DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2016

Detecting Synchronisation Problems in Networked Lockstep Games

HAMPUS LILJEKVIST

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF COMPUTER SCIENCE AND COMMUNICATION

KTH Computer Science
and Communication

Detecting Synchronisation Problems in Networked Lockstep Games

(Upptäcka synkroniseringsproblem i nätverksuppkopplade lockstep-spel)

Master of Science in Engineering in Computer Science and Engineering
Master of Science (120 Credits) in Computer Science
KTH Royal Institute of Technology
School of Computer Science and Communication

HAMPUS LILJEKVIST

hlilje@kth.se

Master's Thesis at KTH CSC

KTH CSC Supervisor:	Dilian Gurov
Principal Supervisor:	John Wordsworth
Examiner:	Mads Dam
Principal:	Paradox Development Studio
Course:	DA222X

Stockholm, Sweden 2016-07-07

Abstract

The complexity associated with development of networked video games creates a need for tools for verifying a consistent player experience. Some networked games achieve consistency through the lockstep protocol, which requires identical execution of sent commands for players to stay synchronised.

In this project a method for testing networked multiplayer lockstep games for synchronisation problems related to nondeterministic behaviour is formulated and evaluated. An integrated fuzzing AI is constructed which tries to cause desynchronisation in the tested game and generate data for analysis using log files. Scripts are used for performing semi-automated test runs and parsing the data.

The results show that the test system has potential for finding synchronisation problems if the fuzzing AI is used in conjunction with the regular AI in the tested game, but not for finding the origins of said problems.

Referat

Upptäcka synkroniseringsproblem i nätverksuppkopplade lockstep-spel

Komplexiteten förenad med utveckling av nätverksuppkopplade dataspel skapar ett behov av verktyg för att verifiera en konsistent spelarupplevelse. Vissa nätverksspel hålls konsistenta med hjälp av lockstep-protokollet, vilket kräver identisk exekvering av skickade kommandon för att spelarna ska hållas synkroniserade.

I detta projekt formuleras och evalueras en metod för att testa om nätverksuppkopplade flerspelarspel lider av synkroniseringsproblem relaterade till ickedeterministiskt beteende. En integrerad fuzzing-AI konstrueras som försöka orsaka desynkronisering i det testade spelet och generera data för analys med hjälp av loggfiler. Skript används för att utföra halvautomatiserade testkörningar och tolka data.

Resultaten visar att testsystemet har potential för att hitta synkroniseringsproblem om fuzzing-AI:n används tillsammans med den vanliga AI:n i det testade spelet, men inte för att hitta de bakomliggande orsakerna till dessa problem.

Preface

This degree project in computer science and communication was performed at the School of Computer Science and Communication (CSC) at KTH Royal Institute of Technology and Paradox Development Studio.

I would like to thank Dilian Gurov for supervising this project at KTH CSC and John Wordsworth for supervising this project at Paradox Development Studio. I would also like to thank Mads Dam for examining this project and Ann Bengtsson for coordinating this project.

Contents

Glossary

AI Artificial Intelligence.

bug A software error caused by a subset of the source code not working as intended.
build One version of a software system.

checksum Defined in § 4.2.
codebase All the source code compiled into a program.
command The internal representation of an intent or request sent by a player to perform an action in the game.
CSV Comma-Separated Values.

debugging Testing software for bugs.

fuzzing Defined in § 4.1.
- **coverage** Defined in § 4.1.1.
- **depth** Defined in § 4.1.1.
- **efficiency** Defined in § 4.1.1.

fuzzing AI Defined in § 5.4.
- **stage** Defined in § 5.4.2.

game state The collection of all game entities and information such as units and statistics for one instance of a game.
GUI Graphical User Interface.

in-synchronisation Defined in § 2.1.

lockstep protocol Defined in § 2.3.
log file Defined in § 4.3.

multiplayer game A game with support for multiple players, often via a network connection.

OOS Out-Of-Synchronisation.
out-of-synchronisation Defined in § 2.1.

PRNG Pseudorandom Number Generator.

regular AI Defined in § 5.2.
- **indifferent** Defined in § 6.1.3.
- **passive** Defined in § 6.1.3.

SUT System Under Test.

tick Defined in § 5.2.
timestamp The logged time when an event occurred.

Chapter 1

Introduction

In this chapter, the problem statement, its context and relevance are introduced.

1.1 Background

Video games are complex systems developed by large numbers of people. This means that bugs and performance issues emerge frequently, often without being detected until later stages of development. Networked multiplayer games are additionally challenging in that they must reliably synchronise their game states over a network connection. This gives rise to a new family of bugs related to synchronisation, which are inherently hard to discover and fix in the unpredictable environment they occur. Identifying such problems earlier would significantly increase the quality of the final product.

Synchronisation in networked games can be achieved in multiple different ways. Inherent to these techniques are also different types of problems. Many games use some form of *dead reckoning* over a client-server connection (§ 2.2.1) to allow loose consistency with prediction and error correction. What is particularly interesting, is synchronisation in *lockstep* games (§ 2.3). In essence, any inconsistencies are prohibited as all clients synchronise their game states each turn. By assuming that the game simulation is deterministic across all participants, it is possible to only distribute action commands in the network. This is a necessity for games with game states too large to continuously send over networks.

Now that games have become an integral part of our culture, the implications of how they are being developed cannot be overlooked. Time and cost estimates are hard to make, often resulting in stressful work to meet deadlines and budgets. Having access to good testing tools and methodologies will alleviate some of this burden. If less resources are spent on testing, it also indirectly makes the development process slightly more sustainable.

This project concerns synchronisation problems in lockstep games, in this case using a client-server server connection. Paradox Development Studio have a long history of developing grand scale strategy games of this type, one of which is used

for testing. The system under test (SUT) is the networked multiplayer lockstep game *Europa Universalis IV*.

1.2 Intended Readers

The project is of interest for game developers who are looking for new methods for finding synchronisation problems in networked games. When developers focus on functionality over correctness, there is always a chance that proper testing tools and methodologies get overlooked. Due to the universal appeal in testing applications for bugs related to synchronisation, this project should also be of interest outside the area of game development. Developers of general distributed applications may use parts of the testing methodology for finding synchronisation problems, provided their applications are sufficiently similar in their architecture. Little formal research has been done on testing networked games for synchronisation problems, making this work interesting for researchers in the area of software testing.

1.3 The Problem

The purpose of this thesis project is to investigate how it is possible to test networked multiplayer lockstep games for synchronisation problems. In this section, the problem is more precisely defined.

1.3.1 Research Question

How can one dynamically detect whether a given version of a networked multiplayer lockstep game will cause synchronisation problems using analysis of data produced by simulated players, assuming the network architecture and networking code are working as intended?

1.3.2 Problem Definition

Discovering bugs related to network synchronisation during development is both hard and time consuming. This creates a need for tools and methods for evaluating whether a given version of an application has these problems. Networked multiplayer games are especially challenging since their entire game states need to be synchronised between players. This opens up for a number of possible sources of errors, such as different hardware specifications yielding different results for equal game input.

Lockstep games continuously post a list of commands to each client, but these commands may behave differently on different types of hardware. As the game session progresses, any potential desynchronisation will cause the game states to drift further apart from each other. It is not uncommon for these games to go out-of-synchronisation (§ 2.1) and need to be reset in order to continue playing. This nondeterminism (§ 2.4.2) is the bane of lockstep games, and this project evaluates a method for finding nondeterministic behaviour.

1.3.3 Delimitations

There are many different reasons for desynchronisation in networked games, which can be broadly divided into network-related and implementation problems. To limit the scope of this project and make the task feasible, it is assumed that the network architecture and networking code are working correctly. In other words, packets are assumed to arrive in the correct order without significant latency (§ 2.4.1) at the clients. This project instead focuses on synchronisation problems related to how messages are processed once they arrive.

1.4 Choice of Methodology

In static program analysis, the behaviour of code is reasoned about without executing it. Utilising this technique would require formalising the notion of nondeterminism stemming from differences in hardware, in order to mathematically prove which parts of the code may suffer from it. This type of analysis was deemed out of this project's scope. Combinatorial explosion of the game state space would also make a model checking treatment exceptionally troublesome. For these reasons, a highly dynamic approach has been chosen. When using dynamic testing, one must make sure that the test system has proper coverage. If done properly, it enables you to find synchronisation problems that could manifest themselves during real-world use.

It is possible that the SUT is free of synchronisation bugs; therefore representative bugs (§ 5.9) have been manually added to the codebase. This naturally introduces a bias in what bugs are chosen and where they are added, but it also enables finer analysis since the probable cause of desynchronisation can be deduced for comparison.

1.5 Contributions of This Work

The research question is primarily connected to current research in its focus on network synchronisation, and the problems surrounding it. Today, applications are increasingly being used over network connections. Better methods for software testing are constantly being researched for handling the inevitable increase in scale and complexity of these applications. In spite of this, few papers have been published on testing networked games for synchronisation problems, making this project particularly interesting for researches and developers working in said area. Here, a dynamic fuzzing method (§ 4.1) for finding synchronisation problems in networked lockstep games is proposed and evaluated.

1.6 Structure of This Thesis

Chapter 1: Here a brief background to the problem is given and the research question is defined.

Chapter 2: Here the theoretical background of this project is presented.

Chapter 3: Here some related work in the area is outlined.

Chapter 4: Here the necessary concepts for understanding the test methodology are explained.

Chapter 5: Here the implementation of the test system and the test methodology are described.

Chapter 6: Here the obtained results are provided.

Chapter 7: Here the results are interpreted and discussed to form the conclusion of this project.

Chapter 2

Background

Little formal research has been done on detecting synchronisation problems in networked multiplayer games. While many different methods for synchronisation have been proposed, no study has been found to evaluate how to test them for desynchronisation. In this chapter, the crucial concepts for the research question are described.

2.1 Definition of Being In-Synchronisation

Mauve et al. discuss *consistency* and *correctness* for "continuous replicated applications" changing over time without external input, such as networked computer games [22]. They use the term *site* to describe a replicated application located on a computer connected to other replicated applications over a network. Consistency is defined as follows:

> At any time t the state at any two sites i and j must be the same, if both sites have received all operations that are supposed to be executed before t [22, pp. 48–49].

This is requirement is weak by itself since it only requires sites to agree on their states, while posing no requirements on the states being correct.

A virtual *perfect* site P has the equivalent state of a single non-distributed application on which the exact same operations would have been performed [22]. Correctness is defined as follows:

> At any time t the state at any site i must be the same as the state of the virtual perfect site P, provided that i has received all operations that are supposed to be executed before t [22, p. 49].

The correctness requirement does not explicitly make use of concurrency concepts such as linearisability, sequential consistency or quiescence, instead it ensures the application has the desired properties by relating it to a non-distributed application.

Correctness is strictly stronger than consistency since the former implies the latter but not the converse.

For the sake of this project, a networked game is considered *synchronised* or *in-synchronisation* if its game state is both consistent and correct. Likewise, a networked game is considered *desynchronised* or *out-of-synchronisation* (OOS) if its game state is not both consistent and correct.

2.2 Networked Game Architectures

This section provides an overview of the most common types of networked game architectures and their advantages and disadvantages, to contribute to the understanding of synchronisation on a networking level. The tested game in this project is of the client-server kind, but lockstep games may be implemented with other types of network architectures.

2.2.1 Client-Server Architecture

The traditional network model for multiplayer games is the *client-server* model [4, 11]. In essence, it consists of one authoritative server connected to a number of clients (players), as illustrated in figure 2.1 [4, 11, 30]. The server is responsible for the game simulation while the clients send the input commands to be executed. When the server has updated the game state, the new states are sent back to all clients for them to render [4, 30]. Since one authoritative server maintains the entire game state, it is easy to uphold consistency and avoid cheating [21, 30]. The client-server model is architecturally simple, and administrators only need to worry about one connection [30].

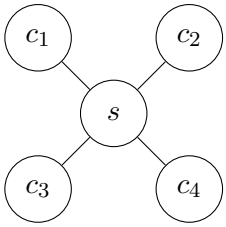

Figure 2.1. Client-server architecture.

However, the client-server model suffers from its simplicity; if the latency of the network connection is large, then the same latency will be perceived by the players [4]. The server itself introduces latency when messages are being relayed to players [30]. If this network model shall be used over the Internet, it is necessary to use additional techniques to alleviate the perceived latency [4]. The centralised nature of the architecture means that a failing server will disrupt the entire system, and the server itself may act as a network bottleneck [11].

2.2.2 Peer-to-Peer Architecture

In the *peer-to-peer* model, there is no central server responsible for the game simulation [11, 30]. Instead, each peer (player) in the network has its own local copy of the global game state which it simulates. This architecture is illustrated in figure 2.2. The primary reason for choosing a peer-to-peer network architecture is its resistance to failure [11]. Unlike the client-server model (§ 2.2.1), no central server can disable the entire system [5, 11]. Another advantage is the reduced network latency when players do not need to communicate via a server [5, 30].

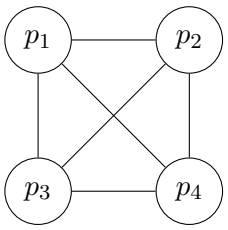

Figure 2.2. Peer-to-peer architecture.

The bandwidth requirement of a peer-to-peer network is higher than that of a corresponding client-server network [11, 30]. This is because all players need to broadcast their changes to the game state to each other when no central server upholds a consistent game state. Compared to client-server networks, peer-to-peer networks are more difficult to configure and implement [30]. Another disadvantage is that players are able to cheat in the absence of a central authoritative server [11, 30]. Upholding game state consistency among players is harder with a peer-to-peer architecture than a client-server architecture [30].

2.2.3 Distributed Architecture

The *distributed* network architecture is an umbrella term for architectures where multiple servers upholding the game state are geographically distributed [11]. Ferretti refers to multiple papers concluding that the distributed architecture is in many situations superior to other networking models. According to Ferretti, there are three different types of distributed architectures.

One type of architecture consists of multiple client-server networks (§ 2.2.1) connected together [11, 31], where the servers manage different game sessions [11]. This is illustrated in figure 2.3. In these implementations there are fewer clients (players) connected to each server, thus reducing the individual capacity requirements [11, 31]. The shortcomings of the client-server model are however still present [11]. Each server becomes a point of failure for its network cluster, and adds a hard limit on how many clients can connect to the game session. One alternative is to use a peer-to-peer network of so called super-peers [35]. Each super-peer acts as a server controlling a set of clients, while simultaneously working as a peer in the peer-to-peer network. By combining aspects of the client-server model with distributed search,

super-peer networks can achieve good performance. With added redundancy, no super peer becomes a single point of failure in the network, though it can still become a bottleneck.

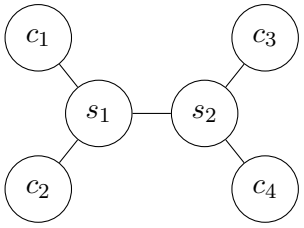

Figure 2.3. Distributed architecture.

Another implementation is that of distributed servers holding a subset of the game state [20, 31]. This method can have the advantage of reduced individual server load, as shown by distributing pieces of a virtual world to multiple servers [20]. Players may suffer from increased latency in a virtual world if they happen to enter an area of which a geographically distant server is responsible [11]. As with client-server architectures, fault tolerance for failing servers is low.

As a third alternative, Ferretti describes a network architecture where all distributed servers hold a replica of the game state [11]. This architecture is also known as mirrored game server architecture [9] or proxy-server topology [25]. Such architectures require special algorithms for ensuring consistency between all servers, but there are multiple advantages to be had [11]. No server becomes a single point of failure, the network load can be balanced, and the problem of network congestion may be reduced. Another advantage is that the latency may be greatly reduced, provided the servers are sufficiently distributed [9]. Ferretti points out how it is possible to maintain large game worlds by combining replicated servers with servers containing partitions of the virtual world, thus coupling alternative two and three [11].

2.3 Lockstep Synchronisation

In this section, the common variants of lockstep synchronisation are described, which is the synchronisation method used in the SUT.

2.3.1 Basic Lockstep

Baughman and Levine introduce the cheat-proofed *lockstep protocol* as a method of synchronising game states over networks [3]. It highly resembles the "simultaneous simulations" used in networked real-time strategy games such as Age of Empires, where all players run the exact same game simulation in synchrony [5]. While these types of games typically require the simulation to run in discrete time steps, it is not sufficient to stop and wait for all other players before sending the decision for

the next turn [3]. A cheating player could exploit the system by waiting for other players' commands before announcing their own action for the upcoming turn. By doing so, it would be possible to counter the moves of the opponents. Baughman and Levine therefore propose a modified version of this protocol consisting of a "decision commitment step". Unfortunately this type of protocol still comes with an inherent performance penalty, since the game simulation can only run as fast as the slowest player [3, 5].

The idea behind the protocol is to create verifiable hashed *commitments* of the players' decisions [3]. When one turn is complete, all players make their decisions for the next turn but do not announce them immediately. Instead they compute cryptographically secure one-way hashes of their decisions, which may include randomised padding for obscuring what decisions were made and avoid collisions. The players then broadcast these hashed decisions to each other, thus making commitments solely based on their knowledge of the current turn. After all players have made their announcements, they reveal their non-hashed decisions. The integrity of these decisions can be verified by comparing their hashes to the values received previously. Baughman and Levine explain how each player can only base their decisions for the next turn on their knowledge of the current turn, meaning there is no advantage in waiting. They then mention a possible optimisation; if all players except one have already committed their decisions, the remaining player may reveal their decision immediately without commitment.

2.3.2 Asynchronous Synchronisation

Baughman and Levine also present the concept of *asynchronous synchronisation* as an improvement to the lockstep protocol [3]. As before, hashes are used to verify the integrity of players' intentions. However, clients only enter into lockstep when they need to interact with each other, otherwise they asynchronously advance their game clocks. To detect client interaction, dynamically changing virtual *spheres of influence* are used. If the spheres of two players do not intersect, they can safely advance their game states independently. The decentralised game clock makes this improved protocol more resilient to packet loss or slow clients, while keeping the correctness and fairness properties.

2.3.3 Pipelined Lockstep

The *pipelined lockstep protocol* (PLS) is an optimisation of the basic lockstep protocol which exploits the presentation delay in *bucket synchronisation* [19]. Ho Lee et al. explain how it is sufficient for some games to delay interaction resolution some number of turns. If a player makes a decision which does not disagree with the pending plain texts of other players, then the new commitment may be sent before their own pending plain texts are. By introducing a delay before resolving an interaction, it is possible to mitigate some irresolvable interactions caused by network latency (§ 2.4.1). In the PLS protocol, the *pipe size* refers to the number of commitments

that can be sent by a player without receiving the commitments of the other players (the basic lockstep protocol has a pipe size of 1).

2.3.4 Adaptive and Sliding Pipeline

Cronin, Filstrup and Jamin further improve the lockstep protocol by making it dynamically adapt to network conditions [8]. Their *adaptive pipeline protocol* uses *dead reckoning* to disguise perceivable jitter and make the frame rate constant. They also introduce the *sliding pipeline protocol* which uses a "send buffer" to hold the moves being generated while the pipeline size is dynamically adjusted. These protocols are able to prevent certain types of cheats possible in other lockstep protocols.

2.4 Synchronisation Problems

The primary causes for synchronisation problems in networked games which have been identified are described in this section.

2.4.1 Network Latency

There are multiple reasons behind network latency (delay), such as IP packet travel time, network congestion [7], routing and packet processing time [31]. Compensating methods to handle packet loss have the largest impact on networked game performance, often manifested as latency [7]. While it is sometimes possible to decrease latency, it can never be completely removed due to the inherent travel time of light [29]. If a networked game uses the client-server architecture (§ 2.2.1), the latency between the client and the server may impact the responsiveness and performance of players [7]. How much latency is tolerable depends on game type. Games played in first-person are typically more sensitive than third-person or so-called omnipresent games, omnipresent meaning that the player can view and manage different parts of the game world simultaneously with full control of their resources.

Apart from affecting the responsiveness of a game, latency may cause sent player actions to be discarded completely [10]. If games did not use synchronisation methods, network latency could cause the game states to diverge over time [9]. Even in synchronised games, latency may cause short-term inconsistencies [22], particularly if *client-side prediction* is used [4]. Latency may cause errors when computing the game state due to events arriving out-of-order [11].

2.4.2 Nondeterminism in Clients

Networked games based on lockstep synchronisation (§ 2.3) require the game simulation to be entirely deterministic for all players, as only the player actions are sent between clients [23]. Floating point calculations may be performed differently

on different types of hardware, causing the game states to become desynchronised. Uninitialised variables may also lead to desynchronisation [12, 28, 33].

Pseudorandom number generators (PRNGs) can potentially cause synchronisation problems [23]. To avoid desynchronisation, they need to produce the same numbers for all clients. This can normally be achieved by initialising all the PRNGs with the same seed, and making sure that the same number of calls to the PRNGs are made each game turn in the same sequence and in the same part of the program [15, ch. 6]. Alternatively, the server can precompute the pseudorandom numbers and distribute them to the clients [23]. One caveat is that the underlying PRNG of a pseudorandom function in a programming language may sometimes be unspecified, and thus potentially different between platforms or compilers [15, ch. 6]. Networked games must therefore only use pseudorandom functions which are clearly specified.

In programming languages, the order of evaluation of subexpressions in expressions may be undefined [32, ch. 10]. This can introduce nondeterminism when compilers are free to decide whether the function `f()` or `g()` in the expression `f() + g()` should be evaluated first, as an example.

Algorithms can have weak guarantees of consistency leading to nondeterministic behaviour. For instance, *unstable* sorting algorithms may yield different results across different compilers when the ordering of identical elements in an array is not retained [33].

2.4.3 Unmatched Game Files

While the game implementation logic may be perfectly synchronised, desynchronisation can still occur if players have modified or different versions of game files [12, 28, 34]. Since this is an issue unrelated to implementation of games, it is not considered in this project.

Chapter 3

Related Work

While no academic papers on testing networked lockstep games for synchronisation problems have been found, research on related methods of software testing has been identified. These methods are described in this chapter. Outside academia, there are examples of directly related work on detecting desynchronisation in networked games. This work is also presented in this chapter.

3.1 GUI Testing

When testing the graphical user interface (GUI) of an application, it is common to use *capture/replay tools* [26]. GUI testing serves two purposes, it verifies the quality of the tested application from a user interaction point-of-view, and it verifies the functionality of the application. Capture/replay tools are advantageous in that they allow developers to save time by not having to write hand-made tests, though they suffer from lack of automatic test generation, inability to interface with the application under test, and are inherently tied to the current layout of the GUI.

A capture/replay tool works by recording the tester's actions in a log file (§ 4.3), which is later used to reproduce the same input [26]. The tester thus has to manually perform the test procedure once to construct the test. Some capture/replay tools also record interaction on a lower GUI component level and are thus sensitive to code changes [6]. After recording, the log file is extended with assertions to verify the behaviour of the tested application [16]. Lastly, the logged data is used to replay the application and evaluate the assertions.

Visual GUI testing is a more novel technique which uses scripts and image recognition to interact with GUI components such as buttons or images [6]. The scripts are commonly written by hand, as opposed to recording them. Visual GUI testing has some advantages over record/replay, it is not sensitive to layout changes of the GUI which would otherwise break recorded scripts, and GUI code changes do not affect it. On the other hand, the visual technique suffers from sensitivity to visual changes such as colour or shape of objects.

Another method for GUI testing is to define models by generating test cases from

finite state machines [6]. These models are often expensive in terms of manual construction and scalability. However, work has been done on automatically generating models through *GUI ripping*.

While the SUT in this project is normally interacted with through its GUI, GUI testing is only moderately utilised here. Since the source code of the SUT is available, finer control of the testing is obtained by removing this layer of abstraction and modifying the system itself. This also makes it possible to detect desynchronisation bugs which may be hard or impossible to trigger using the GUI alone. In this project, GUI testing is merely used to initiate the multiplayer game session for testing.

3.2 Distributed Replay

Geels et al. (2006) present a programming library for performing deterministic *replay debugging* of distributed C/C++ applications, known as `liblog` [14]. The library intercepts calls to `libc` in order to log their results, and it also logs the contents of all incoming network messages. The execution of the analysed system is then replayed locally using downloaded logs and checkpoints. They claim that the biggest downside of their library is the large amount of storage needed for the log data, while it is efficient in terms of networking performance and has "sufficiently small" CPU overhead for many applications. Unfortunately, `liblog` is unable to log or replay threads in parallel on a multiprocessor machine.

Geels et al. (2007) propose an improvement to the described technique in a later paper under the name *Friday*, which combines `liblog` with a scripting language for expressing conditions and actions [13]. Friday implements a central debugging console which supports the embedded scripting language and connects to replay instances. The tool uses distributed *watchpoints*, which can for instance watch variables, and *breakpoints*, which akin to traditional breakpoints let the debugger react to specified points in the source code. The commands triggering these watchpoints and breakpoints are written in Python.

Friday inherits the downsides of `liblog`, and the replay may be slowed down by false positives [13]. Some examples of downsides specific to Friday are that it may be hard to start execution from the middle of a replay trace, and that runtime errors can arise from the scripting. The authors claim that Friday gives developers sophisticated methods for checking global invariants during distributed executions.

DDOS is the name of a more recent tool for replay or distributed systems, which provides fully deterministic behaviour [17]. It is made by Hunt et al. and provides two modes, record/replay with space-efficient logs and deterministic execution. When set to the latter, DDOS executes the entire distributed system deterministically. External inputs from other distributed systems such as network packets and user input are deterministically simulated. The tool supports all types of distributed POSIX-compatible applications, and does not modify application binaries or save contents of network messages. DDOS is built upon *dOS*, an operating system implementing "deterministic process groups". It expands the previous work by

reducing the size of log files and allowing deterministic inter-node communication. The authors claim that DDOS may be used for both debugging and replication of distributed systems.

This project does not make use of distributed replay. The primary reason for this is the complexity of the SUT. Ignoring potential problems with parallelism and network communication, the generated replay files would quickly become unmanageable. The SUT also needs to be given enough running time for an OOS event to occur.

3.3 Testing a Multiplayer Lockstep Game

Weinkove describes how their game development studio tested a multiplayer lockstep game (§ 2.3) [33]. To detect desynchronisation, they chose to serialise the entire game state each frame. Together with the initial launch data structures and input messages, the full game state was continuously written to a log file (§ 4.3). That gave them the ability to start the game in arbitrary saved states and compute checksums (§ 4.2).

Their primary technique for finding synchronisation issues was to run the game simulation one step, reload the previous step, and compare the newly generated state for differences [33]. Weinkove mentions that a simpler technique would be to keep the game running in a regular and a debugging state for comparison. By including code for deducing which byte was the first to be OOS in the serialised data stream, they were able to quickly find which instance of which class was causing it. They only made assertions in the code after having reloaded the serialised game state upon detecting desynchronisation. The assertions then triggered when getting to the first desynchronised byte.

It proved to be valuable to run automated nightly simulations of the game, according to Weinkove [33]. Since they had established systems for detecting nondeterministic behaviour, they could detect problems by letting one instance of the game run overnight. They later included automated multiplayer testing by having two computers play against each other.

The method used in this project is similar to that of Weinkove et al. in that nightly multiplayer test runs of games, log files and checksums are involved. Instead of modifying the system for detecting desynchronisation, an AI is created which forcibly tries to desynchronise the game.

3.4 Desynchronisation in Open-Source Games

In this section, three examples of open-source networked games and how they detect desynchronisation are given.

3.4.1 0 A.D.

The Wildfire Games wiki page contains descriptions of some techniques for debugging their open-source networked multiplayer game *0 A.D.* [12]. Their game uses log files (§ 4.3) to keep track of the game states of each player, though they are not hashed every turn by default for performance reasons (see checksums in § 4.2). All commands which have been issued by players during the game are logged in another file.

As a starting point, Wildfire Games suggest one should compare the logged game states for differences, which might give clues on what part of the game code caused desynchronisation [12]. If the differences are plentiful, it might be good to construct a simpler test case by using a smaller map with fewer units. Another technique to test for desynchronisation is to run two instances of the game on the same machine.

The game has support for a special "serialisation test mode" which tests the simulation state every game turn [12]. When desynchronisation is detected, multiple before/after log files are created containing data from the primary game simulation and a secondary simulation which is reconstructed every turn. Another special debugging mode saves the entire game state in binary and human-readable form every game turn.

3.4.2 The Battle for Wesnoth

The open-source networked multiplayer game *The Battle for Wesnoth* has a wiki page with information on game state synchronisation [34]. As explained on the page, the game only sends client commands with the assumption that they result in the same game state for all clients (see lockstep in § 2.3). The game uses a synchronised pseudorandom number generator (§ 2.4.2) which generates the same values for all clients. This generator gets reseeded every time a synchronised user action is performed. The game differentiates between synchronised and unsynchronised player actions. Examples of synchronised actions are movement commands, while selection events are unsynchronised.

3.4.3 OpenClonk

A wiki page for the open-source networked multiplayer game *OpenClonk* contains information on synchronisation loss in the game [28]. The game is only synchronised by player commands sent over the network, and synchronisation is continuously verified by sending packets containing checksum parameters (§ 4.2). An authoritative host has the master game state which the game states of the clients are compared to.

Chapter 4

Detecting Synchronisation Problems

Some general methodology and concepts relating to detection of synchronisation problems are necessary to understand the implementation and test procedure used in this project. They are explained in this chapter.

4.1 Testing Software Using Fuzzing

Fuzzing is a proven software testing method where the SUT is given large amounts of input generated by another program in order to discover errors [24]. In their report, McNally et al. summarise the state-of-the art of fuzzing. They describe the basic structure of a *fuzzer*, which is the system used for fuzzing. The first component consists of a *fuzz generator* feeding the input to the SUT using a *delivery mechanism*. How the input is generated varies depending on application. The second component is the delivery mechanism itself, which takes the input of the fuzz generator and sends it to the SUT. The third component is the *monitoring system* used to detect any observable errors, the importance of which should not be underestimated.

4.1.1 Fuzzing Concepts

There are multiple concepts relating to fuzzers [24]. They need to have good

- *coverage*, meaning that sufficiently many code blocks should execute from the given input,
- *depth*, meaning that sufficiently many code segments which could reject invalid input in one execution path should be passed,
- *efficiency*, meaning that the fuzzer should have satisfactory code coverage at sufficient depth while keeping the computational work to a minimum.

Invalid input to the fuzzer might break the function call chain early resulting in shallow testing depth, and ideally the fuzzer should avoid repeating execution of code blocks processed earlier in order to efficiently reach the desired depth.

McNally et al. explain the difference between black-box, white-box and grey-box testing [24]. In the classical approach of black-box testing, the internals of the SUT are completely hidden and only inputs and outputs are observable. White-box testing is a more recent development, which allows the fuzzer to exploit design details and source code to enhance the fuzzing process. Grey-box testing falls in-between black- and white-box testing, but the classification is somewhat unclear.

4.1.2 Advantages and Disadvantages to Fuzzing

There are many advantages to fuzzing as a method for testing software [24]. History has shown fuzzing to be effective for finding hidden errors which have been missed by other types of testing, it allows you to test systems without having access to the source code, and fuzzing can be simple to implement by generating random data. Fuzzing may reduce the cost of testing compared to expensive manual testing, it reduces the impact of human error, and tends to produce fewer false positives. Fuzzers can, depending on the structure of the SUT, also be relatively portable between different applications.

Classical fuzzers producing purely random data may be inefficient, according to McNally et al. [24]. If the input does not conform to the expected format, it may be discarded immediately. Input consisting of checksums (§ 4.2) may be particularly troublesome to fuzz. Finally, classical fuzzers can usually not detect other types of errors than pure crashes or lock-ups.

4.1.3 Input Generation

When it comes to generating input, the two major approaches are *mutative* and *generative* generators [24]. Mutative generators take some existing set of data and mutates it to create new input. This somewhat simple approach is convenient if the input has a complex structure which may be hard to generate in other ways. One drawback is that the input becomes dependent on the original data, which may give poor general coverage. A generative fuzzer instead incorporate logic to generate completely new input data adhering to the expected format of the SUT. This typically requires more effort of the developer, but once implemented, gives high levels of coverage with excellent efficiency.

An *oblivious* fuzzer randomly modifies or generates input without any concerns regarding the expectations of the SUT [24]. A *template* based fuzzer uses a template as a starting point for the produced input. *Block* based fuzzers bundle data of different types to make it easier to create functions operating on said data. A *grammar* based fuzzer uses grammar to produce valid input to the SUT. Lastly, *heuristic* based fuzzers employ heuristics to generate input in a smarter way than doing so randomly.

4.1.4 Delivery Mechanisms

Delivery mechanisms tend to be quite simple, typically consisting of files, environment variables, invocation parameters, network transmissions, operating system events and resources [24]. McNally et. al explain how file based fuzzers effectively encapsulate data in one entity, while protocol based fuzzers are more advanced in that they must uphold valid sessions possibly consisting of multiple interactions which can depend on earlier interactions. A more exotic approach is to randomly modify the runtime configuration and memory of the SUT.

4.1.5 Monitoring Systems

The monitoring systems of fuzzers can broadly be divided into *local monitoring systems* and *remote monitoring systems* [24]. Local systems run on the same system as the SUT and typically use some sort of augmented runtime environment to detect errors. Remote systems are only able to interact with the SUT by observing input and output, like a server plugin.

4.1.6 Modern Fuzzers

Modern fuzzers have been developed to mitigate the shortcomings of classical fuzzers [24]. These modern fuzzers reject the black-box methodology of classical fuzzers in order to increase their efficiency. They can use both generative and mutative methods, or a combination of the two, in their fuzz generators. Since the source code of the SUT may be unavailable, reverse engineering tools such as debuggers or disassemblers may be used. While the delivery systems can be reminiscent of those in classical fuzzers, modern fuzzers can sometimes modify memory directly. Modern grey-box fuzzers can also have more advanced monitoring systems to analyse runtime behaviour. Symbolic execution can be used to modify values in programs, and input may be tagged to analyse execution flows. Techniques also exist to hook into library calls, and a *genetic algorithm* has been used for mutating data.

Monitoring systems of modern fuzzers can also be more advanced than the monitoring systems of classical fuzzers [24]. Test *oracles* provide the right answer for the given input and *report engines* construct human-readable bug reports. Modern fuzzers exist which use model and protocol descriptions as opposed to plain ASCII.

4.1.7 Relation to Random Testing

Random testing is a related method of testing software where coverage is achieved by generating random test cases [2]. Arcuri, Iqbal and Briand explain how oracles with assertions about the expected behaviour of the code are used as test targets. Since the method works by causing failures of assertions, it cannot by itself necessarily discover where problems in the code are located. Test cases are created from the input domain according to statistical distributions, and while a uniform distribution may remove biases, selections are normally made according to usage profiles. If

the input does not consist of numerical values, it can be non-trivial to create a distribution of test cases to sample from. Using random testing, it is possible to generate many test cases without spending significant resources, provided target coverage does not require manual verification.

Fuzzing is a type of random testing, but instead of randomly creating test cases and comparing the results with test oracles, both valid and invalid input is fed to the SUT to see how it reacts to it. Fuzzing can thus be thought of as more of a "stress test" for finding crashes, failed internal assertions or other errors. Like in fuzzing [24], *coverage* can in terms of random testing refer to the number of code branches being executed [2].

4.2 Checksum Verification

For lockstep games, checksums of game states are crucial to verify that game states at clients are consistent [23]. It is therefore reasonable to use this as a technique to detect when desynchronisation occurs. To detect games being OOS, these checksums should be sent at the end of each turn [15, ch. 6]. In essence, a checksum describes a block of data using a relatively small sequence of characters, such that the probability of two different game states having the same checksum is very low [23]. Normally, problems related to the network engine do not cause the game to crash, therefore it is important to frequently verify the game states. Checksums also help locate which part of the game code is erroneous.

4.3 Log File Analysis

Software log files are files containing events recorded by software [27]. They can be used for a number of different purposes, such as detecting errors, evaluating performance, predicting future resource needs, reporting system usage, and gathering information for business analytics. Log files may consist of unstructured data, but they are often structured using XML or other formats [18]. Structured data is preferred when automation is involved, as it allows software logic to be based on contextual information.

The procedures for analysing log files vary, a common simple method being to manually search for known keywords [27]. A more advanced yet common method is to employ machine-learning, for instance to detect anomalies in the generated data. Sophisticated forms of statistical analysis and modelling may be performed on log files using regression, Markov chains and clustering techniques. One formal technique for verifying test results is to use a state machine to analyse log files [1]. A more recent approach to log file analysis is to adopt a framework based on mind maps with a corresponding scripting language [18].

The challenges in log file analysis are plentiful: it may be hard to choose appropriate keywords for searching, log messages may be misleading, and the complex interactions between software components can obscure events recorded in

the log file [27]. Software often requires methods of internal synchronisation to produce logs, the log files themselves may be large in size, and it might be difficult to modify the log data such that it can be statistically analysed. The statistical methods can be influenced by the act of logging and may require specific domain knowledge to be applicable. There are multiple design decisions to make: how logs should be generated, what should be logged and how long it should be retained.

In this project, log files are used to output information during test runs such as OOS errors, enabled and triggered bugs, and fuzzing AI stages. This information is machine-readable and later compiled by a separate script.

4.4 Artificial Intelligence in Games

The use of artificial intelligence (AI) in video games is varied, consisting of things like non-player character behavioural learning, pathfinding, player modelling, procedural content generation and game design [36]. Research of AI in commercial games is quite separated from that in academia. It may be sufficient for an AI to work well in a particular context in a commercial game, while academic researchers tend to search for more general solutions. This project makes use of a testing AI which extends the AI already present in the SUT, and is thus highly coupled with the SUT itself. However, the method of how it is used is applicable in more general contexts by modifying any regular game AI to perform structural fuzzing.

Chapter 5

Implementing the Test System

In this chapter, the test methodology used in this project when building a system for detection of synchronisation problems in a networked lockstep game is explained. The chapter begins with an overview of the implementation and experiments, after which the functionality of the system is described in parts.

5.1 Overview

The purpose of this project is to develop a method for testing networked lockstep games for synchronisation problems. To achieve this goal, a test system has been built based on the background described in chapter 2 and 3, and methodology described in chapter 4. The test system runs the system under test (SUT) and tries to cause an out-of-synchronisation (OOS) (§ 2.1) event while generating test data. After the SUT has executed, said data is analysed in order to find the cause of desynchronisation (if the SUT went OOS). The results from performing multiple test runs is presented in chapter 6. Bugs have been artificially added to the SUT to form a guaranteed basis for testing.

5.1.1 Experimental Setup

The system used for testing primarily consists of

- the SUT itself (§ 5.2),
- a client-server connection with multiple clients (§ 5.3),
- a fuzzing AI (§ 5.4),
- checksums of the game states (§ 5.5),
- log files (§ 5.6),
- a script for running semi-automated repeated tests (§ 5.7),

- a script for parsing the logged data (§ 5.8),
- and injected synchronisation bugs (§ 5.9).

An overview is given in figure 5.1, which illustrates a system containing two instances of the SUT. Only one instance of the fuzzing AI is activated during testing, though in practice more could be instantiated. The primary reason to only keep one fuzzing AI active at a time is to avoid log file pollution. At least two SUTs need to be connected to create a valid multiplayer session. Technically one of the two SUTs also acts as the server, but that is conceptually irrelevant.

To get a sufficient amount of test data and provide enough continuous execution time to allow for desynchronisation to occur, only modifications which do not cause overly significant slowdown are made to the SUT.

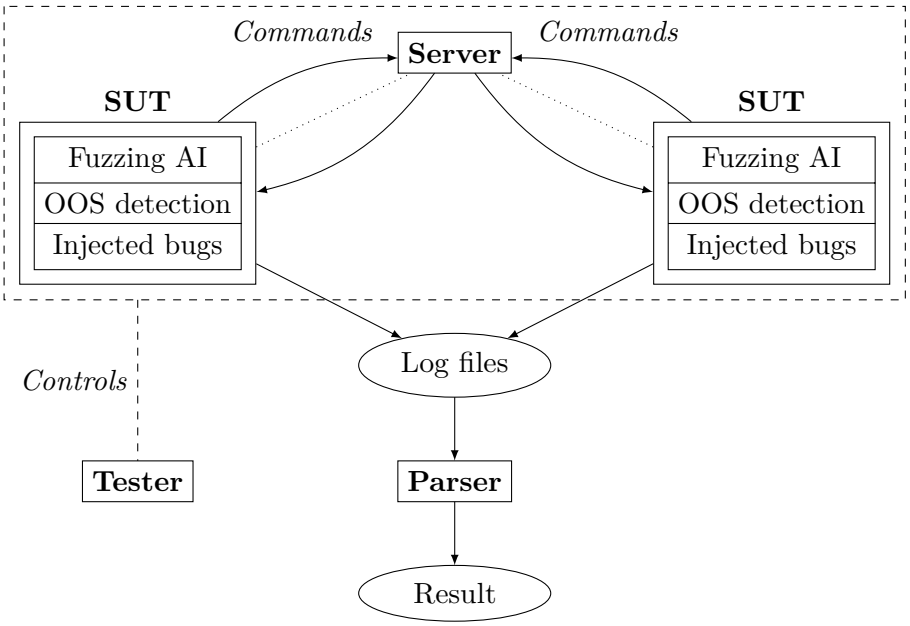

Figure 5.1. Overview of the test system.

5.1.2 Interpreting the Results

The script which parses the log files (§ 5.8) outputs the most likely causes of desynchronisation, if any desynchronisation occurred. These results are interpreted in two ways. First, an assessment is done on how likely the test system is to detect the enabled bugs, by analysing how often it manages to trigger enabled bugs and become desynchronised. Then a comparison is made to see if the proposed most likely cause is related to the command or code point which caused the OOS event. As such, the analysis requires reasoning of what code is being executed in the fuzzing AI stages (§ 5.4.2).

5.2 Description of the SUT

The system under test (SUT) is *Europa Universalis IV*, which is a networked multiplayer game using a variant of the lockstep protocol (§ 2.3), released in 2013. As a player you control a country trough a historical time period, engaging in diplomacy, trade, exploration and warfare. Each non-player-controlled country in the SUT is controlled by an AI (§ 4.4), which is referred to as a *regular* AI in this report. The game simulation advances in *ticks*, each tick representing what is presented as a "day" in the game. Commands in the SUT are processed on a "daily" basis, and as such in-game dates are used as timestamps. Which time quantum should be used depends on how the SUT is implemented; if the simulation runs in real-time, it may be more convenient to log events and change stage based on wall-clock time instead.

Though lockstep games often use a peer-to-peer model (§ 2.2.2) for communication, the SUT uses a client-server model (§ 2.2.1). While conceptually irrelevant, the server of a multiplayer session will be hosted on one of the client computers.

5.3 Connected Clients

Two clients (game instances) are running on one computer connected over a local network when performing the testing, where the server (host) is responsible for distributing collected action commands in the network to all clients. This setup is representative of a regular networked game session, with the addition of a fuzzing AI (§ 5.4). Since only one server can be used in a multiplayer session and is integrated in the SUT, no testing is done with clients connected to multiple servers. This means that synchronisation problems relating to interaction between multiple servers are not considered.

The network architecture and networking code of the SUT are assumed to be working correctly. Packets are assumed to arrive on-time in the correct order without significant latency (§ 2.4.1) at the clients. These assumptions are made to limit the scope of the project and make the task more feasible. This project instead focuses on OOS problems stemming from how commands are processed once they arrive. While OOS events caused by the network architecture or networking code may still be detected, their underlying causes are abstracted by the assumptions made.

5.4 Fuzzing AI

At the heart of the system used for testing lies the fuzzing AI (§ 4.4). It extends the already present AI classes in the game, but it does not strive to make player-like decisions. Instead, its goal is to generate as much valid and invalid input commands as possible in order to trigger any potential OOS bug (§ 4.1). Fuzzing can be thought of as a "stress test" to verify that the SUT does not crash from bad input, or as in this case, goes OOS. Since there is no clear way to test applications for

hardware-related synchronisation problems using static analysis, this highly dynamic method was chosen.

5.4.1 Input Generation

The input generated by the AI is sent to the game server using the source code functions present in the game. By being integrated in the game itself, the fuzzing AI is able to send commands directly as opposed to generate input externally as a player would. This gives larger control of what is being sent, and allows sending input which would not be possible through the graphical user interface (GUI) of the game. In addition, the fuzzing AI can interface with the game's internals to make decisions based on game information not presented to the player. This approach is somewhat unusual, since typically fuzzing is done by a separate program using a delivery mechanism to feed input to the SUT.

5.4.2 Fuzzing Stages

The fuzzing AI works in *stages*, where a single type of command is given in each stage. It is designed as a large map between stage numbers and functions implementing the stages. At a given tick interval (§ 5.2), the stage counter is increased and the next mapped stage executed. Stages can either execute as long as the fuzzing AI runs, or they can be limited to only execute a maximum number of times.

The stages represent possible causes for desynchronisation in the code, and their granularity thus directly influence the precision in the analysis. By knowing what stage the AI was in during an OOS event, you get an indication of which part of the codebase caused the issue. Of course, there is also a possibility that the OOS event was caused by a completely unrelated action.

5.4.3 Fuzzing Performance

To get proper code *coverage*, the fuzzing AI should generate as much input of different types as possible. This increases the likelihood of triggering an action which leads to the game going OOS. In total 200 stages have been added to the fuzzing AI for reasonable coverage. An important caveat is that the SUT uses validation of the sent commands, meaning the vast majority of given (invalid) input is immediately discarded. In that sense the fuzzing AI is *inefficient*, and there is a trade-off between crafting stages with sufficient *depth* and increasing the number of stages for better coverage.

5.4.4 Remarks on the Fuzzing AI

Since there is already AI functionality present in the SUT, it is reasonable to ask why the fuzzing AI was created in the first place. Due to the fact that the regular AI (§ 5.2) strictly follows the rules of the game, you miss potential problems caused by invalid actions being sent. The regular AI is also meant to imitate a human player,

so its input is restricted to single actions being performed at suitable times. This probably holds true for AI in games in general; if the virtual players controlled by the AIs of the game are left to their own devices, then there is no telling of what is currently being tested. Since OOS problems may be caused by nondeterministic behaviour (§ 2.4.2), actions should be performed multiple times to increase the probability of one of these actions triggering an OOS bug.

When performing the testing, it might be desirable to run two instances (or more) of the fuzzing AI to effectively double the amount of commands being issued. The SUT has been modified to allow an almost arbitrary number of instances of the AI to be run during a test session. However, having two fuzzing AIs active simultaneously also means that the amount of logged information will be doubled, making the log files harder to interpret.

5.5 Checksums

Multiple checksums (§ 4.2) and error messages are used for detecting when the SUT goes OOS. The system for generating said checksums and error messages was already implemented in the game logic, and the checksums consist of a subset of the game state which has been deemed representative for its current state of synchronisation. Since not the entire game state is included in the checksums, some level of desynchronisation may go undetected. An alternative approach is to generate checksums from entire blocks of memory, which might give a better representation of the full game state.

OOS events are detected by continuously comparing the checksums between all participating players. This functionality was already implemented in the game logic. If any player has a differing checksum, an error message is logged and the game is stopped. From the error messages it is possible to deduce which of the multiple checksums are different. This in turn gives an indication of which part of the codebase may contain the synchronisation bug. In general, by increasing the number of checksums and decreasing the number of things included in them, it is possible to get a more fine-grained indication of what caused the OOS event to occur.

An alternative method to using checksums is to continuously and frequently save the entire game state to disk for analysis as was done in [33], but that would likely have caused significant performance overhead in the SUT and was therefore discarded in favour of the already present checksum system.

5.6 Log Files

Log files (§ 4.3) are used to record data produced by the SUT during execution. The built-in logging system has been modified to produce logs in the CSV format for easier parsing (§ 5.8). An information log file is used by the SUT to output changes of fuzzing AI stage (§ 5.4.2), enabled and triggered bugs (§ 5.9), OOS error

codes with inconsistent checksums (§ 5.5), and general debugging information. All commands sent by all AI players, both regular AIs (§ 5.2) and the fuzzing AI, are logged in a separate command log file. A typical log file CSV row looks like

⟨STAGE,<stage number>,<timestamp>,<description>⟩.

5.7 Test Automation

In order to gather sufficient test data, a script is used to perform semi-automated repeated test runs of the SUT. It starts two instances of the game, one acting as the host and one as a client. The script then uses GUI test automation (§ 3.1) to initiate a multiplayer match by clicking on the relevant buttons in the game's interface. Since GUI testing is generally inefficient and dependent on the current layout of the GUI, the ambition has been to utilise this method as little as possible.

Once the game has started, the fuzzing AI (§ 5.4) is activated and the game state set up using already present functionality for loading in-game console commands from a file. During each test run, one bug (§ 5.9) is activated in the SUT. Only one bug is ever active during each test run to ensure that the real cause of any detected desynchronisation is known with high probability, which is later used to evaluate the test system's ability to find synchronisation problems.

When the SUT has been running for a certain amount of time, the host and client processes are terminated by the script, after which the accumulated log files (§ 5.6) are copied to a storage location for later analysis by the log file parser script (§ 5.8). The automation script then repeats the process by starting up two new instances of the SUT with another bug, and continues as such for a finite number of test runs.

5.8 Log File Parsing

After multiple clients have played some number of sessions against each other, a script is used to analyse the generated log files (§ 5.6). This subsection describes the functionality of this script.

5.8.1 Sent AI Commands

First the information log file is searched line-wise to find all recorded OOS events. Once they are found, the command log file is offset to some row containing the same date as the recorded OOS event. The command log file is large in size, but since the rows are sorted by timestamp it is possible to do a binary search to quickly find one row with a matching timestamp. Since an OOS event is likely to have been the result of a command issued with an earlier timestamp, an offset date is used as the target date to search for. After the date is found, all commands up to the point of the OOS event are extracted. The most frequent commands sent by the AI players are output by the script, which gives a slight hint on what caused the SUT to go

OOS. The gathered command information is highly generic, and is therefore not used to score likely causes of desynchronisation.

5.8.2 Enabled and Triggered Bugs

Each time an OOS bug is manually triggered, the current timestamp is logged and later found by the script. The SUT also logs when the intentionally faulty code is executed. Using the logged data, the script outputs how long it took before an enabled bug was triggered, and how long it took the SUT to go OOS after having executed the faulty code.

5.8.3 Finding the Likely Cause

The stages of the fuzzing AI (§ 5.4.2) represent possible causes for desynchronisation in the code. Checksums (§ 5.5) have been manually associated with the stages to which they have been deemed relevant (for instance, stages which are believed to modify player money in the game are associated with the money checksum).

The script searches the information log file for which nearby stages the fuzzing AI has been in from an offset up to the detected OOS event. Each possible stage i is then given a score S_i of how likely it caused the OOS event, calculated according to formula 5.1. The number of inconsistent checksums for stage i is denoted n_i, N is the total number of checksums, j is the 1-based integer index in the list of stages near the OOS event (counting from the stage furthest away), M is the total number of stages, and m is the number of stages near the OOS event. If stage i is not selected as being near before the OOS event, the second parenthesis in the formula is not added.

$$S_i = \left(n_i \cdot \frac{N}{10}\right) + \left(j \cdot \frac{M \cdot 10}{m}\right) \tag{5.1}$$

The idea is to assign a higher score to stages corresponding to a larger number of inconsistent checksums, with a higher contribution to the score if the stages was near before the OOS event. The weight added for inconsistent checksums is higher if there are more checksums in total, since more checksums gives a more fine-grained idea of where the desynchronisation occurred. The score for each nearby stage is higher if there are fewer stages close to the OOS event and more stages in total. This is because a smaller number of close stages means that each such stage more likely caused the OOS event, and more stages in total means that the individual stages should be more representative of the OOS cause. Constant weights of value 10 are added to make nearby stages contribute even more to the score than inconsistent checksums, though a higher or lower value could be used to further increase or decrease this contribution.

Finally, the k fuzzing AI stages which have the highest score are selected and output by the script with a confidence based on how large percentage of the total awarded score they were given. Note that the scoring function has not been signific-

antly calibrated based on test data since the correlation between AI stage and time of desynchronisation turned out to be insignificant.

Example output of the parser script is found in appendix A.

5.9 Injected Synchronisation Bugs

The SUT may be free of synchronisation bugs, and even if it is not, nondeterministic bugs are unlikely to manifest if the game instances run on the same computer as the execution environments are identical. For this reason, artificial bugs have been introduced to form a guaranteed foundation for testing. These bugs can be enabled or disabled during execution of the SUT. They are inserted at different code points, and may be set to trigger after a minimum amount of game time and/or with a certain probability. In total 39 artificial bug points have been added to the codebase.

The following methods of introducing bugs are used:

B_1: Modifying a numerical value at one client, representing a potential error caused by floating point values being rounded differently on different hardware.

B_2: Calling a non-synchronised PRNG.

B_3: Sorting an array using an unstable sorting algorithm.

B_4: Inverting the value of a Boolean variable, imitating it being uninitialised.

B_5: Initialising a variable with a random value, imitating it being uninitialised.

B_6: Reproducing a real historical bug which has at some point been present in the codebase of the SUT and caused desynchronisation.

Bug type B_1 represents the case when nondeterminism caused by hardware optimisations is not accounted for. Bug types B_2 and B_3 represent the situation when the functionality of algorithms are misunderstood. Bug types B_4 and B_5 represent code errors leading to undefined behaviour. Bugs of type B_6 are of varying nature, like misuse of cached or client-local values.

Bugs of of types B_1 to B_5 have been chosen because they are simple to implement and are plausible to occur during development. More delicate bugs like those caused by indeterminate evaluation order of expressions could also be added, but imitating this behaviour in code would either require finding function calls where side-effects are different based on order of execution, or require making larger changes to the original code of the SUT. The ambition has been to make as small changes to the SUT as possible when introducing bugs, to avoid interfering with the original functionality. The reason for adding bugs of type B_6 is self-explanatory, though they should be considered a bonus and are not part of the primary analysis performed in this project.

Chapter 6

Results

This chapter contains the results of this project. The first section is an explanation of the results, and in the next section the results are presented.

6.1 Explanation

In this section, it is explained how the experiments were performed and how the test data should be read. The term *regular AIs* refers to the AIs administering the non-player-controlled entities in the SUT (§ 5.2) which normally are enabled.

6.1.1 Common Test Properties

In all tests, a number of consecutive test runs were performed with one bug point enabled in each run. After a given duration, the current test run was aborted and the next test run started. In test **A**, **B**, **C**, **D** and **E**, each test run lasted 60 min, while in test **F** they lasted 30 min. The fuzzing AI had a total of 200 stages, some of which were modified between tests. The stage interval was set to 30 ticks. During log file parsing, an offset of 160 ticks was used. Test **A**, **B**, **C**, **D** and **E** were all performed with the same build (version) of the game, while test **F** was performed with a more recent build.

6.1.2 Injected Bugs

The following list contains information about which bug types were possible to trigger and how bugs were activated in all tests:

- In test **A**, a total of 33 different bugs could be enabled,
  - 12 of type B_1,
  - 5 of type B_2,
  - 3 of type B_3,
  - 8 of type B_4 and

  - 5 of type B_5.

  In each test run of test **A**, a random bug point in the code was enabled, and bugs of type B_4 were triggered with 50 % probability. This randomness was initially added to make the testing more realistic, but it was removed in later tests (all tests excluding **A**) to make the comparison more fair.

- In test **B** and **C**, a total of 33 different bugs could be enabled,

  - 10 of type B_1,
  - 5 of type B_2,
  - 3 of type B_3,
  - 8 of type B_4 and
  - 7 of type B_5.

  Note that some of these bug points were different from those used in test **A**.

- In test **D**, a total of 6 real historical bugs could be enabled (type B_6), which have all at some point been present in the SUT.

- In test **E**, all bugs used in test **B**, **C** and **D** could be enabled, for a total of 39 possible bugs.

- In test **F**, all bugs used in test **E** could be enabled, though one bug of type B_5 was changed.

- In test **B**, **C**, **D**, **E** and **F**, one bug was enabled in each test run, starting from the first bug and counting up until the last bug was reached and the counting started over.

6.1.3 Enabled, Disabled, Passive and Indifferent AIs

The following list contains information about the state of the AIs and visited stages in the different tests:

- In test **A**, only the fuzzing AI was enabled. The fuzzing AI had time to enter all stages at least once before the next test run started.

- In test **B**, all regular AIs including the fuzzing AI were enabled. The fuzzing AI rarely managed to enter all stages during the test runs.

- In test **C**, all regular AIs were enabled while the fuzzing AI was disabled.

- In test **D**, the fuzzing AI and all regular AIs were enabled. The fuzzing AI rarely managed to enter all stages during the test runs.

- In test **E**, the fuzzing AI was enabled and all regular AIs were *passive*, meaning they responded to received commands but sent no commands of their own. The fuzzing AI had time to enter all stages at least once before the next test run started.
- In test **F**, the fuzzing AI was enabled and all regular AIs were passive but responding "positively" to received commands (during normal play, the regular AIs are likely to decline received requests or proposals). For brevity, passive and positively responding AIs are going to be referred to as *indifferent*, since they accept everything. The fuzzing AI had time to enter all stages at least once before the next test run started.

6.1.4 Test Runs

Table 6.1 lists in column order:

1. The test.
2. The number of completed test runs.
3. The number of failed test runs where some problem caused it to not initiate or finish prematurely.
4. The number of test runs where a desynchronisation occurred without any bug being triggered. Those OOS events are excluded from the results, as their causes are highly uncertain.

Note that test **C** consisted of three combined test runs, while all others were cohesive.

Test	Completed	Failed	Uncertain
A	106	13	4
B	62	0	5
C	45	0	5
D	62	0	15
E	62	0	9
F	160	7	4

Table 6.1. Information about the total number of test runs, number of failed runs and number of runs with OOS events without triggered bugs in all tests.

6.1.5 Data Table Descriptions

Table 6.2, 6.5, 6.8, 6.11, 6.14 and 6.17 list in column order:

1. Bug type (§ 5.9).

2. The number of times the bug was enabled.
3. The number of test runs the bug was triggered in.
4. The number of times the bug led to detected desynchronisation.
5. The number of times the (uniquely) most likely presented cause of desynchronisation corresponded to the enabled bug. Here, "correct" entails a direct correspondence. For example, a bug in the code for "sending gifts" is not considered correctly identified if the presented cause is "improve relation", even if sending gifts is a way of improving relations in-game. If the presented cause in the example would have been "send gift" or similar, then it would have been considered correct.

Table 6.3, 6.6, 6.9, 6.12, 6.15 and 6.18 list in column order:

1. Bug type.
2. The percentage of all enabled bugs of that type which was triggered.
3. The percentage of all triggered bugs that led to desynchronisation.
4. The percentage of all detected desynchronisations that had the correct cause presented.

Table 6.4, 6.7, 6.10, 6.13, 6.16 and 6.19 list in column order:

1. Bug type.
2. The number of ticks (§ 5.2) it took on average until the bug was triggered (T_t).
3. The number of ticks it took on average until desynchronisation was detected for the triggered bug (T_d).

6.2 Data

This section contains the numerical test results.

6.2.1 Test A

Table 6.2, 6.3 and 6.4 summarise the obtained results with the fuzzing AI enabled and all regular AI disabled.

Bug Type	**Enabled**	**Triggered**	**Detected**	**Correct**
B_1	35	16	0	0
B_2	18	6	3	3
B_3	11	2	0	0
B_4	16	6	0	0
B_5	13	7	0	0

Table 6.2. Test results from test **A**.

Bug Type	**Triggered** (%)	**Detected** (%)	**Correct** (%)
B_1	45.7	0	0
B_2	33.3	50	100
B_3	18.2	0	0
B_4	37.5	0	0
B_5	53.8	0	0
Avg.	37.7	10	20

Table 6.3. Percentages of triggered, detected and correctly identified bugs based on table 6.2.

Bug Type	T_t	T_d
B_1	206	0
B_2	1157	2
B_3	1	0
B_4	414	0
B_5	2248	0
Avg.	805	0

Table 6.4. Average number of ticks until the bugs were triggered (T_t) and detected (T_d).

6.2.2 Test B

Table 6.5, 6.6 and 6.7 summarise the obtained results with the fuzzing AI enabled and all regular AIs enabled.

Bug Type	Enabled	Triggered	Detected	Correct
B_1	20	12	10	0
B_2	9	7	5	2
B_3	3	2	1	0
B_4	16	12	7	0
B_5	14	8	6	0

Table 6.5. Test results from test **B**.

Bug Type	Triggered (%)	Detected (%)	Correct (%)
B_1	60	83.3	0
B_2	77.8	71.4	40
B_3	66.7	50	0
B_4	75	58.3	0
B_5	57.1	75	0
Avg.	67.3	67.6	8

Table 6.6. Percentages of triggered, detected and correctly identified bugs based on table 6.5.

Bug Type	T_t	T_d
B_1	374	1332
B_2	29	364
B_3	1	1718
B_4	899	360
B_5	662	57
Avg.	393	766

Table 6.7. Average number of ticks until the bugs were triggered (T_t) and detected (T_d).

6.2.3 Test C

Table 6.8, 6.9 and 6.10 summarise the obtained results with the fuzzing AI disabled and all regular AIs enabled, to provide reference data.

Bug Type	Enabled	Triggered	Detected	Correct
B_1	10	7	5	0
B_2	10	5	1	0
B_3	9	6	1	0
B_4	9	6	4	0
B_5	7	3	3	0

Table 6.8. Test results from test **C**.

Bug Type	Triggered (%)	Detected (%)	Correct (%)
B_1	70	71.4	0
B_2	50	20	0
B_3	66.7	16.7	0
B_4	66.7	66.7	0
B_5	42.9	100	0
Avg.	59.2	55	0

Table 6.9. Percentages of triggered, detected and correctly identified bugs based on table 6.8.

Bug Type	T_t	T_d
B_1	308	1929
B_2	7	2
B_3	1	734
B_4	260	303
B_5	49	56
Avg.	125	605

Table 6.10. Average number of ticks until the bugs were triggered (T_t) and detected (T_d).

6.2.4 Test D

Table 6.11, 6.12 and 6.13 summarise the obtained results with the fuzzing AI enabled and all regular AIs enabled when testing on real historical bugs in the SUT.

Bug Type	**Enabled**	**Triggered**	**Detected**	**Correct**
B_6	62	32	17	0

Table 6.11. Test results from test **D**.

Bug Type	**Triggered** (%)	**Detected** (%)	**Correct** (%)
B_6	51.6	53.1	0

Table 6.12. Percentages of triggered, detected and correctly identified bugs based on table 6.11.

Bug Type	T_t	T_d
B_6	1	477

Table 6.13. Average number of ticks until the bugs were triggered (T_t) and detected (T_d).

6.2.5 Test E

Table 6.14, 6.15 and 6.16 summarise the obtained results with the fuzzing AI enabled and all regular AIs passive (§ 6.1.3).

Bug Type	Enabled	Triggered	Detected	Correct
B_1	20	8	4	0
B_4	5	2	1	1
B_5	3	1	0	0
B_2	16	8	4	0
B_3	12	6	3	0
B_6	6	3	1	0

Table 6.14. Test results from test **E**.

Bug Type	Triggered (%)	Detected (%)	Correct (%)
B_1	40	50	0
B_4	40	50	100
B_5	33.3	0	0
B_2	50	50	0
B_3	50	50	0
B_6	50	33.3	0
Avg.	43.9	38.9	16.7

Table 6.15. Percentages of triggered, detected and correctly identified bugs based on table 6.14.

Bug Type	T_t	T_d
B_1	284	3162
B_4	1157	2
B_5	1	0
B_2	1524	1049
B_3	2752	1654
B_6	22	51
Avg.	957	986

Table 6.16. Average number of ticks until the bugs were triggered (T_t) and detected (T_d).

6.2.6 Test F

Table 6.17, 6.18 and 6.19 summarise the obtained results with the fuzzing AI enabled and all regular AIs indifferent (§ 6.1.3).

Bug Type	Enabled	Triggered	Detected	Correct
B_1	46	23	3	0
B_4	20	8	4	4
B_5	12	4	0	0
B_2	31	15	4	0
B_3	27	14	5	0
B_6	24	12	5	0

Table 6.17. Test results from test **F**.

Bug Type	Triggered (%)	Detected (%)	Correct (%)
B_1	50	13	0
B_4	40	50	100
B_5	33.3	0	0
B_2	48.4	26.7	0
B_3	51.9	35.7	0
B_6	50	41.7	0
Avg.	45.6	27.8	16.7

Table 6.18. Percentages of triggered, detected and correctly identified bugs based on table 6.17.

Bug Type	T_t	T_d
B_1	428	425
B_4	1157	2
B_5	1	0
B_2	1471	1827
B_3	1846	12
B_6	5	570
Avg.	818	473

Table 6.19. Average number of ticks until the bugs were triggered (T_t) and detected (T_d).

6.2.7 Visual Comparison of Test A, B, C, E and F

Figure 6.1 shows a visual comparison of the data in table 6.3, 6.6, 6.9, 6.15 and 6.18.

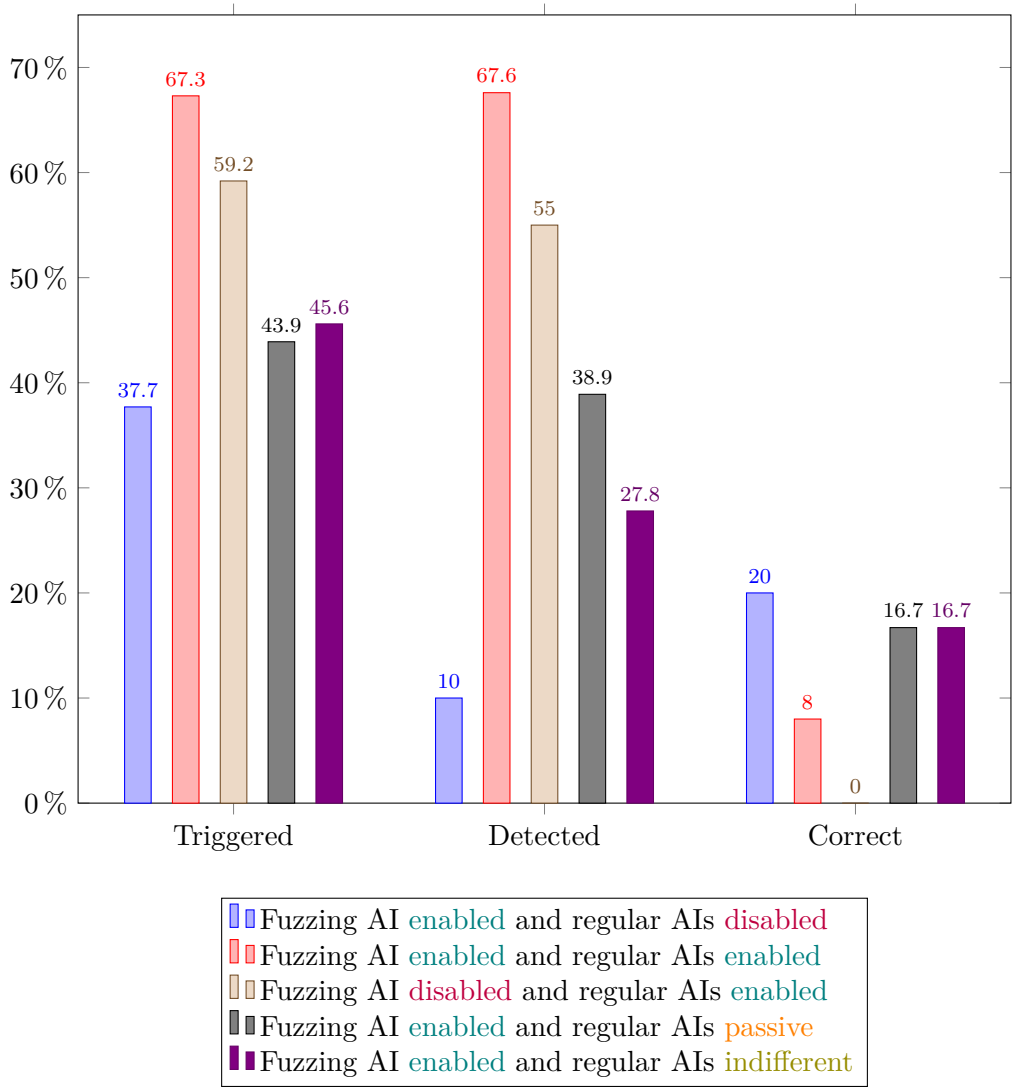

Figure 6.1. Comparison of average percentages in table 6.3, 6.6, 6.9, 6.15 and 6.18.

Chapter 7

Discussion and Conclusion

This chapter contains the analysis of the results and the conclusion drawn from it. It is discussed how accurate the method of choice and results are, and how feasible the method is for practical use. It is brought up how the method could be improved, and aspects surrounding the project are assessed. Some pointers to potential future work are also given.

7.1 Discussion

This section contains the discussion of the results and the methodology used to obtain them.

7.1.1 Obstacles During Testing

Running semi-automated tests turned out to be more complicated than expected. Since the test system may not be interfered with during testing, and because the test system needs a lot of time to run through all the stages of the fuzzing AI, the test runs were made during nights or free days. It was generally hard to get the fuzzing AI to run through all stages in the set test time, and increasing the time would generate less test data. As a result of the testing being hands-off, it was also not possible to correct problems during test runs, and bugs in the implementation were not discovered until after tests were made. Consequently, the iteration time during development was inconveniently long.

GUI testing was unreliable, and time was wasted on client-server sessions failing to be initiated. The SUT crashed on occasions due to real bugs, and at times the fuzzing AI would itself erroneously cause the SUT to always go OOS. Therefore much generated test data was discarded. Sometimes the SUT became desynchronised without bugs being triggered, and multiplayer sessions ended for unknown reasons.

7.1.2 Analysis of the Results and Methodology

In this subsection, the results and the methodology in the context of the results are analysed. The discussion is broken down into sections for readability.

Triggered Bugs and Desynchronisation

The results in table 6.3 and 6.6 show that around 40 or 70 percent of all active bugs on average were at some point triggered, which indicates that the fuzzing AI has potential for reaching sufficient fuzzing depth. If the fuzzing AI would have entered all stages in the second test run, it is reasonable to expect that even more bugs would have been triggered. Furthermore, one can see that essentially no triggered bugs caused the AI to go OOS when only the fuzzing AI was active, while almost 70 percent of all triggered bugs led to desynchronisation when all AIs were active. The primary reason for this seemed to be that the regular AIs of the game needed to be active in order to respond to the actions performed by the fuzzing AI.

Even though the results were clearly better with all regular AIs of the game active alongside the fuzzing AI, all triggered bugs still did not cause desynchronisation. The immediate plausible reason for this is that the injected bugs did not modify the game state enough for it to become inconsistent. One can also consider the coverage of the checksums in the SUT. These checksums only cover a subset of the game state, meaning it is possible that desynchronised game states go undetected. The system for detecting desynchronisation in the SUT has not been modified in this project, as the focus rather has been on generating OOS events and preferably finding their cause.

Tests were made with the fuzzing AI disabled and all regular AIs enabled to see if the fuzzing AI contributed to causing desynchronisation. The data in table 6.9 shows that the results are only slightly worse in terms of triggering bugs and causing desynchronisation. It is possible that the time spent implementing the fuzzing AI is not worth this small increase in performance.

Out of interest, tests were also performed with real historical bugs which at some point in time have been present in the SUT. The data in table 6.12 displays that around 50 percent of all bugs points were triggered, and that around 50 percent of all triggered bugs caused desynchronisation. In other words, if the test system would have been used during development, these bugs could potentially have been detected.

As a possible improvement, the regular AIs were modified to be passive but respond to received commands. The results in table 6.15 show that it did not significantly increase the average number of triggered bugs compared to test **A**, but it did make around 40 percent on average of triggered bugs cause desynchronisation. In theory, this change would mean that bugs are no longer triggered at arbitrary points during execution due to interference by the regular AIs. Unfortunately, it did not improve the precision in finding the causes of desynchronisation, as the average times until bugs triggered were still long.

Finally, a test was made with the regular AIs indifferent, i.e. passive and always responding positively to received commands. As seen in table 6.18, it did not result in a significant difference compared to test **E**. This may be because at least one regular AI would always respond positively even without this change.

Precision of the Test System

Looking at the results in table 6.6 for instance, one can see that the precision of the test system is low. There are multiple reasons for this. As seen in table 6.7, the average times until triggered bugs caused desynchronisation were long. This is partially due to extreme values, but it also indicates a problem in the methodology. The test system requires two key things in order to properly deduce the cause of desynchronisation. First, the bug must be triggered in the related stage of the fuzzing AI (assuming such a stage exists). Naturally, bugs can be triggered at arbitrary points during testing, especially if all AIs are active and interfering with the fuzzing AI. Secondly, the desynchronisation must be detected before the next stage of the fuzzing AI is entered. The stage duration was kept at 30 ticks to keep each test duration reasonable, which is much lower than the average desynchronisation time after a bug being triggered of 766 ticks in table 6.7.

Fuzzing Coverage, Depth and Efficiency

The fuzzing AI far from covers the entire codebase, which is one significant flaw in the testing methodology. The coverage has been made as large as the time frame of this project allowed, but more stages could easily be added to further improve the likelihood of the fuzzing AI triggering an OOS bug. If the coverage would be better, then more triggered bugs would likely have been present in the results.

Even though the fuzzing AI is capable of triggering bugs and causing desynchronisation, it generally lacks depth and is very inefficient. This is arguably the largest flaw in the testing methodology. Many commands sent by the AI get discarded immediately for being invalid during a sanity check, which creates a barrier for what code is being tested by the AI. Achieving sufficient fuzzing depth is, contrary to achieving sufficient coverage, relatively hard. It takes a good understanding of the SUT to make sure that the generated commands are valid, and in many cases they are impossible to make valid in the current game state. During testing, a trade-off has been made between expanding the code coverage and increasing the testing depth, where the former has been prioritised over the latter. The rational for this has been that testing larger parts of the codebase is more important than testing fewer parts of the code more thoroughly.

Concerns Regarding AI Stages

When adding stages to the fuzzing AI, one should prioritise testing parts of the code which are frequently executed and thus likely to cause problems if they contain bugs. Ideally, these stages should not influence other stages. That has not been

the case here since each stage potentially modifies the game state, which can affect remaining stages. It is possible that some stages never got to achieve the intended fuzzing depth due to them being blocked by effects from earlier stages.

Another aspect which directly influences the result is the granularity of the stages of the fuzzing AI. If each individual stage tests a smaller part of the codebase, one gets a better idea of what caused the SUT to become desynchronised. Creating a large number of smaller stages will naturally add to the development time of the fuzzing AI, so one must make a decision of what precision is required.

Randomness in Testing

There is an amount of randomness to consider in the testing, since a random bug was enabled in each test run of test **A** and bugs of type B_4 were also triggered with a random probability. This randomness was removed in the later tests, and some bug points and stages were changed, which may have made the SUT more prone to desynchronisation. In some tests the SUT became desynchronised in spite of no bug being triggered, which may be because of real bugs, though it sometimes seemed to be caused by the multiplayer session not being properly initiated. The actual causes of said desynchronisations were never found, but it is still an important source of error to acknowledge. Test **F** was performed on a more recent build of the game with the hope that it would lack real synchronisation bugs, which in itself is a source of error.

Human Error

When it comes to log file analysis, checksums have been manually associated with the stages they have been deemed relevant for to improve the precision of the test system. This means that human error might skew the process of finding the cause of an OOS event, and partly because of this reason checksums are weighted less than nearby stages of the fuzzing AI when computing the probable cause of desynchronisation. By introducing a mapping between checksums and functions in the codebase one could lessen this problem somewhat, but doing so would require making major changes to the SUT.

Bias in Results

Since bugs have been manually added to the SUT, there is an inherent bias in the results presented in this report. The chosen locations for bugs have largely been decided by bug type, such that bugs have been inserted where they might occur judging from the codebase. The advantage of manually adding bugs is that it is possible to deduce what caused an OOS event for comparison. An alternative testing approach is to have an external person adding the bugs to the SUT without revealing where they are located. Doing so might make the testing more realistic, but it would constrain the possibilities for analysis. One bug of type B_2 caused the SUT

to immediately detect desynchronisation upon being triggered, and it unfortunately turned out the be the only correctly identified bug.

Confidence in Results

Due to the many acknowledged sources of error in the results and general unpredictability associated with dynamic testing, the obtained results are considered to be fairly imprecise. To counteract this problem, the analysis has been made correspondingly loose, primarily looking at average values while reasoning in general terms. As such, the results should only be taken as indicative of the methods viability. When comparing these results with the results from other studies on the topic of detecting synchronisation problems in networked games, one must take this disclaimer into consideration.

Summary of the Method's Feasibility

To summarise, the test data condensed in figure 6.1 indicates that the test system can be used to find if a given build of the SUT suffers from *some* synchronisation problems, but probably not to find out *what* those problems are. The fuzzing AI can be employed to augment the regular AIs of the SUT, but not act in solitude. It is left to the reader to decide if it is worth implementing the fuzzing AI considering its performance over only using the regular AIs of the SUT.

7.1.3 Implementational Concerns

The results notwithstanding, it is recommended that the fuzzing AI is developed in conjunction with the game itself for this type of test system to be useful. As it stands, the test system needs a significant amount of development time to become effective, which can make it impractical to implement. A better idea might be to add a stage to the fuzzing AI once the corresponding functionality has been implemented in the SUT during development; that way the test system grows organically with the game itself.

Not only is much development time needed, the test runs themselves are also time consuming. Therefore it is important that the fuzzing AI does not make the game simulation run too slow to gather sufficient test data. If the number of stages of the AI becomes too large, or if individual stages are too computationally expensive, the system might not have time to trigger an OOS event. This trade-off between code coverage and execution time is ingrained in the testing methodology.

7.2 Improvements to the Test System

The fuzzing coverage and depth of the test system are always candidates for improvement. Increasing the coverage is straightforward; it is only a matter of adding more stages to the fuzzing AI. As explained previously, that will also increase the required

test duration, but in practice it would hardly be a problem since the test system can be left running over night. During this study, it has been necessary to run the test system multiple times in succession to generate test data, but in reality you would only need to run the system once per build. Running the system once also increases the likelihood of finding the correct cause of desynchronisation, since the stage duration can be considerably longer. The fuzzing depth is harder to increase, but with more knowledge of the inner workings of the SUT it is certainly doable. By more carefully crafting the individual stages of the fuzzing AI, it should be possible to achieve greater fuzzing depth.

Another possible improvement is to perform fuzzing AI stages multiple times to iteratively increase the likelihood of causing desynchronisation. More invalid commands can be sent to ensure that they are properly handled by the SUT. One interesting approach which has not been tested is to "cheat" by amplifying the capabilities of the fuzzing AI to increase coverage. For instance, it could be given more resources or authority, which in turn should make it possible to issue a larger number of valid commands. Multiple test runs could also be made with the fuzzing AI controlling different entities in the game, to make even more commands valid.

In terms of analysis, the logging and log parsing systems could be extended to include more information of recorded events. More checksums could be added to the SUT for better and finer coverage of the game state. One advanced and presumably infeasible modification is to somehow annotate commands in the code with the checksums they modify. If it can be done programmatically, it would remove the human error involved in manually associating stages with checksums.

Finally, the GUI testing involved could benefit from an overhaul to make it more robust. It is possible that other APIs than those used are less prone to failure. An even better solution is to remove parts of the required GUI testing by adding proper support for initiating multiplayer sessions via the command line.

7.3 Ethics, Sustainable Development and Societal Aspects

Today video games are ubiquitous in our society, and the art form still continues to increase in popularity. But due to the complexity and scope of video game development, it is not uncommon that projects get delayed and budgets exceeded. If an implemented fuzzing AI were to reach sufficient coverage, depth and efficiency, the test methodology could be used to lessen the work load for game developers. A semi-automated test system should decrease the amount of time spent on manual testing, time which could be better spent on developing the functionality of the game or sleeping. With less time spent, less money and resources are also spent. Therefore there is an ever so slight contribution to making game development more sustainable, and more compelling as a career choice in society.

7.4 Future Work

This project has focused on desynchronisation caused by nondeterministic behaviour. A natural topic for further research would be how one can test lockstep games for desynchronisation caused by problems in the networking code and network architecture. One could also investigate techniques for detecting synchronisation problems in other types of games than lockstep games, where some amount of inconsistency may be tolerated. This project uses a fuzzing AI and log file analysis, but it would be interesting to see if there are other more suitable dynamic methods for finding synchronisation problems which might have been overlooked here.

By the reasons given in the introduction, a theoretical approach to the problem was decided against. One can still ponder how *model checking* could be used to tackle the problem of detecting desynchronisation in lockstep games. You would need to specify deterministic simulation of the tested game, as well as figure out how to handle the combinatorial blow-up of the game state space. It would be necessary to mathematically capture the notion of nondeterminism caused by hardware aspects. One solution might be to limit the analysis to finding certain causes for nondeterminism, like uninitialised variables. If networking aspects should also be considered, one could construct a probabilistic model where the unreliability of network communication is accounted for.

Generally speaking, you would probably need high levels of abstraction and weakening of desired properties in order to verify even small subsets of the codebase, considering how sophisticated video games are as systems. Even if it would be possible to create a model for finding synchronisation problems, it may be too weak to be useful in practice. Still, if a theoretical treatment of the problem could be defined, it would make for a highly interesting future research topic.

7.5 Conclusion

The purpose of this project has been to investigate and evaluate how one can test networked lockstep games for synchronisation problems, using analysis of data produced by simulated players. To that end, a semi-automated test system comprised of a fuzzing AI and log file analysis has been developed based on the research done. The results show that the test system has potential for finding synchronisation problems if the fuzzing AI is used in conjunction with the regular AIs in the tested game, but not for finding the origins of said problems. The inclusion of the fuzzing AI makes the test system marginally more likely to cause desynchronisation than if only the regular AIs are enabled. By also modifying the regular AIs, it is possible to desynchronise the SUT without them interfering.

Bibliography

[1] James H Andrews and Yingjun Zhang. 'General Test Result Checking With Log File Analysis'. In: *IEEE Transactions on Software Engineering* 29.7 (2003), pp. 634–648.

[2] Andrea Arcuri, Muhammad Zohaib Iqbal and Lionel Briand. 'Random Testing: Theoretical Results and Practical Implications'. In: *IEEE Transactions on Software Engineering* 38.2 (2012), pp. 258–277.

[3] Nathaniel E Baughman and Brian Neil Levine. 'Cheat-Proof Playout for Centralized and Distributed Online Games'. In: *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*. Vol. 1. IEEE. 2001, pp. 104–113.

[4] Yahn W Bernier. 'Latency Compensating Methods in Client/Server In-Game Protocol Design and Optimization'. In: *Proceedings of Game Developers Conference 2001*. Mar. 2001.

[5] Paul Bettner and Mark Terrano. '1500 Archers on a 28.8: Network Programming in Age of Empires and Beyond'. In: *Proceedings of Game Developers Conference 2001*. Mar. 2001.

[6] Emil Börjesson and Robert Feldt. 'Automated System Testing Using Visual GUI Testing Tools: A Comparative Study in Industry'. In: *Verification and Validation (ICST), 2012 IEEE Fifth International Conference on Software Testing*. IEEE. 2012, pp. 350–359.

[7] Mark Claypool and Kajal Claypool. 'Latency and Player Actions in Online Games'. In: *Communications of the ACM* 49.11 (2006), pp. 40–45.

[8] Eric Cronin, Burton Filstrup and Sugih Jamin. 'Cheat-Proofing Dead Reckoned Multiplayer Games'. In: *International Conference on Application and Development of Computer Games*. Citeseer, 2003.

[9] Eric Cronin et al. 'An Efficient Synchronization Mechanism for Mirrored Game Architectures'. In: *Multimedia Tools and Applications* 23.1 (2004), pp. 7–30.

[10] Christophe Diot and Laurent Gautier. 'A Distributed Architecture for Multiplayer Interactive Applications on the Internet'. In: *IEEE Network* 13.4 (1999), pp. 6–15.

[11] Stefano Ferretti. 'Synchronization in Multiplayer Online Games'. In: *Handbook of Digital Games* (2014), pp. 175–196.

[12] Wildfire Games. *Debugging*. Feb. 2016. URL: `http://trac.wildfiregames.com/wiki/Debugging` (visited on 29/02/2016).

[13] Dennis Geels et al. 'Friday: Global Comprehension for Distributed Replay'. In: *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*. NSDI'07. Cambridge, MA: USENIX Association, 2007, pp. 21–21.

[14] Dennis Geels et al. 'Replay Debugging for Distributed Applications'. In: *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference*. ATEC '06. Boston, MA: USENIX Association, 2006, pp. 27–27.

[15] Joshua Glazer and Sanjay Madhav. *Multiplayer Game Programming. Architecting Networked Games*. Addison-Wesley Professional, 2015.

[16] Steffen Herbold et al. 'Deployable Capture/Replay Supported by Internal Messages'. In: *Advances in Computers* 85 (2012), pp. 327–367.

[17] Nicholas Hunt et al. 'DDOS: Taming Nondeterminism in Distributed Systems'. In: *ACM SIGARCH Computer Architecture News* 41.1 (2013), pp. 499–508.

[18] Dileepa Jayathilake. 'Towards Structured Log Analysis'. In: *2012 International Joint Conference on Computer Science and Software Engineering (JCSSE)*. IEEE. 2012, pp. 259–264.

[19] Ho Lee et al. 'Multiplayer Game Cheating Prevention With Pipelined Lockstep Protocol'. In: *Entertainment Computing*. Springer, 2003, pp. 31–39.

[20] Kyungmin Lee and Dongman Lee. 'A Scalable Dynamic Load Distribution Scheme for Multi-Server Distributed Virtual Environment Systems With Highly-Skewed User Distribution'. In: *Proceedings of the ACM Symposium on Virtual Reality Software and Technology*. ACM. 2003, pp. 160–168.

[21] Martin Mauve, Stefan Fischer and Jörg Widmer. 'A Generic Proxy System for Networked Computer Games'. In: *Proceedings of the 1st Workshop on Network and System Support for Games*. ACM. 2002, pp. 25–28.

[22] Martin Mauve et al. 'Local-Lag and Timewarp: Providing Consistency for Replicated Continuous Applications'. In: *IEEE Transactions on Multimedia* 6.1 (2004), pp. 47–57.

[23] Colt McAnlis et al. 'Real-Time Multiplayer Network Programming'. In: *HTML5 Game Development Insights*. Springer, 2014, pp. 195–209.

[24] Richard McNally et al. *Fuzzing: The State of the Art*. Tech. rep. DSTO–TN–1043. Australian Government, Department of Defence, Defence Science and Technology Organisation, 2012.

[25] Jens Müller and Sergei Gorlatch. 'Rokkatan: Scaling an RTS Game Design to the Massively Multiplayer Realm'. In: *Computers in Entertainment (CIE)* 4.3 (2006), p. 11.

[26] Stanislava Nedyalkova and Jorge Bernardino. 'Open Source Capture and Replay Tools Comparison'. In: *Proceedings of the International C* Conference on Computer Science and Software Engineering*. ACM. 2013, pp. 117–119.

[27] Adam Oliner, Archana Ganapathi and Wei Xu. 'Advances and Challenges in Log Analysis'. In: *Communications of the ACM* 55.2 (2012), pp. 55–61.

[28] OpenClonk. *Sync Losses*. Sept. 2011. URL: `http://wiki.openclonk.org/w/Sync_losses` (visited on 29/02/2016).

[29] Lothar Pantel and Lars C Wolf. 'On the Impact of Delay on Real-Time Multiplayer Games'. In: *Proceedings of the 12th International Workshop on Network and Operating Systems Support for Digital Audio and Video*. ACM. 2002, pp. 23–29.

[30] Joseph D Pellegrino and Constantinos Dovrolis. 'Bandwidth Requirement and State Consistency in Three Multiplayer Game Architectures'. In: *Proceedings of the 2nd Workshop on Network and System Support for Games*. ACM. 2003, pp. 52–59.

[31] Jouni Smed, Timo Kaukoranta and Harri Hakonen. 'Aspects of Networking in Multiplayer Computer Games'. In: *The Electronic Library* 20.2 (2002), pp. 87–97.

[32] Bjarne Stroustrup. *The C++ Programming Language*. Pearson Education, 2013.

[33] Andrew Weinkove. *Minimizing the Pain of Lockstep Multiplayer*. Tundra Games. Nov. 2015. URL: `http://www.tundragames.com/minimizing-the-pain-of-lockstep-multiplayer/` (visited on 24/02/2016).

[34] The Battle for Wesnoth. *OOS (Out of Sync)*. May 2015. URL: `https://wiki.wesnoth.org/OOS_(Out_of_Sync)` (visited on 29/02/2016).

[35] Beverly Yang and Hector Garcia-Molina. 'Designing a Super-Peer Network'. In: *Proceedings. 19th International Conference on Data Engineering*. IEEE. 2003, pp. 49–60.

[36] Georgios N Yannakakis and Julian Togelius. 'A Panorama of Artificial and Computational Intelligence in Games'. In: *IEEE Transactions on Computational Intelligence and AI in Games* 7.4 (2015), pp. 317–335.

Appendix A

Example Parser Output

```
Enabled out-of-sync bugs:
Bug 3, 1444-11-11, Change value: Gift

Triggered out-of-sync bugs:
Bug 3, 1445-07-05, Change value: Gift

Times until bugs triggered:
Bug 3, 236 day(s), Change value: Gift

Offset:             160 day(s)
OOS date:           1446-08-28
Offset OOS date:    1446-03-21
First command date: 1446-03-21
Last command date:  1446-08-28

Total commands: 9746

Near command occurrences:
personal_deity              62.89%
diplomaticactioncommand     22.49%
move_order                  3.54%
merge_units                 2.09%
needs_regiments_command     1.42%
use_ai_money_command        1.20%
buildunit                   1.05%
cancel_movement             0.87%
consolidate                 0.81%
spend_power_command         0.64%
transfer_subunit            0.56%
root_out_corruption_command 0.32%
assign_leader               0.28%
unsafe_move_command         0.28%
change_advisor              0.24%
toggle_main_army            0.19%
```

```
needs_buildings_command     0.19%
needs_ships_command         0.15%
set_break_alliance_command  0.12%
send_merchant               0.12%

Near stages:
Stage 18, 1446-04-06, Invite to federation
Stage 19, 1446-05-06, Declare war
Stage 20, 1446-06-05, Request peace
Stage 21, 1446-07-05, Give war subsidies
Stage 22, 1446-08-04, Improve relation

OOS detected 419 day(s) after first bug was triggered
Hotjoin did not happen

Inconsistent checksums:
9 : nCashCheckSum
14: nCountryMaxManpower
75: nConstructions

Likely causes:
Improve relation       30.85%
Give war subsidies     24.86%
Request peace          18.51%
Declare war            12.34%
Invite to federation   6.17%
Sell province          0.36%
Raze province          0.36%
Construct building     0.36%
Construct regular unit 0.36%
Send colonist          0.36%
```