An Approach for Model Based Testing of Augmented Reality Applications

Porfirio Tramontana, Marco De Luca and Anna Rita Fasolino

University of Naples "Federico II", Napoli, Italy

Abstract

The popularity of Augmented Reality (AR) applications has strongly been increased with the worldwide success of the Pokemon Go videogame released by Niantic in 2016. However, AR offers tangible benefits in many further areas beyond entertainment, such as advertisement, education, navigation, maintenance, health, and so on. With the growing spread and success of AR applications in these fields, there has also been a growing necessity for approaches and technologies for assuring the quality of these applications, such as testing. A few technologies and frameworks have been recently proposed supporting the implementation and execution of test scripts that can be used to exercise the applications, but there still is a lack of effective techniques and tools for the automatic generation of executable test cases. In this paper, we investigate the possibility of using Model Based Testing techniques to generate executable test scripts from Finite State Machines modeling the behaviour of the GUI of AR applications, similarly to other GUI based applications. We have applied several model coverage criteria to design test suites and we have shown the feasibility of this approach by testing two small example applications involving Unity3D and Vuforia technologies.

Keywords

Augmented Reality, Model Based Testing, Finite State Machines

1. Introduction

Extended reality (XR) is an umbrella term to describe different kinds of technologies that are able to merge the physical and virtual worlds. More in details, it is possible to distinguish between:

- Virtual reality (immersive or non-immersive VR), where the application simulates a completely different environment around the user;
- Augmented reality (AR), where the experience enhances the real world with digital details such as images, text, and animation;
- Mixed Reality (MR), where the application combines its own digital environment with the user's real-world environment and allows them to interact with each other.

In particular, AR is a way to provide users with a sensorial experience beyond the reality. Differently from Non-Immersive VR that is usually implemented in the context of console or desktop interactive applications and Immersive VR, that needs special glasses or Visors,

Joint Proceedings of RCIS 2022 Workshops and Research Projects Track, May 17-20, 2022, Barcelona, Spain
ptramont@unina.it (P. Tramontana); marco.deluca2@unina.it (M. D. Luca); fasolino@unina.it (A. R. Fasolino)

Augmented Reality is now usually deployed on smartphones in form of Android, iOS or cross-platform apps. In the following, we will focus on AR applications.

Several factors have recently fueled the research and development of AR: the emergence of dedicated AR devices and powerful development kits (such as Unity3D and UnReal), the improvements in the performance of mobile devices and sensor integration, and advances in computer vision (CV) technologies. The popularity of AR applications has strongly been increased with the worldwide success of the Pokemon Go AR videogame released by Niantic in 2016[1]. However, AR offers tangible benefits in many further areas beyond entertainment, such as advertisement, education, navigation, maintenance, health, and so on. With the growing spread and success of AR applications in these fields, there has also been a growing necessity for approaches and technologies for assuring the quality of these applications, such as testing.

Testing of AR applications can be carried out at different levels. Unit testing can be carried out to test source code methods and involves general techniques and tools of the XUnit family. Unit testing does not suffice to reveal the faults of an application which can be exposed instead by testing it at system level, by sequences of user and system events. AR applications are event-based systems: event-based testing techniques can be used to test them likewise any GUI-based or event-based system. A few technologies and frameworks have been recently proposed supporting the implementation and execution of test scripts that can be used to exercise the applications at system level. Although approaches for the functional testing of VR applications have been recently proposed [1, 2], according to [3], more effective methods and tools supporting the systematic design and execution of AR application testing are still needed.

In this paper, we investigate the possibility of using Model Based approaches for testing the behaviour of AR applications. These approaches are gaining popularity in literature and have recently used with proficiency to teach testing activities to students [4]. We propose to use Finite State Machines models to represent the behaviour of the GUI of AR applications, similarly to other GUI based applications. We apply model coverage criteria to design test suites that can be implemented as automatically executable test scripts. We demonstrate the feasibility of this approach and report some encouraging results we obtained by testing two small example applications involving Unity3D and Vuforia technologies.

The paper is structured as follows. In Section 2 and 3, respectively. an introduction about AR applications and the works in literature discussing their testing are reported. Section 4 presents the proposed Model Based approach for the generation of test suites, while Section 5 shows the feasibility of the approach and the effectiveness of the test suites on two example applications. Finally, Section 6 discusses conclusions and future works.

2. Background

AR applications are composed of a client side responsible for the rendering of a 3D environment mixing real camera images and virtual widgets that can be statically designed or generated on the fly when specific *marker* images are recognized. The behaviour of the AR application in response to user and system events is defined by using general purpose programming languages (such as C#).

[1] https://pokemongolive.com/

An Execution Engine is required for rendering and running the application in the context of a device. The two most popular engines for the development of 3D games and applications are Unity3D[2] and UnReal[3]. The image detection capability can be provided by components such as the *Vuforia Engine*[4], that provides a *Tracking* service that allows the client to query a remote image recognition service provided by Vuforia to know when a specific *Marker* image is shown on the camera.

In this paper we have focused our attention on Augmented Reality applications developed with Unity and Vuforia. Each Unity project is composed of *Scenes*. A scene can be conceptually represented by an instance of a GUI, that is rendered on the user device (e.g. on a smartphone) and can be three-dimensionally navigated, exploiting the touch events and the inputs from sensors, including motion sensors and camera. Each scene is composed of objects, which can be specialized in *GameObjects* and *Components* that can be interconnected between them. GameObjects represent graphic items that can move around the scene and with which the user can interact. Components represent parts of the GameObjects and they can be associated with code that describes the dynamic behavior of the components and of the GameObjects. In particular, a component may implement listeners to *Events*. *Events* include both user events, system events and the recognition and loss of *Marker* images. Figure 1 shows a class diagrams depicting a metamodel including the main elements of a Unity AR application.

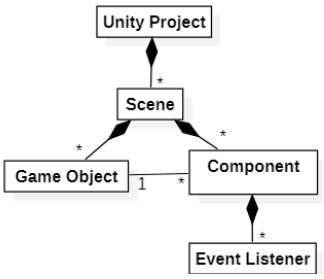

Figure 1: Metamodel of the elements of an Unity AR application

3. Related Work

In the last years, several works investigating the issues of extended reality applications and possible approaches for detecting them have been proposed in the literature.

Lehman et al. [5] stress the aspect that AR apps are different from conventional apps in that the augmented images and labels are generated and positioned based on the user's behavior and environment. As a consequence, they identify four categories of common failures in AR applications that are difficult to detect using conventional software engineering testing. The failures they focus on consist of object classification failures (due to the impossibility to train a classifier for every variation of inputs that it may receive, or to control exactly the movement

[2]https://unity.com/
[3]https://www.unrealengine.com/en-US/
[4]https://library.vuforia.com/

and behavior of the user), placement failures, resource limitation failures, and style failures. These failures need to be detected in the wild and, to this aim, the authors proposed the ARCHIE framework. ARCHIE collects user feedback and system state data in situ to help developers identify and debug issues important to testers.

In 2020, Li et al [6] focused on bugs in XR applications deployed on the Web that exploit the WebXR Device API. This technology enables users to interact with browsers using XR devices. However, many WebXR applications are insufficiently tested before being released and they suffer from various bugs that can degrade user experience or cause undesirable consequences. To better understand the nature of bugs in WebXR applications, the authors performed an empirical study where they collected 368 real bugs from 33 WebXR projects hosted on GitHub. Via a seven-round manual analysis of these bugs, they built a taxonomy of WebXR bugs according to their symptoms and root causes. They found three main types of issues: (1) functional issues, (2) crashing issues, and (3) performance issues. Functional issues were further classified into Application-Specific Functional Issues (consisting in unexpected behaviors often caused by improper lifecycle event handling and erroneous design of interactive logic) and Rendering issues (misrendering of objects or missing objects issues). Crashing issues consisted in runtime exceptions or immediate application crashes. Performance issues in WebXR projects have various symptoms including high memory consumption, high CPU utilization, abnormal hanging of applications, and low frame rate or resolution. They observed six major root causes of WebXR bugs, including: (1) incompatible runtime environment, (2) event handling mistakes, (3) improper handling of diversified user interaction mechanisms, (4) wrong arguments, (5) buggy dependencies, and (6) redundant operations. A further study investigated quality issues (bad smells) in Unity projects [7]. The authors proposed UnityLinter, a static analysis tool that supports Unity video game developers to detect seven types of bad smells.

All the considered works show that several issues may affect the quality of XR applications. Defining approaches and technologies for testing these applications and detecting such issues is absolutely necessary for the XR developer community.

As to the technologies supporting AR application testing, a few frameworks and libraries have recently emerged and are currently available to the tester community. The AirTest framework[5] allows to implement test cases replicating sequences of interactions with an AR application. The *airtest.core.api* library allows to trigger different types of events on Unity3D applications, including user events (e.g. click on buttons), system events (e.g. application opening and closing) and Vuforia related events (e.g. marker identification and disappearing). In addition, the *poco* library[6] included in AirTest provides methods useful to implements locators that return references to widgets and other objects present on the GUI of the AR under test.

Another solution is offered by *AltUnity Tester*[7], a free tool for testing of applications built with Unity. AltUnity Tester allows to write tests in C#, Python and Java. AltUnity Tester consists of *AltUnity Server*, which allows to access objects in the GUI hierarchy by opening a TCP socket on the device running the application and waiting for the connection of an *AltUnity Client*, used to connect to AltUnity Server by accessing and interacting with objects through written tests.

[5] https://airtest.netease.com/

[6] https://github.com/AirtestProject/Poco-SDK

[7] https://altom.com/testing-tools/altunitytester/

These technologies doubtless provide a support to implement and execute test cases that can be used to exercise the applications at both unit and system level. According to [3], more effective methods and tools supporting the design and execution of XR application testing are needed.

4. Model Based Testing of AR applications

Model-based testing (MBT) relies on models of a system under test and/or its environment to derive test cases for the system. It encompasses the processes and techniques for the automatic derivation of abstract test cases from abstract models, the generation of concrete tests from abstract tests, and the manual or automated execution of the resulting concrete test cases [8].

A MBT process relies on some fundamental activities: (1) Modeling of the system under test, (2) Definition of Test Selection Criteria and Test case design, (3) Implementation and execution of the test cases in the context of the system under test. In this paper we have faced out these problems in the context of AR applications, with specific focus on the automation of the last activity.

4.1. Modeling the GUI of an Augmented Reality application

The behaviour of the front end of AR applications can be modeled by Finite State Machines (FSM) as the one of other event-based GUIs [9, 10, 11]. In this case *States* correspond to instances of Scene objects with a specific set of widgets, while *Transitions* correspond to changes between scenes showing different widgets. Transitions are activated by *Event* triggers when a possible *Guard* condition is true. The Guard condition may also depend on data variables locally defined in the Component code.

A possible way for obtaining such FSM model consists in reverse engineering it by static and dynamic analysis of the application. Static analysis should take into account both the application structure (e.g. Scenes, GameObjects, Components) and the source code of the scripts (e.g. variables, event listeners and guard conditions), in order to identify the states of the app. Dynamic analysis can be exploited to explore the behaviour of the application at runtime and inferring the state transitions.

For example, Figure 2 shows the FSM modeling the behaviour of an example AR application. This application is composed of a single Scene. When the application is started, a *Language menu* is shown on the device screen. When the user specifies the preferred language, the application goes in a *Marker Waiting* state where the device camera output is shown and the Vuforia listener observes when a marker image is framed by the camera. When a marker is recognized, the corresponding animation is shown on the device screen (*AR Animation* state). When the marker disappears from camera image, the application returns to the *Marker Waiting* state. When the application is either on the *Marker Waiting* and on the *AR Animation* it is possible to return to the *Language Menu* state by means of a settings button. From the *Language Menu* state it is possible to quit the application. Figure 3 shows, from left to right, the screenshot of the GUI application in the *Language Menu* state, the *Marker* that has to be recognized, and a screenshot of the animation shown by the application when it is in the *AR Animation* state.

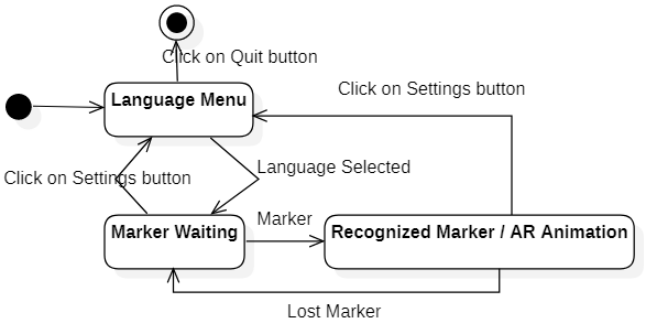

Figure 2: An Example of a FSM modeling the behaviour of an AR application

Figure 3: Details from the example AR application: the *Language Menu* state, the marker to be identified, and the *AR Animation* state

4.2. Definition of Test Selection Criteria and Test Case Design

Testing strategies guided by specific FSM model coverage objectives can be used to define the test suites. In particular, we have considered the following coverage objectives:

- All States Coverage, in which each state of the FSM model is reached by at least a test case;
- All Transitions Coverage, in which each transition of the FSM model is reached by at least a test case;
- All Prime Paths Coverage [12], in which each prime path on the FSM is covered by at least a test case.

For each of the considered testing strategies, different test suites built to satisfy the model coverage objectives can be defined.

4.3. Test Case Implementation and Execution

In this step, the designed test cases have to be implemented using the features of a test automation framework or library, in order to make them automatically executable. To this aim, some features offered by the AirTest library can be exploited.

In general, the implemented test cases will exploit (1) functions for pre-condition setting (the *setup* function), (2) functions for state identification, (3) a sequence of operations triggering the actions constituting the test case, (4) assertions to check the app behaviour, (5) a function for post-condition tear down (the *tearDown* function).

Listing 1 shows an excerpt of a test script written in Python and using the AirTest library. The test script code uses the AirTest *poco* object to obtain references to menus, buttons and other widgets on the scenes. The *verify* methods are used to recognize the occurrence of the FSM states on the basis of the values of the state variables (in this example we used the *language* variable), the widgets shown on the GUI (e.g. the button), and the identification of the image marker. Assertions have been inserted into the test script to evaluate the occurrence of the expected sequence of states. The final set of statements represents the sequence of events constituting the test case. In this example, after the application is started, the *LanguageMenu* state should be recognized. On this GUI the *start* button is clicked. When the marker reported in Figure 3 is recognized by Vuforia, the *ARAnimation* state should be reached and the test ends.

Listing 1: Excerpt of a test script code

```
from airtest.core.api import *
from poco.drivers.unity3d import UnityPoco
poco = UnityPoco()

def setup():
    os.system(...)
    global poco= UnityPoco();
    global language = "English";;

def tearDown():
    os.system(...)

def verifyLanguageMenuState():
    global startButton;
    menu = poco("LanguageSelectionGroup").children();
    button = poco(type = 'Button');
    identified = False;
    if(len(menu) > 0 and len(startButton) == 1 and startButton.attr('name') == "StartButton"):
        identified = True;
    assert_equal(identified,True);

def verifyARAnimationState():
    global button;
    scene = poco("LiftAnimation").children();
    button = poco(type = 'Button');
    identified = False;
    if(len(scene) > 1 and len(button) == 1 and button.attr('name') == "Settings"):
        marker = poco("LIFT - "+language);
        identified = marker.exists();
    assert_equal(identified,True);

setup()
verifyLanguageMenuState()
button.click();
verifyARAnimationState()
tearDown()
```

5. Examples

We have carried out two testing activities on two example AR applications with the goal *to show the feasibility of the proposed Model Based Testing technique on AR applications and evaluate its effectiveness.*

The considered applications were two small open source AR apps, both implemented with

Table 1
Size metrics of the two AUTs

	Code Metrics				FSM Metrics	
	#Classes	#Methods	#LOCs	#Branches	#States	#Transitions
A1	3	8	156	13	4	15
A2	3	7	153	14	3	9

Table 2
Number of generated test cases and coverage metrics for the three generated test suites and the two AUTs

	#Test Cases			State Coverage			Transition Coverage			Branch Coverage		
	TS1	TS2	TS3	TS1	TS2	TS3	TS1	TS2	TS3	TS1	TS2	TS3
A1	2	6	12	4/4	4/4	4/4	4/15	15/15	15/15	9/13	12/13	12/13
A2	3	6	7	3/3	3/3	3/3	2/9	9/9	9/9	10/14	13/14	13/14

Unity3D and Vuforia. PointAR (A1)[8] is a concept app showcasing the usage of Augmented Reality to assist the foreign workforce with the induction process through the use of 3D animation for visualisation and built-in translations. SafaryAnimal (A2)[9] is a simple AR educative game using Unity and Vuforia where animals 3D renderings appear and disappear when specific marker images are observed.

Table 1 reports some metrics about the applications under test (AUTs) and the reverse engineered FSMs describing their behaviour. The table reports on the left part some source code metrics (number of classes, number of methods, number of LOCs and number of branches of the scriping source code), and on the right part some FSM metrics (the number of states and the number of transitions). For each of the two AUTs three different test suites have been generated from the FSM models, according to the three different coverage criteria. The test suites named TS1 have been generated having the objective to cover all the FSM states, whereas the test suites TS2 have been implemented to cover all the FSM transitions. Finally, the test suites names TS3 have been written to achieve the coverage of all the prime paths on the FSM model. In order to evaluate the effectiveness of the Model based test suites, the coverage of states, transitions and prime paths have been measured, together with the source code branch coverage. To obtain these measures we manually inserted probes in the application source code (in correspondence of each method declaration and each control structure branch) and in the source code of the test cases (in correspondence of the state and transition identification statements).

Table 2 reports the number of test cases composing each test suite and the measured coverage values for each considered AUT. The obtained results show that the test suites designed to cover transitions or prime paths achieve better coverage than the ones aiming at covering model states. In fact, the TS1 ones cover the minority of the states and do not cover several code branches that are instead reached by the other two test suites. On the other hand, the TS2 and TS3 test suites also show some lacks in coverage (one branch for each AUT).

In order to understand the causes of the coverage lacks, we analyzed the branches that have not been covered by the test suites. In AUT1 there is a branch that is not covered by any test

[8]https://github.com/abdullahibneat/PointAR
[9]https://github.com/abdullahibneat/SafariAnimalsAR

suite: it corresponds to the code that is activated when the default language (English) is selected after having previously selected another language. This branch could be covered by test cases executing longer loops between the same states, including a language change and a return to the initial language. It is thus unsurprising that the branch has not been covered by test cases aiming at avoiding loop repetitions. In addition, TS1 does not execute most of the branches related to language changes since they are not necessary to discovery new states of the FSM.

In AUT2 there are selection buttons to change the animal shown on the screen while remaining in the *Recognized Marker* status. The interactions with these buttons have not been triggered by the TS1 test suite as they do not cause transitions toward new states. There is also a branch that has not been covered by any test suite. It is activated by the condition in which an animal with an incorrect index is selected. This condition is not feasible with the current version of the application, thus it can be classified as dead code. For this reason, we can conclude that both TS2 and TS3 have been able to cover all the feasible branches of the source code.

In conclusion, we have observed how the strategies aiming at the coverage of transitions and prime paths have been able to provide a complete coverage of states and transitions and an almost complete coverage of the branches of the code. Although the example applications are tiny and simple, the obtained results are promising and future work is necessary to generalize them with respect to larger and more complex AR applications. The set of tools, the implemented test cases and the output of their execution are available on a Github repository[10].

6. Conclusions

In this paper we have investigated the possibility to implement Model Based Testing techniques on AR applications, exploiting their similarity with other types of GUIs on which MBT were applied with success in the past. We have modeled the behaviour of the GUI of AR applications with Finite State Machines that can be manually reverse engineered on the basis of the analyses of the structure of the client side of the application, of its source code (including the code of the listeners of user and system events, such as the ones related to the identification and loss of markers, that are typical of AR applications) and of the observed behaviour. The obtained FSM model have been exploited to design test suites aiming at covering states, transitions and prime paths. These test suites have been implemented in form of automatically executable test scripts exploiting the features offered by AirTest. We have demonstrated the feasibility of this approach and some encouraging results on two small example applications involving Unity3D and Vuforia technologies.

This paper represents a preliminary work, for which we plan to carry out several activities in the future in order to extend its applicability to test larger applications. We have recognized the need to implement reverse engineering techniques and tools helping the modeling process and supporting the automatic generation of model based test scripts. In particular, more general solutions to the problem of state identification will be studied, together with the extension of the support to user and systems events, exploiting the features offered by more recent emulators.

[10] https://github.com/PorfirioTramontana/MBT-AR-applications

Acknowledgments

This work was partially funded by the grant FFABR of the italian ministry for university and research (MIUR) and by the grant REYNA of the University of Naples Federico II. The authors would also thank S. D. Bevilacqua for the implementation of the test suites in the context of his Laurea Degree Thesis activity.

References

[1] A. C. Corrêa Souza, F. L. S. Nunes, M. E. Delamaro, An automated functional testing approach for virtual reality applications, Software Testing, Verification and Reliability 28 (2018). URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1690.

[2] S. A. Andrade, F. L. S. Nunes, M. E. Delamaro, Towards the systematic testing of virtual reality programs, in: SVR, 2019, pp. 196–205. doi:10.1109/SVR.2019.00044.

[3] R. Prada, I. S. W. B. Prasetya, F. M. Kifetew, F. Dignum, T. E. J. Vos, J. Lander, J. Donnart, A. Kazmierowski, J. Davidson, P. M. Fernandes, Agent-based testing of extended reality systems, in: ICST, 2020, pp. 414–417. doi:10.1109/ICST46399.2020.00051.

[4] B. Marín, S. Alarcón, G. Giachetti, M. Snoeck, Tescav: An approach for learning model-based testing and coverage in practice, Lecture Notes in Business Information Processing 385 LNBIP (2020) 302 – 317. doi:10.1007/978-3-030-50316-1_18.

[5] S. M. Lehman, H. Ling, C. C. Tan, Archie: A user-focused framework for testing augmented reality applications in the wild, in: VR, 2020, pp. 903–912. doi:10.1109/VR46266.2020.00013.

[6] S. Li, Y. Wu, Y. Liu, D. Wang, M. Wen, Y. Tao, Y. Sui, Y. Liu, An exploratory study of bugs in extended reality applications on the web, in: ISSRE, 2020, pp. 172–183. doi:10.1109/ISSRE5003.2020.00025.

[7] A. Borrelli, V. Nardone, G. A. D. Lucca, G. Canfora, M. D. Penta, Detecting video game-specific bad smells in unity projects, in: MSR, 2020, pp. 198–208. doi:10.1145/3379597.3387454.

[8] M. Utting, A. Pretschner, B. Legeard, A taxonomy of model-based testing approaches, Software Testing, Verification and Reliability 22 (2012) 297–312. doi:https://doi.org/10.1002/stvr.456.

[9] D. Amalfitano, A. R. Fasolino, P. Tramontana, Reverse engineering finite state machines from rich internet applications, in: WCRE, 2008, pp. 69–73. doi:10.1109/WCRE.2008.17.

[10] M. Sama, S. Elbaum, F. Raimondi, D. S. Rosenblum, Z. Wang, Context-aware adaptive applications: Fault patterns and their automated identification, IEEE Transactions on Software Engineering 36 (2010) 644–661. doi:10.1109/TSE.2010.35.

[11] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, A. M. Memon, Mobiguitar: Automated model-based testing of mobile apps, IEEE Software 32 (2015) 53–59. doi:10.1109/MS.2014.55.

[12] N. Li, U. Praphamontripong, J. Offutt, An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage, in: ICST Workshops, 2009, pp. 220–229. doi:10.1109/ICSTW.2009.30.