An Automated Model Based Testing Approach for Platform Games

Sidra Iftikhar
Quest Lab, National University of Computer and Emerging Sciences (FAST-NU), Islamabad, Pakistan
sidra.iftikhar@questlab.pk

Muhammad Zohaib Iqbal
Quest Lab, National University of Computer and Emerging Sciences (FAST-NU), Islamabad, Pakistan
Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg, Luxembourg
zohaib.iqbal@nu.edu.pk

Muhammad Uzair Khan
Quest Lab, National University of Computer and Emerging Sciences (FAST-NU), Islamabad, Pakistan
uzair.khan@nu.edu.pk

Wardah Mahmood
Quest Lab, National University of Computer and Emerging Sciences (FAST-NU), Islamabad, Pakistan
wardah.mahmood@questlab.pk

***Abstract*—Game development has recently gained a lot of momentum and is now a major software development industry. Platform games are being revived with both their 2D and 3D versions being developed. A major challenge faced by the industry is a lack of automated system-level approaches for game testing. Currently in most game development organizations, games are tested manually or using semi-automated techniques. Such testing techniques do not scale to the industry requirements where more systematic and repeatable approaches are required. In this paper we propose a model-based testing approach for automated black box functional testing of platform games. The paper provides a detailed modeling methodology to support automated system-level game testing. As part of the methodology, we provide guidelines for modeling the platform games for testing using our proposed game test modeling profile. We use domain modeling for representing the game structure and UML state machines for behavioral modeling. We present the details related to automated test case generation, execution, and oracle generation. We demonstrate our model-based testing approach by applying it on two cases studies, a widely referenced and open source implementation of Mario brothers game and an industrial case study of an endless runner game. The proposed approach was able to identify major faults in the open source game implementation. Our results showed that the proposed approach is practical and can be applied successfully on industrial games.**

***Index Terms*—Model based testing (MBT), game testing, black box testing, functional testing, system-level testing, and unified modeling language (UML).**

I. Introduction

Game industry has recently emerged as a major software development industry. The number of games being developed has increased exponentially over the last few years. According to a recent survey [1], in 2013 the gaming industry in US contributed over $21 billion and sold over 160 million games. Major reasons for the increased number of games include the availability of high processing display devices, powerful hand-held devices, mobile games and introduction of new game genres. Games are highly interactive, state-based, and are typically real-time software that the end users can interact with for a long time period [2].

A significant part of the game industry comprises of platform games. Platform games have seen a recent revival with a large number of 2D and 3D versions of such games being introduced for hand-held devices, consoles and personal computers (e.g., *Flappy birds* [3], *Subway Surf game* [4], *Caveman Mario Adventure* [5], *Temple Run* [6]). The basic theme of a platform game is that a character (commonly referred to as an avatar) runs through a series of obstacles defined in the game logic and meanwhile gathers points. The goal is to continue playing in order to complete a level while maximizing the points scored.

A major challenge faced by the gaming industry is the automation of functional system-level testing. A common approach of testing games is by manually playing a large number of potential scenarios that end users may exercise [7]. A major focus of such testing is to ensure the functional correctness of games, for example, identifying the points where the game crashes and checking for correct scoring. Similar to traditional software development, manual testing does not scale up to the requirements of game testing. Current game testing practices are labor intensive and become tedious and monotonous with the passage of time, since the testers are required to play a game (or even the same scenarios) several times to test the various versions. Manual test case generation is a monotonous and error prone task, is not systematic and hence is not scalable. This becomes a major challenge when changes appear frequently.

Semi-automated techniques also exist for game testing [8]. Such techniques use record and replay tools, such as *Replay* for *Team Fortress 2*, for automated test execution. The test engineer still needs to design test cases, select test sequences, generate test data, record test cases, and generate test oracle manually. This becomes a major challenge as the games are frequently modified to enhance the game play. All test cases have to be re-recorded and evaluated manually, which is neither efficient nor scalable. Consequently, there is a need of an approach for testing games that allows automated test case generation, execution, and evaluation.

In this paper, we discuss such an automated model based testing approach for system-level functional testing of platform games. The proposed approach allows for automated test case generation, automated test execution, and automated test oracle (test verdict) generation. We discuss our proposed model-based testing approach (with a focus on modeling the game for testing) and its application on an open-source game variant of the famous Super Mario Brothers game [9], which is also used as running example in the paper. We also present the results of applying our approach on an industrial case study, a game under development by our industrial partner. We propose a detailed methodology, including a modeling profile for game testing and a set of guidelines for modeling the game concepts and the relevant behaviors for testing. Finally, we present a test case generation strategy and show how to evaluate and execute the generated test cases to test a game using the developed testing tool.

The intended users of our approach are functional game testers, who are software engineers, therefore we develop a modeling methodology that uses the well-established software engineering modeling standards (UML and its extensions). We intentionally keep our approach simple that closely relates to the game play experience. For modeling purposes, we use the UML state machine diagrams to model the game behavior required for testing the game and UML class diagram notations to modeling the various game concepts and their relationships. The models are developed from the perspective of an Avatar, which is a key character in the platform games. The proposed modeling profile provides a set of game application related concepts that are required for efficient testing of games. The results of applying the approach on the two case studies show that the proposed approach can be used for automated game testing as a replacement of prevalent manual testing practices. The approach was able to identify two major faults in the open source implementation of the Mario Brothers game. Such automation can result in potentially significant savings in time and efforts.

The rest of the paper is organized as follows: Section II discusses the related work, Section III describes our modeling methodology, Section IV discusses the proposed testing strategy, Section V discusses the case studies, Section 0 discusses the limitations of our approach and finally Section VII concludes the paper.

II. Related Work

A number of strategies are discussed in the literature that test games manually. Most of the manual approaches focus on evaluating usability of games. Choi [10] conducts an experiment to evaluate the effectiveness of usability evaluation for games. Ferre *et al.* [11] propose a number of playability heuristics and evaluate these heuristics using an implemented Olympic game. Diah *et al.* [12] discuss an approach that allows usability testing of games. Desurvire *et al.* [13] propose heuristics for playability evaluation of video, board and computer games. Korhonen and Koivisto [14] propose usability heuristics for evaluation of mobile phone games. Moreno-Ger *et al.* [15] propose a usability testing technique for games. Controlled experiments have also been used to evaluate game usability.

A few approaches discussed in the literature use semi-automated techniques for testing games. Buhl and Gareeboo [8] discuss a smoke testing script-based technique for game testing. Dandey [16] develop a virtual client cell game based on the concept of cells in biology and test for functional conformance against a legacy system. Peterson *et al* [17] present a test script based approach for platform independent game testing for video game testing. Schaefer *et al* [18] propose a model based technique that supports client-server based and event driven games. Zhao *et al* [19] propose a general rule model for mobile games that is used to generate test cases for games running over wireless application protocol (WAP). Smith *et al* [20] propose a state based approach for testing game prototypes, using game sketches. Chan *et al* [21] proposes a technique that finds the unwanted behavior of games using genetic algorithm. Nates *et al* [22] present an approach that combines artificial intelligence and computer vision technology to test games. Thunputtarakul and Kotrajaras [23] proposes a technique that tests artificial intelligence mechanisms that are applied in a game. Cho *et al.* [24] propose a black box scenario-based testing technique that uses the game grammar and game map to test the games. Tillmann *et al.* [25] propose a technique for automated test case generation of online developed games. The approach is implemented as a tool named as pex4fun. Choi *et al.* [26] propose a technique for automated load testing and develop a tool for evaluation. Fernandes [27] presents an approach for load testing of online game servers by using a web-based game interfaces for client simulation. Knizia [28] proposes a technique used to design and test a board-game. The testing is done using a scenario-based approach. Ostrowski and Aroudj [29] present a technique for automated execution of regression tests for video games. All of the above semi-automated testing techniques focus on automated test-case execution and test cases are generated manually, except [21] , [22] , and [27]. Approaches discussed in [21] and [22] use AI techniques to generate test cases. However, they do not automatically generate test oracle, which is an important aspect of completely automating the testing process. Focus of [27] is on load testing of game servers and is not directly related to our proposed work.

To summarize, manual testing is largely discussed in literature and used in practice to perform functional testing, usability testing, and measure the fun factor of games. The semi-automated techniques are also used for functional testing, but they only support automated test case execution and the testers still needs to develop the test cases manually and in most cases evaluate them manually. Manual testing in general is laborious and not scalable. In particular manual testing of games require huge investment of time and resources. Our approach proposes a complete automation of game testing, i.e., automated test case generation, automated test oracle, and automated test execution. Our proposed technique allows for system level black box, functional testing of games, using a model-driven approach. Our approach automates the three major steps of software testing: test case generation, test oracle generation, and test case execution and is the only testing approach that specifically targets platform games, which are an important and popular genre of games.

III. Modeling Methodology for Testing Platform Games

Games are different from traditional software systems since they require a high level of continuous user involvement during their execution. In the context of platform games, an avatar is the main character of the game that has to run through different levels, while avoiding or killing enemies, jumping over obstacles and scoring points. When testing the functionality of the games, the various game play rules need to be taken into account for testing. For this purpose the game testers play the games for long continuous hours to cover all the possible scenarios in a game. Playing games manually is laborious and gets uninteresting with the passage of time, especially when the testers are required to repeatedly play the exact same scenarios for every change in the software.

To automate model-based testing for game applications, we propose to model the conceptual and behavioral details that are to be tested. For this purpose, we provide a detailed modeling methodology in this section, which includes a UML profile for modeling platform games for testing and a set of guidelines for modeling game concepts and behavior for testing purposes. Such a methodology is essential to guide the tester on how to apply the provided profile and notations [30]. The conceptual details of the game are modeled using domain model which is created using UML class diagram notation. The behavioral details of the game are modeled using UML state machines. We selected UML as the modeling notation as it is commonly used and an established modeling standard.

Our testing approach focuses on testing the avatar's behavior by simulating the inputs required for playing games. Therefore, the focus of our modeling approach is on modeling the behavior of avatar. The following sub-sections describe the proposed profile (Fig. 1), the domain model (Fig. 2) and the behavioral model for our case study the Mario brothers game shown in Fig. 3. In Fig. 4 we present the internal view of the run forward state (a sub-state of running state). In case of Mario brothers, the run forward and run backward states have similar functionalities, so we only present the run forward state. We intend to keep our approach simple, easy to model and understand, and closely relating to the game play experience.

A. Modeling Profile for Platform Game Testing

We select UML as the modeling language for modeling platform games for testing. While UML provides a number of diagrams for modeling, these diagrams and models are generic and do not capture the details specific to testing games. Therefore, we extend the existing UML models by creating a profile that caters to the modeling requirements specific to testing of platform games, shown in Fig. 1.

The profile includes both conceptual and behavior details. The stereotypes we define for applying on domain model include *Characters*, *Avatar*, *Player*, *Game*, *Obstacles*, *Collectables*, *Boosters* and *Actions*. *Characters* are the roles in a game that the user simulates for playing the game. *Avatar* is the main character in the platform game. *Player* is the actual user who is playing the game. *Game* is used to specify the platform game that is being modeled. *Obstacles* are the objects that cause hindrance for the avatar in a platform game. *Collectables* are various benefits that the avatar collects during the game play, e.g., coins. *Boosters* are the collectables that enhance the powers of an avatar for a specific time interval. *Actions* are the set of activities performed by the avatar during the game play. These stereotypes can be applied to capture the conceptual details specific for each platform game. The stereotypes for behavioral details are *Jump*, *Running*, *Win*, *Dead*, *Loose*, *Attacked by enemy*, *See obstacle*, *Stationary*, *Pause*, *User generated events*, *Start up* events and *System generated events*.

The profile concepts are significant during test oracle generation, for example, the system generated events are used to evaluate whether the system under test dispatches a given event not. The stereotypes depict the basic behavior of an avatar in a platform game. For example, in case of *Temple Run* game, the avatar runs and jumps to save itself from the enemy by avoiding obstacles and is attacked by the enemy when it becomes stationary. In case of *Subway Surf* game, the avatar sees the obstacle, i.e., the board, and moves downward to avoid the obstacle otherwise the enemy catches the avatar. Similarly for the game *Run Sheeda Run* [31], the avatar dies when it has been attacked by the enemy at least three times. *Loose* and *Win* are the game behaviors that are dependent on the behavior of the avatar.

User-generated events are the events initiated by the user to derive the game behavior. For example the run command, given to the avatar by the user is a user-generated event. The *startup events* are a form of user-generated events that initialize the game play, e.g., selecting the type of player. The *system-generated events* are either time-based events or are the responses to user-generated events that are programmed in the game. For example in case of *Temple Run*, when the avatar falls off the edge and lands on another platform, the landing event is the system-generated event. The *game query events* are the events that can be called to check for various system states and run time values when playing game. This information is necessary for testing purposes. For example, in case of Mario game, when the ghosts hit the avatar, the game checks for the life count of Mario in order to decide the next action to be taken. The discussed stereotypes are applied on UML state machines to model the behavior of an avatar specific to a platform game.

B. Modeling Domain Model

We model the conceptual details, i.e., the game concepts and the relationship of concepts, using a domain model. Domain model is a commonly used model to understand the various concepts of the system and is not restricted to modeling of software entities as suggested by [32].

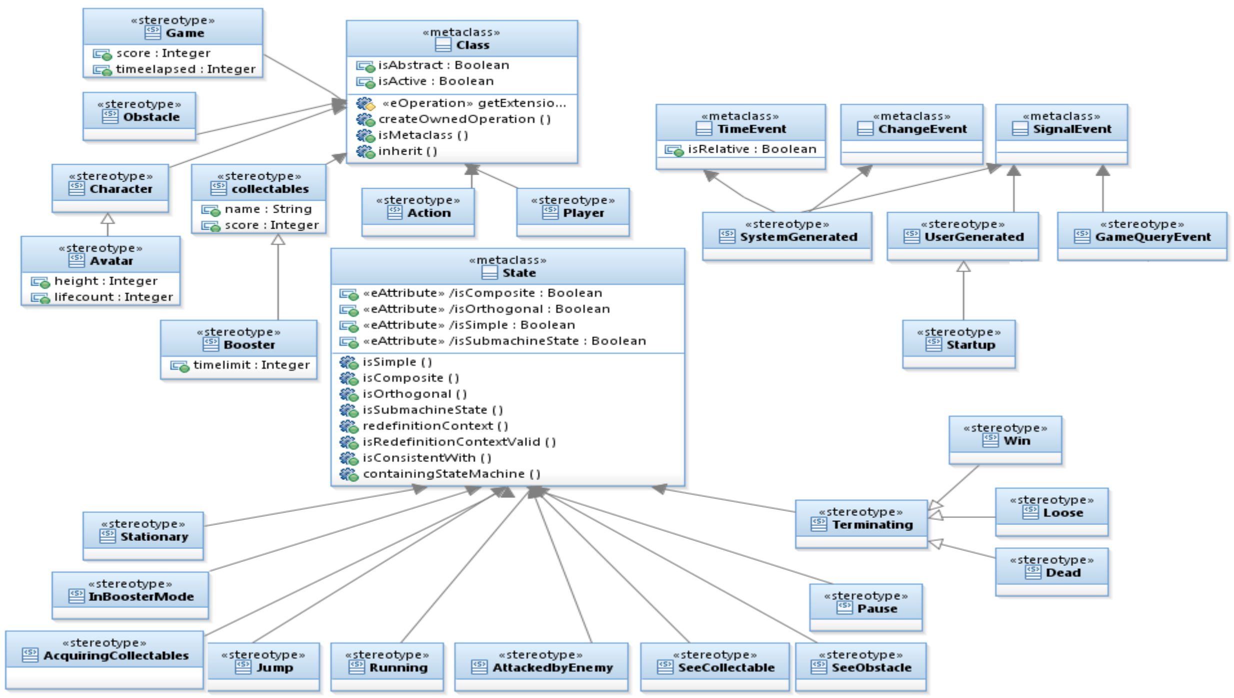


Fig. 1. UML profile for modeling platform games

Modeling of domain model is an important step in understanding the vocabulary of game application that is being modeled for the testers. The behavior of various identified concepts directly impacts the avatar behavior that is modeled in the Avatar state machine. The profile for platform games (Fig. 1) is applied on the domain model. The concepts for platform games provided in the profile are mapped to game specific concepts, for example to the Mario brothers game when modeling it (Fig. 2). Following we explain the identification and modeling of concepts, identification of relationships and identification of properties.

1) Identifying and Modeling Concepts: The first step of developing the domain model is to identify the various concepts that are visible during game play. These can be identified by manually observing the game. The game under test itself is also modeled as a separate concept and is assigned the stereotype «Game». Other important concepts that are to be modeled in the domain model are the concept representing the Avatar role (represented by the «Avatar») and the concept representing the game player («Player»). For example, in the case of Mario Brothers, Mario concept is modeled as an avatar. Other concepts that are modeled in the domain model are the various obstacles and collectables in the platform game and various actions of the avatar. For example, in the Mario brothers game, there are a number of coins that can be collected, therefore we modeled the Coin concept with the «Collectables» stereotype.

2) Identifying Relationships: The relationships between various concepts can be identified by observing the interaction of game concepts during the game play. Generally the concepts of game play area (the actual map on which the avatar is running) are related to the game and the concepts of the storyline are related to the game characters. For example in Mario brothers (Fig. 2), the obstacles are the game play area concepts and they are related to the game, whereas the various actions Mario performs are story-line concepts and are related to Mario. The relationships are modeled using the associations in the UML class diagram notation. The cardinalities corresponding to the relationships are also modeled. For example in Fig. 2, the concept Mario and Run are related with the cardinality of one to many. This means that Mario is a single character that can run several times in a game play. It is important to note is that the constraints and relationships in the meta-model are important for correct modeling of state machines and domain model and hence have an indirect impact on test case generation.

3) Identifying Properties: Concepts in a game can have properties and can be identified by observing the concepts while playing game. For example, during game play the concept avatar has a property life count. The properties corresponding to each concept are modeled as attributes of the concepts. For example in Fig. 2 the score and time elapsed are the properties of the concept Mario brothers' game and life count is the property of the Mario avatar.

C. Behavioral Modeling

We capture the expected behavior of the game using UML state machines. State machines have been widely used in industry for modeling runtime behavior [33, 34]. We model the behavior of the avatar, since avatar is the main character in the game and is simulated by the player. As the primary focus is on the avatar, the various states and actions of the avatar should be tested for functional correctness of the platform game to be tested. As we aim to model the platform games using black box approach, the identification of avatar states and the actions is done by playing the game from the user's perspective. It is important to note that our MBT approach is a conformance testing approach, in which the models are assumed to be correct and the conformance of implementation is evaluated against these models. Any differences found between the model and the implementation indicates a bug in the implementation.

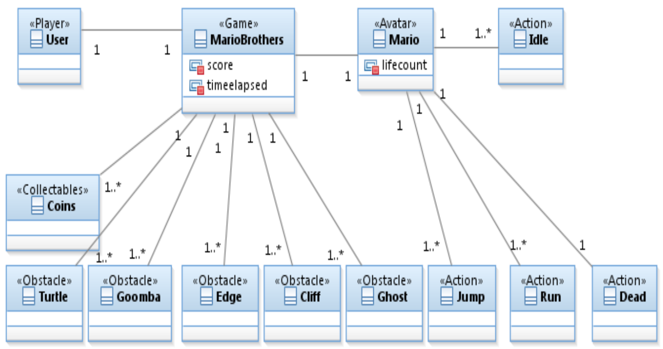


Fig. 2. Mario game domain model.

1) Identifying and Modeling of Avatar States: Avatar is the main character of the game that is typically running through various obstacles in a platform game. The states of the avatar are very important to model as they derive the overall game play. Observing the game play and identifying the visible changes in avatar behavior can help in identifying various states of an avatar. For example, in the case of Mario game, a possible state is that the Mario remains still when the game starts, which can be modeled as an idle state in the state machine. Fig. 3 and Fig. 4 show the state machine developed corresponding to the Mario game.

2) Modeling Event: The second important detail to model is the various events in the game. For our context, i.e., modeling for testing of platform games, we are interested in three types of events: events initiated by the user, the events generated by the game and the game query events.

a) Modeling User-Generated Events: The user-generated events correspond to the events that are initiated by a user during the game. In case of games deployed on a PC, these events are typically initiated via keystrokes or mouse clicks, for example, the user presses the 'up' arrow key to make the player start running. For other device platforms, for example, in the case of android games on mobile devices, these events are initiated via screen swipe or by gyroscope sensor or accelerometer in the device. Since such events are asynchronous, we suggest modeling them as signal events. In case of Mario game, the user-generated events are move forward, move backward and jump.

b) Modeling System-Generated Events: The system-generated events are programmed in the game and are triggered by passage of time or are triggered in response to a user-generated event. The first category, the time bound events, are based on time and take place after a certain time limit has passed in the game. These events can be identified based on the specifications or by observing the behavior of the game. For example, in the scenario where an avatar dies when it stands on the edge of a cliff for more than 10 seconds, the time event that generates the transition to the dead state is a system-generated event. These events are modeled as time events in the state machine. In case of the Mario game, modeled in Fig. 4, Mario dies after 100 seconds when it is in the idle state.

The events in the second category (i.e., the events corresponding to user response) are triggered when a user generates a certain event. Such events can be identified by observing the system's response after every user-generated event. For example, when an avatar in a game hits an object the avatar dies. In this case the dying event of the avatar is generated by the system that leads to the dead state of the avatar. The system-generated events can be modeled using signal events, time events, or change events. In case of Mario game, when a ghost is visible and Mario keeps on running, then the system generates the collide event, which leads to Mario's dead state.

c) Modeling Game Query Events: The game query events are the events that are initiated by the test engine to check the various events of a game. These events are of special use when dealing with time delay issues as they allow the test engine to query about the occurrence of various game events. For example in the scenario when an avatar is expecting a ghost, the arrival of the ghost can be queried from the game. The game query events can be modeled using signal events. For example, for Mario game the game is queried for life count of Mario after getting hit by a ghost.

3) Identifying and Modeling Simultaneous States: An avatar can be in more than one state at the same time in such cases the avatar is said to be in simultaneous states. These states can be identified by observing the avatar for various actions when it is performing one particular action, e.g. while running the avatar can kill a ghost. The simultaneous states can be modeled as orthogonal states. In Mario game, the Mario runs and performs various other actions that lead to various other states while running, e.g., cliff visible state and all the states modeled inside the running state (Fig. 4). While remaining in a state the avatar can check for various events that don't lead to another state of the avatar rather the avatar remains in the same state. For example, while running when an avatar collects coins it increments the score. Events leading to the same state can be modeled using self-transitions. In case of Mario (Fig. 4), if coins are present while Mario is in run forward or run backward state, Mario collects coins and score is incremented.

4) Identifying and Modeling Conditional Situations: Conditional situations occur when there are multiple scenarios that can be reached from a state. These scenarios are typically based on user-generated events. For example when an avatar jumps at the edge in a game map and achieves a certain defined height, the avatar climbs the edge otherwise it falls off and dies. We can model conditional situations using guards (or decision nodes) in the state machine. It is necessary to consider the conditional situations to capture the states and events an avatar can perform. In case of Mario game (Fig. 4), Mario comes in the idle state if a certain height is not achieved when jump event takes place; otherwise Mario climbs the cliff.

5) Identifying and Modeling Parameter Driven Behavior: There are various parameters of the avatar that need to be taken in account, e.g., the speed of the avatar or the height acquired by the avatar when jumping. It is necessary to consider the various parameters of an avatar to model the scenarios that can occur due to different parameter values. The parameters can be modeled using guards. In case of Mario, (Fig. 3) the avatar Mario has a parameter height that is modeled using guards when Mario jumps.

IV. Model-based Testing Strategy

In this section, we discuss the proposed model-based testing strategy for automated game testing. A test case in our context is a sequence of user-generated events and corresponds to a path of the state machine. The user-generated events are obtained by traversing the various paths of the avatar's state machine. The execution of the user-generated events may cause a change in the avatar state machine (by triggering a transition). Note that in this paper, we are focusing on functional testing of games, therefore, we assume that the games will be tested either on emulators (in case of games designed for hand-held devices) or actual desktop machines. Following we discuss the test case generation strategy, test oracle generation, and the test execution related details corresponding to model-based testing approach for Game testing.

A. Test Case Generation

We use the avatar's state machine and the game state machine to generate test cases. As per the modeling guidelines, the two user-generated events in the game state machine that refer to initialization are the 'initialize' and 'play' events. These events represent the game initialization and starting of game play respectively (See Fig. 5).

In our context, these events are the start events for all the test cases and can have complex sequence of events (e.g., configuration steps, user creation), therefore we use a record and replay tool to record the initial set of steps required to start the game play. These recordings correspond to the 'initialize and 'play' events of the game state machines and are replayed every time a new test case is executed. To avoid starting the game repeatedly, the initialize event can be skipped. In this case the reset event will reset the game play to an initial state. To record these events, we developed a desktop-based record and replay tool.

For the avatar state machine, the first step is to generate a corresponding transition tree. For this purpose the avatar state machine is first flattened. We use the algorithm suggested in [35] to generate a transition tree from the flattened avatar state machine. For example, the transition tree corresponding to Mario's avatar is shown in Fig. 6. Our test case generation is based on the N+ strategy proposed by [35]. N+ testing strategy is a well-accepted approach used for generating test cases based on state machines to avoid state explosion problem [35] and involves generation of a transition tree that contains end-to-end paths from the state machine.

For this purpose we traverse the generated transition tree according to round trip transition strategy and sneak path strategy for user-generated events to obtain a test path. The user-generated events in the test paths are abstract events and a mapping to actual concrete events is required before execution of the test cases. The concrete events can be keystrokes or mouse clicks in the case of a PC-based game and taps and swipes for games on hand-held devices. Since, we are assuming that the testing is being done on emulators (a common practice in industry), therefore the mapping from the abstract events in our context will be to key strokes or mouse clicks. This mapping is provided in a simple text file as key-value pairs. Fig. 7 shows an excerpt of the mapping file.

B. Test Oracle

The system-generated events and the state transitions in the state machines (due to a change, signal, or a time event) that are triggered as a result of the user-generated events are considered as test oracles. The information that whether a state has been changed or a system-generated event is triggered is obtained via a testing interface written specifically for this purpose. The interface contains lookup methods that are responsible for providing the details of the internal game states during testing. The interface is to be written by the test engineers. Examples of lookup methods include checking for arrival of obstacles in a game. For the Mario game, the interface written required around 150 lines of code.

During test execution if the system fails to generate a system-generated event corresponding to a user-generated event or the testing interface fails to verify a state change corresponding to a transition or the application crashes, the application will be considered to have failed the testing.

C. Test Execution

The first step in test execution is the execution of initialize and play recordings corresponding to the game state events. After this a test case is selected from the generated test cases, its user-generated events are mapped to concrete events and the test case is executed. The test cases are generated using the N+ strategy that covers scenarios from the initial state to a certain state in a way that all round-trip paths in the state machine and the sneak paths are covered.

The test generation strategy allows round-trip path generations that cover the various paths of the state machines. For testing games, running these paths individually results in very short test cases that fail to exercise the interesting scenarios only visible after long game play. In order to support long and continuous game execution for testing the tester defines a time period for individual test case execution. During this period a random test execution strategy can also be selected that selects various matching test sequences randomly and executes them automatically. The tester can also select the test cases or particular sequences of test cases that should be executed. If at any time during the game execution, system-level events or state change events that are specified in the state machine are not triggered or the game crashes, the situation is considered as a game failure corresponding to the test case being executed.

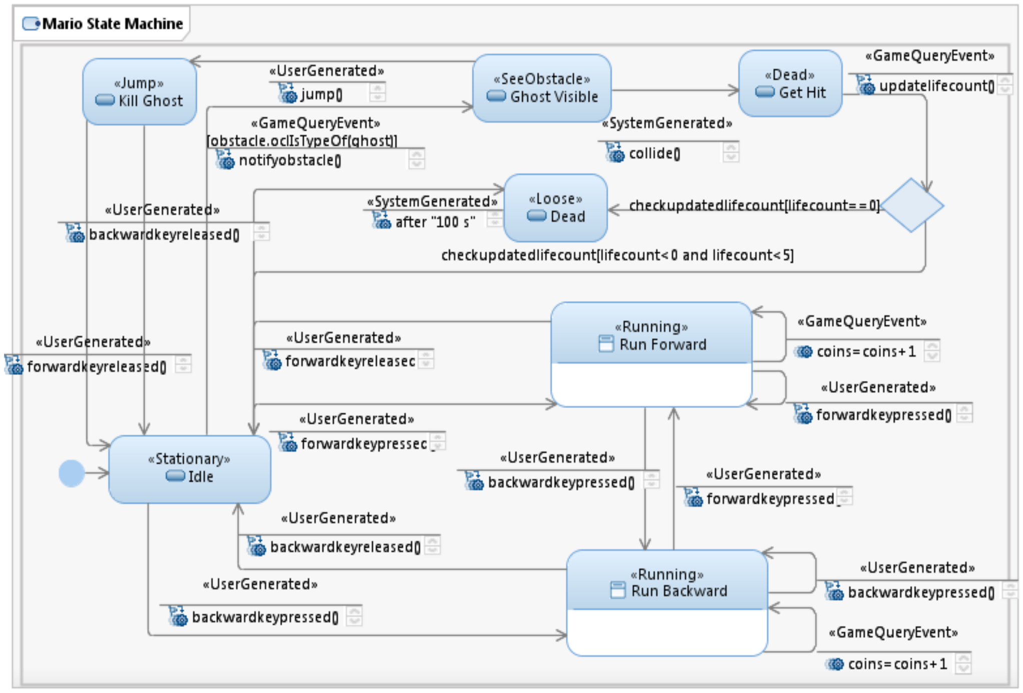

Mario State Machine
«Jump» Kill Ghost
«SeeObstacle» Ghost Visible
«Dead» Get Hit
«UserGenerated» jump()
«GameQueryEvent» [obstacle.oclIsTypeOf(ghost)] notifyobstacle()
«GameQueryEvent» updatelifecount()
«SystemGenerated» collide()
«UserGenerated» backwardkeyreleased()
«SystemGenerated» after "100 s"
«Loose» Dead
checkupdatedlifecount[lifecount==0]
checkupdatedlifecount[lifecount<0 and lifecount<5]
«Running» Run Forward
«GameQueryEvent» coins=coins+1
«UserGenerated» forwardkeyreleased()
«UserGenerated» forwardkeyreleasec
«UserGenerated» forwardkeypressed()
«UserGenerated» forwardkeypressec
«Stationary» Idle
«UserGenerated» backwardkeypressed()
«UserGenerated» forwardkeypressed
«UserGenerated» backwardkeyreleased()
«Running» Run Backward
«UserGenerated» backwardkeypressed()
«GameQueryEvent» coins=coins+1

Fig. 3. Mario's avatar state machine

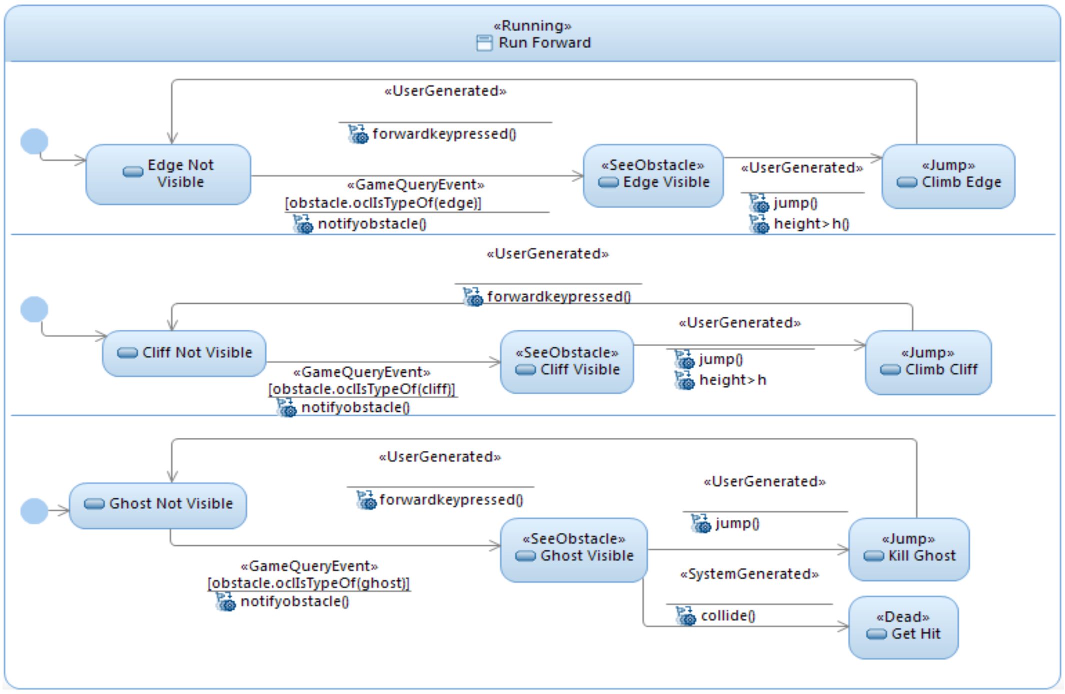

«Running» Run Forward
«UserGenerated» forwardkeypressed()
Edge Not Visible
«GameQueryEvent» [obstacle.oclIsTypeOf(edge)] notifyobstacle()
«SeeObstacle» Edge Visible
«UserGenerated» jump() height>h()
«Jump» Climb Edge
Cliff Not Visible
«GameQueryEvent» [obstacle.oclIsTypeOf(cliff)] notifyobstacle()
«SeeObstacle» Cliff Visible
«UserGenerated» jump() height>h
«Jump» Climb Cliff
Ghost Not Visible
«GameQueryEvent» [obstacle.oclIsTypeOf(ghost)] notifyobstacle()
«SeeObstacle» Ghost Visible
«UserGenerated» jump()
«Jump» Kill Ghost
«SystemGenerated» collide()
«Dead» Get Hit

Fig. 4. Run forward state machine.

V. Case Study

To demonstrate the application of our proposed game testing methodology, we applied it on two case studies. The first case study is of an open source variant of a famous platform game, Super Mario Brothers [9]. The second case study is of an endless runner platform game being developed by our local industry partner.

Mario game was first introduced in 1983 for Nintendo console. Since then, a large number of versions of the game have been introduced. In Mario brothers' game, the avatar moves through various obstacles and collects coins. While moving, Mario dies when it hits a ghost or falls of the edge.

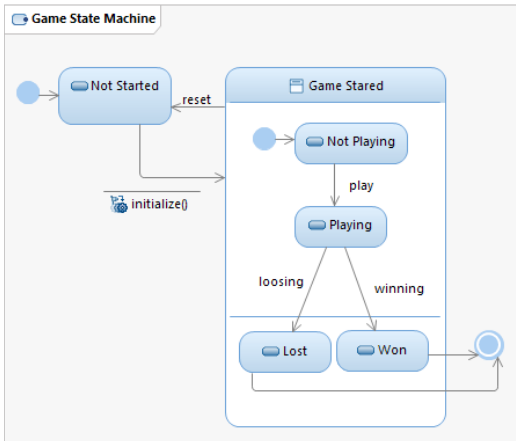


Fig. 5 Mario game state machine

The modeling details for the game were discussed in Section 3. The conceptual model for the game includes various concepts that are present during the game play.

The conceptual model for the game Mario is shown in Fig 2. The two state machines developed for the game include the Avatar state machine (shown in Fig. 3 and Fig. 4) and the Game state machine (Fig. 5). The Avatar state machine was flattened and a transition tree was generated as per the testing methodology. The modeling statistics for Mario are shown in TABLE I. Fig. 8 shows the Automated Platform Game Testing (APGET) tool developed for our approach, while Fig. 9 shows automated execution of test cases on the Mario Bros. The testing strategy generated and executed 313 test cases. The round trip strategy generated 53 test cases based on the paths in the transition tree. For the sneak path strategy corresponding to various user-generated events, a total of 260 test cases were generated.

On execution of the generated test cases, two major faults were triggered and detected. In the first detected fault, the avatar Mario doesn't reach the dead state when it gets hit by a ghost rather it remains in the Run state. The fault was detected while executing the following two test paths according to the round trip strategy:

- Start – Idle – Run Forward – Ghost Not Visible – Ghost Visible – Get Hit – Dead
- Start – Idle – Run Backward – Ghost Not Visible – Ghost Visible – Get Hit – Dead

The second fault deals with the scenario where Mario jumps on the cliff, but the cliffs are unreachable, i.e., even after achieving its maximum height Mario cannot climb the cliff. After manual inspection we concluded that the cliff was too high for Mario to reach during the game. The fault was identified, while testing for the following two sneak paths:

- Start – Idle – Run Forward – Cliff Not Visible – Cliff Visible – Climb Cliff
- Start – Idle – Run Backward – Cliff Not Visible – Cliff Visible – Climb Cliff

We also apply our modeling methodology on a local industrial case study. The testing strategy could not be applied due to unavailability of the interface required for testing (Section IV). The industrial game that we modeled has an avatar that runs endlessly to save its chicken from a butcher. The avatar runs on the roads and switches lanes to save the chicken.

We show the modeling statistics for the industrial game in TABLE I. There were a total of 54 states capturing the behavior of the Avatar and the Game startup. Ten of these states represented orthogonal regions and had their independent state machines.

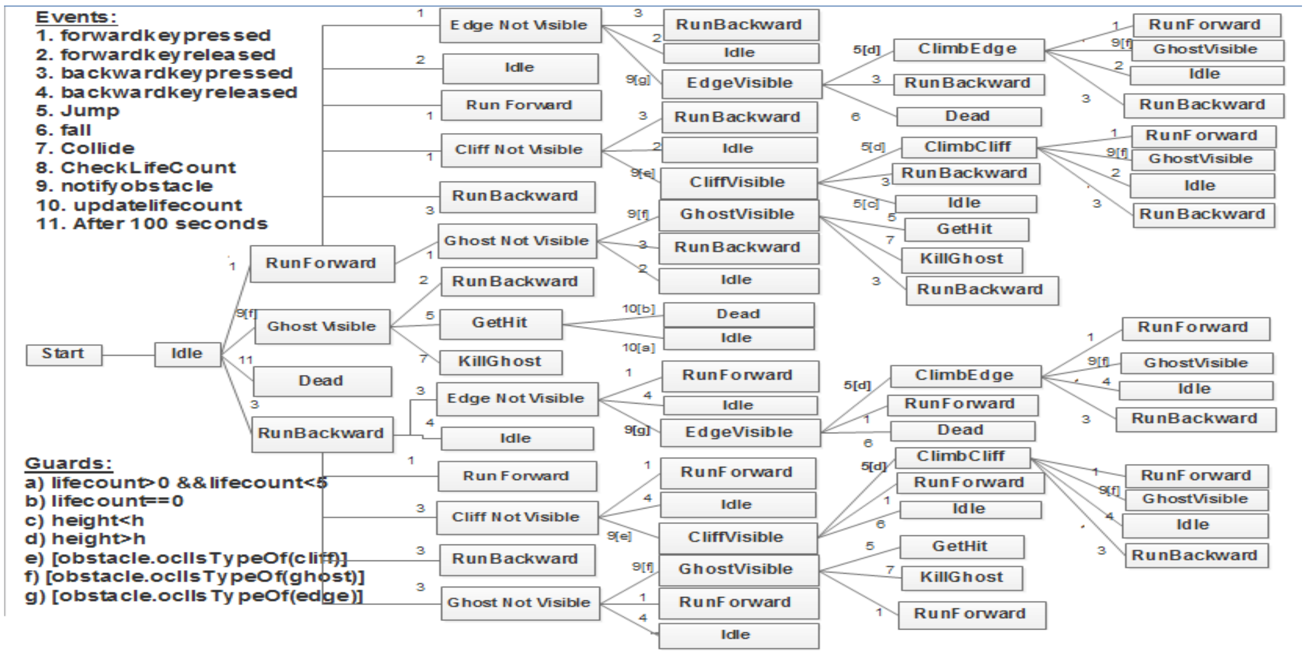


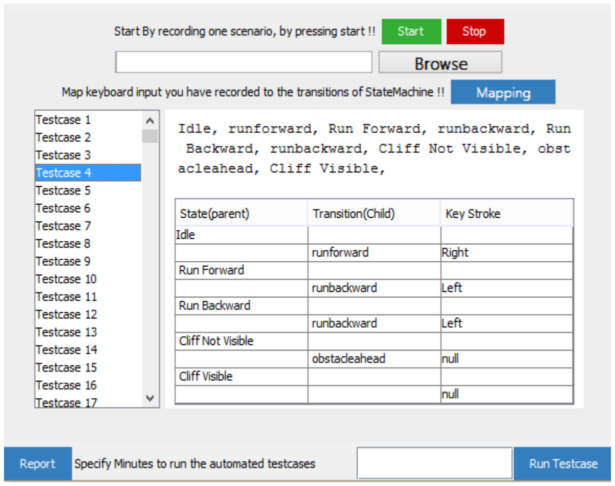
Fig. 6. Transition Tree for Mario state machine

TABLE I. MODELING STATISTISTCS FOR CASE STUDIES

Components	Mario	Industrial Game
States	27	54
Orthogonal regions	6	10
Composite states	2	6
Transitions	51	66
Guard conditions	2	13
Time based transitions	1	10
User-generated events	25	20
Game query events	12	10
System-generated events	5	13

```
→,moveforward
←,movebackward
Spacebar,jump
Esc,pause
```

Fig. 7. Excerpt of mapping file.



Start By recording one scenario, by pressing start !!

Map keyboard input you have recorded to the transitions of StateMachine !!

Testcase 1
Testcase 2
Testcase 3
Testcase 4
Testcase 5
Testcase 6
Testcase 7
Testcase 8
Testcase 9
Testcase 10
Testcase 11
Testcase 12
Testcase 13
Testcase 14
Testcase 15
Testcase 16
Testcase 17

```
Idle, runforward, Run Forward, runbackward, Run
 Backward, runbackward, Cliff Not Visible, obst
acleahead, Cliff Visible,
```

State(parent)	Transition(Child)	Key Stroke
Idle		
	runforward	Right
Run Forward		
	runbackward	Left
Run Backward		
	runbackward	Left
Cliff Not Visible		
	obstacleahead	null
Cliff Visible		
		null

Specify Minutes to run the automated testcases

Fig. 8 Screenshot of APGET tool

VI. Limitations

Our approach relies on building test ready models of the platform game under test. Game developers have to invest time and resources in developing the test ready models. However, once the models are available, our approach can save significant amount of effort due to automation of test case generation and test case execution. Though the proposed approach seems to be promising, the exact effort saved by the proposed approach has not been shown. Such statistics require a comprehensive empirical study, such as a controlled experiment, which has not been conducted yet.

We use the N+ strategy described in [35] to generate test cases from the avatar's state machine. The N+ strategy demands traversing of state machine's transition tree using the round trip and sneak path strategy. This generates test paths from the initial state of an avatar to any particular state. However when testing games, there are scenarios that need to be tested from any particular state to any other particular state.

To overcome this limitation, we use the time bound approach that allows the tester to set a time limit and within that time limit the test cases execute continuously. Further strategies for test selection, for example based on the importance, need to be investigated.

Fig. 9. Screenshot of Mario Bros during text execution

VII. Conclusion

Manual testing of platform games is a laborious and time-consuming task, and is not scalable for industrial games. Model driven approaches have been successfully applied to automate software testing for various domains. In this paper we proposed an automated model-based testing approach for testing platform games. Platform games constitute a major portion of recently developed games for mobile devices. For our model-based testing approach, we first devised a modeling methodology that includes a UML profile and guidelines for modeling games for testing. We used UML class diagrams and UML state machines for the modeling purpose. Our game testing strategy uses the developed state machines for the automatic generation and execution of test cases. From the state machine, we generate test paths using round trip and sneak path strategies. To evaluate our approach we applied it on two industrial cases studies. The aim of the application is to evaluate the applicability of the proposed approach on actual games. The generated test cases revealed two major faults in one case study. For the second case study we only generate test cases without executing them as the testing interface for the game is under development. It was observed that the profile and the methodology were sufficient to model both the case studies for testing. The successful generation of test cases for both case studies and identification of real bugs in the open source case study demonstrates the applicability and practical significance of the approach as an alternative to prevailing manual testing practices. Our experience show that the effort involved in developing test ready models is significantly less than the effort involved in manually testing all game scenarios.

Acknowledgment

The work presented in this paper was supported by ICT R&D Fund, Pakistan under the MBT Toolset project (ICTRDF/ MBTToolset/ 2013). Muhammad Zohaib Iqbal was also supported by the Fonds National de la Recherche, Luxembourg (FNR/P10/03).

REFERENCES

[1] Xlocinc. Sales, Demographic and Usage data, Entertainment Software Association, October 2014. Available: Sales, Demographic and Usage data, Entertainment Software Association, October 2014, https://xlocinc.wordpress.com/tag/gamer-demographics/

[2] A. Perry and R. Cook, Real Sound Synthesis for Interactive Applications: I A. K. Peters, Ltd. , 2002.

[3] G. Studios. Flappy Birds. Available: http://flappybird.io/

[4] Kiloo and S. Games. Subway Surf. Available: http://www.kiloo.com/games/subway-surfers/

[5] B. Adventure. Caveman Mario Adventure. Available: https://play.google.com/store/apps/details?id=com.creativadev.games.superancientworld&hl=en

[6] I. Studios. Temple Run. Available: http://www.imangistudios.com/

[7] C. Schultz, R. Bryant, and T. Langdell, Game testing all in one: Course Technology, 2005.

[8] C. Buhl and F. Gareeboo, "Automated testing: a key factor for success in video game development. Case study and lessons learned," in proceedings of Pacific NW Software Quality Conferences, 2012, pp. 1-15.

[9] Fmc3. Mario Brothers. Available: http://javamarioplatformer.codeplex.com/SourceControl/latest#src/devforrest/mario/objects/mario/Mario.java

[10] Y. J. Choi, "Providing novel and useful data for game development using usability expert evaluation and testing," in proceedings of Sixth International Conference on Computer Graphics, Imaging and Visualization, CGIV'09, 2009, pp. 129-132.

[11] X. Ferre, A. d. Antonio, R. Imbert, and N. Medinilla, "Playability Testing of Web-Based Sport Games with Older Children and Teenagers," in proceedings of the 13th International Conference on Human-Computer Interaction. Interacting in Various Application Domains, 2009, pp. 315-324.

[12] N. M. Diah, M. Ismail, S. Ahmad, and M. K. M. Dahari, "Usability testing for educational computer game using observation method," in proceedings of the International Conference on Information Retrieval & Knowledge Management,(CAMP), 2010, pp. 157-161.

[13] H. Desurvire, M. Caplan, and J. A. Toth, "Using heuristics to evaluate the playability of games," in proceedings of Human factors in computing systems, 2004, pp. 1509-1512.

[14] H. Korhonen and E. M. Koivisto, "Playability heuristics for mobile games," in proceedings of the 8th conference on Human-computer interaction with mobile devices and services, 2006, pp. 9-16.

[15] P. Moreno-Ger, J. Torrente, Y. G. Hsieh, and W. T. Lester, "Usability testing for serious games: Making informed design decisions with user data," Advances in Human-Computer Interaction, p. 4, 2012.

[16] S. R. Dandey, "An Automated Testing Framework for the Virtual Cell Game," Master of Science, Computer Science, North Dakota State University, Fargo, N.D, 2013.

[17] K. Peterson, S. Behunin, and F. Graham, "Automated testing on multiple video game platforms," 2012.

[18] C. Schaefer, H. Do, and B. M. Slator, "Crushinator: A framework towards game-independent testing," in proceedings of Automated Software Engineering (ASE), 2013 IEEE/ACM 2013, pp. 726-729.

[19] H. Zhao, J. Sun, and G. Hu, "Study of Methodology of Testing Mobile Games Based on TTCN-3," in proceedings of Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing, 2009. SNPD'09. , 2009, pp. 579-584.

[20] A. M. Smith, M. J. Nelson, and M. Mateas, "Computational Support for Play Testing Game Sketches," in proceedings of Fifth Artificial Intelligence and Interactive Digital Entertainment AIIDE, 2009.

[21] B. Chan, J. Denzinger, D. Gates, K. Loose, and J. Buchanan, "Evolutionary behavior testing of commercial computer games," in proceedings of Evolutionary Computation, 2004, pp. 125-132.

[22] A. Nantes, R. Brown, and F. Maire, "A Framework for the Semi-Automatic Testing of Video Games," in proceedings of Fourth Artificial Intelligence and Interactive Digital Entertainment AIIDE, 2008.

[23] W. Thunputtarakul and V. Kotrajaras, "AI-TEM: Testing AI In Commercial Game with Emulator," in proceedings of Computer Games: AI, Animation, Mobile, Educational & Serious Games (CGAMES06), 2006.

[24] C.S. Cho, K. M. Sohn, C.J. Park, and J.H. Kang, "Online game testing using scenario-based control of massive virtual users," in proceedings of the 12th International Conference on Advanced Communication Technology (ICACT), 2010, pp. 1676-1680.

[25] N. Tillmann, J. d. Halleux, T. Xie, and J. Bishop, "Pex4Fun: A web-based environment for educational gaming via automated test generation," in proceedings of Automated Software Engineering (ASE), 2013 IEEE/ACM 2013, pp. 730-733.

[26] Y. Choi, H. Kim, C. Park, and S. Jin, "A case study: Online game testing using massive virtual player," in proceedings of Control and Automation, and Energy System Engineering, 2011, pp. 296-301.

[27] B. Fernandes, "Load testing online game server environment using web-based interface," 2010.

[28] R. Knizia, "The design and testing of the board game Lord of the Rings," in Rules of play: Game design fundamentals, ed, 2004, pp. 22-27.

[29] M. Ostrowski and S. Aroudj, "Automated Regression Testing within Video Game Development," Journal on Computing (JoC), vol. 3, 2014.

[30] M. Z. Iqbal, A. Arcuri, and L. Briand, "Environment Modeling and Simulation for Automated Testing of Soft Real-time Embedded Software," Software & Systems Modeling, pp. 1-42, 2013.

[31] W. R. Play. Run Sheeda Run. Available: http://werplay.com/runsheedarun/

[32] C. Larman, Applying UML and patterns: an introduction to object-oriented analysis and design and iterative development: Pearson Education India, 2005.

[33] M. U. Khan, M. Z. Iqbal, and S. Ali, "A Heuristic-Based Approach to Refactor Crosscutting Behaviors in UML State Machines," in Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on, 2014, pp. 557-560.

[34] M. Z. Iqbal, M. Usman, M. U. Khan, "A Model-driven Approach to Generate Mobile Applications for Multiple Platforms," in 21st Asia Pacific Software Enigneering Conference (APSEC), Jeju, Korea, 2014.

[35] R. Binder, Testing object-oriented systems: models, patterns, and tools: Addison-Wesley Professional, 2000.