Adversarial Behaviour Debugging in a Two Button Fighting Game

Nathan John and Jeremy Gow
Game AI Research Group
School of Electronic Engineering & Computer Science
Queen Mary University of London, UK
{n.m.john-mcdougall, jeremy.gow}@qmul.ac.uk

***Abstract*—We introduce the concept of Adversarial Behaviour Debugging (ABD), using intelligent agents to assist in the debugging of human-authored game AI systems, and *Bonobo*, an ABD system for Unity. To investigate the differences between ABD and traditional playtesting, we collected gameplay data from *Bonobo* agents and human testers playing against a buggy AI opponent. We present a comparison of the differences in gameplay and an online study on whether these differences affect observers perceptions of the AI opponent. We found that while there were clear differences, ABD compares favourably to human testing and has the potential to offer a distinct perspective.**

***Index Terms*—Games development, debugging, game AI, bugs, testing, neuroevolution**

I. Introduction

AI agents acting as opponents and NPCs are common in modern video games, with behaviours crafted by the developers using techniques such as behaviour trees, utility, and planning [1]. The more complicated the game, the more sophisticated these *authored agents* must be in order to deliver the required player experience. And the more sophisticated the authored agent, the more time and effort developers typically must dedicate to debugging its behaviour [2].

The standard approach to debugging agent behaviour is to use human playtesters: observing and analysing people playing the game to identify *bugs* — where the agent doesn't work as intended or doesn't meet technical criteria — or *exploits* — where the agent may work as intended but detracts from the experience. This is a relatively expensive process which must be repeated throughout development [3].

In this paper, we propose *Adversarial Behaviour Debugging* (ABD) as an alternative approach. ABD is a form of semi-automated testing where a developer uses other intelligent agents to gain insights into the behaviour of a game's AI systems. While this is unlikely to replace the need for human testers, we examine whether ABD can provide an additional source of insight for developers and hence help them identify bugs and exploits before expensive human testing.

In Section III we describe ABD as a general approach to game development. Section IV puts the case for using neuroevolution in ABD, while Section V introduces *Bonobo*, a neuroevolution-based ABD system for Unity. The rest of the paper looks at how ABD compares to human playtesting in the context of *DropFeet*, a simple two button fighting game.

978-1-6654-3886-5/21/$31.00

We developed a challenging but exploitable AI opponent for *DropFeet* and collected playthrough data from tests with human players, Bonobo's adversarial agents, and a random test agent. We present two views of these playthroughs and their relative usefulness for debugging: an analysis of the objective differences in test gameplay (Section VI), and a study of observers' perceptions of the authored agent (Section VII). We find that playthroughs generated by ABD perform comparably to human replays for gaining insights into an authored agent, providing potential for further exploration.

II. Related Work

A. Debugging and Software Engineering

In 1997, Lieberman described debugging as the "dirty little secret of computer science" [4], and observed that debugging is primarily done through trial and error, a fact that has not changed since thirty years prior. Beller et al. [5] found that twenty years later the methods used by programmers to debug had not changed much. Their study of programmers' debugging habits found that print statements more used than in-built debugging features and more powerful tools. While unable to give a conclusive answer as to why, Beller et al. was able to establish that programmers were aware of these alternatives. One explanation for these practices is the lack of immediacy provided by debugging tools [6]. Alternatively, it could be that tools and techniques have not kept up with changes in the software engineering landscape [7].

Search-Based Software Engineering (SBSE) [8] suggests that aspects of software engineering can be considered an optimisation problem, and encourages the use of search techniques. A common area of interest in SBSE is testing and debugging, with Nguyen et al.'s test case generation through neuroevolution being relevant here [9].

There are some unique considerations when the piece of software being debugged is a computer game. Functionality is more likely to change during development compared to traditional software, so technical infrastructure needs to support fast prototyping [10]. Traditional automated testing methods can be too rigid for use in game development. Murphy-Hill et al. [11] posit writing automated tests for games can be difficult partly because "it is harder to explore the state space in games". This could explain industry reliance on human testers, which is also cheaper and more flexible than test engineers. When automated

tests suites are developed for games, the tests must deal with the idiosyncrasies of game development, as documented in case studies from Riot Games [12] and Electronic Arts [13].

The specific problem of debugging the AI systems of computer games presents yet another set of unique challenges. In a previous study, we found that the three primary reasons that AI programmers found debugging game AI agents difficult were that AI bugs are difficult to identify; reliably reproducing bugs in the AI can be difficult and that AI systems are conceptually complex [2]. A 2017 GDC panel on game AI testing highlighted the conceptual complexity of debugging and the interconnected nature of game systems: one must consider *"how the AI is going to change the game-state, and then how the AI is going to react to that because you could put in a new feature that breaks something else, somewhere else because it's changing game-state now that you weren't prepared for"* [14]. Hence developers have adopted AI architectures to reduce the mental load to debug [15, 16] as well as specialised debugging tools [17, 18]. Even so, the process of debugging game AI can be extremely time consuming [19].

B. Playtesting with intelligent agents

The use of intelligent agents as a tool for playtesting games is an area of active research. There are two related kinds of testing that works generally apply to. One is finding/detecting defects in the game which relates to bug-hunting and helping the programmers. The other is play-testing the game, relating to aiding with the design of the game, which is more commonly explored[20, 21, 22]. While ABD focuses on the former, both branches of literature contain some relevant insights. For example Keehl et al.'s work on Monster Carlo is framed within the context of game design, however a similar framework could be effective for debugging.

The most relevant work is Machado et al's Cicero[23], a mixed initiative system that provides AI assisted debugging including heat-maps, game agents playing using Monte-Carlo Tree Search (MCTS) [24], and a replay analysis tool — SeekWhence [25]. Here the authors acknowledge that a shortcoming of many AI-assisted game design tools are that they are often closely tied to the implementation of the specific game they're being used for, and for this reason it "becomes difficult when a developer wants to apply the same techniques to another project without re-implementing." Ariyurek et al. also used MCTS as the intelligent agent of choice, exploring the use of human-like agents as QA testers. They found that these human-like agents found a comparable amount of the bugs to the human testers.

C. Neuroevolution

Neuroevolution is a hybrid machine learning algorithm that applies evolutionary algorithms to artificial neural networks, instead of backpropagation [26]. A population of networks is instead evaluated by a fitness function and the highest performing are bred together to create a new generation of networks, with this cycle repeating hundreds or thousands of times. Various approaches have been proposed to encoding and breeding networks [27].

NEAT is a popular neuroevolution algorithm where both the weights and the topology of the ANN is evolved, which outperforms evolving the weights alone [28]. NEAT also maintains a good level of diversity within the population via speciation, which helps avoid local maxima. NEAT has had success in multiple domains [29, 30], including video game playing agents [31].

There has been a variety of research into the application of neuroevolution to video games [26], including game-playing agents, content generation and player behaviour modelling. However, Risi and Togelius [26] notes that there have been no notable examples of neuroevolution in the game industry for playing agents, with game developers citing lack of control and clarity as issues when working with neural networks.

III. Adversarial Behaviour Debugging

We define *Adversarial Behaviour Debugging (ABD)* as the use of intelligent agents to play against human-authored AI behaviours, in order to aid the developer in debugging those behaviours. As there are two distinct kinds of AI agent in ABD, we will use the term *authored agents* to refer to the game systems crafted by the developers and *adversarial agents* to refer to the external systems used to test authored agents.

We propose ABD as an approach to debugging behaviours *before* the game is sent to Quality Assurance (QA) or playtesting. By observing the adversarial agents interacting with the authored agent, a developer is able to gain insights into the behaviour of their authored agent, and use those insights to make improvements.

It is important to note that these adversarial agents do not need to be good players of the game, nor to play in a human-like manner. Instead, the adversarial agents are primarily looking to explore the behaviour of the authored agent and reveal its flaws in such a way that gives a debugging developer insights into how to repair them.

ABD is a general approach to software engineering in games, which does not depend on any particular details of the game or adversarial agents. For example, it does not assume the game has a particular kind of environment or action space, or the presence of a forward model, although a developer could exploit these in ABD. For the adversarial agents, it does not assume how their behaviour is defined or generated, only that they sufficiently explore the authored agents' behaviour.

Adversarial agents could be implemented in a variety of ways, depending on the tools and resources available to the developer. Various AI methods could be applied, whether that be a suite of MCTS agents with varying personas similar to Holmgård's procedural personas [20], a population of agents evolved with neuroevolution (explored below), or — if the developer has a glut of compute resources available to them — agents trained using deep reinforcement learning. Alternatively, the adversarial agents could simply make use of the authored agents, with tweaks to emphasise the exploration of behaviour rather than playing the game well.

Considering how ABD could be adopted in game development, there are three core components that developers would need to provide: a *test environment*, a *control interface*, and some form of *behaviour visualisation*.

a) Test Environment: An environment where adversarial and authored agents can interact. For small games this could be the entire game, e.g. DropFeet described in Section VI-A. For larger games, this could be a scaled-down white-box area where the authored behaviour could be explored. Multiple test environments could be used, each acting as a separate automated test case.

b) Control Interface: An API for the adversarial agent to perceive and control the game. Adversarial agent inputs provide some representation of the current game state, and a driver translates the agent's output into game actions. In some projects, it may be possible to use sensory data that is provided for the authored AI, and game metrics that are already implemented as inputs.

c) Behaviour visualisation: One of the benefits of ABD is being able to observe the adversarial agents as they interact with the authored agent. By observing these interactions a developer is given the opportunity to make insights of the behaviour of their authored agents. As such, and implementation requires some method of displaying the behaviour of the adversarial agents, while the process is running.

IV. ABD and Neuroevolution

While the adversarial agents used in ABD could be implemented with a variety of techniques our initial approach uses neuroevolution, and specifically NEAT. Statistical Forward Planning techniques such as MCTS could also be used as adversarial agents, however these requires the developer to implement a forward model into their game, which is an assumption that machine learning methods do not require. When comparing machine learning methods for suitability as adversarial agents, reinforcement learning techniques generally require large amounts of data and computation in order to learn even simple behaviours, whereas neuroevolution has shown itself to often solve problems faster and more efficiently.

Looking specifically at NEAT, its speciation encourages a range of agent behaviours, acting as a broad exploration of the behaviour space. And as different species generally represent different behaviours, a user can look at a representative from each species to get an idea of the behaviours that exist in the system, without having to exhaustively observe every member of the population.

One cost of neuroevolution is the specification of a fitness function to evaluate the adversarial agents. However, provided developers can craft numerical definitions of interest and potential bugginess specific to their game, they are a fairly friendly interface which ignores the underlying learning model. Additionally, interactive evolution is a fairly common paradigm that could allow developers to input their insight into the evolution of testing agents. Given bugginess can be very game and experience specific, using a technique that a designer can influence is a potential advantage.

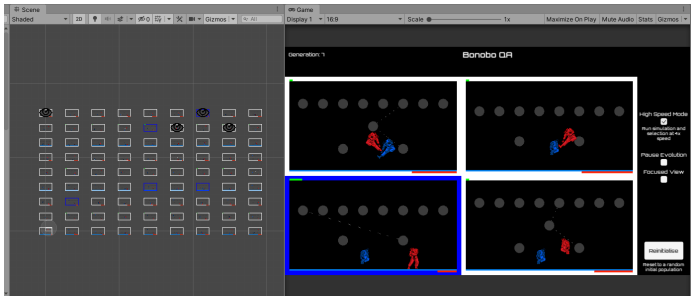

Fig. 1. *Bonobo* running DropFeet in Unity's editor mode. The entire population of 81 adversarial agents is visualised on the left, with a selection of game instances (marked with an eye) displayed in detail so that the agent behaviours can be compared.

V. The Bonobo System

Bonobo is a system for Adversarial Behaviour Debugging in the Unity3D game engine. When supplied with a *game instance*, which contains the gameplay logic, the authored agents under inspection, and a game state evaluation function, *Bonobo* manages the testing process, training adversarial agents using NEAT [28] with a population of game instances. It is based on SharpNEAT [32], a popular implementation of NEAT for C#, the native language of Unity. SharpNEAT is highly modular and extensible: we use a custom implementation of its Experiment class to define fitness functions for game instances based on gameplay metrics. It also supports custom implementations of speciation and selection, which we intend to make use of in future versions of *Bonobo*.

Currently, we have developed two simple Unity games for use with *Bonobo*: *Ping*, an air hockey like game, and *DropFeet* (described in Section VI-A), a clone of the cult two button fighting game *Divekick*. To be compatible with *Bonobo*, a game needs to provide a Unity prefab defining a game instance, allowing *Bonobo* to instantiate multiple versions for parallel evaluation.

The *game instance* inherits from a base game instance class which defines an API for receiving an adversarial agent and reporting back a game state evaluation value. This class is also where each game implementation defines how many inputs and outputs the adversarial agent's neural network should have. The developer is responsible for defining these network inputs (player perceptions), outputs (player actions), and the game state fitness function.

Additionally, *Bonobo* allows a developer to provide a Boolean test for "interesting" game instances, which the system can highlight during evolution. This helps the developer identify and investigate gameplay which they suspect represents bugs or exploits in the authored agent. For example, DropFeet game instances are classified as interesting simply if the adversarial agent has scored more points than the authored agent.

A developer is free to create any game logic they wish using the features of Unity, with a few technical constraints which are specific to the design of *Bonobo*, such as having all gameplay code in FixedUpdate so the game is deterministic

when played at high speeds, and restricting game objects to a defined space. The constraints reflect choices we made to prototype ABD more easily within Unity, but may not be required by other ABD systems. For example, one could evaluate the game instances sequentially, use separate scenes, parallelise across multiple machines, or run the game at normal speeds.

The core *Bonobo* system manages the testing process including the main UI, which game instance prefab to be tested, and settings for how many instances will be in the population, how long a generation of evaluation lasts, and how many instances will be displayed at one time in the main view. These settings allow the exact behaviour of *Bonobo* to be tweaked depending on the requirements of the game instance being tested.

Running *Bonobo* instantiates multiple copies of the *game instance* arranged as a grid in world space. Each instance has a member of the adversarial agent population and runs for a specified time. The fitness of each instance is then evaluated and passed to the NEAT algorithm. There is a high-speed mode which increases Unity's timescale by four, which still allows a developer to keep track of the gameplay. In a future version we hope to include automatic isolation of interesting or erroneous behaviour, allowing higher speeds.

Rather than displaying the entire population of game instances, the main view of *Bonobo* shows a selection. This is based on a combination of state evaluation function and NEAT's speciation, sorting the species by their fitness and displaying representative members from the top species.

VI. Human Versus ABD Gameplay

As a first step to understanding how ABD compares to human playtesting, we conducted a study with the *Bonobo* system and DropFeet, a simple two button fighting game (described in Section VI-A). We looked at the objective differences in gameplay when testing a challenging but exploitable authored agent for DropFeet (described in Section VI-B). From these differences we can make inferences about how useful each approach will be for debugging. A random test agent was also included, representing a cheaper baseline approach to supplementing human playtesting. Three sets of DropFeet playthrough data were collected (Section VI-C) — with humans, with *Bonobo* adversarial agents, with our random agent — and a comparative analysis performed (Section VI-D).

A. DropFeet

DropFeet is a simplified version of Divekick, a two button fighting game for two players. The players are coloured red and blue, and the goal is to hit your opponent with a diving kick. Landing a kick to the body earns 1 point, landing a kick to the head earns 2 points. Once a point is scored, the round is over and the players are reset to their starting positions. Pressing the Jump button on the ground makes the character jump directly upwards. Pressing the Kick button on the ground makes the character take a short hop backward. Finally, pressing the Kick button in the air immediately causes the character to kick diagonally downwards in the direction they are facing. The higher in a jump arc a player presses kick, the further a kick will travel. Once a player commits to a kick, they are locked in until they hit the ground. If both buttons are pressed at the same time, jump is prioritised on the ground, and kick is prioritised in the air. There are no other inputs for this game. For this experiment, the red fighter is always be the authored agent, and the blue fighter is the tester — either a human, the random agent, or *Bonobo* adversarial agent.

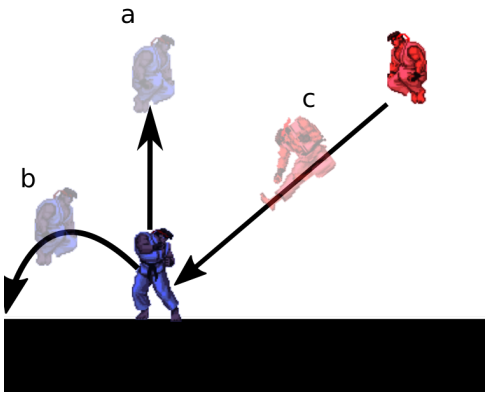

Fig. 2. The possible movement options in DropFeet `a`: Jump directly up. `b`: Hop backwards. `c`: Launch a diving kick from the air.

B. The Exploitable Authored Agent

An authored agent was developed to play DropFeet, which is controlled through a set of utility functions that evaluate which action to take based on the game state. Of note is the fact that the authored agent has some randomness included in its actions, manifesting in the agent sometimes jumping in order to advance or gain positional advantage when it is a safe choice. This unpredictability makes the agent a better opponent, as predictability is a big weakness in fighting games. For this analysis, a bug was included in the agent which causes the agent to launch attacks from the top of its jump arc which can easily be avoided and counterattacked by the agent's opponent. This behaviour is highly exploitable, as it will perform these jumps both as an attempt to counter attack, and randomly instead of doing a low attack to advance safely. This bug was chosen from a set of three buggy variations of the agent that were developed, as it stuck a balance between being observable, but not too obvious, representing the kind of bugs that would be expected to be picked up by QA.

C. Playthrough Collection

The game logged the full state and player actions, which could later be replayed to reproduce the gameplay. For each condition, playthrough logs were programmatically edited to produce roughly 15 second playthrough segments, always ending on a score to include complete rounds. A set of eight

segments per condition were selected for analysis, allowing us to compare more easily across conditions. For the human and random testers, those segments with the highest score advantage for the tester were selected, representing gameplay where the authored agent could be manifesting a bug or exploit. This reflects the preprocessing a developer would do in order to look for the most informative sections of a playthrough session. The adversarial agents were quite repetitive in their gameplay, so we simply generated segments of the required length.

1) Human Playthroughs: To gather human test data, four players were given a version of DropFeet with the authored agent. These players were selected for their familiarity with the fighting genre, so they could adequately serve as testers. One works as a game developer. They were given instructions on how to play the game and asked to play for at least two minutes. For each of the four players, the two segments with the largest score difference in the player's favour were selected.

2) Random Playthroughs: The authored agent played against a random agent that took an action every 4 frames (DropFeet's game logic runs at a fixed 60fps). It had an equal probability of pressing the Jump button, the Attack button, both, or neither. 30 minutes of data was collected, to reflect the amount of human test data.

3) Adversarial Agent Playthroughs: Bonobo used a fitness function that encouraged a variety of actions and scoring against the authored agent. It was penalised for pressing no buttons, and rewarded if it presses both buttons through the match. There was a large reward for scoring points and a small negative reward for conceding points. Playthrough data was gathered from two runs of *Bonobo*, each with a population size of 81 and lasting 200 generations (about 30 mins). We did not tune the population size or generations, but had observed this maintained a reasonable variation in behaviour and was long enough to develop interesting adversarial behaviours. Two separate runs of *Bonobo* were used as the neuroevolution process is stochastic, so this would mitigate against one lucky or unlucky run, while avoiding cherry picking the results from a unrealistic number of runs. The eight playthrough segments were generated from the fittest member of the top four species generated by each run.

D. Playthrough Analysis

Figure 3 shows the scores achieved by the testers and authored agent for each playthrough segment, and the bounding ellipses for each type. Looking at scores we can get an idea of how the various testers performed against the authored agent in the selected playthroughs. The human and random agents sets of scores are both more tightly grouped than the adversarial agent, with the human players performing better, and the random agents performing close to 1 for 1. On the other hand there are a wide range of score differences from the adversarial agent. This seems to indicate that the adversarial agent playthroughs have more variation than the other types

Fig. 3. A scatterplot of the scores achieved by the playthroughs used in the survey. The line represents where games of an even score would lie. Ellipses are multivariate normal distribution confidence ellipses with a confidence level of 0.95.

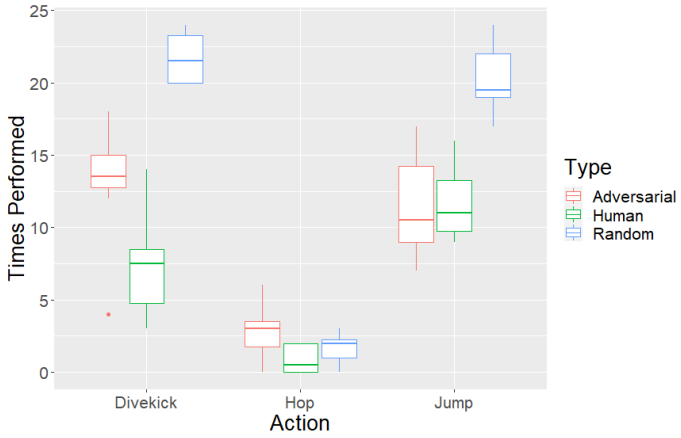

Fig. 4. A Boxplot taking each collection of 8 playthroughs and plotting how many time each action was performed per second.

of playthough, which may make sense due to the differing selection method.

Figure 4 shows a boxplot of how many times each action was performed per second in each playthrough. This helps us get an idea of differences and similarities between how the actions are selected. In general, the adversarial agents are more similar to human players than the random agent, with fewer actions per second. There are a low amount of hops in all three conditions. When looking at the movements of the agents, a similar story appears. Low use of hops seems to show that score differences were lower when hops were used more often — authored agent often won when the opponent used hops — since the random agent selected their actions with even probability and yet not many hops were present in the selected representative gameplay segments. It is interesting that the adversarial agent used the most hops, but averaged a better score difference. Hops were used more intentionally than with the random opponent, showing the adversarial behaviour was not random.

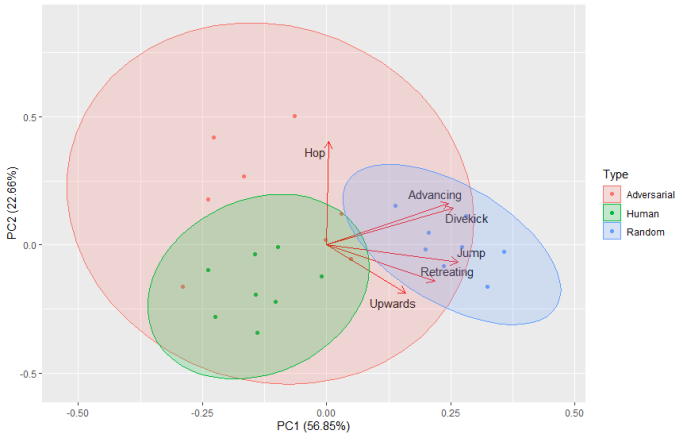

Fig. 5. PCA biplot of all the statistics gathered from the playthrough segments. The wide range of positions the adversarial agents cover seem to show a wide coverage of the behaviour space

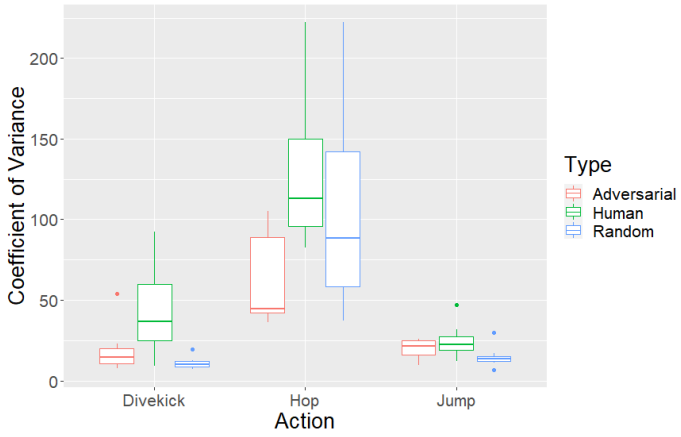

Fig. 6. A Boxplot that plots the coefficient of variance of how many times each action was taken for a sliding 5-second window of each playthrough.

Figure 5 is a biplot of a PCA based on the frequency of actions taken per second, and distances moved per second by the tester in each playthrough of each type, and helps us characterise the different playthoughs. The X-axis could be described as *buttonsy*: high horizontal movement and pressing lots of buttons, therefore taking lots of actions. The Y-axis can be described as *hoppy*: using the back hop often and having lower vertical distance travelled. Random appears to be highly buttonsy, whereas humans are far less buttonsy, and less hoppy. Adversarial agents cover a wide range of this space, overlapping with both other groups. This indicates adversarial agents explore a wider range of test behaviours.

The most striking visual difference between the playthrough types is that the adversarial agents are far more repetitive and consistent. In any single playthrough segment, the human players have more variance in what they attempt, whereas an individual adversarial agent will attempt to repeat the same strategy over and over. This can also be seen by looking at the statistics for a sliding 5 second window which moves in 1 second increments for each replay segment, then calculating the coefficient of variance for the statistics. Figure 6 shows plots of the coefficient of variance for each playthrough's stats. The highly repetitive nature of the adversarial agents leads to a low coefficient of variance, whereas the human's wide range of gameplay styles lead to higher values. The random agent has a low coefficient of variance since the actions taken are uniformly selected through the whole replay.

E. Discussion

The most significant difference between the human replays and the adversarial agents is the consistency of actions taken within a single replay. This consistency could be useful for a developer observing the playthrough in isolating which action or circumstance is causing unwanted behaviour. Also, adversarial agents play-styles cover a wide range of behaviours that overlap both the clustered human players, and the clustered random players. This can be interpreted as adversarial agents covering a wider range of the behaviour space that the two other options. This may mean it's possible that adversarial agents will bring up bugs that might not be caught by a random agent, or a human tester.

VII. Effects on Perceptions of Authored Agents

Given the observed differences between the human and ABD testing, our next question was how these might impact debugging. Specifically, do the differences affect people's ability to infer bugs or exploits in the authored agent? We performed an online study to investigate this question.

A. Perception Survey

Participants were given a description of the game DropFoot and told that its developer has written an AI opponent for the game and gathered some video playthroughs to see how the AI performs. They were then presented with a video of either humans or adversarial testing. Below the video they are asked to rate how exploitable or buggy they think the authored agent is, from 1-10, and are able to scroll through the video while they settle on their answer. On the next page, they are shown the same video again for reference and asked to select which of the following statements describes the error/exploit that exists in the agent, two of which were correct:

- The red player launches some attacks from too far away (Correct)
- The red player is predictably defensive
- The red player doesn't react to the blue players motions
- The red player can be baited into an easily attacked jump (Correct)
- None of the above

Finally, the participants are asked to rate how useful the set of replays were for finding exploits in the authored agent on a 7-point scale from Extremely Useless to Extremely Useful.

B. Results & Discussion

The survey has a total of 163 responses, with 57 having seen the human replays, 48 having observed the adversarial agents,

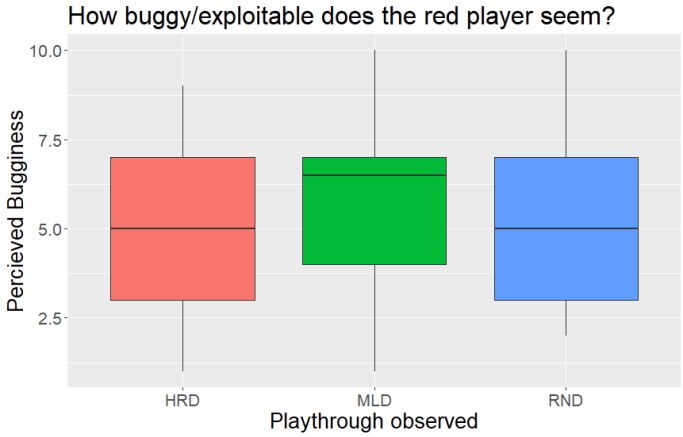

Fig. 7. Compared to watching replays from humans, replays from adversarial agents are rated as seeming more exploitable, whereas replays from random agents are rated similarly.

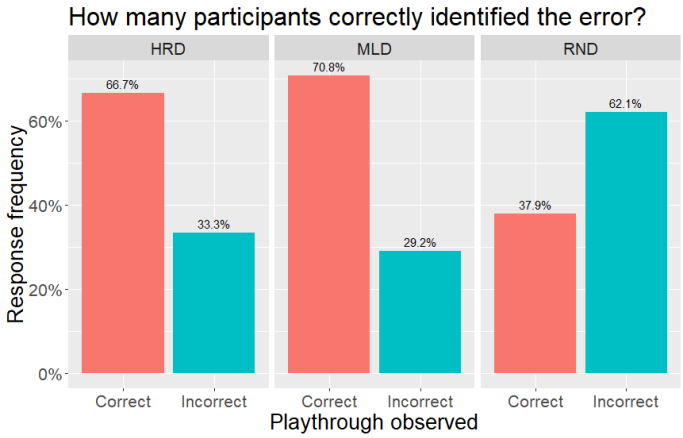

Fig. 8. Compared to the human replays, participants who observed the random agent performed significantly worse at correctly identifying the error.

and 58 having seen the random agent. Looking at responses to the first question investigating how buggy or exploitable the participants rated the authored agent, shown in Figure 7 the agent was rated as more buggy when the players observed the adversarial agents with a mean exploitable rating of 5.7, compared to the human replays 4.8 (Wilcoxon rank sum p-value=0.039). On the other hand, comparing the human replays to the random playthroughs, no significant effect seemed to appear from this sample (Wilcoxon rank sum p-value=0.322).

Considering the ability of participants to correctly identify the error in the authored agent, the only significant effect when compared to the human replay data was that participants observing the random replays performed significantly worse than the human replays (Chi-squared p-value: 0.00376). No significant effect appeared when comparing the adversarial agents to the human replays.

Finally, neither the random agent (mean 5.3) nor the adversarial agent (mean 5.1) were significantly different to the human replay data in terms of how useful the participants evaluated the usefulness of the replays (mean 5.2).

These results indicate that using replays from ABD can provide value to developers attempting to debug their game AI, when compared to using human testers. As participants that observed the adversarial agent replays rated the authored agent as more exploitable, it could be imagined that adversarial agents are a suitable tool for helping developers get an inkling about whether their authored agents are behaving properly. The fact that the adversarial agent replays performed similarly to the human replays when it came to identifying the error is also of significance, as whereas with human replays, adding a breakpoint while the tester is playing is usually out of the question, with an adversarial agent playthrough, by re-running the exported agent a developer would be able to inspect the insides of their authored agent while the adversarial agent was playing against it.

VIII. Conclusions and Further Work

We proposed Adversarial Behaviour Debugging as a technique to aid in debugging of authored agents; described *Bonobo*, an implementation of ABD in the Unity game engine; performed analysis on how ABD playthroughs perform compared to human and random testers, and performed a study investigating how these different testers affect an observer's perception of a buggy authored agent.

Playthrough segments from adversarial agents cover a wider range of behaviour than human or random testers, while performing behaviours more consistently than humans. This could be useful for discovering or reproducing bugs and exploits that might get overlook during human testing. However, how an observer interprets *Bonobo* testing compared to human testing suggests that ABD playthroughs provide *different* insights, rather than exclusively *better* insights. Rather than acting as a replacement for human testers, ABD is likely better suited as a supplement.

There are still several different questions that could be asked, and we've only looked at one, 'How does ABD effect participants perception an agents bugginess'. There are many other questions that could be asked and would be interesting, such as presenting participants with playthroughs of buggy agents and non-buggy agents, to investigate if ABD playthroughs effect participant accuracy in identifying if a bug is present or not. We could present participants with a range of different buggy agents to investigate how ABD playthroughs effect participants ability to identify different bugs.

DropFeet is a simple game environment and further work could generalise this work to more complicated agents and environments. We also use the original NEAT algorithm, and have yet to explore whether alternative neuroevolution algorithms, such as RBF-NEAT[33] or Cascade-NEAT [34] could create different explorations of the behaviour space and be even more useful for debugging.

Acknowledgments

Nathan John is supported by the IGGI EPSRC Centre for Doctoral Training (EP/L015846/1).

References

[1] M. Dawe, S. Gargolinski, L. Dicken, T. Humphreys, and D. Mark, "Behavior Selection Algorithms An Overview," in *Game AI Pro*. CRC Press, 2013, pp. 47–60.

[2] N. John, J. Gow, and P. Cairns, "Why is debugging video game AI hard?" in *Proc. AISB AI & Games symposium*, Apr. 2019, pp. 20–24.

[3] H. M. Chandler, *The Game Production Handbook*. Jones & Bartlett Publishers, 2009.

[4] H. Lieberman, "The Debugging Scandal and What to Do About It," *Commun. ACM*, vol. 40, pp. 26–29, 1997.

[5] M. Beller, N. Spruit, D. Spinellis, and A. Zaidman, "On the dichotomy of debugging behavior among programmers," in *IEEE/ACM ICSE*, 2018, pp. 572–583.

[6] D. Ungar, H. Lieberman, and C. Fry, "Debugging and the experience of immediacy," *Communications of the ACM*, vol. 40, no. 4, pp. 38–43, 1997.

[7] M. Kassab, J. F. DeFranco, and P. A. Laplante, "Software Testing: The State of the Practice," *IEEE Software*, vol. 34, no. 5, pp. 46–52, 2017.

[8] M. Harman and B. F. Jones, "Search-based software engineering," *Information and Software Technology*, vol. 43, no. 14, pp. 833–839, Dec. 2001.

[9] C. D. Nguyen, S. Miles, A. Perini, P. Tonella, M. Harman, and M. Luck, "Evolutionary testing of autonomous software agents," *Autonomous Agents and Multi-Agent Systems*, vol. 25, no. 2, pp. 260–283, Sep. 2012.

[10] J. Kasurinen, J.-P. Strandén, and K. Smolander, "What do game developers expect from development and design tools?" in *Proc. Int. Conf. on Evaluation and Assessment in Software Engineering*. ACM, 2013, pp. 36–41.

[11] E. Murphy-Hill, T. Zimmermann, and N. Nagappan, "Cowboys, Ankle Sprains, and Keepers of Quality: How is Video Game Development Different from Software Development?" in *Proc. of the 36th Int. Conf. on Software Engineering*. ACM, 2014, pp. 1–11.

[12] J. Merrill, "Automated Testing for League of Legends," Feb. 2016. [Online]. Available: https://technology.riotgames.com/news/automated-testing-league-legends

[13] C. Buhl and F. Gareeboo, "Automated testing: A key factor for success in video game development. case study and lessons learned," in *Proc. Pacific NW Software Quality Conferences*, 2012, pp. 1–15.

[14] D. Mark, E. Johansen, S. Ocio Barriales, and M. Robbins, "Behavior is Brittle: Testing Game AI," 2017, published: Presentation at GDC. [Online]. Available: https://youtu.be/RO2CKsl2OmI

[15] K. Dill, "Structural Architecture: Common Tricks of the Trade," in *Game AI Pro*, S. Rabin, Ed. A. K. Peters, Ltd., 2013, pp. 61–71, section: 5.

[16] D. Isla, "Handling Complexity in the Halo 2 AI," 2005. [Online]. Available: https://www.gamasutra.com/view/feature/130663/

[17] J. Gillberg, "Tom Clancy's The Division: AI Behavior Editing and Debugging," 2016, published: Presentation at GDC. [Online]. Available: https://www.youtube.com/watch?v=rYQQRIY_zcM

[18] M. Lewis and D. Mark, "Building a Better Centaur: AI at Massive Scale," 2015, published: Presentation at GDC. [Online]. Available: https://www.gdcvault.com/play/1021848/Building-a-Better-Centaur-AI

[19] G. Alt, "The Suffering: A Game AI Case Study," in *Challenges in Game AI workshop, 19th National Conf. on AI*, 2004, pp. 134–138.

[20] C. Holmgård, M. C. Green, A. Liapis, and J. Togelius, "Automated Playtesting with Procedural Personas through MCTS with Evolved Heuristics," *CoRR*, vol. abs/1802.06881, 2018.

[21] O. Keehl and A. M. Smith, "Monster Carlo: an MCTS-based framework for machine playtesting unity games," in *IEEE CIG*, 2018, pp. 1–8.

[22] S. Stahlke, A. Nova, and P. Mirza-Babaei, "Artificial Players in the Design Process: Developing an Automated Testing Tool for Game Level and World Design," in *Proc. CHI PLAY*. ACM, Nov. 2020, pp. 267–280.

[23] T. Machado, D. Gopstein, A. Nealen, O. Nov, and J. Togelius, "AI-Assisted Game Debugging with Cicero," in *IEEE Congress on Evolutionary Computation*, 2018.

[24] G. Chaslot, S. Bakkes, I. Szita, and P. Spronck, "Monte-Carlo Tree Search: A New Framework for Game AI." in *AIIDE*, 2008.

[25] T. Machado, A. Nealen, and J. Togelius, "SeekWhence a retrospective analysis tool for general game design," in *Proc. Int. Conf. on the Foundations of Digital Games*. ACM, Aug. 2017, pp. 1–6.

[26] S. Risi and J. Togelius, "Neuroevolution in games: State of the art and open challenges," *IEEE Trans. on Computational Intelligence & AI in Games*, vol. 9, no. 1, pp. 25–41, 2015.

[27] D. Floreano, P. Dürr, and C. Mattiussi, "Neuroevolution: From architectures to learning," *Evol Intell*, vol. 1, 2008.

[28] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary computation*, vol. 10, no. 2, pp. 99–127, 2002.

[29] ——, "Competitive coevolution through evolutionary complexification," *Journal of artificial intelligence research*, vol. 21, pp. 63–100, 2004.

[30] K. Stanley, N. Kohl, R. Sherony, and R. Miikkulainen, "Neuroevolution of an automobile crash warning system," in *Proc. conf. on Genetic and evolutionary computation*. ACM, 2005, pp. 1977–1984.

[31] K. O. Stanley, R. Cornelius, R. Miikkulainen, T. D'Silva, and A. Gold, "Real-time Learning in the NERO Video Game." in *AIIDE*, 2005, pp. 159–160.

[32] C. Green, *SharpNEAT*, 2018. [Online]. Available: https://github.com/colgreen/sharpneat

[33] N. Kohl and R. Miikkulainen, "Evolving neural networks for fractured domains," in *Proc. Conf. on Genetic and evolutionary computation*, 2008, pp. 1405–1412.

[34] ——, "Evolving neural networks for strategic decision-making problems," *Neural Networks*, vol. 22, no. 3, 2009.