TALLINN UNIVERSITY OF TECHNOLOGY
Faculty of Information Technology
Department of Software Science

Model-Based Testing of Unity Applications: VR Quiz Case Study

Master's thesis

Student: Madis Taimre
Student code: 144115IAPM
Supervisor: PhD Juhan-Peep Ernits

Tallinn 2016

Declaration

I declare that this thesis is the result of my own research except as cited in the references. The thesis has not been accepted for any degree and is not concurrently submitted in candidature of any other degree.

Madis Taimre

January 8, 2017

....
(Signature)

Abstract

In this thesis an automated testing approach for system testing of applications made in Unity is proposed. Unity is a platform, which is used to create 3D or 2D games for PCs, mobile devices and consoles. Furthermore a framework for testing Unity applications was developed as part of this thesis. The framework enables controlling a Unity application remotely and allows to get information about the state of the system. The proposed testing approach is based on model-based testing.

To understand the need of the Unity developers further, currently used methods for automated system testing was researched. Additionally several game development companies was contacted including Unity Technologies.

As a proof of concept a case study was made on a game named VR Quiz which the author of the thesis is currently developing. The test suite was able to detect several bugs in the system.

The thesis is in English and contains 45 pages of text, 6 chapters, 18 figures, 1 table and over 2800 lines of code in an open Gitlab repository [1]. The thesis also contains test suites for VR Quiz made with NModel and TestCast MBT.

Annotatsioon

Selles töös on välja pakutud lahendus, kuidas luua automatiseeritud süsteemiteste Unity-s tehtud programmidele. Unity on platvorm, millega saab teha 3D ja 2D mänge personaalarvutitele, nutiseadmetele ja konsoolidele. Pakutud lahendus kasutab mudelipõhist testimist. Samuti on töö käigus loodud testimise raamistis Unity-s loodud programmide jaoks, millega saab informatsiooni programmi oleku kohta ja juhtida selle käitumist. Kuigi pakutud üldlahendus kasutab mudelipõhist testimist ja tööriista TestCast MBT, saab loodud raamistikku kasutada ka teiste tööriistadega ja süsteemitestimise meetoditega.

Töö raames tehti uuring, kuidas tehakse süsteemitestimist mängudele ja analüüsiti olemasolevaid võimalusi süsteemitestimise automatiseerimiseks. Lisaks uuritud artiklitele küsiti ka mitmetelt mängude arendajatelt, kuidas nad teevad süsteemitestimist. Väga olulist informatsiooni saadi Unity ettevõtte töötajalt, kes rääkis oma kogemusest kuidas tema Unity-s töödates püüdis luua testimise raamistikku Unity-s tehtud programmide jaoks ja millised nõuded peaksid sellisel süsteemil olema.

Pakutud testimise lahendust kasutati Unity-ga loodud mängu VR Quizi peal, mida arendab samuti töö autor. VR Quiz on mõeldud muuseumites üles panemiseks ja see toimib nii, et kui külastaja paneb pähe VR prillid, satub ta muuseumi teemaga haakuvasse virtuaalmaailma. Kasutaja näeb selles maailmas sinna asetatud küsimust, mille on välja mõelnud muuseumitöötajad, ning pildilisi vastusevariante.

VR Quizile loodud automaattestid katavad umbes 60% funktsionaalsusest ja on praeguseks leidnud neli viga. Lisaks kasutatakse automaatteste regressioonitestideda arenduse käigus.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 45 leheküljel, 6 peatükki, 18 joonist, 1 tabel ja üle 2800 rea koodi avalikus Gitlabi hoidlas [1]. Samuti lõputöö sisaldab teste VR Quizi jaoks, mis on tehtud kasutades NModelit ja TestCast MBT-d.

Glossary of terms, acronyms and abbreviations

AR	Augmented reality
DOF	Degree of freedom - the number of independent motions a body, in this thesis the player, can do.
FSM	Finite-state machine - an abstract machine that can be in one of the finite number of states
GUI	Graphical user interface
MBT	Model-based testing
PC	Personal computer
QA	Quality assurance
SUT	System under test
TTCN-3	The Testing and Control Notation Version 3
VR	Virtual reality

Contents

List of Figures

List of Tables

1. Introduction

Unity is a platform for creating 2D, 3D, virtual reality (VR) and augmented reality (AR) games and apps. It has a graphics engine and an editor, where the game can be created. Unity has multiplatform support meaning that the programs created in Unity can be deployed for PC-s, the web, mobile devices, home entertainment systems, embedded systems and head-mounted displays [3]. Unity can be used for free as long as the company, for which the game is being developed, has an annual gross revenue lower than 100 000 $ per year [4]. The number of Unity users has increased to almost 5.5 million and 770 million people play games made with Unity. Despite the heavy usage there is very little options for automated system testing of applications made with Unity.

1.1 Problem

To understand further how game development companies do system testing some studios were contacted and interviewed. The companies were Housemarque, Rovio, Creative Mobile, Frogmind Games and an employee of Unity Technologies ApS. The questions asked from them were:

- Do they use automated system testing?
- What kind of automated testing methods do they use?
- What problem do they see in automated system testing?

One of the companies created an extensive test suite for one of their games, but the others have never done any automated system testing or has used bots for very simple smoke testing.

The person from Unity worked on developing an automated system testing solution for Unity developers. According to him the need for automated system testing approach is quite high and because of that they tried to develop one using two different tools: Graphwalker and Spec Explorer. These tools are both used for model based testing. They never made their solutions public since both of them did not fulfill the need of the developers that well. The reasons for that were that the Graphwalker had a couple of limitations for modelling and the Spec Explorer was too complex for test developer to learn. Therefore the main requirements for the testing approach are that it needs to be simple for the testers to implement and have a high enough modelling capability.

1.2 Goal of the thesis

The aim of this thesis is to propose a testing approach for testing programs made with Unity. Furthermore as a proof of concept a game called VR Quiz will be tested in the context of this thesis. VR Quiz is a game currently developed by the author of the thesis, which is meant to be deployed in a museum.

In the context of this thesis the following will be done:

1. Currently used methods for automated system testing of games are analyzed. The tools Graphwalker and Spec Explorer are studied and analyzed to further understand why they were not good enough as stated by person working in Unity (see section Problem).

2. A testing approach is proposed and a testing framework for Unity is created. The requirements for the approach are that it should be simple enough for the tester to learn and implement and that it should be possible to test more difficult games with it.

   The framework should allow to control the behaviour of the system and should have high enough abstraction, that it can be used for testing different applications.

3. Automated system testing of VR Quiz. The tests should cover at least 60% of the functional requirements of the VR Quiz menu and game.

In the end of the thesis the following problems will be analyzed based on the work done:

1. Is it feasible to use model-based testing on 3D games, where the user has six DOF to look around?
2. How to model and generate tests of a 2D menu so that all the transitions of the menu are covered without state explosion?
3. Does the model-based testing approach help to discover previously unknown errors in the application that has been manually tested?
4. Is it possible and to what extent can the models created for testing of VR Quiz be used for other applications?

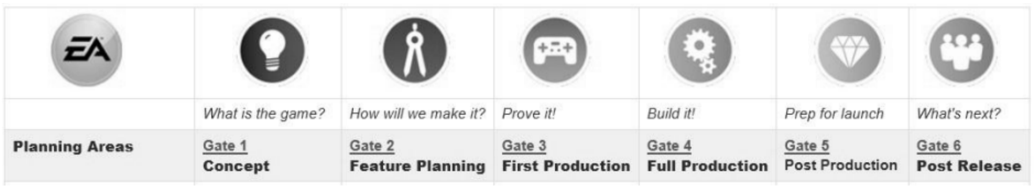
2. Automated system testing of games

Figure 2.1 shows the game development process of EA's Orlando studio. They have developed several major sports video games for Electronic Arts like Madden NFL Football, NCAA Football, Tiger Woods PGA TOUR Golf and NBA LIVE [5]. The post production of their games also divides into Alpha, Beta and Final phases.

According to [2] the biggest concern in game development is the Alpha phase when the game is feature complete. It is the biggest problem, because after that phase there should be no longer any critical bugs in the system and therefore it is at that time, when significant resources are assigned to testing. Only manual testing to confirm that the game is feature complete is done before Alpha. This leads to a very unstable game once the program goes to Alpha phase and the development team has to either work overtime or hire additional staff to get the game ready for launch date [2].

Automated testing during the development phase can substantially increase the stability of a game being developed, which would lead to a much more cost-effective overall production.

The next chapters will give a short overview of different automatic system testing approaches, which could be used in game testing, and evaluate their effectiveness for testing during the development phase.



	What is the game?	*How will we make it?*	*Prove it!*	*Build it!*	*Prep for launch*	*What's next?*
Planning Areas	Gate 1 **Concept**	Gate 2 **Feature Planning**	Gate 3 **First Production**	Gate 4 **Full Production**	Gate 5 Post Production	Gate 6 **Post Release**

Figure 2.1: EA game development process [2].

2.1 Capture and Replay

There are several tools available, which allow for a capture and replay testing of games. They work so that they record user interactions with the application and convert those actions into test scripts. These scripts can then be used to automatically run back those interactions.

Tools that support capture and replay based on paper [6] are:

1. HP QuickTest
2. Rational Robot
3. CAPBAK

The problem with capture and replay tools is that the events are captured at a very low level of abstraction. For example tests are made based on cursor and element position [6]. This means that when the system under test (SUT) is modified or new functionality is added, the tester needs to redo all the automated tests. Furthermore capture and replay tools are more semi-automatic, since they still require the tester to design the test case, select test sequences, generate test data, record test cases and generate tests manually [7].

Therefore the capture and replay technique can not be used as successfully as needed during the development phase of the game. It is an easy technique to use for regression testing, but without the high level of abstraction, the tests are not flexible enough for testing a game, which is in development and is rapidly changing.

2.2 Bots

With some games it is possible to implement bots that implement some game playing strategy, which goes through the levels and plays the game as a tester would. This is possible, when one of the features of the game is an agent with similar game-play as the player. For example in the case of the EA football game [2] bots were used for testing.

Furthermore the author of the thesis spoke with a developer in a Finnish game development company, Housemarque, who said that they have used bots for very simple testing.

The agent implementing some game strategy was able to move through levels and perform a small number of tasks. Using bots generally involves no Test Oracle and a failure can be discovered only by looking though game logs or checking whether the game crashed.

Therefore bots can be used for a very basic smoke tests, but further testing is definitely needed since they provide no Test Oracle and no systematic approach for testing.

2.3 Model-based testing

One form of system testing is state transition testing of the SUT. It works so that instead of writing test cases, an abstract state machine, based on the SUT's behaviour, is created and the transitions in the state machine are checked. The transitions can be traversed manually or by using automating tools. One of the biggest benefits of state transition testing is that it provides an observable test coverage which is additionally repeatable, documented and does not depend on the tester's creativity [8]. The systematic testing approach to testing is great for increasing coverage.

Figure 2.2 shows the number of bugs discovered during the Alpha phase of the four different football games, which were developed by EA. For the last football game, NCAA Football 2012, they used bots and scripts for smoke tests and for the previous ones they used only manual testing. The thicker line represents the number of bugs of the NCAA Football 2012. A - bugs are major stopping errors in the software, which stop further testing. C - bugs are defined as medium level bugs and D - bugs are very minor [2]. As can be seen from Figure 2.2 the number of major bugs reduced, but the number of C and D bugs increased when they used automated smoke tests and bots for testing. Since MBT provides a systematic approach to test coverage, it can also discover the C and D bugs and therefore would have been much better suited for testing.

Furthermore when using automated solutions, which generate the test sequences, using MBT can further reduce the time and costs of testing [8].

Model-based testing is especially useful when a state machine is created in the design phase of the program. There are several tools available for Unity, which allow for visual scripting using finite state machines (FSM). One example is Playmaker, which can be bought in the Unity asset store [9]. Since Playmaker was at the time of writing this thesis the second most paid asset in the asset store, FSMs are often used in game development.

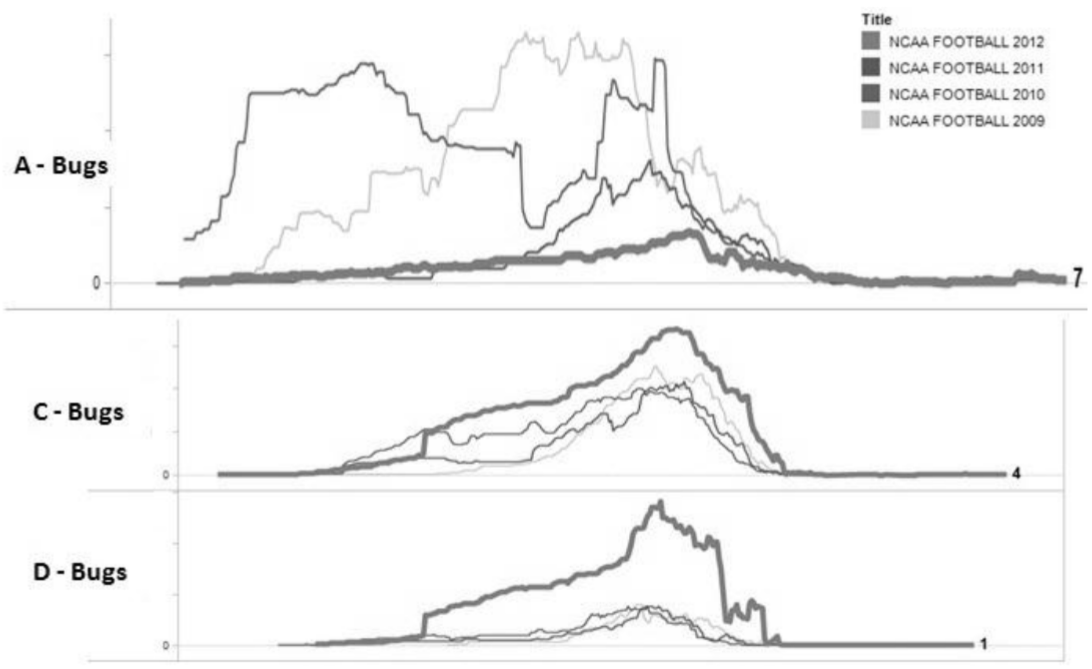


Figure 2.2: Alpha phase bugs of four NCAA football games [2].

The state machines created during the development of the game can probably be reused up to some extent for state transition testing. Therefore transition testing could be the most optimal method for system testing in those situations.

There are a number of tools available for model-based testing. Tools, which take FSM or UML as input are BPM-Xchange, Conformiq Designer, JSXM, GraphWalker, MBT-suite, Modbat, ModelJUnit, MoMuT::UML, RT-Tester, Smartesting CertifyIt, TestCast and TestOptimal [10]. Most of them are commercial tools. In the next chapters some of them are more thoroughly analyzed.

2.3.1 GraphWalker

Graphwalker is a model-based testing tool, which reads models in the shape of directed graphs and is able to generate paths from them. The graphs for it can be made using the yEd program, which is a free desktop application. Graphwalker is written in Java and is open source.

GraphWalker features based on the Graphwalker homepage [11]:

1. Offline path generation. The path is generated once and is not directly connected to

any test automation code. The path can be stored in a file so that it can also be used later.

2. Online path generation. The path is generated while executing a test at run-time. This allows for such tests to be made, that run in an infinite amount of time while testing paths at random. This feature is much sought after by the game developers as stated by a source in Unity QA.

3. Multiple models. It is possible to create a shared state in different models.

Graphwalker has two limitations in modelling. Firstly the variables used in one model can not be used in another, which may lead to the need of having the whole SUT modelled in one graph. This would severely reduce the readability of the model.

Secondly Graphwalker does not support hierarchical approach to modelling, which would significantly reduce the workload when creating the model. Chapter 2.3.4 explains how hierarchical approach using sub-machines work. Figure 2.3 shows the problem of not having such sub-machines. The same two states were modelled as in figures 2.5, 2.6, 2.7, 2.8, but this time in yEd and for GraphWalker. There are a couple of duplicate transitions (e_SetNewGameName, e_SaveReadyMadeGame), which could have been modelled once when using sub-machines.

Furthermore actual graph would be much bigger, if all the possible states and transitions were to be modelled in one file as needed for the project, which was tested in the context of this thesis, VR Quiz. For example from each different tab a separate transition has to be made to another tab. Figure 2.4 shows how moving between different tabs can be modelled in yEd. In it a helper state (ChoosingNewTab) is used so that adding each new tab would not increase the number of new transitions from other tabs exponentially. Nevertheless since all the states have to be in one model and because of extra helper states or exponential rise in the number of transitions when adding new states, using GraphWalker is not optimal for modelling more complex systems.

2.3.2 Spec Explorer

Spec Explorer is a tool developed by Microsoft for testing object-oriented reactive systems. The tester provides user scenarios and test cases for the Spec Explorer from which it can generate infinitely large state space. The systems behaviour is described by a model

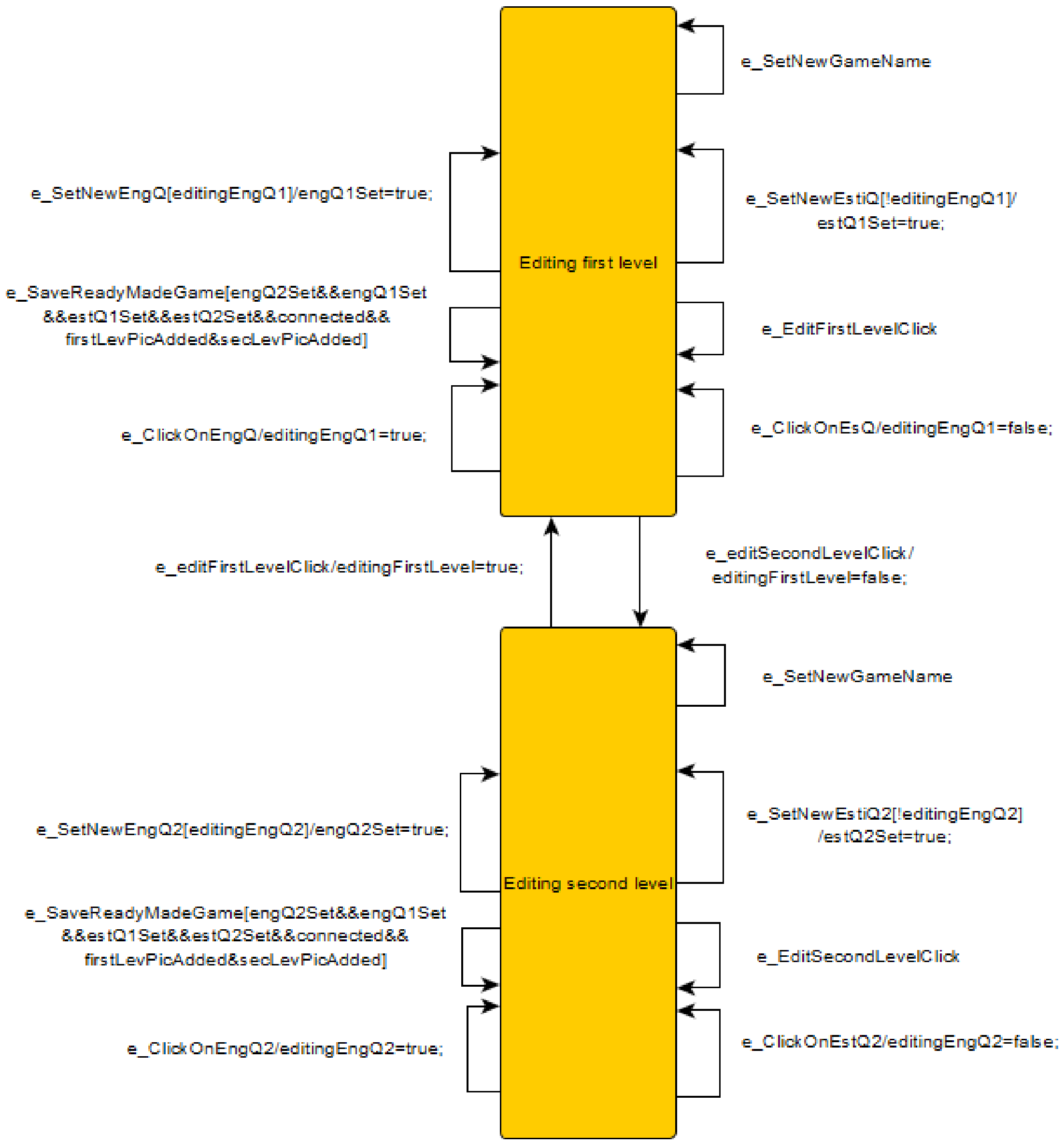

e_SetNewGameName
e_SetNewEngQ[editingEngQ1]/engQ1Set=true;
e_SetNewEstiQ[!editingEngQ1]/estQ1Set=true;
Editing first level
e_SaveReadyMadeGame[engQ2Set&&engQ1Set&&estQ1Set&&estQ2Set&&connected&&firstLevPicAdded&secLevPicAdded]
e_EditFirstLevelClick
e_ClickOnEngQ/editingEngQ1=true;
e_ClickOnEsQ/editingEngQ1=false;
e_editFirstLevelClick/editingFirstLevel=true;
e_editSecondLevelClick/editingFirstLevel=false;
e_SetNewGameName
e_SetNewEngQ2[editingEngQ2]/engQ2Set=true;
e_SetNewEstiQ2[!editingEngQ2]/estQ2Set=true;
Editing second level
e_SaveReadyMadeGame[engQ2Set&&engQ1Set&&estQ1Set&&estQ2Set&&connected&&firstLevPicAdded&secLevPicAdded]
e_EditSecondLevelClick
e_ClickOnEngQ2/editingEngQ2=true;
e_ClickOnEstQ2/editingEngQ2=false;

Figure 2.3: VR Quiz editing game state machine.

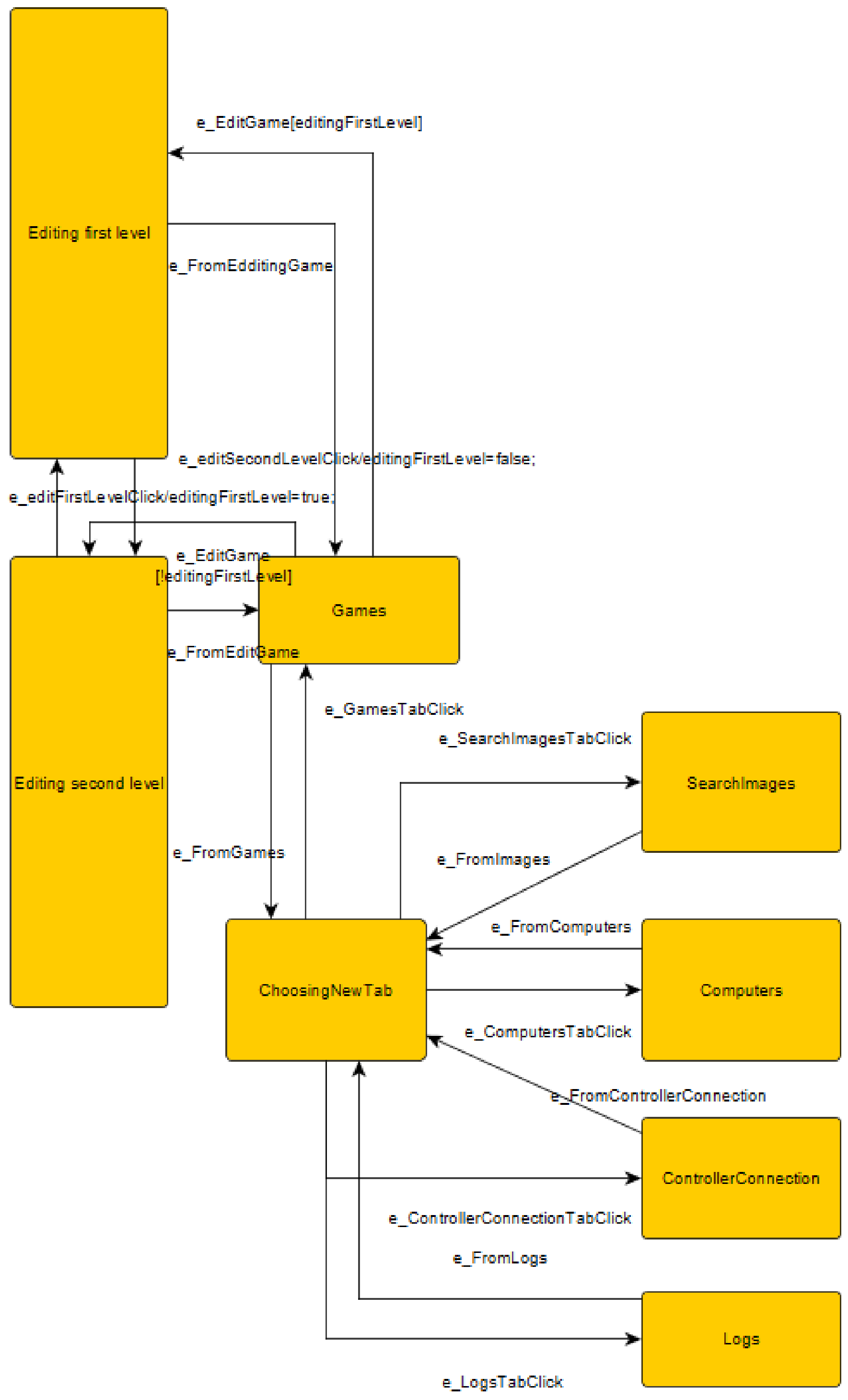


Figure 2.4: VR Quiz moving between tabs state machine.

program, which is written in Spec# or C#, which defines variables and update rules of the state machine. Each state is defined by the snapshot of the variables. Spec explorer uses the variables and provided rules to generate states [12].

Spec explorer features [12]:

1. Creating of state space from a specification.
2. Online testing
3. Offline testing
4. Dynamic object creation
5. State space exploration, which means one can see the generated state space.

Spec Explorer can be very difficult to use for games testing, since a tester can not directly specify the states of the system and test scenarios. The states and scenarios are generated based on restrictions written in code like parameter generators, state filters, method restriction, state grouping, etc [12]. All these restrictions will mount up to a quite large overhead, when using Spec Explorer for a larger project. Furthermore it would not be possible to see the whole state space even with a small project as VR Quiz, because every variable change is a new state, which would create a major state explosion in the model. The state explosion is avoided when using Graphwalker or another testing tool TestCast MBT, since the tester has direct control over the state machine unlike Spec Explorer, which uses restrictions in code.

2.3.3 NModel

NModel is another model-based testing and analysis framework tool written in C#. Similarily to Spec Explorer a model program has to be written in C#, which is then used by the NModel to generate the model. The tool also supports both online and offline testing and visualization of the generated model. NModel is open source.

Part of the VR Quiz was modelled and tested with NModel to better understand the program. The same two models 2.6, 2.7 were created with NModel and part of the generated model can be seen in figure 2.9.

NModel has a limitation to the number of transitions it will generate for offline testing. This issue can also seen in figure 2.9 as there should not be any final states in the model. From states 2, 5, 8, 11, 13, 9, 10, 44, 43, 12, 15, 6, 3 there should be transitions to other states. Because of that limitation only online testing was done with NModel. This means that when testing VR Quiz with NModel additional code has to be written to make sure that all the transitions have been covered.

The issue of state explosion when the model is generated from restrictions in code was also discovered when trying to generate offline tests for VR Quiz. With the NModel tool, that visualises the model, it was possible to increase the maximum number of transitions generated and thus create a model, which did not have such final states as described earlier. The created visual model had about 1300 states and took about two minutes to render even with the very small part of the VR Quiz menu modelled.

For testing VR Quiz with NModel, the Unity testing framework described in section Unity testing framework was used. The framework provided the means to control the SUT and to get information about the state of the SUT.

The behaviour of the SUT was described similarly to that described in section Menu.

Another problem with NModel is that it shows the whole model in one graph. This makes reading even a simple model almost impossible and therefore with NModel one can not visually confirm that the generated model is correct and the one tester wanted to create. Nevertheless NModel provides readability checks, which can be used to visually confirm that some state are reachable.

2.3.4 TestCast MBT

TestCast MBT is a commercial eclipse [13] based tool, which uses TTCN-3 as the test executor and is developed by Elvior [14]. It has an easy to use drag and drop modelling GUI similar that of Graphwalker and yEd.

TestCast MBT features:

1. Offline testing. The tool generates a TTCN-3 script from the model and test objectives, which could be run any number of times.

2. Hierarchical modelling capability and extended finite state machines.

3. Support for TTCN-3 and Robot automation framework.

Boolean and integer type variables defined in TestCast and TTCN-3 can be used in the state diagrams as guard conditions for the transitions. The variables can be updated and changed also in the transitions. This means that unlike in Graphwalker and yEd TestCast MBT allows to use the same variable over different state diagrams. This means that there is no longer any limitations to using several graphs to model the SUT.

Furthermore with sub-machines it is much easier to model complex interactions in the case of menus, where many states can be reached from one state. Figures 2.5, 2.6 and 2.7 show the use and benefit of sub-machines. The figures depict modelling of a part of the menu of VR Quiz. In figure 2.7 the state diagram for the functionality of editing the first level of the game is shown. In the menu the user can click on English question, set new English question, click on Estonian question and choose new Estonian question. Those actions can be also seen in figure 2.7.

VR Quiz also allows editing a second level and some of the transitions, that can be traversed while editing the second level like starting creating a new game, choosing a new name for the game and saving the game, can also be done in the first level. Using sub-machines it is possible to model the latter transitions only once. It works so that all the states, which are defined in the upper layer of the sub-machine, are also connected to all the states defined in the lower levels. In this case the figure 2.5 shows the upper level of the sub-machine and all the transitions defined there can be traversed both in states Editing first level and Editing second level.

Using TTCN-3 can be quite complicated and an adapter should be created that allows communication between the test executor and the SUT, but other than that almost no programming skills are needed when using TestCast MBT, since the program generates the test cases. Furthermore compared to Spec Explorer and NModel the states and transitions are defined using a drag and drop GUI making it much simpler to use, because the model is visible to the user at all times. The tool also allows controlling the messages sent and expected responses from the SUT in the same GUI. Therefore beside creating an adapter only the test data should be defined and handled using TTCN-3.

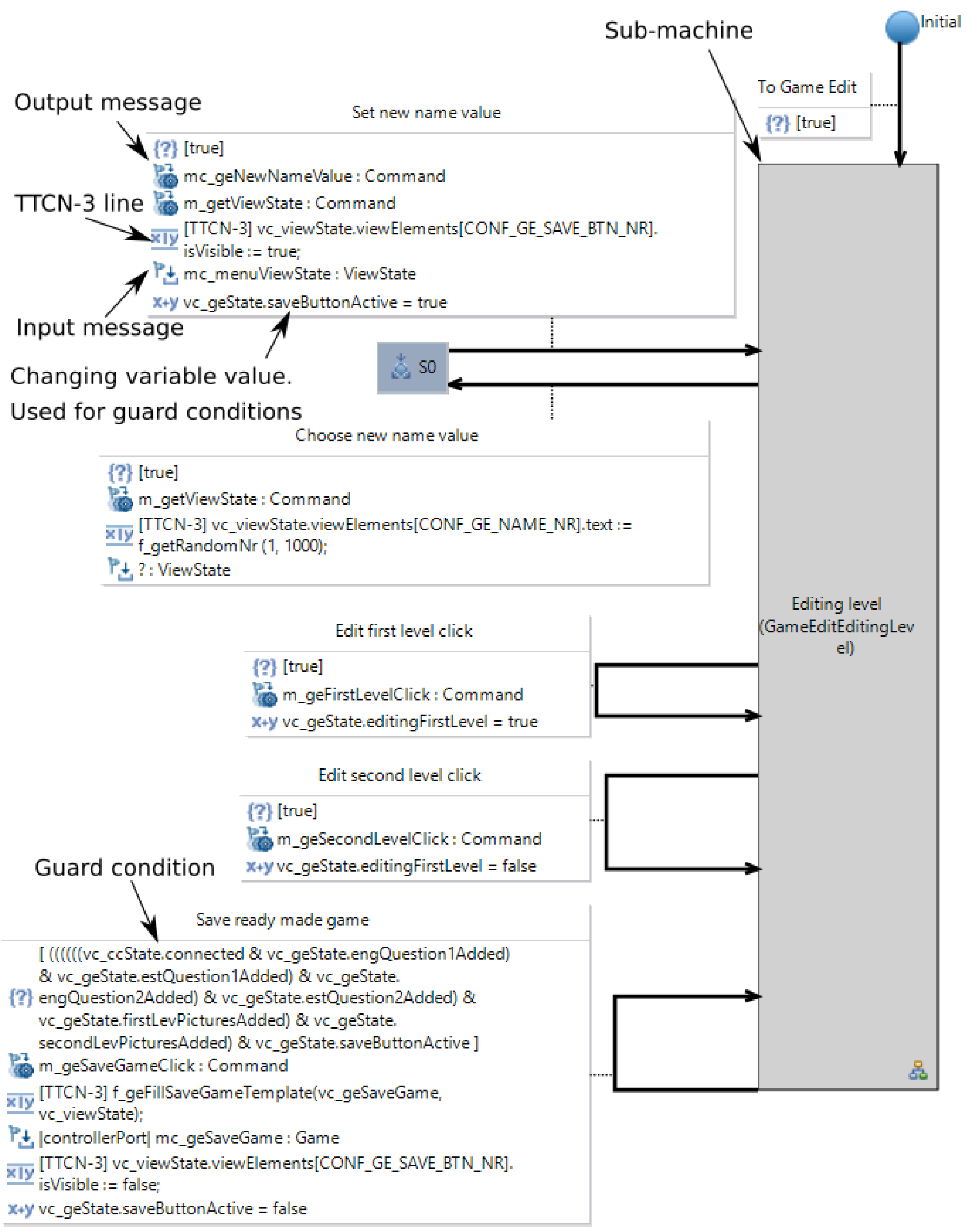
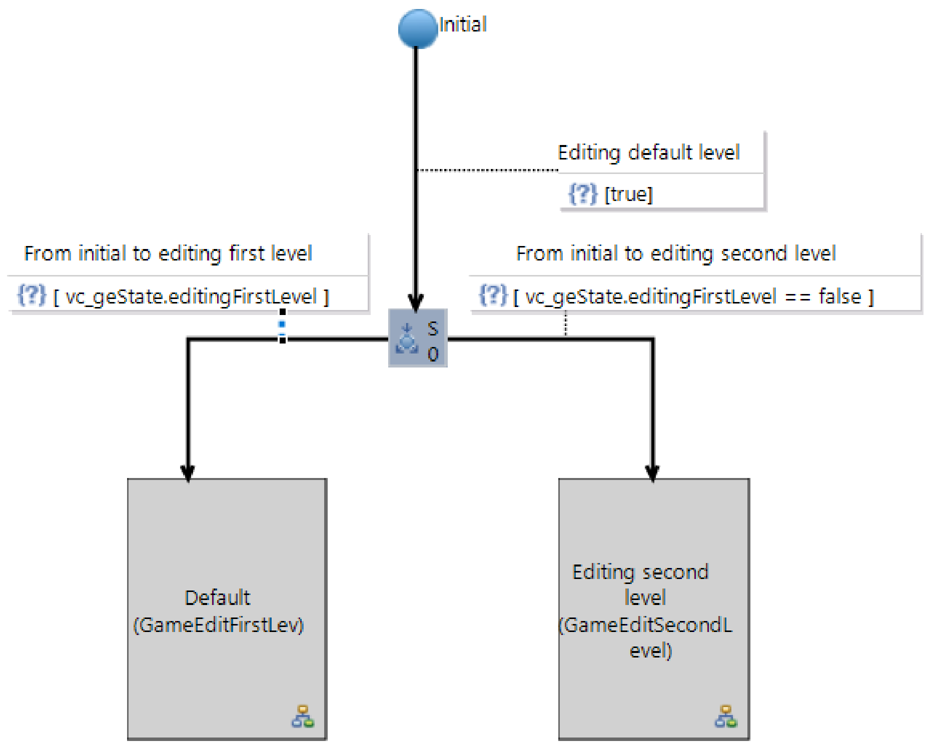
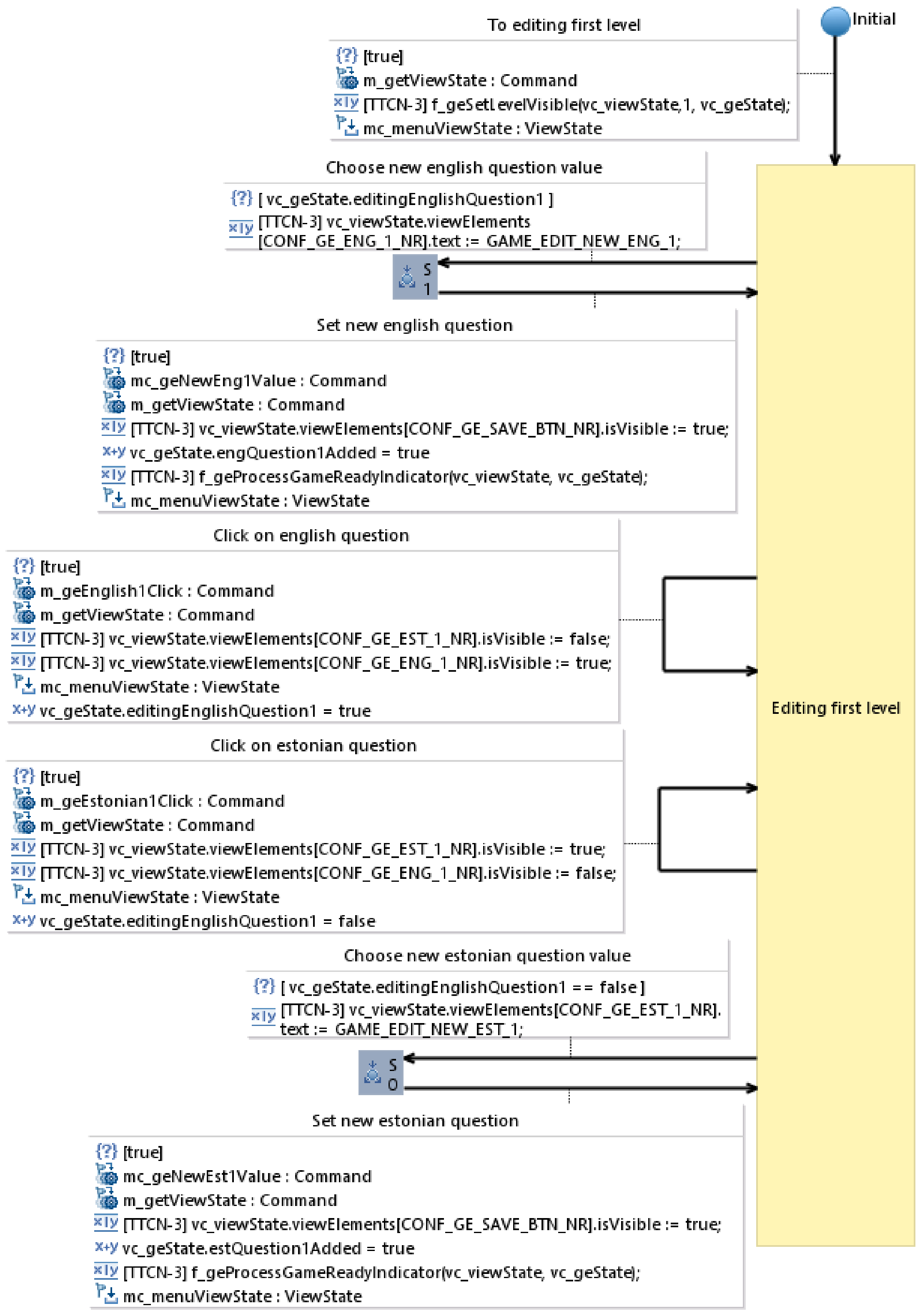

Sub-machine
Initial
To Game Edit
{?} [true]
Output message
Set new name value
{?} [true]
mc_geNewNameValue : Command
TTCN-3 line
m_getViewState : Command
[TTCN-3] vc_viewState.viewElements[CONF_GE_SAVE_BTN_NR].isVisible := true;
mc_menuViewState : ViewState
x+y vc_geState.saveButtonActive = true
Input message
S0
Changing variable value.
Used for guard conditions
Choose new name value
{?} [true]
m_getViewState : Command
[TTCN-3] vc_viewState.viewElements[CONF_GE_NAME_NR].text := f_getRandomNr (1, 1000);
? : ViewState
Edit first level click
{?} [true]
m_geFirstLevelClick : Command
x+y vc_geState.editingFirstLevel = true
Editing level (GameEditEditingLevel)
Edit second level click
{?} [true]
m_geSecondLevelClick : Command
x+y vc_geState.editingFirstLevel = false
Guard condition
Save ready made game
{?} [((((((vc_ccState.connected & vc_geState.engQuestion1Added) & vc_geState.estQuestion1Added) & vc_geState.engQuestion2Added) & vc_geState.estQuestion2Added) & vc_geState.firstLevPicturesAdded) & vc_geState.secondLevPicturesAdded) & vc_geState.saveButtonActive]
m_geSaveGameClick : Command
[TTCN-3] f_geFillSaveGameTemplate(vc_geSaveGame, vc_viewState);
|controllerPort| mc_geSaveGame : Game
[TTCN-3] vc_viewState.viewElements[CONF_GE_SAVE_BTN_NR].isVisible := false;
x+y vc_geState.saveButtonActive = false

Figure 2.5: VR Quiz edit game model.



Initial
Editing default level
{?} [true]
From initial to editing first level
{?} [vc_geState.editingFirstLevel]
From initial to editing second level
{?} [vc_geState.editingFirstLevel == false]
S
0
Default (GameEditFirstLev)
Editing second level (GameEditSecondLevel)

Figure 2.6: VR Quiz edit level model.



```
To editing first level
{?} [true]
m_getViewState : Command
x|y [TTCN-3] f_geSetLevelVisible(vc_viewState,1, vc_geState);
mc_menuViewState : ViewState

Choose new english question value
{?} [ vc_geState.editingEnglishQuestion1 ]
x|y [TTCN-3] vc_viewState.viewElements
[CONF_GE_ENG_1_NR].text := GAME_EDIT_NEW_ENG_1;

Set new english question
{?} [true]
mc_geNewEng1Value : Command
m_getViewState : Command
x|y [TTCN-3] vc_viewState.viewElements[CONF_GE_SAVE_BTN_NR].isVisible := true;
x+y vc_geState.engQuestion1Added = true
x|y [TTCN-3] f_geProcessGameReadyIndicator(vc_viewState, vc_geState);
mc_menuViewState : ViewState

Click on english question
{?} [true]
m_geEnglish1Click : Command
m_getViewState : Command
x|y [TTCN-3] vc_viewState.viewElements[CONF_GE_EST_1_NR].isVisible := false;
x|y [TTCN-3] vc_viewState.viewElements[CONF_GE_ENG_1_NR].isVisible := true;
mc_menuViewState : ViewState
x+y vc_geState.editingEnglishQuestion1 = true

Click on estonian question
{?} [true]
m_geEstonian1Click : Command
m_getViewState : Command
x|y [TTCN-3] vc_viewState.viewElements[CONF_GE_EST_1_NR].isVisible := true;
x|y [TTCN-3] vc_viewState.viewElements[CONF_GE_ENG_1_NR].isVisible := false;
mc_menuViewState : ViewState
x+y vc_geState.editingEnglishQuestion1 = false

Choose new estonian question value
{?} [ vc_geState.editingEnglishQuestion1 == false ]
x|y [TTCN-3] vc_viewState.viewElements[CONF_GE_EST_1_NR].
text := GAME_EDIT_NEW_EST_1;

Set new estonian question
{?} [true]
mc_geNewEst1Value : Command
m_getViewState : Command
x|y [TTCN-3] vc_viewState.viewElements[CONF_GE_SAVE_BTN_NR].isVisible := true;
x+y vc_geState.estQuestion1Added = true
x|y [TTCN-3] f_geProcessGameReadyIndicator(vc_viewState, vc_geState);
mc_menuViewState : ViewState
```

Figure 2.7: VR Quiz edit first level model.

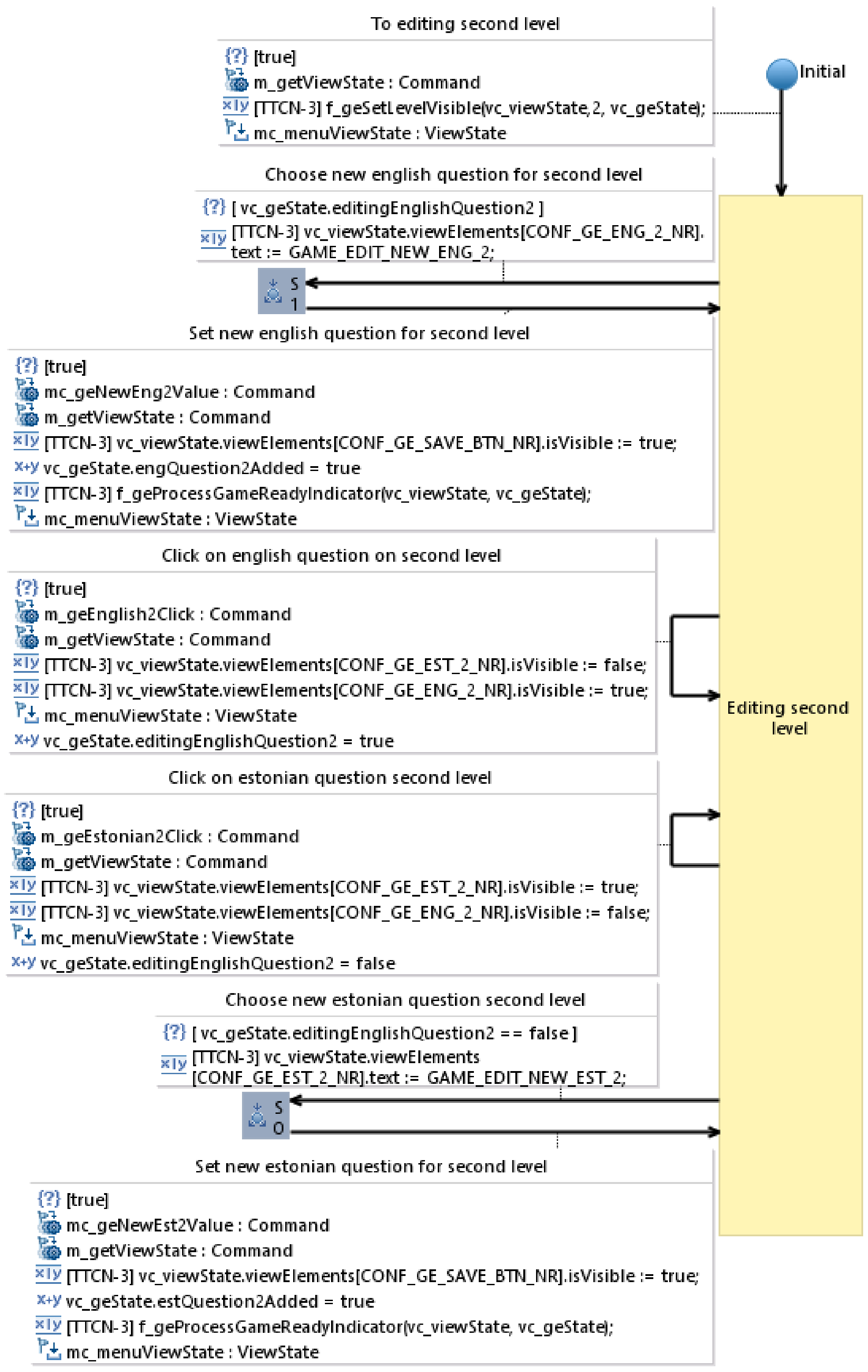

To editing second level
[true]
m_getViewState : Command
[TTCN-3] f_geSetLevelVisible(vc_viewState,2, vc_geState);
mc_menuViewState : ViewState
Initial
Choose new english question for second level
[vc_geState.editingEnglishQuestion2]
[TTCN-3] vc_viewState.viewElements[CONF_GE_ENG_2_NR].
text := GAME_EDIT_NEW_ENG_2;
S
1
Set new english question for second level
[true]
mc_geNewEng2Value : Command
m_getViewState : Command
[TTCN-3] vc_viewState.viewElements[CONF_GE_SAVE_BTN_NR].isVisible := true;
vc_geState.engQuestion2Added = true
[TTCN-3] f_geProcessGameReadyIndicator(vc_viewState, vc_geState);
mc_menuViewState : ViewState
Click on english question on second level
[true]
m_geEnglish2Click : Command
m_getViewState : Command
[TTCN-3] vc_viewState.viewElements[CONF_GE_EST_2_NR].isVisible := false;
[TTCN-3] vc_viewState.viewElements[CONF_GE_ENG_2_NR].isVisible := true;
mc_menuViewState : ViewState
vc_geState.editingEnglishQuestion2 = true
Editing second level
Click on estonian question second level
[true]
m_geEstonian2Click : Command
m_getViewState : Command
[TTCN-3] vc_viewState.viewElements[CONF_GE_EST_2_NR].isVisible := true;
[TTCN-3] vc_viewState.viewElements[CONF_GE_ENG_2_NR].isVisible := false;
mc_menuViewState : ViewState
vc_geState.editingEnglishQuestion2 = false
Choose new estonian question second level
[vc_geState.editingEnglishQuestion2 == false]
[TTCN-3] vc_viewState.viewElements
[CONF_GE_EST_2_NR].text := GAME_EDIT_NEW_EST_2;
S
0
Set new estonian question for second level
[true]
mc_geNewEst2Value : Command
m_getViewState : Command
[TTCN-3] vc_viewState.viewElements[CONF_GE_SAVE_BTN_NR].isVisible := true;
vc_geState.estQuestion2Added = true
[TTCN-3] f_geProcessGameReadyIndicator(vc_viewState, vc_geState);
mc_menuViewState : ViewState

Figure 2.8: VR Quiz edit second level model.

2.4 TTCN-3

To evaluate TestCast MBT further, the TTCN-3 is compared against general purpose languages like Java or C#, which are used by the Spec Explorer, Graphwalker and NModel.

The Testing and Control Notation Version 3 (TTCN-3) is a formal language, which is specifically tailored for testing. It is developed and maintained by the European Telecommunication Standards Institude (ETSI). It has been used since 2000 initially for testing communication protocols, but it has become a universal testing language that is also used in e.g. automotive and medical domain [15].

TTCN-3 is used for black box testing and it features a rich typing system and powerful matching mechanisms. It supports message and procedure based communication, timer handling, dynamic test configuration and concurrent test behaviour. It also has a built in test oracle. TTCN-3 is being continuously improved and extended [16].

Since TTCN-3 is made for testing, it provides constructs, that are not found in general purpose programming languages. One of them is its unique data matching mechanism for comparing incoming data against test expectations. Because the language is standardized and has a well defined semantics, it is not possible for one tool vendor to lock in as a sole provider for the tool. The tests written in one tool can be used in another [15].

Therefore a testing approach using TTCN-3 is superior to one that uses general purpose language like Java or C#. Furthermore since TTCN-3 is less complex than Java or C#, it is much easier for a tester to learn and use. Complexity is one of the major issues with automated tools for testing.

2.5 Conclusion

Model-based testing seems to be most optimal for automated testing of games, because it is a systematic approach providing a large coverage, which is needed for discovering medium to minor bugs. Furthermore Unity developers often use state machines during the development of the games and it could be possible to use the created state machines to help with testing. Additionally MBT provides a high abstract level, which is needed for testing a game in development, since the latter is being modified rapidly.

TestCast MBT is best suited of the three tools since it is much easier to use than Spec Explorer and compared to Graphwalker allows more complex models to be created. Furthermore with TestCast MBT the model can be viewed in a much better way than with NModel. TestCast MBT creates the model of the SUT directly from the user defined EFSM and not generated it based on restrictions in code as with Spec Explorer and NModel. This means that in case of a SUT, which almost any state can be reached from any other state like in the case of a menu, state explosion can easily be avoided and it is possible to do offline testing. The NModel tool, which generates the test, does not support changing the maximum number of transitions.

The drawbacks of TestCast MBT are that it is a commercial tool, but the author of this thesis was able to use it for free for this thesis. It also lacks online testing, but it is not a must-have in state transition testing.

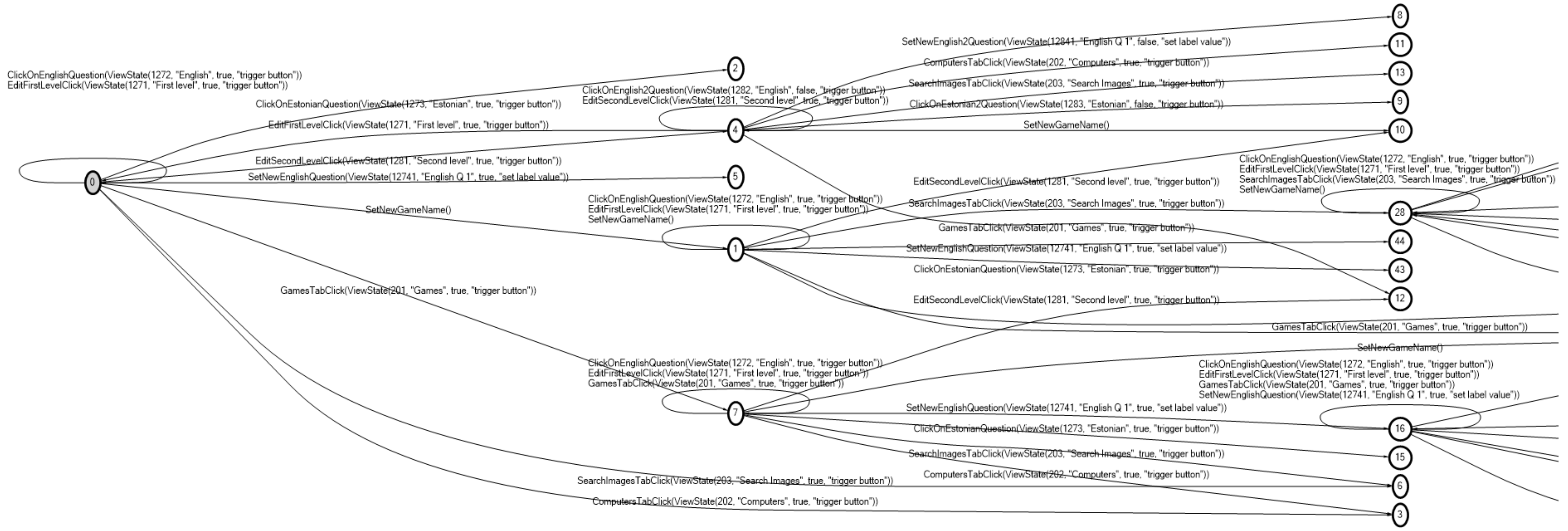

ClickOnEnglishQuestion(ViewState(1272, "English", true, "trigger button"))
EditFirstLevelClick(ViewState(1271, "First level", true, "trigger button"))
ClickOnEstonianQuestion(ViewState(1273, "Estonian", true, "trigger button"))
EditFirstLevelClick(ViewState(1271, "First level", true, "trigger button"))
EditSecondLevelClick(ViewState(1281, "Second level", true, "trigger button"))
SetNewEnglishQuestion(ViewState(12741, "English Q 1", true, "set label value"))
SetNewGameName()
GamesTabClick(ViewState(201, "Games", true, "trigger button"))
SearchImagesTabClick(ViewState(203, "Search Images", true, "trigger button"))
ComputersTabClick(ViewState(202, "Computers", true, "trigger button"))
ClickOnEnglish2Question(ViewState(1282, "English", false, "trigger button"))
EditSecondLevelClick(ViewState(1281, "Second level", true, "trigger button"))
SetNewEnglish2Question(ViewState(12841, "English Q 1", false, "set label value"))
ComputersTabClick(ViewState(202, "Computers", true, "trigger button"))
SearchImagesTabClick(ViewState(203, "Search Images", true, "trigger button"))
ClickOnEstonian2Question(ViewState(1283, "Estonian", false, "trigger button"))
SetNewGameName()
ClickOnEnglishQuestion(ViewState(1272, "English", true, "trigger button"))
EditFirstLevelClick(ViewState(1271, "First level", true, "trigger button"))
SetNewGameName()
EditSecondLevelClick(ViewState(1281, "Second level", true, "trigger button"))
SearchImagesTabClick(ViewState(203, "Search Images", true, "trigger button"))
GamesTabClick(ViewState(201, "Games", true, "trigger button"))
SetNewEnglishQuestion(ViewState(12741, "English Q 1", true, "set label value"))
ClickOnEstonianQuestion(ViewState(1273, "Estonian", true, "trigger button"))
EditSecondLevelClick(ViewState(1281, "Second level", true, "trigger button"))
GamesTabClick(ViewState(201, "Games", true, "trigger button"))
ClickOnEnglishQuestion(ViewState(1272, "English", true, "trigger button"))
EditFirstLevelClick(ViewState(1271, "First level", true, "trigger button"))
SearchImagesTabClick(ViewState(203, "Search Images", true, "trigger button"))
SetNewGameName()
ClickOnEnglishQuestion(ViewState(1272, "English", true, "trigger button"))
EditFirstLevelClick(ViewState(1271, "First level", true, "trigger button"))
GamesTabClick(ViewState(201, "Games", true, "trigger button"))
SetNewGameName()
ClickOnEnglishQuestion(ViewState(1272, "English", true, "trigger button"))
EditFirstLevelClick(ViewState(1271, "First level", true, "trigger button"))
GamesTabClick(ViewState(201, "Games", true, "trigger button"))
SetNewEnglishQuestion(ViewState(12741, "English Q 1", true, "set label value"))
SetNewEnglishQuestion(ViewState(12741, "English Q 1", true, "set label value"))
ClickOnEstonianQuestion(ViewState(1273, "Estonian", true, "trigger button"))
SearchImagesTabClick(ViewState(203, "Search Images", true, "trigger button"))
ComputersTabClick(ViewState(202, "Computers", true, "trigger button"))
0
1
2
3
4
5
6
7
8
9
10
11
12
13
15
16
28
43
44

Figure 2.9: Partially unrolled labelled transition system generated by a tool of NModel.

3. Proposed testing approach for Unity games

A testing approach using TestCast MBT is proposed as a solution for automated system testing of games made with Unity. Furthermore a Unity testing framework was developed, which enables test executor to control the behaviour and to check the state of the SUT.

The testing approach consists of four layers as shown in figure 3.1. Each of the layers are separate programs. The SUT is modelled using TestCast MBT, which uses data from the test executor to generate test sequences and the test scripts. The test executor runs the generated scripts and communicates with the SUT via the adapter.

The adapter and the scripts in Unity are part of the developed Unity testing framework. The adapter consists of a module for communicating with the test executor according to the TTCN-3 standard. Furthermore it is able to send messages to the SUT and to the test manager using socket connection. The test manager is a script running in the SUT, which controls the behaviour and returns information about the state of the SUT to the adapter. For each of the objects in the game, which state needs to be checked or controlled, a separate script has to be attached.

3.1 Unity testing framework

To further reduce the time and effort for testing a Unity testing framework was developed. It is used in the testing of VR Quiz, but it provides a high enough abstraction, that it should be possible to also test other systems made with Unity. Furthermore it can be used with test executor other than TestCast and TTCN-3. The framework consists of a .NET project and scripts, which should be added to the SUT in the Unity environment. The .NET project and DLL-s provides the tester with an API, which could be used with .NET projects.

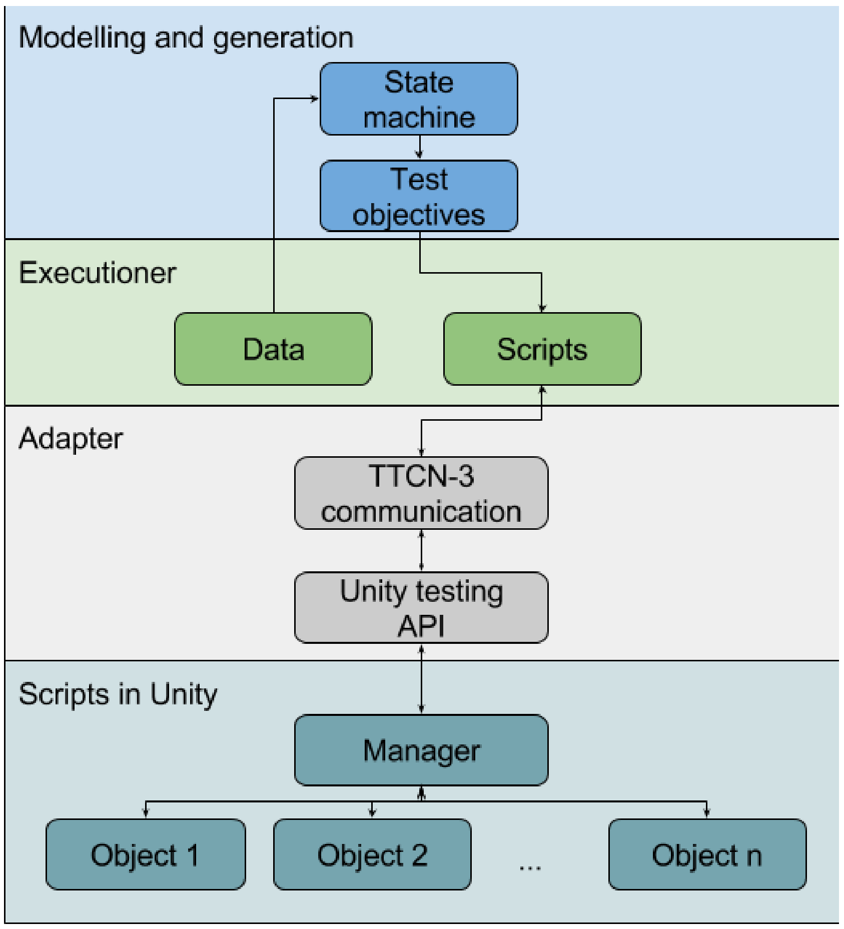


Figure 3.1: Architecture of the proposed testing approach.

The framework is open source and can be downloaded from [1].

3.1.1 .NET API

The API uses socket connection to communicate with the scripts added to the SUT. To use the framework the UnityTesting.dll should be added to the test executor or its adapter. The .dll gives access to UnityTesting namespace which has two classes: TesterConnectionHandler and AdapterViewState.

The messages going back and fourth are structured so that the first 48 bytes is a string and the next bytes are one or several of the structures seen in code example 3.1.

```
[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Ansi)]
public struct ViewStateStructure
{
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 24)]
    public string id;
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 24)]
    public string flags;
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 512)]
    public string text;
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 12)]
    public string position;
}
```

Code example 3.1: Structure used in communication in the Unity testing framework

All the tested objects should have a unique ID. Currently only one flag is used and that states whether the object is visible or not. The text is either any text associated with the object or new text to be set for the object. The position was not used in testing of VR Quiz.

The AdapterViewState, which is part of the UnityTesting namespace, packages the ViewStateStructure and the message can be accessed using this class.

The TesterConnectionHandler handles the connection with the SUT. It is a singleton to prevent multiple connections to be made over the same port and provides five methods for the tester. All of them send a message to the SUT. They are:

- SendGetViewState. This method takes no parameter. It sends a message to the Unity asking for the view state of the SUT.
- SendTriggerButton. This takes an AdapterViewState object as a parameter. The method sends a message to the SUT to trigger the object with the same ID as the parameter's ID. This method can be used to trigger buttons in a menu or to click on objects in a game.
- SendSetLabelValue. This method also takes an AdapterViewState object as a parameter. This methods send a message to the SUT to set a new value to an object in the game with the same ID as the parameter's ID. The new value to be set is the Text property of the parameter.
- SendMessageToUnity. This method takes two parameters. One is a string and the other one is AdapterViewState object. The string is the command that the SUT should do. This method can be used the same way as the previous three, if the command is set to "get view state", "trigger button" or "set label value". It can further be used for creating custom commands, which functionality can be implemented in the Test Manager of the SUT.
- SendData. This method takes two parameters: a string and a byte array. This method can be used to send any data to the SUT's connection handler. It is possible to write additional code in the SUT's connection handler, which is able to use the data. The frameworks' connection handler is able to send messages of any length. When the message is very large it splits the message up to smaller pieces and sends them one by one.

Even though only the ID is required in case of triggering a button, the whole ViewStateStructure is sent to make the communication more robust.

3.1.2 Unity testing manager

Figure 3.2 shows the class diagram of the Unity part of the testing framework.

The tester is connected with the SUT via the TestingConHandler script. It decodes the incoming messages and triggers an event based on the command of the message.

The events are picked up by the TestingManager script, which executes the received command. TestingManager and TestingConHandler scripts are singletons so that only one of them can be running. The TestingManager controls the SUT using TestObject scripts, which should be attached to the objects, which need controlling or which states are tested.

In Unity it is possible to divide the game into different scenes, which are for example different areas in the game. Each scene has its own objects and only static script instances are the same between each scene. To handle the different objects between each scene the TestObjManager was created, which gives access of the TestObjects to the TestManager. Each scene needs a new instance of the TestObjManager and all the test objects have to be added to the TestObjManager. An example of the TestObjManager in Unity editor can be seen in figure 3.3.

The TestSubManager is used when new objects are added to the game at run time. Each time the TestObjects are requested from the TestSubManager, it scans the child components of the object the script is attached to for TestObjects and returns them.

Since triggering, setting a new state or checking the properties of unique objects are done differently, the TestObject class should allow the tester to use it as a base class for implementing new scripts. For example triggering a button in a menu is done differently than triggering an object in a game. Therefore a new script should be created for a menu button, which is a superclass to the TestObject and the method for triggering the button should be overwritten.

The virtual methods or properties, which can be overwritten in the TestObject class, are these:

- IsVisible - This property sets the visibility flag of the ViewStateStructure (see code example 3.1).
- GetParameter - Sets the "text" of the ViewStateStructure.
- GetViewState - Returns UnityViewState. Can be used to implement new flags or the position property of the ViewStateStructure.
- PreGetViewState - Can be used when some preprocessing of the object is needed before getting its state.
- Trigger - Used by the TestManager to trigger the object, when the "trigger button" (see section .NET API) command is received.

- SetText - Used by the TestManager to set a new value to the object, when the "set label value" (see section .NET API) command is received.

Even though adding a test script one by one to each of the objects in Unity can be time consuming, it is superior to that of asking for all the objects in the scene and checking their parameters. The latter case would create too much useless noise in the response of the SUT since most of the objects are part of the environment and not needed for testing. Furthermore using dedicated test scripts offer a higher level of abstraction and the ability to better define how a state property is checked. For example in case of testing VR Quiz to check whether an object is visible, it was needed to turn the avatar so that is would face the object being tested (see section Unity test objects).

3.2 Conclusion

The proposed approach for testing solves two of the most important problems with model-based testing of games: the tools are too complex and take too long to learn and tools do not provide enough modelling capability. TestCast MBT supports variables over several models and hierarchical states, which help out modelling more complex finite-state machines. The solution is easy to use since TestCast MBT has a simple drag and drop GUI for modelling. Furthermore it generates much of the more complex code like sending, receiving and matching messages.

The Unity testing framework, which was developed for this thesis, takes care of the adapter and communicating with the SUT. The tester only needs to specify the data and changing of the data after each transition on the test execution side. On SUT side the framework provides scripts to help with getting the state of the system and for controlling the system remotely. The scripts are abstract enough to test different systems with the framework. For these reasons the proposed testing method is rather easy to use for a tester and do not need much learning.

The framework can be used with different test executors as it was used to test VR Quiz with TestCast and with NModel.

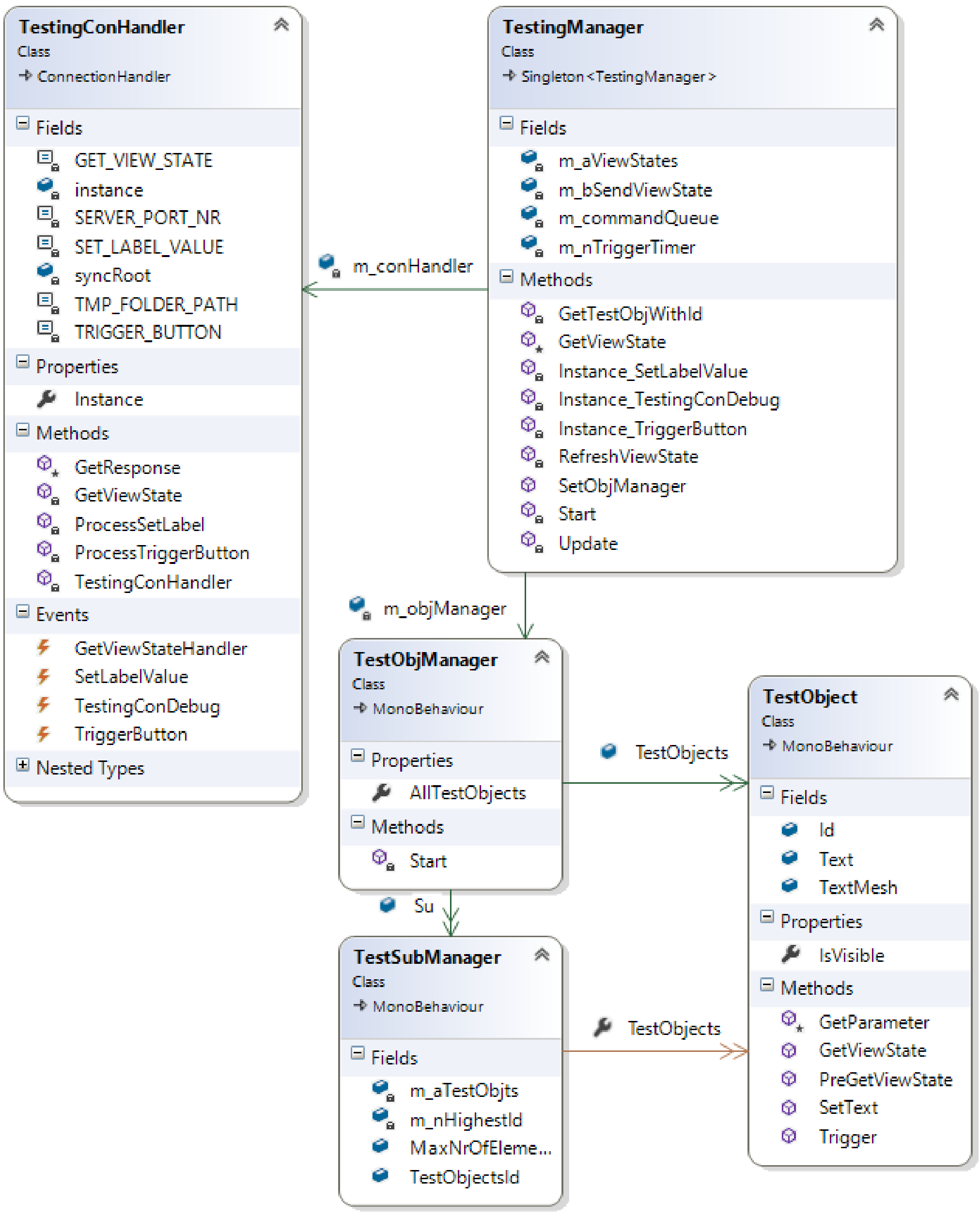
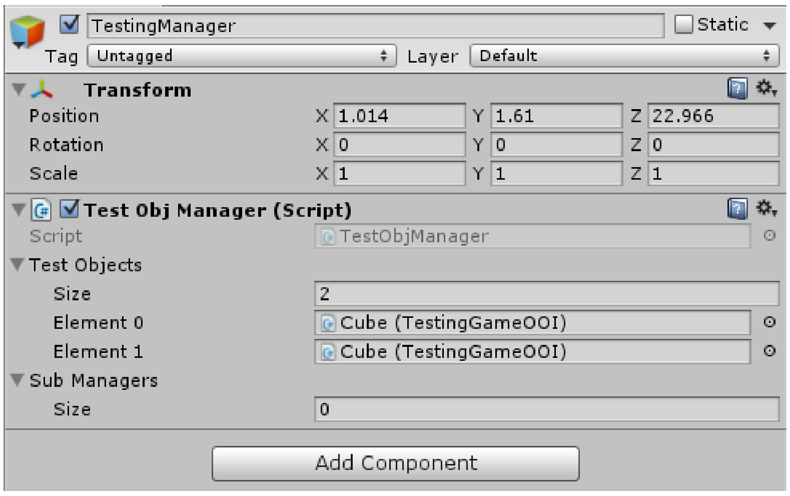


Figure 3.2: Class diagram of the Unity part of the testing framework.



TestingManager Static

Tag Untagged Layer Default

Transform

Position	X 1.014	Y 1.61	Z 22.966
Rotation	X 0	Y 0	Z 0
Scale	X 1	Y 1	Z 1

Test Obj Manager (Script)

Script	TestObjManager
Test Objects	
Size	2
Element 0	Cube (TestingGameOOI)
Element 1	Cube (TestingGameOOI)
Sub Managers	
Size	0

Figure 3.3: An example TestObjManager script in Unity editor.

4. Testing of VR Quiz

As a proof of concept the proposed testing approach is used for testing VR Quiz, which is in turn also developed by the author of the thesis.

4.1 VR Quiz

VR Quiz is a system, which is meant to be deployed in a museum and it provides an immersive experience for the museum visitors by using virtual reality.

VR Quiz is made of three different parts:

1. The Environment, what the museum visitor sees once he puts on the goggles. The environment is made using 3D models and can be for example a medieval bar as can be seen in figure 4.1.
2. The game, which the museum visitor plays while being in the environment. The game is very simple, where the user is given a question and a set of five pictures. The user has to guess which picture is meant by the question.
3. The control menu for the museum worker. With the menu the museum worker can set up games for the visitors to play. It can be used to set the questions and pictures for the game. It also allows the museum worker to see logs from played games. The logs include for example, when a game was played and how long it took to finish the game.

The architecture of the VR Quiz can be seen in figure 4.2. The VR Quiz consists of three different projects. The game and the menu, which were explained earlier, and the controller. All three programs can be set up on separate computers and they communicate

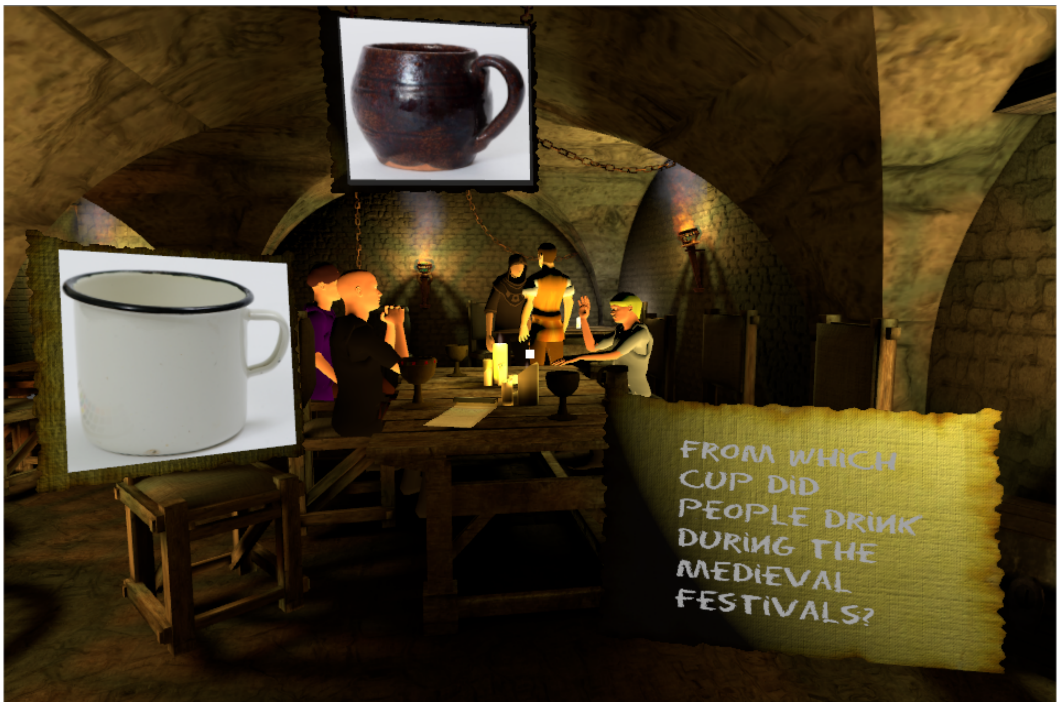


Figure 4.1: Caption from VR Quiz game.

with each other over a local network using socket connection. Each museum runs a separate instance of the controller, which communicates with a database. The database holds games made by the museum workers and images used by them, logs of played games and configured computers. Having a central controller allows the system to have several different menus connected and still use the same data and also allows for several computers to run a different game. At the moment the server is not implemented yet.

4.2 Development of VR Quiz and the test suite

VR Quiz is developed by four people. All the programming is done by the author of this thesis. The design for the environment, business model and PR is done by other members of the team.

In the beginning of the testing phase it was decided, that only the author of the thesis will do the testing, since he has all the necessary equipment and experience. No formal test plan document was to be created. Furthermore the black box method of testing for the menu and game was to be state transition testing. In addition to testing all the transitions, new test cases were to be generated from the model based on the specification of VR

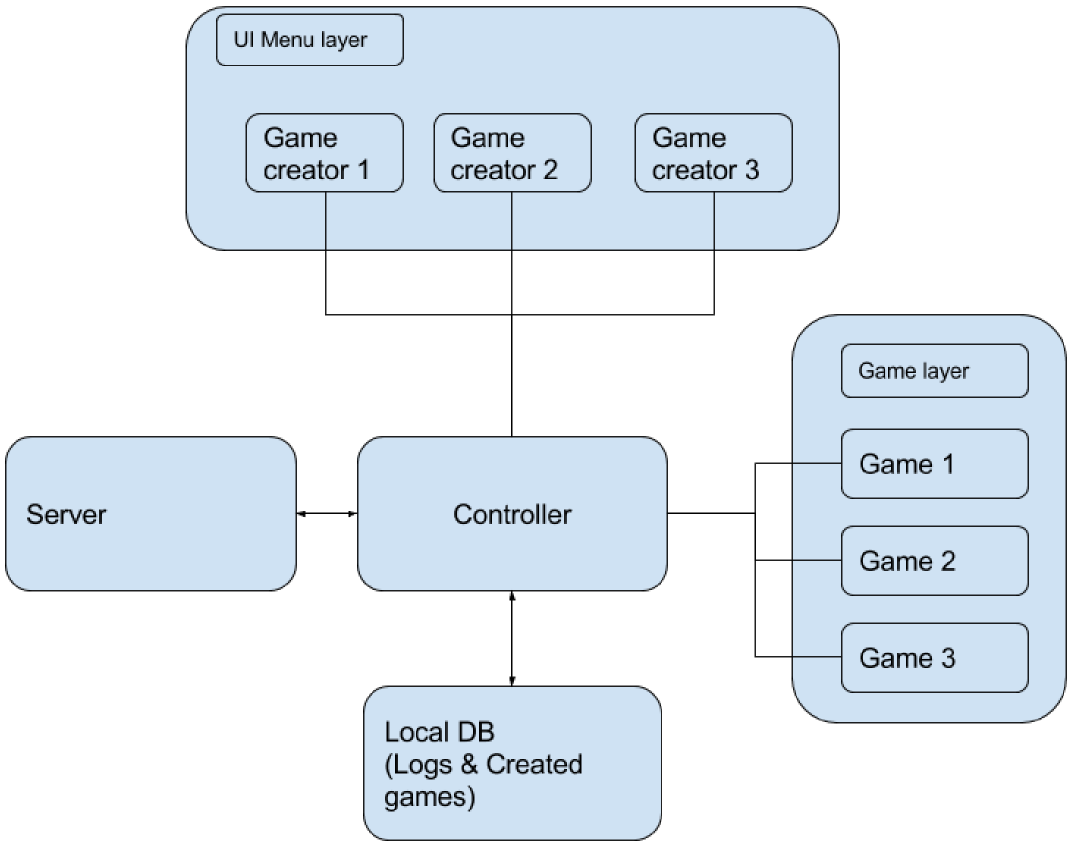


Figure 4.2: The architecture of the VR Quiz.

Quiz. From the number of stories covered in the specification the test coverage were to be evaluated.

Since the menu and the game send information to the controller, it was decided, that these programs were to be tested separately and that the communication with the controller was to be checked by mocking the behaviour of the controller.

Testing when the menu, controller and game are all working was to be done manually using test cases, to increase the test coverage from the automated tests. The menu and the game mostly work separately so automated testing for the whole system was not required. No separate automated testing was to be done for the controller since the controller is rather simple and the manual tests will cover most of its functionality.

Both the development of VR Quiz and its testing was done using agile principles with some artifacts from Scrum. All the principles of Scrum were not used since only one person did the actual development. All the requirements of VR Quiz and the test suite were added to the product backlog using the story format. Four types of items were added to the backlog depending on how detailed the description of the item was. The least detailed was a Feature, which would describe a requirement at a very high level. A feature was divided into epics and epics into stories, which in turn into tasks. A task was about 30 minutes to 6 hours of work. The backlog was written in Estonian and because of that only an example of a requirement in the backlog was translated and can be seen in Table 4.1.

The development was done using sprints, which were three weeks long. After every three weeks a sprint backlog was created and tasks were added to it from the product backlog. Story points were used to predict the time it would take to finish a task. The sum of these points were used in predicting how many tasks there should be in a sprint.

Developing of the test suite and systematic system testing started several months after developing of the VR Quiz started. It would have been better, if testing would have started at the same time, because the method chosen for testing was model-based and therefore a model-based approach to design and development would have helped greatly with the testing. For further development of the VR Quiz a model based development tool, Playmaker [9], will be used.

Nr	User story name	User story	SP	Type
10_0	Creating a game	As a worker in the museum I would like to be able to specify a game by myself, which is interesting and educational for museum workers	-	Feature
10_1	New game	I wish that I can start creating a new game and save with by pressing a button	-	Epic
10_1_1	Button	I wish for button, which can be pressed to start creating a game	-	Story
10_1_1_1	Button	Button for new game	2	Task
10_1_1_2	Button	Procedure for new game	8	Task
10_1_1_3	Button	Adding a default name	3	Task
10_1_2	Name	I wish to add a name to my game	-	Story
10_1_2_1	Name	Name input label	3	Task
10_1_2_2	Name	Saving the name change procedure	5	Task

Table 4.1: Example requirement from the VR Quiz backlog.

4.3 Modelling of VR Quiz

Since the menu and the game were to be tested separately while mocking the behaviour of the controller, two separate models were created. In both cases the models were created based on the complete system and not on the specification, since it is difficult to create an exact model from the user stories. Table 4.1 shows an exaple requirement from the VR Quiz backlog.

4.3.1 Menu

The different states of the menu are based on what can be seen in the menu. The transitions in the model are button presses or setting new values to input labels.

Every time a new button is pressed and the view state changes, the system is in a new state. But it would be unpractical to model the SUT this way, since the size of the state machine of the VR Quiz menu would rise exponentially. For example in case of the model in figure 2.7 there would be a new state when an English question is entered and Estonian question is not entered, English question is entered and Estonian question is entered, Estonian question is entered and English question is not entered etc.

To overcome this so called state explosion, the menu is modelled using a specific guideline, which is that the view state of the system is kept in one of the variables together with the state machine. In this case the variable is vc_viewState, which holds information of the current state of the system. Every time a transition is made, the variable vc_viewState is also updated. The variable is also used by the test oracle to check that the system is in a correct state. So the guideline is that for a transition the first thing to do is change the system, update the state variable and then to check that the system corresponds to the updated variable. This method of modelling can be seen in almost every transition in figure 2.7 and allowed the state machine to be more abstract and not have so many states in the model.

This method was also used when testing VR Quiz with NModel.

Another option would be to check only specific values, which were specified prior to running the tests in the test data, after each transition. For example checking the visibility of only one label when going to a new page. Using this method would remove updating the variable part from the transitions, but it would increase the size of the test data and the number of states in the model. Furthermore using the one variable increases the test coverage since every object in the SUT is tested every time.

4.3.2 Game

The VR Quiz game part is very simple. A user can only look around and select object with a click of a button. Furthermore there are a small number of places where the user is able to move. Thus the simplest way to model the VR Quiz game was to have the states as the different locations the user is in. The locations were: street area, inside the inn, jail, by the stairs, dining room and game finished.

With the method described in the section Menu it was possible to model the whole game using only these six states. These models can be seen in figures 4.3, 4.4, 4.6, 4.5.

In each of the states it is checked what a user can see and if a user can or can not click on some objects. For example when in the inn the test checks whether it is possible to click on one of the possible answers and checks that it is not possible to click on the stairs, which would lead the user straight to the next level. Testing whether it is not possible to click on certain objects in the game is important, because there was previously an error, which allowed the user to click on objects, which were supposed to be disabled.

Another option would have been to model the game based on what the user is doing. In VR Quiz the user looks around using a virtual reality headset. With the headset the user

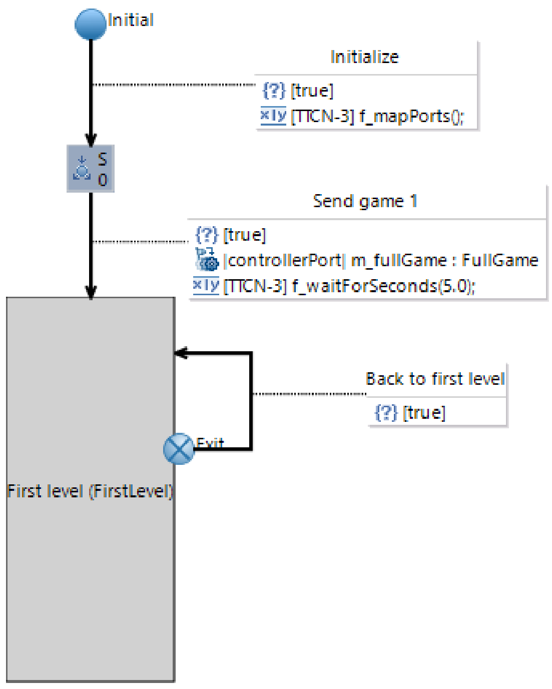


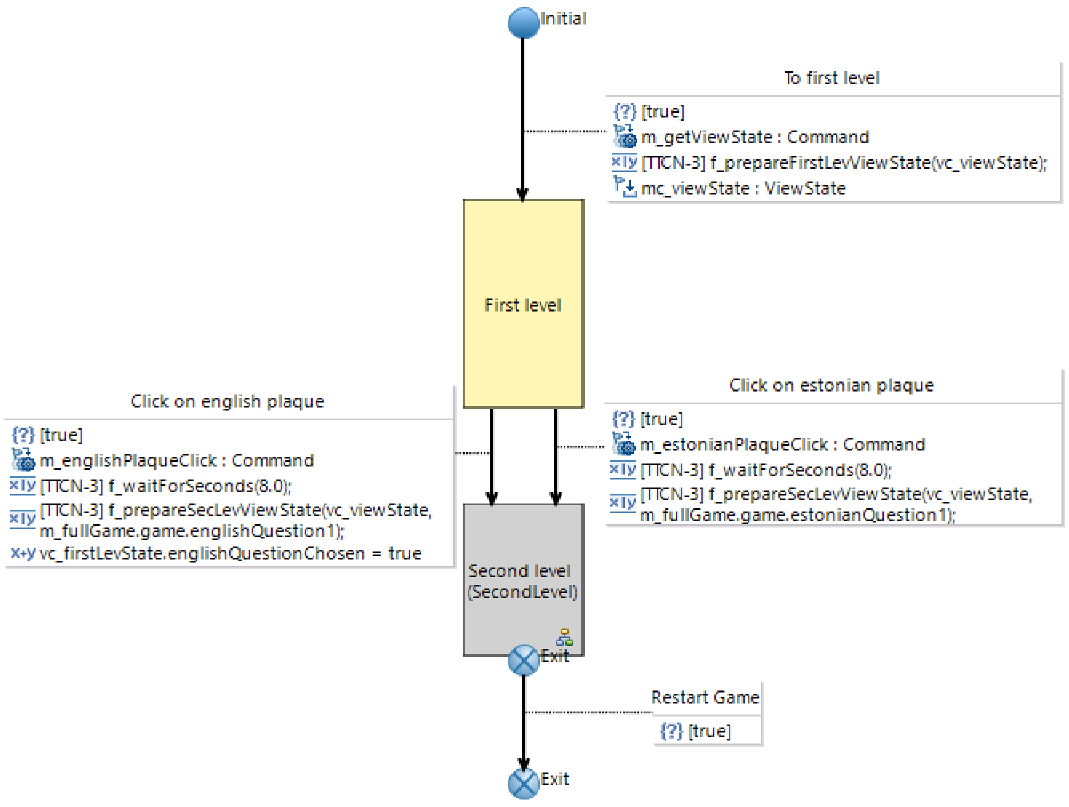
Figure 4.3: Model of initializing the game.

has six degrees of freedom to move in the game. Thus the states could have been turning head up, turning head down, turning right, turning left etc. In the paper [7] a platform game was tested using MBT and there the states were created based on what the avatar is doing. In a platform game there are only two DOF and therefore it is much more easier to model it this way. Furthermore there are no easily recognizable position states for the avatar like there are in case of the VR Quiz game.

4.4 Unity test objects

To test the game a new superclass to the TestObject class had to created in Unity, because the methods for triggering the object and checking whether the object is visible had to be overwritten. This was also described in section Unity testing manager.

Checking whether an object is visible in the game is done by making the player view turn towards the object and then checking whether there are any obstacles between the player and the object and checking whether the object is rendered. Triggering an object is done by turning the player view towards the object and then triggering a mouse click event. An


Initial
To first level
{?} [true]
m_getViewState : Command
[TTCN-3] f_prepareFirstLevViewState(vc_viewState);
mc_viewState : ViewState
First level
Click on english plaque
{?} [true]
m_englishPlaqueClick : Command
[TTCN-3] f_waitForSeconds(8.0);
[TTCN-3] f_prepareSecLevViewState(vc_viewState, m_fullGame.game.englishQuestion1);
vc_firstLevState.englishQuestionChosen = true
Click on estonian plaque
{?} [true]
m_estonianPlaqueClick : Command
[TTCN-3] f_waitForSeconds(8.0);
[TTCN-3] f_prepareSecLevViewState(vc_viewState, m_fullGame.game.estonianQuestion1);
Second level (SecondLevel)
Exit
Restart Game
{?} [true]
Exit
Figure 4.4: Model of the first level.

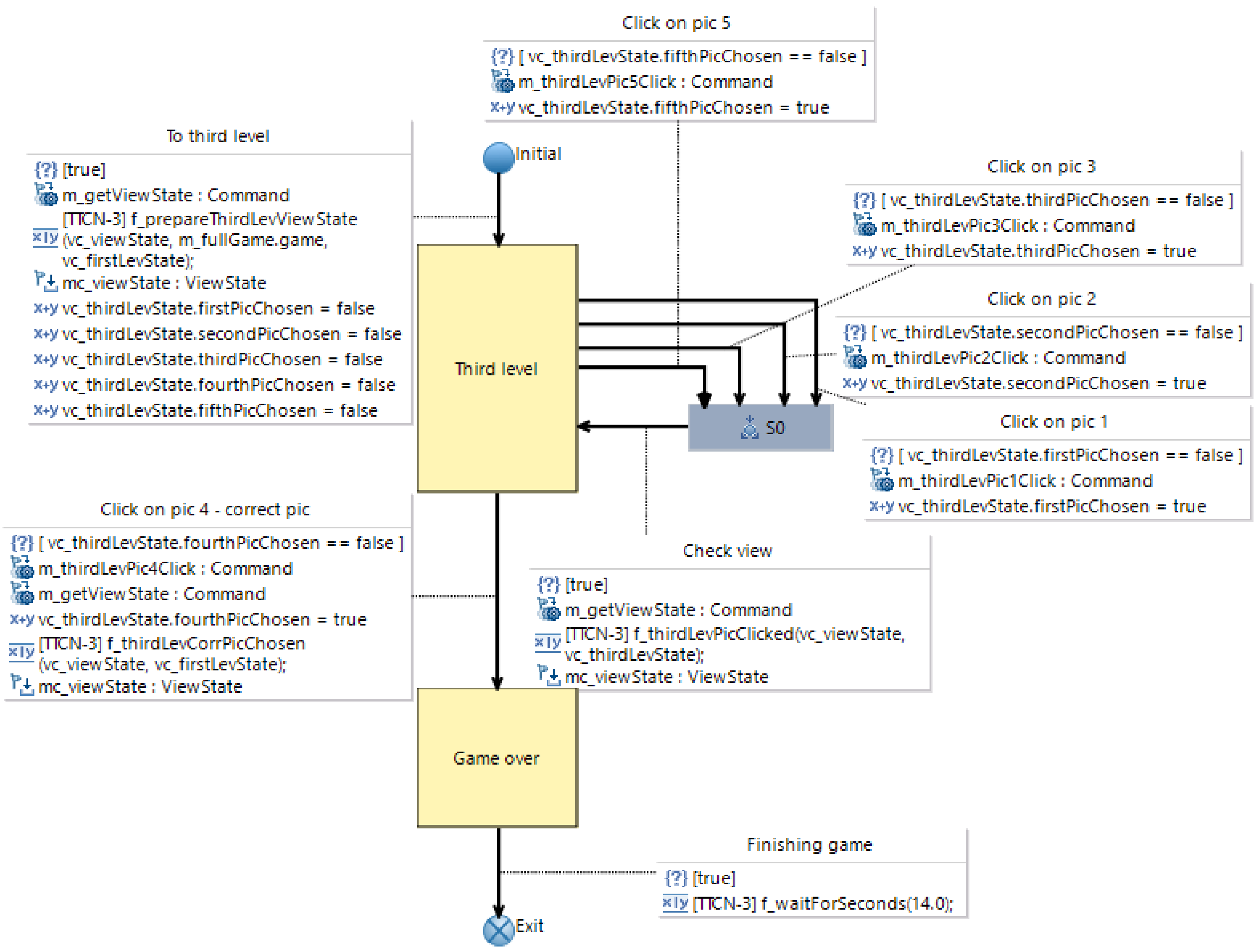

Click on pic 5
{?} [vc_thirdLevState.fifthPicChosen == false]
m_thirdLevPic5Click : Command
x+y vc_thirdLevState.fifthPicChosen = true
To third level
{?} [true]
m_getViewState : Command
[TTCN-3] f_prepareThirdLevViewState (vc_viewState, m_fullGame.game, vc_firstLevState);
mc_viewState : ViewState
x+y vc_thirdLevState.firstPicChosen = false
x+y vc_thirdLevState.secondPicChosen = false
x+y vc_thirdLevState.thirdPicChosen = false
x+y vc_thirdLevState.fourthPicChosen = false
x+y vc_thirdLevState.fifthPicChosen = false
Initial
Third level
S0
Click on pic 3
{?} [vc_thirdLevState.thirdPicChosen == false]
m_thirdLevPic3Click : Command
x+y vc_thirdLevState.thirdPicChosen = true
Click on pic 2
{?} [vc_thirdLevState.secondPicChosen == false]
m_thirdLevPic2Click : Command
x+y vc_thirdLevState.secondPicChosen = true
Click on pic 1
{?} [vc_thirdLevState.firstPicChosen == false]
m_thirdLevPic1Click : Command
x+y vc_thirdLevState.firstPicChosen = true
Click on pic 4 - correct pic
{?} [vc_thirdLevState.fourthPicChosen == false]
m_thirdLevPic4Click : Command
m_getViewState : Command
x+y vc_thirdLevState.fourthPicChosen = true
[TTCN-3] f_thirdLevCorrPicChosen (vc_viewState, vc_firstLevState);
mc_viewState : ViewState
Check view
{?} [true]
m_getViewState : Command
[TTCN-3] f_thirdLevPicClicked(vc_viewState, vc_thirdLevState);
mc_viewState : ViewState
Game over
Finishing game
{?} [true]
[TTCN-3] f_waitForSeconds(14.0);
Exit

Figure 4.5: Model of the third level.

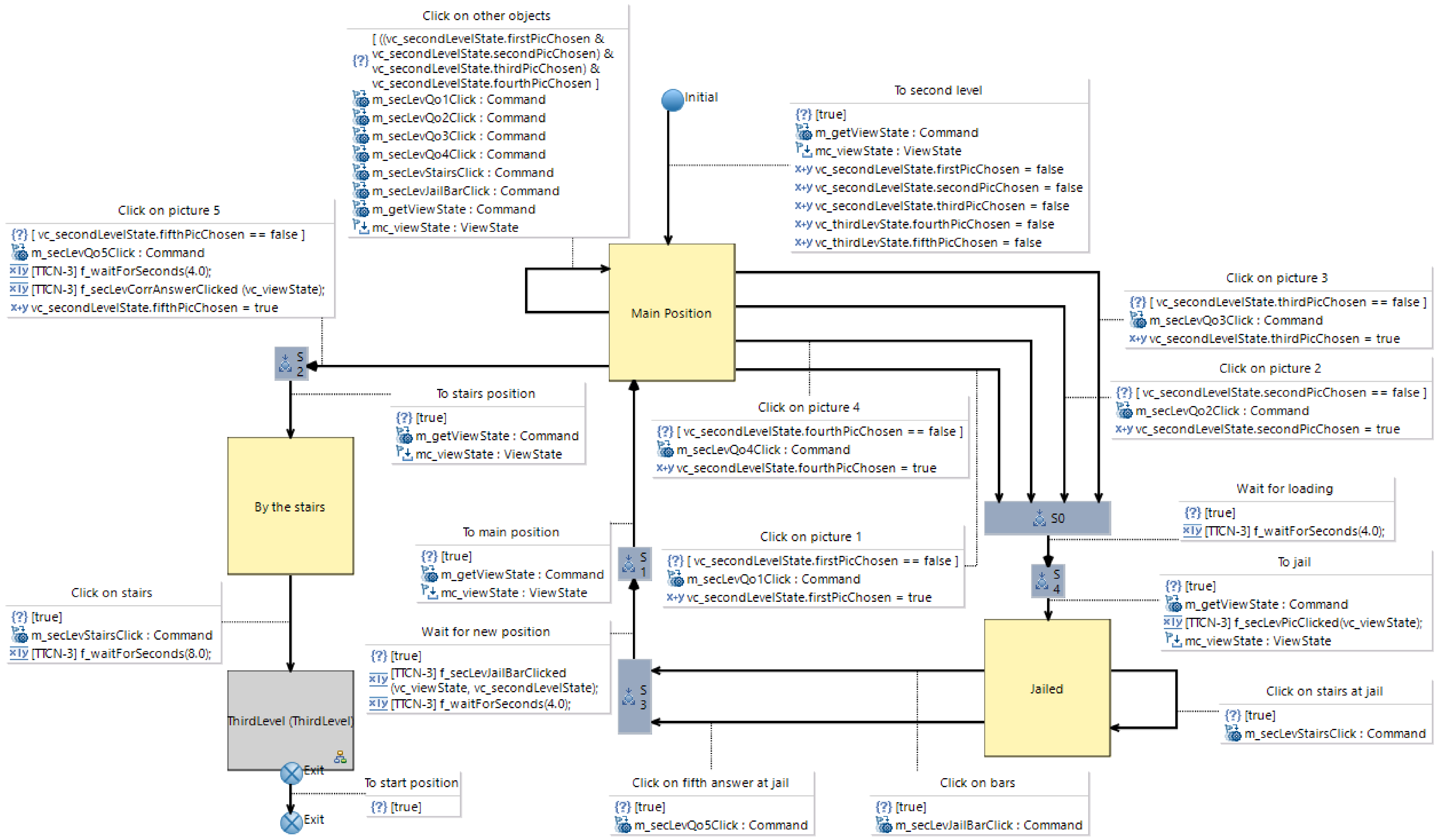

Click on other objects
[((vc_secondLevelState.firstPicChosen & vc_secondLevelState.secondPicChosen) & vc_secondLevelState.thirdPicChosen) & vc_secondLevelState.fourthPicChosen]
m_secLevQo1Click : Command
m_secLevQo2Click : Command
m_secLevQo3Click : Command
m_secLevQo4Click : Command
m_secLevStairsClick : Command
m_secLevJailBarClick : Command
m_getViewState : Command
mc_viewState : ViewState
Initial
To second level
[true]
m_getViewState : Command
mc_viewState : ViewState
vc_secondLevelState.firstPicChosen = false
vc_secondLevelState.secondPicChosen = false
vc_secondLevelState.thirdPicChosen = false
vc_thirdLevState.fourthPicChosen = false
vc_thirdLevState.fifthPicChosen = false
Click on picture 5
[vc_secondLevelState.fifthPicChosen == false]
m_secLevQo5Click : Command
[TTCN-3] f_waitForSeconds(4.0);
[TTCN-3] f_secLevCorrAnswerClicked (vc_viewState);
vc_secondLevelState.fifthPicChosen = true
Main Position
Click on picture 3
[vc_secondLevelState.thirdPicChosen == false]
m_secLevQo3Click : Command
vc_secondLevelState.thirdPicChosen = true
S2
To stairs position
[true]
m_getViewState : Command
mc_viewState : ViewState
Click on picture 2
[vc_secondLevelState.secondPicChosen == false]
m_secLevQo2Click : Command
vc_secondLevelState.secondPicChosen = true
Click on picture 4
[vc_secondLevelState.fourthPicChosen == false]
m_secLevQo4Click : Command
vc_secondLevelState.fourthPicChosen = true
By the stairs
Wait for loading
[true]
[TTCN-3] f_waitForSeconds(4.0);
S0
To main position
[true]
m_getViewState : Command
mc_viewState : ViewState
Click on picture 1
[vc_secondLevelState.firstPicChosen == false]
m_secLevQo1Click : Command
vc_secondLevelState.firstPicChosen = true
S1
S4
To jail
[true]
m_getViewState : Command
[TTCN-3] f_secLevPicClicked(vc_viewState);
mc_viewState : ViewState
Click on stairs
[true]
m_secLevStairsClick : Command
[TTCN-3] f_waitForSeconds(8.0);
Wait for new position
[true]
[TTCN-3] f_secLevJailBarClicked (vc_viewState, vc_secondLevelState);
[TTCN-3] f_waitForSeconds(4.0);
S3
Jailed
ThirdLevel (ThirdLevel)
Click on stairs at jail
[true]
m_secLevStairsClick : Command
Exit
To start position
[true]
Exit
Click on fifth answer at jail
[true]
m_secLevQo5Click : Command
Click on bars
[true]
m_secLevJailBarClick : Command

Figure 4.6: Model of the second level.

additional script was created for the avatar object that allows the other test scripts to turn towards test objects.

For the menu three additional scripts were created, that are superclass to the TestObject class. These are:

- TestLabel, which is used for menu object, which have a text. The script also allows changing of the text by the test manager.
- TestButton, which is used for buttons in the menu and it allows to trigger the button, to which the script is attached to.
- TestIndicator, which is used by the on or off indicators in the menu. The script is needed to check whether the indicator is in an on or off state.

4.5 Generated tests and test results

TestCast MBT generates the test cases based on the created model. It is possible to generate them in a number of different ways. It can be done by going through all the transitions, going through either one or several transitions in a specific order or in any order or by reaching a specified state.

For testing the game and menu it was possible to generate one test for each of them, which would go though all the transitions. For the menu TestCast MBT generated a test script with 458 actions to process. For the game 118 actions were generated. The two test cases cover 64 % of the functional requirements. The percentage is calculated based on the number of tasks in the sprint backlogs, which is used in development of the VR Quiz as stated in section Development of VR Quiz and the test suite. No additional tests were generated since it would not have covered additional items in the specification.

Up to now four bugs have been found while running the tests. These were:

1. The menu freezes when trying to connect to a controller and clicking cancel.
2. When clicking on disable button, the edited game name does not reset to correct value.

3. When clicking disable button, the new game being created is also reset. Only a chosen game from the list should be reset.
4. The name length of the game is limited.

Not all of the created functionality is modelled yet, but the rest of will be done in later iterations, since the testing method has found bugs, which were previously not found during development.

5. Evaluation of the proposed solution

The proposed solution works great when testing menus and very simple games. It could be that it is not possible to use the testing approach on much more complex systems. There seems to be no one way for modelling a game thus the most difficult part is to figure out the best way to model it. At least the solution uses TTCN-3, which has proven to be able to test very complex systems.

The proposed solution uses a commercial tool, which would add the cost of the tool to testing, but the benefits from using it outweighs the cost. Other commercial should also be evaluated fully, but it was not done, since the main goal was on getting a working system ready and evaluating all the tools would have taken too much time. Furthermore the author works in the company, which is creating the tool used, TestCast MBT, and the knowledge gained from testing games will be used for developing the modelling tool further so that it could better help Unity developers to test their applications.

The created framework provides enough abstraction, that it is possible to use it with different test executors. As part of this thesis VR Quiz was tested with NModel and with TestCast with no changes to the framework.

Testing of the VR Quiz was done by the same person who developed the system. This is a very bad practice as the developers tend to test only the scenarios they know works and not any new ones. Unfortunately this was unavoidable since only the developer had the necessary skills and the knowledge of testing in the team.

5.1 Answers to research questions

One of the goals of this thesis was to find answers to four questions stated in the beginning of the thesis in section Goal of the thesis.

1. Is it feasible to use model-based testing on 3D games, where the user has six DOF to look around?

   It was feasible in the case of VR Quiz, which has a very simple interaction system (only look at object and click). Furthermore in VR Quiz there were only a small amount of areas where the user's avatar can be and he is not able to move the avatar around by himself. The game moves the avatar after the user interacts with some objects, which meant that it was very easy to create a state machine, which has a small number of areas as the states. This method may not work with games, where the user can move his avatar to almost infinite number of positions.

2. How to model and generate tests of a 2D menu so that all the transitions of the menu are covered without state explosion?

   It was possible to model a menu without the state explosion by using the test data as part of the model. The model was created by following a principle, that every transaction has a triggering command, for example pressing a button, followed by changing the test data according to what the command should do and then checking the test data to the actual data to verify, that the SUT worked properly to the command.

3. Does the model-based testing approach help to discover previously unknown errors in the application that has been manually tested?

   Four previously unknown errors were discovered with the automated tests. Two of them were because of the test generator generated a non regular scenario. They were the second and third in the list of bugs in section Generated tests and test results. Those bugs would not have been found by the tester, when he had chosen using test cases and test ideas for testing method and manual testing.

4. Is it possible and to what extent can the models created in testing of VR Quiz be used for other applications?

   The created models are too domain specific to be used for other applications.

6. Conclusion

The main goal of the thesis was to propose an automated system testing approach for applications made with Unity. Additionally to find out if it is feasible to use model-based testing on 3D games and to test a program named VR Quiz.

To understand the need of game developers further interviews were conducted and research was done on currently used methods for automated system testing. Based on the interviews the main issue with automated system testing is that they are either too difficult to learn by the testers or they lack the required functionality to test more difficult systems. Furthermore the test suite should be easily maintainable so that it would be possible to use it during the development phase of a game.

In this thesis a model-based approach for automated system testing is proposed using TestCast MBT and a Unity testing framework, which was created as part of this thesis. The framework allows controlling a program made with Unity remotely and get information about the state of the program. The testing framework can be used with different tools and is open source.

As a proof of concept a program made with Unity named VR Quiz was tested. The test suite proved to be useful as it discovered four errors in the system. Furthermore the test suite will be used for regression testing throughout the development of the VR Quiz.

Therefore the goals of the thesis were met because the proposed solution proved to be feasible since it was possible to test VR Quiz with it and find errors in the system. The solution is rather easy to use without requiring much programming skills.

References

[1] Madis Taimre. Unity testing framework repository. URL `https://gitlab.com/madis-taimre/UnityTestingFramework.git`. [Online] (07.01.2017).

[2] Fazeel Gareeboo and Christian Buhl. Automated testing: A key factor for success in video game development. case study and lessons learned. 2012. [Online] Pacific NW Software Quality Conference (02.10.2016).

[3] Unity public relations, 2016. URL `https://unity3d.com/public-relations`. [WWW] (11.20.2016).

[4] Unity subscriptions, 2016. URL `https://store.unity.com/products/unity-personal?_ga=1.257825931.1632734414.1466867012`. [WWW] (11.20.2016).

[5] EA Orlando, 2016. URL `http://www2.ea.com/locations/orlando`. [WWW] (02.10.2016).

[6] Omar el Ariss, Dianxiang Xu, Santosh Dandey, Bradley Vender, Philip E. McClean, and Brian M. Slator. A systematic capture and replay strategy for testing complex GUI based java applications. In *Seventh International Conference on Information Technology: New Generations, ITNG 2010, Las Vegas, Nevada, USA, 12-14 April 2010*, pages 1038–1043, 2010. doi: 10.1109/ITNG.2010.216. URL `http://dx.doi.org/10.1109/ITNG.2010.216`. [Online] (15.10.2016).

[7] Sidra Iftikhar, Muhammad Zohaib Iqbal, Muhammad Uzair Khan, and Wardah Mahmood. An automated model based testing approach for platform games. 2015. *Model Driven Engineering Languages and Systems (MODELS), 2015 ACM/IEEE 18th International Conference.* [Online] IEEE Xplore (11.10.2016).

[8] Guilherme de Cleva Farto. Evaluating the model-based testing approach in the context of mobile applications. 2015. *CLEI 2014, the XL Latin American Conference in Informatic* [Online] ScienceDirect (14.10.2016).

[9] Playmaker, 2016. URL `https://www.assetstore.unity3d.com/en/#!/content/368`. [WWW] (14.10.2016).

[10] Zoltán Micskei. Model-based testing. URL `http://mit.bme.hu/~micskeiz/pages/modelbased_testing.html`. [Online] (07.01.2017).

[11] Graphwalker features, 2016. URL `http://graphwalker.github.io/features/`. [WWW] (10.10.2016).

[12] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson. *Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer*. 2008. *Formal Methods and Testing* [Online] Springer Verlag (14.10.2016).

[13] Eclipse Mars, 2017. URL `https://eclipse.org/mars/`. [WWW] (06.01.2017).

[14] Elvior OÜ, 2016. URL `http://www.elvior.com/`. [WWW] (20.10.2016).

[15] Georg Panholzer, Christof Brandauer, Stephan Pietsch, and Jurgen Resch. On Investigating the Benefits of TTCN-3-Based Testing in the Context of IEC 61850. 2015. *ENERGY 2015 : The Fifth International Conference on Smart Grids, Green Communications and IT Energy-aware Technologies* Think Mind [Online] 10.10.2016.

[16] Introduction to TTCN-3, 2013. URL `http://www.ttcn-3.org/index.php/about/introduction`. [WWW] (10.10.2016).