Automatic Gameplay Testing for Message Passing Architectures

Jennifer Hernández Bécares, Luis Costero Valero
and Pedro Pablo Gómez Martín

Facultad de Informática, Universidad Complutense de Madrid.
28040 Madrid, Spain
{jennhern,lcostero}@ucm.es
pedrop@fdi.ucm.es

Abstract. Videogames are highly technical software artifacts composed of a big amount of modules with complex relationships. Being interactive software, videogames are hard to test and QA becomes a nightmare. Even worst, correctness not only depends on *software* because *levels* must also fulfill the main goal: provide entertainment. This paper presents a way for automatic gameplay testing, and provides some insights into source code changes requirements and benefits obtained.

Keywords: gameplay testing, testing, automatisation

1 Introduction

Since their first appearance in the 1970s, videogames complexity has been continuously increasing. They are bigger and bigger, with more and more levels, and they tend to be non-deterministic, like Massively Multiplayer Online games where emergent situations arise due to players' interactions.

As any other software, videogames must be tested before their release date, in order to detect and prevent errors. Unfortunately, videogames suffer specific peculiarities that make classic testing tools hardly useful. For example, the final result depends on variable (nearly erratic) factors such as graphics hardware performance, timing or core/CPU availability. Even worst, correctness measure is complex because it should take into account graphic and sound quality, or AI reactivity and accuracy, features that cannot be easily compared.

On top of that, videogames are not just *software*. For example, it is not enough to test that physics is still working after a code change, but also that players can end the game even if a level designer has moved a *power up*. Videogames quality assurance becomes nearly an art, which must be manually carried out by skilled staff. Unfortunately, this manual testing does not scale up when videogames complexity grows and some kind of automatisation is needed.

This paper proposes a way for creating *automatic gameplay tests* in order to check that changes in both the source code *and levels* do not affect the global gameplay. Next section reviews the existing test techniques when developing

software. Section 3 describes the component-based architecture that has become the standard for videogames in the last decade and is used in section 4 for creating *logs* of game sessions that are replayed afterwards for testing. Section 5 puts into practise all these ideas in a small videogame, and checks that these tests are useful when levels are changed. The paper ends with some related work and conclusions.

2 Testing and Continuous Integration

Testing is defined by the IEEE Computer Society [1] as the process of analysing a software item to detect the differences between existing and required conditions and to evaluate the features of the software item. In other words, testing is a formal technique used to check and prove whether a certain developed software meets its quality, functional and reliability requirements and specifications. There are many testing approaches, each one designed for checking different aspects of the software. For example, a test can be done with the purpose of checking whether the software can run in machines with different hardware (*compatibility tests*), or whether it is still behaving properly after a big change in the implementation (*regression tests*). Alternatively, expert users can test the software in an early phase of the development (*alpha or beta tests*) to report further errors.

Unit testing is a particularly popular test type designed to test the functionality of specific sections of code, to ensure that they are working as expected. When certain software item or software feature fulfills the imposed requirements specified in the test plan, the associated unit test is passed. Pass or fail criteria are decision rules used to determine whether the software item or feature passes or fails a test. Passing a test not only leads to the correctness of the affected module, but it also provides remarkable benefits such as early detection of problems, easy refactoring of the code and simplicity of integration. Detecting problems and bugs early in the software development lifecycle translates in decreasing costs, while unit tests make possible to check individual parts of a program before blending all the modules into a bigger program.

Unit tests should have certain attributes in order to be good and maintainable. Here we list some of them, which are further explained in [2, Chapter 3]:

- **Tests should help to improve quality**.
- **Tests should be easy to run**: they must be fully automated, self-checking and repeatable, and also independent from other tests. Tests should be run with almost no additional effort and designed in a way that they can be repeated multiple times with the exact same results.
- **Tests should be easy to write and maintain**: test overlap must be reduced to a minimum. That way, if one test changes, the rest of them should not be affected.
- **Tests should require minimal maintenance as the system evolves around them**: automated tests should make change easier, not more difficult to achieve.

3 Game Architecture

The implementation of a game engine has some technological requirements that are normally associated to a particular game genre. This means that we need software elements specifically designed for dealing with different characteristics of a game like the physics system, the audio system, the rendering engine or the user input system. These software pieces are used by interactive elements of the game like the avatar, non-player characters (NPCs) or any other element of the game logic with a certain behaviour. The interactive elements of the game are called "entities", and they are organised inside game levels so that the user experience is entertaining and challenging but still doable. Entities are specified in external files that are processed during the runtime. This way, level designers can modify the playability of the game without involving programmers.

One of the most important tasks of a game engine is the management of the entities that are part of the game experience. An entity is characterised by a set of features represented by several methods and attributes. Following an object-oriented paradigm, each feature is represented by one or more classes containing all the necessary methods and attributes. Connecting the different classes that define the features of an entity leads to different game engine architectures, which can be implemented in very different ways.

The classical way of relating the different entity features is using inheritance. This way, an entity would be represented by a base class that will inherit from multiple classes, granting the entity different features. This method, in addition to the amount of time it requires to design a class structure and hierarchy, present certain additional problems described in [5]:

- **Difficult to understand:** the wider a class hierarchy is, the harder it is to understand how it behaves. The reason is that it is also necessary to understand the behaviour of all its parent classes as well.
- **Difficult to maintain:** a small change in a method's behaviour from any class can ruin the behaviour of its derived classes. That is because that change can modify the behaviour in such a way that it violates the assumptions made by any of the base classes, which leads to the appearance of difficult to find bugs.
- **Difficult to modify:** to avoid errors and bugs, modifying a class to add a method or change some other cannot be done without understanding all the class hierarchy.
- **Multiple inheritance:** it can lead to problems because of the diamond inheritance, that is, an object that contains multiple copies of its base class's members.
- **Bubble-up effect:** when trying to add new functionalities to the entities, it can be inevitable to move a method from one class to some of its predecessors. The purpose is to share code with some of the unrelated classes, which makes the common class big and overloaded.

The most usual way to solve these problems is to replace class inheritance by composition or aggregation associations. Thereby, an entity would be composed

by a set of classes connected between them through a main class that contains the rest of them. These classes are called *components*, and they form entities and define their features.

Creating entities from a set of components is called *Component-Based Architecture*. It solves all the problems mentioned before, but because it does not have a well-defined hierarchy, it is necessary to design a new mechanism of communication between components. The proposed mechanism is based on the use of a message hierarchy that contains useful information for the different components. Whenever a component wants to communicate with any other component, the first one generates a message and sends it to the entity that owns the receiver component. The entity will emit a message to all of its components, and each of them will accept or reject the message and act according to the supplied information. This technique used for communicating is called *Message Passing*.

Message passing is not only important for communicating between components, but also for sending messages from one entity to another. These messages between entities are essential when designing the playability of the game. The reason for using messages between entities is that they need to be aware of the events and changes in the game in order to respond accordingly.

4 Recording Games Sessions for Testing

Traditional software tests are not enough to check all the features that a game has. Things such as playability and user experience need to be checked by beta testers, who are human users that play the game levels over and over again, doing something slightly different each time. Their purpose is to find bugs, glitches in the images or incorrect and unexpected behaviours. They are part of the Software Quality Assurance Testing Phase, which is an important part of the entire software development process. Using beta testers requires a lot of time and effort, and increases development costs. In fact, testing is so important and expensive that has become a business by itself, with companies earning million of dollars each year and successfully trading on the stock market[1].

We propose an alternate form of testing, specifically designed for message-passing architectures with a component-based engine design. The objective is to have "high-level unit tests", based on the idea of reproducing actions to pass the test even when the level changes. To achieve that, we record game sessions and then execute the same actions again, adjusting them slightly if necessary so that the level can still be completed after being modified. Then, we check if the result of this new execution is the expected. Next sections give a detailed explanation on how to do this.

4.1 Using a New Component to Record Game Sessions

In order to record game sessions easily, having a component-based design is a great advantage. Our solution is based on the creation of a new component called

[1] `http://www.lionbridge.com/lionbridge-reports-first-quarter-2015-results/`, last visited June, 2015.

`CRecorder` added to any of the existing entities in the game. After that, when an entity sends a message to all of its listeners, the `CRecorder` component will also receive the message and act accordingly.

For example, a component aimed at recording actions in a game needs to be registered as a listener of the entity Player. Thus, whenever an action takes place in the game the component will be notified. Once the component has been created and the messages handled, saving all the actions to an external file is an easy task. Two different kind of files can be generated with this `CRecorder` component. One of them contains which keys have been pressed and the mouse movements, and the other one contains interesting events that happened during the gameplay, such as a switch being pressed by the player.

4.2 Raw Game Replay

Today, it is not uncommon that keyboard and mouse input logs are gathered by beta testers executables so programmers can reproduce bugs when, for example, the game crashes during a game session. Our use of those logs is quite different: *compatibility* and *regression tests*. Using logs of successful plays, the game can be *automatically* run under different hardware configurations, or after some software changes in order to repeat the beta testers executions to check if everything is still working properly. Loading recorded game sessions and replaying them contributes towards having repeatable and automated tests, which were some of the advisable attributes of unit tests mentioned in section **??**.

Our approach can go further by using the high-level logs for providing feedback when *the execution fails*. While replaying the log, the system not only knows what input event should be injected next, but also *what should happen* under the hood thanks to the high-level events gathered during the recording phase. If, for example, after two minutes of an automatic game session an expected event about a collision is missing, the test can be stopped by reporting a problem in the physics engine.

4.3 Loading Recorded Game Sessions and Replicating The State

The previous raw game replay is not suitable when, for example, the map level has changed, because the *blind* input event injection will make the player wander incorrectly. For that reason, we introduce a new approach for replaying the game that starts with knowing which actions can happen in the game and trying to replicate the state to make the replay of this actions accurate.

Some of the attributes of a game are the actions that can happen, the physical map or the state of the game. When the objective of recording a game is replaying it afterwards, it is necessary to think about what attributes need to be stored in the log. As an example, imagine an action in which a player picks up a weapon from the ground. To replay that, it is required to know which player (or entity) performs the action, what action is taking place and the associated entity (in this case, the weapon). Another important thing to take into account is the time frame for completing the action and the state of the game when it happens.

A player cannot take a weapon if the weapon is not there or if he is not close enough to take it. Therefore, the state needs to be as close as possible when the time frame approaches so that replicating the action is feasible. On the other hand, storing the position of the weapon is not required, as using that position could lead to replaying a wrong action if the map changes. That information is stored in a different file (a map file), it is loaded into the game and it can be accessed during the run time.

With the purpose of modeling all these attributes, we use a powerful representations called Timed Petri nets, which can be very helpful to replay recorded games.

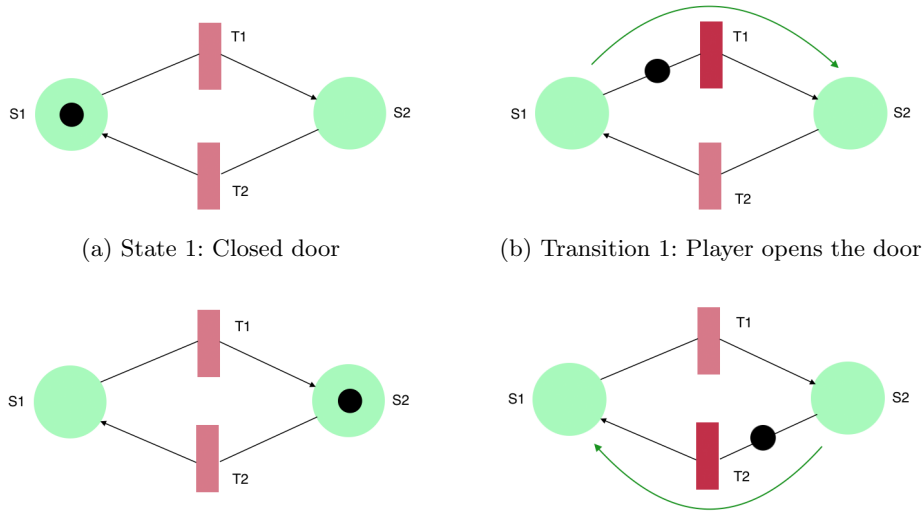
4.4 Modeling the Game With Timed Petri Nets

Petri nets [3, 4] are modeling languages that can be described both graphically and mathematically. The graphical description is represented as a directed graph composed by nodes, bars or squares, arcs and tokens. Elements of a Petri net model are the following:

- **Places**: they are symbolized by nodes. Places are passive elements in the Petri net and they represent conditions.
- **Transitions**: bars or squares represent transitions, which are the actions or events that can cause a Petri net place to change. Thus, they are active elements. Transitions are enabled if there are enough tokens to consume when the transition fires.
- **Tokens**: each of the elements that can fire a transition are called tokens. A discrete number of tokens (represented by marks) can be contained in each place. Together with places they model system states. Whenever a transition is fired, a token moves from one place to another.
- **Arcs**: places and transitions are connected by arcs. An arc can connect a place to a transition or the other way round, but they can never go between places or between transitions.

Figure 1 shows an example on how to model the actions of opening and closing a door with a Petri net. Figure 1a shows the initial state, in which a door is closed and a token is present. Figure 1b shows the transition that takes place when a player opens the door. Then, the new state becomes 1c, making the token move from S1 to S2. If then the player performs another action and closes the door (showed in transition 1d), the token returns to the initial state again, 1a. Notice that in figures 1b and 1d the tokens are in the middle of one of the arcs. Petri Net models do not allow tokens to be out of places, but in this example they have been put there to highlight the movement of the token.

Classic Petri nets can be extended in order to introduce an associated time to each transition. When transitions last more than one time unit, they are called timed Petri nets. Introducing time in the model is essential for replaying games because actions are normally not immediate. For instance, if we want to replay an action such as “opening a door”, firstly the player needs to be next



(a) State 1: Closed door

(b) Transition 1: Player opens the door

(c) State 2: Opened door

(d) Transition 2: Player closes the door

Fig. 1. Modeling the actions of opening and closing a door with a Petri net.

to the door, and then perform the action of opening it. That means that the transition could be much longer than just a time unit, and other actions could be in progress at the same time. For that reason, modeling the game as a timed Petri net makes it easier than modeling it as a state machine.

After loading a recorded file to the game with the purpose of replaying it, actions need to be performed in a state as close as possible as the original state. Moreover, actions are normally ordered: a player cannot walk through a door if the door has not been opened before. In practice, some actions cannot be performed until some other actions have finished.

If the game has several entities capable of firing transitions, they can be represented as different tokens in the Petri net model. A token firing a transition may cause some other tokens to change their place, which is exactly what happens in games. Actions may affect several entities, not just one, so using Petri nets to model that behaviour seems to be reasonable.

When we detect that a player made an action in the first execution of the game and the corresponding Petri net transition is enabled (it is possible to complete the action), the appropriate messages have to be generated and injected to the application. There is no difference between messages generated when a real person is playing and the simulated messages injected. Components accept the messages, process them and respond accordingly. For that reason, the resulting file should be exactly the same and it is possible to see that the actions that are happening in the game have not changed.

4.5 Replaying Game Sessions and Running Tests

There are two possible ways of replaying a recorded game session: replicating the exact movements of the user or trying to reproduce specific actions using an artificial intelligence (AI). Replicating game sessions can be useful when we want to make compatibility tests (running tests using different hardware) or regression tests (introducing software improvements but not design changes). However, replicating game sessions consists on simulating the exact keys pressed by the user. These tests are very limited, since the slightest changes on the map make them unusable. Reproducing specific actions can solve this limitation. Saving detailed traces of the game sessions that we are recording gives us the chance to use that information to make intelligent tests using an AI. That way, we can still use the recorded game sessions to run tests even if the map also changes.

Once that replaying games is possible, it can be used to design and run tests. An input file with information for the tests can be written. In that file, the tester can define several parameters:

- **Objectives**: the tester can specify which messages should be generated again and if they should be generated in order or not. If those messages are generated, it means that the particular objective is fulfilled.
- **Maximum time**: sometimes it will not be possible to complete one of the tasks or objectives, so the tester can set a maximum time to indicate when the test will interrupt if the objectives are not completed by then.
- **User input file**: the name of the file containing all the keys pressed when the user was playing and the associated time.
- **Actions input file**: the name of the file with all the high level actions that the user performed, when he did them and the attributes of those actions.

Using those two ways of replaying the game can lead to the generation of very different tests. It is also possible to combine both ways and run a test that reproduces the actions in the input file and if and only if the AI does not know how to handle a situation it replicates the movements in the user file.

Several different tests can be run to check whether it is possible to complete a task. If any of them succeeds, then the objective is fulfilled. It is also possible to launch the game more than once in the same execution so that various tests are run and the objectives checked.

5 Example of Use: Time and Space

Time and Space is a game developed by a group of students of the *Master en Desarrollo de Videojuegos* from the Universidad Complutense de Madrid. This game consists on several levels with switches, triggers, doors and platforms. The player has to reach the end of each level with the help of copies from itself. These copies will step on the triggers to keep doors opened or platforms moving while the player goes through them. Sometimes clones will even have to act as

```
{
 "timestamp" : 73400,
 "type" : "TOUCHED",
 "info" : {
   "associatedEntity" : {
     "name" : "PlayerClone1",
     "type" : "PlayerClone"
   },
   "entity" : {
     "name" : "DoorTrigger1",
     "type" : "DoorTrigger"
   },
   "player" : {
     "name" : "Player",
     "position" : "Vector3(37.0423, -2.24262e-006, -6.53315)",
     "type" : "Player"
   }
 }
}
```

Fig. 2. Example of a trace generated when a copy push a button

barriers against enemies to keep them from shooting the player. Also, there are platforms in the game that cannot be traversed by the player, but only by his copies. These copies are not controlled by the person that is playing the game. Their movements are restricted so they just reproduce the actions they made in the previous execution of the level, before the player copied itself.[2]

When the tester activates the recording of traces, a file in json format is generated. This format was chosen because of its simplicity to read and write to a file using different styles (named objects or arrays). Figure 2 shows a trace recorded when a player clone touches a button. This file contains the information of the execution: what actions were performed, when they took place and the entities that were associated to that action (player, switch, enemy, etc). Note that the entity position is not recorded because it can be read from the map file. The player position is also necessary to imitate the movements when this trace is replayed.

To reproduce the previously recorded traces adapting them to the new level, two different type of events can be distinguished:

- **Generated by the player:** these are the actions generated by the player itself. The recorded trace available in the file consists on the timestamp, the type of the action performed and the entity related to the action. Some other information may be stored depending on the type of the action. The actions can either be reproduced immediately or reproduced using the AI of the videogame.

[2] Full gameplay shown at `https://www.youtube.com/watch?v=GmxV_GNY72w`

– **Generated by some other entity:** in this case, we try to make the state of the player as close as possible as the player state when the trace was recorded. With that purpose, we store the information needed to know the player state along with the recorded event. In Time and Space, the state of the player only consists on its position in the map, so that is the only information we need to save in the trace log. Figure 2 shows an example of this type of traces. For replicating the game state, we use again the AI of the videogame, which is responsible for the moves of the player, making sure that they are valid.

In order to detect and reproduce traces, the actions have been modeled by a Petri net, introduced in section 4.4. Thanks to these models, it is possible to replicate the actions in the same order that was recorded in the first place. This is not something trivial. Some of the actions can be reproduced before some previous ones if the player does not know how to carry out that action and gets stuck without doing anything. Because of the nature of Time and Space and the possibility of creating clones, all the Petri nets generated have a fed back structure, with multiple tokens moving from state to state inside the net.

Timed Petri nets are used instead of simple Petri nets because most of the actions cannot be reproduced until previous actions are done. For example, when a clone of the player presses a button to open a door, it is necessary to wait until the door is open before starting to go through the door, even if the clone has already pushed the button. For that reason, performing all the actions from a trace in order makes it easy to have almost an exact reproduction of the gameplay.

Even if this solution is almost exact, this method has some limitations. Using Petri nets to reproduce traces means that the videogame needs to have an AI implemented, capable of controlling the player inside the game. Fortunately, there are a lot of games (like Time and Space) that use an AI for directing all non-player characters movements that can be reused for that. However, despite the fact that the results we have from Time and Space are very promising, there are some use cases in which the reproduction of traces is not going to work properly. One of these examples is when the player needs to push a button that is not on the same level as the ground in which the player is standing. In this case, the AI of the videogame cannot find out where the button is, so the reaction of the player will just be waiting there without moving. This situation shows that the testing method proposed is valid, but remarks that an AI designed for control the player is needed.

In order to automatise the testing phase we created a configuration file where the tester can choose if he wants to record the game or load previous executions that were recorded before. If he chooses to reproduce something recorded, then he can specify the name of the file that contains the actions that are going to be loaded and reproduced.

We have recorded game traces from a level that the user played and completed successfully, that is, reaching the end of it. To check that the programmed replay system works with Time and Space, we reproduced those exact traces without

any modification in the same level. However, the map was slightly changed in those new executions. Some of the tests that we carried out were the following:

- Firstly we tried to change the map and move the switches to reachable places from the position in which they were initially placed. We then repeated the same test but we also changed the end of level mark. Both of the tests were still feasible after all the changes, and by replaying the same traces it is still possible to complete the objectives and reach the end of the level. Another test we made consisted in placing one of the switches behind a closed door. In this case, we could see that the player detected that the new position was not reachable and therefore he did not move from the position he had before detecting he had to press the switch.
- After that, we recorded traces from a level in which the player needs three different copies to win the level. To reach the end of the level, the player has to go through a moving platform and some rays have to be deactivated. That is why the player needs his clones. If the player tries to go through the rays before deactivating them, he dies. To make the tests, several changes were added to the map. For example, we tried to pass the test after interchanging the switches between them. By doing that, they were closer or further away from the player. Running the test allows us to see that despite of all the changes, it is still possible to complete the level without difficulties. We recorded a video that shows the reproduction of two of these tests[3].

Because the game has been implemented following a standard component-based architecture, it was not necessary to make major changes in it. To record the game session we only added the `CRecorder` component as described in 4.1, which receives all the messages generated during the gameplay. The code for the `CRecorder` component and the required changes made in the original implementation are about 8 KB. Moreover, two new modules of about 127 KB were created for recording and replaying the messages. These modules were designed with a general purpose and only slightly modified to work with this game.

6 Related Work

With systems growth in size and complexity, tests are more difficult to design and develop. Testing all the functions of a program becomes a challenging task. One of the clearest examples of this is the development of online multiplayer games [6]. The massive number of players make it impossible to predict and detect all the bugs. Online games are also difficult to debug because of the non-determinism and multi-process. Errors are hard to reproduce, so automated testing is a strong tool which increases the chance of finding errors and also improves developers efficiency.

Monkey testing is a black-box testing aimed at applications with graphical user interfaces that has become popular due to its inclusion in the Android Development Kit[4]. It is based on the theoretical idea that a monkey randomly

[3] `https://www.youtube.com/watch?v=1OBlBKly1pk`
[4] `http://developer.android.com/tools/help/monkey.html`

using a typewriter would eventually type out all of the Shakespeare's writings. When applied to testing, it consists on a random stream of input events that are injected into the application in order to make it crash. Even though this testing technique blindly executes the game without any particular goals, it is useful for detecting hidden bugs.

Due to the enormous market segmentation, again specially in the Android market but more and more also in the iOS ecosystem, automated tests are essential in order to check the application in many different physical devices. In the cloud era, this has become a service provided by companies devoted to offer cloud-based development environments.

Unfortunately, all those testing approaches are aimed at *software*, ignoring the fact that games are also maps, levels and challenges. We are not aware of any approach for automatic gameplay testing as described in this paper.

7 Conclusions and Future Work

Although some methods for automatising gameplay tests exist, they are aimed at checking software aspects, not taking into account the necessity of checking that both the maps and levels are still correct. Because these levels and maps also evolve alongside software while developing games, finding a way to run automatic tests to check that all the modifications introduced into levels are consistent is a must.

In this paper we have introduced a proposal on how to carry out these tests. Taking advantage of the component-based architecture, we have analised the cost of introducing the recording and replaying of traces in games, which allow us to automatically repeat gameplays after modifying the levels. This proposal has been tested with a simple game, proving the viability of the idea.

Despite the promising results, the work we carried out is still on preliminary stages. It is still necessary to test this technique in more complex games, as well as proving its stability to more dramatic changes in them.

References

1. Software Engineering Technical Committee of the IEEE Computer Society: IEEE Std 829-1998. IEEE-SA Standard Board (1998)
2. Meszaros, G: XUnit test patterns: refactoring test code. Addison-Wesley, (2007)
3. Popova-Zeugmann, L: Time and Petri Nets. Springer-Verlag Berlin Heidelberg (2013)
4. Estevão Araújo, M., Roque, L.: Modeling Games with Petri Nets. DIGRA2009 - Breaking New Ground: Innovation in Games, Play, Practice and Theory (2009)
5. Gregory, J.: Game Engine Architecture. A K Peters, Ltd. (2009)
6. Mellon, L.: Automatic Testing for Online Games. Game Developers Conference, 2006.