation score is calculated with the number of killed mutants.

Table 5.3 shows mutation scores after the equivalent mutants are filtered out. Results display the percentage that how many mutants are killed. The mutation score for the C# traditional mutants is 1.00 for all the case studies with the mean of 1.00 and a median of 1.00 showing that all the mutants are killed by the test cases designed for the traditional mutation operators. For the proposed Unity 3D C# mutation operators, mutation scores ranged from 0.19 (in Monster Kill game) to 0.47 (in Archery game) with a mean of 0.30 and a median of 0.27 excluding the Android Manifest XML files. The mutation score of the mutants killed by the traditional test cases executed on the mutants generated with the Unity 3D C# mutants shows that only mutants generated with the traditional mutation operators are killed for the Unity 3D games and mutants specific to the Unity 3D games special programming features are not killed with the traditional mutation operators which can be seen with the mutation scores of the traditional test cases executed on the Unity 3D mutants. Thus, traditional mutation operators test cases are not sufficient to kill the mutants generated with the new mutation operators specific to Unity 3D special programming features and new test cases are required to kill the mutants generated with new mutation operators.

Table 5.3: Mutation Analysis Results for C# and Unity 3D C# Mutation Operators with Traditional C# Test Cases

Game	**Files**	**C# Mutants**				**Unity 3D Mutants**			
		Total	**Killed**	**Equi**	**MS**	**Total**	**Killed**	**Equi**	**MS**
Sample	C#	15	9	6	1.00	40	9	6	0.23
	XML	N/A				0	0	0	0.00
Archery	C#	48	36	12	1.00	76	36	12	0.47
	XML	N/A				5	0	0	0.00
Tank	C#	55	40	15	1.00	131	40	15	0.31
	XML	N/A				7	0	0	0.00
Monster	C#	63	43	20	1.00	230	43	20	0.19
	XML	N/A				5	0	0	0.0s0
	Total	181	128	53		494	128	53	
	Median	51.5	32	13.5	1.00	76	38	13.5	0.27
	Mean	45.25	32	13.25	1.00	118.22	32.67	13.28	0.30

Experiment shows that the existing test cases used to kill the mutants generated from the C# traditional mutation operators for all the case studies were not sufficient as very low number of mutants were killed which were generated using the proposed mutation operators. Thus, the Unity 3D C# special programming features mutants require additional test cases to detect the new faults seeded.

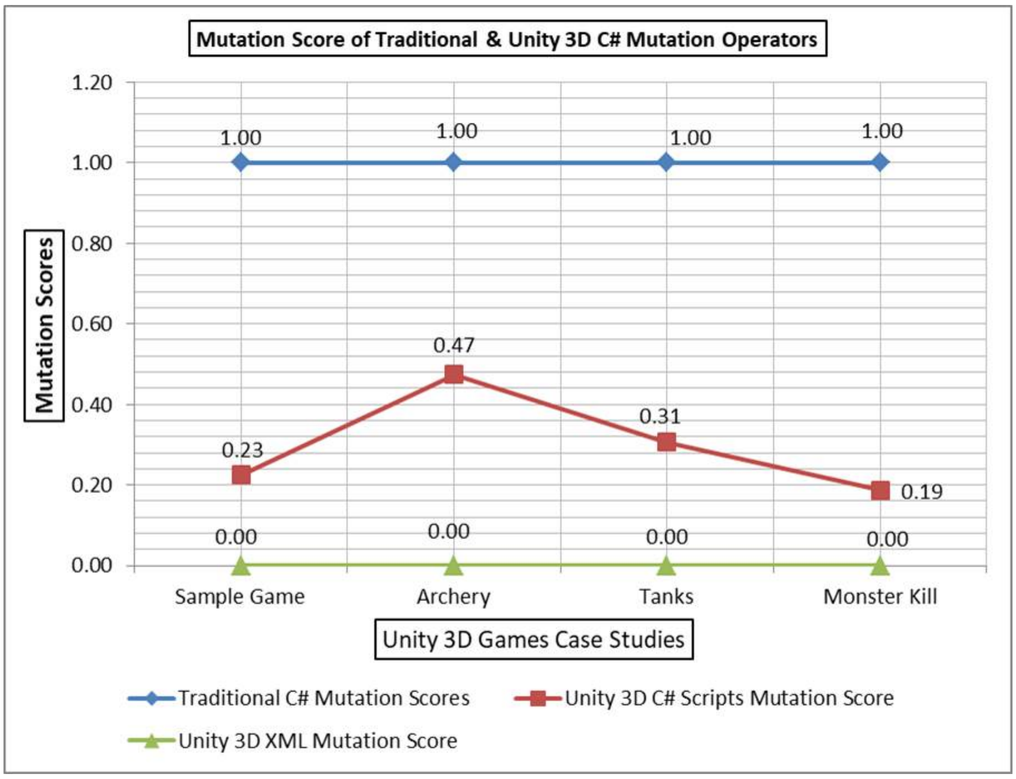


FIGURE 5.9: Mutation Scores of Traditional C# Test Cases for Existing and Proposed Operators

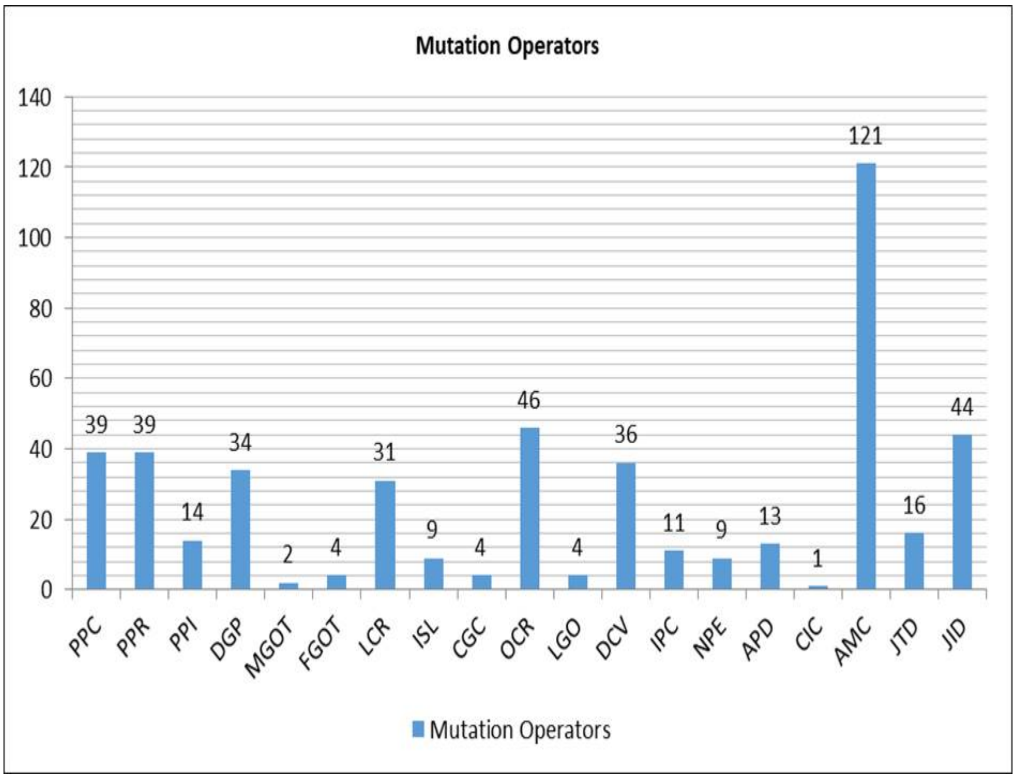


FIGURE 5.10: Total Number of Mutants Generated with Traditional and Proposed Mutation Operators

With the new proposed mutation operators, the mutants generated are strong and can not be killed with the traditional C# test cases. To kill the mutants generated with the proposed mutation operators new additional test cases were designed which are generated using Def-Use (DU) Path coverage to kill the mutants as for multiple features used for Unity 3D, most of the special programming features are not used in same function or script in which they are defined. Test cases were generated and input is provided for test cases from the device such that complete DU Path is executed and the fault introduced is detected by recording the output of mutant program and comparing it with original program output.

128 of 181 C# traditional mutants and 437 of 494 Unity 3D C# mutants were killed by the DU Path coverage test cases. Equivalent mutants were identified by manual analysis. Mutation score is calculated with the number of killed mutants. Table 5.4 display the number of C# traditional and proposed mutants killed with new test cases designed using DU Path coverage criteria.

Table 5.4: Mutation Analysis Results for C# and Unity 3D C# Mutation Operators with New Test Cases

Game	**Files**	**C# Mutants**				**Unity 3D Mutants**			
		Total	**Killed**	**Equi**	**MS**	**Total**	**Killed**	**Equi**	**MS**
Sample	C#	15	9	6	1.00	40	34	6	1.00
	XML	N/A				0	0	0	0.00
Archery	C#	48	36	12	1.00	76	64	12	1.00
	XML	N/A				5	4	1	1.00
Tank	C#	55	40	15	1.00	131	116	15	1.00
	XML	N/A				7	5	2	1.00
Monster	C#	63	43	20	1.00	230	210	20	1.00
	XML	N/A				5	4	1	1.00
	Total	181	128	53		494	437	57	
	Median	51.5	38	13.5	1.00	76	64	12	1.00
	Mean	45.25	32	13.25	1.00	118.22	104.22	14.00	1.00

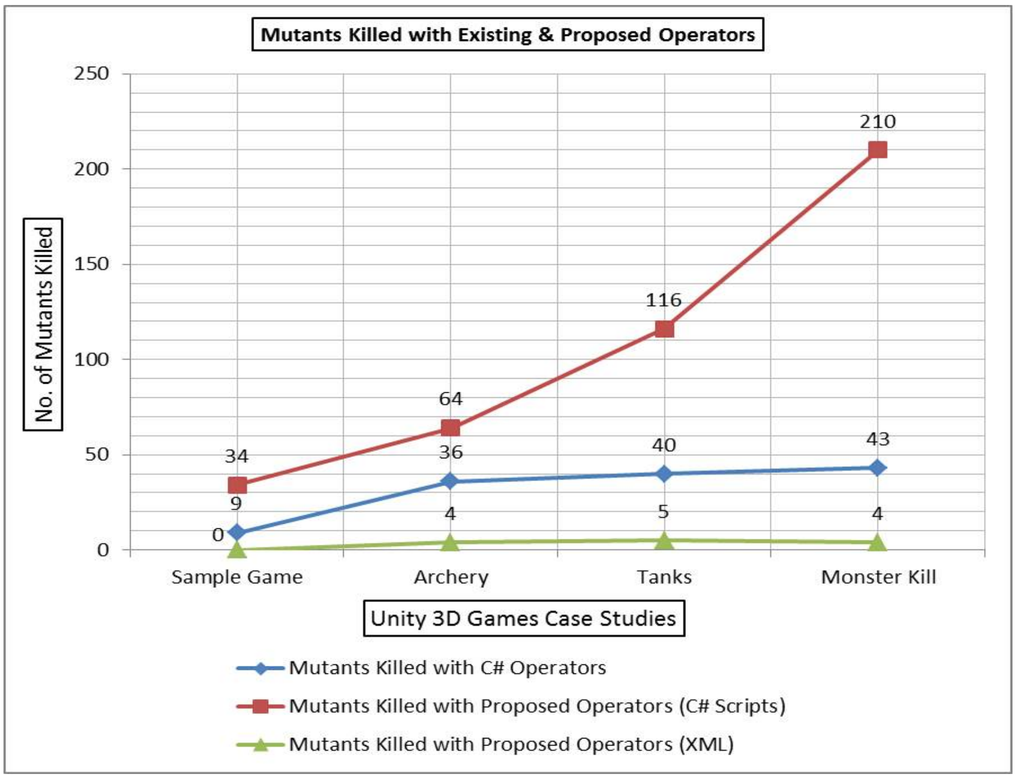


FIGURE 5.11: Total Number of Mutants Killed Generated with Traditional and Proposed Mutation Operators with New Test Cases

By the comparison of mutation scores it can be concluded that the faults introduced by the proposed mutation operators are not seeded with existing C# traditional mutation operators thus making more strong and Unity 3D special programming features mutants. Our proposed mutation operators are useful as they introduce the Unity C# special programming features faults in the program that are currently not seeded by the existing traditional mutation operators. For the detection of such faults, additional test cases are required. By combining our proposed mutation operators with the existing mutation operators of C#, assessment of the test cases can be measured in a better way.

Through the literature review and experimentation of different case studies for our proposed solution the following research questions can be answered.

RQ. 1: *Can we use the traditional C# programming language and Android mutation operators for Unity 3D?*

Through literature review it was observed that the work is only done for the Android mobile applications special programming features mutation testing and no mutation operators exist which can cover the special programming features for the Unity 3D C#. Traditional C# mutation operators can be used for the general C# programming features mutation but cannot be used for the Unity 3D special programming features mutation. While Android specific mutation operators are used for native android application programming features which are JAVA based and can not be used to detect Unity 3D C# specific programming features.

RQ. 2.1: *What are the special features of the Unity 3D C# programming language which are not covered by the traditional C# and Android mutation operators?*

Special programming features for Unity 3D C# includes scenes creation in Unity editor, game objects initialization, physics implementation, animations, handling input touch events, game logic, scenes rendering, GUI rendering, coroutines, lifecycle methods handling, decommissioning, backend and local services, permissions, software development kit version compatibility. Traditional C# mutation operators can be used for the general C# programming features mutation but cannot be used for the Unity 3D special programming features mutation, as well as Android special programming features using JAVA language so they cannot be used for Unity 3D C# programming language features.

RQ. 2.2: *How to design new mutation operators for the Unity 3D C# special programming features?*

Game development is studied and discussed along and Unity 3D features categories. Using the Unity 3D features categories developed for the Unity 3D, some of the basics and general programming features are selected which are used in 3D and 2D game development using Unity 3D game engine for proposing new mutation operators to seed the faults for specific Unity 3D programming features not covered by the tradition C# mutation operators.

RQ. 3: *How effective are the new proposed mutation operators?*

A number of C# mutation operators have been proposed in the literature that can be used to introduce the faults in Unity 3D C# programs basic syntax. Traditional mutation operators of C# are used to mutate very basic features of the C# program for the .NET development. As Unity 3D game development uses C# programming language for the game coding with new special programming features introduced for game development which is not covered by the traditional mutation operators. Using those traditional and new mutation operators the experiments are performed for the multiple case studies and the test cases are executed.

Through detailed experimentation it is found that by executing the traditional C# test cases for mutants, mutation score of test cases is very low for mutants generated with proposed mutation operators as explained in Table 5.3. Through the results it can be concluded that mutants generated with new mutation operators are difficult to kill with test cases designed to kill the mutants generated with traditional C# mutation operators.

So, new test cases with the stronger coverage criterion are required to kill the mutants specific to Unity 3D special programming features generated with new proposed Unity 3D C# mutation operators.

Chapter 6

Conclusion and Future Work

In this research work, we have proposed an innovative approach to test Unity 3D C# games by using the mutation analysis. We have defined new mutation operators specific to Unity 3D games using the C# programming language for the game development which covers the unique characteristics of Unity 3D. Tool is implemented to generate the mutants using proposed mutation operators. Experiments are conducted on the 4 Unity 3D games of different genre and mutation analysis is performed. Results from the experiments show that mutation testing can be enhanced by taking in to account new programming features used for Unity 3D games development. Our approach provides more comprehensive testing for Unity 3D games by taking in to consideration the special Unity 3D C# programming features, game configuration or Manifest file along with the traditional C# features as well.

Our proposed mutation operators introduced diverse faults in the Unity 3D C# scripts and XML files that existing operators are unable to seed so it would not be wrong to say that existing C# traditional mutation operators do not subsume proposed mutation operators.

We performed experiments on different real world case studies and the results indicate that mutation score for test cases designed to detect faults introduced with traditional mutation operators are unable to kill mutants generated with new proposed mutation operators. Experiments results also indicates that with new mutation operators strong mutants are designed which covers Unity 3D C# special programming features. To kill mutants generated with proposed mutation operators new additional test cases are required with stronger coverage criterion.

To kill mutants with proposed mutation operators large amount of additional test cases are required with stronger coverage criterion to detect the faults introduced with proposed mutation operators. During experiments 50% to 75% new test cases were generated to detect Unity 3D C# faults introduced with new proposed mutation operators.

Some steps for the mutation testing for the Unity 3D C# are done manually and also the cost of mutation testing for the Unity 3D games is expensive due to the test execution and recording of the test cases results. Excessive time is required to perform the experiment for a single small scale case study. Some of the manual steps of the mutation testing for Unity 3D C# can be automated and performance can be improved, as well as there are many aspects of the Unity 3D game development which are not yet considered which includes the features used for the server based games, game services, real time multiple player game coding features. So, work on additional mutation operator can also be done.

Bibliography

[1] M. A. Jamil, M. Arif, N. S. A. Abubakar, and A. Ahmad, "Software testing techniques: A literature review," in *Information and Communication Technology for The Muslim World (ICT4M), 2016 6th International Conference on*. IEEE, 2016, pp. 177–182.

[2] F. Redmill, "Theory and practice of risk-based testing," *Software Testing, Verification and Reliability*, vol. 15, no. 1, pp. 3–20, 2005.

[3] M. Grindal, J. Offutt, and S. F. Andler, "Combination testing strategies: a survey," *Software Testing, Verification and Reliability*, vol. 15, no. 3, pp. 167–199, 2005.

[4] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing*, 3rd ed., 2011.

[5] M. Jorgensen and M. Shepperd, "A systematic review of software development cost estimation studies," *IEEE Transactions on Software Engineering*, vol. 33, no. 1, pp. 33–53, 2007.

[6] E. Dustin, *Effective Software Testing: 50 Ways to Improve Your Software Testing*, 1st ed. Addison-Wesley Longman Publishing Co., Inc., 2002.

[7] M. E. Khan, F. Khan *et al.*, "A comparative study of white box, black box and grey box testing techniques," *International Journal of Advanced Computer Science and Applications*, vol. 3, no. 6, pp. 12–15, 2012.

[8] B. Korel, "Automated software test data generation," *IEEE Transactions on Software Engineering*, vol. 16, no. 8, pp. 870–879, 1990.

[9] I. Jacobson, *The unified software development process*, 1st ed. Pearson Education India, 1999.

[10] M. R. Woodward, “Mutation testing—its origin and evolution,” *Information and Software Technology*, vol. 35, no. 3, pp. 163–169, 1993.

[11] A. J. Offutt, R. P. Pargas, S. V. Fichter, and P. K. Khambekar, “Mutation testing of software using a mimd computer,” in *1992 International Conference on Parallel Processing.* Citeseer, 1992, pp. 257–266.

[12] L. Deng, J. Offutt, P. Ammann, and N. Mirzaei, “Mutation operators for testing android apps,” *Information and Software Technology*, vol. 81, pp. 154–168, 2017.

[13] W. E. Wong and A. P. Mathur, “Reducing the cost of mutation testing: An empirical study,” *Journal of Systems and Software*, vol. 31, no. 3, pp. 185–196, 1995.

[14] Y. Jia and M. Harman, “An analysis and survey of the development of mutation testing,” *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011.

[15] C. Byoungju and A. P. Mathur, “High-performance mutation testing,” *Journal of Systems and Software*, vol. 20, no. 2, pp. 135–152, 1993.

[16] A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward, “Mutation analysis.” Georgia Inst. of Tech. Atlanta School of Information and Computer Science, Tech. Rep., 1979.

[17] M. Woodward and K. Halewood, “From weak to strong, dead or alive? an analysis of some mutation testing issues,” in *Software Testing, Verification, and Analysis, 1988., Proceedings of the Second Workshop on.* IEEE, 1988, pp. 152–158.

[18] P. Ammann and J. Offutt, *Introduction to software testing*, 2nd ed. Cambridge University Press, 2016.

[19] A. J. Offutt and R. H. Untch, "Mutation 2000: Uniting the orthogonal," in *Mutation testing for the new century.* Springer, 2001, pp. 34–44.

[20] M. E. Delamaro, J. Maidonado, and A. P. Mathur, "Interface mutation: An approach for integration testing," *IEEE Transactions on Software Engineering*, vol. 27, no. 3, pp. 228–247, 2001.

[21] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur, "Integration testing using interface mutation," in *Software Reliability Engineering, 1996. Proceedings., Seventh International Symposium on.* IEEE, 1996, pp. 112–121.

[22] K. N. King and A. J. Offutt, "A fortran language system for mutation-based software testing," *Software: Practice and Experience*, vol. 21, no. 7, pp. 685–718, 1991.

[23] M. E. Delamaro, J. C. Maldonado, and A. Mathur, "Proteum-a tool for the assessment of test adequacy for c programs user's guide," in *PCS*, vol. 96, 1996, pp. 79–95.

[24] J. Offutt, Y.-S. Ma, and Y.-R. Kwon, "The class-level mutants of mujava," in *Proceedings of the 2006 International Workshop on Automation of Software Test.* ACM, 2006, pp. 78–84.

[25] S. Kim, J. A. Clark, and J. A. McDermid, "Investigating the effectiveness of object-oriented strategies with the mutation method," in *Mutation Testing for the New Century.* Springer, 2001, pp. 4–4.

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

[27] O. A. L. Lemos, F. C. Ferrari, P. C. Masiero, and C. V. Lopes, "Testing aspect-oriented programming pointcut descriptors," in *Proceedings of the 2nd workshop on Testing aspect-oriented programs.* ACM, 2006, pp. 33–38.

[28] U. Praphamontripong and J. Offutt, "Applying mutation testing to web applications," in *Software Testing, Verification, and Validation Workshops*

(ICSTW), 2010 Third International Conference on. IEEE, 2010, pp. 132–141.

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

[30] R. M. Hierons and M. G. Merayo, "Mutation testing from probabilistic finite state machines," pp. 141–150, 2007.

[31] M. Trakhtenbrot, "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, 2007, pp. 151–160.

[32] S. Fabbri, J. Maldonado, P. Masiero, M. Delamaro, and E. Wong, "Mutation analisys applied to validate specifications based on petri nets," in *Proceeding of the 8th IFIP Conference on Formal Descriptions Techniques for Distribute Systems and Communication Protocols*, pp. 329–337.

[33] R. Nilsson, J. Offutt, and J. Mellin, "Test case generation for mutation-based testing of timeliness," *Electronic Notes in Theoretical Computer Science*, vol. 164, no. 4, pp. 97–114, 2006.

[34] B. Lindström, S. F. Andler, J. Offutt, P. Pettersson, and D. Sundmark, "Mutating aspect-oriented models to test cross-cutting concerns," in *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on.* IEEE, 2015, pp. 1–10.

[35] R. A. Oliveira, E. Alégroth, Z. Gao, and A. Memon, "Definition and evaluation of mutation operators for gui-level mutation analysis," in *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on.* IEEE, 2015, pp. 1–10.

[36] A. J. Offutt and J. Pan, “Automatically detecting equivalent mutants and infeasible paths,” *Software testing, verification and reliability*, vol. 7, no. 3, pp. 165–192, 1997.

[37] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, “The oracle problem in software testing: A survey,” *IEEE transactions on software engineering*, vol. 41, no. 5, pp. 507–525, 2015.

[38] Y. Jia and M. Harman, “Higher order mutation testing,” *Information and Software Technology*, vol. 51, no. 10, pp. 1379–1393, 2009.

[39] M. Linares-Vásquez, G. Bavota, M. Tufano, K. Moran, M. Di Penta, C. Vendome, C. Bernal-Cárdenas, and D. Poshyvanyk, “Enabling mutation testing for android apps,” in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering.* ACM, 2017, pp. 233–244.

[40] A. Derezińska, “Advanced mutation operators applicable in c# programs,” in *Software Engineering Techniques: Design for Quality.* Springer, 2006, pp. 283–288.

[41] InnovativeGames, “Archery,” (Last Accessed July, 2018). [Online]. Available: https://play.google.com/store/apps/details?id=com.innovativeGames.archery

[42] GooglePlay, “Innovative,” (Last Accessed July, 2018). [Online]. Available: https://play.google.com/store/apps/developer?id=Innovative+games

[43] UnityAssetStore, “Bow,” (Last Accessed July, 2018). [Online]. Available: https://assetstore.unity.com/packages/templates/bow-arrow-32783

[44] UnityTechnologies, “Tanks,” (Last Accessed July, 2018). [Online]. Available: https://play.google.com/store/apps/details?id=com.unity3d.tanksiii

[45] GooglePlay, “Unity,” (Last Accessed July, 2018). [Online]. Available: https://play.google.com/store/apps/developer?id=Unity+Technologies+ApS

[46] UnityAssetStore, "Tutorial," (Last Accessed July, 2018). [Online]. Available: https://assetstore.unity.com/packages/essentials/tutorial-projects/tanks-reference-project-80165

[47] HalfBrainGames, "Monster," (Last Accessed July, 2018). [Online]. Available: https://play.google.com/store/apps/details?id=com.halfbrain.monster.kill.shootout.adventure

[48] GooglePlay, "Halfbrain," (Last Accessed July, 2018). [Online]. Available: https://play.google.com/store/apps/developer?id=HalfBrain+Games