Automated Regression Testing within Video Game Development

Michail Ostrowski
Related Designs Software GmbH
Mainz, Germany
ostrowski@related-designs.de

Samir Aroudj
TU Darmstadt
Darmstadt, Germany
samir.aroudj@stud.tu-darmstadt.de

***Abstract*—Automated testing reduces the costs of software development. We propose a testing model that is specifically designed for the creation and execution of fully automated regression tests within video game development without the need of test isolation. The proposed model combines the usability and veracity of record and playback techniques and the possible test coverage of tests written in a game specific scripting language. The model has an intuitive structure that enables professional video game testers to create meaningful tests by using only a rudimentary set of programming skills within a graphical user interface. The resulting tool establishes a network connection with one or several concurrently running applications under test while the required performance can be distributed between several dedicated computers. The application under test is modified to process incoming function calls and to generate output that is used to dynamically control the course of the tests and to evaluate if the tested application operates within acceptable parameters.**

Keywords-software testing; automated regression testing; video game development; video game testing

I. INTRODUCTION

The development of video games is a complex process which is prone to software regression due to frequently changed software design. The risk of software regression is usually compensated by a quality assurance team. Since the manual execution of regression tests is expensive in terms of time and money, automated testing techniques are capable of enhancing the testing process. However, a poll of 120 professional game developers at the *Game Developer Conference* in 2002 documented that only 18% use automated testing techniques within their current projects [1]. Due to the lack of suitability of chosen tools, the implementation of an automated test process fails frequently [2].

This paper describes script-based and user-interface-based testing in the background section and discusses the applicability of those techniques within video game development. In section 3 we introduce the structure and test process of our hybrid approach. The discussion summarizes the most relevant possibilities of the proposed architecture.

DOI: 10.5176/2251-3043_3.2.257

II. BACKGROUND

Modern games utilize multiple programming languages. The core component of the game is often realized with high-performance languages such as C++ while high-level components that are prone to frequent modifications are realized with scripting languages like Lua. Plumlee describes an approach on automated regression testing which utilizes the integration of a scripting language [3]. The test scripts are written in the scripting language of the application under test (AUT). Test scripts modify the state of the AUT and verify the result with assertions. Assertions compare the result of a method call with an expected value. The creation of test scripts requires knowledge of the used scripting language. Professional video game testers have domain specific knowledge that is required to create meaningful tests. However, since most video game testers do not have an engineering background, creation of tests based on a scripting language is limited to programmers. This method is therefore difficult to integrate in development processes that mainly rely on video game testers. Furthermore, the proposed test scripts interact with the AUT through function calls and not with user input. The validity of the generated test results is therefore limited.

The record and playback technique [4] is used to test applications with the help of recorded user interactions. An application tester records a test by generating a list of events which detail the position and time of mouse clicks or typed key sequences. The execution of the created list recreates the previously recorded sequence of user interactions. A changed position of a user interface element (UIE) in the AUT renders the playback of corresponding test scripts invalid since recorded mouse clicks still use outdated position data. The maintenance effort of such recordings is high and prone to playback errors. **Functional decomposition** divides long recordings into independent parts. Those parts are utilized in multiple tests. The separation of data and recording within **data-driven testing** enables a recording to be used with different input values. Both techniques reduce the maintenance effort of the record playback technique significantly [5]. There are two approaches to automatically evaluate the test result and further decrease the maintenance effort.

Image based methods generate screenshots that represent the current state of the AUT. Those screenshots are compared to picture templates that represent UIEs. The position of a UIE is dynamically determined by moving the corresponding template over every possible position of the generated screenshot. The template position which offers a perfect match corresponds to the position of the UIE. This method is robust against changes in position of UIEs, but needs maintenance if the appearance of the UIE changes. The application state is evaluated by comparing a generated screenshot with a predefined picture template that represents the expected application state. The test fails if the generated screenshot does not match the template. Since video games tend to use animations and transparency effects, the resulting screenshot varies in appearance and a perfect match, even for a single UIE, is therefore very unlikely. The probability of a match decreases with the magnitude of the variation of a generated screenshot that represents a certain application state. For this reason, it is very unlikely to recognize the application state of a video game that composes a scene with multiple instances of rotated, animated and projected geometry correctly. Modifications of the application programming interface (API) which is used for rendering the scene, enable the tester to mark interactive objects with distinctive and invariant markers in order to enhance image based processes. This technique is widely known in video game based competitions where so called "Aimbots" mark enemies with a distinctive color in order to automatically recognize them by means of image based methods. The recognized enemy position is used to automatically move the mouse cursor over the enemy. However, this technique is not further reviewed for automated regression testing by this paper.

The 2nd approach to enhance the simple record and playback technique is based on the usage of standard technology. Tools like *Ranorex*[1] detail the structure of applications written in Java, .NET or Flash during runtime. User interactions are associated with the application component they are involved with. Those tools are capable of dynamically determining the positions of a UIE by detailing the structure of the user interface. This technique is therefore robust against changes in appearance and position of UIEs. The application status is determined by the status of the application components like the current string of a text label or the selection status of a radio button. Since video games do not exclusively utilize standard technology, the test coverage is limited. Record and playback techniques are easy to use and only demand very fundamental programming skills. Thus the domain specific knowledge of professional video game testers can be easily utilized for creating meaningful automated tests [6].

[1]More information about *Ranorex* is available at http://www.ranorex.com/ [Last Visited: 9/20/2012].

III. Results

A. Definitions

The proposed test model defines several structures. An **event** is a structure that contains a string, a multiplier and client identification number (**CIN**). A **condition** is a structure that holds a string, a mode, a threshold, a counter and a CIN. A condition registers an event by comparing its stored string and CIN with the string and CIN of the event. If both strings as well as both CINs match, the counter of the condition is increased by the value of the multiplier that is stored within the event. A condition has two modes which determine if the threshold is a maximum or a minimum value. A condition is fulfilled if the counter is according to the mode less than or equal to the threshold or greater than or equal to the threshold. A **command** is a structure that contains a string, a delay value, and a CIN. The string of a command describes a method call that is known by the recipient and processed on arrival. The delay value describes the estimated execution time of the method call. A command is executed if the command is dispatched and the time stamp defined by the sum of the delay value and the dispatching time has passed. A **control unit** is a structure that holds a list of commands, a list of conditions and three Boolean values that specify if the control unit is optional, if it is repetitive and if it has a high priority. A control unit registers an event by passing it to every condition it holds. A control unit is ready when its conditions are fulfilled and none of its commands are executed. A control unit is processed if its conditions are fulfilled and every command it holds is executed. An **application definition** contains a file path to the AUT and a list of parameters. A **test case** is a structure that contains a list of control units, a list of application definitions, a queue of commands, a time limit and a time stamp. A test case registers an event by passing it to every control unit it holds. A test case passes if its control units are optional or processed. The rank of a control unit is determined by its position in the list of control units. The rank of control unit A is higher than the rank of control unit B if control unit A has a lower list index. A **test suite** contains a list of test cases. A test suite passes if every test it holds passes. Figure 1 shows the hierarchical structure of a test suite.

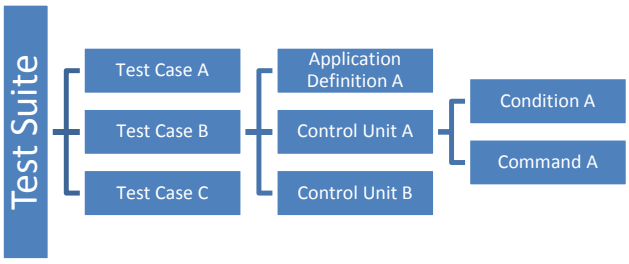

Figure 1. Hierarchical structure of a test suite

B. Test Process

The execution of a test case starts with the creation of a **test server**. The test server accepts incoming TCP/IP connections from AUTs. This test model proposes the extension of the

AUT by a **test mode**. An application that is run in test mode establishes an outgoing TCP/IP connection to the local test server and creates messages that document the status of the application. The proposed test process iterates sequentially over every test case of a given test suite. The test server generates a CIN for every accepted AUT. The CIN is equal to the number of previously accepted connections within the current test case. The server wraps all received messages into events and stores them in its **event queue**. Events that contain a special keyword (such as "CONTROL::") are not associated with the CIN of the client but are rather set to a CIN of -1. The test process stores a queue of commands that is called **execution queue** and an execution time stamp. A command becomes executable if it is the first element of the execution queue and the execution time stamp has passed.

The execution of a test case includes the repeated registration of events and the determination of executable commands. Control units become ready by registering events. If the execution queue is empty, the control unit that is ready and that has the highest rank is selected by the test case. The commands of a selected control unit are added to the end of the execution queue and the time stamp of the test case is set to the current time. When removing an executable command from the execution queue, the time stamp is set to the sum of the current time stamp and the delay value of the removed command. If the delay value is zero, the next command in the queue becomes executable immediately. A query that returns a list of executable commands removes every command that is or becomes executable from the execution queue. Commands of control units that were assigned priority are added to the front of the queue, regardless of the content of the queue. The test process stores those commands within a **command queue** and passes them to the server. The server sends them to a client that has a CIN that matches the CIN of the command. The execution of a test case ends when the test case passes or the time limit defined in the test case is exceeded. The test server and any applications started within the test case are then stopped.

The test process filters commands from the command queue and events from the event queue with a CIN of -1 to execute them within a local function call. There are four commands that are typically executed by the test process. The test process executes a **run command** (such as "CONTROL::RUN-0") by starting an application with the given index which specifies an application definition stored in the test case. A **click command** (such as "CONTROL::CLICK-100-200-1") is executed by imitating a mouse click that uses the given parameters as x and y position on the computer screen. The last parameter specifies the mouse button to be pressed. A **key command** (such as "CONTROL::KEY-myUserName") specifies a sequence of key strokes that is imitated by the test process. A **reset command** is executed by iterating over every control unit of the current test case. The conditions and commands of control units that are marked as repetitive are set to initial values.

Most video games utilize a scripting language or at least a game specific console. The functionality of those systems is easily accessible by redirecting incoming commands within the AUT (such as "CONSOLE-MyConsoleCommand" or "SCRIPT-MyScriptCommand"). A broad functionality with low implementation costs is hence available. We recommend to use and extend existing systems in order to reduce code scattering. A system to load and save the state of the AUT keeps test cases short, even for complex aspects.

C. User Interface Maps

A user interface map (UIM) stores UIEs of an AUT that can be accessed by unique reference names [6]. The proposed test model sends **request commands** containing a keyword (such as "CLICK-ID_BUTTON01") to the AUT. Those commands are processed by retrieving the UIE with the corresponding key from the UIM. An event is created that contains a keyword that marks the event for command processing (such as "CONTROL::CLICK-100-200-1"). A click command that is generated by a UIM is robust against changes in position or visual representation of UIEs. The interaction with UIEs that are represented by visible geometry within the scene graph requires the projection of the center point from 3D scene graph coordinates into 2D display coordinates.

The interaction of a tester with the AUT is easily recordable with the implementation of a recording mode. In recording mode, the AUT generates a list of unique names which correspond to the UIE that were used. This list is directly convertible to a list of commands that can be inserted into a control unit.

D. Repository

A repository is a map of control units, commands and conditions each of which is accessed by a unique reference name. A test case is stored as a list of references to the repository. They are resolved when loading a test suite to the test process. Changed components of the AUT require the maintenance of corresponding conditions, commands and control units. Since those structures are stored as references, multiple tests are updated by simply modifying single elements in the repository. A test case can utilize different repositories in order to use a single test script with multiple value sets. The content of the UIM is suitable for generating a basic set of request commands since it contains all keywords that are required for the interaction with the interface elements of the AUT. We therefore propose to extend the AUT by an **index function** that sends the keywords of the UIM to the test server where they are converted to request commands and stored in the repository. A list of possible events within the AUT is usable to extend that index function in order to generate a basic set of conditions. Thus an index function enables the AUT to easily update the repository when new events and interface elements are defined during development.

E. Test Design

A simple test case consists out of three control units and an application definition. The first control unit has no conditions and is therefore ready and selectable by the test process. The control unit holds a run command which starts the AUT in accordance to the application definition of the test case. The second control unit has conditions that are fulfilled by events the application generates on startup and commands that interact with the application by imitated user interactions or direct function calls which are processed by the application. These interactions initiate processes within the AUT that generate events. The test fails if the events do not fulfill the conditions of the third control unit within the time limit. Figure 2 shows the hierarchical structure of a simple test case, while the equivalent process is shown as an activity diagram represented in the Unified Modeling Language in Figure 3.

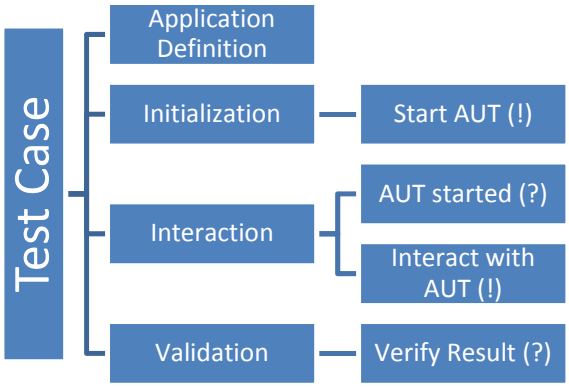

Figure 2. Hierarchical structure of a simple test case - Conditions are marked with a question mark and commands are marked with an exclamation mark.

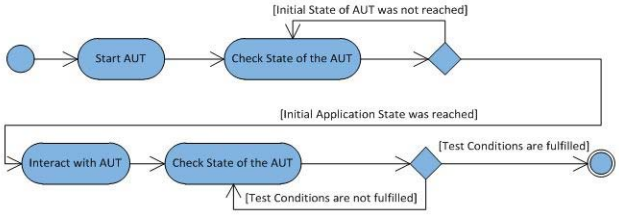

Figure 3. Process visualization of a simple test case

Optional control units do not need to be processed in order for the corresponding test case to pass. Therefore a test script is able to adjust the course of a test to the current parameterization of the AUT. Since the repeated recreation of exact test preconditions is difficult for complex systems, deviations in the behavior of the AUT occur. Those deviations are not necessarily errors but a result of pop-ups, random events or connection issues with a remote server. Randomly occurring deviations such as pop-ups render some user interface interaction invalid. Control units with a high priority interrupt the command execution of the current test case by inserting their commands at the beginning of the execution queue. Randomly occurring deviations are therefore suppressible by high priority control units that remove pop-ups or reconnect to remote servers at any point in the test. Repetitive control units define a set of commands that can be executed multiple times. This means that the executed actions have to recreate an application state that makes the set of commands applicable again. The status of a repetitive control unit is set to initial values by a reset command that is triggered after the last command of the control unit is executed. Thus the conditions of the repetitive control unit can be fulfilled again.

F. Handling Multiple Applications

The multiplayer aspect of video games describes the simultaneous interaction of several game instances using a shared subset of application variables. This shared subset of variables is called a **session**. The proposed test model is able to handle multiple AUTs within a test case by defining sets of control units that contain conditions and commands with different CINs. The number of concurrently running AUTs on a single machine is limited by the required performance of each AUT and the available hardware. The proposed test model facilitates coverage of the multiplayer aspect of video games. A test case that creates a session by user interface interactions with all involved AUT needs to ensure the validity of simulated user input by a preceding focus and push of the corresponding application window to the front of the display. Applications with high performance requirements (**AHPR**) limit the number of concurrent application executions on a single computer to one. In order to test the multiplayer aspect of an AHPR, the test process is distributed between dedicated computers that are connected within a network. A **test agent** is an application that runs on a dedicated computer and wraps the communication between the test server and the AUT. The test agent creates a **remote application server** and establishes an outgoing connection to the test server. A test agent receives commands from the test server and forwards them to the AUT. Events from the AUT are forwarded to the test server. The test agent detects run, click or key commands and executes them. When executing a run command, the application under test connects to the remote application server. The proposed test model is therefore able to control several AHPRs. The number of concurrently running AUT within a test case is only limited by the number of available machines and application specific boundaries.

IV. Discussion

The proposed test model was realized during the development of *Anno 2070 – Deep Ocean*[2]. The test process was organized by an in-house quality assurance team which completely relied on manual testing. Regression testing was realized by a weekly test plan which was used as a blue print for the created test suite. The realized test tool was installed on a dedicated

[2]*Anno 2070 – Deep Ocean* is a video game that was developed by *Related Designs Software GmbH* and *Ubisoft Blue Byte*. It was published on 10/04/2012 by *Ubisoft Entertainment*. Additional information about the game are available at http://anno-game.ubi.com/anno-2070/en-GB/home/[Last visited: 10/19/2012].

test machine. A new version of the AUT was frequently built on a remote machine. Every build process triggered the execution of the created test suite. The test results were mailed to corresponding employees afterwards. The transition from manual to automated regression testing was backed by a professional game tester. The test model is now utilized in the production of *Might and Magic Heroes Online* that is currently under development by *Related Designs Software GmbH* and *Ubisoft Blue Byte*. The proposed test model allows professional video game testers without previous programming experience to easily create and execute powerful and maintainable automated tests within a graphical user interface. The corresponding test suite contains several simple test cases that verify the functionality of multiple menus with buttons and text fields by executing a list of recorded clicks.

Furthermore, there are test cases that detect memory leaks within the AUT by defining a set of repeatable actions within a repetitive control unit. The test process executes the repetitive control unit several times within one test case. The memory consumption of the AUT is logged each iteration. Such a control unit defines a set of actions that could prove the existence of a memory leak like a repeated change of position within the game world since this forces the application to constantly provide new assets like textures and geometry. Furthermore, the required execution time of each test case is stored within the test documentation. This enables the quality assurance team to easily evaluate the loading times of new software iterations.

Might and Magic Heroes Online utilizes an artificial intelligence (AI) that plays against a human contestant within turn-based battles. There are several tests that replace the human player with another instance of the AI and therefore realize fully automated battles. Since those tests involve a high number of visual effects and complex AI algorithms new errors are frequently found.

Some UIEs in *Might and Magic Heroes Online* are represented by geometry within a scene graph. Those UIEs can be visually obstructed by other players. The game provides a selection menu to interact with the obstructed UIE. A optional high priority control unit handles possible obstructions using that selection menu. The test process reliably operates without the requirement of complete test isolation. This means that there is no special environment needed to execute automated testing which decreases the required implementation effort and increases the relevance of the test results. The usage of a repository keeps the required maintenance effort of a test suite low. UIMs realize the playback of scripted user interactions that are robust against changes of the user interface layout and the visual representations of UIEs. The proposed model is able to handle multiple applications under test within one test case and is therefore suitable for verifying client-server based video game concepts.

REFERENCES

[1] Noel Llopis and Brian Sharp. "Convexhull." Online: http://www.convexhull.com/sweng/GDC2002.html, 3/31/2002 [Last visited: 7/18/2012]

[2] Konrad Schlude, "Risk investment Test Automation" *Testing Experience*, no. 17, p. 47, December 2008.

[3] Phlip C. Plumlee. "Integration Test Servers for Videogames." Online: http://flea.sourceforge.net/gameTestServer.pdf, 3/16/2005 [Last visited: 9/20/2012]

[4] Koen Wellens, "The Record & Playback Fairy Tale" *Testing Experience*, no. 17, pp. 22-23, December 2008.

[5] Keith Zambelich. "Totally Data-Driven Automated Testing." Online: http://www.oio.de/public/softwaretest/Totally-Data-Driven-Automated-Testing.pdf, 6/20/2002 [Last visited: 9/20/2012]

[6] Mark Michaelis, "Boon and Bane of GUI Test Automation" *Testing Experience*, no. 17, p. 25-28, December 2008.

Michail Ostrowski has received his master's degree in computer science from Otto von Guericke University Magdeburg in 2012. He is currently working as a game developer for Related Designs Software GmbH in Mainz, Germany.

Samir Aroudj studies Visual Computing at the technical university of Darmstadt. He completed the bachelor degree course Digital Media and Games in Trier. His bachelor thesis was written in conjunction with the game developer studio Related Designs for which he currently works part-time.