Using Probabilistic Model Checking to Balance Games

William Kavanagh

Submitted in fulfilment of the requirements for the
Degree of Doctor of Philosophy

School of Computing Science
College of Science and Engineering
University of Glasgow

November 2021

Declaration

I declare that this thesis was composed by myself, the the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

Abstract

In this thesis, we consider problem areas in game development and use probabilistic model checking to address them. In particular, we address the problem of multiplayer game balancing and introduce an approach called Chained Strategy Generation (CSG). This technique uses model checking to generate synthetic player data representing a game-playing community moving between effective strategies. The results of CSG mimic *the metagame*, an ever-evolving state of play describing the players' collective understanding of what strategies are effective. We expand upon CSG with optimality networks, a visualisation that compares game material and can be used to show that a game exhibits certain qualities necessary for balance.

We demonstrate our approach using a purpose-built mobile game (RPGLite). We initially balanced RPGLite using our technique and collected data from real world players via the mobile app. The application and its development are described in detail. The gathered data is then used to show that the model checking did lead to a well-balanced game. We compare the analysis performed from model checking to the gameplay data and refine the baseline qualities of a balanced game which model checking can be used to guarantee.

We show how the collected data via the mobile app can be used in conjunction with the prior model checking to calculate action-costs – the difference between the value of the action chosen and the best action available. We use action-costs to evaluate player skill and to consider other factors of the game.

Acknowledgements

I thoroughly enjoyed every aspect of my PhD. I joked that I spent four years sitting in a basement thinking about how accurate Wizards are, which in fairness, is not far from the truth. Whilst I love the subject area and find the tools fascinating, it was the people around me who made it such a pleasant experience. To that end, I would like to say thank you to the following:

- **Ellen**, I am so very thankful for your support and encouragement. You made the decision to stay in Glasgow so very easy. I owe you everything.
- **Alice**, my supervisor, without whom this work would not have begun, let alone have reached this stage. Your effort and counsel were peerless throughout, for that I will always be hugely grateful. I am sorry I spent so much of our time talking about anything other than the work, but we got there in the end.
- Similarly **Gethin**, who supported me and my work far more than he needed to as my co-supervisor. Your passion and empathy were hugely useful, the school is very lucky to have you.
- To **my friends and colleagues at the University**, past and present. There are some lovely people diligently doing brilliant work at the school, it was a very happy environment in which to work. I can't list everyone, but notably Oana, Matt, Tim, Jeremy, Frances, Craig, David, Will, Blair, Ben and Kyle, I am indebted to you all.
- In particular **Tom Wallis**, my friend and now co-author. Your passion and enthusiasm is something I will always seek to emulate, as is the joyous manner with which you interact with the world.
- Back home, **my family** have always uplifted me and offered a calm, reassuring presence when I let myself get overwhelmed. A large part of what motivates me is to make you proud. Mum and Dad, whilst I hope I have outgrown asking you to read through my work, I am grateful for all the help you have given me. And Maeve, I am very lucky to have you as a role-model to look up to.

- Finally, **Penelope, Parker and now Emi**. Though you are sadly no longer with us, Parker and Penelope will always be in my heart. And now Emi, who was only around for the final days of this work, but has been a good boy throughout.

Research Artefacts

The following publications were all written and released during my PhD study. I am the lead author on all of these papers except one, for which I was a major contributor. These works are all included and expanded upon in this thesis.

1. **Kavanagh, W. J.**, Miller, A., Norman, G. and Andrei, O. (2019), Balancing Turn-Based Games with Chained Strategy Generation, in *IEEE: Transactions on Games*. Available at: https://ieeexplore.ieee.org/document/8846763. This paper introduces Chained Strategy Generation, a technique for game balancing using model checking, and describes our case-study RPGLite. Chapter 4 to Section 4.7 in this thesis is based on this work, recent advances upon it are included in the rest of the chapter.

2. **Kavanagh, W. J.**, Miller, A. (2019), Chained Strategy Generation: A Technique for Balancing Multiplayer Games Using Model Checking, In *Proceedings of ARW19: Automated Reasoning Workshop*, article 8, pages 15-16, 2019. Available at: http://eprints.gla.ac.uk/190780/. This extended abstract is a continuation of the work presented in (1.) and is expanded upon in Section 4.8 of this thesis.

3. Wallis, W., **Kavanagh, W. J.**, Miller, A. and Storer, T. (2020), Designing a mobile game to generate player data — lessons learned, in *Proceedings of GAME-ON 2020*, pp. 13–15. A short experience report on the development on the mobile game RPGLite and a critique of the processes employed. The specific *lessons* from this paper are given in Section 5.4. The published version of this paper is available at: https://eprints.gla.ac.uk/223600/ and an extended version is available at: https://arxiv.org/abs/2101.07144.

4. **Kavanagh, W. J.**, Miller, A. (2020), Gameplay Analysis With Verified Action Costs, in *Springer's Computer Games Journal*. Available at: https://link.springer.com/article/10.1007/s40869-020-00121-5#article-info. This paper includes the description of action-costs, how they are obtained from the model checking of RPGLite and how they can contextualise play analysis with measures of success per move. Chapter 7 is taken from this publication with light alterations to correlate more closely with the thesis.

In addition to the published research, the latter half of this thesis is centred around the mobile application RPGLite and the dataset collected from players.

- RPGLite the mobile application is available on iOS and Android platforms published by the University of Glasgow, developed by **William Kavanagh** and Tom Wallis. A host page is available at: http://RPGLite.app, with links to the store pages, a FAQ and contact information. The game is an extension of the case study described in Section 4.4, the peripheral systems in the application that support the game and incentivise play are detailed in Chapter 5 and the game itself is described in Section 6.2.

- **Kavanagh, W. J.**, Wallis, W. and Miller, A. (2020), RPGLite player data and lookup tables, available as a JSON document hosted by the University of Glasgow at: http://researchdata.gla.ac.uk/1070/. The dataset is used for the work carried out in both Chapter 6 and Chapter 7. A description of the database contents is given in Section 5.3.4

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Games should be fun to play. Irrespective of medium, genre or style, a game that is not fun to play will be unsuccessful. Competitive games, where multiple players play against each other, rely on the interplay between players for engagement. The game itself is a sandbox for players to inhabit. The boundaries can be defined and tools can be provided to be played with, but the way the game is actually played cannot be mandated by developers. In balancing a game, developers are ensuring that none of the possible ways of playing a game detract from the fun of any of its players.

Game balance is crucial. One needs only to visit the website of any popular game title or the social media of any developer to see the emotion players attach to their favourite game characters, weapons or cars and the outrage or elation that follows when they are changed even slightly. These changes, in the name of *balance*, are commonplace. With some games releasing balance patches every few weeks.

Competitive multiplayer games often constitute a short experience no more than an hour long, which players will gladly repeat over and over again. The interplay between the players and the various ways of playing the game are what make it interesting to play repeatedly. Two matches of the same game played back-to-back can be very different. This is because the games are designed in a way that allows a variety of styles of play. The aim of game balancing is to maintain the relationships between the styles of play, all of which exist in a fragile equilibrium. Developers must ensure as many of these ways of playing as possible are viable for success and that they are distinct from one another, to ensure that their games are *replayable*.

However, game balancing is notoriously difficult. Developers must identify all of the ways of playing a game, compare each to each-other and ensure all will be effective in certain situations whilst none are best in all situations. Just enumerating the ways in which a game can be played can be hugely complex. The most popular games have millions of players every day, exploring the boundaries of what is possible in the game and constantly coming up with new ways to play.

All of these play styles must be considered in conjunction with all others. Once an equilibrium is reached, if the game is updated with any form of expansion to the playable environment, then the entire game balancing process must start again.

Games are designed to be balanced, with ways of playing of roughly equivalent strength. When configuring game balance in competitive games there are several comparisons to consider: *asymmetry*, *materials* and *strategies*. Most games are asymmetric, one player has access to interactions that the other players do not, but all players should still have the same opportunities for success. The materials of a game are the player avatars and tools, these can all be numerically described. Ensuring that all the materials of a game are good enough to be used sometimes and none so good as to always be used is a form of game balancing. Similar to materials, the strategies available to players have to be considered for dominance, this is especially difficult as first these strategies have to be identified, then enumerated. The way that games can be balanced differs from game to game. In some instances materials have a cost to acquire, this can easily be used to off-set stronger material with higher costs. Games with a power-to-cost ratio are common and the balancing of these games is performed similarly, many adopting the "mana curve" [1], coined by developers of Magic: The Gathering (Wizards of the Coast, 1993) [https://magic.wizards.com/en]. The "mana curve" is a function of strength to value to which all material should adhere to some degree. In other games, where material is not accrued throughout the game's play, more than simply adjusting the numbers is required. When games can be played in very different ways that cannot be compared in a singular sense, balancing then requires new calculations specific to the game. For example, where game material comes with different victory conditions. Consider Dead By Daylight (Behaviour Interactive, 2016) [https://deadbydaylight.com/en] where 4 "survivors" attempt to escape from a single "killer" who in turn attempts to capture and sacrifice the survivors. In trying to balance this game there are several issues that must be addressed including:

- How can the strength of a single survivor, in terms of movement speed, ability to fight back and navigation skills compare to the lethality of the various killers and their abilities?
- How much additional utility does the teamwork of the survivors provide?
- To what extent does map design favour one side or the other?

As these questions demonstrates balancing concerns are varied and specific.

Diplomacy is a popular board game first published in 1959 that is still widely played to date. In it, 7 players take the roles of the "Great Powers of Europe" in Spring of 1901 and take turns trying to expand their empires to cover enough of Europe to be declared the winner. Inspired by the historical context, Russia starts with 4 capitals whilst the other powers start with only 3. This should give the Russia player a distinct advantage, yet Russia wins no more often than any other nation. Having 4 of the 18 territories needed to win rather than 3 is not enough of an advantage to

affect Russia's probability of winning, and in having a small lead in turn 1 they are seen as a legitimate target for the other players. Similarly, the Tom Clancy's Splinter Cell series of first-person shooters includes a Spies vs Mercs mode in 4 games, where teams of powerful mercenaries face off against far weaker secret agents. The pay-off for the secret agents being weaker is their ability to sneak around a map, where the majority of maps are shrouded in darkness and are designed to help the spies in stealthily navigating to elude the mercenaries. These asymmetric designs lead to interesting gameplay, despite factors seemingly tip the balance in favour of some of the players. The only way to judge whether or not they constitute balanced experiences is to see how the games are actually played.

A game will only be played repeatedly if players find it *interesting* and *fun*. However, since these are subjective qualities, how can the developer ensure that the systems they design are either of these to all players? The short answer is that they cannot. Playing a game is an emotive experience, no amount of mathematical analysis can solve the game balancing problem without considering how the game is actually played. In the seminal game design textbook "The Art of Game Design: A Book of Lenses" [2], author Jessie Schell states:

> *"Balancing a game is far from a science; in fact, despite the simple mathematics that is often involved, it is generally considered the most artful part of game design."*

No game balancing attempts can be successful without addressing the subjectivity of the player. Indeed prominent game developer Jeff Kaplan, the leader designer of the game Overwatch (Blizzard, 2016), famously proclaimed:

> *"The perception of balance is more powerful than balance itself."*

in a blog post of 2017. Games can be vastly complex systems which even the developers cannot consider the entirety of, let alone the players. When considered in conjunction with human bias this can lead to considerable misconceptions of a game's balance. Some of the best examples of this come from the game League of Legends (Riot, 2009). The following quote is from a developer describing their experience of *nerfing* (making worse) the character Vladimir from a developer blog:

> *I submitted a Vladimir nerf,* ***but forgot to actually submit the files.*** *As a result the patch notes went out and sentiment was that we had killed the champion. Vladimir's play rate plummeted and his win rate decreased a bit, even though the changes never went out.*

Here playable game material was not changed, but the playerbase were told that it did. This lead to players believing the material had been ruined by the developers. There are other examples from the same game of this happening in reverse, a supposed *buff* (making a character better) was announced, but not implemented, and the playerbase reported how great the character now felt to play. A final example comes from the beta testing of the game Wolfenstein: Enemy Territory (Gray Matter, 2003). This game is a team-based first-person shooter where the Axis and Allies

face each other in a alternative 1950s history. Players reported how the Axis' machine gun, the MP40, was far worse than the Allies' counterpart, the Thompson. They claimed the Thompson did more damage with a slower fire-rate. The gameplay data backed their claims up too, players did better using the Thompson. However, the game was designed with all weapons having a mechanically identical counterpart for the other team, the Thompson and MP40 were identical in all but visual and audio design. The misconceptions were eventually traced back to the "bassier" audio for the Thompson, which was reduced, leading to a converging of the performance by players using either gun, thereby *balancing* the game.

The contemporary style is for multiplayer video games to be supported with additional content and adjusting of old content, long after their release. Greg Street, a senior game designer gave a talk on balancing League of Legends at the Game Developers Conference in 2017 where he described game balance as a "contract with the player". In a game where the stated aim is for skill to matter, he explains that the aim is never for "the state of balance to trump skill". By this he means that the primary factor in the outcome of a game should be whichever team is better at the game, rather than whichever is exploiting the powerful material. He goes on to explain that an imbalanced game is easily solvable, a solvable game is boring and people do not play boring games. Games which offer players a variety of material and strategies, but where only some are ubiquitous and others ignored, are likely imbalanced. This phenomena is common. The aim of game balancing is to prevent it from ever happening.

1.2 Approach

Techniques for game balancing are highly specific to the game being developed. There is no general best-practice, developers must simply test the game, tweak it, test it some more, tweak it again, and so on. To non-game developers it seems strange that there is not some school of mathematics designed purposely for game balance. Another often cited game design textbook, "Andrew Rollings and Ernest Adams on Game Design" [3] touches on the possibility for this:

> *"It's possible that one day there might be some sort of "game calculus" invented to handle these problems, but we're not going to hold our collective breath. Besides, that still doesn't solve the problem of how to break down the game into a list of strategies and variables to fit into the equations."*

In this thesis we attempt to do just that, or to at least provide some foundations. We consider simple games, designed to be easily changed, and use formal verification techniques rarely employed for game design beyond academia.

We use model checking, an automated verification technique, to both model games and to reason about strategies. We are not the first to use model checking to reason about games, or even game balancing, but we go to a depth others have not and use model checking in a wider

variety of the stages of game development. With model checking we can explore all reachable states in a game and consider all feasible strategies, without having to enumerate them manually. Model checkers have their foundations in game theory, and as such there is some tie-in between their use and reasoning about *real* games (i.e. those played for fun).

To balance games we define objective properties of the features which we believe will lead to fun, replayable games. We then use the model checker to verify if these properties hold, or the extent to which they hold. We also use a process called strategy synthesis, possible through the use of model checking, to synthesise communities of game players playing a game over a long period of time. In this thesis we introduce the concept of *Chained Strategy Generation* (CSG) for this purpose. CSG involves the repeated synthesis of adversarial strategies in turn to mimic game communities and the shared understanding of the best ways of playing as they evolve over time. Starting with some known strategy, we repeatedly find the best way of playing against it, considering all playable material, before finding the best way of playing against the new best way of playing. This process is repeated until no better strategy can be found or a cycle of strategies is found.

From the model checking of games we can obtain objective truths about which actions are superior. These truths can be compared to player data to give quantitative measures of the accuracy of the moves made. This expands on the tools available for analysing player actions and behaviour, allowing us to reason about rates of learning, player types and the links between motivating factors and player skill.

One cannot evaluate the state of a game's balance without considering the way in which the game is played. To ensure that our approach is valid we developed a complete game and gathered data from hundreds of players across thousands of games. Model checking was central to the design and configuration of the game we developed, RPGLite. The design of the game itself is of value as, to ensure it is a good case study for competitive multiplayer game research, it includes several systems engineered to incentivise competitive play. The data acquired from players is thoroughly analysed to evaluate the balancing techniques we describe.

Furthermore, having used model checking to analyse RPGLite, we can calculate the *cost* associated with any action taken in the game. We expand upon the gameplay analysis of RPGLite with these cost measures to gain deeper insights into how it was played and perceived by players.

1.3 Thesis Overview

1.3.1 Reader Guide

This thesis brings together two seemingly unrelated areas of research in probabilistic mode checking and game development. We include a reader guide Figure 1.1 to show how the elements of both fit together and guide the reader per chapter on what areas are covered. Every content chap-

ter of this thesis (3-7) includes brief narration and a copy of the guide, with the concepts covered in that chapter highlighted.

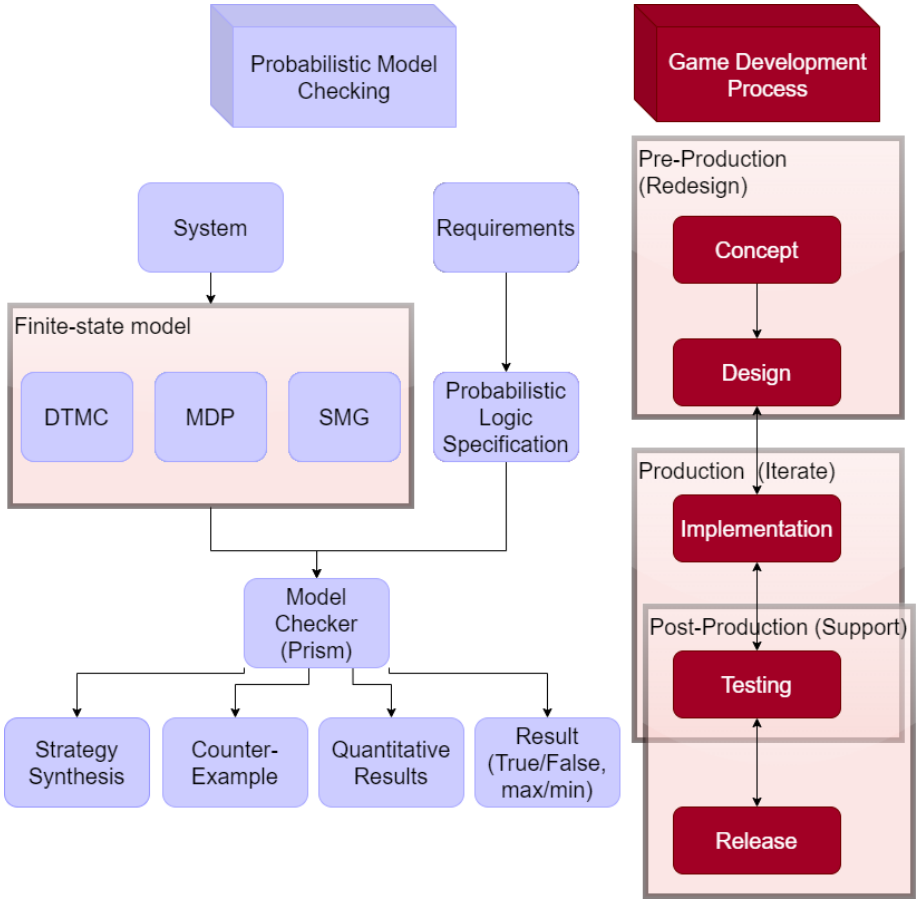

Figure 1.1: Thesis reader guide.

Model checking and game development can not easily be represented simultaneously. In our reader guide we represent probabilistic model checking based on the process outlined by the PRISM creators [4] and game development as a workflow through the three stages of pre-production, production and post-production. The 5 steps in the development process are representative of widely followed best-practice guidelines.

1.3.2 Contributions

The key contributions of this thesis are:

1. the advocacy of model checking, specifically of strategy synthesis performed with probabilistic model checking, as a game design tool which can be used at various stages in the development of a game to ensure balance;

2. the presentation of Chained Strategy Generation to synthesise an game-playing audience's understanding of how a game is best played which can in turn be used to reason about the state of game balance;

3. the creation of RPGLite and the dataset of the games, players and user logs, as well as a description of the ways in which competitive play is promoted;

4. the application of model checking methods to the balance and analysis of RPGLite;

5. the use of action-costs in player analysis to measure player learning and quantifying factors of success to ensure a balance between skill, luck and knowledge.

1.3.3 Thesis Structure

This thesis is composed of a further 7 chapters, which are outlined below.

2. **Literature Review.** Here we summarise the state-of-the-art in game balancing research and wider games research that is related to our methods and objectives and compare existing approaches to our own. We also review recent model checking work on similar areas, in particular, those using strategy synthesis.

3. **Preliminary Definitions and Results.** In this chapter we introduce the key technical concepts used in this thesis. This includes demonstrating the model checking process when applied to games and introducing the various types of models used. We also touch upon the core concepts of games and game playing which are needed to outline the aims of this thesis and introduce game-playing terms alongside the methods of play analysis used in commercial game development.

4. **Chained Strategy Generation.** This chapter introduces the first key contribution of the thesis, Chained Strategy Generation (CSG). We describe the aim and methodology behind the approach and the full algorithm, showing how it can be implemented using a simple case study RPGLite. The results of the algorithm performed on 5 configurations of RPGLite are compared and analysed to judge which results in the most balanced form of the game. We describe the use cases for CSG and how it can be expanded upon with similar techniques more suitable for games with larger pools of material. Building on the original implementation of CSG, we describe the nuances of representing game material and material selection and introduce optimality networks which enable developers to consider the relationships between game materials ensuring the desired cyclical nature.

5. **RPGLite, the Application.** Taking the case study of RPGLite in an expanded form, this chapter describes how it was used as the basis for a mobile game developed and published for this research. In this chapter we describe the design of the application, including the system architecture, design philosophy and processes. We give examples of typical paths through the application and the data that was collected, as well as the incentivisation and gamification systems employed in the application. The RPGLite dataset is a large, open

resource. Our hope is that it will prove useful for future research, even beyond game balancing. This chapter also includes an experience report of designing and releasing a mobile game as team of amateur developers.

6. **Balancing the Application.** In this chapter we use the techniques described in chapter 4 to analyse the balance of RPGLite. Alongside detailed player analysis, we compare predictions of play based on strategy synthesis with observed behaviours and statistics. We show how model checking was used to release a sufficiently balanced game and how insights from further automated analysis suggested a superior configuration of the game. The two configurations released are then compared. We introduces two baseline qualities of game balance with asymmetric material. These can be both verified using model checking prior to release, and demonstrated through gameplay analysis in line with current industry techniques. We also consider the metagame under both configurations and track the popularity of the materials over time.

7. **Gameplay Analysis with Verified Action Costs.** Here we introduce state and action lookup tables and the concept of action-costs. Action-costs can be used to quantify the value gained or lost with each action made in a game and so increase our ability to judge player ability. We re-examine the conclusions from chapter 6 through the lens of action-costs. With action-costs we expand upon standard forms of material comparison to include the competency with which different materials are employed by players and study the rate of player learning. We show how some players actually got worse as they played more games.

8. **Conclusions.** Finally we sum up the significance of this body of work. We discuss the value of our approach and how it can be advanced upon to have tangible impacts for game development at all levels. We analyse the limitations and the assumptions made. Finally, we describe feasible extensions stemming from the work presented in this thesis.

Chapter 2

Literature Review

In this chapter we highlight and discuss research relevant to this thesis. Our survey is divided into the following sections: research using model checking; model checking specific to games; game balancing research, and; research on play analysis. Of the model checking research considered, we focus on Prism [5], the model checker used in this thesis for reasons addressed later in this chapter, as well as research that employs strategy synthesis. Game-play analysis is surveyed as knowing how games are actually played is the only way game balance can be verified.

2.1 Model Checking

Model checking is an automated process used to verify that a mathematical system description satisfies a given property. A more in-depth description of model checking is given in Section 3.3 together with an explanation of how model checking was used for our work. Model checking is an expensive process. It requires finite-state models of a limited scale to run efficiently. In addition to this, it is a highly technical. Models rarely translate smoothly from the system being considered into a manageable description which a model checker can interpret. For these reasons, model checking is not a common commercial technique, except in safety critical contexts, where the costs of model checking are offset by the precision it can afford.

The model checking of software systems is common where the objective is to identify bugs, rather than to gain deeper understanding of the system. A survey on the state of model checking for software [6] concluded that, despite advances in model checking capability, it is best suited to the analysis of small portions of code, allowing programmers to focus on wider implementation concerns. This is the first clear link between game development and model checking. Commercial games have a reputation for being released with several bugs. This has lead to research on identifying and preventing these game bugs [7], [8]. Model checking has already been proposed to address this issue (see Section 2.2.1), but it is not well suited to encapsulating complex game systems that exhibit concurrency or erratic human behaviour.

Model checking has been used to verify aspects of safety critical systems since the 1990s.

For example, the model checkers NuSMV [9] and UPPAAL [10] have been used in the verification of logic controllers used in nuclear engineering [11]. Violated properties are demonstrated showing patterns of behaviour which lead to undesirable and dangerous states. Similarly, in [12], purpose-built statistical model checking tools were used to approximate probabilities of property satisfaction in hybrid systems, specifically, fault-tolerant fuel control systems. The approximation methods used traded accuracy for precision by not performing a full verification of the statespace. Repeated simulation as opposed to full verification is an effective technique for reducing the cost of model checking, but it must be used whilst remaining mindful of the trade-off. It is good for estimating upper and lower bounds, but does not provide an exhaustive analysis as there is only a probability that a statistical model checking tool is correct. In a game playing context, where the system is non-critical, this may be a suitable approach.

Model checking software is available for systems with various attributes across several domains. The properties of the system being modelled change the way in which they can be represented, requiring purpose-built model checking tools. For example, real-time systems can be model checked using timed automata [13] which can only be parsed by certain model checkers. Other features which require specialist model checking tools include concurrent systems, probabilistic systems and nondeterministic systems. Modern model checking systems such as Prism, Storm [14] and ePMC [15] are extensible. They can be run in a number of different configurations or have modular support for different purposes.

When modelling games we needed to model players as they made decisions and to capture the stochasticity that games exhibit, which can be used to model a variety of game effects. In its most simple form, the stochasticity in games may be a random event such as the roll of a die or the ordering of a deck of shuffled cards. More advanced uses for stochasticity include the representation of a player's dexterity as they attempt to perform difficult manual tasks such as taking a corner at high-speeds in a racing game or attempting to hit a distant, moving target in a first-person shooter. In these examples, the probability of success is a function of the player's skill and the difficulty of the task (and possibly of external factors such as opponent skill or network latency). Having surveyed several model checking tools, the one that we used for this work was Prism as it models probabilistic systems and supports strategy synthesis of multi-agent systems. Prism also has support for formal game models such as stochastic games [16].

2.1.1 Prism

Prism is a symbolic, probabilistic model checker first released in 2000. A full explanation of how we use Prism, with examples pertaining to games and game playing strategy, is given in Section 3.3.1. Prism is in continual development and has been extended multiple times, with recent advances supporting the model checking of stochastic games and concurrent systems culminating in work on concurrent stochastic games [17].

There is an important distinction to be made here about the use of the term 'games'. In a model

checking context, 'games' is used to refer to formal systems of competition and cooperation amongst rational agents. In the games research context, 'games' refers to the commercial games produced and played for fun, but also to serious games, those developed for a societal benefit such as vocational training or education. Modern games are systems of competition and cooperation populated by *rational agents* in the form of human players; albeit players rarely act entirely rationally. As such they can be mapped on to the formal definition given in classical game theory, although it is not a natural way of thinking about them. Theoretically speaking researchers are interested in perfect play, exhibited by rational agents. For game balancing we typically consider the highest level of play, which by definition is closest to optimal play, i.e.: the players considered at the most rational of those available.

The verification of classical games using Prism is possible for turn-based stochastic games, concurrent stochastic games and turn-based probabilistic timed games (an extension of stochastic games using real-valued clocks) [18]. We only consider turn-based stochastic games (TSGs) [19] in our work to limit complexity. Research that similarly uses verification of stochastic games includes reasoning about the need for explanation of semi-autonomous systems [20]. Self-adaptive systems that perform complex tasks are considered and model checking is used to identify when descriptions of decision-making should be given at the expense of time to aid human input. This is far apart from the aims of this thesis – developing games which are fun to play, yet the approach is similar.

One of Prism's key features is the ability to synthesise optimal strategies. Many model checkers can generate counter-examples when verifying properties. When those properties are probabilistic, the counter-example which maximises or minimises the satisfaction of a property is neatly mapped into the games research context as an optimal game-playing strategy. With multiple rational agents (i.e. in multiplayer games) the synthesis of strategies for an individual player is more complex to identify, but can be done through Prism and is used extensively in this thesis. Examples of this are given in Section 3.3.1. The generation of optimal strategies is akin to planning [21], a common use case for model checking.

Strategy synthesis can be useful in a wide variety of areas, for example, in aiding the devising of search-and-rescue patrol patterns [22]. This work is a study of multiple scenarios, all modelled as Markov decision processes (MDPs) [23] in Prism, with slight variations in how an autonomous drone operates. Using strategy synthesis they identify the best routes to explore a grid where various objects need to be identified and acted upon. The time taken for the objects to be dealt with, with optimal planning and naive planning are compared, as is expected the optimal controller performs much better, taking half as much time in some instances. The methods of this work are similar to the synthesis we perform on games where the strategies of all players except one are known. Another similarity is the use of real-world data to inform the transitions of the model and the reintegration of an optimal solution in later simulations. This thesis uses discrete time Markov chains (DTMCs) [24], MDPs and TSGs for various purposes.

2.2 Model Checking for Games

In this section we discuss the limited application of model checking for all stages of development of games, both analogue and digital.

2.2.1 Bug Detection

The detection of bugs in software systems is a classic application for model checking software. Video games have a reputation for being released with several bugs, some of which are later fixed through patches. The scale of video games is expanding as the hardware expands and an emphasis on exploration and emergent gameplay epitomised by player freedom leads to more possible scenarios and interactions in the game. The issue of bugs in games will persist, as will the need for procedures to identify them.

Work which closely resembles the research carried out for this thesis involves the probabilistic model checking of a small 2-player computer game, Penguin Clash [25]. Two-players control the 2D movement of two penguins on a small iceberg who can bump into one another and throw snowballs, the winner being the player who remains of the ice the longest. In this work the authors introduce *Safegame*, a tool which uses the NuSMV model checker [9] to verify desirable game properties. The verification that is performed is limited, one penguin is restricted from dying or throwing snowballs and the properties include verifying that these disabled states cannot be reached. Even for a small game like Penguin Clash and after considerable effort to limit the state space, the model sizes were in the order of 10^9 states and verification took over 2 hours.

Model checking has been suggested as a tool for successful, commercial games in [26], where a procedure for identifying bugs in Fallout 3 (Bethesda, 2008) is detailed. Two known bugs from the game are used as examples, one where a companion dies and the player cannot replace them and another where dialogue options can be exploited for unlimited experience point rewards. This work takes known bugs and generates simplified examples of XML diagrams which can be translated into ProMeLa, the language interpreted by the SPIN model checker [27]. This work outlines an approach only and does not actually verify any game properties, but the authors argue that the potential benefits warrant further investigation.

Game development is more accessible today due to the abundance of dedicated game development engines. Prominent examples of these are Unity3D and the Unreal Engine. All software development is prone to bugs and techniques for ensuring quality of work, which could easily be used in the games industry or adhered to more strictly, are well-established. Where model checking could be beneficial to game development is through the development of specialist model checking software akin to Java Pathfinder [28] for these game-specific tools. The similarity of bugs found in video games could be exploited by model checkers designed to efficiently identify them, however that is not an avenue explored in this thesis.

2.2.2 Design Analysis

Model checking has wider uses than identifying issues in implementation. By expanding the systems being modelled to incorporate some representation of the users model checking can be a tool for analysing larger design questions. This is the use of model checking which is employed in this thesis. Modelling player behaviour and solving games through model checking to ensure games are fun to play. Other research has been carried out of this nature.

Model checking of games to verify that varied strategies can be employed successfully, and for other measures of *replayability*, has been performed with board game based examples in [29]. Here the authors use 3 examples and the model checker Prism to reason about various features of these games. The case study which most closely resembles the work presented in this thesis is a simplified version of the classic board game Risk where the 42 territories in the original for up to 6 players are reduced to a 6-vertex graph navigated by only 2 players. In [29], a parameter determining the aggression of the players is used and different combinations of these values for either player are considered. The aim being to try and identify if a single optimal way of playing exists. Although the authors use Prism, they do not perform verification for this example, instead they perform repeated simulations for each property and take an average. This approach is used because it is much faster than actual verification. From their results, the authors surmise that the optimal strategy [30] is to vary aggression levels, becoming aggressive after a number of rounds, but that no single answer for how many rounds to wait is best. They conclude from this that it does not appear that an optimal strategy exists. A significant portion of this thesis is focused on identifying optimal strategies using verification. A detailed explanation of what we mean by optimal strategy is given in Section 3.2.4. Properties of balance in board game scenarios can be *indicated* using model checking [29], this differs from the objective truth that we strive for. Furthermore, our work takes insights from model checking and uses them to redesign and rebalance a game.

Model checking as part of the game design process has been used in the <e-Adventure> framework [31], a tool for developing educational adventure games. Game properties are translated into a labelled transition system and verified using Computation Tree Logic (CTL) in the NuSMV [9] model checker. The authors give an example of a game for teaching correct medical procedure and describe how the built-in verification of the engine can take properties and generate animated counterexamples of play should they exist. The benefits of this approach are primarily to prevent bugs, but unlike the examples from the previous section, this approach uses model checking during the design process, rather than model checking a designed game.

2.3 Game Balancing

A recent survey on the ontology of "game balancing" [32] from fourteen leading voices in games research and development concluded that no singular definition exists and further research would

be required to reach one. Where there was consensus among those surveyed was on the aim of game balancing being the design goal of "fun", and in particular, of avoiding anti-fun scenarios in games. Beyond that however, the disparity of concepts used by the researchers demonstrates how nebulous game balancing is. Balance for both single player and multiplayer games are included in the survey and not fully disaggregated. Some terms are used by multiple sources referring to different ideas, such as *exploit* which is used in the explore/exploit strategy paradigm and game exploit being a player-identified bug which can be used to gain an advantage. Several other terms refer to loosely defined ideas, like *fairness* and *meaningfulness*, which will mean different things to different players and in different genres of game. Metagames too are a concept often used, but with multiple definitions, as discussed in [33]. Singular definitions of the key terms would support game balancing as a research field to evolve efficiently. It is also why this thesis requires substantial preliminaries. If new definitions are agreed upon at some point in the future, this work will be able to be mapped onto them.

Automated game balancing can be performed in a number of different ways. Commonly it is considered to be an optimisation problem, where versions of games are judged to be more or less balanced than one another and the most balanced version is sought. There are two hurdles to this approach. First, the judgement of game balance is hard for the reasons of subjectivity and varying definitions as described earlier. Also, small changes in games rarely have predictable effects on the balance of the full game. In our work we show how making even minor changes to characters' attributes can cause significant changes in the relationships between all the characters when considered together. It is very difficult to say with certainty that one form of a game is *more balanced* than another, only that a game is *sufficiently balanced* if it is enjoyable to play. For this reason the optimisation approach to automated multiplayer balancing is ill-suited.

A co-evolutionary algorithmic approach to balancing is introduced in [34]. The game studied is a 2D two-player game where players control ships equipped with tractor beams which are raced to capture crates. Three distinct high-level strategies are described as 'crate running', 'territory grabbing' and 'crate stealing', and the balance condition is narrowly defined as solely the absence of a dominant strategy. From the visualisation of evolved strategies the authors refine the time taken to perform actions as well as which actions are allowed (e.g. in later iterations of the game, crates could not be stolen). Analysis of later iterations shows that the evolved agents favour varying strategies rather than sticking with any of the three outlined, which the authors claim is proof that the game was sufficiently balanced. Similarly Top-Trumps inspired games have been balanced through evolutionary single-objective [35] and multi-objective [36] optimisation. Both adapt the decks used in the games, but only [35] includes a player survey on whether or not the game was enjoyable, where-as [36] includes objectives which the authors believe will lead to 'exciting' gameplay.

Another approach is the use of deep player behaviour modelling presented in [37]. The behaviour of a set of over two hundred players of the massively multiplayer game Aion (NCSOFT,

2008) is used, players are tracked daily over 6 months. This work measures the proficiency of players across the multiple available in-game 'classes', whilst attempting to mitigate the effects on results of the classes' intended design. This is a common issue seen in development and analysis of game materials, in this instance the different classes are intended to excel in different scenarios, but not in all. For example, in [37] the support characters of the *Cleric* and *Bard* are found to perform less well when considered individually, which is to be expected as their design is specifically to *support* other characters rather than excel alone. This approach to automated game balancing, starting from player data rather than synthesising it, is potentially very effective as it should allow for more specific targeting of the subjective qualities of game balance and future work in this area is certainly warranted. However it requires either an established game with numerous (in the order of tens of thousands) active players, perhaps beta testers or the balancing of released games, or it requires QA testing, which is often highly expensive.

A similar approach as that in this thesis of using synthesised play, is used to study ancient games [38]. Games are described using the LUDII [39] system, a purpose-built strategic game description language, and Monte Carlo tree search [40] (MCTS) is used to generate strategies through multi-agent self-play. MCTS is the state of the art for game-playing AI. The work on ancient games requires strategies which are "just above average human play", for which MCTS is well suited. Strategies at this level could be used to inform decisions on game balancing, but they would not model the highest level of play and are not guaranteed to find ways of playing which might be 'anti-fun'. A similarity between this and the work in this thesis is that particular and constrained games are modelled in both in order to give insights into all games. They are trying to build out the genealogy of strategy games going back to the earliest recorded games, we compare actual and optimal play to consider the intricacies of human play.

A related field to game balancing is AI for game-playing. All efforts for automated game balancing require observation of a game being played and for that some game-playing AI must be created. The use of approximation techniques, where true Nash equilibria are not needed only suitably close values, can overcome complexity issue inherent in solving complex systems. For example, mathematical games with thousands of players can be modelled and solutions approximated [41], given there is suitable symmetry in the roles.

2.4 Play Analysis

From the research on game balancing, it can be concluded that the objective of game balancing is to produce a game that is fun to play. Our work on game balancing therefore is validated only by balancing a real game and analysing player data. In this thesis we do just that, and in analysing game data we introduce novel techniques that employ model checking results gained from the automated balancing process incorporated in initial design. These techniques, specifically the use of action-costs Section 7.2, are described in full. The analysis of player data is integral

to any game balancing effort and as such we survey selected research on play analysis and the utilisation of the insights gathered into redesign and reconfiguration.

Game developers have access to vast collections of player data, the analysis of which should confirm whether their games are balanced. In [42], developers of Candy Crush Saga and Candy Crush Soda Saga (King, 2012, 2014), popular match-3 games, train a convolutional neural network to play proposed new content in a *human* way. The approach is highly successful. The method of training agents is significantly more accurate than state-of-the-art MCTS. The agents can then be used by the developers to predict difficulty of new game levels in under a minute. The objectives of this approach are similar to those which inspire our chained strategy generation technique Chapter 4. Specifically, to synthesise human play, use it to model gameplay and then judge if new content will be successful. Our research focuses on multiplayer games however whilst this work looks at single-player games. Additionally the training of the neural network requires large amounts of player data on the game before development, whereas we model play without any data at all.

Even chess is the subject of game balancing research with cutting edge chess playing AI used to inform potential alterations of the game. In chess, all non-pawn pieces are ascribed values in terms of the equivalent in pawns. This is used to assess position strength and can be used by players to decide if certain move sequences are beneficial. These values are reductive as material value is dynamic, depending on position and board state. For example, the rooks offer little value at the beginning of a game as they cannot move before several others pieces are moved to accommodate them, but rooks are considered more valuable than bishops and knights in all valuations. With recent advancements in chess-playing AI these values have become more specific as the automated decision making processes rely on them. The oldest valuation known comes from the Modenese school [43] in the 18^{th} century, since then several amendments have been suggested. Originally Knights and Bishops were considered 3 times as valuable as pawns, Rooks 5 times as value and a Queen 9 times as valuable. American grand master Bobby Fischer in 1982 suggested Bishops should be valued slightly higher than Knights [44]. DeepMind's AlphaZero, a cutting edge solver that identifies near-optimal strategies through learning, is the current best known chess-playing algorithm. It evaluates the material differently, albeit similarly to Fischer. The values from AlphaZero have been suggested as a starting point for balancing the rules of chess [45], specifically for inventing asymmetric variations which should offer players an even probability of winning. These valuations give an insight into the perception of comparative game material strength. In Chapter 7 we introduce the concept of *action costs*, which can be used in a similar way to chess piece values. We use these costs to analyse balance and learn about player skill.

Chapter 3

Preliminary Definitions and Results

"In which the results of, and processes for, model checking games are outlined and the key ludological terminology is introduced."

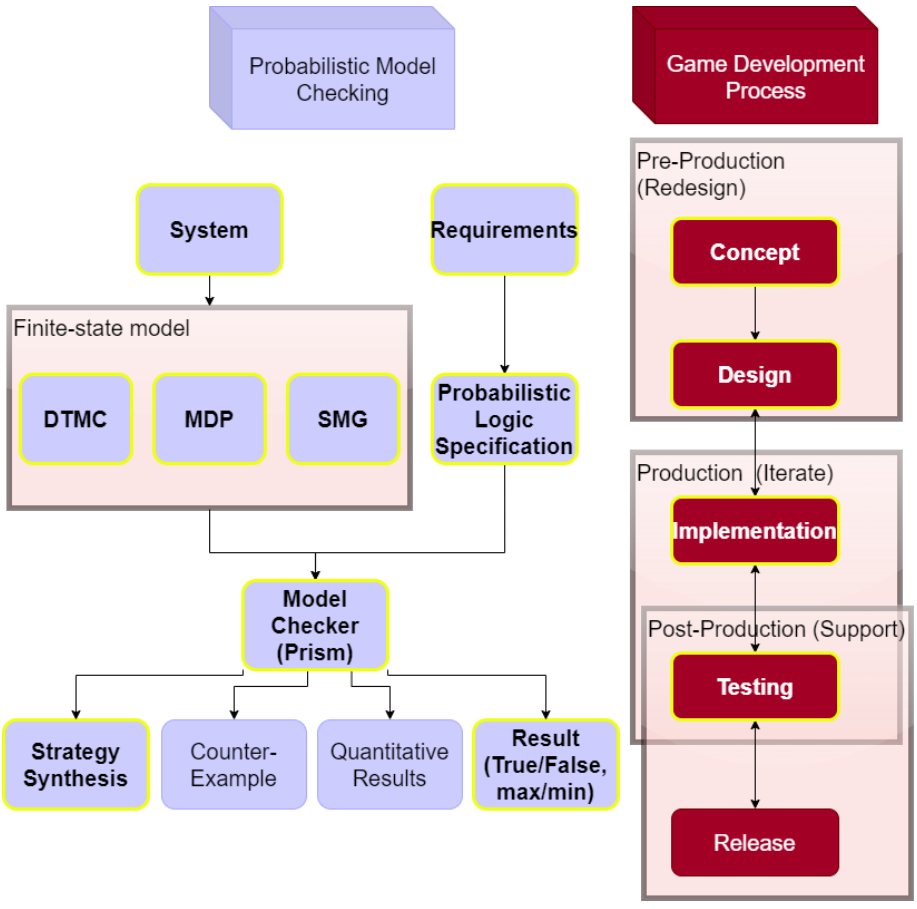

Figure 3.1: Chapter 3 areas.

3.1 Introduction

In this chapter we introduce the techniques and definitions used in our work and discuss the intricacies of games. We begin with definitions and examples of discrete time models and briefly explain how we will apply them to games. We then describe the model checker PRISM, including its extension PRISM-Games, and show how it can be used to model games and to generate player strategies. We do not discuss the intricacies of the mechanics of PRISM, but instead concentrate on how it can be used. Specifically, we show how PRISM can be used to reason about simple games, to synthesise strategies and to verify the existence of dominant strategies. Finally we describe the aspects of games that should be considered when they are modelled. These includes game material and strategies, as well as player specific details such as motivation and skill.

3.2 Models

We use discrete time Markov chain variants to represent turn-based games. There are three models which we use. First we introduce Discrete Time Markov Chains (DTMCs), then Markov Decision Processes (MDPs) and finally Stochastic Multiplayer Games (SMGs).

The difference between the models that we use is the level of non-determinism they admit. To demonstrate this we use a single example throughout, a game called *Twenty-One*, which is a 2-player variation of the game Nim. In Twenty-one players take turns adding 1, 2 or 3 to a shared total that starts at 0. The losing player is the player who is *forced* to make the total 21 (or higher). Practically, the losing player is therefore whoever has their turn after the player to bring the total to 20. For our examples, we label the two players “Red” and “Blue” and assume Red starts the game. There is a winning strategy for the player who does not start the game, i.e. Blue say. More precisely, if on their turn, Blue repeatedly adds a value that brings the total to a multiple of 4, then Red cannot stop them from reaching 20 on their fifth turn. This winning strategy is always available to Blue as they can always increase the total by 4 minus what Red last increased it by.

All models used build upon the concept of labelled transition systems (LTS), which consist of a set of states and a set of transitions between the states. A state represents a feasible assignment of values to all of the variables in the system. From a game playing perspective, this will encapsulate the condition of player material as well as environmental variables such as time, turn or level. Of these states, only some will be *reachable*, situations that the players could actually find themselves in. Consider a simple racing game, the state may be described by the speed, direction and position of each car along with the time since the race began and the current racecourse. A more complex game may include information on each car’s acceleration, the number of laps completed, even the condition of their tyres. A state where multiple cars have identical positions or the race has just started and cars are already moving at high speed would not be reachable. As games get more complex the information required to describe a state increases, as do the number

of reachable states.

In Twenty-one the state is represented by a tuple $(total, turn)$, where $total \in [0...20]$ denotes the current score and $turn \in [R, B]$ indicates whether it is Red or Blue's turn to give a number. As a player wins when they add to the total to reach 20, total values over 20 are omitted, thus the winning state for Red is $(20, B)$ and similarly for Blue the winning state is $(20, R)$.

3.2.1 Discrete Time Markov Chains

A DTMC is an LTS where the set of transitions is replaced by a probability transition matrix.

Definition 1. *A **DTMC** D is a tuple $D = (S, s_0, P, L)$ where:*

- *S is a finite set of states;*
- *$s_0 \in S$ is an initial state;*
- *$\boldsymbol{P}$ is a transition probability matrix $S \times S \to [0, 1]$ such that $\sum_{s' \in S} \boldsymbol{P}(s, s') = 1$ for all $s \in S$;*
- *L is a state-labelling function $S \to 2^{AP}$ assigning a set of atomic propositions to each state from the set of all propositions AP.*

For any $s, s' \in S$, the value $\mathbf{P}(s, s') \in [0, 1]$ is the probability of transitioning to s' when in state s, and therefore we require the sum of probabilities from one state to all others is always 1. If $\mathbf{P}(s, s') = 0$ then no transition exists from state s to state s'. The use of atomic propositions is to label states of interest, for example, the proposition that Blue has won the game.

Models, such as DTMCs, can be used to capture the entire state-space of a game, showing every possible position the players could find themselves in and the moves that can be made. To refer to a single *play* of a game, we refer to paths. Paths give an account of the moves made by players in a game and the outcome of the moves, e.g. a players move may be to roll a dice, but the outcome would be a specific value.

Definition 2. *A **path** (or trace) through a DTMC is a non-empty, possibly infinite sequence of states $s_0, s_1, s_2, \ldots$ such that $\boldsymbol{P}(s_k, s_{k+1}) > 0$ for all $k \geq 0$.*

We use a DTMC to model a game when the strategies of all players are known. A DTMC can be examined to give the probability of any player winning through summing the probabilities along all paths that lead to a winning state for a player. Winning states are identified by the value of a boolean value in their label and are terminal, i.e.: there are no transitions from a winning state (as the game is now over). It is possible for multiple states to be a winning state for a player. With known strategies for all opposing players we can consider multiple candidate strategies for a player and determine which is superior by constructing the DTMC for that strategy against the known opposing strategies and determining for which DTMC the probability of reaching a winning state is greatest for the player being considered.

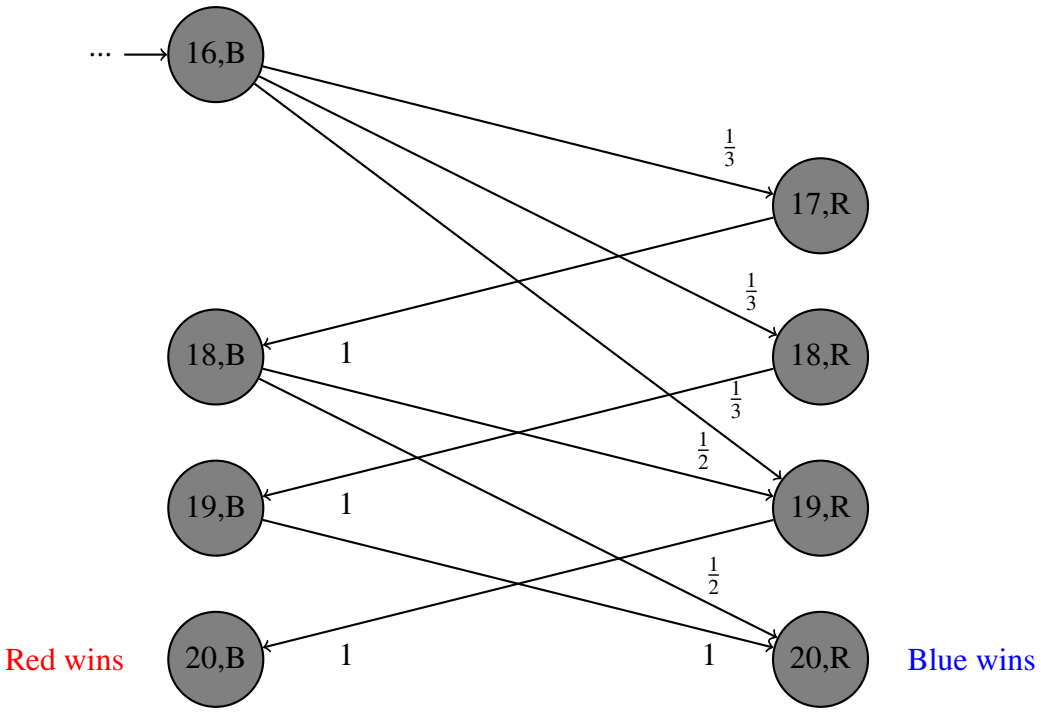

Figure 3.2: A DTMC representation of the last moves of a game of Twenty-one.

As an example, consider a game of Twenty-one where Blue uses a fair die to decide by how much to increase the total and Red always increases the total by just 1. A DTMC representing the late stages of a game of Twenty-one where Blue has just increased the total to 16 is given in Figure 3.2. There are 2 paths by which Red can win: *{(16,B), (17,R), (18,B), (19,R), (20,B)}* and *{(16,B), (19,R), (20,B)}*. The probability of Red winning from state $(16, B)$ is the sum of the products of the probabilities along these paths: $(\frac{1}{3} \cdot 1 \cdot \frac{1}{2} \cdot 1) + (\frac{1}{3} \cdot 1) = \frac{1}{2}$.

3.2.2 Markov Decision Processes

An MDP is an extension of a DTMC that includes non-deterministic behaviour. Before we give the definition we require the notion of a probability distribution. For a finite set X a probability distribution of X is a function $\mu : X \to [0, 1]$ such that $\sum_{x \in X} \mu(x) = 1$ and we let $Dist(X)$ denote the set of distributions over X.

Definition 3. *An **MDP** M is a tuple $M = (S, s_0, A, P, L)$ where:*

- *S is a finite set of states;*
- *$s_0 \in S$ is an initial state;*
- *A is a finite set of actions;*
- *$P : (S \times A) \to Dist(S)$ is a partial transition probabilistic function;*

- *L is a state-labelling function $S \to 2^{AP}$ assigning a set of atomic propositions to each state from the set of all propositions AP.*

Since $\mathbf{P}$ is a partial function, not all actions are available at all states, we denote by $A(s)$ the set of actions available at state s, i.e. the actions a for which $\mathbf{P}(s,a)$ is defined. For any states s, s' and action $a \in A(s)$, $\mathbf{P}(s,a)(s')$ gives the probability of transitioning to state s' if action a is chosen in state s.

We use MDPs to model games when we know strategies for all players except one, using the nondeterminism to represent the unknown strategy. Is is standard to refer to a solution to the nondeterminism, giving a distribution over $A(s)$ for each state where $|A(s)| > 1$ as a strategy or an adversary. We use the term **adversary**, to distinguish this from the notion of strategy in games. There is a strong overlap between the two, as we will see later in this thesis. With an MDP, adversaries which maximise or minimise the probability of reaching some set of states can be identified. In a game context, we use this to calculate the best strategy to use against known opposing strategies. Since for a fixed adversary an MDP reduces to a DTMC, the probability of reaching the associated set of states can be calculated for a strategy as before.

Definition 4. *A **path** through an MDP is a non-empty, possibly infinite sequence of states and action transitions $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \cdots$ where $a_k \in A(s_k)$ and $P(s_k, a_k)(s_{k+1}) > 0$ for all $k \geq 0$.*

As an example, consider Twenty-one where Blue uses the strategy of choosing randomly from all available options, but Red's strategy has not be fixed. This is illustrated in Figure 3.3 with black arrows representing the probabilistic choices under Blue's strategy and red arrows denoting the choices available to Red in each state. In general, each non-deterministic choice has an associated probabilistic distribution over consequent states, however in this example the probabilistic choice is not evident as there is only one possible resulting state in each case. The actions available to Red are to add 1, 2 or 3: {"R+1", "R+2", "R+3"}. As *total* is constrained to ≤ 20, not all actions are available in all Red states. There are two states from which $|A(s)| > 1$, $\{(17, R), (18, R)\}$. Replacing the non-deterministic choice over the available actions from each of these states, with a probabilistic choice, is analogous to a strategy for Red.

The best strategy for Red in this example is clear as from each state where Red has an action to take they can win the game, so the optimal strategy for Red is to choose, with probability 1, the action "R+3" in state $(17, R)$ and the action *"R+2"* in state $(18, R)$. An action choice for state $(19, R)$ is not needed to describe a strategy as only one action is available in this state.

3.2.3 Stochastic Multiplayer Games

SMGs differ from MDPs in that they can have multiple sources for nondeterminism, these sources are the players of the SMG.

Definition 5. *An **SMG** G is a tuple $M = (I, S, (S_i)_{i \in I}, s_0, A, P, L)$ where:*

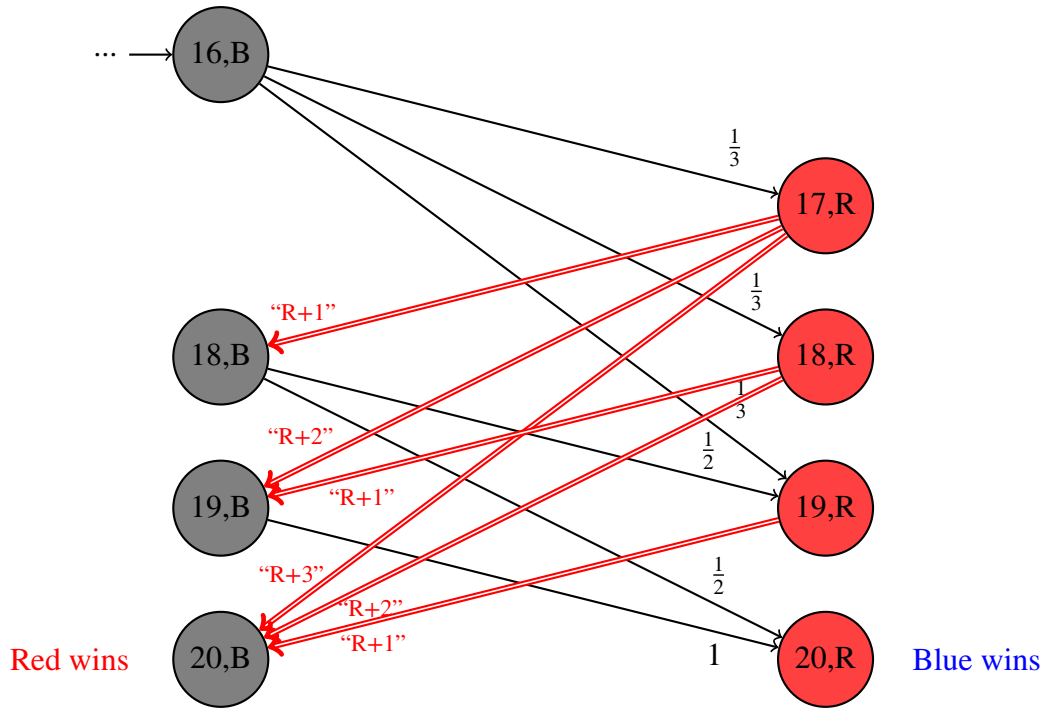

Figure 3.3: An MDP representation of the last moves of a game of Twenty-one where red states indicate states where Red must choose an action.

- I *is the non-empty, finite set of players;*
- S *is a finite set of states;*
- $(S_i)_{i \in I}$ *is partitioned of* S*, i.e.* $\cup_{i \in I} S_i = S$ *and* $S_i \cap S_j = \varnothing$ *for all* $i \neq j$*;*
- $s_0 \in S$ *is an initial state;*
- A *is a finite set of actions;*
- $\boldsymbol{P} : (S \times A) \to Dist(S)$ *is a partial transition probabilistic function;*
- L *is a state-labelling function* $S \to 2^{AP}$ *assigning a set of atomic propositions to each state from the set of all propositions* $\boldsymbol{AP}$*.*

A state $s \in S_i$, is an **action state** for player i, i.e. they are due to make an action decision from $A(s)$, the set of actions available at the state.

We use SMGs to model games when we do not know the strategies of more than one player. We restrict attention to modelling zero-sum two-player games ($|I| = 2$) in which when one player wins the other loses and it is not possible to draw. For such games we can use a SMG to synthesise the **optimal strategy** for each player. An optimal strategy for a player of a two-player game is a strategy which yields the maximum probability of reaching a winning state for the player against the best opposing strategy of the other player, this is also known as the minimax strategy.

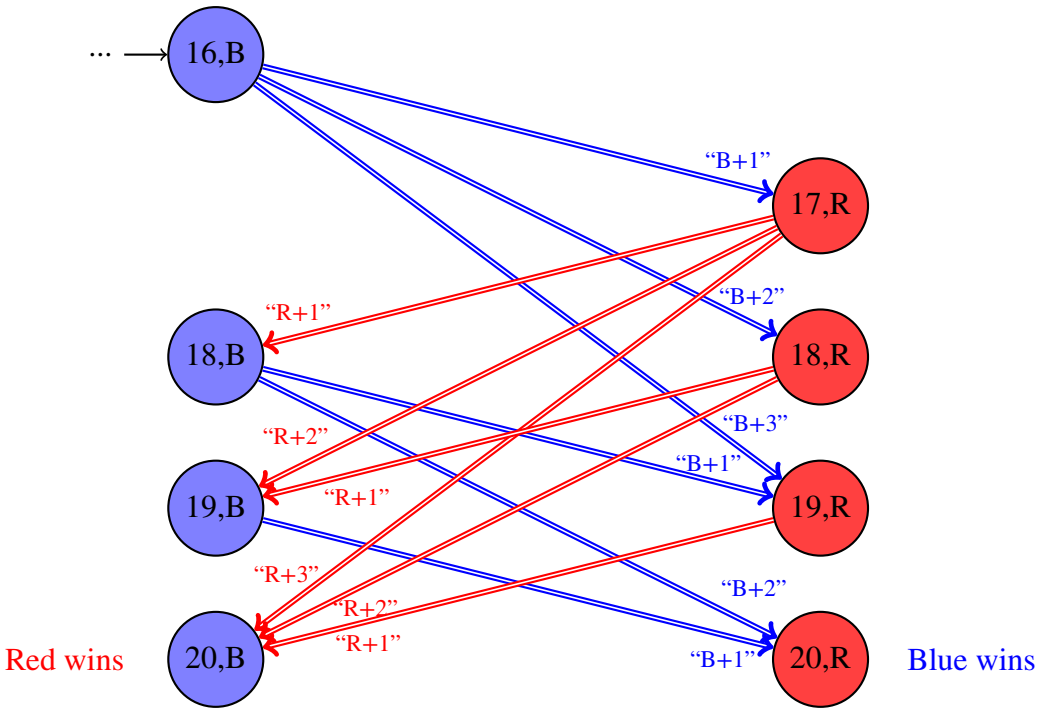

Figure 3.4: An SMG representation of the last moves of a game of Twenty-one where state colour represents the player whose turn it is.

It is known that optimal strategies exist for both players in two player zero-sum games of perfect information (where players know the current state) with sequential moves, by the minimax theorem [46], [47]. The generation and application of optimal strategies is central to our work. One cannot calculate the adversary of an SMG with multiple sources of nondeterminism in the same way as for an MDP, this would be the equivalent of one player playing optimally whilst all others intentionally lose. Instead adversaries are calculated for a coalition of players or a single player where all non-coalition players seek to minimise their pay-off – to reduce the coalition's maximum probability of winning.

An SMG description of the latter stages of a game of Twenty-one is given in Figure 3.4. The position described by the initial state, Blue has to add to the total of 16. The optimal strategy for Red in this SMG is equivalent to the adversary calculated in the MDP example. The probability of winning for this strategy is 1.0 as there are no actions Blue can take to prevent Red from winning should Red play perfectly.

3.2.4 Further Definitions of Strategies

To better describe the processes we use with different modelling formalisms used for a single game, we need definitions for what constitutes a strategy in these forms. A path of an SMG is a sequence $\omega = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \cdots$ such that $\mathbf{P}(s_k, a_k)(s_{k+1})>0$ for all $k\geq 0$. Let $\omega(k)$ denote the kth state of the path ω. A **strategy** for player i is a way of resolving the action choices of the player, based on the game's execution so far. More precisely, a strategy for player i is a function

σ_i which maps each finite path with a last state that is controlled by player i to distributions over actions available in this last state. The **set of strategies** of player i is denoted Σ_i. A strategy is **deterministic** if it always selects actions with probability 1, and **memoryless** if it makes the same choice for all paths that end in the same state. A Markov decision process (MDP) is a single-player SMG; in this case we denote the set of strategies for the single player by Σ.

A **strategy profile** for an SMG has the form $\sigma=\langle\sigma_i\rangle_{i\in I}$ listing a strategy for each player. We use $IPaths_s^\sigma$ for the set of infinite paths corresponding to the choices of the profile σ when starting in state s. For a profile σ and starting state s, the behaviour of an SMG is fully probabilistic and can be modelled as a DTMC. We can also define a probability measure $Prob_s^\sigma$ over the set of infinite paths $IPaths_s^\sigma$ [48]. A fundamental property of SMGs is the probability of reaching a target set. For SMG G, profile σ and set of target states T, the probability of reaching T from the state s under profile σ is given by:

$$\mathbb{P}_G^\sigma(s,T) = Prob_s^\sigma\{\omega \in IPaths_s^\sigma \mid \omega(k) \in T \text{ for some } k \in \mathbb{N}\}$$

For a two-player SMG G and target set T we assume the game is zero-sum, i.e., player 1 tries to maximise the probability of reaching T and player 2 tries to minimise it. In this zero-sum setting, an optimal strategy $\sigma_1^\star$ for player 1 in state s of G is a strategy that maximises the probability of reaching T no matter the strategy of player 2, formally we have:

$$\inf_{\sigma_2\in\Sigma_2} \mathbb{P}_G^{\sigma_1^\star,\sigma_2}(s,T) = \sup_{\sigma_1\in\Sigma_1} \inf_{\sigma_2\in\Sigma_2} \mathbb{P}_G^{\sigma_1,\sigma_2}(s,T)$$

An optimal strategy for an MDP M in state s for reaching a target set F, is a strategy $\sigma^\star$ that maximising the probability of reaching the target:

$$\mathbb{P}_{\mathsf{M}}^{\sigma^\star}(s,T) = \sup_{\sigma\in\Sigma} \mathbb{P}_{\mathsf{M}}^{\sigma}(s,F)\,.$$

For both two-player SMGs and MDPs there exist memoryless deterministic optimal strategies. Efficient algorithms exist for generating such optimal strategies and the corresponding optimal probabilities [49], [50] which have been implemented in the PRISM model checker.

For a two-player game G and memoryless strategy σ_2 for player 2, we can construct an MDP $\mathsf{M}_{\sigma_2}^G$ where the choices of player 2 are resolved according to σ_2. The optimal strategy of $\mathsf{M}_{\sigma_2}^G$ is the most effective *counter* to the strategy σ_2, and therefore we refer to this optimal strategy as the *adversarial strategy* to σ_2 as it is only locally optimal.

3.3 Model Checking

Model checking is a technique to automatically check logical properties in finite-state models. Several specialist model checking tools are available, supporting the verification of systems with

different properties such as probabilistic, concurrent and real-time behaviour. Model checkers either require models specified in certain modelling languages, such as the use of ProMeLa (Process Meta Language) for the Spin model checker [27] or the PRISM language which can be used with the PRISM model checker, Storm, ePMC as well as many others, or they parse software written in standard programming languages into finite-state models for verification, such as Java Pathfinder [28] or CMC for C and C++ [51].

Typically the model checking procedure involves the verification of *safety* and *liveness* properties. Liveness properties specify that desirable events will *eventually* occur, whereas safety properties state that undesirable events *do not* occur. Consider a coffee machine modelled as a finite-state transition system, we may wish to verify that both "the water will *eventually* reach 95°" and that "hot water is *not* poured from the group head without a portafilter fitted".

Probabilistic model checking is a form of model checking which analyses systems that exhibit probabilistic behaviours, unlike other forms of model checking which give True/False results, probabilistic model checking is often used to give quantitative results. Examples of typical probabilistic model checking problems are "The probability of a plane landing without the wheels being deployed is less than 0.05%" or "The expected temperature of the water when coffee is poured from a machine is above 95°"

3.3.1 The PRISM Model Checker

PRISM [5] is a probabilistic model checker which allows for analysis of Markov chain variants such as DTMCs, MDPs, SMGs and Probabilistic Timed Automata (PTAs). All of the verification we perform on games is performed in PRISM. PRISM was used for this work because it incorporates models of stochastic games and allows for strategy synthesis. Both of these are described in this chapter.

The PRISM Language

PRISM models are described in the PRISM guarded-command language, a state based language based on the Reactive Modules formalism [52]. We continue using the Twenty-one example from the previous section to demonstrate the PRISM language. Consider the DTMC of Figure 3.2 corresponding to the game of Twenty-one where Red increments by 1 every turn and Blue chooses at random from all available actions. The corresponding PRISM specification is described in Listing 3.1. First the model type is given on line 1. The model uses a single module named 21_dtmc declared on line 3. Note that PRISM models can contain multiple interactive modules. Lines 4-7 define the variables used in the module and the initial state, in which total is 0 and turn is 1. PRISM supports typed variables as integers with a defined finite range or booleans, it does not support strings. So we represent which player is to take the next turn by an integer (unlike the string representation "R" and "B" used previously). The range is required when declaring

variables and the initial value can be given, if omitted the lowest value in the range is used as the initial value. Lines 10-22 describe the game itself and warrant more explanation. PRISM's guarded-commands take the following form:

```
[label] guard -> prob_1 :  update_1 + ...  + prob_n :  update_n;
```

where:

- `label` is an optional descriptive name for the action being taken;
- `guard` is a predicate which describes the state(s) in which the command is available;
- `prob(_1, ..., _n)` is the probability of an update being performed (note that the sum of the probabilities for each command must be 1.0);
- `update(_1, ..., _n)` is a transition which changes the state by updating the value of one or more variables with `var'` indicating the variable to be updated. Updates involve at least one statement of the form `(var' = new_value)`.

So a natural language description of the guard-command on lines 16-18 would be: *"The action 'B_1_2' can be taken when it is Blue's turn and the total is exactly 18. When the action is performed there is a 50% chance that the total will change to 19 and it will become Red's turn, and a 50% chance that the total will change to 20 and it will become Red's turn."* Finally lines 27 and 28 label the winning states for either player.

```
1  dtmc
2
3  module twenty_one_dtmc
4      // current score
5      total : [0..20]    init 0;
6      // player turn: 1-Red, 2-Blue
7      turn : [1..2]      init 1;
8
9      // Red actions
10     [R_1] turn = 1 & total < 20 ->
11         (total' = total + 1) & (turn' = 2);
12
13     // Blue actions
14     [B_1] turn = 2 & total = 19 ->
15         (total' = 20) & (turn' = 1);
16     [B_1_2] turn = 2 & total = 18 ->
17         1/2 : (total' = 19) & (turn' = 1) +
18         1/2 : (total' = 20) & (turn' = 1);
19     [B_1_2_3] turn = 2 & total < 18 ->
20         1/3 : (total' = total + 1) & (turn' = 1) +
21         1/3 : (total' = total + 2) & (turn' = 1) +
```

```
        1/3 : (total' = total + 3) & (turn' = 1);

endmodule

// winning states
label "Red_wins" = (turn = 2 & total = 20);
label "Blue_wins" = (turn = 1 & total = 20);
```

Listing 3.1: PRISM code representing the DTMC example for Twenty-one

Since in Listing 3.1 there are no states for which multiple guards are satisfied, this PRISM model is a DTMCs (no non-determinism is present).

There are only minimal changes required to change this description to specify the MDP example given in Figure 3.3, see Listing 3.2. This MDP has 3 actions: "R_1", "R_2", "R_3" as all 3 guards are satisfied, recall in this MDP Blue uses the strategy of choosing randomly from all available options, while Red has nondeterministic actions.

```
mdp

module twenty_one_mdp
    // current score
    total : [0..20]    init 0;
    // player turn: 1-Red, 2-Blue
    turn : [1..2]      init 1;

    // Red actions
    [R_1] turn = 1 & total < 20 ->
        (total' = total + 1) & (turn' = 2);
    [R_2] turn = 1 & total < 19 ->
        (total' = total + 2) & (turn' = 2);
    [R_3] turn = 1 & total < 18 ->
        (total' = total +3) & (turn' = 2);


    // Blue actions
    [B_1] turn = 2 & total = 19 ->
        (total' = 20) & (turn' = 1);
    [B_1_2] turn = 2 & total = 18 ->
        1/2 : (total' = 19) & (turn' = 1) +
        1/2 : (total' = 20) & (turn' = 1);
    [B_1_2_3] turn = 2 & total < 18 ->
        1/3 : (total' = total + 1) & (turn' = 1) +
        1/3 : (total' = total + 2) & (turn' = 1) +
        1/3 : (total' = total + 3) & (turn' = 1);

endmodule

```

```
31 // winning states
32 label "Red_wins" = (turn = 2 & total = 20);
33 label "Blue_wins" = (turn = 1 & total = 20);
```

Listing 3.2: PRISM code representing the MDP example for Twenty-one

Another feature of the PRISM language which we use is the ability to have multiple initial states (by defining them as a predicate inside an `init ... endinit` block). We use this often in later work on Chained Strategy Generation Section 4.8.

Verification using PRISM

PRISM uses properties described in the PRISM property specification language, which incorporates various logics including LTL [53], PCTL [54] and PCTL* [55] which are used for verification of DTMCs and MDPs. We only use a small set of properties in this thesis. For this reason, rather than describing each of these logics in full, we instead provide enough detail to present properties of interest.

The properties of interest for DTMCs are of the form:

P=? [F *prop*]

Which is equivalent to "What is the probability that *eventually* the specified proposition *prop* is satisfied". `F` is the eventually path operator. A property is *eventually* true if it becomes true at some point along a path. The propositions we refer to in our properties are either atomic propositions such as "Has Red won the game?", or boolean combinations of similar propositions. These propositions are defined using a label in the model code (e.g.: `"Red_wins"` in the previous examples).

A property of interest for the DTMC Twenty-one example is

P=? [F "Red_wins"]

Verification of this property for the DTMC model (Listing 3.1) gives 0.388 (3dp), which is the probability of Red winning using their *"add 1 every time"* strategy against the Blue strategy of using a fair die to decide how much to increase the total by. It is worth noting that the value given by PRISM is an exact calculation and not an approximation. As the game is zero-sum and games cannot be drawn, the probability of Blue winning is 1 minus the probability of Red winning, but we can use PRISM to verify this by ensuring the probability of either player eventually winning is exactly 1 using:

P=? [F "Red_wins" | "Blue_wins"]

Verification of properties for models which include nondeterminism (such as MDPs) must be handled differently as different resolutions of the non-determinism (i.e. different adversaries) will yield different probabilities. There is no suitable probability to be obtained for the whole set

of adversaries. For this reason we can instead ask, what are the minimum and maximum probabilities over the set of adversaries. The following property used for the MDP model (Listing 3.2) for finding the maximum probability of Red winning and the a strategy that achieves this value:

```
Pmax=? [F "Red_wins"]
```

This returns 0.996, which means that there is a strategy for Red which will guarantee that they win almost every time, regardless of Blue's actions. The strategy which is used for Red to achieve this probability is the best strategy against Blue's naive strategy of choosing 1, 2 or 3 at random. If we were to change the initial value of turn to 2, representing Blue moving first, then the same property would return the probability 1.0, indicating Red has a strategy that guarantees victory. It is important to note at this stage that the reason we know this is sufficient for a proof that there is a winning strategy for Red is that every strategy for Blue will be considered as all are available, albeit with various probabilities. The reason Red does not have a winning strategy when going first is because Blue could play the winning strategy themselves, adding 4 minus what Red added previously. There is a very low probability of this happening when Blue is choosing actions randomly, requiring the correct choice from 3 available, 5 times. ($\frac{1}{3}^5$ = 0.004 (3dp)).

Strategy Synthesis

When finding maximum or minimum reachability probabilities of MDPs, PRISM will *synthesise* an adversary which results in the maximum or minimum value. A textual version of the corresponding DTMC can be obtained from PRISM in the form of a list of states and a list of transitions. Using `-exportstates <file> -exportadv <file>` during verification we get a states file that enumerates all states in a model and a transitions file with all the choices of the optimal adversary. For our MDP example, snippets of both outputs are given in Listing 3.3 and Listing 3.4. The strategy described by the two files is the optimal strategy for Red when they go first. The states where it is Red's turn have the second variable equal to 1.

The form of the adversary file is

```
state, state', prob, action
```

where `state` describes the state transitioned from, `state'` describes the state transitioned to, `prob` gives the probability of a transition and `action` is the label of the action used. By considering the two outputs together, a more easily understood form of describing the strategy can be read as *"when the total is 2 and turn is 1, the optimal action for Red is to add 2."*

3.3.2 PRISM-Games

PRISM-Games [18] is an extension of PRISM which supports models of mathematical games in addition to the models already supported by PRISM. SMGs are one such model supported by

Listing 3.3: Example state output from strategy synthesis

```
(total,turn)
0:(0,1)
1:(1,2)
2:(2,1)
3:(2,2)
4:(3,1)
5:(3,2)
6:(4,1)
7:(4,2)
8:(5,1)
9:(5,2)
10:(6,1)
11:(6,2)
12:(7,1)
13:(7,2)
14:(8,1)
15:(8,2)
16:(9,1)
17:(9,2)
18:(10,1)
19:(10,2)
20:(11,1)
21:(11,2)
22:(12,1)
23:(12,2)
24:(13,1)
25:(13,2)
26:(14,1)
27:(14,2)
28:(15,1)
29:(15,2)
30:(16,1)
31:(16,2)
32:(17,1)
33:(17,2)
34:(18,1)
35:(18,2)
36:(19,1)
37:(19,2)
38:(20,1)
39:(20,2)
```

Listing 3.4: Example adversary output from strategy synthesis

```
40 72
0 5 1 R_3
1 2 0.333333 B_1_2_3
1 4 0.333333 B_1_2_3
1 6 0.333333 B_1_2_3
2 7 1 R_2
3 4 0.333333 B_1_2_3
3 6 0.333333 B_1_2_3
3 8 0.333333 B_1_2_3
4 7 1 R_1
5 6 0.333333 B_1_2_3
5 8 0.333333 B_1_2_3
5 10 0.333333 B_1_2_3
6 13 1 R_3
7 8 0.333333 B_1_2_3
7 10 0.333333 B_1_2_3
7 12 0.333333 B_1_2_3
8 15 1 R_3
9 10 0.333333 B_1_2_3
9 12 0.333333 B_1_2_3
9 14 0.333333 B_1_2_3
10 15 1 R_2
11 12 0.333333 B_1_2_3
11 14 0.333333 B_1_2_3
11 16 0.333333 B_1_2_3
12 15 1 R_1
13 14 0.333333 B_1_2_3
13 16 0.333333 B_1_2_3
13 18 0.333333 B_1_2_3
14 17 1 R_1
15 16 0.333333 B_1_2_3
15 18 0.333333 B_1_2_3
15 20 0.333333 B_1_2_3
16 23 1 R_3
17 18 0.333333 B_1_2_3
17 20 0.333333 B_1_2_3
17 22 0.333333 B_1_2_3
18 23 1 R_2
19 20 0.333333 B_1_2_3
19 22 0.333333 B_1_2_3
19 24 0.333333 B_1_2_3
20 23 1 R_1
21 22 0.333333 B_1_2_3
21 24 0.333333 B_1_2_3
21 26 0.333333 B_1_2_3
22 29 1 R_3
23 24 0.333333 B_1_2_3
23 26 0.333333 B_1_2_3
23 28 0.333333 B_1_2_3
24 31 1 R_3
25 26 0.333333 B_1_2_3
25 28 0.333333 B_1_2_3
25 30 0.333333 B_1_2_3
26 31 1 R_2
27 28 0.333333 B_1_2_3
27 30 0.333333 B_1_2_3
27 32 0.333333 B_1_2_3
28 31 1 R_1
29 30 0.333333 B_1_2_3
29 32 0.333333 B_1_2_3
29 34 0.333333 B_1_2_3
30 33 1 R_1
31 32 0.333333 B_1_2_3
31 34 0.333333 B_1_2_3
31 36 0.333333 B_1_2_3
32 39 1 R_3
33 34 0.333333 B_1_2_3
33 36 0.333333 B_1_2_3
33 38 0.333333 B_1_2_3
34 39 1 R_2
35 36 0.5 B_1_2
35 38 0.5 B_1_2
36 39 1 R_1
```

PRISM-Games. As the use of SMGs is common throughout our work, when we refer to PRISM in the thesis from here onward we are referring to PRISM-Games.

When defining SMGs in PRISM, the players and the actions which they control are defined within `player` *`<name>`* *`<action>*`* `endplayer` blocks. The Twenty-one example written as an SMG where both players have a non-deterministic choice of adding 1, 2 or 3 at every available state is given in Listing 3.5. There are 3 differences from the MDP model: the model declaration on line 1, the addition player definitions for Red and Blue on lines 3-8, and the re-specifying of Blue's commands to be symmetric to Red's, on 25-30.

```
1  smg
2
3  player Red
4    [R_1], [R_2], [R_3]
5  endplayer
6  player Blue
7    [B_1], [B_2], [B_3]
8  endplayer
9
10 module twenty_one_smg
11     // current score
12     total : [0..20]    init 0;
13     // player turn: 1-Red, 2-Blue
14     turn : [1..2]      init 1;
15
16     // Red actions
17     [R_1] turn = 1 & total < 20 ->
18         (total' = total + 1) & (turn' = 2);
19     [R_2] turn = 1 & total < 19 ->
20         (total' = total + 2) & (turn' = 2);
21     [R_3] turn = 1 & total < 18 ->
22         (total' = total + 3) & (turn' = 2);
23
24     // Blue actions
25     [B_1] turn = 2 & total < 20 ->
26         (total' = total + 1) & (turn' = 1);
27     [B_2] turn = 2 & total < 19 ->
28         (total' = total + 2) & (turn' = 1);
29     [B_3] turn = 2 & total < 18 ->
30         (total' = total + 3) & (turn' = 1);
31 endmodule
32
33 // winning states
34 label "Red_wins" = (turn = 2 & total = 20);
35 label "Blue_wins" = (turn = 1 & total = 20);
```

Listing 3.5: PRISM code representing the SMG example for Twenty-one

Property specification for SMGs is similar to that for MDPs. The difference is that with SMGs the player, or a coalition of players, for which an adversary must be generated must be specified. This is include as a $\langle\langle\cdots\rangle\rangle$ block at the beginning of the property. The property required to verify the optimal probability with which Red can win in Twenty-one is therefore

$$\langle\langle \text{Red} \rangle\rangle \, \mathbf{P}\texttt{max=? [F "Red\_wins"]}$$

The strategy synthesised from this property will be the strategy for Red which performs best against the best Blue strategy against it.

3.4 Games

A significant problem with model checking complex systems is that of state-space explosion [56]. This is where the number of states increases exponentially with the number of components and variables in the model. The problem is worse for non-deterministic systems, such as the games we model, as the entire strategy space must be explored. Model checking is an exhaustive process, checking every possible state of a system. Due to this when model checking large systems, abstractions must be used. A modern commercial multiplayer game, where several players inhabit the same 3D space, would not be suitable for model checking, for example.

The size and complexity of games is not likely to increase significantly in the future as the capacity of players to comprehend them will not change. However, the capability of model checking software and computing hardware will increase. Our approach constitutes an early investigation into the feasibility of model checking for game balancing, as such we consider games which are detailed enough to be interesting, but not too complex for model checking.

Recall that in this thesis we restrict the games studied to 2-player, turn-based, zero-sum stochastic games. These games can be modelled using the Markov models described in Section 3.2 with variations based on what is known about the players' strategies. There are examples within this class of games which are too complex to model check effectively. Efficient model description and construction are important for combating state-space explosion. As are established state-space reduction techniques such as symmetry reduction [57] and abstraction [58], which can also be used in specific cases.

To handle efficient verification of strategies, it is often beneficial to use singularly-sourced nondeterminism where possible as this does not require consideration of every opposing strategy. This means that using MDPs with codified opposing strategies against a nondeterministic player is preferred to SMGs representing two nondeterministic players. This process is useful for one of the earlier checks we believed model checking was suitable for, identifying winning strategies.

3.4.1 Winning Strategies

In non-stochastic games where states are reached based purely on player decision making without any randomness, a **winning strategy**, is a strategy for a player that lead to a win regardless of the opponent's strategy. The presence of a winning strategy does not preclude a game from being fun to play, but it limits replayability. Twenty-One has a winning strategy, as shown in Section 3.2. When players are unaware of the winning strategy of a game, the game is interesting to play as it remains competitive. Thereafter, competitively motivated players will employ this strategy, ruining the game for both players.

Twenty-One is a very simple game that is trivial to model check. The models which we used for it had only 39 states and the winning strategy is easily described in natural language. With more complex games identifying winning strategies can be considerably more difficult. We now present an extended example of the game Fox and Geese an asymmetric, strictly determined 2-player board game with many variants worldwide including Fox and Hounds, Wolf and Sheep, Devils and Tailors or the Swedish "Vargen och Fären" (Wolf and Ferry).

The variant we use is played on a standard 8×8 chequerboard. One player controls a fox that goes first and always moves one space diagonally, and the other controls 4 geese which also move diagonally, but only in a forward direction. All pieces are restricted to the black squares with the fox starting on the back row and the geese on the 4 front row black squares. The Fox wins by *escaping* to the front row, evading the geese that restrict its movement, whilst the geese win by *trapping* the fox, preventing it from moving.

Fox and Geese can be modelled with a simple turn variable and by variables representing the positions of the fox and geese (see the positions numbered 0 to 31 in Figure 3.5). A state in the game is represented by the tuple $[fox,\ goose_1,\ goose_2,\ goose_3,\ goose_4,\ turn]$. Each goose can move to up to 2 spaces and the fox can move to up to 4. The fact that the geese are not allowed to move backwards means that there can be no more than 27 moves made by either player before the game ends (i.e., the highest number of moves made by both players is 56).

There is a winning strategy for the geese detailed in full in [59]. This strategy entails the geese adopting one of 5 positions (or one of their right-left reflected counterparts) and moving to another of these positions depending on whether they are *in danger* as a result of the position of the fox. In each position the spaces that a fox can be in which are regarded as *dangerous* to the goose player are noted in the strategy description. Some of the transitions between positions require a sequence of moves, but they can be performed without regard of the fox's moves. In the strategy's description, the authors suggest geese players can close their eyes during these transitions, assured that when they reopen them the geese will be back in one of the 5 positions.

Model checking can be used to verify that there is a winning geese strategy. A PRISM model of fox and geese used for this purpose is given in Appendix A. In the model the fox uses a **naive** strategy, they act based on a random decision of all available actions. This ensures that every reachable state is visited during model checking, without requiring the synthesis of every fox

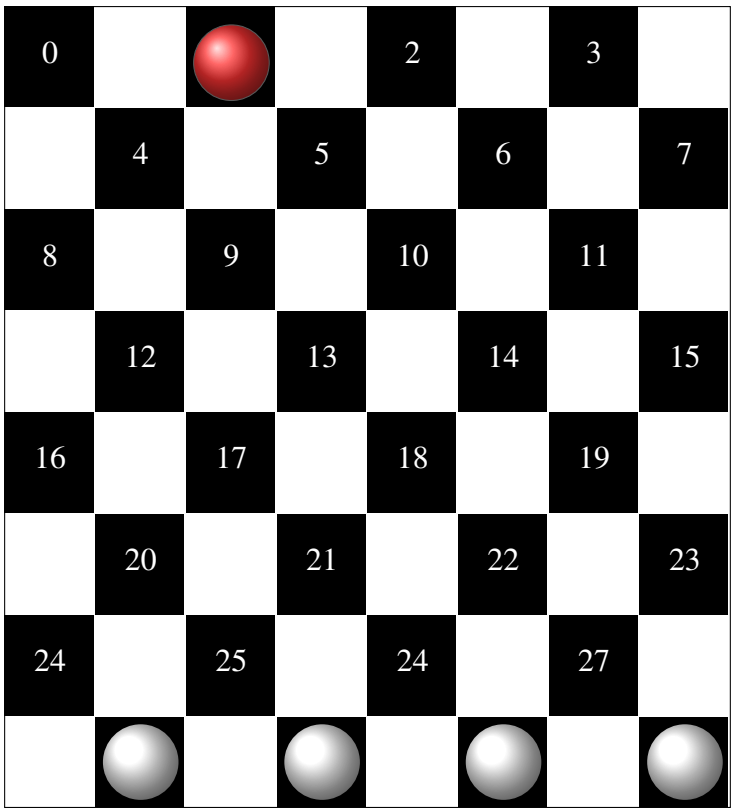

Figure 3.5: Fox and Geese starting position. The fox is a single red piece and the geese are 4 white pieces

strategy. Verification of the property:

$$\mathbf{P}\texttt{max=? [F "deadlock" \& !"fox\_wins"]}$$

returns 1.0, proving that a winning strategy does exist for the geese. The model has 6,991,560 states and 40,780,473 transitions and the resulting adversary file is 8,461,427 line long. It would be extraordinarily difficult to get a high-level description of the strategy from the results.

The alternative use for model checking in this instance would be to model the known winning goose strategy against a nondeterministic fox opponent and prove that it is indeed a winning strategy. Writing the model by hand for this would be very difficult as every state would have to be expressed in the strategy description for the goose player. This is an example where PRISM specification must be generated automatically from a template before model checking can occur. This approach, which we use throughout this thesis, cannot be done from within PRISM itself. We use Python scripts to generate our PRISM models in this way. The utility of checking candidate strategies to see if they are winning strategies is limited and likely not worth the effort. Identifying that a game has some winning strategy in the first instance is more interesting.

Models of this size are laborious to manually debug. Any errors in the models are due to issues in the code created to generate them. By performing verification of simple properties with the model checker (such as that both optimal strategies sum to 1.0 in the earlier 21 example) one can recognise if a model is incorrect, but this does not locate the inaccuracies. The size of the

files make this locating difficult. For the creation of the models in this work, a roughly equivalent amount of time was spent debugging models as implementing the generation of them. This is a notable drawback with any modelling of this form.

Using model checking to check for winning strategies in pre-existing games and verifying candidate strategies is interesting, but the impact is limited and the cost is high. For this reason we did not explore this avenue of potential model checking applications to games any further.

3.4.2 First Move Bias

In typical action-based games where acting brings with it progress towards an end goal, it is likely that going first will afford a player an extra action, in turn increasing the odds of their winning. This is known as **first move bias**. However, as seen with Twenty-one, acting first is not always beneficial. There are some positions in games from which the imperative to act puts a player at a disadvantage [60], this situation is called *zugzwang*.

In turn-based 2-player games where a fair coin-flip decides who acts first the bias is overcome through the equal chance of each player going first. However, this fairness on behalf of the players (i.e., either can have first move bias with an even probability) is not enough to ensure a balanced game. A game which has a strong bias towards whoever goes first (or second), even if the player that ultimately goes first is decided through chance, is not fair. Players will win or lose based on fortune, rather than having their skill or knowledge of the game rewarded. It is an aim of game balancing therefore, to diminish the effects of first move bias as much as possible. Model checking of games can also be used to measure the extent of first move bias, quantifying how dependant the outcome of a game is upon winning that initial coin-flip. Solutions to limit first move bias include playing repeated games, extending the game so an extra action is less impactful, including the first move as part of the game mechanics and offering minor advantages to players who don't act first, such as greater resources.

3.4.3 Game Material

Game material refers to the elements of a game which represent the player and with which they interact. In a racing game for example, the game material could be a selection of cars to choose from or it may go deeper and include a series of components that make up the cars such as tyres, engine, transmission etc. This material is described using numerical **attributes**, perhaps *top speed*, *acceleration* and *handling* in a racing game. In a **symmetric game** all players have access to the same material. It can be said of any symmetric game that it is fair as there is an even playing field, any player can use any material, but this does not mean that a game is balanced. If there is a known *best* material that is strictly better than all others, then competitively motivated players will essentially have their material choice diminished to just those known. An aim of game development is to keep viable player choice as broad as possible. Furthermore, developers will

not want to spend time and effort developing game material that is never used. Game balancing is frequently thought of in terms of fairness, we argue that the fairness between material is as important as fairness between the players.

If there is no best material, or subset of material, then intransitive relationships exist between material – some material a is better than material b which is better than c which is better than a. Material that is effective against another is described as a **counter** to it. This type of relationship is captured in the game of rock-paper-scissors (*paper* beats *rock*, which beats *scissors*, which beats *paper*). It is central to the design of several competitive games, it is even advocated for as a design methodology [61]. A cyclical relationship in a popular game will naturally lead to trends in how the game is played. If material a is popular and the community discover that material b is a counter to it, then b will rise in popularity, which will lead to a fall in a's strength and likely a fall in its popularity. This progression through the material is known colloquially as **the metagame**, the ever-evolving state of play [62]. The currently perceived best way of playing is known as "the meta". A *healthy metagame* is constantly changing as new material is used and different strategies are employed by the community. Amongst the games research community the term *"metagame"* is used to refer to various different game-adjacent concepts [33], in this work we strictly refer to it as the trends in popularity and comprehension of what strategies are currently *good* by a playerbase. This concept is integral to this work and what we believe to be the most commonly used form.

To non-game players the concept of an abstract system, the metagame, is a peculiar one. It is equivalent to cyclical trends in other areas, such as fashion. Certain events can alter the metagame's organic advancement, or speed it up. In a presentation entitled "Breaking the Metagame: Seventeen Seconds of Dota 2" [63], a single moment from professional play of Dota 2 is examined, along with the resulting change in the way certain characters in the game were considered by all players. The way esports players play games is often emulated by the community en masse. Furthermore their need for a competitive edge sees esports players more frequently identifying novel, effective strategies in the current metagame, in turn directing the metagame. There are other scenarios which can affect the metagame. Games with in-game shops can also see changes in the metagame occur as a result of new popular items for given characters or sales to certain material, for example.

There are some basic principles followed when designing material for competitive games. First, no material should be strictly better than any other, in the car example from before there should not be a car with a higher top-speed, better handling and greater acceleration than another, because then the other car would never be chosen. Where this occurs, the material that is weaker in every aspect is said to be **strictly dominated**. Additionally, material should be *orthogonally differentiated* [64], it should differ mechanically rather than numerically. This widens the pool of potential material and diversifies the way in which a game can be played. In our racing example we could then have a seemingly strictly dominated vehicle that also had access to short-cuts

around the track leading to more interesting decision making offered to players. This orthogonal differentiation obfuscates the relative effectiveness of the material, which may well be desirable for players, but it makes balancing them more difficult. What is the numeric equivalent of access to short-cuts in terms of slower speeds? Quantifying these qualitative differences is key to balancing asymmetric material in competitive games.

Some games involve sets of material, team games constitute the majority of these. In these instances we refer to **material sets**, the full group used by a player or team and **material units**, an individual material selected as part of a set. When reasoning about the balance of games with material sets, it is not the case that every material set needs to be part of the metagame at some point. All material units should at some point be useful, otherwise their inclusion in a game does not increase the options available to players, but having material sets which universally perform poorly is acceptable. As a rule, all material must at some point be good, but no material can be good always. Practically this means material units be in some effective set, but no set is better than all others.

3.4.4 Material Selection

In designing games with large pools of material which players can choose from, one must consider how that material is chosen. We refer to a selection of material for all players as a **matchup**. If players take turns selecting material then whoever goes first would be at a disadvantage, should the relationships between material be known, unless the game does not allow **mirrored matchups** (multiple players selecting the same material), in which case they may be at an advantage by being able to take the best material. Material selection in a symmetric game that occurs concurrently without knowledge of what opponents have chosen will result in a fair game for the players. Players of equivalent skill have an equal optimal probability of winning. But as with first move bias, this does not ensure that a game is balanced.

When modelling games we consider various forms of material selection, sometimes different to those used by the game we are modelling. In the same way that we change the models that we use depending on our aims, with different forms of material selection we can consider a full game, identify optimal counter material or examine a single matchup more closely.

3.4.5 Strategy and Metagame Representation

Different actions are available to players when they choose different material, by extension this means different strategies are also available to the players. Whilst in the models themselves a strategy for a player may include navigating material selection, we consider a high-level abstraction that encompasses the material chosen and the strategy used as being part of a wider **play style** or way of playing the game. The level of specificity that model checking provides is greater than how strategies are naturally described. To ensure a healthy metagame we want to observe

a range of material choices and strategies used within the same material, all encompassed by players naturally adapting their play style to maximise their probability of winning.

Capturing a snapshot of the metagame is difficult. The period of time considered must be decided upon and strategies used should be a part of the current meta as much as material selection is. As a simple indicator of the metagame we often consider only the most effective and popular play styles in a given period.

One aim of ours is to predict the metagame ahead of time. If transitions between ways of playing are observable and developers can extrapolate to future transitions then they can use this knowledge to better support the game. This may take the form of developing more content to support material that is soon to be popular or even allowing developers to *guide* the metagame in a healthier direction by incentivising the play of material, should predictions show the metagame stagnating.

3.4.6 Player Motivation

Playing to win is just one of several motivations for play. Game theory reasons about rational players attempting to maximise their payoff, the expected-utility function [65], but this is not how games are played. Even in seemingly competitive games, *deviant play*, where players act to explore the boundaries of what is possible or even to spoil the experience of others, is common. Some game systems, such as skill-based matchmaking, are set up in a way to punish successful players by matching them against similarly successful opponents. These systems can lead to intentionally sub-optimal play (or *throwing*) being in the best interest of the player.

Games are not always designed to encourage players to play competitively. Some multiplayer games involve simultaneous components of competitive and collaborative play. Others encourage exploration through hidden rewards or collectables. A multinational study of over 30,000 game players [66] identified 6 player motivations, of which *competitor* was only one, Completionists, Competitors, Escapists, Story-driven and Smarty-pants. Similarly [67] expands on a traditional model of the components of player motivation [68], to show that 3 distinct motivation groups exist: *achievement*, *social* and *immersion* in Massively-Multiplayer Online Role-Playing Game (MMORPG) players. Competition is merely a sub-component of one of these (achievement).

In this work we consider all agents to be driven solely by competition. Where we examine the gameplay data of real players we take measures to ensure we consider only data from players in situations where motivation is purely competitive. This assumption may appear to compromise on accuracy. However balancing games based on the highest level of play is the accepted best practice, in keeping with a focus on esports.

3.4.7 Player Skill

A large set of players will exhibit a wide range of *skill*. The form of player skill depends on the game being played. In Twenty-one or Fox & Geese a player is not higher-skilled if they have greater manual dexterity. However, if they are more familiar with the game system or have superior spacial reasoning then they can be considered *higher skilled*.

With the rise of esports, the focus of game balancing on the top-end of player skill has increased. This is partly due to the fact that esports bring in more players and more professional players who have greater sway over design decisions. This is more of an issue with dexterity based games like Shooters or Multiplayer Battle Arena Games (MOBAs) where small differences in player skill result in considerable disparity in player success. For example, in the MOBA League of Legends, a character Akali had a 44% win-rate amongst all players, but at a professional competition, she was the most banned character and had a 72% win-rate (teams can ban some characters from being picked in competitive formats of League of Legends during character selection). Amateur players were failing to use Akali effectively, whilst professionals were able to use her with devastating effect leading to her overwhelming presence at the competition. The developers of League of Legends, Riot games, then had to decide whether to nerf Akali, making her worse, which would likely push her win-rate lower even than 44% in general play. The eventual solution was to rework Akali, changing her abilities to be easier to use, but less effective in the hands of a skilled player.

Balancing for varied player skill levels is beyond the scope of this work, but it should be addressed in the context that we assume players are playing to win.

3.4.8 Statistical Player Analysis

Player data must be used to inform the balancing of games as balance is subjective. The collective experience of the players is distilled into a series of metrics, used to analyse how a game is played, what material is employed and which strategies players favour. Win-rates give the percentage of the games played in which a material unit wins, where 50% describes material which wins half of the games in which it is selected. The aim is to keep all material at approximately 50%, any deviance then suggests material which is either too weak or too strong in the current metagame. Pick-rates are similar, measuring the proportion of games in which material is selected. The target for each game will differ based on the material pool and material set sizes, but ultimately the aim is for all material to be chosen a similar number of times. Material which is too popular is not ideal and might suggest rebalance is needed, while material that is too unpopular suggests the experience of using that material is not enjoyable and should be further investigated.

Pick-rates and win-rates can be considered in unison. A healthy metagame will see material popularity progress through the entire material set over a period of time. When developers observe drops in either metric for material which they know will later increase as the metagame

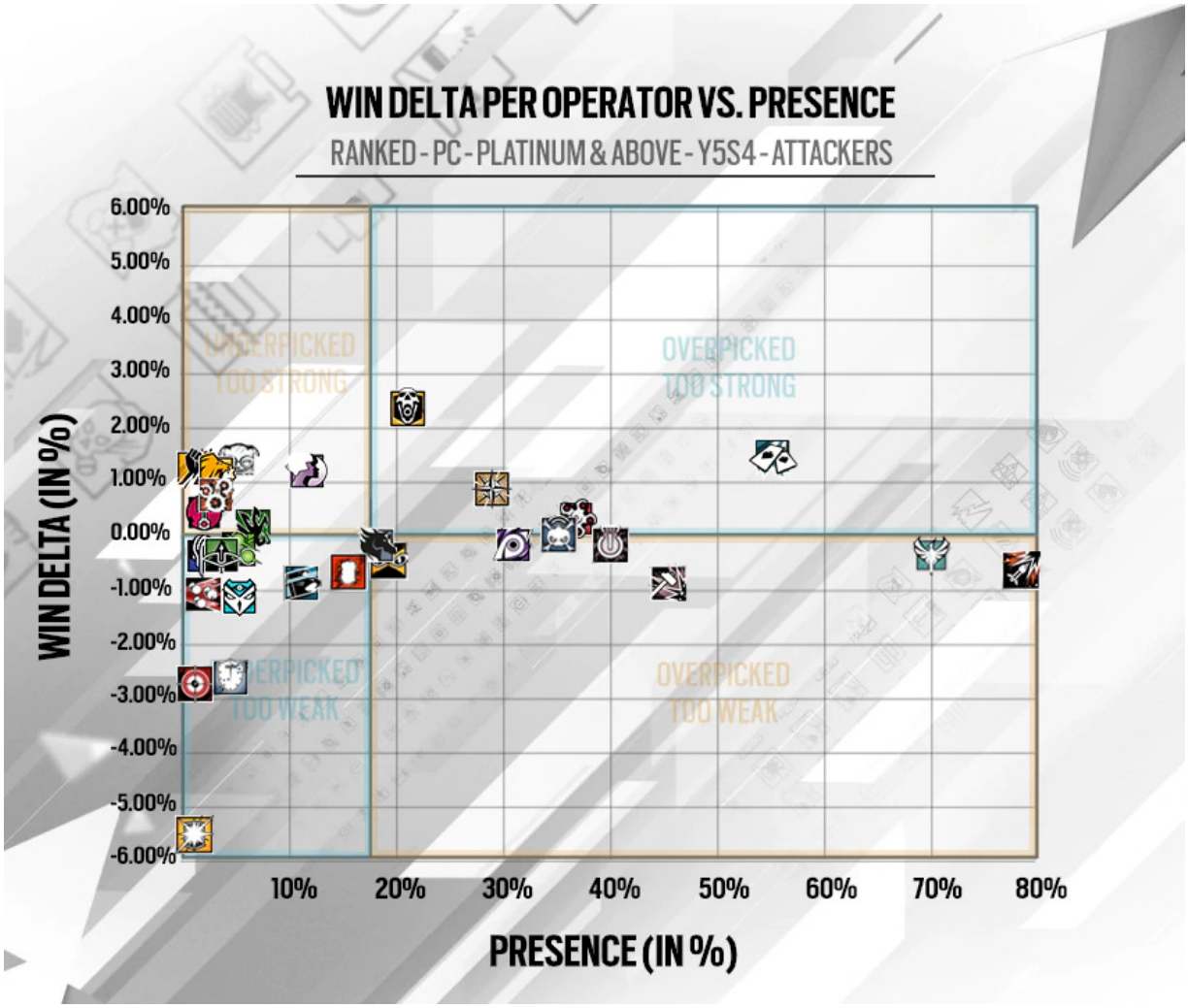

Figure 3.6: Balancing matrix taken from 25/1/2020 developer blog, Ubisoft Montreal

progresses, can confidently wait for the metrics to improve, rather than prematurely update the game. The developers of Tom Clancy's Rainbow Six Siege (Ubisoft Montreal, 2015) created the *balancing matrix*, which plots the win-rate of all game material against its pick rate. This provides a useful way of visualising the current meta game and can be used to explain the reasoning behind any updates. A recent example is given in Figure 3.6 where half of the *operators* in the game (the *Attackers*) are displayed using their iconography. The game has a ban phase before operators are chosen, where teams can ban operators from being selected, the *presence* metric gives the % of games played by operators who were not banned. Two very popular operators with 70% and 80% presence were slightly nerfed in the corresponding patch, despite their negative win delta, as their popularity was seen as the more pressing issue. This example illustrates the intricacies of game balancing.

Game balancing is often initially implemented through a network of countering materials. This naturally gives rise to the metagame. To measure the success of these counters developers need to consider the matchup data, how all material performs against all other material. A well

designed game will see all material performing well against some opponents and poorly against others. When material sets are used this analysis can be difficult to perform as some material will work well together, whilst others will be less complementary. Where possible the matchups of the sets should also be considered, but with large material pools this can result in an unfeasibly large set of comparisons to consider.

A recent trend with online games is that of developers describing the processes for balancing explicitly in the name of transparency. League of Legends developers Riot announced a balancing framework in May of 2019 [69]. They divide the playerbase into 4 groups: *average play*, *skill played*, *elite play* and *professional play* and describe the metrics for champions (the game's material) which are unacceptable at each band, e.g.: a champion having a win-rate above 54% on average across all levels is considered justification to nerf them. Interestingly Riot state that: *"While some may argue that the game should be balanced around only the very best players of the game, we think a balanced experience is an important part of what makes League compelling regardless of skill level."* This differs from the philosophy of other developers who consider only the highest level of play and assume other players will move towards the metagame of professional play.

3.5 Conclusion

In this chapter we have defined key terms which we will use in the rest of the thesis. We have shown how model checking can be used to balance games and introduced the models used while describing how they are used and for what purpose. We have briefly explained how we can specify games, how a model checker can be used to analyse them and given examples of writing PRISM code and the verification of models with PRISM. We have also explained the terminology used for game development and analysis and highlighted areas pertinent to our work on game balancing.

Chapter 4

Chained Strategy Generation

"Where repeated adversary generation via model checking mimics players coming to terms with a game and affords developers insights prior to release."

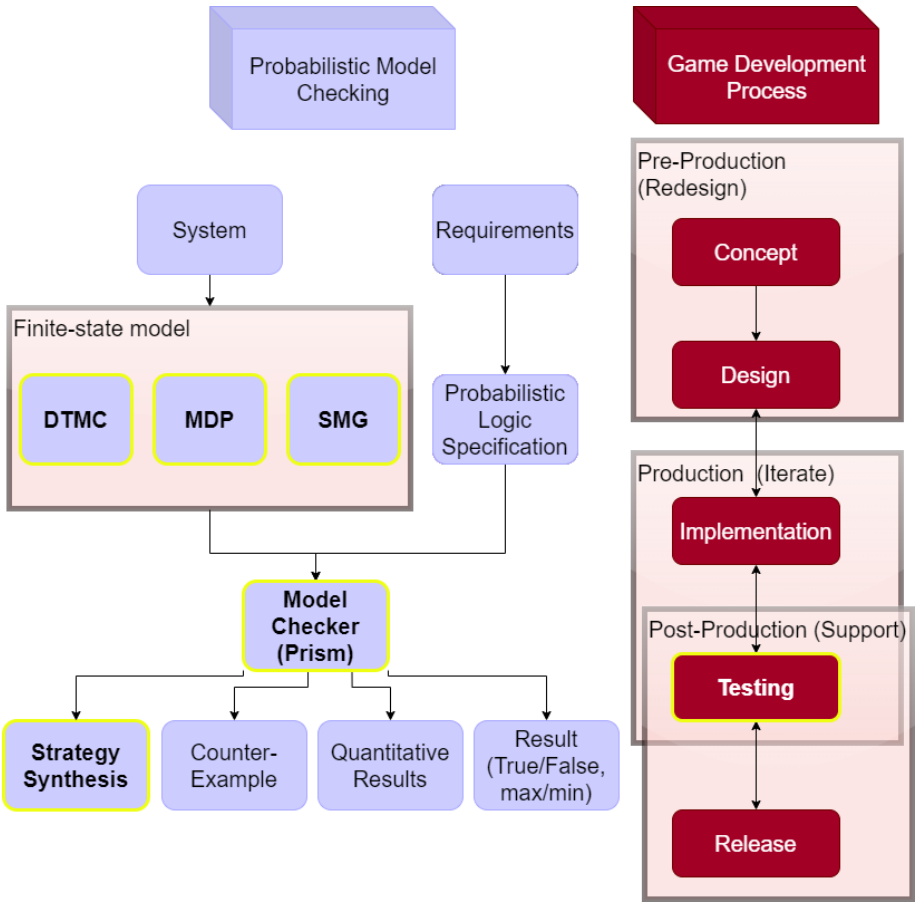

Figure 4.1: Chapter 4 areas.

4.1 Introduction

In this chapter we introduce Chained Strategy Generation (CSG), a novel approach to competitive game analysis. CSG uses repeated strategy synthesis to generate ways of playing that mimic game playing communities moving from locally optimal strategies against known effective strategies. This chapter begins in Section 4.2 by describing the motivation for collecting synthetic gameplay data. This is followed by a full and detailed algorithmic description of CSG in Section 4.3. We go on to describe the case-study used for this work in Section 4.4, a game – RPGLite, and show how CSG can be used to analyse it without player data in Section 4.5. What was learned about the game from those results is given in Section 4.6 and the approach in totality is discussed in Section 4.7.

In Section 4.8 we expand upon the ideas of CSG, improving upon the precision of results and reducing the cost of verification. In Section 4.9 we introduce optimality networks which allow us to quantify the relationship between all material units based purely on synthesised data. We show how these networks can be used to automatically reconfigure a more balanced version of RPGLite. Finally in Section 4.10 we conclude the chapter by describing its usefulness and how it could be adapted to other games.

4.2 Motivation

Game balancing in practise is more often reactive than proactive. Game developers alter the game in response to feedback from players or issues highlighted through internal analytics. The balancing that takes place prior to release is based on internal play-testing. Play-testing is hugely expensive and prone to biases and human error. In-house quality assurance teams will play the game repeatedly to replicate player behaviour, trying to find bugs and identify overly strong or overly weak material. But, without a huge team given a long period of time, it is not possible to explore the entirety of the game's strategy space.

Game balance is regularly cited as being the key reason behind game longevity. Players must be offered numerous viable play styles so they can explore and exploit the game simultaneously, rather than simply identifying the best way to play and repeatedly employing only that style. With CSG we map the evolution through various ways of playing a simple game. Most simple games are likely to be *solvable*. There will be a single superior way of playing. This arises primarily because of imbalance from poor design. However, if the solution is not found for a long period of time, then the game will remain engaging to players as they search for it. With CSG we can observe a synthesised version of the search for a solution to a game to make judgements about the game itself and whether or not it is fun to play. There are clear benefits of an automated system that can be efficiently employed as opposed to expensive testing.

Term	Description
Strategy	A set of actions, or distributions over available actions, from every state in which a player could find themselves.
Adversarial strategy	The strategy which maximises the probability of winning against a known opposing strategy (the *counter* to it).
Dominant strategy	A strategy that is superior to all other available strategies.
Optimal strategy	The strategy which has the highest probability of winning against its own adversary.
Effective strategy	A strategy which may be chosen by rational players.

Table 4.1: Simplified description of variations of strategy found in this thesis.

4.3 Methodology

In this section and the remainder of the thesis we refer to various forms of strategy. For clarity we summarise the terminology of the strategies used in Table 4.1. Note that a dominant strategy is its own adversarial strategy and is an optimal strategy. There can be multiple strategies which are adversarial to a given strategy if they all have an equal probability of winning against it, similarly there can be multiple optimal strategies, however there can only ever be at most one dominant strategy.

4.3.1 Description

We use PRISM to generate *effective strategies* by finding the counter to a strategy that is popular in the metagame. CSG is the process of generating an effective strategy, then generating the adversarial strategy against it. This lets us reason that, when finding adversaries of adversarial strategies, we are always finding effective strategies – a strategy that is best against some previous effective strategy must itself be effective. By identifying a small subset of effective strategies from all of those available, we overcome one of the major challenges with automated game balancing, the need to compare all strategies to each other – a task which is often intractable. We can use CSG to identify two possible issues with a game's balance: the existence of a dominant strategy against any material; and all strategies being dominated for some material.

Identifying Dominant Strategies

As with game balancing, what constitutes a dominant strategy is different depending on the genre of game. Generally a dominant strategy can be thought of as a strategy so effective as to be superior to all others. We define a dominant strategy as a strategy which is best played against by itself, i.e., a strategy which is its own adversarial strategy when all possible material is considered.

In order to identify dominant strategies quickly we exploit the fact that a dominant strategy must be optimal against a player with the same material. There can only be one optimal strategy for each allocation of material to players for the type of games we consider. We can identify potential dominant strategies by only investigating strategies which are optimal against their own material, limiting the number under investigation to the size of the material. We can then test these candidate strategies against other material to see if they are indeed dominant.

Identifying Dominated Material

Dominated player material is that which an informed player will never use, because no effective strategy exists for it. In order to identify dominated material we generate a series of effective strategies and measure how effective individual material is against them. Using CSG we can compare adversarial strategies for all material against known effective strategies. If any material is significantly worse than all others against all effective strategies we claim it is dominated. Whilst not exhaustive, this approach will give a good representation of how particular material could be used in the best case.

The methodology of CSG is shown in Algorithm 1. First we generate a seed strategy for some material, a strategy with an action chosen at random from those available at every occasion where the player has a choice to make. We then calculate the **adversarial probabilities**, i.e., the maximum probabilities of winning, for all material against the seed strategy. We generate the adversarial strategy for the material with the greatest adversarial probability and then calculate adversarial probabilities against the newly generated strategy. We continue in this manner, calculating probabilities and generating the strategy which performs best, until we generate a strategy that we have generated before – i.e., we have found a cycle of strategies. The algorithm always terminates, the proof for this is trivial, given in Theorem 4.3.1. Upon termination, either a dominant strategy or a cycle of effective, non-dominant strategies is identified. In addition to the strategies identified, the adversarial probabilities for all material at each iteration are returned, however further analysis is required to contextualise the results.

Theorem 4.3.1 (CSG termination)**.** *CSG will eventually terminate.*

Proof of Theorem 4.3.1. There are a finite number of actions available at any state and a finite number of states, therefore the strategy space is also finite. Prism's adversary generation is deterministic, given a strategy it will always identify the same adversarial strategy. Hence a

Algorithm 1: Chained Strategy Generation

```
   output: Returns the adversarial probabilities for all material at each iteration
1  probs := [][]
2  strats := []
3  k := 0                                                          // iteration
   /* Start with a randomly generated strategy                                */
4  strats[k] := seed_strategy
5  while strats[k] ≠ strats[j] for all j < k do
6      best_m := null                                              // best material
7      best_probability := 0
8      for m ∈ material do
           /* Find best opponent                                              */
9          calculate adversarial probability against strats[k]
10         probs[m][k] := probability
11         if probability > best_probability then
12             update best_probability
13             update best_m
14         end
15     end
16     k++
       /* store new strategy                                                  */
17     strats[k] := strategy for best_m against strats[k−1]
18 end
19 return probs
```

previously encountered adversarial strategy will eventually be reached, and so the algorithm terminates. □

By comparing the material used by the strategies generated at each step, we can analyse the comparative effectiveness of each material independent of strategy. Dominated material is identified as the one used to generate adversarial strategies that consistently perform worse than strategies for other material. In comparing the probabilities of winning for all strategies per material, game designers can make value judgements on comparative strength across all material.

4.3.2 Statistical Analysis of CSG

CSG outputs a sequence of probabilities for all material in a game at every iteration of the algorithm. Analysis of these results gives a greater understanding of how fair and interesting the game is. We introduce a series of metrics for objective comparison between similar games. Our definition of an effective strategy is recursive: *an effective strategy is a strategy which performs best against another effective strategy*. Because of this, in our analysis we do not consider all iterations of CSG, starting instead at a lower-bound to allow the strategies time to settle. This is the base case for our recursive definition of effective strategies. The delay will need to be

configured by game developers as games have varying complexity and the length of time taken before strategy generation settles will differ. We set this value to be one quarter of the number of iterations performed before a cycle is identified. The process of using a delay in this way is equivalent to waiting for players to familiarise themselves with a game before assuming they are effective players. We reached the value of $\frac{1}{4}$ through trial-and-error.

We now introduce some novel formulae. These were created alongside CSG to quantify the state of game balance. With these it is possible to compare two candidate configurations and support an argument for one being more balanced than another. To describe the formulae used, we introduce the following notation:

- $\boldsymbol{M}$ is the set of playing material (of size $|\boldsymbol{M}|$) and m denotes some material in $\boldsymbol{M}$;
- k^* is the first iteration of CSG where a strategy is identified to be effective, i.e., the value for the delay before players are assumed to be using effective strategies in a single execution of CSG (this is the base case for our definition of effective strategies);
- K and $K^\star (= K - k^*)$ are the number of total iterations and the number of iterations generating effective strategies by CSG, respectively;
- $winProb(m, k)$ is the maximum probability of winning for any strategy using material m at iteration k;
- $winProb(\boldsymbol{M}, k)$ is the maximum probability of winning for any strategy using any material in the set $\boldsymbol{M}$ at iteration k.

Material Robustness First we study how effective specific material is over an extended period of time. We refer to this as material robustness. We calculate material robustness by taking the mean of the maximum probabilities of winning against effective strategies for some material:

$$\rho(m) \stackrel{\text{def}}{=} \frac{\sum_{k=k^*}^{K} winProb(m, k)}{K^\star}$$

This gives a measure of how viable material is over time. It can be used to compare the strength of different materials and as evidence to suggest redesign. If the robustness for material m is lower than 0.5, then m is too weak as it loses more often than it wins, even when employed with the strategies that maximise the probability of a player using material m winning.

Mean Robustness

The mean of material robustness over all material $\boldsymbol{M}$ gives a measure of how strong the effective strategies are when compared to the best ways of playing against them. We call this the mean

robustness of M, denoted $\mu_\rho(M)$ and defined as follows:

$$\mu_\rho(M) \stackrel{\text{def}}{=} \frac{\sum_{m \in M} \rho(m)}{|M|}$$

A game with a high mean robustness is one in which players can always find multiple ways of successfully playing against effective strategies. A game with low mean robustness, i.e., one with mean robustness close to 0.5, would be one in which the effective strategies identified were overly powerful. For this reason we claim that a higher mean robustness indicates a more interesting game, as effective strategies do not limit the choices of the opponents as much.

Win Delta and Loss Delta

The results of CSG can be used to measure the variability of potential effectiveness for material against effective strategies. We do this by calculating the win and loss deltas for all material. The win delta of material m, denoted by $\delta^{win}(m)$, is the average probability with which material can beat effective strategies. The loss delta of material m, denoted by $\delta^{loss}(m)$, is the average minimal probability with which material will lose to effective strategies. Formally, for $result \in \{win, loss\}$:

$$\delta^{result}(m) \stackrel{\text{def}}{=} \frac{\sum_{k=k^*}^{K} \delta^{result}(m,k)}{K^\star}$$

where

$$\delta^{win}(m,k) = \begin{cases} winProb(m,k) - 0.5 & \text{if } winProb(m,k) > 0.5 \\ 0 & \text{otherwise} \end{cases}$$

$$\delta^{loss}(m,k) = \begin{cases} 0.5 - winProb(m,k) & \text{if } winProb(m,k) < 0.5 \\ 0 & \text{otherwise} \end{cases}$$

These values allow us to measure effectiveness of a particular material m without considering situations where material m is very unsuited to playing against a given strategy. This is to be expected in a healthy metagame – some strategies are always effective against certain material. A *fair* game is one for which all material can win by similar, significant amounts. Any material with a win delta of 0 never wins against any effective strategy. We claim that in this instance the game is unbalanced as that material will never be used in high-level play. These values are measures of risk and reward and could be exploited by game developers for whom it would be desirable to have some material with low risk and low reward and others with high risk and high reward.

Figure 4.2: The Knight, Archer and Wizard from RPGLite 1

Outplay Potential

CSG provides an indication of the level of strategic depth in a game, showing how important the use of good strategies is to success. We call this a measure of outplay potential and calculate it as the mean of the difference between each maximum adversarial probability and the mean of the adversarial probabilities for all material once strategies have settled. More precisely *outplayPotential* is defined as:

$$\frac{\sum_{k=k^*}^{K} \left(winProb(M,k) - \left(\sum_{m \in M} winProb(m,k)\right)/M\right)}{K^\star}$$

A higher value of outplay potential suggests a greater spread of potential effectiveness between material. Outplay potential shows how important material choice is to the probability of winning, with higher values implying greater importance. This is a significant measure because it allows developers to gauge how important game knowledge is compared to strategic skill, i.e., knowing what material to use rather than what strategy. We argue that a game is more *interesting* if the maximum probability of success is highly dependent upon material choice, provided the material choice is *fair*.

4.4 RPGLite 1: The Case-study

RPGLite is a case-study we have developed which uses archetypal fantasy role-playing game (RPG) tropes. It involves turn-based, stochastic combat such as that used in battling from Pokémon games or combat in table-top role-playing games like Dungeons & Dragons. We refer to this form of RPGLite as RPGLite 1 because it is the first (and most simple) of a sequence of versions referred to in this thesis. In this chapter we will refer to it as only RPGLite and make clear when, later in the thesis, we are alluding to a later version.

RPGLite is a two-player game in which each player chooses two different characters out of three available. The winner is the first player to reduce the health of both of their opponent's

characters to 0 or less. The characters are: the *Knight*, who attacks a single opponent; the *Archer*, who attacks both opponents simultaneously; and the *Wizard*, who attacks a single opponent and attempts to *stun* them, preventing them from performing an action on their following turn.

A coin is flipped to decide who goes first and play continues in a turn-based fashion. On their turn, a player chooses one action to perform from any of their alive characters that are not stunned and a target for that action from any alive opposing character. These actions result in a *hit* or a *miss*, with a probability determined by the character's attributes. The *Archer* is unique in that their action can *hit-twice*, *hit-one*, *hit-other* or *miss-both* rather than simply *hit* or *miss*. Each of the three characters have three attributes: *health*, *accuracy* and *damage*, which govern how much damage they can sustain before dying, how likely their actions are to *hit* and how much damage their actions inflict. Our purpose is to identify values for these attributes for each character that will make the game fair and interesting to play.

RPGLite is intended to be a simple game which will allow us to clearly illustrate our approach without the use of complex game mechanics. The game has an associated state space S. Every state $s \in S$ in RPGLite is a tuple of values of the form:

$$(attack, turn, p1c1, p1c2, p1\_stun, p2c1, p2c2, p2\_stun)$$

where each value is a realisation of a corresponding variable. All variables, with ranges, are described in Table 4.2. A lower bound for the initial value of each of the health variables is given by 1 minus the maximum damage attribute of all characters.

The characters are intended to excel at different times during a play of the game. The *Archer* is meant to be able to do the most damage early in the game when both opposing characters are alive. For this reason we constrain the product of the Archer's *damage* and *accuracy* to greater than half of the product of the Knight's *damage* and *accuracy*. The *Wizard* is intended to be more powerful towards the end of a play of the game, when one of the opponent's characters is dead. When only a single opposing character is alive a *Wizard* could stun them repeatedly with a high probability, forcing an opponent to skip several turns.

The material choice for players in our case study is a pair of characters. We abbreviate the character pairs to initials, i.e,. Knight-Archer is represented as KA. A matchup for a game is denoted similarly, e.g. KAvKW. Moves are specified using analogous notation, e.g. p1A_p2K represents the move "Player 1 uses their Archer to attack player 2's Knight".

Example Game

Using a KAvKW game as an example, a game may play out as follows:

1. *p1c1* is set to 8, *p1c2* is set to 7 (as the Archer has 7 as its initial health and the Knight has 8), similarly *p2c1* is set to 8 and *p2c2* is set to 7;

2. A coin is flipped to decide who acts first, player 1 wins and *turn* is set to 1;

Variable	Range	Description
attack	$0, \ldots, 9$	Last action selected (0 – no action)
turn	$0, \ldots, 2$	Player turn indicator
p1c1	$-2, \ldots, 8$	player 1, character 1 health
p1c2	$-2, \ldots, 8$	player 1, character 2 health
p2c1	$-2, \ldots, 8$	player 2, character 1 health
p2c2	$-2, \ldots, 8$	player 2, character 2 health
p1_stun	$0, \ldots, 2$	player 1 character stunned (0 – neither)
p2_stun	$0, \ldots, 2$	player 2 character stunned (0 – neither)

Table 4.2: RPGLite variable for an example configuration.

3. Player 1 has a choice of 6 actions on their first turn: They can do one of the following:

   - Use their Knight to attack the Knight of player 2 (p1K_p2K);
   - Use their Knight to attack the Wizard of player 2 (p1K_p2W);
   - Use their Archer to attack the Knight of player 2 (p1A_p2K);
   - Use their Archer to attack the Wizard of player 2 (p1A_p2W);
   - Use their Archer to attack both opposing characters of player 2 (p1A_p2Kp2W), or;
   - Skip their turn (p1_skip);

   The player choose to use the Archer to attack both of player 2's characters and *hit_twice*. This deals 2 damage to both characters, so *p2c1* and *p2c2* are reduced to 6 and 5 respectively and *turn* is set to 2.

4. Player 2 has a choice of 5 actions and attempts *p2K_p1A*, but misses, *turn* is set to 1.

5. ... and so on, until ($p1c1 \leq 0$ and $p1c2 \leq 0$) or ($p2c1 \leq 0$ and $p2c2 \leq 0$).

4.4.1 CSG for RPGLite

CSG is written as a batch job in Python3. The program takes the path of a configuration file as an argument and generates the required models at each step, running PRISM and parsing the output to guide future iterations. The PRISM models used are MDPs for identifying dominated material (running the full CSG algorithm) and SMGs for identifying the candidate dominant strategies. In all PRISM specifications character selection is omitted with characters being chosen before the file is generated. Each PRISM specification is made up of three composite parts which are generated independently and listed below.

- A prefix, describing the model used, the constants describing the attributes from the configuration file and the variables describing the current state.

- A suffix, giving the properties and formulae used and the actions for either player.
- The action decision states for either player with players either:
  - modelled as *free* nondeterministic agents;
  - modelled as *educated* agents who are following a specific, generated strategy;
  - modelled as *naive* agents who are following a simple, randomly-generated strategy.

The program also stores each strategy generated as a plain-text file which can be copied in as an *educated* player and is used for byte-wise file comparison to check if the execution of CSG should be terminated (line 5 of Algorithm 1). The code used to perform CSG on RPGLite is available at [70], output files are not included in the repository to limit memory usage, but an abbreviated file from an early iteration of CSG is included in Appendix B. In the example the prefix is lines 1-33, player 1 is lines 35-39, player 2 is lines 40-51 (unabbreviated they would be lines 40 - 62,250) and the suffix is lines 53-72. Explicitly stating the action from each reachable state requires considerably more description than listing the available actions, hence why player 2 requires over 60,000 lines of code as opposed to 4 lines for player 1.

CSG in this form does not contextualise the opponent in the states. More precisely, each state does not explicitly describe what character is in what *position*, only that some character is in position 1 for each player and another is in position 2. This means that the learning which occurs between iterations of CSG, where adversarial strategies are calculated, could exploit the position of the current meta material. To counteract this we consider both orderings of all pairs (e.g.: we consider KA and AK pairs) and use the worse of the two as the true value for the pair. The modelling of positions, rather than characters explicitly is important to the working of CSG, without it educated strategies would have "gaps" against material that has not been encountered. For example, if we have found KA to be a meta strategy and KW is the most effective against it, we need to encode an informed KW strategy against the meta KA strategy that has a strategy against AW, material that has not been considered. This is an abstraction that will increase the disparity between strategies identified through CSG and by actual players, but considering all orderings and using the less effective one goes some way to reconciling these differences.

We obtained results using a desktop machine running Ubuntu 18.10 with an Intel® Core™ *i*7−7700 CPU with 2×8GB of RAM. Identifying dominant strategies took between 10 and 20 minutes and required 5GB of memory, per configuration. The largest of the models has 816,480 states and was built by our machine in 4.38 seconds. Each execution took between 20 and 120 minutes and required 8GB of memory. It takes approximately 20 seconds to build each model and fewer than 2 seconds to calculate the probability of a player winning and generating an optimal strategy. The majority of the time is spent converting the files generated by PRISM into adversarial strategies.

CSG terminates reasonably quickly with manageable hardware requirements for the case study used. Effectively each iteration discounts large tranches of the strategy space as new states

are visited at which an optimal action is calculated. For RPGLite, there are several actions available which are always strictly worse than others, for example it is never optimal to skip in this version of RPGLite, nor is it optimal for the Archer to attack a single target when they can attack two. When states such as this are considered, the number of strategies which feasibly could be effective or indeed dominant is significantly reduced.

4.5 Results

We define several configurations for RPGLite which we analyse using the approach of Section 4.3. The configurations are described in Table 4.3. These configurations have been developed over a period of time to illustrate how CSG can be used during game development. Having started with a configuration with minor differences between the material, new configurations were created in response to the results gathered from analysis of previous configurations. Candidate configurations are created and analysed for effectiveness. Our approach can also identify the material and general strategies that are weak, informing decisions about future configurations. We check every configuration for dominant strategies and search for dominated material six times, using a random seed strategy each time. We make sure that all material was selected for the seed strategy at least once in the six executions. Configurations A - D were created in series, with small changes based on the results of CSG on the previous configuration, but E was found to have interesting properties and is included to highlight the possibility of multiple terminal cycles, discussed later.

An example of how we devised the configurations is as follows. Analysis of CSG results show a dominant strategy for KA in configuration A. To counteract this we decided to make the Wizard stronger by increasing Wizard accuracy in configuration B by 0.1, to 0.85. We wanted to change configurations minimally, to allow us to examine the effect of small changes on strategies employed at high-level competitive play. We also wanted to use “friendly” values (multiples of 5 or 10) rather than highly specific values, as this is more in keeping with current game design best-practice.

4.5.1 Dominant Strategy Identification

The results of our dominant strategy identification technique are shown in Table 4.4. Rows represent the optimal strategy for a pair when playing against itself, columns represent the adversarial probability for a pair against a strategy. Therefore a cell represents the probability of the column material winning against the optimal strategy for the row material. The two dominant strategies are highlighted (other material cannot beat them with probability greater than 0.5). In the KA row for configuration A there is no value greater than 0.5, this shows that there is a dominant strategy for a Knight-Archer pair. A player adopting the optimal strategy for KA in KAvKA cannot be beaten with a probability greater than 0.5 by any strategy for any opposing material. In a

Config.	Character	Health	Accuracy	Damage
A	Knight	8	0.70	3
	Archer	7	0.80	2
	Wizard	7	0.75	2
B	Knight	8	0.70	3
	Archer	7	0.80	2
	Wizard	7	0.85	2
C	Knight	8	0.70	3
	Archer	6	0.80	2
	Wizard	7	0.85	2
D	Knight	9	0.70	3
	Archer	7	0.80	2
	Wizard	7	0.85	2
E	Knight	9	0.90	2
	Archer	6	0.60	2
	Wizard	8	0.60	2

Table 4.3: Configurations for RPGLite, buffs highlighted blue, nerfs highlighted red.

real world example, players would eventually discover this strategy. A competitively motivated player aware of such a strategy has no motivation to use any material other than KA. For this reason we state that configuration A is uninteresting to play.

4.5.2 Dominated Material Identification

The results of identifying dominated material are shown in Figure 4.3–Figure 4.8. A figure is included for all configurations showing a single execution of the search for dominated material, labelling all material at every iteration, on the left and showing 6 simultaneous executions to demonstrate the repeating patterns found on the right. The individual plots show all unique strategies identified, ending when a cycle is found, with a line showing the trend in adversarial probabilities. A zoomed-in sample is given in Figure 4.4 to further aid description. The grouped executions graphs plot only the highest probability of winning with any material against the previously identified adversary or seed. The seed material is displayed in brackets for each execution. The seed strategy is different for each execution, having been generated at run-time with a full, pure strategy chosen at random.

We can show with CSG that gameplay will tend towards a dominant strategy if one exists, as illustrated in Figure 4.3 (left). The adversarial probabilities reach 0.5 and then stay at 0.5 as the best way to play against the previous strategy is shown to be to play the same strategy. When

Config.	Material choice	Opposing material		
		KA	KW	AW
A	KA	0.500	0.470	0.483
	KW	0.623	0.500	0.723
	AW	0.579	0.409	0.500
B	KA	0.500	0.594	0.546
	KW	0.559	0.500	0.617
	AW	0.528	0.449	0.500
C	KA	0.500	0.716	0.665
	KW	0.434	0.500	0.532
	AW	0.500	0.516	0.500
D	KA	0.500	0.625	0.350
	KW	0.472	0.500	0.526
	AW	0.716	0.526	0.500
E	KA	0.500	0.539	0.368
	KW	0.479	0.500	0.424
	AW	0.641	0.625	0.500

Table 4.4: Comparison of adversarial probabilities against optimal strategies for the same material in all 5 configurations.

a dominant strategy is present in a given configuration of a game, executions of CSG tend to show the strategies identified converging to it. Figure 4.3 (right) shows how every execution of CSG performed on configuration A converges to the same probability of 0.5. By examining the strategies we can confirm that the same (dominant) strategy is identified in each instance. Our implementation of CSG stops when a strategy is identified which is identical to some strategy generated before. If a strategy is identical to the strategy generated immediately before it, then that strategy is dominant. A game developer would endeavour to develop a game for which a longer cycle of effective strategies exists.

Configuration B has a cycle of four effective strategies which are identified in every execution, as shown in Figure 4.5 (left). Figure 4.5 (right) shows that eventually the same cycle is identified in multiple executions of CSG, albeit out of step. Recall that termination of CSG is contingent on a single dominant strategy being identified or on a cycle of effective, non-dominant strategies being identified (at which point no new strategies will be found), the latter occurred in each execution for B. In the cycle of strategies there is at least one strategy for each material. This shows that high-level competitive play of RPGLite using configuration B would at various points employ all material, which is something game developers would strive for. Once the strategies

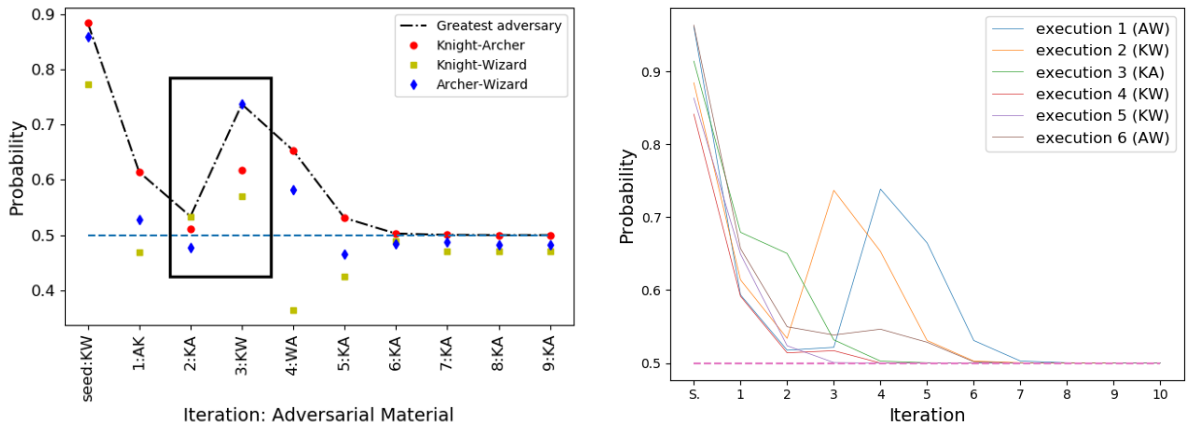

Figure 4.3: Configuration A: CSG. (left) A single execution and (right) multiple executions. The boxed area (left) is shown in detail in Figure 4.4.

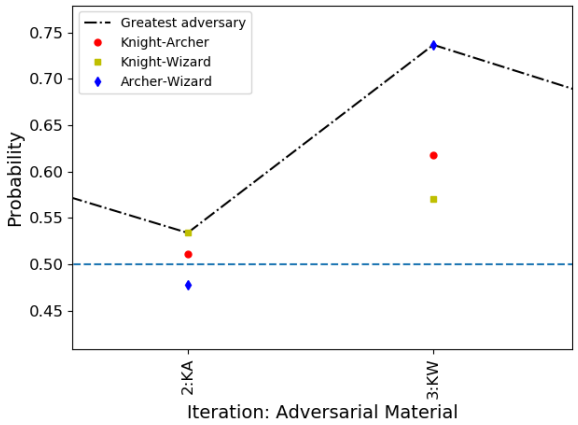

Figure 4.4: Configuration A: A closer examination of the boxed area from Figure 4.3 (left). The points represent the maximum probability of winning against the previously identified best strategy using the material denoted. The blue diamond at the top of iteration 3 is the maximum probability a player using AW can win by against the KW strategy in iteration 2. Iteration 4 will show the maximum probabilities achievable against the AW strategy in iteration 3.

settle for configuration B (from iteration 3 and beyond in Figure 4.5) there is a strategy for all material against every effective strategy identified which can win more often than it loses. This is shown by all values being above the blue line at probability 0.5.

Configurations C and D converge on a cycle of effective strategies and eventually identify the same cycles of strategies every time, as shown in Figure 4.6 and Figure 4.7. Configuration C has a cycle of length 8 whilst D has a cycle of length 6. Both cycles include strategies for all material, although unlike configuration B, some effective strategies cannot be beaten with a probability greater than 0.5 by any strategy for specific material. For example in Figure 4.7 (left) the value for AW at iteration 17 is 0.331. This means that the most effective strategy using AW against the strategy identified for KA at iteration 16 wins less than a third of the time.

The results of searching for dominated material with configuration E are important for two

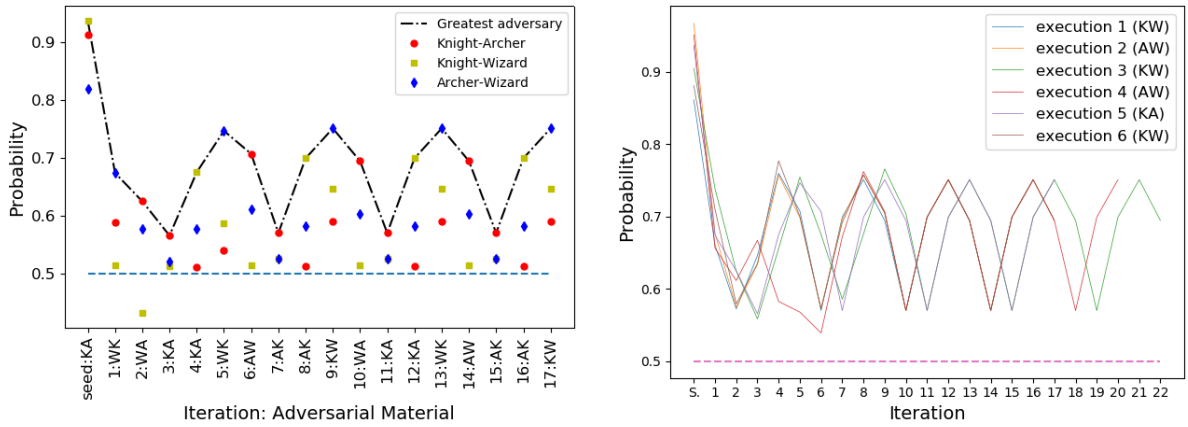

Figure 4.5: Configuration B: CSG. (left) A single execution and (right) multiple executions.

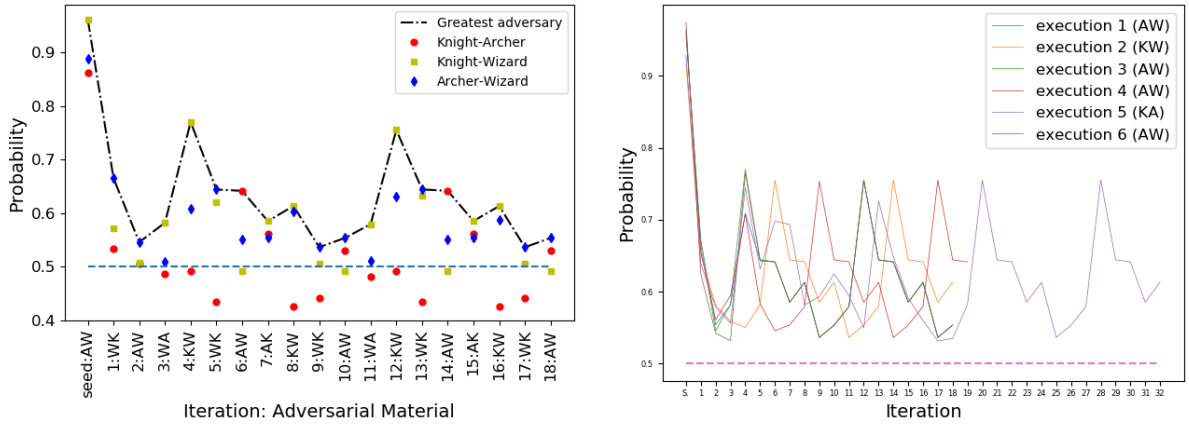

Figure 4.6: Configuration C: CSG. (left) A single execution and (right) multiple executions.

reasons. First, because there is a dominant strategy for KW in configuration E, as shown in Table 4.4 that is not identified in any of the six executions of CSG (unlike for configuration A). Second, two different cycles of effective strategies are identified, one of length 3 shown in Figure 4.8 (top-left) and in executions 1, 4 and 6 of (bottom), and the other of length 16 (top-right) and in executions 2, 3 and 5 of (bottom). The second cycle is found for a seed strategy using all material. Despite this, analysing the results shows clearly that AW is dominated by the other material for configuration E. Once the strategies have settled in both cycles, there are no strategies for AW that can win with a probability greater than 0.5 against any effective strategy. In fact, AW is the worst choice of material at every iteration once the strategies have settled, suggesting it would never be used during high-level play.

Statistical analysis of the results of CSG allow for clearer comparisons of how strategies develop across different configurations. Table 4.5 presents the measures described in Section 4.3.2 for the 5 configurations.

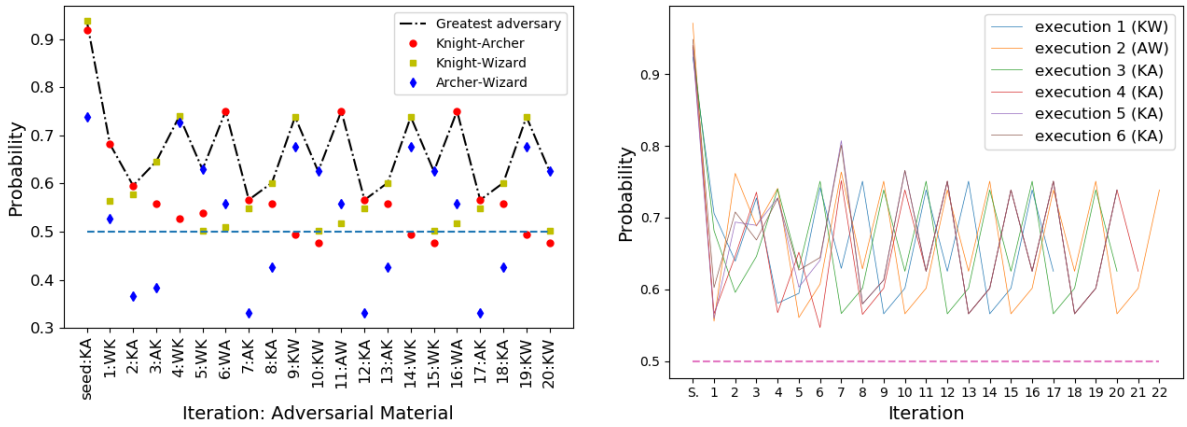

Figure 4.7: Configuration D: CSG. (left) A single execution and (right) multiple executions.

4.6 Analysis of Results

We claimed that a fair game is one where all material can win by similar, significant amounts, therefore configurations A and E are less fair than the others. The results for $\delta^{win}(KW)$ in A and $\delta^{win}(AW)$ in E (0.005 and 0.0 respectively) as well as the consistently low values for the other win deltas demonstrate this. Furthermore, the material and mean robustness values, compared in Figure 4.9, show that they are uninteresting to play too. For both configurations mean robustness is only slightly above 0.5 meaning the choice of winning strategies for players is limited. KW is dominated in configuration A and AW is dominated in configuration E, as shown by their low robustness values. Both configurations A and E can be discounted as unbalanced.

Analysis of configurations C and D show why it is important to consider the delta values as well as material robustness. Consider KA in configuration C and AW in configuration D, both have significantly lower values for robustness than the other material. However, we can use the delta values to show that AW is viable in configuration D whilst KA is not viable in configuration C. Comparing the loss deltas with other material illustrates the risk associated with playing the material, both are far lower than the alternatives. The win deltas are more significant, in C this is 0.029 for KA compared to 0.084 and 0.073 for KW and AW respectively, suggesting that the risk outweighs the potential reward for playing KA. In D, the win delta for AW is 0.074 compared to 0.071 and 0.092 for KA and KW respectively, a far more justifiable risk to the player, given the potential reward. Although it may appear that AW is dominated in configuration D, the delta values show that it is highly viable, but only at certain times.

Having shown that configurations A and E are imbalanced and that KA is too weak in configuration C, we are left only with configurations B and D. B has the greatest μ robustness, therefore it could be argued that it is the most *interesting*, but D has greater outplay potential, suggesting it has a more varied metagame, that the meta strategies would be more different from each other. Game developers who want to decide on a configuration based on these results could justifiably choose either configuration B or D as the optimal one depending on the type of game they wanted

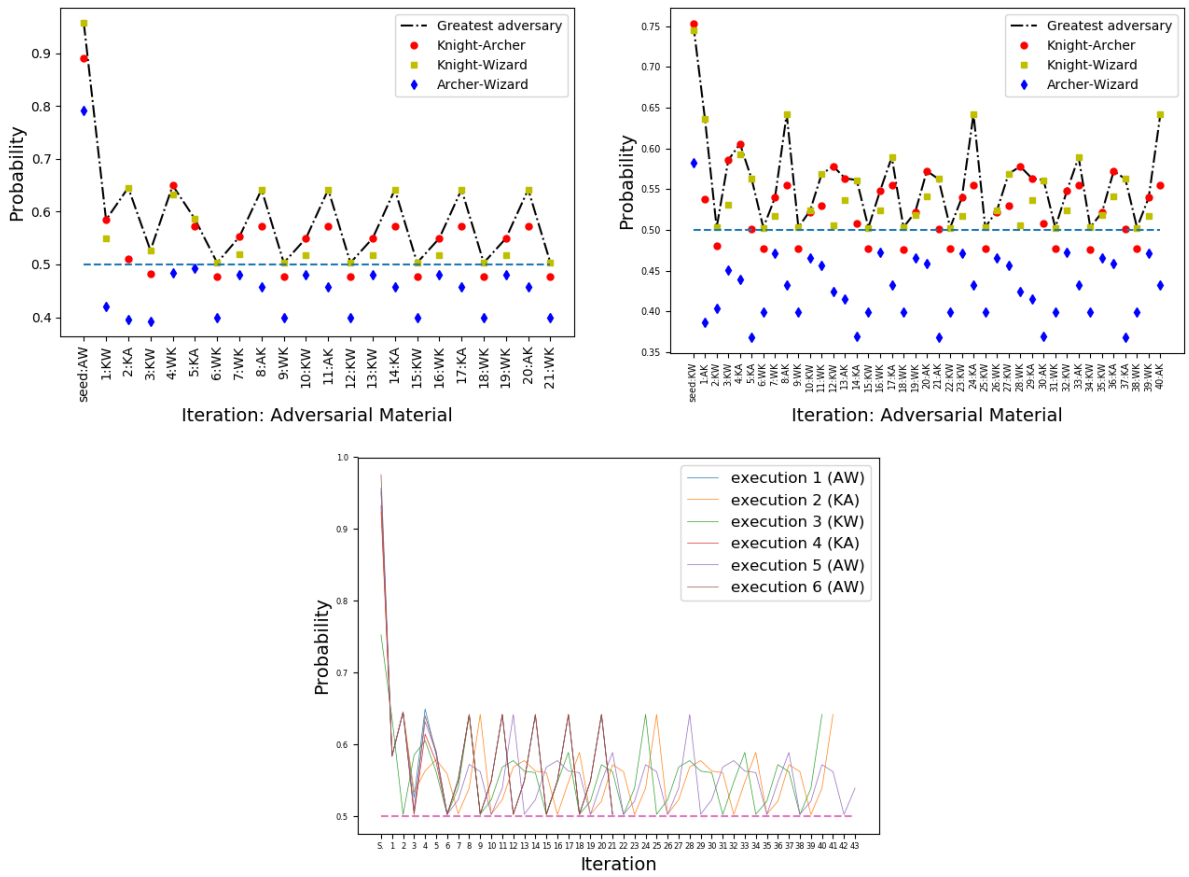

Figure 4.8: Configuration E: CSG. (top-left, top-right) Single executions and (bottom) multiple executions.

to make.

Configuration B would be a more player friendly game then D because it is always possible to win with any material against the effective strategies identified for B and the loss deltas are 0.0 or 0.001 meaning there is almost always a winning counter for all material against any effective strategy. Figure 4.5 shows a clear cyclical hierarchy for configuration B where KA beats AW, AW beats KW and KW beats KA. Configuration D has the same hierarchy, Figure 4.7 shows that this is less clearly defined than in B, as many optimal strategies are only slightly better than strategies using different material. The value of 0.122 for $\delta^{win}(AW)$ in B suggests that AW is under powered compared to the other material, whereas the values for configuration D are less varied.

A key point to note about the cycle of effective strategies identified for configurations B, C and D is that they include at least one strategy using each material. This property alone is enough to show that a game is well developed as natural competitive play employs all of the choices offered to the players, which can only be positive for game developers. This is an example of orthogonally differentiated game material constituting an intransitive relationship at high-level play, the confirmation of which is one of the aims of this work.

Metric	Configurations				
	A	**B**	**C**	**D**	**E**
$\rho(\text{KA})$	0.524	0.589	0.505	0.565	0.528
$\delta^{win}(\text{KA})$	0.024	0.089	0.029	0.071	0.034
$\delta^{loss}(\text{KA})$	0	0	0.024	0.006	0.006
$\rho(\text{KW})$	0.472	0.589	0.582	0.592	0.543
$\delta^{win}(\text{KW})$	0.005	0.09	0.084	0.092	0.043
$\delta^{loss}(\text{KW})$	0.033	0.001	0.002	0	0
$\rho(\text{AW})$	0.505	0.622	0.573	0.523	0.433
$\delta^{win}(\text{AW})$	0.019	0.122	0.073	0.074	0
$\delta^{loss}(\text{AW})$	0.014	0	0	0.051	0.066
outplayPotential	0.033	0.082	0.062	0.096	0.052
μ_ρ	0.5	0.6	0.553	0.56	0.502

Table 4.5: Statistical analysis for the five configurations considered based on the three material metrics (robustness, win delta and loss delta) and the two game configuration metrics (outplay potential and mean robustness).

4.7 Discussion

The results of CSG on our case study were not what we expected in a number of ways. The fact that every execution terminates in a cycle of strategies in a relatively short number of iterations was surprising given the number of strategies available. We assumed that the results would reflect those for identifying dominant strategies, or that we would not necessarily get more information than simply identifying the probability of winning calculated for all material playing optimally against all others. This was not the case – a great deal of extra context is given by generating strategies in turn. For example, simply calculating optimal values would give no appreciation of the risk associated with using specific material or of the extent to which material is dominated. We were surprised by the dramatic effects caused by small changes in the configuration of the game. Some configurations change only slightly from previous ones: C was created from B by reducing the health value of the archer from 7 to 6, while D was created from B by increasing the health value of the knight from 8 to 9. However, the differences in the results between configurations is profound, demonstrating the value of CSG.

With CSG we have shown that model checking can be used to quickly present game developers with information they would not have been able to discover without extensive testing or the use of player data. Although our methods only model how the game will be played, when one considers the cost of generating similar results, the benefits are clear.

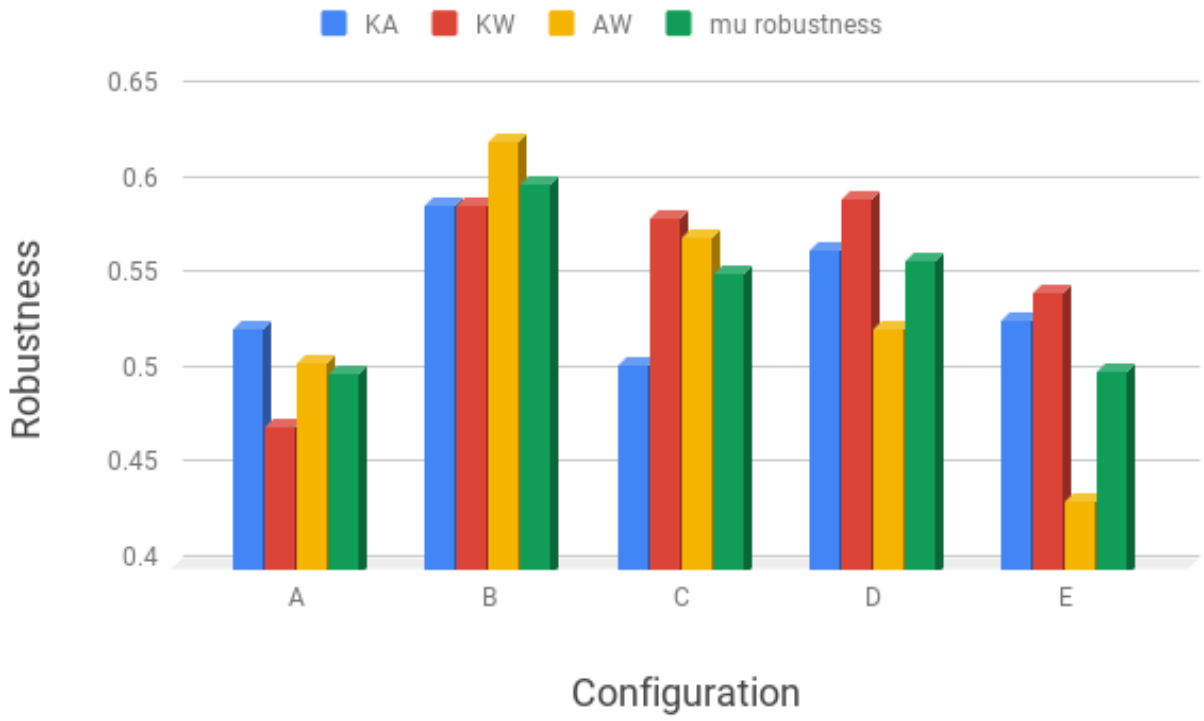

Figure 4.9: Robustness scores for all material and mean robustness for all configurations.

4.7.1 Strategies Generated

Executions of CSG create multiple files describing the meta strategy found at each iteration as PRISM code that models player 1 using that strategy. These files are very large (over 100,000 lines of code) and very dense, each line either being a PRISM guard or a matching command. The following snippets come from a strategy generated as the 21^{st} adversary under configuration D:

```
[p1_turn_3]      attack = 0 & turn = 1 & p2c1 = 2 & p2c2 = 2
    & p2_stun = 0 & p1c1 = 2 & p1c2 = 2 & p1_stun = 0 ->
        (attack' = 3) & (p1_stun' = 0);
...
[p1_turn_4]      attack = 0 & turn = 1 & p2c1 = 2 & p2c2 = 2 &
    p2_stun = 0 & p1c1 = 2 & p1c2 = 3 & p1_stun = 0 ->
        (attack' = 4) & (p1_stun' = 0);
```

The material used here is KW, attack = 3 refers to Wizard attacks *p2c1* and attack = 4 refers to Wizard attacks *p2c2*. This snippet details the actions taken from a state where all characters have 2 health remaining and none are stunned and where all characters have 2 health apart from *p1c2* which has health 3 and no characters are stunned. The strategy dictates that when all characters have health equal to 2, player 1 attacks the first character with their Wizard, but when their Wizard has 3 health, player 1 attacks the second character with their wizard. This strategy was

state	p1W_p2K	p1W_p2W
p1K=2, p1W=2, p2K=2, p2W=2	0.468	0.500
p1K=2, p1W=3, p2K=2, p2W=2	0.593	0.552

Table 4.6: Comparison of optimal probabilities for KW vs WK using the 18thth adversary found from CSG under configuration D

learnt against a WK opponent. This result is interesting because it is not apparent why one of the player's characters having more health should affect the action they take. By constructing an intermediate model and restricting the move from the listed states to p1W_p2W and p1W_p2K we compare the resulting optimal probabilities, given in Table 4.6. The disparity in optimal action is due to the Wizard only causing damage 2, so a Wizard with 3 health can survive a single successful Wizard attack. This illustrates how effective model checking is for finding solutions that human players may struggle to identify. Numerous such examples appear in every matchup, under every configuration.

Modelling individual players of a game would involve representations of game knowledge and learning, including how stimulus alters understanding, all run synchronously with different agents to represent player types. With CSG, we can get results without having to synthesise player understanding. What CSG is modelling rather is a community of players all working together to find the current best way of playing. Solving globally requires considering all strategies for all materials against all other strategies for all other materials. We assume that human players will not be capable of calculations such as these and we instead model learning locally where the opposition (the current meta) is fixed, this means that only strategies for materials against a given strategy for a known material need to be considered.

4.7.2 Limitations

CSG works on RPGLite 1 because RPGLite 1 is small. Games last only a few turns and take less than a minute. It is a simple game that would not entertain players for more than a few plays. Additionally, there is not much variance in the way the game can be played and most of the decisions are straight forward. Despite the size of the game, executing the algorithm is relatively slow and the chained nature, where previous results are used in future iterations, would make parallelism difficult to implement.

In order to implement CSG for a game, a designer would need a strong grounding in model checking, which is not common. The nature of model checking requires each state to be handled individually without abstraction techniques which would need to be purpose-built for each new game. However, despite this initial effort our technique makes it easy to compare different configurations of a game and to get objective measures of balance which even extensive player testing cannot provide. It is also possible that CSG (or a simplification of it) could be built into

Figure 4.10: The Rogue and Healer from RPGLite 2

a system designed for balance analysis which is more intuitive for game designers. Such a tool may not require model checking experience to operate.

4.8 Advancement on CSG

In this section we improve upon the techniques of CSG to deal with more complex games and expand upon the depth of insights available. We also expand upon the case study as these new techniques can more efficiently cope more detailed game systems.

4.8.1 RPGLite 2: An Extension

RPGLite 1 allowed players a choice of 3 material sets, having 3 units (the characters) and a set size of 2 (the character *pairs*). An important consideration for CSG was how it handled growth of a game for which it had already been configured. The release of new game materials long after a game is first published is common. If the system used to balance a game had to be entirely remade every time new material was introduced then it would not be an effective solution. In order to test how well our approach adapts to game evolution we created an extended version: RPGLite 2. For this new version two additional characters (Figure 4.10) were added and a team size of 3 was experimented with. Whilst CSG could still be performed with 5 characters and pairs, using triples and 5 characters was too complex to model and would require specialist hardware to parse results files of the magnitude generated.

The Rogue and The Healer

The two characters added to RPGLite 2 both have a further attribute in addition to the *health*, *accuracy* and *damage* attributes of the original 3 characters. The Rogue has an *execute* attribute, if the Rogue successfully attacks an opponent with at most the value of *execute*, then that character drops to 0 health. The Healer has a *heal* attribute, which allows it to heal (increase the health of) either itself or a chosen ally by the value of *heal*, up to a maximum of the target's *health* attribute. A configuration of RPGLite 2 consists of values for 17 attributes describing all 5 characters, including *health*, *damage* and *accuracy* for all characters, and a value for *heal* and *execute*.

4.8.2 Updating Strategy Encoding

The same approach used for RPGLite with 3 characters can be expanded for 5 characters. However the issue of learning without context is exacerbated with more characters. Strategies in CSG for RPGLite 1 are based on the opponent's first or second character, with 5 characters either of those could be 4 of the 5 in the game (i.e., a Knight could never be the first character), so a strategy learnt to counter a Knight could be being used to counter an Archer, a Wizard or a Rogue, for example. To prevent this from causing CSG to generate uninformed strategies for RPGLite 2, the representation of strategies was changed. The code used for CSG on RPGLite 2 is available at [71].

For RPGLite 1, a player's characters were represented as p_c1 and p_c2, denoting their first and second character, this was changed to p_K, p_A, and so on, to represent each character a player could potentially have. This means that a state in RPGLite 2 is represented by the 14-tuple:

(attack, turn, p1K, p1A, p1W, p1R, p1H, p1_stun, p2K, p2A, p2W, p2R, p2H, p2_stun)

This change means strategies are generated with full knowledge of what the opposing character can do at the cost of slightly larger models and slower strategy generation. With RPGLite 1 pairs were considered in either permutation (e.g.: KA and AK) which solved the issue as there were only 2 possible characters for each position, the same would not be necessary for RPGLite 2.

To ensure all states are considered by the model checker against all opposing materials, the naive seed strategy used at the beginning of CSG is also updated. Rather than a naive strategy for a randomly selected material, naive material selection, followed by a naive choice of all possible actions is used as the seed. Naive material selection takes the form of an initial transition to one of the 10 pairs being chosen by the seed player, each with a probability of 0.1. For RPGLite 1 the different representation of states meant every state was considered so this was not necessary.

Strategy generation is simplified where all action states from which only a single non-skip action is available, or only skipping is available, are disregarded from generation and instead grouped into a single guard which matched multiple states (we refer to this as *clumping*). In

addition, we assume that when a player can perform an Archer action targetting multiple opponents, they will do this rather than target a single opponent. For example, all KA strategies begin with guard-commands of the following form:

```
[p2] attack = 0 & turn = 2 & p2K > 0 & p2_stun = 3 & p2W > 0 &
  p1K > 0 & p1A+p1W+p1R+p1H = 0 ->
    (attack' = 56) & (p2_stun' = 0);
[p2] attack = 0 & turn = 2 & p2A > 0 & p2_stun = 5 & p2H > 0 &
  p1K > 0 & p1A > 0 & p1W+p1R+p1H = 0 ->
    (attack' = 62) & (p2_stun' = 0);
```

In this snippet, the clumped guards lead to commands for `p2K_p1K` as the Wizard is stunned (attack 56) and `p2A_p1Kp1A` as Archer's attacking multiple opponents is preferred (attack 62), respectively.

The final change is that at each iteration a strategy is generated for all materials, rather than just the optimal probability being calculated. The strategies are updated at each future iteration with the number of viewed transitions and the number of updated transitions being reported to give an indication of the extent of change between iterations. This also allows for more specific comparison of strategies to determine whether a cycle has been identified, rather than simply considering optimal values.

The same technique of generating portions of PRISM specifications of MDPs representing a fixed strategy for player 1 against a non-deterministically chosen player 2 is used. The adversarial strategy for player 2 is synthesised and then translated into a strategy for player 1 to be used in future analysis. The models constructed are often smaller than they were for RPGLite 1 as they are better described (primarily due to clumping).

4.8.3 Results of CSG on RPGLite 2

Consider the configurations of RPGLite 2.0 given in Table 4.7. Full CSG analysis identifies a dominant strategy for a Rogue-Healer pair under Z_1 and a 4-cycle of effective, non-dominant strategies under Z_2, demonstrated in Figure 4.11.

From the peaks on the CSG results graph we see that several strategies for various materials under Z_1 are considered *meta* (i.e., the best way of playing), before the eventual dominant Rogue-Healer strategy is identified. At various points in the execution of CSG strategies for KA, KH, AH, WR, WH and RH were considered meta. This diversity suggests the game would have a fairly *healthy* metagame before the dominant strategy was identified. It should also be noted that the dominant RH strategy is not overly dominant, KW can guarantee winning with a probability of 0.494 against it, KA 0.482 and KH 0.482 (to 3dp). It is difficult to predict whether players would notice that the strategy was dominant, with optimal values using other material so close to 0.5. The results for Z_2 show a 4-cycle of effective, non-dominant strategies. The materials used

Config.	Character	Health	Accuracy	Damage	Execute/Heal
Z_1	Knight	11	0.65	3	
	Archer	7	0.85	2	
	Wizard	6	0.9	2	
	Rogue	7	0.65	3	5
	Healer	8	0.65	2	2
Z_2	Knight	11	0.75	3	
	Archer	7	0.95	2	
	Wizard	6	0.9	2	
	Rogue	7	0.65	3	5
	Healer	8	0.65	2	2

Table 4.7: Configurations for RPGLite 2.0, buffs highlighted blue.

in the cycle of effective strategies are WR, KH, AH and KA. This is significant as each material unit is used in at least one of these sets, i.e., under Z_2, all material units can be considered optimal, whilst no material sets are dominant. The cycle seems to appear after 20 iterations, but there are in-fact small changes to the strategies generated after this point. It is only after 52 iterations that they finally settle.

4.8.4 Analysis of CSG on RPGLite 2

We claim that a balanced game is one where there are numerous distinct viable ways of playing. One way to measure the variety of strategies with CSG which the expanded version offers is the number of actions which are updated for each material's strategy at every iteration. For all material at every iteration the total number of actions, the number of actions *seen* and the number of states updated are calculated. The number of actions seen is determined by the material and strategy used by the opponent. Over the entire execution of CSG with a varied metagame most actions will be seen, and therefore the final strategies will have informed actions at most states. The number of actions updated, of those seen, could be indicative of the extent to which players need to adapt their strategies to the current meta.

The proportions of seen actions that were updated can be plotted for all material to give an indication of the level of strategic depth available to players using that material. Examples for Z_1 and Z_2 are given in Figure 4.12. Higher values represent a more significant shift in the adversarial strategy used by the material. This information could be used by game developers to identify material which has a high degree of variability in the strategies available. The results shown are what one would expect having considered the CSG results, i.e., Strategies under Z_2 exhibit little to no change after roughly 25 iterations as the algorithm begins to converge on a small subset of materials.

As the strategies synthesised during CSG develop from each other, a later strategy can be

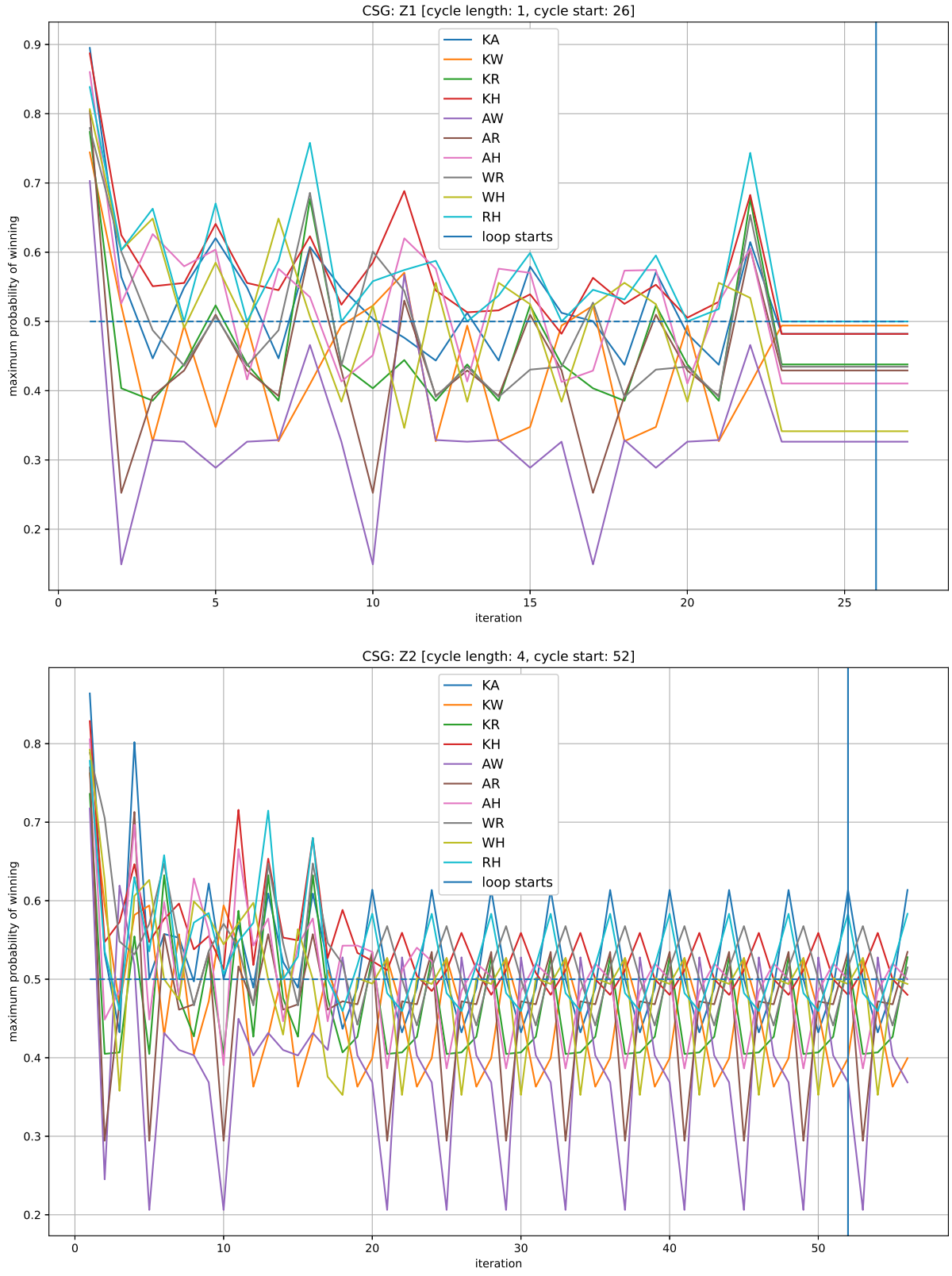

Figure 4.11: CSG performed on configurations Z_1 (above) and Z_2 (below). A vertical line denotes where all identified strategies are identical to those in the final iteration.

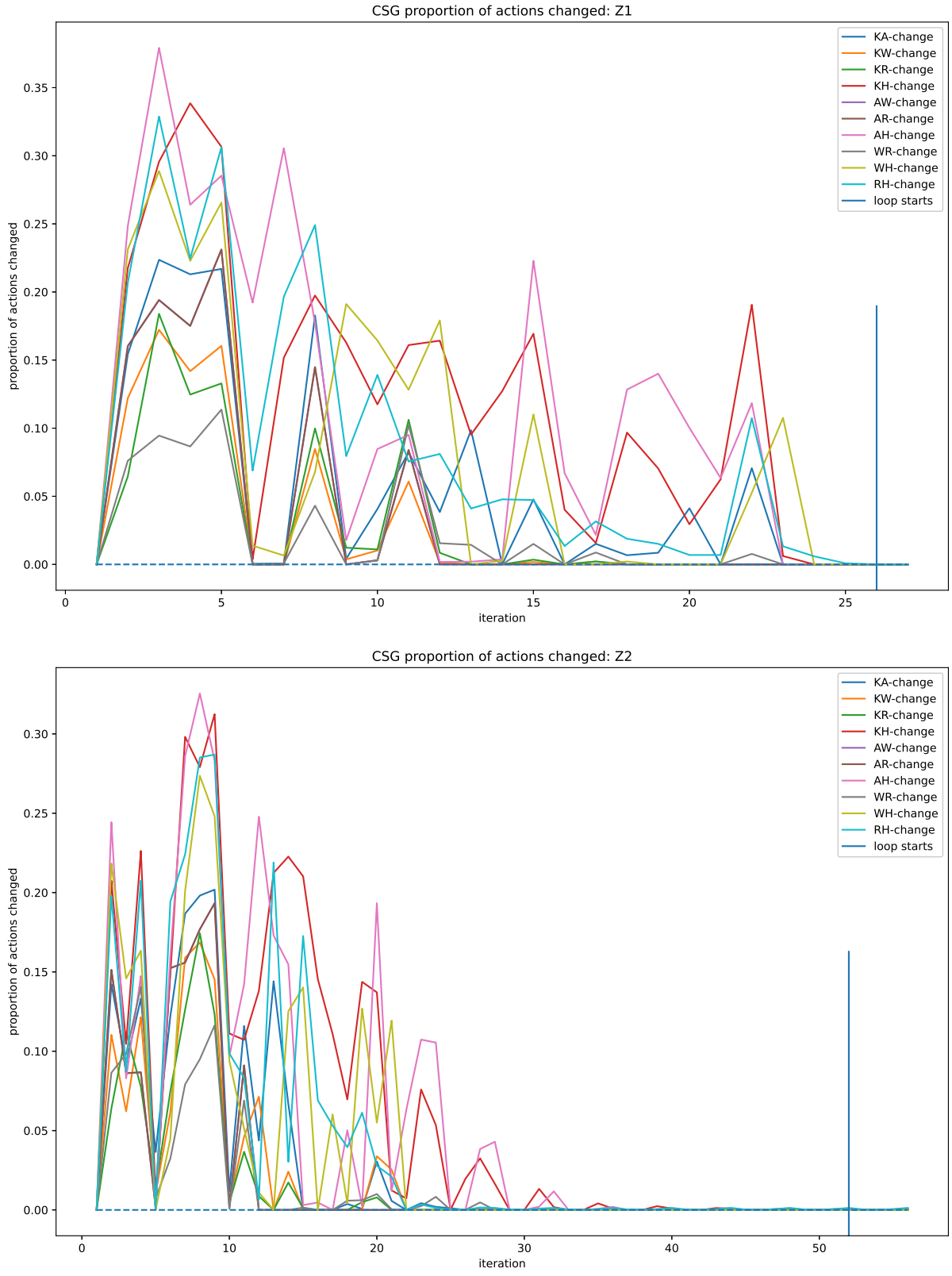

Figure 4.12: Proportion of actions changed during CSG performed on configurations Z_1 (above) and Z_2 (below).

$\mathbf{Z}_1$	KA	KW	KR	KH	AW	AR	AH	WR	WH	RH
KA	0.5	0.477	0.596	0.437	0.850	0.747	0.571	0.472	0.614	0.482
KW	0.522	0.5	0.406	0.327	0.625	0.454	0.347	0.570	0.409	0.494
KR	0.403	0.593	0.5	0.385	0.638	0.434	0.523	0.444	0.676	0.437
KH	0.562	0.672	0.614	0.5	0.671	0.607	0.472	0.608	0.682	0.482
AW	0.149	0.374	0.361	0.328	0.5	0.184	0.288	0.567	0.466	0.326
AR	0.252	0.545	0.565	0.392	0.815	0.5	0.509	0.530	0.605	0.429
AH	0.428	0.652	0.476	0.527	0.711	0.490	0.5	0.569	0.605	0.410
WR	0.527	0.429	0.555	0.391	0.432	0.469	0.430	0.5	0.653	0.434
WH	0.385	0.590	0.323	0.317	0.533	0.394	0.394	0.346	0.5	0.341
RH	0.517	0.505	0.562	0.517	0.673	0.570	0.589	0.565	0.658	0.5

$\mathbf{Z}_2$	KA	KW	KR	KH	AW	AR	AH	WR	WH	RH
KA	0.5	0.521	0.595	0.488	0.793	0.705	0.613	0.432	0.476	0.517
KW	0.478	0.5	0.412	0.363	0.550	0.483	0.399	0.459	0.428	0.511
KR	0.404	0.587	0.5	0.426	0.554	0.411	0.528	0.406	0.632	0.475
KH	0.511	0.636	0.573	0.5	0.597	0.531	0.480	0.558	0.498	0.482
AW	0.206	0.449	0.445	0.402	0.5	0.287	0.368	0.527	0.431	0.409
AR	0.294	0.516	0.588	0.468	0.712	0.5	0.534	0.471	0.557	0.461
AH	0.386	0.600	0.471	0.519	0.631	0.465	0.5	0.485	0.542	0.416
WR	0.567	0.540	0.593	0.441	0.472	0.528	0.514	0.5	0.647	0.540
WH	0.523	0.571	0.367	0.501	0.568	0.442	0.457	0.352	0.5	0.375
RH	0.482	0.488	0.524	0.517	0.590	0.538	0.583	0.459	0.624	0.5

Table 4.8: Matchup tables from final strategies synthesised through CSG for Z_1 and Z_2. Values given are row vs column.

considered *better* than a previous one, in that it is formed in response to more opposing strategies. CSG stores strategies for all materials at every stage including at the final iteration where a loop is identified. These strategies were used to develop matchup charts (one of the key tools used by developers to analyse balance) for informed RPGLite 2 strategies. Example charts for configurations Z_1 and Z_2 are given in Table 4.8.

Our matchup charts give a good indication of the relationships between the material and can be used to identify the desirable cyclical relationships. As RH is dominant under Z_1 there are no values in the row for RH in Z_1 which are below 0.5. For Z_2 there is a value that is above and a value below 0.5 in every row, showing all materials can be used viably at some point in the metagame.

The strategies found at the end of a CSG execution are effective strategies, but they are not guaranteed to be optimal. The formation of the strategies depends on what materials were seen as part of the metagame, if any materials were never seen then the final strategies will not be informed in terms of how to deal with those materials. This issue is similar to a flaw in the rate of change results, there will be a spike in the rate of change if a novel material is identified as meta. This is understandable, the strategies only have to account for meta materials so will not

have adapted for these materials before. In this way, the final strategies found are less effective against the weakest materials (which have never been meta) than they should be.

The rate of change is too incidental to be used as a reliable metric for material analysis and the final strategies identified from CSG can be underdeveloped if the metagame does not cover the entire material set, which is unlikely. A better strategy to consider for the matchup analysis would be the optimal strategy for all materials, which can be synthesised using SMGs and PRISM-Games. An optimal strategy will be globally optimal, as opposed to the final CSG strategies, which will be locally optimal.

4.8.5 Limits of RPGLite for CSG

Performing CSG for larger systems is increasingly difficult. A working implementation of RPGLite for 5 characters played with teams of 3 seemed promising initially, but the strategies described for the equivalent models were too complex. Using a machine that allowed for the configuration of PRISM with 600gb of RAM available for building the models enabled the creation of 3-on-a-team RPGLite 2 models in 20 minutes and the calculation of optimal strategies in 20-30 minutes. Full CSG performed at this level of complexity would take several hours at each iteration, it was decided that ultimately this was not a sensible avenue to pursue.

In the CSG visualisations for RPGLite 2, a vertical line showing when the loop begins is included for clarity. However, the figures are difficult to interpret and the results are better comprehended numerically. The number of iterations before convergence is higher than with only 3 characters in total and at each iteration more computation is required (as 10 material selections are considered, rather than 3). Configurations with higher health attributes significantly expand the state space and would therefore slow down computation, however the values used represent a more interesting game. Chipping away at large health pools is ultimately unsatisfying and any advantages from the material are emphasised, removing tension from the outcome of a game.

4.9 Optimality Networks

In this section we introduce optimality networks, an alternative to the iterative form of chained strategy generation.

4.9.1 Methodology

CSG is a useful tool for creating a representation of the metagame. For more complex games it will slow down and eventually be too expensive for the information it offers. The insight gained is also prone to incidental factors which move the strategies generated away from the realistic play that was the intention of the modelling. CSG for RPGLite 1 was presented as two complimentary techniques, the chained portion was intended to identify dominated material, the

other technique – to identify dominant material – did not have an element of *chaining*. The use of optimal strategies can be used in a way which improves upon both of the original techniques presented as CSG.

The updated form of modelling used for RPGLite 2 (described in Section 4.8.2) allow for SMG model descriptions of the game. With SMGs, PRISM-Games can be used to identify optimal strategies, the best strategy for a player when their opponent also plays their best strategy Section 3.4.1. By representing material selection as a non-deterministic choice for an opponent, we can use optimal strategy synthesis to identify *counter* material and the extent of these relationships. The technique involves the following steps.

1. For each material set:
   (a) generate a model for that set with nondeterministic action choice against an opponent with nondeterministic action choice and nondeterministic material selection;
   (b) calculate the optimal value that the material can guarantee;
   (c) identify the counter material set used against it.
2. Create the directed graph representing the counter materials and reason about dominance with loop detection.

For RPGLite 2, the 10 models needed can be created and verified in under a minute. An example of the models generated to synthesise optimal strategies is given in Appendix C. The property verified to synthesise the optimal strategy for player 1 is:

```
<<p1>>Pmax=? [ F "p1_wins" ]
```

By labelling the transitions corresponding to actions in the model used for opponent character selection, the adversary file can be searched for the transition used, giving the counter material for the material under inspection. We call the resulting directed graph an **optimality network**. Optimality networks show all material sets with the set that counters them – the set which is best used against their optimal strategy, as well as the value by which they are countered. Edges along optimality networks can be read as *"Beats an optimal strategy for this material, winning with probability p"*. E.g.: Under Z_2, KA beats AH with a probability of 0.61364. Any vertex in an optimality network with no incoming edges denotes a dominant material set.

4.9.2 Analysis

The optimality networks shown in Figure 4.13 can be used to explain the balance under both configurations. The conclusions reached are similar to what one would surmise from CSG. Namely, that RH is a dominant material under Z_1 and that there is a cycle of effective, non-dominant material under Z_2, specifically KA → WR → KH → AH → KA.

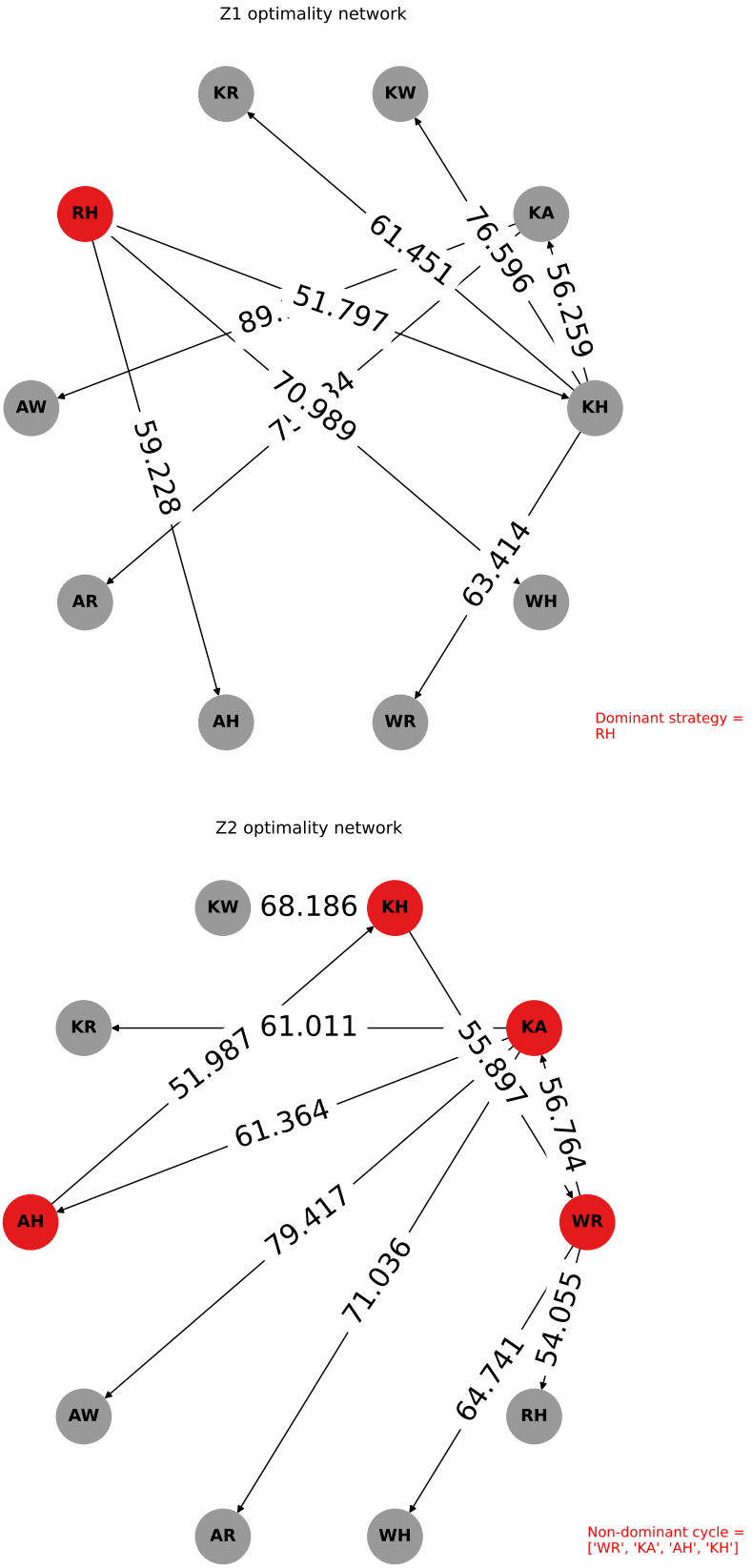

Figure 4.13: Optimality network for Z_1 and Z_2

	CSG		Optimality Network	
Material	Counter	Value	Counter	Value
KA	WR	0.432	WR	0.432
KW	KH	0.363	KH	0.318
KR	KA	0.404	KA	0.390
KH	AH	0.480	AH	0.480
AW	KA	0.206	KA	0.206
AR	KA	0.294	KA	0.290
AH	KA	0.386	KA	0.386
WR	KH	0.441	KH	0.441
WH	WR	0.353	WR	0.353
RH	WR	0.459	WR	0.459

Table 4.9: Comparing the counter materials identified by CSG and optimality networks under configurations Z_2

4.9.3 Comparison of Optimal Strategies With Final CSG Strategies

CSG executions that do not identify dominant strategies do not prove that no dominant strategy exists under a given configuration. This is shown by CSG performed on configuration E for RPGLite 1 Figure 4.8, where a dominant KW strategy is not found. Optimality networks will find a dominant strategy if one exists. Table 4.9 compares the counter materials identified by CSG and by using optimality networks under Z_2. The same opposing material is identified every time, but 3 of the values found by CSG are slightly inaccurate. This is because the strategies have not developed fully, they were optimal against the strategies found in the meta previously, but those strategies were not optimal strategies.

Considering both the accuracy of results and the reduced cost of calculating the optimal strategies, it is clear that optimality networks are superior to CSG for a holistic view of material comparisons. What CSG provides that optimality networks do not is a representation of the game being played. In this work we assume that it is more realistic for players to be able to find the best strategy against a known strategy, than for players to be able to find the best strategy against any possible opposing strategy. But the proximity of the final strategies identified through CSG to optimal strategies suggest that in doing the former, players find globally optimal strategies or very similar ones at least. For this reason we focus more on optimal strategies and their use in development and analysis of gameplay.

4.9.4 Automated Reconfiguration

The technique used to generate optimality networks for RPGLite 2 is simple and efficient. Given a candidate configuration for RPGLite 2, a graph representing the counters to optimal strategies for all material can be generated in minutes using reasonable hardware. From the results one

can suggest which material units are too weak or too strong and update the configuration accordingly. For this reason we experiment with automated reconfiguration of RPGLite 2 as a result of optimality network generation.

We use automated analysis of optimality networks to test for *suitably balanced* configurations. Automated reconfiguration takes an initial configuration for RPGLite 2 and updates it based on predefined conditions, until a satisfactory configuration is identified. This fully automated process can be used by developers to find a starting point for a balanced configuration of their game. In addition to the ability to generate optimality networks, when combined with the analysis of generated graphs it can provide insight into comparative material strength and intelligent ways of altering configurations.

What determines a good optimality network is hard to narrow down to a single factor. For this experiment we define two main goals; to maximise the minimum optimal value in the network and to expand the non-dominant cycle to as much of the superset of materials as possible. These will indicate configurations with high variability of meta materials and minimise the degree to which material is dominated. However other features of a network may signal an improperly balanced game, for example a material set performing too well against another could be undesirable. An imbalanced game is easier to identify, either a single material set is dominant, such as Z_1, or a single material unit is in no set which counters another.

The difference between Z_1 and Z_2 is minor – The Knight and Archer have their accuracy increased by 0.1, but the effect on the projected metagame from CSG is dramatic. The reasoning behind the change was to counteract the strength of the dominant RH pair. Reducing the accuracy of the Rogue and Healer in Z_1 by 0.1, a more natural response to their being a dominant pair, results in a configuration with a dominant KA strategy. But improving the accuracy of the Knight and Archer (the change made to get configuration Z_2) resulted in a healthy cycle. This kind of unpredictability makes automated reconfiguration difficult. The strength of the approach comes primarily from being able to test multiple configurations in a short space of time.

Once material that is imbalanced has been identified, their attributes need to be updated. With RPGLite in particular, accuracy is the fluid attribute whilst the others (health, damage, heal and execute) are more sensitive. Changing a damage value from 3 to 4 for example will have a significant impact on the character's strength, whereas changing accuracy from 0.75 to 0.74 will have less of an effect. When a character is found to be too weak, but their accuracy was already very high, the other characters were weakened instead by having their accuracy lowered. This is possible with RPGLite, with its small character pool, but could lead to too much change in games with larger material pools, nullifying the attempts at fine-tuning. We tried changing the more sensitive attributes when accuracy could not be increased, offsetting the change by reducing accuracy whilst increasing another attribute. However, this lead to configurations far removed from those we described ourselves that the process begun with, defeating the purpose of the exercise.

In total 150 configurations for RPGLite 2 were tested over the course of this experiment, with several interesting results. The largest cycle identified used 7 of the 10 material pairs whilst another configuration had a weakest counter material of 0.59922. Very similar configurations had considerably different results, some even had dominant materials. The configuration with the 7-cycle had a Rogue with 6 health and an execute range of 6, meaning opposing Rogues could eliminate each other in a single action. Whether this would be a satisfying configuration to play is unclear. When initially configuring the Rogue, the ability to eliminate any unit in a single action was purposefully avoided. But the Rogue is strong compared to other materials in this configuration, that assumption may well have been incorrect.

4.10 Conclusions

In this chapter we have introduced CSG and given several examples of its utility using the case study RPGLite. Using strategy synthesis for game playing strategies can overcome a considerable portion of the complexity inherent when game balancing. Initially this work focused on the repeated synthesis of strategies to represent the metagame, but it became apparent that this was less useful and too expensive for larger systems compared to considering optimal strategies from the start. CSG is useful technique and the synthetic metagame it creates could be valuable to developers, especially considering that no other methods for this currently exist. Other automated game balancing efforts consider individual players and their journey through the strategies and materials available, unlike the larger-scale view offered with CSG.

The viability of our approach is possible only because we consider simple systems. Larger systems would require significant abstraction and some games, such as multiplayer dexterity based games, simply would not be suitable for model checking in this way.

Moving away from the chained nature of earlier methods to considering a single adversary per material, as optimality networks do, can provide a good starting point for developers from which to fine-tune their games. This point can be reached quickly and allows for multiple configurations to be considered with some interesting details learnt from the process, even if the final configuration is found immediately. With RPGLite 2 we determined changes in the configuration to balance orthogonal design, such as the wizard's stun. We also found that several of our initial assumptions about the materials were incorrect. For example, we had assumed having very high accuracy values (above 95%) would make a character too strong, but this consistency was actually another characteristic of the material we had not originally considered, a character which hit almost every time is itself a unique characteristic which some players may prefer.

The specificity of the automatically generated strategies are not perfect representations of the kinds of strategies that a player might use as they are unlikely to have the capacity to perform such complex calculations. For this reason, the approach must be tailored to relate back to how games are actually played. Having shown that CSG converges on optimal (or almost-optimal)

strategies, this prompted us to ask the question: Do human players eventually employ optimal strategies? In chapter 5 we explore this question, developing RPGLite, the mobile game, and analysing how it is played.

Chapter 5

RPGLite, the Application

"Where the full development and publication of the mobile game RPGLite is described along with a retrospective look at the process."

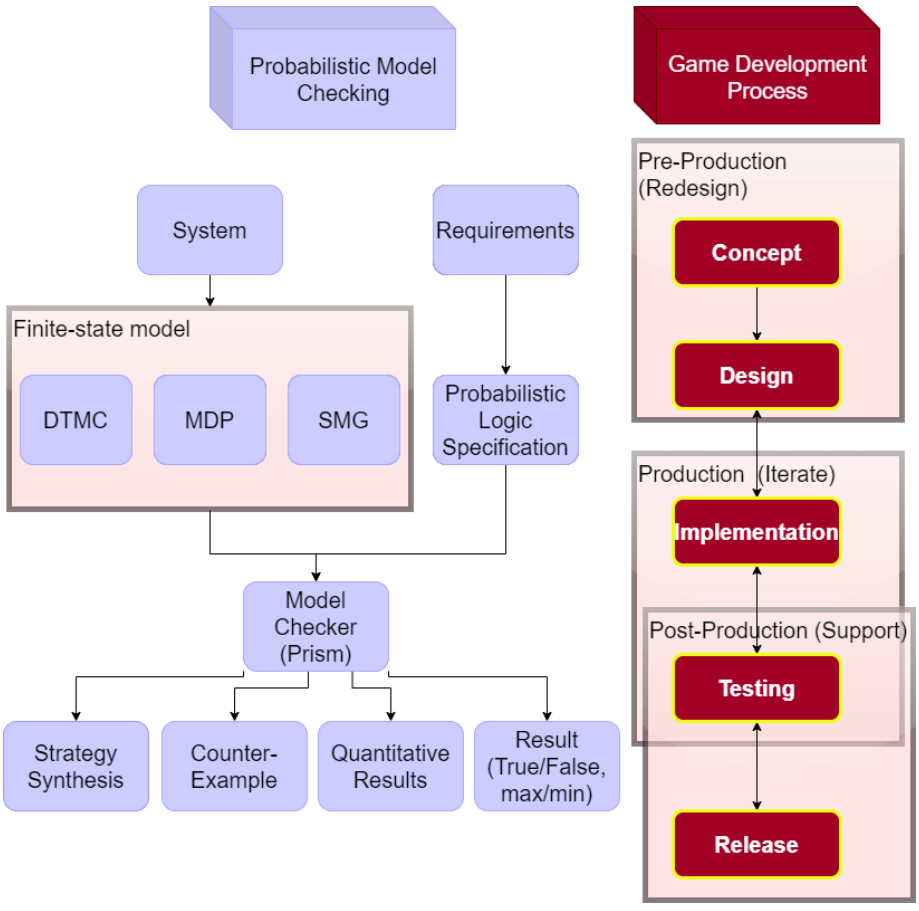

Figure 5.1: Chapter 5 areas.

5.1 Introduction

The question of whether or not a game is balanced is subjective. Whilst in this thesis we attempt to make objective measures of balance, the subjectivity of the matter cannot be escaped. As such for this work to be valid we required a large scale experiment to compare games which we claim to be balanced following our automated testing, with how human players actually play. The aim was to observe if the methods of generating synthetic data (using CSG and optimality networks) were accurate and if a game designed with model checking central to its production could be successful.

In this chapter we introduce RPGLite the game, an application based on an extension of the case study used in the previous chapter. We describe the process of creating the mobile game RPGLite, including development, release and ongoing support. As RPGLite is the major experiment we undertake, it is described in detail. This is different to typical experimental design carried out during research. In developing a game to be released through the same methods as commercial titles it needed to be both engaging and robust enough to allow us to gather sufficient data for experimentation. This chapter begins with the design of the application itself, including the objectives of the experiment, the technologies and principles used and the decisions made. We then describe the final application and its full feature set and the player data, describing how the game was promoted and the subsequent rate of user acquisition and data generation. Finally the lessons learned from development of a mobile game are presented by way of an experience report. In this chapter we do not go into detail about the game itself, focusing on the application surrounding it. The game design is the focus of a later chapter, as is the gameplay analysis.

5.2 Experimental and Application Design

5.2.1 Objectives

RPGLite is a multiplayer mobile game released in April 2020 on both the Google Play and iOS stores. More information on the game is available at rpglite.app. RPGLite is an online multiplayer game based on RPGLite 1 and 2 detailed in Chapter 4. It was developed in collaboration with a fellow PhD student Tom Wallis. Tom used the app to inform his own research on generating synthetic human-realistic data. Tom was responsible for the networking infrastructure of the application, I was responsible for the application design. That work is not included in this thesis. Instead we focus on our own uses for the app, the continuation of our research into the use of model checking for game development. Our objective was to gather a large dataset of player data, which could then be compared to model checking based predictions of player behaviour to measure accuracy.

The experiment was set up to be extensible and data was captured beyond the requirements of the designers in case it is useful for future work. The RPGLite database captured details on 9,693

games played, 370 player accounts, 170 games in progress (when the database was captured) and 1,105,065 user logs detailing interactions with the application. The data used in this thesis is a snapshot from November 2020, which is available online [72].

The game at the core of the application was created in tandem with mathematical models to ensure model checking could be performed efficiently. We extended the character set of RPGLite 2 with a further 3 characters, yielding a total of 8.

5.2.2 Software Architecture

Several technologies were considered for the development of the game. The requirements were for a lightweight, but robust multiplayer system and a database for record storage. Due to the inexperience of the developers with game design, the Unity Engine [73] was used. Unity is a game development platform which supports multiple scripting languages in addition to a drag-and-drop interface. It has a comprehensive set of instructional tutorials and, whilst it is widely used by game development professionals, it has an active amateur development community. The scripting for RPGLite is written in C#. RPGLite makes requests to a public-facing REST api, written in Python3 and runs on servers hosted by the University of Glasgow, with a firewall under the control of the institution's IT services. This server initially handed data processed in the client to the database to avoid a direct connection (and the risk of exposing the database publicly), but became a larger aspect of the engineering as design moved towards a thinner client (one which handled less of the data transfer). The project data was stored within a MongoDB database also hosted locally within the University. The database and the client never interact directly, all requests go through the Python middleware service.

5.2.3 Design Principles

The experiment went through a full ethics approval process. All players are informed of the purpose of the application (to collect research data) and can download a copy of the terms of usage directly from the application. To comply with GDPR, no personally identifying information is stored, this includes email addresses. The only form of communication we have directly with the players is through push notifications sent to their devices, which players can disable from their device's settings or through the game's settings. Accounts are protected through a username and a password, with a private key stored on a device upon registration to be used for account recovery. The central philosophy of the design of RPGLite is that it be simple for players to play the game, without compromising on their motivations. For the data to be useful for comparison to the modelling of competitively-minded players, we need all players to play to win. To achieve this we made creating games and finding opponents quick and easy, moving other screens away from the main user experience. We have also made several design decisions focused on ease-of-use and lowering the associated cost of gathering data, notably by using play-by-correspondence.

5.2.4 Play-By-Correspondence

As RPGLite is a turn-based game, the application does not need players to be online simultaneously for games to be played. Having a large number of players was not expected. Play-by-correspondence is used to alleviate the burden of multiple concurrent active users. Once two players have been matched up only the player whose turn it is can choose an action. The other can view the state of the game, but has to wait for their opponent to act. Play-by-correspondence is used for Chess platforms such as chess.com and other popular mobile games such as Words With Friends [74]. With this form of game, players can be involved in multiple games at the same time. Each player has **game slots** which can be filled with active games. Following testing, we decided that 5 was a suitable number of game slots as more would have cluttered the UI and could have led to games being viewed as less important by the player, compromising their motivation. To ensure players can always play games, if an opponent does not act for over 24 hours then a player can **claim victory**, at which point the game ends and the corresponding game slot is cleared.

5.2.5 Visual Design

The visual UI design of RPGLite is simple, with large rounded buttons and big text. The character artwork was commissioned by a professional artist, Justin Nichol, and is used liberally as it is the strongest graphical asset in the game. The *cards* used to represent characters in the game are based on the character artwork. All assets in the game other than the 8 character images were created during development. On reflection, using the numerous free assets supplied by the community would have been a more effective use of time and resources. Commissioning an experienced UX developer would have significantly improved the visual appeal of the game and likely increased both player acquisition and retention, the poor aesthetic design is a drawback of this experiment. The earliest version of the game was a Java applet which could be played against an AI. This was used as the template for the application's game design. The design of the application was modified so often that the finished product is far removed from initial implementations.

The character cards are important as they allow for the information to be displayed whilst also providing a focus for player interaction. RPGLite players need to know the current health, mechanics, statuses and attributes of all alive characters as well as seeing whose turn it is, and what the outcome of the actions are, and still be able to interface with the game to take actions. Fitting all of that onto a mobile screen is difficult. It was achieved using character cards and overlay screens for the action resolution.

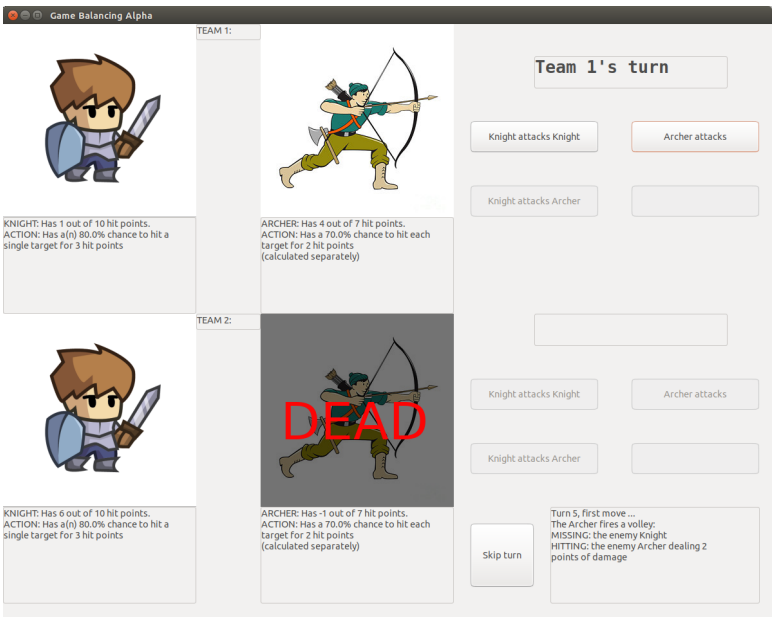

Figure 5.2: Early design for RPGLite developed as a Java Applet

5.2.6 Updates

The application has bug reporting features and links to allow players to contact the developers directly with feedback. Small updates were released when bugs were identified, these were all visual bugs that did not affect the data collected. It was anticipated that it would take several months to gather enough data for analysis of a single configuration of the game which we initially expected to be around 2,000 completed games. However, after just over one month more than 2,000 games had been played so a "Season Two" was planned, including an updated configuration of the game. All progression and *skill* (see Section 5.3.2) carried over into the new season. The Season Two update also included some minor quality of life changes which were unlikely to affect gameplay motivation. This included replacing the list of most recently active users viewable from the home page with the list of the most recently registered players.

5.2.7 Testing and Feedback

Throughout development a small group of testers were consistently used from the first playable prototype until release. A shared document was maintained with feedback encouraged under the headings: 'Game breaking bugs', 'Small issues encountered', 'UI issues', 'Unclear things' and 'Other'. By maintaining a consistent group of testers and maintaining this dialogue, we received high-quality, nuanced feedback that lead to numerous impactful changes on the final application. The group was composed of members with disparate backgrounds, with varying skills and experience with mobile games and game development. We also ensured that the application was tested on a variety of phones so that we could test on different operating systems and resolutions. Some features that were implemented as a result of this group feedback include: the use of a roll

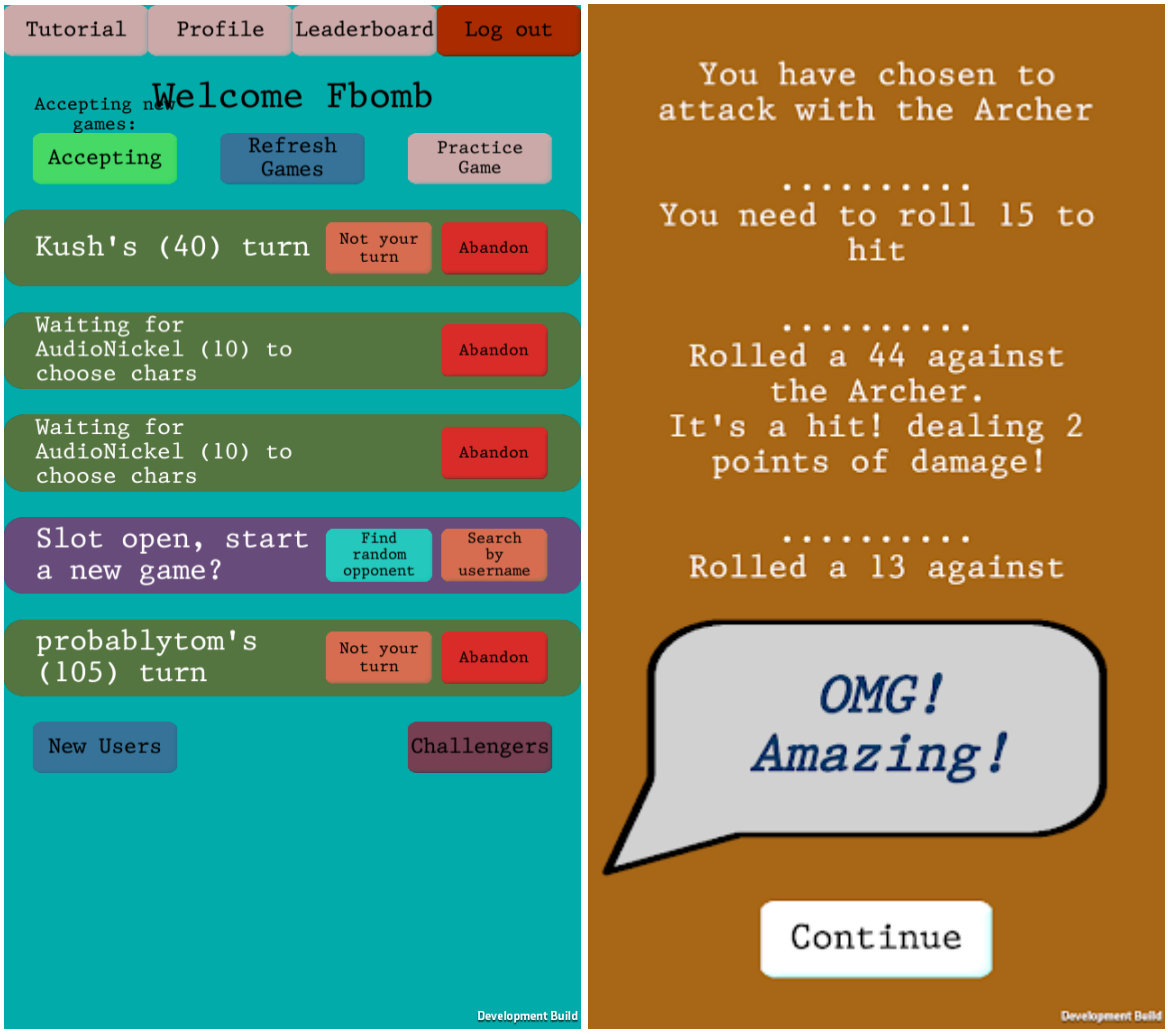

Figure 5.3: UI bugs raised in closed testing. Overlapping text and a player's *skill* not showing (decile and the player not being shown who they have attacked (right).

animation replacing slow scrolling text describing an action's outcome and for the animation to be skippable (the vast majority of rolls were skipped), the ability to claim victory against dormant opponents and the introduction of medals. In addition, numerous UI bugs were found because of this testing, some examples are shown in Figure 5.3.

5.3 Application Specifications

In this section we give an overview of the application and the ways in which users can interact with it.

5.3.1 Walkthrough

In order for a player to play a game they open the application, log in and select a game slot for which it is their turn. The screens visited during this process are shown in Figure 5.4. The home

screen in this case shows the page for a player with 4 active games in the first 4 slots, in the second of these it is the player's turn to act. Once in the game the player first sees a recap of their opponent's last move through a roll animation. To act the player selects the character they want to use and then who they want to target by clicking on the corresponding card. The available cards *wiggle* and there is explanatory text at the bottom of the screen to guide the user. Once an action has been selected players are shown a skippable roll animation which reveals the random number generated and is accompanied by a *quip*. This is an amusing comment randomly selected from a stored set of size greater than 80. Quips respond to: a successful action, a failed action, an opponent's successful action and an opponent's failed action. The roll animation dramatises the die roll with a bar emptying until the roll value is reached, slowing as the value approaches the accuracy of the acting character to add tension. However, of the 209,423 actions viewed in online games (either as player actions or opponent recaps) 193,924 were skipped (92.6%), indicating that players preferred action to tension. The game screen and roll animation are shown in Figure 5.5.

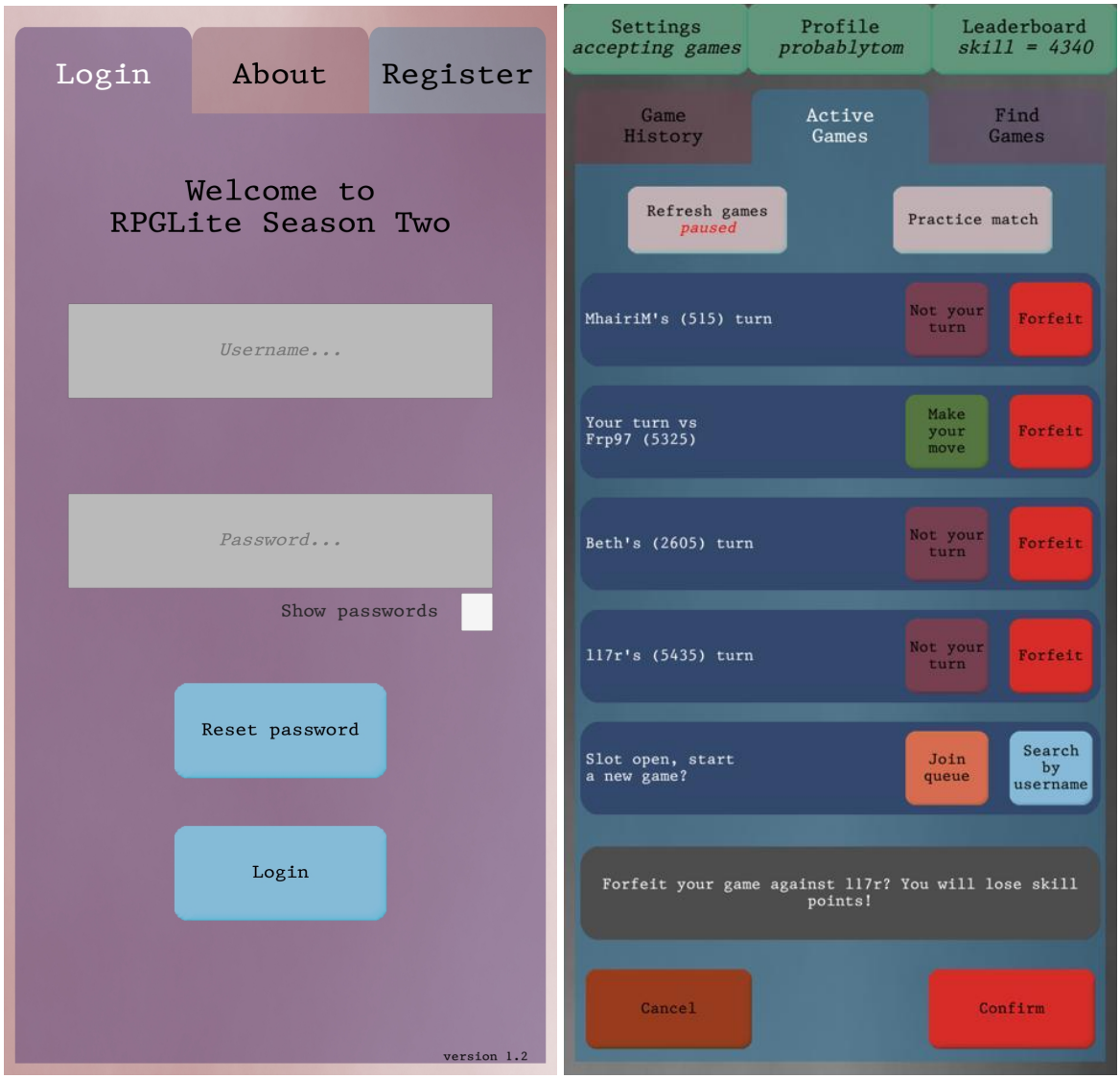

Figure 5.4: RPGLite: login and home screens

Figure 5.5: RPGLite: game screen and roll animation

Starting a Game. There are several ways to start games in RPGLite, however a player can only have 5 games in progress at a time. By default players are *accepting games* which means others can start a new game against them if they have free slots, this can be disabled in the settings. The ways to attempt to start a game against another player are listed below.

- A game can start by a player joining the *waiting list*, a buffer of size 1 which stores the username of a waiting player. The slot which is used to join the waiting list is not available for new games and remains blocked until the player leaves the waiting list or a game is found. When a player attempts to join the list and another is waiting, a game is created between the two players and the waiting list is cleared.

- A game can start by a player searching for an opponent by entering the username of another player. This is only successful if they are accepting games and both players have a free game slot. If a game is created this way then a game is created in both players' first available slot (from top-down).

- A game can start by a player selecting one of up to 5 opponents who have issued a challenge from the *Find Games* tab. This will return a player to the *Active Games* tab with the challenger's name filled in the search bar. On the same tab players can issue a challenge themselves.

- A game can start by a player clicking on an active user in the *Find Games* tab, where the 5 most recently *active players* are displayed. For Season Two this was replaced with a *new users* buffer giving the five most recent registrations. Both of these will populate the search bar in the active games tab.

- A game can start by a player searching for a user from the leaderboard by interacting with another player's position on the board, this also populates the search bar.

- A game can start by a player interacting with a recent opponent in the *Game History* tab, which shows the results of a players last 5 games, populating the search bar.

Of 9,308 games tracked, 6,023 were created through the waiting list (64.7%). It is not possible to determine precisely how many games were initiated as a result of the other methods, but we do know how many times they were attempted (the recipient may not have had free slots or accepted the game, so the attempts could have failed): 422 times from game history and 508 from the leaderboard. The number of times players searched from a combination of the challenge list, new users list and active users list, was 5,595.

Once a match has been made both players must select characters. This can happen in any order and players select characters without knowledge of what their opponents have chosen. Once both are selected one of the two players is randomly selected to go first. As game slots can be changed by others starting games against a user, they are automatically refreshed every 60 seconds, or can be manually refreshed using the refresh games button on the active games tab.

Ending a Game. There are four possible end states for a game listed below.

- A game can end normally, where a player has won by reducing both of their opponent's characters to 0 health. Following the action which won them the game, the player views a congratulatory screen which notifies them of their *skill point* change. The loser will see a commiseratory screen after viewing the recap of the move which made them lose and a similar notification of their loss of *skill*. Both players have the relevant game slot cleared once they have viewed the respective game over screens.

- A game can be **forfeited** where a player clicks forfeit on the game slot from their home screen. Players can forfeit games if it is their turn or if their opponent has not acted for over 24 hours. The player must confirm that they wish to forfeit (shown in Figure 5.4 (left)), once confirmed the game slot is cleared and an alert describes the effect on their *skill*. Their opponent views a recap which states "You win, opponent forfeited", the game concludes as if it were a normal ending from then on.

- A game can be **abandoned** by a player when a match is yet to begin because one or both players have not selected characters yet. An abandoned game has no affect on *skill* and the game slots of both players are cleared immediately.

- A player can **claim victory** on an active game in which it is their opponent's turn and the opponent has not acted for over 24 hours. If the conditions for claiming victory are met then the forfeit button on an active game slot is replaced by a *claim victory* button. In this instance the player is shown an alert with their updated *skill* and the game slot is cleared. The opponent also has their game slot immediately cleared and their *skill* is updated without an alert. This is the only instance where *skill* is updated without the player being notified.

Not all games ended with a player winning the game normally – 2,159 games ended through means other than gameplay. Of those, 1,233 games were abandoned, 770 games ended with a player claiming victory over an absent opponent and 156 games were forfeited. 9,308 games started normally (without an abandonment) meaning that victory was claimed due to an absent opponent in 8.27% of games and 1.68% of games were forfeited.

The Game screen. The game screen is designed to mimic a game of RPGLite being played using cards in the real world. A player's cards appear at the bottom of the screen and their opponent's at the top. Tags (Customisable blocks showing usernames and *skill*) denote player identity and are displayed alongside their playing space. Health is displayed on the cards using red pips at the bottom. The cards for stunned and dead characters have a clear visual effect to denote their status. From the game screen players can toggle to an overlay screen showing all moves made so far using the *Show moves* button and can refresh the current game state with the *Refresh game* button. This is useful if the player knows that their opponent is actively making moves and does not want to return to the home screen in order to refresh the game state.

5.3.2 Incentivisation Systems

'Skill' Points. RPGLite has a ranking system called *skill* which updates for both players once a game is over. One scheme we could have adopted is the Elo ranking system from Chess, which is intended to give an accurate accounting of a player's ability. Typically players have an initial Elo

score of 1,200 which increases or decreases following each game in such a way as to maintain a normal distribution of points among players. Because we believe that the Elo ranking scheme can be disheartening for poorly performing players (as points are lost as well as gained and scores change less over time meaning some players get "stuck" with a low score), we developed our own cumulative scheme. Scores that do not force progress (such as Elo) can exacerbate those with a fear of failure [75]. Players start at 0 *skill* and are awarded 10 every time they log in for the first time in a day, not including the day on which the account was created. After a match between 2 players with *skill* S_1 and S_2 in which the first player has won, the change in *skill* points for each player is calculated as follows:

$$\begin{aligned} new(S_1) &= S_1 + 40 + \delta \\ new(S_2) &= S_2 - 10 + \delta \end{aligned} \quad \text{where} \quad \delta = \begin{cases} 5 & \text{if } S_1 - S_2 > 500 \\ 4 & \text{if } 500 \geq S_1 - S_2 > 400 \\ 3 & \text{if } 400 \geq S_1 - S_2 > 300 \\ 2 & \text{if } 300 \geq S_1 - S_2 > 200 \\ 1 & \text{if } 200 \geq S_1 - S_2 > 100 \\ 0 & \text{if } |S_1 - S_2| \leq 100 \\ -1 & \text{if } 200 \geq S_2 - S_1 > 100 \\ -2 & \text{if } 300 \geq S_2 - S_1 > 200 \\ -3 & \text{if } 400 \geq S_2 - S_1 > 300 \\ -4 & \text{if } 500 \geq S_2 - S_1 > 400 \\ -5 & \text{if } S_2 - S_1 > 500 \end{cases}$$

The aim of the *skill* system was to encourage more games to be played without frustrating players who lost so often that they stopped playing. The *skill* system rewards the number of games played more than the proportion of games won and is designed to enable new players to advance fairly quickly up the rankings. If a game is lost then a player's *skill* will not fall below a multiple of 100. If a player loses points through forfeiting or having victory claimed against them due to a lack of response, then they can drop below a multiple of 100, but not below 0.

Leaderboard. The leaderboard page Figure 5.6 is accessed from the home screen and is split into two halves. The top half shows the user's relative *skill* position amongst all players as a bar chart, with bars for every 10% range, where 100% is the *skill* of the current leader, showing the number of players in each. The bar for the player's range is highlighted. We also show their percentile position amongst all players irrespective of skill, which is why the percentile and decile values are different in the figure. The lower half shows a scrollable leaderboard of all players indicating their rank and *skill*, and displays the position of the user. A toggle to display this information for the either season was introduced with the Season Two update. The

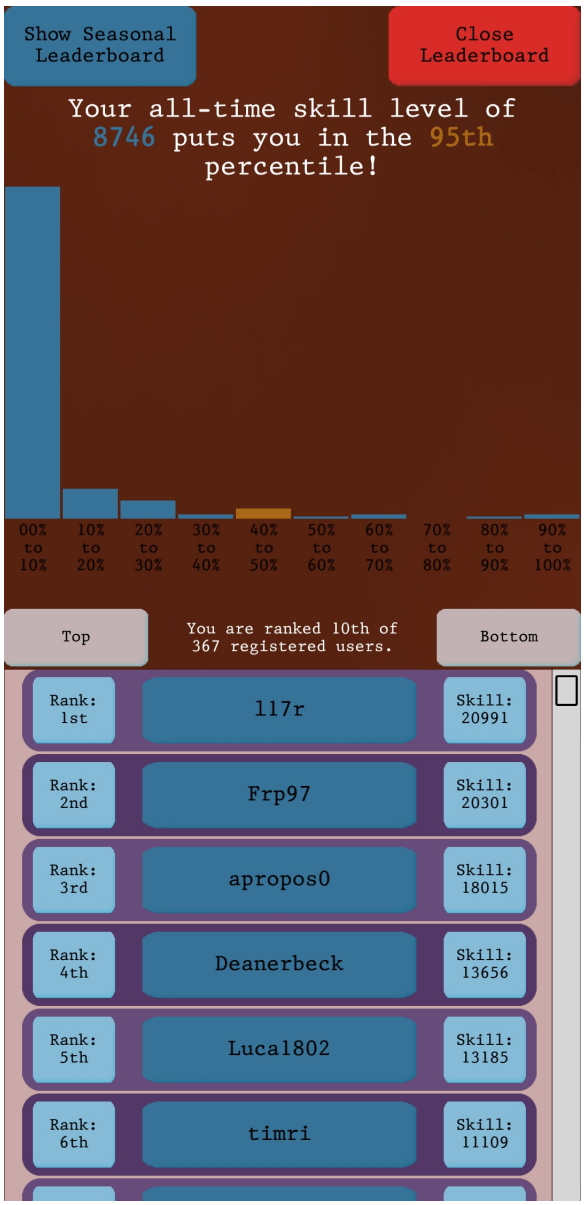

Figure 5.6: RPGLite: Leaderboard

leaderboard was visited 35.4 times by players on average, just over once for every four games played. As shown in Figure 5.7, some players viewed the leaderboard screen a lot.

Unlockable Features and Progression. There are 8 playable characters in RPGLite, players begin with only 4 (the most mechanically simple) and unlock the rest one at a time when they have played one full game with all previous characters. It is possible for a player to unlock all characters after 5 games. While this will affect the way in which material was selected, progression has been shown to be a key incentivisation technique and as such we believed in the long term it would lead to more data being collected.

In a game, players are labelled with a *tag* to emphasise their playing area, giving their username, their *skill* and a background they can select from 5 available through the settings menu.

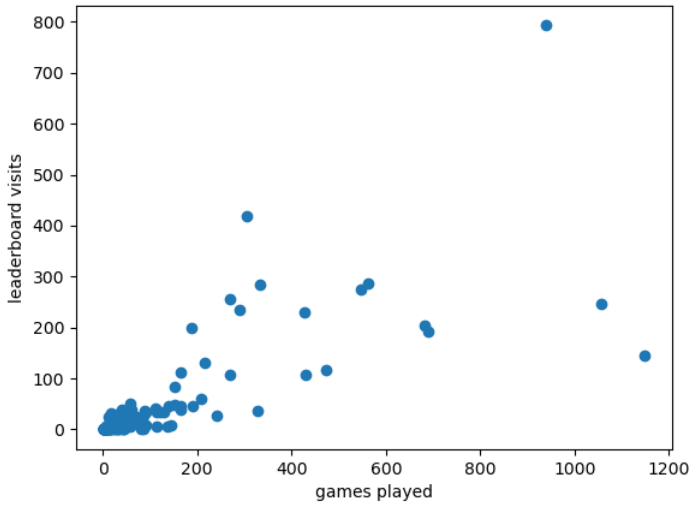

Figure 5.7: Number of games played and number of times the leaderboard was visited

The tag has a border and a coloured star which updates as more games are played. Players with over 20 games played have a bronze star, over 40 have a silver star and over 60 have a gold star. The border colour changes from no border, to bronze, to silver and finally to gold every 5 games between stars. There are no advancements in tag display after 75 games when players have a gold star with a gold border.

Character backstory. Each of the 8 characters in RPGLite were displayed in the character select screen alongside a brief background description. For example, *"The Wizard is wise beyond his years (and that's a lot!) although some think he's a little mad, they can't argue with his knowledge and skill. He also has 10 cats."* This was done to try and increase the level of emotional investment players had in the game.

Medals and Profile Display. The profile page (see Figure 5.8) can be visited by any player who has completed 3 games via a button on the home screen. The profile page gives information about the player's game history. It shows how many times each character has been picked by the player and how many times they have won for the player. It also contains a short description of which character is used most and which has the highest percentage of games won. The rest of the profile pages displays the 111 medals which can be obtained in RPGLite, listed in Appendix D. There are 37 different categories for the medals, each with a bronze, silver and gold award. A category can be selected for an expanded view showing the player's progress towards the next medal. Medals are designed to encourage repeated games and competitive play. For example "Play 5/10/25 games with the Archer" or "Win 3/5/10 games in a row". The profile page was visited 3,027 times, but medals were only examined 231 times. It is possible that users were not aware that medals could be selected for more information as this feature was not highlighted

beyond the tutorial.

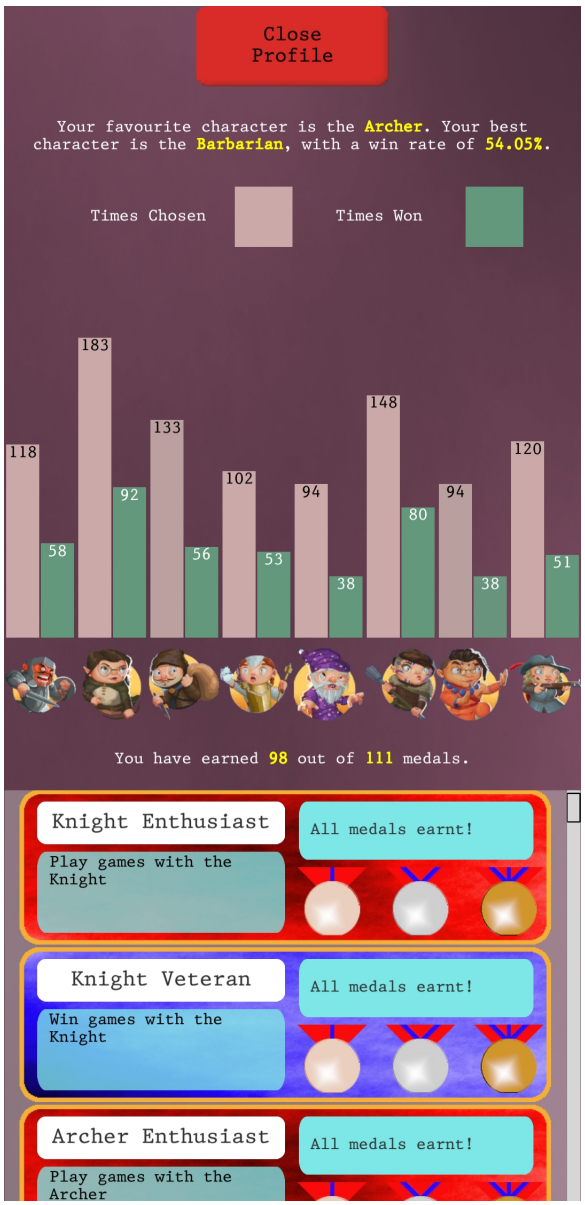

Figure 5.8: RPGLite: Profile

5.3.3 Peripheral Systems

This subsection gives an overview of the additional functionality of the application that was more than just the game itself.

Game History. From the home screen *game history* tab, players can view their last five played games. The view shows which characters were chosen, when the game took place, and who won. There is also a button for each match to search for the opponent again.

Figure 5.9: RPGLite: Game history

Practice Match. From the home screen players can play a practice game against a naive AI opponent who chooses one of the 4 base characters and plays randomly, skipping only when forced. This game cannot be paused and results have no bearing on *skill*. 1,265 practice games were played in total with one player playing 381, far more than the second most practice games played by a player at 69.

Push Notifications. Every time a new game is created or it becomes the user's turn to move, their device is sent a push notification with one of over 20 messages for each instance. These notifications were implemented using the OneSignal API. Push notifications are suppressed when the application is active, they are used instead to trigger refreshes to the currently viewed screen if the user is on a game when their opponent moves or the home screen when a game slot updates.

Settings. From the settings panel users can toggle all push notifications, they can toggle whether they are *accepting games* (whether others can fill their empty game slots) and can visit the *customise tag* screen where they can select a background for their tags. They can also logout and change their password from the settings panel.

Tutorial. A short tutorial is viewable from the home screen and players who have not completed an online game are prompted to view it when they log in (this prompt can be dismissed once or dismissed forever). The tutorial consists of a series of images describing the functionality of the application and the gameplay of RPGLite with back and forward buttons. A total of 361 of the 370 registered users viewed the tutorial with 9 users viewing it twice.

Message of the Day. On logging in all players are greeted with a message of the day. Several variations are available and, if possible, one that the player has not seen before is delivered at random. The various messages include hints and tips, reminders about features that are not being used and a notification to players of the impending season 2. A default message is shown when the user has seen all other messages.

5.3.4 Database Design

The RPGLite database has collections for: completed games, games, page hits and players. There is also a series of backups for the player documents taken at consistent intervals, this makes it possible to track rates of progression. Other collections in the database include a collection of ongoing games from between the seasonal update which were lost as well as *special data* which stores: the unique password reset PINs, information for the waiting list and other buffers for active users and challengers, their *skill*, and when they were last online or when they issued a challenge. The experiment was designed to be open-ended so more information is captured than we required to answer future research questions. Due to this, a large amount of data is captured about every user and every game played. MongoDB documents are a series of key:value pairs and every document has a unique ID as their first entry. We describe each document type in turn:

Games. The unique IDs of each player and their usernames are stored along with the time when a game was created. Once characters are selected a turn attribute is initialised as are each character (*p1c1*, `p1c2`, `p2c1`, `p2c2`), their current health values (e.g., `p1c1_health`) and whether they are stunned (e.g., `p1c1_stun`). This information describes the current state of the game as used with the model checking techniques given in this thesis for RPGLite games, described in Chapter 4. When a game finishes, the end time, *skill* changes for both players and the index of the winning player are stored. Games which began in Season Two have the additional attribute `{balance_code:1.2}`.

Players. Player documents consist of the following information: Their usernames and a hashed form of their password, the state of their game slots, their current *skill*, the number of games they have played and won with each character, the time they last logged in, whether they are accepting games, a list of players they have lost against and a series of unmarked counters used for medal progression. The information on slots is stored as either: the string 'none' if the slot is empty, the string 'waiting' if the player is in the waiting list using that slot or the ID of an active game.

Page Hits. All database accesses are logged in a page hit document detailing the type of activity, the user performing it and the time it was performed. Certain activities have additional information detailing the result of the activity, for example if a player tries to view the leaderboard but is yet to play the required number of games, this is logged as a failed attempt. Page hits can be used to trace users through the application as they navigate between screens. Table 5.1 lists all logged events and the number of logs for each.

The published dataset does not include information that was not likely to be useful for current or future research. Hashed passwords are removed from player documents for example.

5.4 Experience Report

The design and publication of a mobile game is an uncommon activity during research of this kind. The application was developed as a collaboration between two PhD students, neither of whom had any game or mobile app development experience. As such we reflected on our processes to guide future researchers in this area who are planning similar projects. The reflections are organised into four separate *lessons* which we present here.

5.4.1 Resist Temptation

At many points in the development process, we found it difficult to constrain the feature set of the end product. The unbounded nature of the project led to additional features being implemented as development became a "labour of love". These delayed the delivery of the game, and few new ideas were actually discarded. Only some of these features were beneficial to the player experience. An illustrative example is the comparison of two such "peripheral systems": the leaderboard and players' profiles. Players load the leaderboard two-and-a-half times as often as their profiles and, anecdotally, they are a central component of player retention. An equal amount of effort was spent on each. During development it was impossible to know how often a feature would be used in practice.

The ideas that came to us during development were sometimes essential to the project's success, and to resist all of these would have resulted in a poorer product. The danger we identified in our own endeavours was a desire to implement these ideas *for their own sake*, and not for their

Event	Observed	Event	Observed
logout	227	registration	370
click *Find Games*	16611	send challenge	1421
login	83474	join queue	13146
Home manual refresh	25006	search from *Find Games*	5595
search for opponent	5663	visit *Char. Select*	19431
character selected	37687	*Char. Select* → *Home*	19434
Home → *Gameplay*	103783	rolls fast-forwarded	193924
move viewed	209423	*Gameplay* → *Home*	86790
Home → *Tag Customisation*	200	update tag	117
Tag Customisation → *Home*	199	*Home* → *Profile*	3027
click *Game History*	4667	*Home* → *Leaderboard*	7567
notifications toggled	76	*Game* manual refresh	1964
Home → *Tutorial*	370	accepting games toggled	319
view website from settings	32	*Home* → *P: Char. Select*	1161
P: character selected	2596	*P: Char. Select* → *P: Game*	1265
P: move viewed	17714	P: rolls fast-forwarded	15036
P: *Gameover*	1128	P: *Gameover* → *Home*	517
logged in	24303	*Message of the Day*	23517
Reward Overlay	18981	view website from *Registration*	66
tutorial prompt dismissed	245	left queue	1100
P: Char. Select → *Home*	484	game abandoned	2159
dismiss tutorial forever	58	P: move history checked	39
Gameover	15350	*Gameover* → *Home*	15661
P: Game → *Home*	103	opponent searched from history	422
Profile → *Home*	2534	*Leaderboard* → *Home*	6069
leaderboard user zoom	1396	search from leaderboard	508
practice gameover play again	605	medal pressed	231
character deselected	185	P: character deselected	20
purging slot	143	change password	1
move history checked	904	move made	112369

Table 5.1: Forms of user logs stored in RPGLite database. Italics denote screen or tabs, P: denotes practice games and → denotes movement.

benefit to our end product. New ideas must be abandoned where their benefit does not outweigh the additional time they would demand. An agile development approach is the best in these scenarios, where requirements naturally change over time.

Much like implementing new features, we found that the refinement of existing features risked an emotional investment. Existing design components, such as colour schemes and layouts of minor UI elements, were constantly changed prior to release. We found the adage, "don't let the perfect be the enemy of the good", useful in such moments.

We struggled to resist temptation because of our inexperience with app development, our lack of a thorough plan and the fact that we were co-developing and therefore reticent to shoot down each other's ideas. For other developers in similar situations to our own, we recommend a more structured approach. Firstly, a project should have a plan produced at its inception, which is maintained throughout the development process. Second, we suggest adding to this plan a "margin"; a block of unallocated time at the end of the project that can be spent on developing new ideas. As development progresses, this margin can be "spent" on new ideas or refinements to existing design elements. This facilitates necessary discussions by framing them within the context of a shared resource.

5.4.2 Employ Available Research Networks

Advertising is a major cost of app development; new users are expensive. With no money for player recruitment we were forced to promote the application in a similar way to other research experiments within a university context, through participant calls in mailing lists and departmental announcements. Beyond this we sought out opportunities for free publicity from within our research community. We found that there is an appetite for open data and by encouraging people to play our game "for science" our promotions were better received. We anticipated undergraduate students would make up the majority of our users. However, while promotions targeted at undergraduates introduced a large number of users, those users tended to only complete a few games before stopping. For our research we wanted to investigate how players learn over time, we needed high player retention to allow users time to "learn" the system. We observed that retention was highest within players who had a vested interest in us or the research itself, or when the game was adopted by users from a social clique.

In comparing events that we expected would increase player numbers with their effects on new users and games played (a suitable measure of data generation) Figure 5.10, the retention of the different groups recruited is pronounced. Over half of our users failed to successfully complete a single game, and several users installed the app without registering an account. We are fortunate enough to know the chair of the Scottish branch of the International Game Developers Association (IGDA) who kindly shared an advert for the game. The increase in the speed of game completions accompanying the influx of new users from his involvement shows that those players were valuable data generators. Figure 5.10 also shows that the large intake of undergraduate

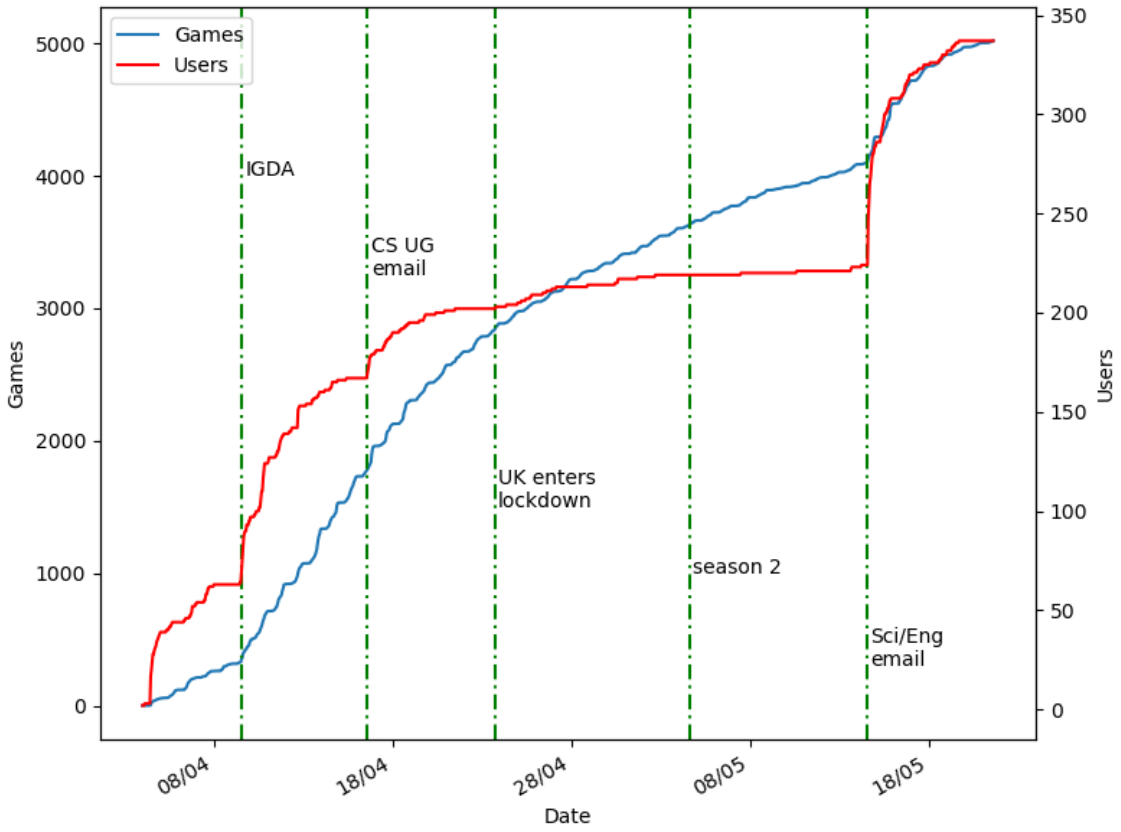

Figure 5.10: The rate of user acquisition in the weeks following RPGLite's release. Important events are also marked: promotion of the application through the Scottish International Game Developers Association branch, an email to Computing Science undergraduates, the date from which UK citizens were told to stay inside if at all possible, the time of a major update to the game and an email to all Science and Engineering undergraduates at the University of Glasgow.

students from Science and Engineering only caused a brief uptake in activity, which quickly dissipated. We believe this is due to either the lack of a relationship with us as the developers or of interest in games research. We also assumed that a large update might increase activity, but found that not to be the case. A single large update changing the configuration of the game, adding seasonal leaderboards and improving existing features had no noticeable effect on the number of games completed. The extent to which our data comes from a small subset of users is shown in Figure 5.11.

Throughout development we sought advice from those around us with some relevant experience, though none with explicit game development experience. Many of our university colleagues had been involved in various aspects of application development and deployment, and advised us throughout. For example, a web designer gave advice on UX design and a gamification researcher suggested various incentivisation systems. We also relied heavily on our department's IT services team for support in deploying the middleware server and administrative staff for promoting the app once it had been released. Application development is multifaceted and

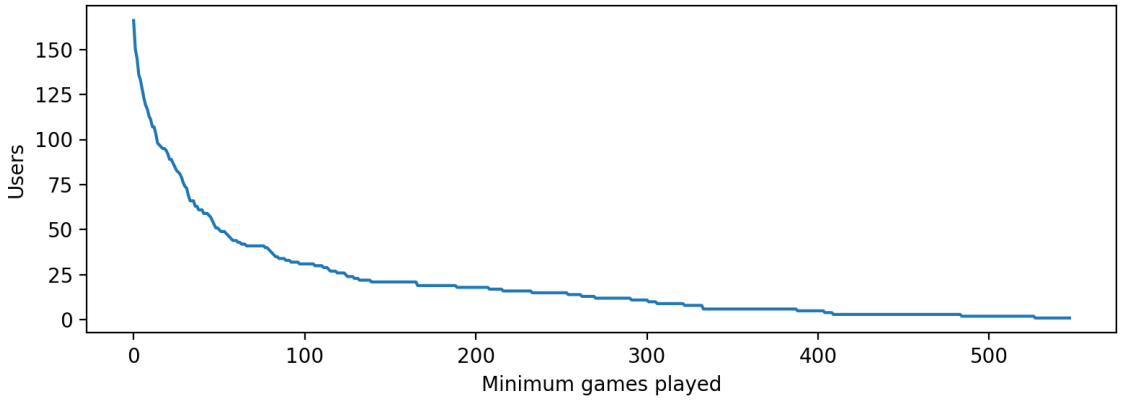

Figure 5.11: The number of users to have played at least a given number of games.

the support of our peers was important in areas where our skills were insufficient.

Without the extensive use of the research communities we belong to, RPGLite would have been an inferior application, producing a less rich dataset. There are numerous skills required to develop a system that people will use willingly. Engaging peers early in the process and being clear in your aims will highlight the areas in which you need support. Where user retention is important your research community is vital, as they already have a connection to you which will see them invested in the project from the outset. Your individual network is unlikely to be enough to generate a significant dataset, so we recommend engaging colleagues to advertise on your behalf. RPGLite never sought to compete with professionally developed games, but through our various communities we manged to generate enough interest for a steady playerbase.

5.4.3 The Smaller the Client, the Better

The one aspect of RPGLite's implementation that we most regret is the amount of game logic being delivered to players in the mobile client rather than the server. There are many reasons for this, the main one being that the server could be replaced immediately if a bug were to be found. This is in contrast to compiling, re-installing and re-testing attempts to fix the given bug, were it to reside in the client. Fixing server-side bugs allowed more rapid iteration when fixing those with origins we did not understand.

The need for moving logic out of our client became apparent after we had pushed production code to app stores and had real players taking part in our experiment. A particularly dedicated player discovered a bug where, after playing enough games, characters that had been unlocked through repeated play would become locked again and could no longer be accessed. Had this bug been in the server, the issue could have been fixed, and a new version deployed in seconds that lightweight clients could connect to. With our larger client, this required testing in Unity, testing on-device (to ensure that there were not platform-specific bugs), and deployment to app stores for approval and distribution. This process took days, even though the bug was trivial to

fix.

Large clients also risk introducing a duplication of code when paired with a secure server. To validate game logic computed by a client, servers must replicate much of the processing the client previously performed, to verify that a malicious user has not supplied corrupted game states. This process requires the implementation of game logic within the server. As a result, a secure server must include the game logic regardless of whether the client does. This means spending time, an already scarce resource, on duplicated code. This is another reason we recommend developing a lightweight client, leaving the majority of computation to a larger server.

When we realised that we had produced a large client, we made efforts to move to a more server-centric design. For example, we considered sending push notifications via APIs directly written into our client. However, the flexibility and control of implementing this server-side caused us to move our notification code to the server. After this, we implemented much of our peripheral systems logic in the server, including the leaderboard, medal logic, password reset, and much of the matchmaking systems.

Overall, we found that areas where the client was lightweight allowed more rapid prototyping and bugfixing. We recommend other projects be constructed with a small client for these reasons, as well as avoiding duplication of code and a reduction in application size by limiting client-side dependencies.

5.4.4 Test Early, Test Often

The best source of feedback and advice we received was from the shared document we circulated alongside our two private test releases. We specifically chose friends and colleagues who knew us well enough to be able to have honest discussions on the weaker aspects of the application. We carried out the testing by sharing Android application packages with Android users and inviting iOS users to participate in private beta testing via Apple's TestFlight system. We were able to implement the majority of the suggestions made, many of which have become central components in the final game. This stage highlighted the importance of push notifications and streamlining the user experience. Specifically, our test users found that they would often forget to check whether they had moves to make. Before testing we had investigated the feasibility of implementing push notifications, but were unsure if they were worth the time to develop. Following testing feedback, we made this a priority.

The user interface, colour scheme and card art of the final application are a result of feedback from our test users. As shown in Figure 5.12, the cards went through a series of designs. Responding to test feedback that character cards were too complicated, the final designs were significantly simpler. We also received specific advice, such as blacking out the action description of a stunned character to make it clear that they could not act. Having an ongoing dialogue throughout development with invested parties, meant that we could rapidly pivot to accommodate their suggestions.

Figure 5.12: Evolution of the Barbarian card artwork throughout the design process from initial prototype (*left*), to internal testing version (*centre*) and current version (*right*)

From analysis of our test data we discovered a gap between the data we were collecting and possible useful information we could capture. Specifically, we realised we could log user interactions with the application, noting actions they performed, when they performed them and what the result was if any (for example, "a user searched for another by their username and found they had no free game slots"). This idea was a result of realising that even amongst our dozen test users, there were distinct styles of interacting with the application. We thought that classifying these interaction styles would be of interest.

Testing allowed us to identify areas in both the application and the dataset that were lacking. We would encourage future researchers to get early versions of their applications into the hands of testers multiple times before finalising their system. There were many improvements made to RPGLite specifically because we had others test it, and could assess it across a suite of target devices. We structured the format of the feedback we received from testers in our shared document by grouping requested feedback under specific headings and directing them to features in which we lacked confidence. This helped to scaffold the insightful conversations amongst our test users, and we strongly recommend others make an effort to facilitate a similar dialogue.

5.5 Conclusion

What we took away from the experience of developing RPGLite does more to reinforce already existing best-practices than to suggest new ones. Many of the roadblocks to progress in development of the game could have been prevented with suitable adoption of agile development techniques [76]. It was due to the aims of the experiment evolving alongside the development of the application and Tom joining after the project's onset that agile methods were not used

throughout. As an anecdotal account of projects undertaken in our particular circumstance these lessons may be useful for future researchers who find themselves in the position we were in.

Chapter 6

Balancing the Application

"In which the balance of RPGLite, the mobile game, is considered using automated reasoning and then analysed through observed play."

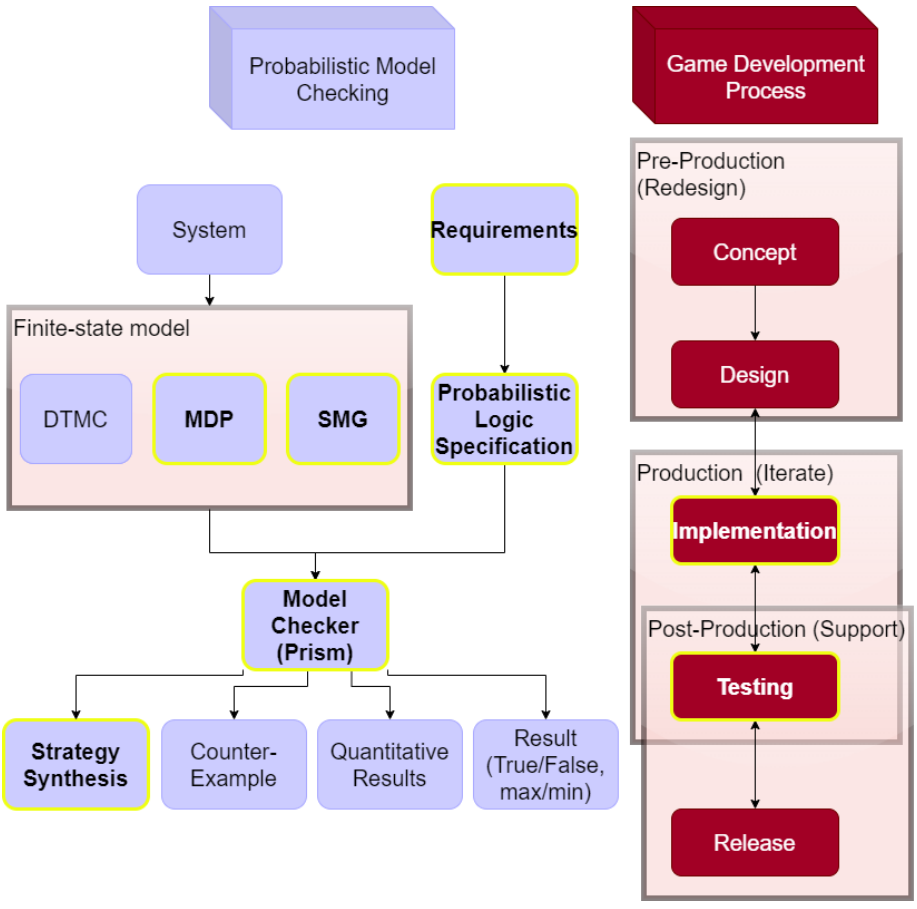

Figure 6.1: Chapter 6 areas.

6.1 Introduction

In this chapter the design of the game portion of RPGLite is described in detail. The application is more than just the game itself, all non-game systems are described in Chapter 5. We go on to analyse the player data and reason about the extent to which the game was balanced. The game was updated with a new configuration, we compare how the two configurations compare and how players learned each system. The analysis performed on the player data is then compared to predictions made using model checking to validate the techniques presented in Chapter 4 as well as extensions to those techniques.

For the analysis of a game configuration's balance we introduce two properties which can be verified both prior to release using model checking and in the player data. We demonstrate how both are verified and discuss whether they alone are sufficient to prove that a game is balanced, concluding that in their original form they were not. We also use model checking results to try and predict the metagame ahead of time. We tune a simple predictive model using results from automated analysis and train it with timed segments of player data, then compare predictions to future player data. Ultimately these investigations prove to be unsuccessful since previous character popularity was a better predictor of future character popularity.

6.2 RPGLite 3

RPGLite 3 is an extension of the forms of RPGLite described in Chapter 4. The rules have been expanded to include forms of forfeiture, to formalise character selection and to limit the time between actions. The main difference is the expansion of the material pool from the 5 characters of RPGLite 2, to 8 characters. The new characters are the Monk, Barbarian and Gunner, their sprites are shown in Figure 6.2. A successful Monk attack allows the player to act again as if their opponent has skipped their turn. The Barbarian does additional damage when they have low health, the value of the additional damage done is described by the Barbarian's **rage damage** and the health at which the barbarian does more damage by the **rage threshold**. The Gunner inflicts a small amount of damage when their attacks miss, the damage done on a miss is given by the Gunner's **graze** attribute. In total 29 values make up a configuration for RPGLite 3.

6.2.1 Character Mechanics

The 8 characters in RPGLite are the Knight, Archer, Wizard, Rogue, Healer, Monk, Barbarian and Gunner. Each has a health, accuracy and damage attribute, some have additional attributes to describe their unique action. The actions for all characters are summarised in Table 6.1. The actions were devised with model checking in mind, i.e., to limit the state space of the models. The Wizard is the only character who requires additional information stored in the state – the

Figure 6.2: The new characters introduced for RPGLite 3, left to right: The Monk, The Barbarian and the Gunner

Character	Action
Knight	Targets a single opponent
Archer	Can target both opponents
Wizard	Stuns opponents they hit which prevents them from acting on their next turn
Healer	Heals an ally when they hit
Rogue	Deals additional damage to low health opponents
Monk	The player can act again when they hit
Barbarian	Deals additional damage when at low health
Gunner	Deals a small amount of damage when they miss

Table 6.1: Character actions in RPGLite 3.

stun values for all characters, all other character abilities are based only on their maximum and current health values.

6.2.2 Modelling the actions

A state in the models of RPGLite is given by the 19-tuple:

$$s = (turn, p1K, p1A, p1W, p1R, p1H, p1M, p1B, p1G, p1\_stun,$$
$$p2K, p2A, p2W, p2R, p2H, p2M, p2B, p2G, p2\_stun)$$

To show how the character actions are translated for the model, an example of all actions of a nondeterministic player is given listed below.

Knight attacks Knight

```
[p1_K_K] turn = 1 & p1K > 0 & p1_stun != 1 & p2K > 0 ->
    Knight_accuracy :
        (p2K' = max(0, p2K - Knight_damage)) &
        (turn' = 2) & (p1_stun' = 0) +
    1 - Knight_accuracy :
        (turn' = 2) & (p1_stun' = 0);
```

Archer attacks Knight-Archer

```
[p1_A_KA] turn = 1 & p1A > 0 & p1_stun != 2 & p2K > 0 & p2A > 0 ->
    pow(Archer_accuracy,2) :
        (p2K' = max(0, p2K - Archer_damage)) &
        (p2A' = max(0, p2A - Archer_damage)) &
        (turn' = 2) & (p1_stun' = 0) +
    Archer_accuracy * (1 - Archer_accuracy) :
        (p2K' = max(0, p2K - Archer_damage))
        & (turn' = 2) & (p1_stun' = 0) +
    Archer_accuracy * (1 - Archer_accuracy) :
        (p2A' = max(0, p2A - Archer_damage))
        & (turn' = 2) & (p1_stun' = 0) +
    pow( (1 - Archer_accuracy),2) :
        (turn' = 2) & (p1_stun' = 0);
```

Wizard attacks Knights

```
[p1_W_K] turn = 1 & p1W > 0 & p1_stun != 3 & p2K > 0 ->
    Wizard_accuracy :
        (p2K' = max(0, p2K - Wizard_damage)) & (p2_stun' = 1) &
        (turn' = 2) & (p1_stun' = 0) +
    1 - Wizard_accuracy :
        (turn' = 2) & (p1_stun' = 0);
```

Rogue attacks Knight, but cannot execute

```
[p1_R_K] turn = 1 & p1R > 0 & p1_stun != 4 & p2K > Rogue_execute ->
    Rogue_accuracy :
        (p2K' = max(0, p2K - Rogue_damage)) &
        (turn' = 2) & (p1_stun' = 0) +
    1 - Rogue_accuracy :
        (turn' = 2) & (p1_stun' = 0);
```

Rogue attacks Knight, and can execute

```
[p1_R_Ke] turn = 1 & p1R > 0 & p1_stun != 4 & p2K <= Rogue_execute ->
    Rogue_accuracy :
        (p2K' = 0) & (turn' = 2) & (p1_stun' = 0) +
    1 - Rogue_accuracy :
        (turn' = 2) & (p1_stun' = 0);
```

Healer attacks Knight, and heals an allied Knight

```
[p1_H_KK] turn = 1 & p1H > 0 & p1_stun != 5 & p2K > 0 & p1K > 0 ->
    Healer_accuracy :
        (p2K' = max(0, p2K - Healer_damage)) &
        (p1K' = min(Knight_health, p1K + Healer_heal)) & (turn' = 2)
        & (p1_stun' = 0) +
    1 - Healer_accuracy :
        (turn' = 2) & (p1_stun' = 0);
```

Monk attacks Knight

```
[p1_M_K] turn = 1 & p1M > 0 & p1_stun != 6 & p2K > 0 ->
    Monk_accuracy :
        (p2K' = max(0, p2K - Monk_damage)) & (p1_stun' = 0) +
    1 - Monk_accuracy :
        (turn' = 2) & (p1_stun' = 0);
```

Barbarian attacks Knight, normal damage

```
[p1_B_K] turn = 1 & p1B > 0 & p1_stun != 7 &
p1B > Barbarian_rage_threshold & p2K > 0 ->
    Barbarian_accuracy :
        (p2K' = max(0, p2K - Barbarian_damage)) &
        (turn' = 2) & (p1_stun' = 0) +
    1 - Barbarian_accuracy :
        (turn' = 2) & (p1_stun' = 0);
```

Barbarian attacks Knight, additional damage

```
[p1_B_Kr] turn = 1 & p1B > 0 & p1_stun != 7 &
p1B <= Barbarian_rage_threshold & p2K > 0 ->
    Barbarian_accuracy :
```

```
        (p2K' = max(0, p2K - Barbarian_rage_damage)) &
        (turn' = 2) & (p1_stun' = 0) +
    1 - Barbarian_accuracy :
        (turn' = 2) & (p1_stun' = 0);
```

Gunner attacks Knight

```
[p1_G_K] turn = 1 & p1G > 0 & p1_stun != 8 & p2K > 0 ->
    Gunner_accuracy :
        (p2K' = max(0, p2K - Gunner_damage)) &
        (turn' = 2) & (p1_stun' = 0) +
    1 - Gunner_accuracy :
        (p2K' = max(0, p2K - Gunner_miss)) &
        (turn' = 2) & (p1_stun' = 0);
```

6.2.3 Configurations

The two configurations used are labelled as the *initial* configuration, under which the application was first released, and the *updated* configuration, which was introduced with the *"Season Two"* update. The values used for the health, accuracy and damage attributes in the application for each version are given in Table 6.2. The reasoning behind the changes between configurations is discussed in more detail in Section 6.4.1. As in previous chapters, for brevity we refer to pairs by the character initials, for example KA refers to a Knight-Archer pair. With 8 characters there are 28 possible character pairs which a player can choose from.

6.3 Methodology

In analysing balance we are concerned primarily with the comparative viability of the characters as potentially game-winning material. Recall that optimal strategies are strategies for a team which perform best against the best opposing strategy against them. As game balance is an ill-defined concept, we instead consider two properties of a balanced configuration of the game:

1. no strategy exists for a material set which is best played against by itself;
2. every material unit is involved in at least one material set which is optimal against another material set.

One of the research questions motivating this work was whether these two are sufficient to ensure a game is balanced. The first property is equivalent to stating that no dominant strategy exists. We consider a dominant strategy in RPGLite to be a strategy so effective that there is no other strategy using any pair of characters which has a probability of winning against it of over 0.5.

Character	Attribute	Initial	Updated
Knight	health	10	10
	accuracy	60	80
	damage	4	3
Archer	health	8	9
	accuracy	85	80
	damage	2	2
Wizard	health	8	8
	accuracy	85	85
	damage	2	2
Healer	health	10	9
	accuracy	85	90
	damage	2	2
	heal	1	1
Rogue	health	8	8
	accuracy	75	70
	damage	3	3
	execute	5	5
Monk	health	7	7
	accuracy	80	75
	damage	1	1
Barbarian	health	10	9
	accuracy	75	70
	damage	3	3
	rage threshold	4	4
	rage damage	5	5
Gunner	health	8	8
	accuracy	75	70
	damage	4	4
	graze	1	1

Table 6.2: Configurations used for RPGLite 3. Blue cells denote attributes that increased between seasons (buffs), red cells denote attributes which decreased between seasons (nerfs).

The second property is equivalent to stating that all available material units are optimal in some material set against some other material set. In RPGLite this means every character should be in a pair which counters another pair. Counters are the best strategy using the best material against a given strategy for a given material. The reasoning behind this is that it will lead to a richer gaming experience, broadening the selection of viable materials to the entire material pool, i.e., every RPGLite character can be justifiably chosen. Note that these requirements do not stipulate that all material sets are effective, only that all units are in at least one *effective* set.

We are concerned solely with competitive play, however playing to win is only one of a

number of possible motivations for play [67], [77]. The implications of motivation on our work are discussed in Section 3.4.6. We assume that players are playing to win and that all actions they take are done so in the belief that they are making the best move available. To encourage this behaviour, the mobile application RPGLite features several incentivisation systems to encourage repeated, competitive play. These systems are described in Section 5.3.2. The application was developed for research purposes all of which assume players will play competitively. As such the app informs players that they are helping to contribute to our research if they try to win every game. As the playerbase largely comes from an academic background or were associates of the developers, we believe the assumption that players played competitively is reasonable.

To validate our baseline requirements for a given configuration we generated a PRISM model for all 28 character-pairs against every possible opposing character-pair. The results of automated analysis calculating the optimal probability of winning for both players across all of these models generated using the initial configuration are shown in Figure 6.3. Values along the top-left to bottom-right diagonal are all 0.5 as these are symmetric games where both players have the same material and therefore the same strategies available to them. For example, from the figure it is clear that AW and AR are not effective pairs (shown by the mostly blue 8^{th} and 9^{th} rows or yellow 8^{th} and 9^{th} columns) as strategies exist for most other pairs which have high probabilities of winning against even the best AW and AR strategies.

Recall that we refer to the pair which has the highest optimal value against another as the strongest *counter* to them, as they best counteract their effectiveness – this is consistent with gaming terminology. The material used in the counter is referred to as the counter material or in the RPGLite context, counter pairs. One way to visualise the relationship between the pairs in RPGLite is to show the counter pairs for each pair using an optimality network Section 4.9, as in Figure 6.4 (top). The networks are enhanced from the form used in Section 4.9 for ease of reading, using colours rather than arrows and weights rather than labels. We also present a new visualisation akin to optimality networks, effectiveness networks, which show all material sets which have optimal values above some bound against other sets. The first requirement of non-dominance is satisfied if every pair is countered by some other, in the optimality network this is shown by every vertex having in-degree exactly 1, without having an edge to themselves. Our second requirement for balance is equivalent to every character being in some pair which counters another. For the initial configuration this property is satisfied because:

- the Monk is best against KA in MB,
- the Barbarian is best against KA in MB,
- the Wizard is best against KW in WM,
- the Knight is best against KR in KB,
- the Gunner is best against KH in MG,

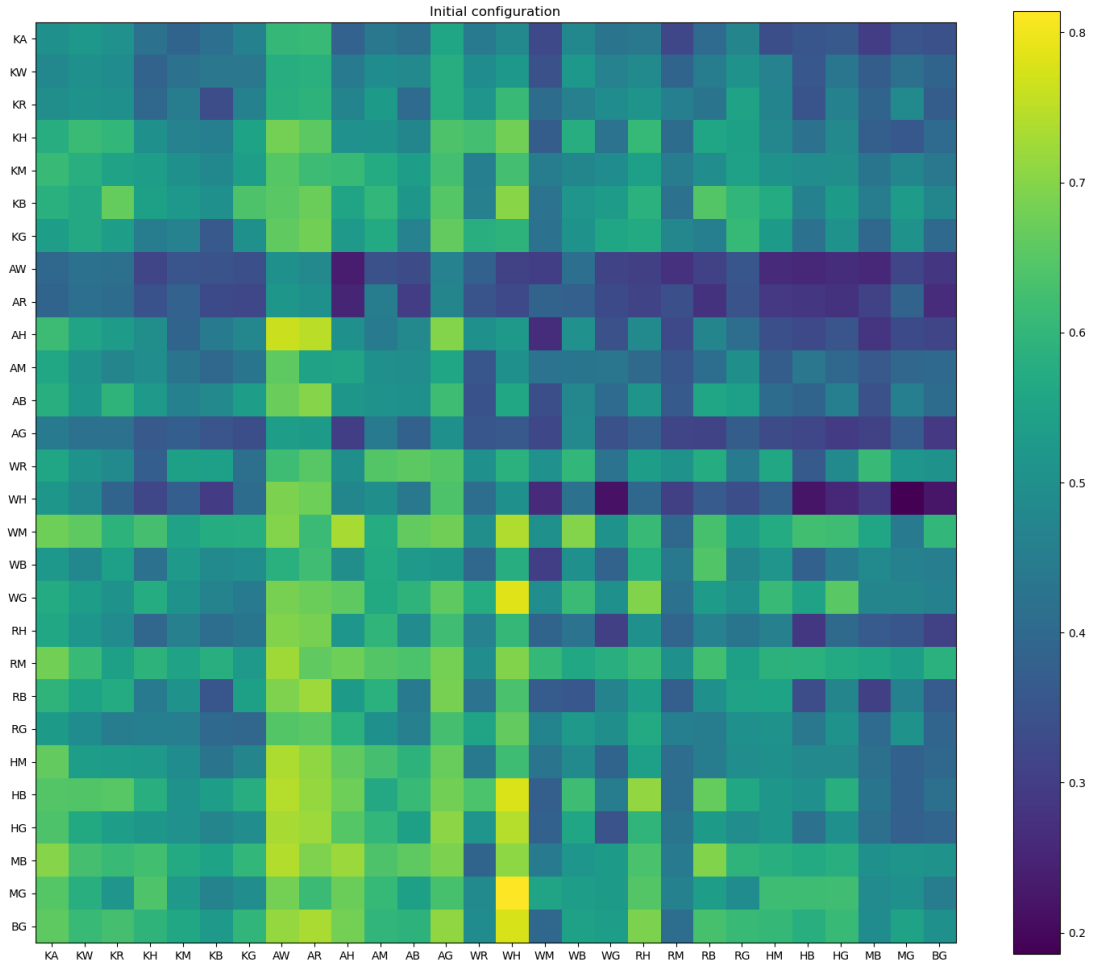

Figure 6.3: Optimality heatmap for the initial RPGLite configuration. Values shown are row material versus column material.

- the Rogue is best against KB in RM,
- the Archer is best against AW in AH and
- the Healer is best against AW in AH.

This example is not unique, e.g., the Monk is also best against AH in WM.

The initial configuration was one of several candidate configurations that arose from the automated processes laid out in Section 4.9.4, scaled up to RPGLite 3. It was chosen because it satisfied the qualities of balance, but also to test if one pair beating another with a probability over 0.5 was the threshold for dominance. The RM pair is countered by WR with a value of 0.5054 (i.e., RM beats WR when both are played optimally with a probability of 0.5054). We wanted to see if players used the RM pair as if it was dominant and if the counter pair (WR) was identified. In effect we were examining the specificity with which players could ascertain which materials were counters.

Full records on 3,413 games for the initial configuration of RPGLite were collected along with a further 4,118 from the updated configuration. All of the gameplay data from the RPGLite

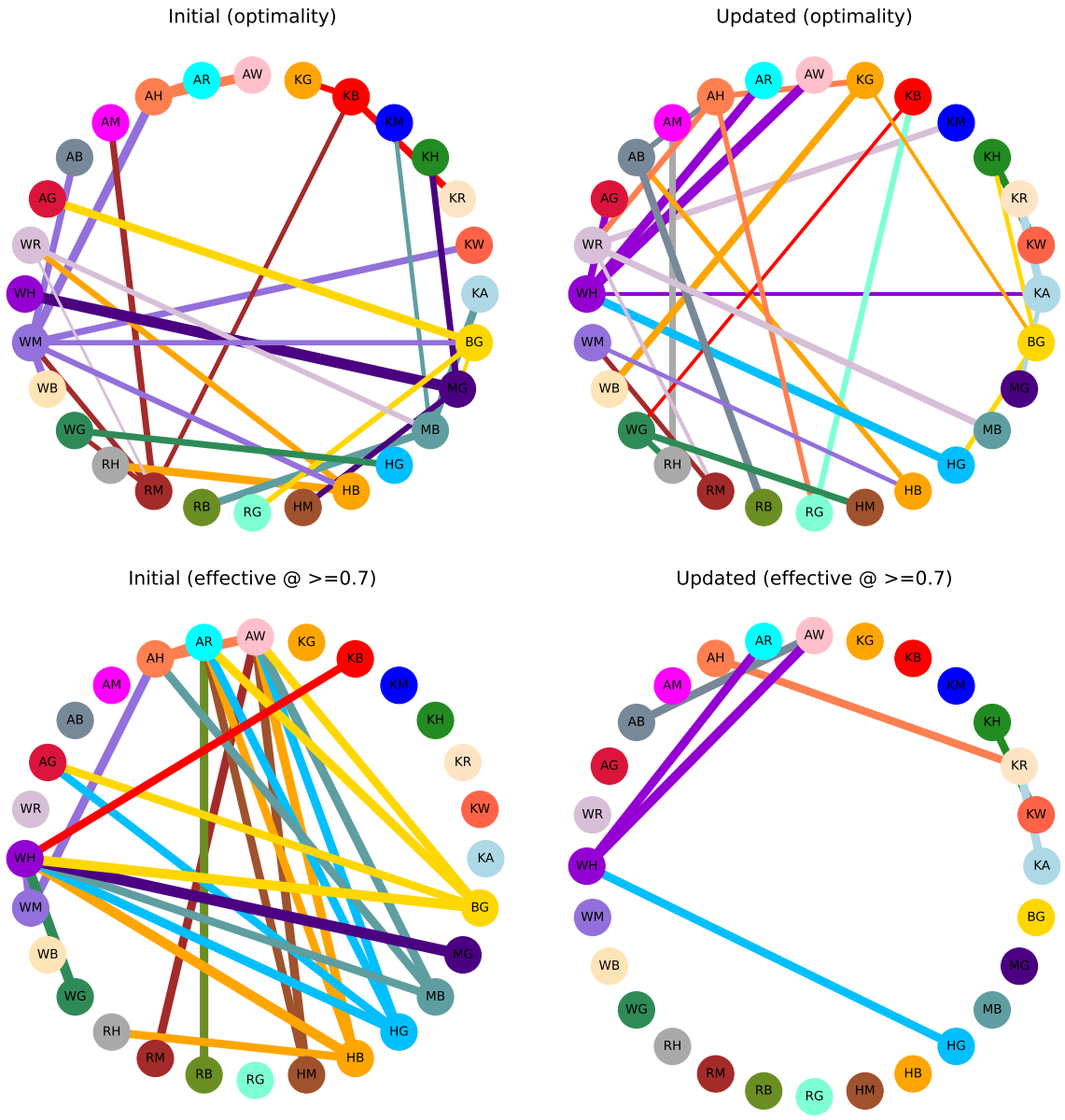

Figure 6.4: Top: coloured lines denote the pair which is a counter to the other pair on an edge. Bottom: coloured lines denote all pairs which are highly-effective against the other pairs on an edge. Thickness denotes probability of winning where thicker lines are a stronger counter.

dataset is available online [72]. We only consider games which were successfully completed rather than those in which a player forfeited or abandoned by taking too long to act. Game records are timestamped at the beginning and the end and detail the players involved, every move made and the result of every roll (0-99). The rolls determine if actions either succeed or fail based on whether they are greater than or equal to 100 minus the accuracy of the acting character, e.g., a Wizard hits on a 15, but misses on a 14 under both configurations. Using this dataset we can make judgements about how balanced the configurations are by considering how varied the choices made by players were.

We describe the metagame over a given time period as a distribution of the most popular character-pairs. For example, the first 200 games played in the initial configuration saw the Rogue picked 206 times, the Archer 129 times and the Knight picked 108 times. The most popular pairs were KR (55 times), AR (52 times) and WR (31 times). A natural evolution for the metagame would be for the next time interval (games 201-400) to see a rise in popularity of materials which are effective against those used in the previous interval. We attempt to predict the metagame by using a function of material popularity over an interval k and the relationships identified by the automated analysis to predict the metagame at interval $k+1$. For some of this work we considered which material-pairs were *effective* against one another, under the assumption that players would not be able to identify the single best counter material, but could identify beneficial matchups between material. This gives results similar to considering only counter materials, the difference being not all pairs are effective against any other while some are effective against multiple others. We show the effective relationships for both configurations using a 0.7 threshold for effectiveness in Figure 6.4 (bottom) where a given pair is considered effective against another if the optimal strategy for that pair against the opposing pair has a probability of winning of greater than 0.7.

6.4 Results

In this section we evaluate the balance of the two released RPGLite configurations.

6.4.1 Game Balancing

Initial Configuration

A common technique for visualising balance in games with asymmetric material such as RPGLite, is to plot the popularity and comparative success of the material using a balancing matrix (see Section 3.4.8). These matrices are used to explain the rationale behind balance patches in video games where the material is tweaked to bring it more in line with expectations. The visualisation of results from the 3,413 games played under the initial configuration are shown in Figure 6.5. The Rogue was the most chosen character, used 2,624 times (it is possible for a character to be chosen by both players) and winning 1,310 times, whereas the Archer was

the least popular character, used 974 times and winning only 412 times. From the results one can see that success and popularity are highly correlated, suggesting that players were indeed motivated to win. If these results alone were used to identify imbalanced characters, then one would likely consider the Archer too weak, as well as the Healer to a lesser extent, and would consider reducing the strength of the Monk and Barbarian.

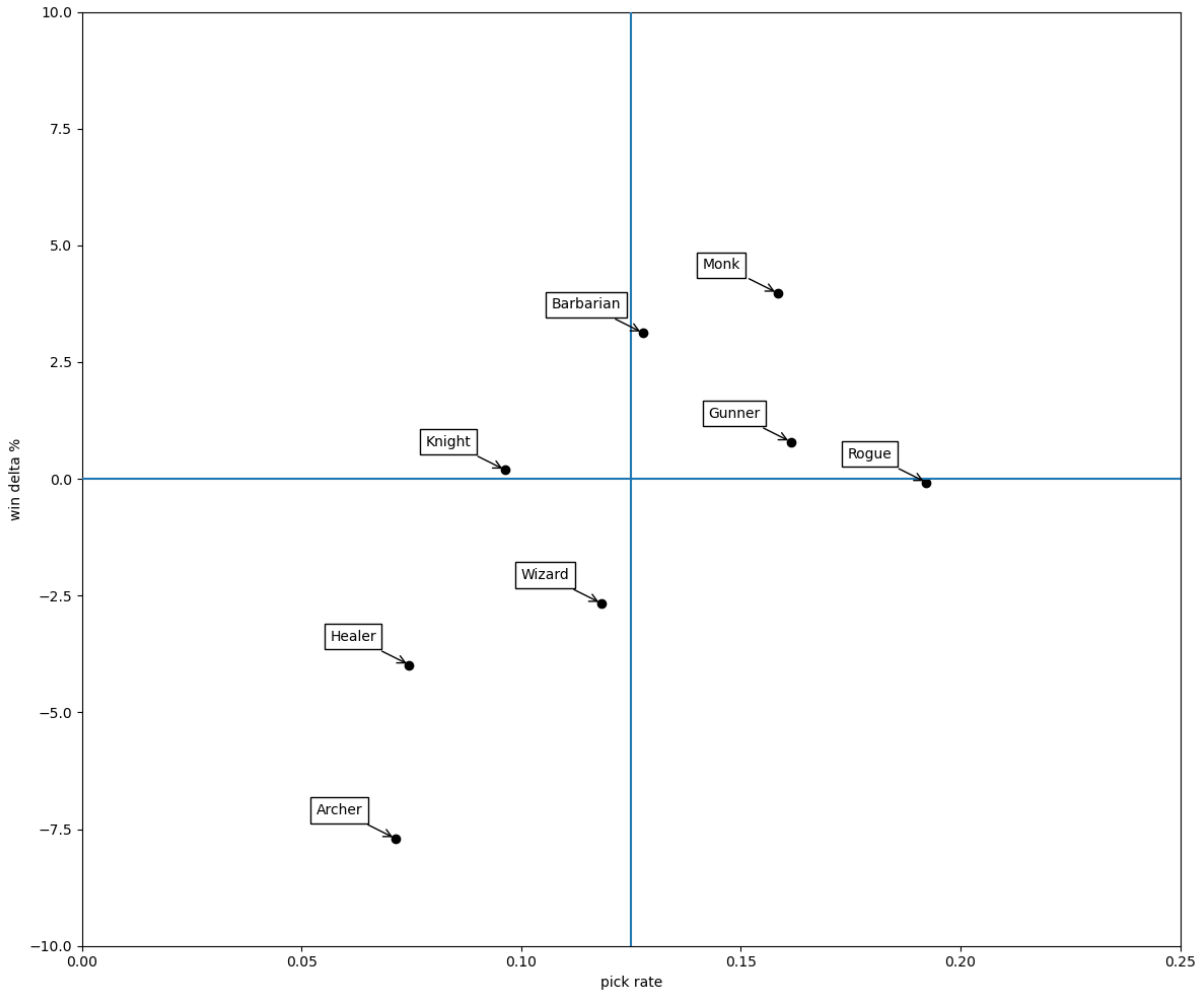

Figure 6.5: Balance matrix under the initial configuration.

We found that aggregated results from automated analysis were aligned with what the players were observing. The average optimal value for each character, across all their pairs against all other pairs, for the initial configuration, gives a similar ranking to win-rates observed from play. The Archer performs the least well at 0.416, followed by the Wizard 0.483, the Rogue 0.489, the Healer 0.495 and the Knight 0.498. Model checking suggests the strongest characters are the Gunner 0.523, the Barbarian 0.544 and the Monk 0.552. Note the similarity between the rankings found through model checking and the success rates shown in the y-axis of the balancing matrix in Figure 6.5. The only difference between the two was that the Healer was overvalued during automated analysis. For the updated configuration, the average optimal values were less accurate as an indicator, overvaluing the Healer and the Knight and considerably undervaluing

the Barbarian, but it did correctly rank the other five characters.

Verification under the initial configuration for the two baseline properties showed they were satisfied, but the RM pair was almost dominant. RM has optimal values of over 0.5 against all other pairs apart from WR, against which the optimal value is 0.4946. This relationship is shown by the thinnest edge in Figure 6.4 (top-left) from WR to RM, representing the 0.5054 optimal value. There is a WR strategy which is more likely to win than lose against any RM strategy, however the next best material to use against RM, is RM itself. We wanted to investigate whether the effectiveness of RM was discovered by the playerbase and whether the marginal advantage of WR over RM was noticed. We found that after some initial discovery RM became very popular and remained so for the duration of the season, ending the period of games played under the initial configuration as the most popular pair, as shown in Figure 6.6. WR did not increase in popularity in response to the rise in use of RM, suggesting that players did not identify it as an effective counter. The ubiquity of RM in later games was what motivated the change in configuration. It is possible that the configuration of the game changed before the playerbase noticed the comparative strength of WR over RM, but we believe it is more likely that the favourable matchup was too slight to be noticeable.

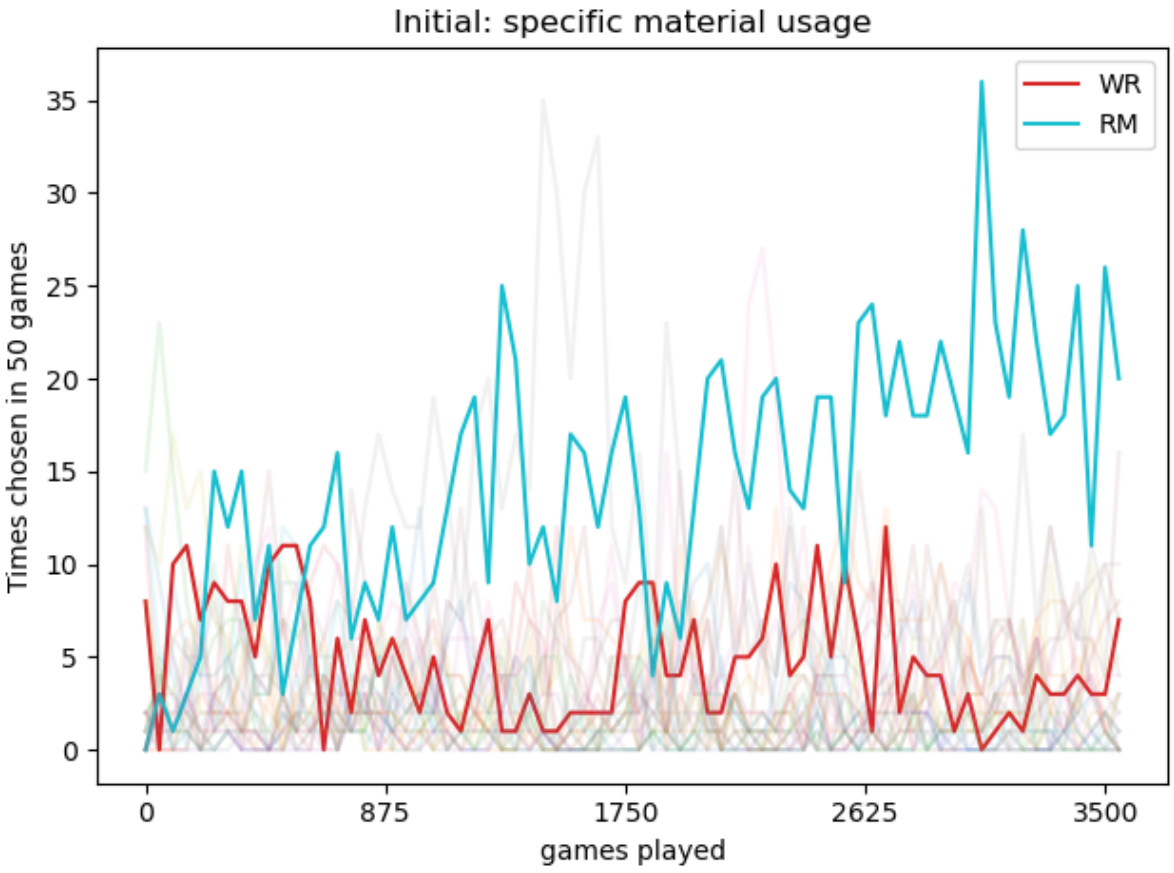

Figure 6.6: WR and RM usage under the initial configuration compared in buckets of 50 games shown sequentially.

The RM pairing is so effective in part because their actions work well together. A high-level strategy which describes the optimal strategy for most RM matchups is to target the more dangerous opponent with the Monk repeatedly until they can be executed by the Rogue. A character with 8 health can be reduced to 0 in a single turn using this method with a probability of 0.384, because a player who hits with the Monk can then act again on their turn. Opposing characters with a health of 10 (Knight, Healer and Barbarian) can be eliminated in a single turn via this tactic with a probability of 0.2458. The WR counter strategy is to stun the Monk with the Wizard and execute them with the Rogue, preventing them from using both characters in tandem. The Wizard, in stunning the Monk, reduces them to a health of 5, from which they can be eliminated by the Rogue in a single action, meanwhile the RM opponent is forced to use the Rogue, who is only strong when opposing characters have low health. This is not a complicated strategy. We are able to describe the strategies in limited detail by cross-referencing the output of strategy synthesis using PRISM when exporting the states and adversary files.

The probability of winning against all opponents may have been a factor. Given RPGLite's concurrent character selection, players did not know what characters they were likely to be up against beyond past games or knowledge of their specific opponent's preferences. For this reason it is reasonable to expect pairs with a high average optimal value against all pairs to be popular, rather than those with few counter pairs. The average optimal value of RM is 0.598, greater than that of WR at 0.531, suggesting it should perform better against other materials, which might explain why WR was less popular than expected. However, WM actually had the greatest average optimal value of 0.5982 and both BG and MB had values over 0.59, but were significantly less popular with players.

Updated Configuration

The main objective when changing the configuration was to strengthen the Archer and weaken the Rogue and Monk. Other changes were introduced in order to produce a configuration which satisfied the baseline requirements whilst still using round numbers (with accuracy values being multiples of 5). We also wanted to ensure no pair had a minimum optimal value as high as for RM in the initial configuration. The updated configuration has several minimum optimal values grouped together near the highest value found, unlike the initial configuration where RM was clearly the strongest. The top 4 pairs in terms of minimum optimal value in the initial configuration were: RM (0.4946), MG (0.4511), KM (0.4312) and WG (0.4203); in the updated configuration they were: WG (0.4675), BG (0.464), KA (0.4541) and RM (0.4524). The updated configuration had lower optimal values in general than those for the initial configuration. This is shown in Figure 6.4 (bottom) where every optimal value over 0.7 is represented by an edge. There are only 7 such edges for the updated configuration and 26 for the initial configuration.

The configuration was changed 27 days after the app was first released. This coincided with efforts to recruit more participants. Of 66 players to play over 10 games under the updated

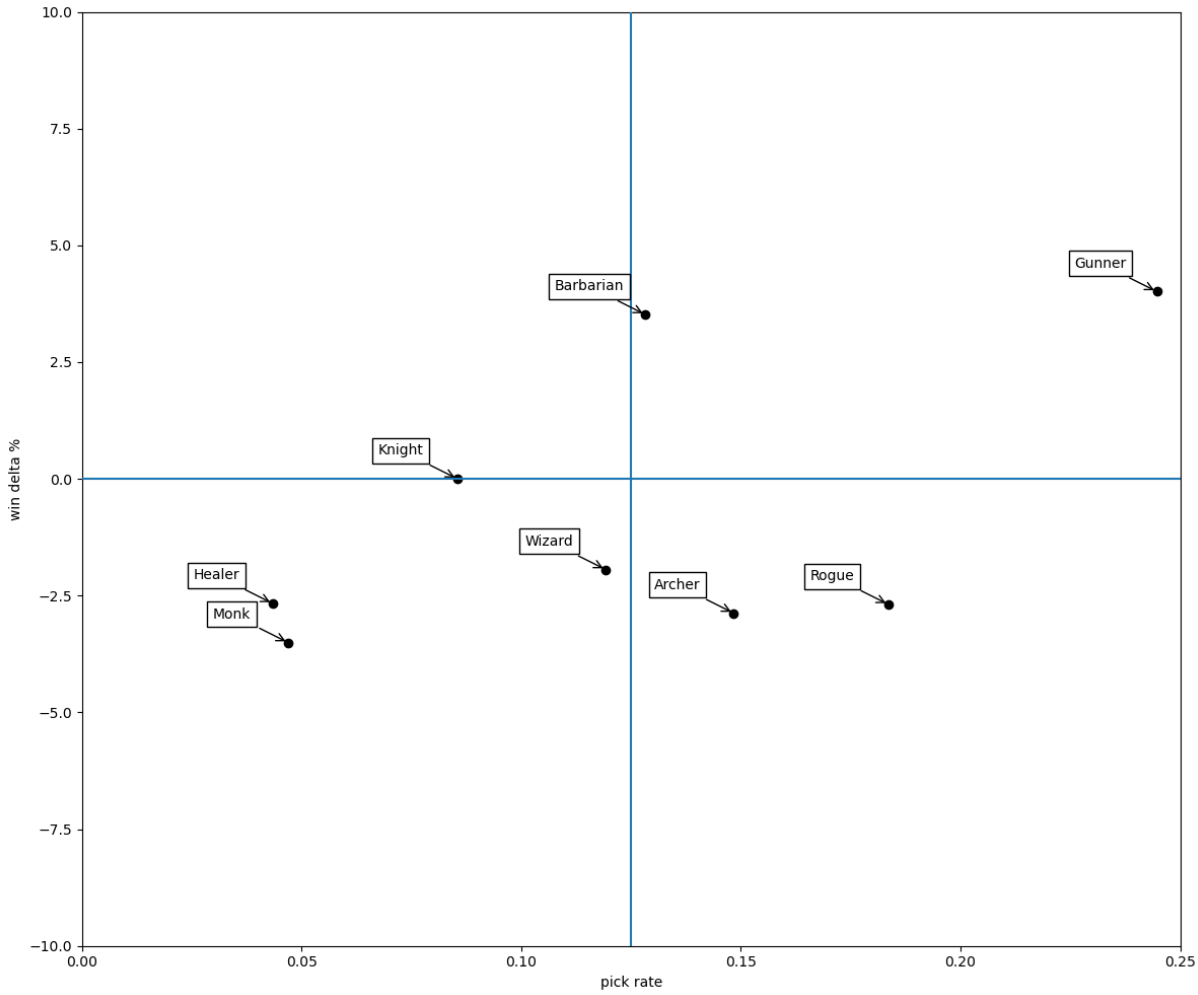

Figure 6.7: Balance matrix under the updated configuration.

configuration, 30 had also played 10 games under the initial configuration. The effect of the different pool of players is not clear, familiarity with the game may have reduced the difficulty in identifying strong pairs in the updated configuration or could have lead to confusion as previously *good* material performed less well. That being said, any effect was mitigated by the changes being forecast a week ahead of time in messages and notifications displayed to players and described afterwards in the same form.

In total there were 4,118 games played under the updated configuration and the Gunner was the most popular character by a significant margin, being chosen $4,038$ times. The second most popular was the Rogue which was chosen $3,021$ times. The Monk was the least successful character, winning only 46.5% of games, a considerable shift from the initial configuration where it was the most successful character. The trend of a correlation between success and popularity remained between seasons, a good example of this is the Archer who improved whilst remaining one of the weaker characters, but saw a rise in popularity from the least to the third most popular, despite losing more often than winning. The balance matrix under the updated configuration

is displayed in Figure 6.7. Whilst the updated configuration was successful in mitigating the weakness of the Archer and bringing the characters more in line with each other, the unrivalled success and popularity of the Gunner indicates a clear flaw.

In both configurations of the game, for each character there was a pair which was effective (winning over 50% of their games), shown in Figure 6.8. This shows that all characters *can* be used successfully. A sensible design aim for a game with asymmetric material such as RPGLite is that all material is effective in some situations, but no material is effective in all situations. At RPGLite's level of complexity, these situations are the pairs chosen, i.e., it is undesirable for all pairs to be effective, some characters shouldn't form an effective pair.

When comparing the results from strategies generated via model checking to the player data there are several notable discrepancies. As with all work modelling human users, there will be errors owing to human fallibility. Consider the balance matrix for the updated configuration in Figure 6.8, here AR is a significant outlier, being chosen far more often (1,145 times) than was justified by their low win-rate (of 44.19%). Other character-pairs in both configurations conform to a logarithmic relationship between success and popularity. AR was selected 789 times by a single user in the updated configuration, accounting for the majority of its uses. The model checking results (visualised in Figure 6.3) shows us that an AR pair is expected to win against only 3 other pairs; AW (0.5357), AM (0.5096) and AG (0.5004). The weakness of AR in reality is shown by the poor win-rate from the player data. Whilst the model checking and subsequent analysis can identify weak material accurately, it cannot account for human biases. There are other reasons why players may choose material than success. This outlier (the high popularity and low win-rate of AR) is in the second season, when players are more likely to have grown apathetic, which may have been a factor. Some players may have chosen characters based on the character artwork or a preference for their play-style, rather than being motivated by competition. For this reason, we do not believe these seemingly anomalous results imply that either configuration is unbalanced. Furthermore, this behaviour will be exhibited by players of any game and to ignore them would not accurately represent how games are played, although it may impact numerical results and weaken inferences.

6.4.2 Metagame Prediction

For our purposes we consider pair popularity at a given interval of games played to be representative of a metagame. Our aim was to use both previous popularity at a game interval and the automated analysis performed to analyse game balance to generate accurate predictions of character-pair populations at future intervals. For example, if over 10 games of Rock-Paper-Scissors two players predominantly choose Rock, you might expect over the subsequent ten games for the players to use Paper far more often, expecting to beat Rock, then in the next ten games you might expect Scissors to be popular, using similar reasoning.

We set the interval value to be 200 games and used various predictive functions over all com-

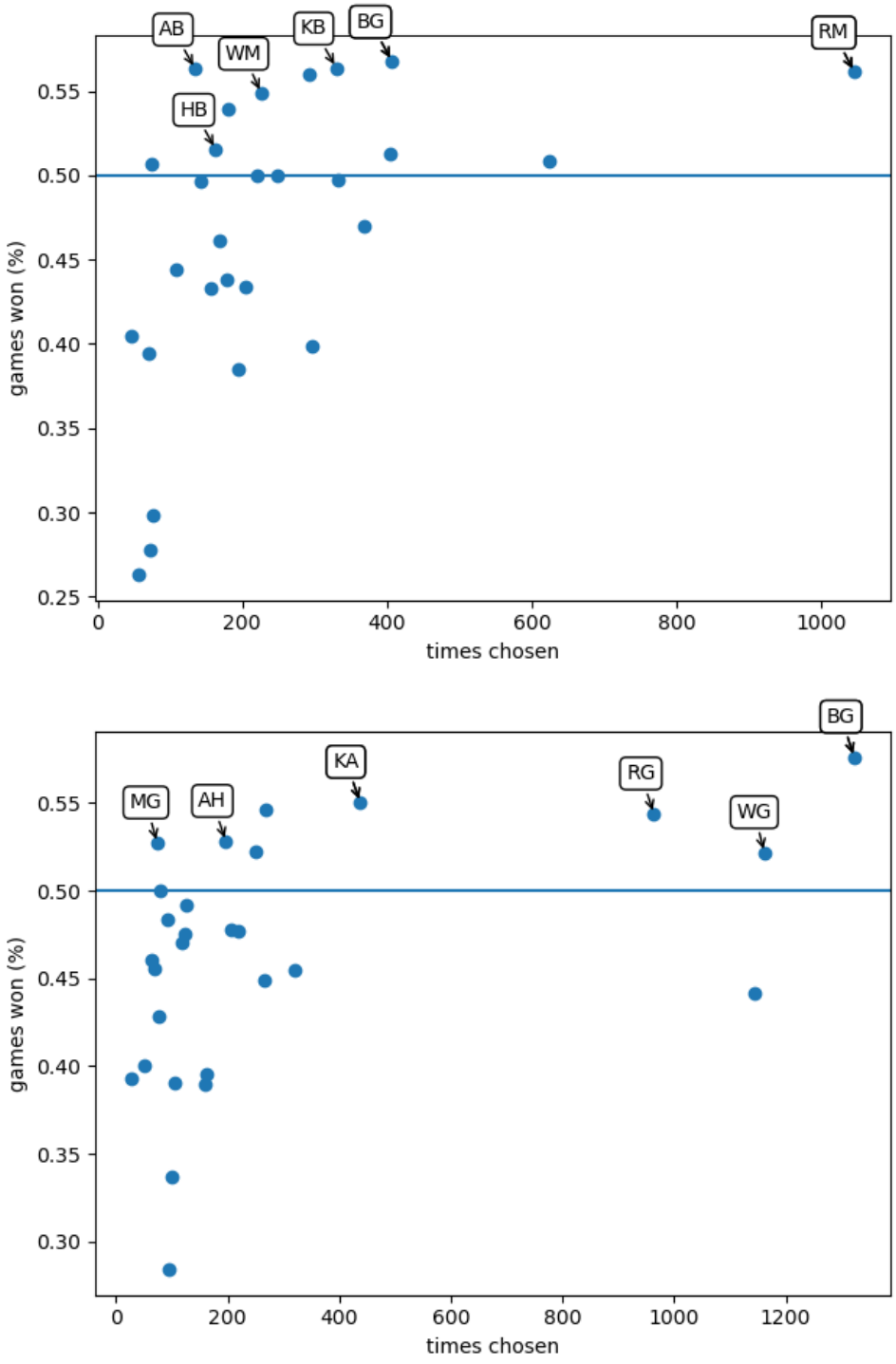

Figure 6.8: Pair-wise balance matrix for the initial configuration (top) and the updated configuration (bottom). Select pairs are highlighted showing that each character was in a *viable* pair, one that won more often than it lost.

plete intervals for both configurations. This meant that we used the first 200 games played to predict the character-pairs used in games 200-400, and then used the actual values from games 200-400 to predict games 400-600, and so on. We measure the accuracy of a predictor by calculating the mean squared error per character-pair (MSE):

$$\sqrt{\frac{1}{28}\sum_{p \in pairs}(predicted(p) - actual(p))^2}$$

and use the mean of these values for every interval under a configuration to give a configuration-wide measure of accuracy. The use of the MSE is a standard measure for calculating the precision of a predictive function.

The first technique we tried for predicting the metagame was to assume players would only use the counter material to popular material in the previous interval. For every instance of a character-pair being played in interval k, we assumed an instance of their counter would be used in interval $k + 1$. This is a naive approach that we assumed would prove to be more accurate the longer a configuration had been released as players identified which pairs were best against others. The predictions using this method identify only a small set of material-pairs which may be used. This is because some material-pairs are counters to many others, whilst others are counter pairs to none. This meant the predictions differ from reality where, in most intervals, every pair was used multiple times. What is peculiar is that this method did not identify the most played material with any consistency, suggesting that being a counter material is less valuable than being effective against many other materials.

Given a vector of the pick-rates at interval k, pr_k, and the matrix of the optimal values between all pairs *opt*, an optimal distribution of pick-rates over interval $k + 1$ is given by calculating the matrix-vector product:

$$opt \cdot pr_k = pr_{k+1}$$

This was the second method that we used. However, this gives values with little variance, mainly due to the random component in RPGLite meaning any pair can win against any other. To counteract this we reduced all values by a constant so that the lowest predicted pair had a value of 0 then multiplied by a scale factor to give the correct sum of predictions of 400 (the number of character-pairs chosen in a 200 game interval). This gives results which are closely clustered together, consistently failing to predict the very popular pairs.

The matrix-vector product was not sensitive enough. The values for each pair are calculated as the sum of their effectiveness against every other pair. This is unrealistic, especially for players familiar with the game who should know that some pairs are ineffective against others. We also saw with the counter material prediction that some trends carry over from the previous intervals and need to be accounted for. In response to this we developed our final method for predicting popularity. For our last method we take the previous interval's values and add x/n to the value

Technique	Initial	Updated
Only counters	24.03	35.08
Matrix-vector product	13.83	20.41
Altered history	10.11	12.79
History	9.82	9.78

Table 6.3: Results of metagame prediction.

for any prediction, where x was the popularity in the previous interval of a pair against which it was *highly-effective* (optimal probability > 0.67) and n was the number of other highly effective pairs. We then multiplied these inflated values by a scale factor to get the correct total number of pairs predicted. We call this altered historical prediction. It was the most accurate of the three we devised.

As a benchmark we compare the popularity at future intervals to the popularity at present intervals without alteration like that performed in our final method. We refer to this as the historical predictor. The average MSE values for the three techniques outlined and for the benchmark are shown in Table 6.3. Although the altered history approach was more accurate than history alone in some intervals, overall it performed worse. The four predictors are shown for each character-pair over three intervals of games under the initial configuration in Figure 6.9.

6.5 Analysis

It is not the case that a game is either *balanced* or *unbalanced*, however we believe that both configurations of RPGLite were balanced to an acceptable degree, and that the updated configuration was *more* balanced than the initial one. They are balanced as all characters were used frequently and the most popular character-pairs were not repeatedly matching up against each other. The overwhelming popularity of RM under the initial configuration is concerning and suggests imbalance, however only 89 of the 3,413 games played featured RM against RM, fewer than 3%. The updated configuration had more diversity amongst the most popular character-pairs with 4 pairs being chosen significantly more than the other 24. The Gunner was prevalent amongst the most popular character-pairs, although not all Gunner pairs were successful, AG had a success-rate of just 38.99%. What would strengthen these statements on acceptable balance would be qualitative data from players stating that the game was fun to play. Regrettably players could not be surveyed, but the amount of games played implies a degree of enjoyment came from our participants / players.

The relative accuracy of the mean optimal values for a character in predicting their win delta through play is a potentially important observation. The values are straightforward to obtain and can highlight potential balance issues. The process of game balancing can be a laborious task

when done manually, any progress towards automated systems for game balancing is going to benefit game development. If this work could be advanced to accurately predict data similar to the pick-rates and win-rates of balance matrices then it could be a powerful game design tool.

A well balanced game should lead to diversity among the game materials used, so the metagame should shift significantly over time as materials rise and fall in popularity. A provably diverse metagame implies a well-balanced configuration of a game. Furthermore the ability to predict which materials will be popular in the near future would benefit developers in preparing appropriate items for in-game shops or guiding future balance changes. Game developers can affect the popularity of material through promotional and incentivisation techniques, such as price drops, trial periods or UI changes to feature the material more prominently. If these changes in popularity could have a predictable effect on future metagames then developers could have more agency in steering the direction of their games.

In trying to predict the metagame we were unsuccessful, failing to improve accuracy beyond previous popularity. This may have been for several reasons. The outcome of a game in RPGLite relies heavily on luck which can make identifying areas for improvement difficult. Players can play optimally with an effective pair and still lose. Furthermore there is little information fed back to the player following an action and the binary outcome of a game without the ability of turn-by-turn analysis means there is nothing to suggest they have made a mistake. These will have made it more difficult for players to identify which pairs were effective against which others, certainly compared to major games with active communities discussing materials and strategies in detail.

For metagame prediction in the form we attempted to be successful, metagames need to develop based predominantly on players recognising what ways of playing the game are good. One of the clear themes that arises from this work is that, even with a relatively simple game like RPGLite, players do not play perfectly so this seems unlikely when considered by players acting individually. Metagame development is likely to be driven by several factors in addition to the playerbase identifying locally optimal ways of playing. These factors may include several intangible factors such as material aesthetics, how fun given material is to play, attempts to quantify the metagame and predictions about the effectiveness of newly released or recently updated material. These factors come from communities of game players, it is also communities who are more likely to be capable of identifying locally optimal ways of playing. Future research on the various factors that influence metagame development and their importance to new ‘metas’ is warranted.

A limitation of this work is that it involves only a single experiment with a limited player pool. Given more data our results would have been more accurate, although we believe we have collected a sufficiently large dataset to conclude that model checking is a viable tool for game balancing. The players of RPGLite were not exclusively game players, some found the game through the app store pages, but most were recruited through advertising via university emails

and among game research groups. We believe that the game is simple enough and has sufficient in-game help that the level of experience of the players did not have a major impact on the results.

Verifying the baseline properties takes approximately 20 minutes for a candidate configuration, depending on the size of the state space which is inflated mostly by high health values. Characters which require extra information to be stored at the state level are a major expense in this regard. The Wizard is one such character requiring a variable to track which characters can be stunned, if any. RPGLite, specifically the characters in the game, were designed to be suitable for model checking. To perform the same analysis on a large modern game would likely not be possible on current hardware. However, the complexity of games is not likely to increase much further as the capacity for players to deal with the complexity will not increase. Meanwhile computational hardware is improving and techniques to tackle the state explosion such as symmetry reduction [57] and better model construction [78] are also advancing. In the future, it is possible that similar verification could be performed on large, professionally developed games, but this will not happen for some time. A more immediate impact from the work in this thesis is the application of the knowledge gained by studying the play of smaller games alongside formal verification to larger games.

6.6 Conclusion

Game balancing often requires extensive trial-and-error testing by the developers or quality assurance teams. The alternative for electronic games is to update them prior to release as part of a *games as a service* model [79]. It is the norm for modern video games to be updated long after release with new content, often expanding on the available game material. These additions can easily upset the balance of a game and require yet more testing. The obvious downside to balancing a live game is that players act as the testers and may end up playing an inferior game as a result. Any move towards automated or semi-automated game balancing will be a significant help to the game development industry.

We have outlined a small-scale approach that uses model checking to quickly and effectively verify that a game configuration is balanced. Our results show that verification of games can be used to predict material strength and ensure predetermined properties of game balance. Whilst we were unable to use automated analysis to accurately predict changes in the metagame, we have described the reasons for this and believe more work in the area is warranted. Future work would benefit from a more detailed investigation of why material was chosen, perhaps incorporating qualitative data on player preference, as it does not appear that this was captured by predictions.

Despite the compromises made in developing a game suitable for model checking and the lack of developer expertise, the feedback on RPGLite was that the game itself was compelling. As a game experience, the application RPGLite is shallow, there is little by way of progression and the outcome of games are largely dependent upon luck, yet the game was played repeatedly

by a wide audience of players. The configurations used for the game were not tested by anyone (developers or players) beforehand, rather only automated verification was performed using the novel model checking techniques outlined in this thesis. This is significant as the number of possible configurations for 29 values is great, approximately 1.975×10^{20}, yet 2 suitable configurations were identified with little manual effort.

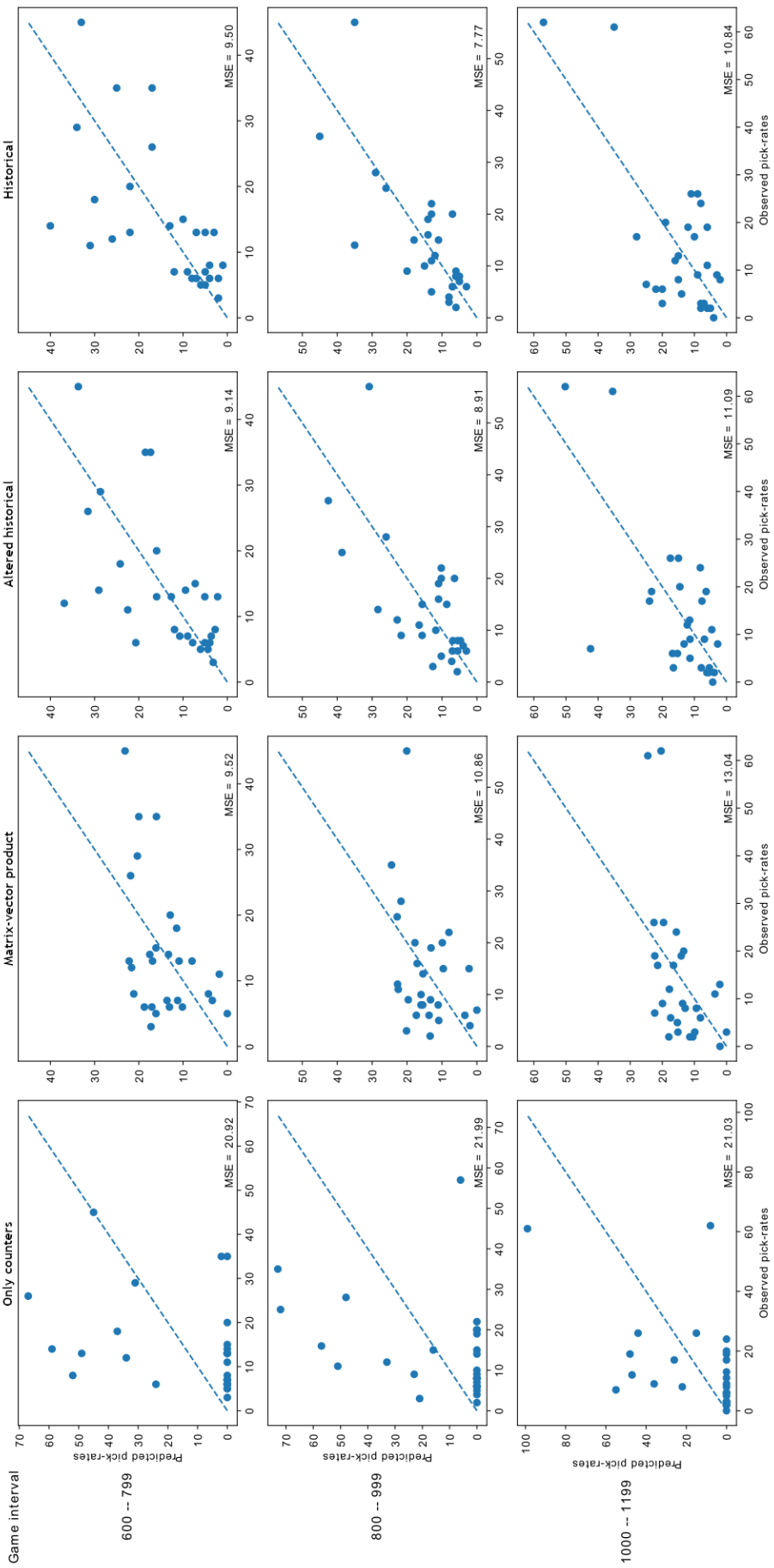

Figure 6.9: Metagame predictions over 3 concurrent intervals for the initial configuration using the four methods outlined in 4.2.

Chapter 7

Gameplay Analysis With Verified Action-Costs

"Taking insight from previous balancing automation and comparing it to player perceptions to analyse play, learning and the game's success."

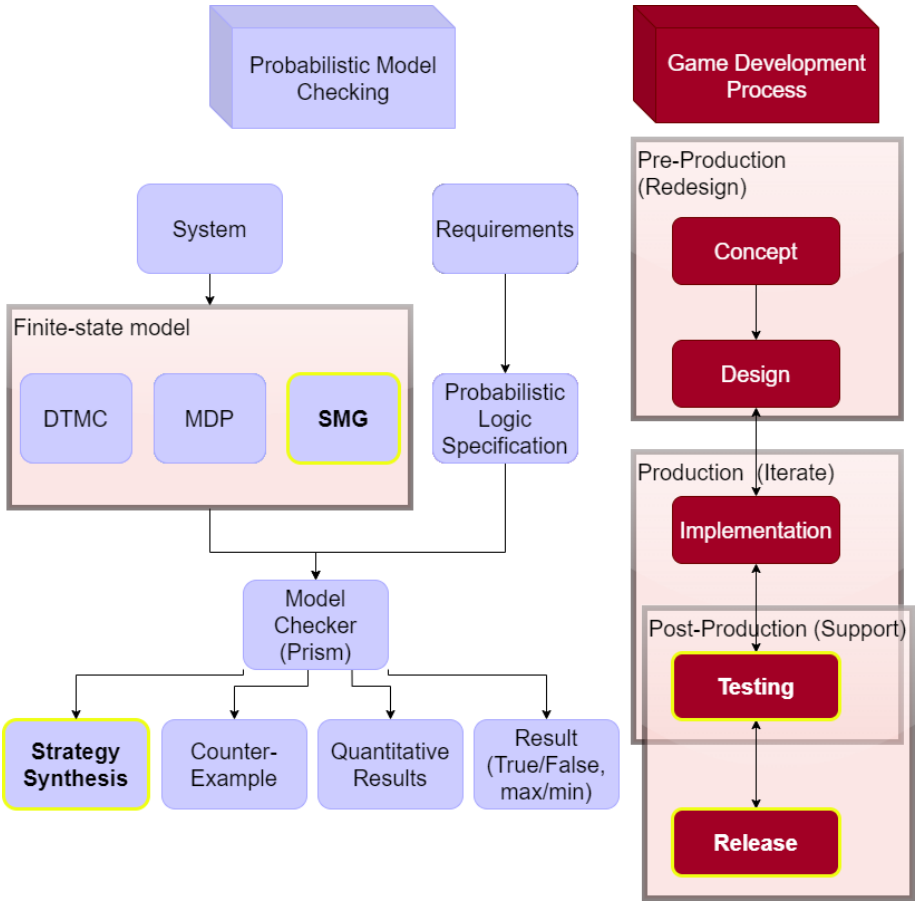

Figure 7.1: Chapter 7 areas.

7.1 Introduction

In this chapter we show how the results of model checking can be compared to player actions to develop insights into how a game is played. We use automated analysis to attribute a cost of every possible move of the game. These costs can then be used to identify areas where players frequently make mistakes when playing the game and measure the rate of player learning. Furthermore these costs have wider application for game analysis and for assessing whether a game is balanced. The techniques for automated balancing with model checking described in Chapter 4 assume that players will converge on locally optimal strategies. We have hypothesised that over time this is equivalent to players converging on globally optimal strategies. The use of costs allows us to measure the extent to which this is true.

Traditionally player skill is measured by a ranking system such as the Elo rating system [80] or Microsoft's TrueSkill [81]. Some variations of these ranking systems encapsulate additional information from games beyond win/loss data, such as the margin of victory [82], the number of player kills [83] or if the player is part of a group [84]. Even then these ranking systems are a compromised form of judging player skill, a better measure would be to consider every action a player makes and compare it to the best action they could have made in the same situation. This would not be possible for the majority of games where the variety of actions available and number of decisions a player makes it infeasible. However, it is possible for RPGLite.

RPGLite was designed to be suitable for such analysis, while still offering players a challenging puzzle to solve in choosing the best moves to make. Players take turns performing actions aiming to reduce their opponent's characters' health to 0. We calculate the *cost* of any action as the degree to which the acting player has reduced their optimal probability of winning by performing this action. We can then consider of the average cost of the actions performed by a player to give an accurate account of the extent to which the player has "solved" the game, i.e., how good they are. We believe this to be a fair indicator of player skill. We use the mean-average in this instance as the majority of moves had a cost of 0 – mode and median would likely be 0.0 in most instances.

With a game like RPGLite it is natural to assume that players will converge upon an optimal strategy relatively quickly. One would expect players to make fewer mistakes in their 100^{th} game than in their 1^{st}, for example. However other experiments have generally failed to identify this behaviour [85]. Unlike in the games often studied, like card games or prisoner's dilemma type games, optimal strategies in RPGLite are not mixed, the optimal action(s) to take are the same every time a state is visited. This is different to Poker for example where players are rewarded for acting unpredictably [86]. We use action-costs to study if players learnt to play RPGLite over time by seeing if their average action-cost decreases because they make fewer mistakes and the mistakes they do make are less costly. One thing we found was that there were several players who actually got worse over time and we will discuss the possible causes for this.

Action-costs can be used in other forms of analysis beyond measuring player skill. In par-

ticular, they can also be used to inform judgements on any area where a game developer would benefit from knowing the frequency and extent of mistakes made. For example, we have used cost values to calculate the average error per action for the different characters of RPGLite, which we can then compare to identify which characters are more complex to play (so may require further explanation to players). This informs decisions on game balancing as a character may be effective enough, but being played poorly due to a lack of clarity in their design. Furthermore we identify when players made mistakes consistently and analyse the reasons why to gain a greater insight into how the game is perceived by players. We present our findings about RPGLite informed by action-costs as a demonstration of how other games could benefit from a similar form of analysis.

In this chapter we first describe action-costs, a key contribution of this thesis, using example scenarios from RPGLite to show how they are calculated. We then detail the gameplay analysis we performed on RPGLite using action-costs, a study on player learning, a taxonomy of game characters and their complexity and an investigation of common mistakes. We then describe how action-costs can also be used to inform player rating systems and to educate players. Finally, we discuss the impact of our work and its suitability to large, professionally-developed games.

7.2 Action-Costs

In this section we introduce action-costs, how they are calculated and their utility.

7.2.1 Methodology

Recall the states in RPGLite2 are encoded as a 19-tuple describing whose turn it is and the health values of either player's 8 characters as well as which are stunned:

```
s = (turn, p1K, p1A, ..., p1_stun, p2K, p2A, ..., p2_stun)
```

Initial transitions involve a coin flip to set the turn and each player reducing the health of 6 of their 8 characters to 0, representing character selection. The stun values give the index of the character that is stunned. It is 0 when no character is stunned, 1 if the Knight is stunned, and so on. When calculating optimal values from each state we only consider player 1's probability of reaching a winning state, this is sufficient because the game is symmetrical, any state player 2 can reach can be rewritten for player 1. When parsing actions for player 2 we rewrite the state and action as if for player 1, for the state this means setting turn to 1 and swapping the 9 variables for player 1's character state with those for player 2. We use PRISM to generate **state lookup tables** giving the optimal probability of player 1 winning from any state. When generating the lookup tables we set the characters used by player 1 to each of the 28 pairs available and calculate the probabilities from each state reachable against any opposing material. Some examples from the Knight-Archer table are:

```
opt(1,8,4,0,0,0,0,0,0,0,0,0,2,6,0,0,0,0,0)=0.7454295392843602
opt(1,8,4,0,0,0,0,0,0,0,0,0,2,8,0,0,0,0,0)=0.4651701837999699
```

This tells us that when it is player 1's turn and they have a Knight with 8 health, an Archer with 4 health, their opponent has a Wizard with 2 heal, a Rogue with 6 health and no character is stunned, player 1's optimal probability of winning is 0.745, whereas if the opposing Rogue had 8 health, it would be only 0.465. The state lookup tables can be generated from the output of the verification performed by PRISM in the creation of optimality networks, described in Section 4.9.

PRISM can also be used to perform strategy synthesis [87]–[89], detailing the optimal strategy for a player – the strategy which has the highest probability of winning against an opponent trying to minimise it. This gives a single (optimal) action from every state which if performed when at that state will maximise the player's probability of winning. However, it does not evaluate the extent to which any other action is sub-optimal, i.e., reduces the probability of winning. It also fails to account for states where there are multiple actions which are optimal. This will happen in RPGLite, especially when a player uses the Rogue-Barbarian pair as these two characters share similar actions and attributes. To measure the extent to which an action is *correct* we need to consider the optimal probabilities of winning from states reached after taking all available actions.

In PRISM, action-costs can be calculated via the strategy synthesis of the SMG representations of RPGLite where players seek to maximise their probability of winning. The property verified for player 1 is:

```
filter(print, ⟨⟨1⟩⟩ Pmax=? [ F "p1_wins"], true)
```

The use of the filter keyword and the `true` predicate results in PRISM returning the optimal value at every reachable state of the model. By combining the optimal values at every state with the actions available and the probability of each action's success, we can generate a useful resource for play analysis, the **action lookup table**. Multiplying the probability of an action's outcome and the optimal value for the player at each state reached by performing the action and then summing these probabilities over all states reached gives the player's optimal probability of winning at the state *conditioned on* the player having chosen the action at the state. The action lookup tables list states and the probabilities associated with each available action of the state, rather than the single optimal value shown in the state lookup tables. We have generated 28 such tables, one for each character pair. The table entries for the two Knight-Archer states given above are:

```
1,8,4,0,0,0,0,0,0,0,0,0,2,6,0,0,0,0,0:
    {
        skip:   0.45563,
        K_W:    0.49406,
        K_R:    0.74543,
```

```
        A_W:    0.49406,
        A_R:    0.49671,
        A_WR:   0.71668
    }
1,8,4,0,0,0,0,0,0,0,0,0,2,8,0,0,0,0,0:
    {
        skip:   0.22326,
        K_W:    0.34959,
        K_R:    0.43816,
        A_W:    0.34959,
        A_R:    0.40916,
        A_WR:   0.46517
    }
```

The action with the highest probability associated to it is the optimal action as it provides the best chance of winning. Here we can see that in the example where the opponent's Rogue has health 6 health, the optimal action is to attack it with the Knight (K_R) rather than attack both opposing characters with the Archer (A_WR), whereas when the opponent's Rogue has 8 health the optimal action is to use the Archer (A_WR). This example demonstrates the subtleties of the game – small changes in the state lead to differing action viability.

The use of action lookup tables is motivated by the expense of calculating action-costs in isolation. When verifying for optimal probabilities the model checker performs the calculations for you. To calculate the optimal probability associated with non-optimal actions, one must manually calculate the probability from the outcome states of an action and use them to calculate the single value for the action itself. For example for an action that can either *hit* or *miss*, to get the optimal value associated with that choice of action at a state one must obtain the optimal probabilities from the states reached by either outcome, multiply each by the probability of the outcome (e.g., hitting will have a probability of 0.6 to 0.9) and then sum the results. The use of an action lookup table prevents the need for multiple costly verifications to calculate a single non-optimal action's optimal probability. The alternative would involve repeating the same calculations every time for intermediate states.

Once the action lookup table has been generated, we can calculate the *cost* of every action taken at each state. This action-cost is calculated as the difference between the optimal probability of a player having chosen the action at the state and the maximum optimal probability possible from all moves available at the state. In the example above, the cost of playing K_R when the opponent's Rogue has 8 health is $0.46517 - 0.43816 = 0.02701$. This costs demonstrates that the player has reduced their optimal probability by almost 3% by playing K_R rather than playing A_WR.

The state and action lookup tables for both configurations using the RPGLite are included in

the RPGLite dataset [72].

7.2.2 Similar Measures

Action-costs as described in this thesis are a novel concept, although there is some overlap in the terminology used which it is useful to clarify. We will briefly compare action-costs to rewards/-costs used with similar mathematical models and the notion of *regret* from decision theory.

Typical definitions accompanying Markovian chains include definitions for rewards/costs (they are synonymous in this context) which are separate values that can be defined along paths. Model checkers also accommodate rewards/costs, they can be defined in Prism models for example. Their usage allows for the description of more quantitative measures than probabilities alone, e.g.: The number of times a given transition is used or a particular state is visited.

The notion of regret from decision theory [90] is similar to our action-costs. Regret is the measure of the difference between the optimal outcome of a decision and the outcome chosen. Regret theory seeks to explain decision making based on calculations of the utility function that incorporate a decrease in line with how much the actor is likely to regret the decision they have made. Regret is calculated as the opportunity cost and is similar to the calculation made to calculate action-costs. The work on action-costs stands apart from the notion of regret in that we are focused on how the precise values can be obtained and considering multiple decisions that have been made by game players rather than identifying optimal decision making processes.

7.2.3 Further Definitions

The *relative cost* of a move is calculated by dividing the cost by the optimal probability. The relative cost of the example above is $0.02701/0.46517 = 0.05806$. Relative cost is important to consider because it contextualises the costs within the game state. A reduction in probability from 0.4 to 0.1 is more significant than from 0.8 to 0.5 since in the first instance the player has reduced their probability of winning by a factor of 4, whilst in the second the reduced value is still greater than half of what it was prior to the bad move. This significance will not be reflected in the cost (both instances have a cost of 0.3), however the relative cost does (the first instance has a relative cost of 0.375 which the second has relative cost 0.6). For this reason we will exclusively use relative cost in our analysis of RPGLite, future uses of cost refer to relative cost unless stated otherwise.

We found that a strong indicator of which player won a game was the number of high cost moves – the winners making fewer high cost moves, rather than the cumulative cost. Amongst other features, the percentage of games won by the player making fewer mistakes is shown in Table 7.1. In season 1, where a player made fewer actions with a cost over 0.1 than their opponent, they won 70% of the time, whereas when the average cost of their moves was more than 0.02 less than their opponent they won only 62% of the time. When analysing game data we consider

Feature	**S1:** true/false/unclear	**S2:** true/false/unclear
Made first move	60.26% / 39.74% / 0.0%	60.62% / 39.38% / 0.0%
Lower average cost ($\delta > 0.02$)	57.68% / 34.88% / 7.44%	54.48% / 35.38% / 10.14%
Lower total cost ($\delta > 0.1$)	46.98% / 35.46% / 17.56%	46.56% / 29.07% / 24.37%
Fewer minor mistakes made ($\delta > 0$)	48.53% / 20.78% / 30.69%	44.51% / 19.37% / 36.12%
Fewer major mistakes made ($\delta > 0$)	31.86% / 6.77% / 61.37%	30.82% / 6.55% / 62.64%

Table 7.1: Predictors of success in RPGLite games in both season 1 (S1) and season 2 (S2). Unclear refers to the percentage of games where the feature cannot significantly distinguish between players, the value considered significant is given in brackets.

these high-cost moves, and refer to them as *mistakes*. For example, we might classify a minor mistake to be a move with a cost of at least 0.1 and at most 0.25 and a major mistake to be a move with a cost of at least 0.25.

There are some game states where players are less likely to be motivated to win, the action-costs in these states should not be considered along with others. For example if a player is losing heavily and defeat is seemingly inevitable, then they are likely to be less motivated to try to win. Similarly states where players have only a single action available to them (such as when players are forced to skip by a Wizard) should be discounted as the single action will be optimal and the cost will be 0.0. Nothing can be learned by considering these actions. We refer to moves made in states where players are motivated to win as *critical actions* which we define as moves made from states where more than one action is available and the player's optimal probability of winning is greater than 0.15, an value chosen to signify a position where winning is still plausible, if unlikely. Note that *critical actions* are taken from *critical states*.

The use of critical states is an additional measure employed to ensure the data captured is that of competitively motivated players. The application RPGLite had several incentivisation systems designed purposely to encourage the desired motivation amongst players, they are described in Section 5.3.2.

Critical states include those states where there is only a single non-skip action and skip available. This is because there are some states where skipping is the optimal action even when other actions are available. The following is a snippet from the action lookup table from the season 2 configuration for a Knight-Barbarian pair where both players have only a Barbarian alive with 5 health remaining:

```
1,0,0,0,0,0,0,5,0,0,0,0,0,0,0,0,5,0,0:
    {
        skip:   0.64497,
        B_B:    0.35503
    }
```

Successfully hitting an opposing Barbarian would reduce their health to below the rage threshold at which point the opponent could win the game with a single successful action due to the

increased damage, hence the optimal action is to skip.

7.3 Player Learning

We use action-costs in several ways to measure player learning – when examined over time they should show how player skill changes. We believe this gives a more accurate measure of skill than matchmaking ranks [91]. As games are played, players will begin to see the effect of their actions (i.e., if they lead to wins or losses) which should lead to their making better decisions. Our hypothesis was that as more games were played by players their average action-costs would decrease as they made fewer mistakes and those they did make would be less significant. In other words, we anticipated players would get better at RPGLite as they played more games.

It is difficult to draw any conclusions from the action-costs of a player when considered as a whole. Action-costs for season 1 cannot be directly compared to those for season 2 as players reach different states in the two seasons, from which non-identical actions are available with different costs. Many players had higher action-costs in season 2 than in season 1, this does not necessarily show that they got worse between seasons, it could instead show that in season 2 it was more difficult to play optimally. However, using relative action-costs across both seasons does give a larger dataset for aggregate and trend-based analysis. Figure 7.2 shows the relative costs of all actions taken by a single player from every critical state reached. The player was chosen at random from those who had played a significant number of games in both seasons. There are $1,796$ actions shown in season 1 with an average cost of 0.021 and $2,363$ in season 2 with an average cost of 0.033. The data is sparse – the majority of actions taken were optimal, with a cost of 0.0. In total 73% of the player's season 1 actions and 77.8% of their season 2 actions were optimal. There is no noticeable trend in the data of all actions or from the averages taken throughout, (below) in the figure. There were more higher cost actions when the player first started playing which could show their initial learning, but this could also be down to the player's material selection. The player used a Rogue-Monk pair for the majority of season 1 after some initial exploration and continued to use them for their first 50 games of season 2, but did not use this pair afterwards. The effect of this can be seen in Figure 7.2 as the dense clustering of actions with costs in the range $(0, 0.1)$ from action 300 in season 1 up to action 300 in season 2.

Having studied all costs for a single player it is clear that action-costs must be used with more context. The actions available to a player are dependent on the game state, which itself is a factor of the material chosen by both players. We use mistakes, as described in Section 7.2.3, to counteract this. To verify the soundness of this approach we must ensure the possibility of making mistakes is similar with all materials. We compare all possible actions for all possible pairs in Table 7.2. *Critical actions* is the number of critical states reachable against any opposing pair. The second and third columns give the proportion of these critical actions which yield at

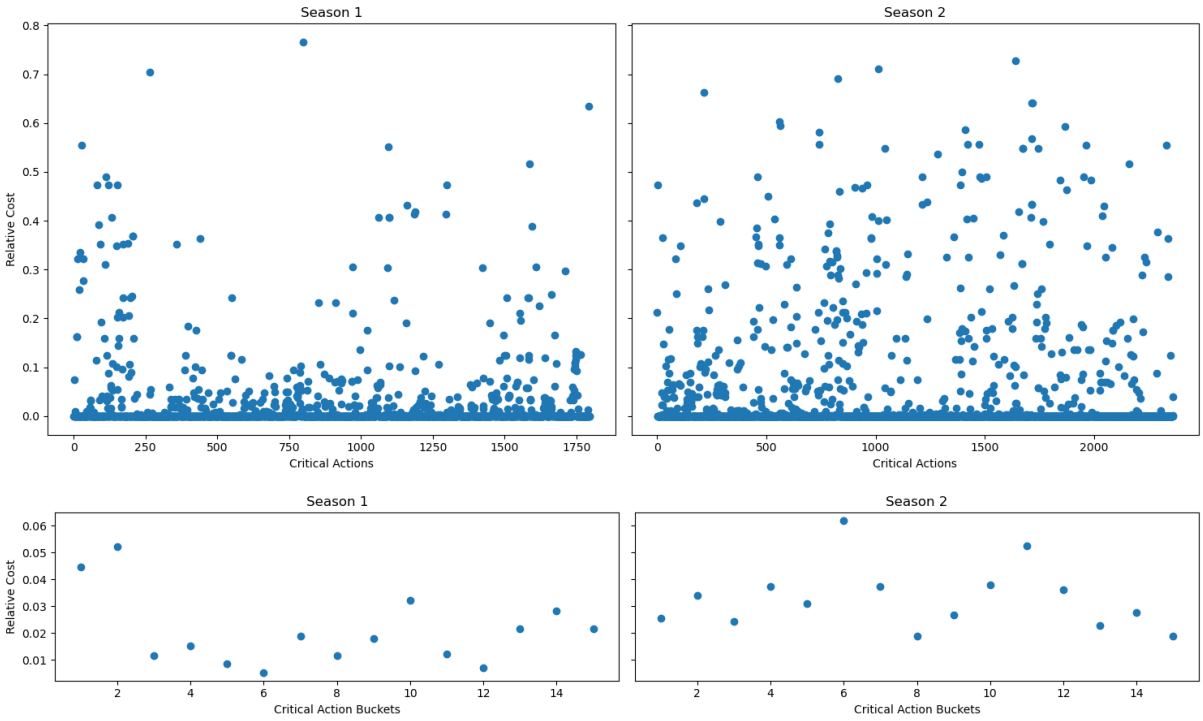

Figure 7.2: (Above) the cost of all critical actions taken by a single user sorted chronologically. (Below) the same data in 15 buckets for each season.

least a minor mistake (> 0.1) and a major mistake (> 0.25). The final column gives the average relative cost of the second best action. Across all pairs, the proportion of critical states from which a minor mistake is available ranges from 0.819 to 0.931 and for a major mistake it ranges from 0.216 to 0.413. The opportunities to make errors do not vary hugely between materials, which we believe allows for the use of mistakes at various thresholds as a unified measure across materials.

Fewer of our players exhibited a clear negative trend in average cost over time than we expected. To measure the rate of learning amongst the entire playerbase we considered the proportion of moves made in each game that were above a mistake threshold. Looking at a small number of sequential games, the rate of learning is consistent, players gradually improve and make fewer mistakes per game. As the number of games considered gets larger approaching 100, the rate of learning slows to the point where mistakes are made at the same rate game to game. What is surprising is that after roughly 150 games had been played, the rate of mistakes being made increased. Figure 7.3 shows this analysis performed considering major mistakes. The same trends of steady initial learning, flattening around 80-100 games and then players getting worse after 150 games, is apparent when considering both minor mistakes and average costs. It is important to note that the populations being considered are not the same, as shown by Figure 7.3 we had data on 87 players who had played at least 25 games, but only 19 who had played at least 200, so the data is more sensitive. The populations overlap, all players to have played 50 games are included in the data for those who played 25 for example, so results should be consistent through-

Pair	critical actions	minor available	major available	avg. cost of 2^{nd} best
KA	158648	0.849	0.291	0.08788
KW	144797	0.867	0.345	0.09070
KR	49695	0.866	0.331	0.10371
KH	215461	0.819	0.288	0.06013
KM	163431	0.904	0.244	0.04540
KB	72649	0.856	0.320	0.08824
KG	173258	0.875	0.318	0.08222
AW	62509	0.859	0.337	0.10216
AR	128341	0.882	0.318	0.09174
AH	102161	0.835	0.273	0.06156
AM	145959	0.910	0.231	0.04076
AB	151490	0.868	0.288	0.08744
AG	152862	0.886	0.315	0.07966
WR	122013	0.892	0.413	0.10211
WH	96496	0.827	0.318	0.07387
WM	135274	0.925	0.306	0.06142
WB	139470	0.883	0.349	0.09082
WG	143785	0.888	0.380	0.09320
RH	178218	0.857	0.336	0.06826
RM	134971	0.931	0.260	0.04446
RB	58971	0.877	0.319	0.06051
RG	143842	0.903	0.331	0.06989
HM	194921	0.897	0.243	0.03202
HB	203672	0.842	0.297	0.06355
HG	209583	0.856	0.333	0.06331
MB	153190	0.912	0.216	0.04019
MG	133216	0.928	0.249	0.03887
BG	164874	0.890	0.302	0.06934

Table 7.2: Material comparison for the updated configuration from season 2.

out all groups. Both seasons are included in this data, to ensure this would not compromise on the results we compared the average cost per season of all players to have played several games in both seasons. We found that players with at least 10 games in either season did better in season 2 with an average cost of 0.038 in season 2 and 0.041 in season 1, as did players who played 50 games in each season. However players who played at least 100 games in either season all did better in season 1 with an average of 0.03, 0.0008 better than in season 2. This aligns with the results in Figure 7.3, and shows that it is not because of the different costs associated with the two seasons. Whilst 0.0008 may appear small, one must consider that the average number of moves made per game was 13.813 and some players played far more than 100 games (one player reached 1004 games played).

From Figure 7.3 it appears that the second game is played less skillfully than the first for all four levels of experience and other early games appear to be played worse than expected. This

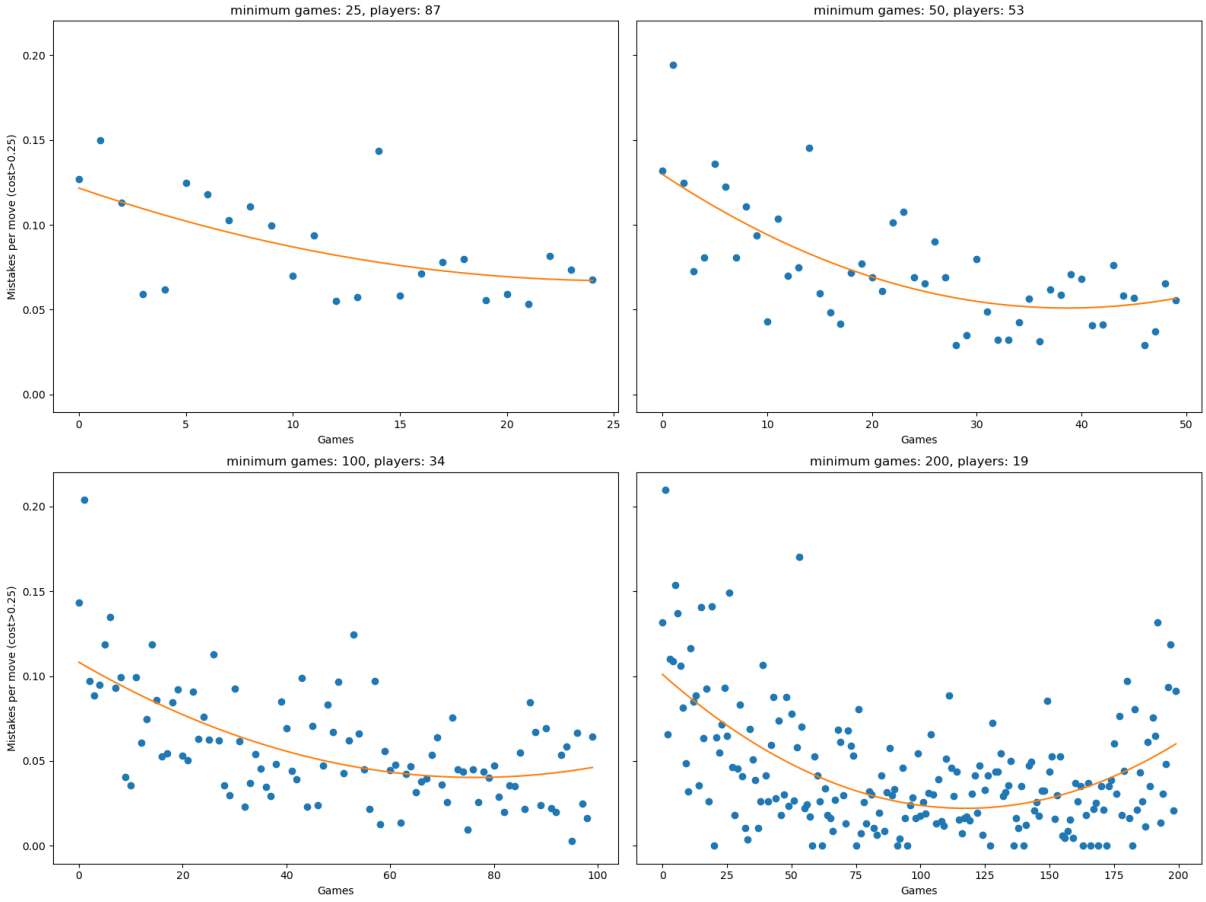

Figure 7.3: Proportion of moves which were a major mistake per game shown for the first *n* games played by all players to have played at least *n* games, with values for n of 25, 50, 100 and 200. A quadratic fit is included to indicate trends.

may be a symptom of the way characters in RPGLite are unlocked. New players can only use the Knight, Archer, Rogue and Healer. After having finished at least one game with all of them they unlock the Wizard, completing a game with the Wizard then unlocks the Barbarian. The Monk and Gunner are unlocked in the same way. It takes a minimum of five games to unlock all of the characters and it was a conscious decision that the characters unlocked at the start of the game were the most simple to understand, with more complex characters coming later. Players are also likely to explore the characters in early games and then settle on preferred materials in later games. Of the 53 players who played over 50 games only 21 used every character more than 5 times.

The increase in the rate of major errors being made by experienced players was not expected. Instead we expected learning to flatten out when players had *solved* the game and then errors would only be the result of carelessness, but the results clearly show a decline. One possible reason for this is players growing apathetic with the simple nature of the game and exploring once again. This does rather contradict anecdotal evidence – many players claimed they were highly motivated by skill-points and the leaderboard – those who played the most were the ones competing for the top positions. Despite this decline in decision making, the average rate of major errors decreased in the final population of players when compared to those who played 100 games, as it did every time the number of games considered increased.

The most precise method for appraising player learning is to consider actions made in states that the player has visited before. Unlike our other approaches to studying player learning, this approach does depend on material or game state. However, a drawback is that many repeated states are likely to be those at the beginning of a game, from which the optimal action is easier to identify. A simple heuristic for choosing which action to take is to choose the action from which more value is *expected* to be gained in terms of damage done to an opponent or undone to the player's own characters. The expected value is calculated by multiplying the beneficial change (opponents damaged and allies healed) in a reachable state by the probability of reaching that state and summing the values over those states. Consider the first move in a Knight-Archer v Knight-Archer game in season 2: the Knight does damage 3 to one target and can *hit* or *miss*, whilst the Archer does damage 2 to two characters and can *hit-both*, *hit-one*, *hit-other* or *miss*. Both characters have an accuracy of 80%. The optimal action is to use the Archer as you can *expect* to deal damage with a value:

$$(0.8 \cdot 0.8 \cdot 4) + (0.8 \cdot (1 - 0.8) \cdot 2) + ((1 - 0.8) \cdot 0.8 \cdot 2) + ((1 - 0.8) \cdot (1 - 0.8) \cdot 0) = 3.2$$

whereas with the Knight you can only *expect* the damage to be:

$$(0.8 \cdot 3) + (1 - 0.8 \cdot 0) = 2.4\,.$$

Having chosen to use the Archer the player's optimal value is 0.6464, i.e., if the player and its

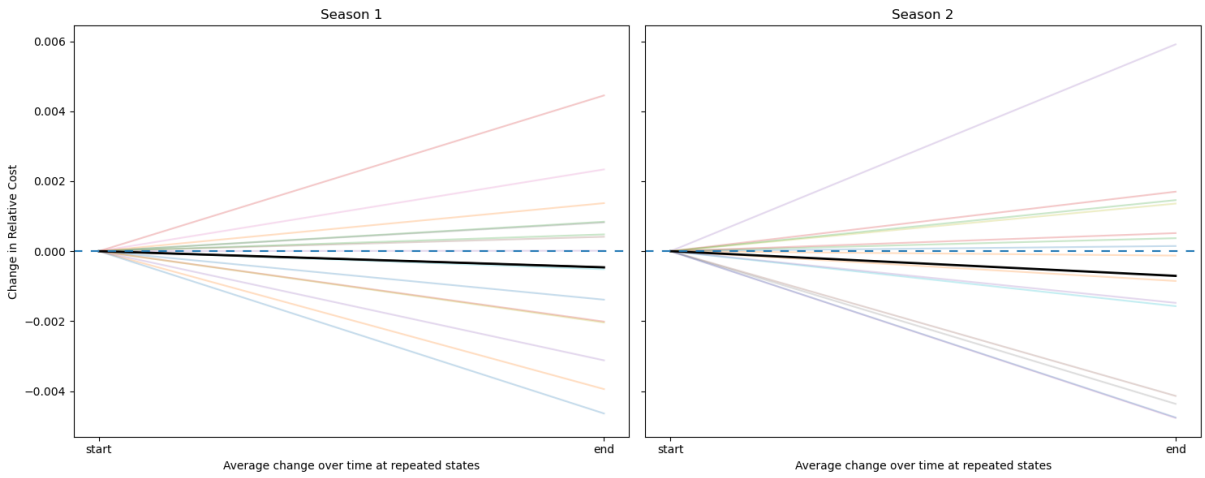

Figure 7.4: The average change in cost for the top 15 players when considering only states visited multiple times. Bold lines represent the average in either season.

opponent play perfectly from there on their probability of winning is 0.6464. If the player had instead chosen the Knight and the game was played perfectly from there on, then the probability would be 0.57446 or 0.5488 depending on whether the player had targeted the opposing Archer or Knight respectively. Now consider the same match up where all characters have only health 3 remaining, the value you can *expect* from your actions remains the same, but you could reduce an opposing character's health to 0 preventing their actions from being used – how much is this worth? Using action-costs the additional value of reducing an opponent character to health 0, thereby preventing them from acting, can be calculated. In this state choosing "Archer attacks Knight, Archer" leads to an optimal value of 0.6494, "Knight attacks Archer" leads to 0.27649, but the optimal action of "Knight attacks Knight" leads to an optimal value of 0.74719. This decision is more complex due to the additional nuance of dealing with low-health characters.

To determine if players improved we calculated the average change from every state that they visited more than 3 times. The results of this calculation being performed on the top 15 players in either season is shown in Figure 7.4. The average change in season 1 is −0.00046 and in season 2 is −0.00071. The values are small because in the vast majority of states players played optimally every time, so there was no change. The results indicate that the top players got slightly better over time, on average, but there are several instances of players getting worse. Calculated for the top 75 players in either season, the averages are 0.00497 in season 1 (they got worse on average) and −0.00395 in season 2. The states from which optimal actions are less apparent occur later in games and these states were visited less often. The calculations of change from the states visited multiple times ignore states visited fewer than 4 times, which likely omitted many of the more difficult decisions.

A notable finding from the costs in repeated states was the extent of player obstinacy. Many players made an error the first time they visited a state and then repeated the error on every future visit. Of the 20 players who played the most games, 18 repeated errors they made when visiting

states for the first time in subsequent visits more often than choosing another action. One player visited 159 states multiple times in which their initial move was a mistake, from 139 of those states they made the same incorrect move every time.

Recall, a state is considered as having been visited *multiple times* if it was visited over 3 times. There are a limited number of states that players visit multiple times. Many of these are inconsequential in that they are solved at the player's first visit. The states which are typically more complex are rarely seen. Too few games were played for us to achieve very accurate results by just considering states visited multiple times. We therefore broadened the criteria from states that a player has seen multiple times to states where they are using material that they choose frequently. We expanded on this too, to consider states in matchups (the combination of characters used by both the player and their opponent) that they experienced repeatedly.

Many players only used a limited subset of the available material, sometimes choosing to play with the same pair of characters every time. We considered the change in their average cost over time within those material choices, suspecting that players who restrict themselves to only some of the pairs will get better at using those pairs, selecting lower cost actions. To study pair-wise learning we consider each season separately, disregarding players who played fewer than 10 games, and find the average trend in mean cost per game using each pair played at least 5 times. Our results are shown in Figure 7.5 (top). In both seasons the numbers of players who improved and players who got worse are similar, 38 out of 70 in season 1 and 30 out of 55 in season 2. The average across all players when weighted by games played in both seasons is between 0 and -0.00001 which we consider to be insignificant.

Figure 7.5 (bottom) shows results for analysis considering matchups players experienced at least 3 times in a season. The results are similar to the pair-wise analysis, 33 out of 59 players improved in season 1 and 27 out of 42 improved in season 2. The average learning restricted to matchups when weighted by games played is -0.00124 in season 1 and -0.00104 in season 2. Players did get slightly better with experience on average. However, several players got worse in both seasons when considering matchups.

Another indicator we looked at for the rate of learning was the point in a sequence of similar games at which players played worst. Following on from the pair-wise and matchup-wise analysis, we looked at every player with over 20 games in a season and calculated the average position in a sequence of at least 10 games played by the same pair or at least 5 featuring the same matchup in which the player made the highest average cost per move. We expected that the players' worst game would come early in these sequences. Of 137 sequences of games played by a player using the same pair in season 1, the average sequence was 31 games long, the worst game coming on average at the 14^{th} position. Of 115 sequences in season 2 the average length was 9 and the worst game was the 4^{th}. Similarly for sequences of matchups, of 196 in season 1, the average length was 39 and worst game was the 19^{th} and of 241 in season 2, the average length was 12 and the worst was the 5^{th}. All of these worst games came at a point between 40%

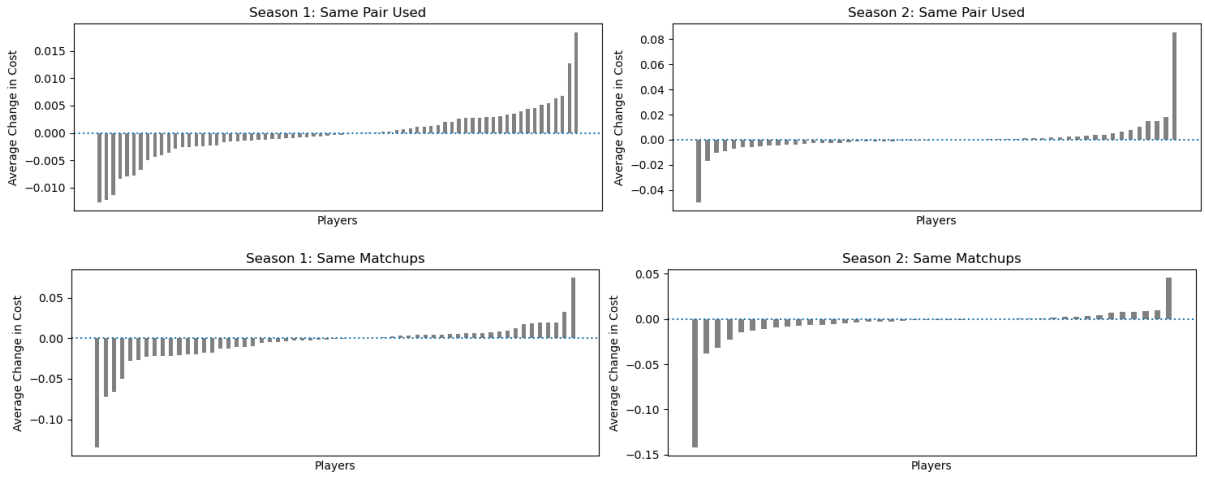

Figure 7.5: Learning within pairs played and matchups experienced. Values below 0.0 denote a player that got better with experience, values above denote a player that got worse. Individual results have been sorted into ascending order.

and 50% of the length of the sequence. If significant learning was taking place one would expect the worst game to come much earlier in the sequence of similar games.

Having used costs in the analysis of player learning we can show that over half of all RPGLite players did improve over time, but several did not. We had presumed that owing to RPGLite's simplicity some players would essentially *solve* the game leading to them incurring minimal costs on actions across several games, which was not the case. Costs are a good measure for analysing player skill. However in order to track learning the various actions and states available have to be considered. Costs can also show surprising patterns in player learning, as illustrated by the eventual drop-off in success we identified amongst keen players.

7.4 Material Comparison

In this section we use our results to compare the characters in RPGLite.

7.4.1 Expanded Balance Matrices

When designing RPGLite's characters we wanted to ensure each of them had unique characteristics. The Knight is the basic framework that the others are built upon, and is intended to be simple. The Gunner and Healer are similar in that they are consistent throughout a game, but the Gunner does bonus damage when they miss and the Healer heals allies with successful attacks. The Wizard, Rogue and Barbarian all get stronger as the game proceeds: the Wizard can repeatedly stun a single target stopping a player from acting if they already have a character at health 0 and the others get more effective at lower health values. The Monk and Archer are often better early on when a character cannot be reduced to health 0 in a single hit by other characters with

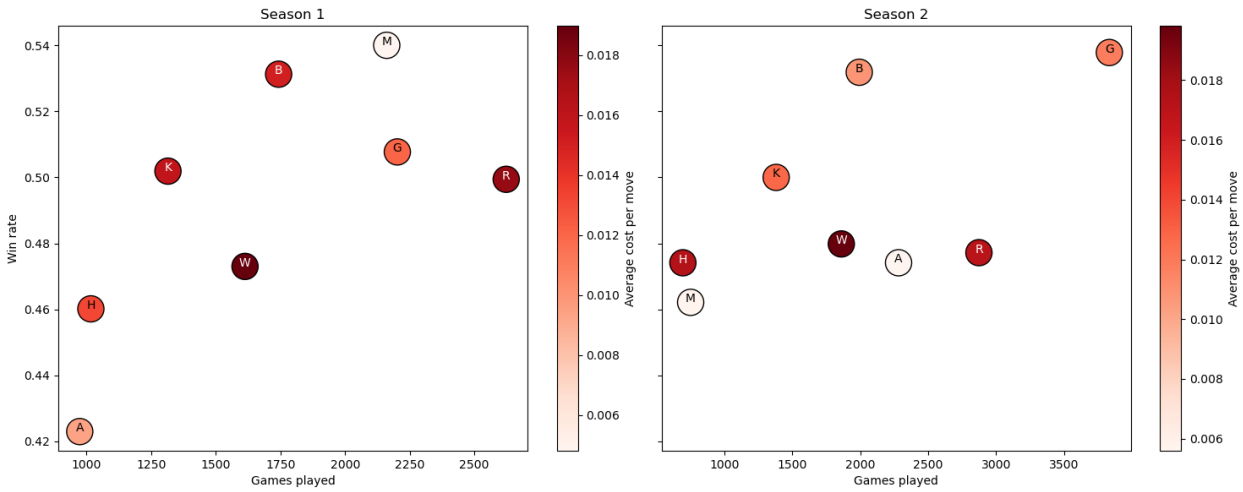

Figure 7.6: Balance matrices for RPGLite enhanced with cost axis.

higher damage attributes. Some characters combine especially well. The Rogue-Monk pair is particularly effective because the Monk can reduce characters to 5 (or fewer) health, setting the Rogue up to *execute* them, all in a single turn. This combination was identified by many players in Season 1, being played 1,046 times compared to the second most popular pair Wizard-Gunner which was played 624 times. A more thorough discussion of high-level Rogue-Monk strategies is included in Section 6.4.1.

Gameplay analysis for game balance often consists of pick-rate, win-rate calculations in the form of balance matrices, which consider lone material units. Using action-costs we can improve on this approach to get more information on how the characters are played and suggest which are confusing to players. In Figure 7.6 we present a balance matrix that also illustrates how well the characters were played, these are an enhancement on the same figures presented for Section 6.4.1. In season 1 for example the Wizard was the worst played character (shown by having the darkest red tone), whilst the Monk was the best played (the lightest tone). This additional information can be used by developers to make better judgements on the state of game balance. The Archer in season 1 was the least successful and least popular character, shown by its position further left and lower. One possible reason for this could be that players were not good at using the Archer effectively. However knowing that the average cost of Archer actions was among the lowest of all characters suggests this is not the case. For season 2 we improved the Archer's attributes (increasing their health from 8 to 9 and slightly reducing their accuracy from 85 to 80) on the basis that it was not the fault of the players that the Archer was under-performing. We made a similar decision in reverse for the Monk, observing that players were making very few errors whilst winning consistently. These characters saw the most significant shift in popularity and success between seasons, even though five others were also modified. We assumed that the Wizard would see a reduction in average cost as players who had played in the first season learned how to use it effectively. However, it remained constant between seasons, as did the Wizard's

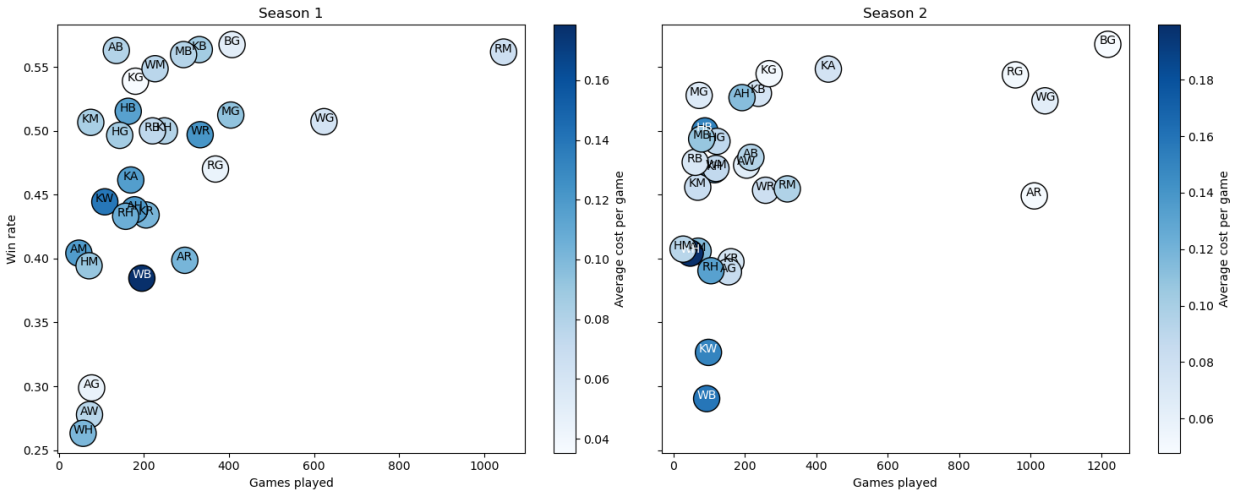

Figure 7.7: Pair popularity, usage and complexity in both seasons. Names are abbreviated so KA refers to a Knight-Archer pair. Obscured in season 2 is the pair with the highest average cost, Wizard-Healer.

pick and win rates.

Developers of games where players or teams use sets of material rather than individual units often cannot easily weigh up the pros and cons of each of those sets because there are too many of them. RPGLite was designed with this analysis in mind, having a set size of only 2 – the pairs that players select. This is small enough to allow for pairwise analysis Figure 7.7, in addition to the singular material units. The analysis of the full sets players use is a better source of data on the state of the game than analysis of individual units. However, those sets cannot be changed directly to try and improve the game. Instead, only the individual units can be adjusted, which affects all sets in which they are included. This is a significant factor in the difficulty of balancing games where material sets are used.

The conclusion which can be derived from RPGLite data relating to the pairs being played is different to that relating to characters alone. Whereas with characters we can calculate the average cost of every move made using those characters, we can not do the same for pairs as actions have a single acting character. Instead the average total cost of moves made in a single game by a player with each pair is used. For example, players using the Wizard-Barbarian pair in season 1 made moves that reduced their optimal probability of winning by 18% in total, on average.

There are some notable observations from the cost data in relation to the popularity and success of RPGLite material. There is no clear correlation between the costs of using given material and its success, although it appears that the most played pairs have a lower average cost. In season 1 the Archer-Gunner pair was played well, as shown by the very low average cost per game, but it was among the least successful with a very low win rate of 30%. This is a clear indicator that the AG pair in season 1 is not an effective one, but is not enough evidence

on its own to suggest changing either character. After all, the Archer-Barbarian and Barbarian-Gunner pairs are among the most successful with success rates over 55%. These discrepancies in the viability of sets made up of similar material units is a desirable feature of games as it suggests that some materials complement others, enhancing their viability. This implies that a deeper system of relationships between the materials exists and is one that players will need to comprehend to be successful. Without costs we could not be as confident in our assessment of the strengths of the various material in RPGLite.

7.5 Identifying Common Mistakes

Being able to find the states where players make mistakes helps us to understand how players interpret the game and where design is not as intuitive as it could be. It is also interesting to see the positions from which players frequently fail to play optimally.

The 'skip' feature of RPGLite was implemented to allow the model to navigate from states where one player had a single stunned character alive. In pre-release testing feedback we were asked why skipping was not automated as there was no situation where a player would want to skip if they could take a character action. This is false, there are states from which skipping is preferable to character actions, as discussed in Section 7.2.3. In season 1, the two states visited at least 5 times with the highest average cost per move were when only the Barbarian was alive for either player, one with health 10 and the other with 7. Because the Barbarian does damage 5 when at health 4 or less and 3 otherwise, if you cannot win in 2 successful actions when above health 4 then it is preferable to wait until your opponent has hit you so that you can achieve this. When at health 7 with an opponent at health 10 the optimal probability of winning if the player chooses skip is 0.64624 and is 0.39296 if the player chooses attack. In the reversed state the values are 0.60704 if the player chooses to skip and 0.37376 if they attack. No players skipped in these states even though they were visited 64 times. This phenomenon, where any progress is to the progressing players' detriment, is known as Zugzwang [60]."

A more detailed example from season 1 that shows the complexity of calculating optimal actions when considering the Barbarian's *rage damage* is shown in 6 consecutive states detailed in Table 7.3. Here a Rogue-Monk player is playing against a Barbarian-Monk player and as the health of the opponent's Monk decreases, the optimal action changes both in terms of the acting character and the target. The state where the opponent's Monk has health 10 was visited 11 times in season 1 and the average cost of all of those moves was 0.14213, the third highest of any state visited more than 5 times.

Action-costs trivialise the identification of poor decision making from the players. Here we have also used them to highlight interesting optimal actions by looking at states at which players slipped up often. This gives a better understanding of the subtleties of the game itself.

p1R	p1M	p2B	p2M	p1Rp2M	p1Rp2B	p1Mp2M	p1Mp2B	p1_skip
8	3	5	10	0.36852	0.39352	0.52907	0.59575	0.26209
8	3	5	9	0.39041	0.4198	0.59999	0.67917	0.28295
8	3	5	8	0.77822	0.6146	0.72196	0.68947	0.49653
8	3	5	7	0.68002	0.53595	0.67947	0.73771	0.43733
8	3	5	6	0.69415	0.56233	0.74548	0.8128	0.46542
8	3	5	5	0.89965	0.76788	0.85306	0.79683	0.65396

Table 7.3: Moves from selected states in season 1 in an RM-BM matchup. Blue cells represent the optimal action to take at each state.

7.6 Uses Beyond Analysis

7.6.1 Cost as a Ranking System

RPGLite has its own ranking system in the form of skill-points. Skill-points were designed to favour players who played more games – far more points are gained by winning (35-45) than are lost by losing (5-15) and it is not possible to fall below multiples of 100 unless repeated games are forfeited. This system was not designed to be an accurate measure of player skill, rather an incentive for players to play more games to climb up the leaderboard. When developing RPGLite we included logic to keep a record of the Elo ranking of all players, but we did not make this value visible to players and intended to use it solely for analysis. To our surprise Elo proved to be a poor indicator of success. This is likely to be because the skill ceiling is far lower in RPGLite than it is in Chess, the game for which Elo was devised and due to the randomness of RPGLite (as opposed to Chess which has no randomness). In fact Elo proved to be less effective than even the skill-points ranking when compared to player win ratios.

To create a better system for ranking players we implemented a simple procedure in which we calculated the average cost that each player's actions deviated from the mean cost of all players at that state. This compensates for the various costs available at different states and the use of different materials. We call this *cost deviance per move* and would expect a player with a lower cost deviance per move to be a higher-skilled player than one with a higher cost deviance. Clearly the average cost deviance per move is 0.0, denoting a player who made an average cost action at every state. For simplicity, any player with a negative cost deviance can be considered *good* – they have understood the game better than most – and any player with a positive cost deviance can be considered *bad*.

When comparing the three ranking systems and their relation to a player's win ratio, as in Figure 7.8, the comparative effectiveness of cost as a ranking system is clear to see. The R^2 values of the three are 0.043 for skill-points, 0.009 for Elo and 0.112 for action-costs. Elo seems to have minimal relevance as a predictor of player success, whereas skill-points appears to be a good predictor, although that may be confounded by experienced players typically having higher skill-points. The strongest correlation between predictor and results however is action-costs, as shown

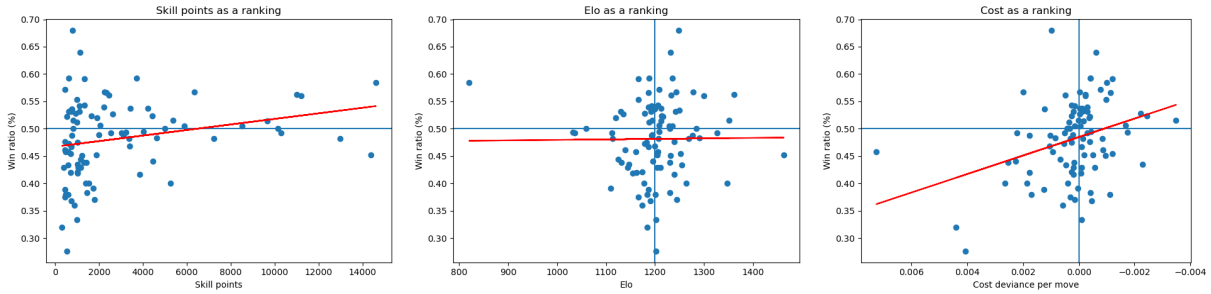

Figure 7.8: Comparison of three ranking systems against the win ratios of all players with over 20 games played. Linear fit indicates correlation.

by the gradient on the third chart. However, the R^2 values are all low suggesting the correlation

7.6.2 Cost as a Teaching Tool

Games increasingly allow players to analyse their own performance after having played. This takes several forms, detailed statistics shown on profile pages, video replays of games played and League of Legends gives letter-grades for player performance [92]. These systems incentivise and support high-level competitive play, a motivation that is regularly desired by game developers. Automated analysis that can grade player performance is expensive and often the calculations are kept secret to prevent players from gaming the system. With the costs from RPGLite it would have been very easy to implement a similar system.

As RPGLite was designed in part to study player behaviour and learning, we did not want to condition our players or steer them in any particular direction. Were this not the case RPGLite games could have ended with an analysis screen showing players all moves made by them and their opponent, detailing the cost of each move and, where the optimal move was not played, what action they should have taken. As professional gaming continues to rise in prominence, the demand for automated analysis will increase [93]. Techniques similar to calculating and sharing action-costs are a good way of providing these tools to players, showing where mistakes were made and what improvements could be made.

7.7 Discussion

Action-costs can be used to track moment-to-moment decision making in fine detail for all RPGLite players and opens up many avenues for studying how the game is played. They show us that despite the game's lack of complexity there were no players who played perfectly. Our research on action-costs in the areas of game development and gameplay analysis is early work. These ideas could be expanded upon to learn more about how games are played and how better games can be developed.

7.7.1 Limitations

The findings from this chapter are based on logged data from RPGLite alone. The identity of the players was kept anonymous which precluded any player profiling to separate those with prior experience of similar games. This experience would have likely affected initial skill and the rate of learning. Owing to how the game was advertised it is likely that the majority of the players were science and engineering postgraduate research students or had signed up to games research newsletters. This may have affected the way the game was played.

It is difficult to calculate the optimal values which inform action-costs and a series of compromises were made in the design of RPGLite to ease this. A comment we received in the early testing of the mobile application was "*having a 2 card deck makes it too heavily weighted towards luck over strategical use of your troop's attacks/abilities.*" This comment is valid in that roll results and winning the coin-flip to go first are some of the most important factors for success in RPGLite, arguably more so than it would be in an equivalent, professionally-developed game. There are 7461 games which finished normally stored in the database, of those 4495 (60.247% (3dp) were won by the player moving first, a large number but not overwhelming odds for the player acting second.

RPGLite is an expansion on a case-study with only 3 characters. Ideally the teams would be triples, not pairs, but this lead to too great an increase in the processing required. For a candidate configuration of RPGLite such as the two used in the mobile app, we can calculate action-costs from all reachable states in around fifteen minutes using a standard desktop PC with 16GB of RAM. The character actions were also limited, common effects from other games such as damage on subsequent rounds or temporarily affecting the attributes of targets (e.g., lowering accuracy in subsequent rounds) were discounted to limit state explosion (an increase in the size of tuples describing states and a subsequent exponential blow-up in the number of possible states). The health values of characters were kept low, although we believe that increasing them would have had little impact on the analysis. Decisions tend to be more impactful later in games so an increase in health values would have resulted in longer games, possibly reducing the size of the dataset we were able to collect. A more simple game makes the analysis of player behaviour more straightforward as we can better understand the mistakes made and misconceptions of the players. This likely differs from the design philosophy of games not created for research purposes.

Action-costs in asymmetric games such as RPGLite must be contextualised. Game material which significantly differs in what it allows the player to do is a design aim of many games, but this leads to differing actions and thereby differing action-costs being available to the players. Comparison between actions made using different materials is not always indicative of player skill. We have looked at ways of minimising the effect of this through limiting the compared actions to identical states or to similar material and through the use of mistake thresholds.

The use of action-costs as a measure for player skill is predicated on players being motivated to win. We are not professional game developers nor psychologists and only implemented mea-

sures which we believed would promote competitive play as well as explicitly describing the purpose of the application and how playing to win was required in greeting messages on the app itself. Some of our results suggested that players began to deviate from this behaviour after a large number of games as their action-costs increased. There is no reason to suggest that this lazy quality to play would not be common in larger games.

7.7.2 Feasibility at Scale

It would be possible for action-costs to be calculated for larger and more complex games, however a more realistic objective for the immediate future may be for alternative values to be used rather than the optimal values, which require automated analysis of the entire state space. For example, historical data of a player or team's success from a given position could be used to estimate their probability of winning from that position. These values could be used in a similar way to how we use optimal values in our action-cost calculations. Another method for tackling the state space of more complex games would be to use abstraction [94], [95] where some precision would be traded for more simple state descriptions, or the exploitation of symmetry to avoid searching redundant parts of the state-space, a technique commonly used in model checking [96], [97]. Alternatively only specific scenarios could be considered to gain insight into problem areas of a game's design, limiting the amount of computation required.

Action-cost analysis is particularly suited to simple turn-based games, which have long been popular. There is already a significant amount of research into turn-based board games and card games which can support further work in the area. The relative success of RPGLite in terms of the number of players and games recorded shows the viability of games which can be analysed in this way as games which players *want* to play. As automated processing capabilities increase, the limit on the complexity of games which are enjoyable for humans to play remains constant.

Chapter 8

Conclusions

8.1 Answering the Research Question

The question that we aimed to address with the work contained in this thesis was "How can model checking be used in the game development process?" Having considered all stages of development, we have focused in particular on applications of model checking for game balancing. Game balancing is crucial to a game's success and current methods involve laborious manual trial-and-error. The motivation for automated alternative methods is clear. In this thesis we have demonstrated multiple uses of probabilistic model checking to configure and analyse balanced game systems.

To assess game balance properly, the way that the game is played must be studied. Doing this manually is problematic. One option is for organised play-testing which is expensive and time-consuming and likely to be beyond the means of many independent game developers. The alternative is to release the game and balance it after launch, which can deter players who experience a poorly balanced game. A third method, presented in this thesis, is to replicate *informed* play through strategy synthesis. We have shown that this provides a good approximation of the way that humans play and can therefore be an analogue for costly manual testing.

For synthesised play to be used by game developers to balance games, it must be analysed. We have shown multiple forms of such analysis and how it can be used to evaluate the state of game balance. One of our analysis techniques involves the use of pairwise optimality matrices – an idea which expands upon basic optimality matrices already used in the game industry, but for material pairs. We have also developed completely novel approaches, such as *chained strategy generation* and new measures of material robustness.

8.1.1 Additional Outcomes

Beyond the synthesised play presented in Chapter 4 and Chapter 6, we have presented further findings and outcomes in this thesis. The most significant is that of action-costs, described in

Chapter 7. By replicating human-like play, objective measures were created by which all players could be judged. This can be compared alongside other player features to discern player aptitude and learning which can in turn bring to light larger issues within the game. For example, with action-costs we identified characters in RPGLite that were being played sub-optimally suggesting non-intuitive design or that poor explanations of game mechanics had been provided.

With RPGLite we have incorporated model checking into the game development process showing the benefits of the process. The cataloguing of the game's development, and its relative success as an engaging experience, suggests the utility of our approach. Our description of RPGLite development can guide future research in game design. Additionally, the RPGLite database created as part of this work was designed to be useful for future work, both related to our research question as well as to wider game balancing efforts.

Finally, in this thesis we have presented the modelling of two-player turn-based games across three probabilistic models based on different models of understanding of player strategies: DTMCs when both strategies are known, MDPs when one player's strategy is unknown and SMGs when both player's strategies are unknown. Separately this type of modelling for games has been done before, but the use of several forms simultaneously and interchangeably is first introduced in this work.

8.2 Limitations

Our examples used in this thesis have focused on turn-based games with a controlled state space. This is not representative of all games that require balancing. The model checking of games has been shown to be possible, as has its effectiveness. Modelling expanded turn-based games is possible and will increasingly become so as hardware and modelling capabilities improve. Model checkers supporting more complex game features, such as real-time or concurrent systems, already exist. Indeed, recent release of PRISM-Games [18] support these systems. Considering games which can be represented by these models is an obvious evolution of the work carried out in this thesis. However, concurrent games require more complex strategies for optimality rather than the deterministic strategies considered in this work.

The techniques in this thesis were applied only to variations of RPGLite. Early forms of the case study were not sufficiently detailed to constitute an engaging game, due to their lack of depth. In releasing RPGLite, the mobile game, we have shown that our techniques can be applied to games which are repeatedly played because they are fun. Assymmetric game material is ubiquitous in games owing to the replayability it leads to. This suggests that our model checking approach could be generalised to many other games.

We have discussed player motivations throughout this thesis. A broad field of games research, there are many motivational models describing why and how games are played. We have considered balancing for only competitively motivated players, where the goal of game balancing is to

maximise fun. Balancing for this group alone is not a complete solution to the game balancing problem. Furthermore, whilst efforts were made to create a game which motivated players in the *correct* way (i.e., playing to win) and to only consider data from motivated players (in the use of critical actions), we cannot be certain that this is what motivated every action recorded.

The experiment performed on RPGLite would have been greater validated through the gathering of subjective player feedback such as perceptions of character balance. This was not possible due to General Data Protections Regulation. The limited feedback to which we do have access is compromised as it came from participants who were mostly friends or colleagues of the researchers. We cannot know that players found RPGLite fun and therefore balanced, but it was downloaded and played repeatedly by several players who were not incentivised to play for any reason other than for engagement and the notion of contributing to scientific work.

The major limitation on the application of model checking to game development is state-space explosion, this is arguably the major limitation on model checking in general. For the vast majority of recently released games, it would not be possible to model check them in their entirety, and even abstracting them to a point where they could be model checked within reasonable time and computational capacities would be difficult. However, that does not diminish the impact of this work. The approaches outlined are initial investigations into the use of model checking in this way and for this purpose. More tailored approaches to specific games or genres may be able to tackle the state space in ways specific to their context. Furthermore, there is a lot that can be learned of the games that sit in the intersection of 'interesting to play' and 'constrained to model check', like the data showing players getting worse after 200 games of RPGLite, for example.

8.3 Avenues for Future Work

There are several possible avenues for future work. The clearest would be to expand the study of model checking techniques being interwoven in game development but using a major game. The research question would be whether model checking early in development could reduce the workload required for game development overall by reducing the need for extensive player testing. Another question arising from the work in this thesis is whether game balancing can be fully automated, with the methods outlined in this work as a component of a larger system. Games are balanced sufficiently if their players believe them to be so. Any implementation of a fully automated system for game balancing must then be able to identify games which would be considered *fun* and *interesting*. The work presented in this thesis will likely not lead to systems capable of balancing games with no human intervention, it is more feasible for it to be extended to create services that can complement the work of skilled game designers. If automated systems such as those presented in this thesis were developed for large commercial titles, they should be usable to quickly discount poorly balanced configurations and to suggest candidates for further consideration. In this thesis we have looked at both material selection and game playing as

areas of game balance, considering the union of the two as a 'way of playing'. Approaches similar to those in this thesis can be used for the analysis of solely material selection given some suitable alternative to action-costs can be used to inform comparisons between material. These alternatives could come from empirical data, although this would not be possible before a game is released to a mass market.

Expanding on the case studies in this thesis could lead to more expansive balancing systems. The work presented in Chapter 4 introducing quantitative measures of balance, if further developed, could lead to classification of more or less balanced configurations of games. If this can be achieved then configuration selection can be automated to identify the single most balanced configuration. This is difficult for reasons discussed before of limited predictability of the effects of configuration changes on relationships between the ways of playing a game and the scale of the configuration space. However, if configurations can be directly compared in terms of *how balanced* they are, then balancing can be performed as a single optimisation task.

In Chapter 2 we discussed alternative applications for model checking in the game development process, specifically bug-checking. In addition to this, there are other potential uses for probabilistic model checking which warrant further investigation. Rates of progression through a game can be measured with model checking estimates for the rate of experience points or currency gained. Level-based progression systems such as those seen in mobile puzzle games use varying difficulty levels to maximise in-app purchases and ensure steady progression. Model checking can be used to support this. Finally, strategy synthesis can identify effective ways of playing that humans may not have considered, as shown by the strategies identified for RPGLite. This could have benefits for professional game players and testers in identifying new strategies for game playing. This thesis has focused on the game balancing problem, but applications of model checking for game development are by no means limited to just this area.

Other game balancing areas can be further examined with model checking, beyond what was presented in this thesis. In Chapter 3 where we described key ludological terminology, several of these areas for research were introduced. Considering the forms of material selection in games could be achieved with probabilistic model checking, without the need for detailed representation of the game itself. Instead the relative success of materials units with respect to each other can be used as an abstraction for fully described play. Additionally, work on proportionately discounting orthogonal design and first-move bias can be supported by probabilistic model checking. First by measuring the extent to which these features benefit a player, and then by ensuring the effect is not significant enough to diminish the enjoyment of the game itself.

8.4 In Summary

This thesis has shown that probabilistic model checking can be incorporated successfully in the game development process. Whilst there are many potential areas of overlap between what ap-

pears to be two distinct areas, we have shown that automated game balancing can be supported by probabilistic model checking. In this work we have demonstrated how the use of rigorous formal methods can make simple, multiplayer games more fun to play. It is our hope that this work is continued to the benefit of future game players.

Bibliography

[1] A. Stiegler, C. Messerschmidt, J. Maucher, and K. Dahal, "Hearthstone deck-construction with a utility system," in *2016 10th International Conference on Software, Knowledge, Information Management Applications (SKIMA)*, 2016, pp. 21–28. DOI: 10.1109/SKIMA.2016.7916192.

[2] J. Schell, *The Art of Game Design: A book of lenses*. CRC press, 2008.

[3] A. Rollings and E. Adams, *Andrew Rollings and Ernest Adams on game design*. New Riders, 2003.

[4] D. Parker, *Probabilistic model checking - 1: Introduction*, 2011. [Online]. Available: https://www.prismmodelchecker.org/lectures/pmc/01-intro.pdf.

[5] M. Kwiatkowska, G. Norman, and D. Parker, "Prism: Probabilistic symbolic model checker," in *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, Springer, 2002, pp. 200–204.

[6] R. Jhala and R. Majumdar, "Software model checking," *ACM Computing Surveys (CSUR)*, vol. 41, no. 4, pp. 1–54, 2009.

[7] C. Lewis, J. Whitehead, and N. Wardrip-Fruin, "What went wrong: A taxonomy of video game bugs," in *Proceedings of the Fifth International Conference on the Foundations of Digital Games*, ser. FDG '10, New York, NY, USA: Association for Computing Machinery, 2010, pp. 108–115, ISBN: 9781605589374. [Online]. Available: https://doi.org/10.1145/1822348.1822363.

[8] S. Varvaressos, K. Lavoie, S. Gaboury, and S. Hallé, "Automated bug finding in video games: A case study for runtime monitoring," *Comput. Entertain.*, vol. 15, no. 1, Mar. 2017. DOI: 10.1145/2700529. [Online]. Available: https://doi.org/10.1145/2700529.

[9] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, "Nusmv: A new symbolic model checker," *STTT*, vol. 2, pp. 410–425, Mar. 2000. DOI: 10.1007/s100090050046.

[10] M. Bernardo *et al.*, *Formal Methods for the Design of Real-Time Systems: International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004. Revised Lectures*. Springer Science & Business Media, 2004, vol. 3185.

[11] J. Lahtinen, J. Valkonen, K. Björkman, J. Frits, I. Niemelä, and K. Heljanko, "Model checking of safety-critical software in the nuclear engineering domain," *Reliability Engineering and System Safety*, vol. 105, pp. 104–113, 2012, ESREL 2010, ISSN: 0951-8320. DOI: https://doi.org/10.1016/j.ress.2012.03.021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0951832012000555.

[12] Y. Kim, M. Kim, and T.-H. Kim, "Statistical model checking for safety critical hybrid systems: An empirical evaluation," in *Hardware and Software: Verification and Testing*, A. Biere, A. Nahir, and T. Vos, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 162–177, ISBN: 978-3-642-39611-3.

[13] R. Alur, C. Courcoubetis, and T. A. Henzinger, "Computing accumulated delays in real-time systems," *Formal Methods in System Design*, vol. 11, no. 2, pp. 137–155, 1997.

[14] C. Hensel, S. Junges, J.-P. Katoen, T. Quatmann, and M. Volk, "The probabilistic model checker storm," *arXiv preprint arXiv:2002.07080*, 2020.

[15] R. Calinescu, K. Johnson, and C. Paterson, "Efficient parametric model checking using domain-specific modelling patterns," in *2018 IEEE/ACM 40th International Conference on Software Engineering: New Ideas and Emerging Technologies Results (ICSE-NIER)*, Los Alamitos, CA, USA: IEEE Computer Society, Jun. 2018, pp. 61–64. [Online]. Available: https://doi.ieeecomputersociety.org/.

[16] L. S. Shapley, "Stochastic games," *Proceedings of the National Academy of Sciences of the United States of America (PNAS)*, vol. 39, no. 10, pp. 1095–1100, 1953.

[17] M. Kwiatkowska, G. Norman, D. Parker, and G. Santos, "Automatic verification of concurrent stochastic systems," *Formal Methods in System Design*, pp. 1–63, 2021.

[18] ——, "Prism-games 3.0: Stochastic game verification with concurrency, equilibria and time," in *International Conference on Computer Aided Verification*, Springer, 2020, pp. 475–487.

[19] A. Kucera, "Turn-based stochastic games," *Lectures in Game Theory for Computer Scientists*, Jan. 2011. DOI: 10.1017/CBO9780511973468.006.

[20] N. Li, J. Cámara, D. Garlan, and B. Schmerl, "Reasoning about when to provide explanation for human-involved self-adaptive systems," in *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, 2020, pp. 195–204. DOI: 10.1109/ACSOS49614.2020.00042.

[21] S. Bogomolov, D. Magazzeni, A. Podelski, and M. Wehrle, "Planning as model checking in hybrid domains," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 28, 2014.

[22] R. Giaquinta, R. Hoffmann, M. Ireland, A. Miller, and G. Norman, "Strategy synthesis for autonomous agents using prism," in *NASA Formal Methods Symposium*, Springer, 2018, pp. 220–236.

[23] M. L. Puterman, *Markov Decision Processes: Discrete Stochastic Dynamic Programming*, 1st. USA: John Wiley & Sons, Inc., 1994, ISBN: 0471619779.

[24] N. Privault, "Discrete-time markov chains," in *Understanding Markov Chains*, Springer, 2018, pp. 89–113.

[25] R. Rezin, I. Afanasyev, M. Mazzara, and V. Rivera, "Model checking in multiplayer games development," in *2018 IEEE 32nd International Conference on Advanced Information Networking and Applications (AINA)*, 2018, pp. 826–833. DOI: 10.1109/AINA.2018.00122.

[26] S. Radomski and T. Neubacher, "Formal verification of selected game-logic specifications," *on Engineering Interactive Computer Systems with SCXML*, p. 30, 2015.

[27] G. Holzmann, *Spin Model Checker, the: Primer and Reference Manual*, First. Addison-Wesley Professional, 2003, ISBN: 0321228626.

[28] K. Havelund and T. Pressburger, "Model checking java programs using java pathfinder," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, pp. 366–381, 2000.

[29] P. Milazzo, G. Pardini, D. Sestini, and P. Bove, "Case studies of application of probabilistic and statistical model checking in game design," in *2015 IEEE/ACM 4th International Workshop on Games and Software Engineering*, IEEE, 2015, pp. 29–35.

[30] J. F. Nash *et al.*, "Equilibrium points in n-person games," *Proceedings of the national academy of sciences*, vol. 36, no. 1, pp. 48–49, 1950.

[31] P. Moreno-Ger, R. Fuentes-Fernández, J.-L. Sierra-Rodríguez, and B. Fernández-Manjón, "Model checking for adventure videogames," *Information and Software Technology*, vol. 51, no. 3, pp. 564–580, 2009, ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2008.08.003. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584908001134.

[32] A. Becker and D. Görlich, "What is game balancing? - an examination of concepts," *ParadigmPlus*, vol. 1, no. 1, pp. 22–41, Apr. 2020. [Online]. Available: https://journals.itiud.org/index.php/paradigmplus/article/view/7.

[33] M. Carter, M. Gibbs, and M. Harrop, "Metagames, paragames and orthogames: A new vocabulary," in *Proceedings of the international conference on the foundations of digital games*, 2012, pp. 11–17.

[34] R. Leigh, J. Schonfeld, and S. J. Louis, "Using coevolution to understand and validate game balance in continuous games," in *Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '08, Atlanta, GA, USA: Association for Computing Machinery, 2008, pp. 1563–1570, ISBN: 9781605581309. DOI: 10.1145/1389095.1389394. [Online]. Available: https://doi.org/10.1145/1389095.1389394.

[35] A. B. Cardona, A. W. Hansen, J. Togelius, and M. G. Friberger, "Open trumps, a data game," in *Proc. Int. Conf. Foundations of Digital Games (FDG'14)*, Society for the Advancement of the Science of Digital Games, 2014.

[36] V. Volz, G. Rudolph, and B. Naujoks, "Demonstrating the feasibility of automatic game balancing," in *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, ser. GECCO '16, Denver, Colorado, USA: Association for Computing Machinery, 2016, pp. 269–276, ISBN: 9781450342063. DOI: 10.1145/2908812.2908913. [Online]. Available: https://doi.org/10.1145/2908812.2908913.

[37] J. Pfau, A. Liapis, G. Volkmar, G. N. Yannakakis, and R. Malaka, "Dungeons replicants: Automated game balancing via deep player behavior modeling," in *2020 IEEE Conference on Games (CoG)*, 2020, pp. 431–438. DOI: 10.1109/CoG47356.2020.9231958.

[38] C. Browne, "Ai for ancient games," *KI-Künstliche Intelligenz*, vol. 34, no. 1, pp. 89–93, 2020.

[39] E. Piette, D. J. Soemers, M. Stephenson, C. F. Sironi, M. H. Winands, and C. Browne, "Ludii–the ludemic general game system," *arXiv preprint arXiv:1905.05013*, 2019.

[40] C. Browne, E. Powley, D. Whitehouse, S. Lucas, P. Cowling, P. Rohlfshagen, S. Tavener, D. Perez Liebana, S. Samothrakis, and S. Colton, "A survey of monte carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4:1, pp. 1–43, Mar. 2012. DOI: 10.1109/TCIAIG.2012.2186810.

[41] Z. Li and M. Wellman, "Structure learning for approximate solution of many-player games," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, 2020, pp. 2119–2127.

[42] S. F. Gudmundsson, P. Eisen, E. Poromaa, A. Nodet, S. Purmonen, B. Kozakowski, R. Meurling, and L. Cao, "Human-like playtesting with deep learning," in *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, 2018, pp. 1–8. DOI: 10.1109/CIG.2018.8490442.

[43] G. Lolli, *Osservazioni teorico-pratiche sopra il giuoco degli scacchi, ossia Il giuoco degli scacchi esposto nel suo miglior lume: Opera novissima contenente le leggi fondamentali, i precetti piu purgati*. Nella stamperia di s. Tommaso d'Aquino, 1769.

[44] B. Fischer, S. Margulies, and D. Mosenfelder, *Bobby Fischer teaches chess*. Bantam Books, 1982.

[45] N. Tomašev, U. Paquet, D. Hassabis, and V. Kramnik, "Assessing game balance with alphazero: Exploring alternative rule sets in chess," *arXiv preprint arXiv:2009.04374*, 2020.

[46] J. von Neumann, "Zur theorie der gesellschaftsspiele," *Mathematische Annalen*, vol. 100, pp. 295–320, 1928.

[47] J. von Neumann, O. Morgenstern, H. Kuhn, and A. Rubinstein, *Theory of Games and Economic Behavior*. Princeton University Press, 1944.

[48] J. G. Kemeny, J. L. Snell, and A. W. Knapp, *Denumerable Markov Chains*. Springer, 1976.

[49] A. Condon, "On algorithms for simple stochastic games," *Advances in computational complexity theory, DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, vol. 13, pp. 51–73, 1993.

[50] C. Baier and J.-P. Katoen, *Principles of Model Checking*. MIT Press, 2008.

[51] M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill, "Cmc: A pragmatic approach to model checking real code," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 75–88, Dec. 2003, ISSN: 0163-5980. DOI: 10.1145/844128.844136. [Online]. Available: https://doi.org/10.1145/844128.844136.

[52] R. Alur and T. A. Henzinger, "Reactive modules," *Formal methods in system design*, vol. 15, no. 1, pp. 7–48, 1999.

[53] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32.

[54] H. Hansson and B. Jonsson, "A logic for reasoning about time and reliability," *Formal aspects of computing*, vol. 6, no. 5, pp. 512–535, 1994.

[55] M. Reynolds, "An axiomatization of pctl*," *Information and Computation*, vol. 201, no. 1, pp. 72–119, 2005, ISSN: 0890-5401. DOI: https://doi.org/10.1016/j.ic.2005.03.005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0890540105000866.

[56] F. J. Lin, P. Chu, and M. T. Liu, "Protocol verification using reachability analysis: The state space explosion problem and relief strategies," in *Proceedings of the ACM workshop on Frontiers in computer communications technology*, 1987, pp. 126–135.

[57] A. Miller, A. Donaldson, and M. Calder, "Symmetry in temporal logic model checking," *ACM Computing Surveys (CSUR)*, vol. 38, no. 3, 8–es, 2006.

[58] E. M. Clarke, O. Grumberg, and D. E. Long, "Model checking and abstraction," *ACM transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 5, pp. 1512–1542, 1994.

[59] E. Berlecamp, J. Conway, and R. Guy, "Winning ways for your mathematical plays, volume 2. academic press, isbn 0-12-091152-3," pp. 669–710, 1982.

[60] J. Uiterwijk and H. van den Herik, "The advantage of the initiative," *Information Sciences*, vol. 122, no. 1, pp. 43–58, 2000, ISSN: 0020-0255. DOI: https://doi.org/10.1016/S0020-0255(99)00095-X. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S002002559900095X.

[61] V. Chelaru. (Jan. 2007). "Rock paper scissors - a method for competitive game play design," [Online]. Available: https://www.gamasutra.com/view/feature/130150/rock_paper_scissors__a_method_for_.php (visited on 01/11/2020).

[62] S. Donaldson, "Mechanics and metagame: Exploring binary expertise in league of legends," *Games and Culture*, vol. 12, no. 5, pp. 426–444, 2017.

[63] mediaxstanford, Jan. 2016. [Online]. Available: https://www.youtube.com/watch?v=b7EmCt_OMjE&feature=youtu.be.

[64] H. Smith, "Orthogonal unit differentiation presentation at game developers conference 2003," *Presentation available at http://www. gdconf. com/archives/2003/Smith_Harvey. ppt*, 2003.

[65] M. Friedman and L. J. Savage, "The expected-utility hypothesis and the measurability of utility," *Journal of Political Economy*, vol. 60, no. 6, pp. 463–474, 1952.

[66] A. S. Kahn, C. Shen, L. Lu, R. A. Ratan, S. Coary, J. Hou, J. Meng, J. Osborn, and D. Williams, "The trojan player typology: A cross-genre, cross-cultural, behaviorally validated scale of video game play motivations," *Computers in Human Behavior*, vol. 49, pp. 354–361, 2015.

[67] N. Yee, "Motivations for play in online games," *CyberPsychology & behavior*, vol. 9, no. 6, pp. 772–775, 2006.

[68] R. Bartle, "Hearts, clubs, diamonds, spades: Players who suit muds," *Journal of MUD research*, vol. 1, no. 1, p. 19, 1996.

[69] Riot, *Champion balance framework*, https://nexus.leagueoflegends.com/en-us/2019/05/dev-champion-balance-framework/, 2019.

[70] https://github.com/WJLKavanagh/chained_strategy_generation/.

[71] https://github.com/WJLKavanagh/csg/tree/master/5char_csg.

[72] W. J. Kavanagh, W. Wallis, and A. Miller, *Rpglite player data and lookup tables*, http://researchdata.gla.ac.uk/1070/, 2020. DOI: 10.5525/gla.researchdata.1070.

[73] J. K. Haas, "A history of the unity game engine," 2014.

[74] Zynga, *Words with friends: Play fun word puzzle game*, version 15.622. [Online]. Available: https://www.zynga.com/ (visited on 12/20/2020).

[75] J. Brophy, *Motivating students to learn*. Routledge, 2004.

[76] J. Shore and S. Warden, *The art of agile development*. " O'Reilly Media, Inc.", 2021.

[77] A. Tychsen, M. Hitchens, and T. Brolund, "Motivations for play in computer role-playing games," in *Proceedings of the 2008 Conference on Future Play: Research, Play, Share*, ser. Future Play '08, Toronto, Ontario, Canada: Association for Computing Machinery, 2008, pp. 57–64, ISBN: 9781605582184. DOI: 10.1145/1496984.1496995. [Online]. Available: https://doi.org/10.1145/1496984.1496995.

[78] J. F. Groote, T. W. Kouters, and A. Osaiweran, "Specification guidelines to avoid the state space explosion problem," *Software Testing, Verification and Reliability*, vol. 25, no. 1, pp. 4–33, 2015.

[79] O. Clark, *Games as a service: How free to play design can make better games*. CRC Press, 2014.

[80] A. E. Elo, *The rating of chessplayers, past and present*. Arco Pub., 1978.

[81] T. Minka, R. Cleven, and Y. Zaykov, "TrueSkill 2: An improved Bayesian skill rating system," Microsoft, Tech. Rep. MSR-TR-2018-8, Mar. 2018.

[82] S. Kovalchik, "Extension of the elo rating system to margin of victory," *International Journal of Forecasting*, vol. 36, no. 4, pp. 1329–1341, 2020, ISSN: 0169-2070. DOI: https://doi.org/10.1016/j.ijforecast.2020.01.006. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0169207020300157.

[83] Y. N. Ravari, P. Spronck, R. Sifa, and A. Drachen, "Predicting victory in a hybrid online competitive game: The case of destiny.," in *AIIDE*, 2017, pp. 207–213.

[84] C. DeLong and J. Srivastava, "Teamskill evolved: Mixed classification schemes for team-based multi-player games," in *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, Springer, 2012, pp. 26–37.

[85] S. D. Levitt, J. A. List, and D. H. Reiley, "What happens in the field stays in the field: Exploring whether professionals play minimax in laboratory experiments," *Econometrica*, vol. 78, no. 4, pp. 1413–1434, 2010.

[86] M. Ponsen, "The dynamics of human behaviour in Poker," in *Proceedings of the Belgian-Dutch conference in Artificiall Intelligence*, 2008, pp. 225–232.

[87] M. Z. Kwiatkowska, "Model checking and strategy synthesis for stochastic games: From theory to practice," 2016.

[88] W. J. Kavanagh, A. Miller, G. Norman, and O. Andrei, "Balancing turn-based games with chained strategy generation," *IEEE Transactions on Games*, 2019.

[89] R. Giaquinta, R. Hoffmann, M. Ireland, A. Miller, and G. Norman, "Strategy synthesis for autonomous agents using PRISM," in *Proceedings of the 10th NASA Formal Methods Symposium (NFM)*, 2018, pp. 230–236.

[90] D. E. Bell, "Regret in decision making under uncertainty," *Operations research*, vol. 30, no. 5, pp. 961–981, 1982.

[91] M. Aung, V. Bonometti, A. Drachen, P. Cowling, A. V. Kokkinakis, C. Yoder, and A. Wade, "Predicting skill learning in a large, longitudinal moba dataset," in *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, 2018, pp. 1–7.

[92] A. R. Novak, K. Bennett, M. Pluss, and J. Fransen, "Performance analysis in esports: Part 1-the validity and reliability of match statistics and notational analysis in league of legends," 2019.

[93] N. Taylor, "The numbers game: Collegiate esports and the instrumentation of movement performance," in *Sports, Society, and Technology*, Springer, 2020, pp. 121–144.

[94] M. Johanson, N. Burch, R. Valenzano, and M. Bowling, "Evaluating state-space abstractions in extensive-form games," in *Proceedings of the international conference on Autonomous agents and multiagent systems*, ser. International Foundation for Autonomous Agents and Multiagent Systems, 2013, pp. 271–278.

[95] T. Sandholm, "Abstraction for solving large incomplete-information games," in *AAAI Conference on Artificial Intelligence*, 2015.

[96] A. Miller, D. A.F., and M. Calder, "Symmetry in temporal logic model checking," *ACM Computing Surveys*, vol. 38, no. 3, 2006.

[97] A. Donaldson, A. Miller, and D. Parker, "Language-level symmetry reduction for probabilistic model checking," in *Proceedings of the 6th International Conference on the Quantitative Evaluation of Systems (QEST'09)*, 2009, pp. 289–298.

Appendix A

Fox and Geese Model

```
mdp

module fox_and_geese

fox : [0..31]   init 1;
goose1 : [0..31] init 28;
goose2 : [0..31] init 29;
goose3 : [0..31] init 30;
goose4 : [0..31] init 31;
turn : [0..1] init 0;       // fox on 0, geese on 1, fox goes first

// fox down and right from 2n row.
[fox_dr] turn = 0 & (fox = 0 | fox = 1 | fox = 2 | fox = 3 |
    fox = 8 | fox = 9 | fox = 10 | fox = 11 |
    fox = 16 | fox = 17 | fox = 18 | fox = 19 |
    fox = 24 | fox = 25 | fox = 26 | fox = 27) &
    (goose1 != fox + 4 & goose3 != fox + 4 & goose3 != fox + 4 & goose4
   != fox + 4) ->
        (fox' = fox + 4) & (turn' = 1);

// fox down and right from 2n+1 row.
[fox_dr] turn = 0 & (fox = 4 |fox = 5 | fox = 6 |
    fox = 12 | fox = 13 | fox = 14 |
    fox = 20 | fox = 21 | fox = 22 ) &
    (goose1 != fox + 5 & goose2 != fox + 5 & goose3 != fox + 5 & goose4
   != fox + 5) ->
        (fox' = fox + 5) & (turn' = 1);

// fox down and left from 2n row.
[fox_dl] turn = 0 & (fox = 1 | fox = 2 | fox = 3 |
    fox = 9 | fox = 10 | fox = 11 |
    fox = 17 | fox = 18 | fox = 19 |
    fox = 25 | fox = 26 | fox = 27) &
```

```
32      (goose1 != fox + 3 & goose3 != fox + 3 & goose3 != fox + 3 & goose4
    != fox + 3) ->
33          (fox' = fox + 3) & (turn' = 1);
34
35 // fox down and left from 2n+1 row.
36 [fox_dl] turn = 0 & (fox = 4 |fox = 5 | fox = 6 | fox = 7 |
37      fox = 12 | fox = 13 | fox = 14 | fox = 15 |
38      fox = 20 | fox = 21 | fox = 22 | fox = 23) &
39      (goose1 != fox + 4 & goose3 != fox + 4 & goose3 != fox + 4 & goose4
    != fox + 4) ->
40          (fox' = fox + 4) & (turn' = 1);
41
42 // fox up and right from 2n row.
43 [fox_ur] turn = 0 & (fox = 8 | fox = 9 | fox = 10 | fox = 11 |
44      fox = 16 | fox = 17 | fox = 18 | fox = 19 |
45      fox = 24 | fox = 25 | fox = 26 | fox = 27) &
46      (goose1 != fox - 4 & goose3 != fox - 4 & goose3 != fox - 4 & goose4
    != fox - 4) ->
47          (fox' = fox - 4) & (turn' = 1);
48
49 // fox up and right from 2n+1 row.
50 [fox_ur] turn = 0 & (fox = 4 |fox = 5 | fox = 6 |
51      fox = 12 | fox = 13 | fox = 14 |
52      fox = 20 | fox = 21 | fox = 22 |
53      fox = 28 | fox = 29 | fox = 30) &
54      (goose1 != fox - 3 & goose3 != fox - 3 & goose3 != fox - 3 & goose4
    != fox - 3) ->
55          (fox' = fox - 3) & (turn' = 1);
56
57 // fox up and left from 2n row.
58 [fox_ul] turn = 0 & (fox = 9 | fox = 10 | fox = 11 |
59      fox = 17 | fox = 18 | fox = 19 |
60      fox = 25 | fox = 26 | fox = 27) &
61      (goose1 != fox - 5 & goose3 != fox - 5 & goose3 != fox - 5 & goose4
    != fox - 5) ->
62          (fox' = fox - 5) & (turn' = 1);
63
64 // fox up and left from 2n+1 row.
65 [fox_ul] turn = 0 & (fox = 4 |fox = 5 | fox = 6 | fox = 7 |
66      fox = 12 | fox = 13 | fox = 14 | fox = 15 |
67      fox = 20 | fox = 21 | fox = 22 | fox = 23 |
68      fox = 28 | fox = 29 | fox = 30 | fox = 31) &
69      (goose1 != fox - 4 & goose3 != fox - 4 & goose3 != fox - 4 & goose4
    != fox - 4) ->
70          (fox' = fox - 4) & (turn' = 1);
71
```

```
72 [g1_r] turn = 1 & (goose1 = 8 | goose1 = 9 | goose1 = 10 | goose1 = 11 |
73     goose1 = 16 | goose1 = 17 | goose1 = 18 | goose1 = 19 |
74     goose1 = 24 | goose1 = 25 | goose1 = 26 | goose1 = 27) &
75     fox != goose1 - 4 & goose2 != goose1 - 4 & goose3 != goose1 - 4 &
    goose4 != goose1 - 4 ->
76         (goose1' = goose1 - 4) & (turn' = 0);
77 [g1_r_] turn = 1 & (goose1 = 4 |goose1 = 5 | goose1 = 6 |
78     goose1 = 12 | goose1 = 13 | goose1 = 14 |
79     goose1 = 20 | goose1 = 21 | goose1 = 22 |
80     goose1 = 28 | goose1 = 29 | goose1 = 30) &
81     fox != goose1 - 3 & goose2 != goose1 - 3 & goose3 != goose1 - 3 &
    goose4 != goose1 - 3 ->
82         (goose1' = goose1 - 3) & (turn' = 0);
83 [g1_l] turn = 1 & (goose1 = 9 | goose1 = 10 | goose1 = 11 |
84     goose1 = 17 | goose1 = 18 | goose1 = 19 |
85     goose1 = 25 | goose1 = 26 | goose1 = 27) &
86     fox != goose1 - 5 & goose2 != goose1 - 5 & goose3 != goose1 - 5 &
    goose4 != goose1 - 5 ->
87         (goose1' = goose1 - 5) & (turn' = 0);
88 [g1_l_] turn = 1 & (goose1 = 4 |goose1 = 5 | goose1 = 6 | goose1 = 7 |
89     goose1 = 12 | goose1 = 13 | goose1 = 14 | goose1 = 15 |
90     goose1 = 20 | goose1 = 21 | goose1 = 22 | goose1 = 23 |
91     goose1 = 28 | goose1 = 29 | goose1 = 30 | goose1 = 31) &
92     fox != goose1 - 4 & goose2 != goose1 - 4 & goose3 != goose1 - 4 &
    goose4 != goose1 - 4 ->
93         (goose1' = goose1 - 4) & (turn' = 0);
94 [g1_rl] turn = 1 & (goose1 = 4 | goose1 = 5 | goose1 = 6 |
95     goose1 = 12 | goose1 = 13 | goose1 = 14 |
96     goose1 = 20 | goose1 = 21 | goose1 = 22 |
97     goose1 = 28 | goose1 = 29 | goose1 = 30) &
98     (fox != goose1 - 3  & goose2 != goose1 - 3  & goose3 != goose1 - 3
    & goose4 != goose1 - 3 ) &
99     (fox != goose1 - 4 & goose2 != goose1 - 4 & goose3 != goose1 - 4 &
    goose4 != goose1 - 4) ->
100        0.5 : (goose1' = goose1 - 4) & (turn' = 0) +
101        0.5 : (goose1' = goose1 - 3 ) & (turn' = 0);
102 [g1_rl_] turn = 1 & (goose1 = 9 | goose1 = 10 | goose1 = 11 |
103    goose1 = 17 | goose1 = 18 | goose1 = 19 |
104    goose1 = 25 | goose1 = 26 | goose1 = 27) &
105    (fox != goose1 - 5 & goose2 != goose1 - 5 & goose3 != goose1 - 5 &
    goose4 != goose1 - 5) &
106    (fox != goose1 - 4 & goose2 != goose1 - 4 & goose3 != goose1 - 4 &
    goose4 != goose1 - 4) ->
107        0.5 : (goose1' = goose1 - 4) & (turn' = 0) +
108        0.5 : (goose1' = goose1 - 5) & (turn' = 0);
109
```

```
110 [g2_r] turn = 1 & (goose2 = 8 | goose2 = 9 | goose2 = 10 | goose2 = 11 |
111     goose2 = 16 | goose2 = 17 | goose2 = 18 | goose2 = 19 |
112     goose2 = 24 | goose2 = 25 | goose2 = 26 | goose2 = 27) &
113     fox != goose2 - 4 & goose1 != goose2 - 4 & goose3 != goose2 - 4 &
    goose4 != goose2 - 4 ->
114         (goose2' = goose2 - 4) & (turn' = 0);
115 [g2_r_] turn = 1 & (goose2 = 4 |goose2 = 5 | goose2 = 6 |
116     goose2 = 12 | goose2 = 13 | goose2 = 14 |
117     goose2 = 20 | goose2 = 21 | goose2 = 22 |
118     goose2 = 28 | goose2 = 29 | goose2 = 30) &
119     fox != goose2 - 3 & goose1 != goose2 - 3 & goose3 != goose2 - 3 &
    goose4 != goose2 - 3 ->
120         (goose2' = goose2 - 3) & (turn' = 0);
121 [g2_l] turn = 1 & (goose2 = 9 | goose2 = 10 | goose2 = 11 |
122     goose2 = 17 | goose2 = 18 | goose2 = 19 |
123     goose2 = 25 | goose2 = 26 | goose2 = 27) &
124     fox != goose2 - 5 & goose1 != goose2 - 5 & goose3 != goose2 - 5 &
    goose4 != goose2 - 5 ->
125         (goose2' = goose2 - 5) & (turn' = 0);
126 [g2_l_] turn = 1 & (goose2 = 4 |goose2 = 5 | goose2 = 6 | goose2 = 7 |
127     goose2 = 12 | goose2 = 13 | goose2 = 14 | goose2 = 15 |
128     goose2 = 20 | goose2 = 21 | goose2 = 22 | goose2 = 23 |
129     goose2 = 28 | goose2 = 29 | goose2 = 30 | goose2 = 31) &
130     fox != goose2 - 4 & goose1 != goose2 - 4 & goose3 != goose2 - 4 &
    goose4 != goose2 - 4 ->
131         (goose2' = goose2 - 4) & (turn' = 0);
132 [g2_rl] turn = 1 & (goose2 = 4 | goose2 = 5 | goose2 = 6 |
133     goose2 = 12 | goose2 = 13 | goose2 = 14 |
134     goose2 = 20 | goose2 = 21 | goose2 = 22 |
135     goose2 = 28 | goose2 = 29 | goose2 = 30) &
136     (fox != goose2 - 3  & goose1 != goose2 - 3  & goose3 != goose2 - 3
    & goose4 != goose2 - 3 ) &
137     (fox != goose2 - 4 & goose1 != goose2 - 4 & goose3 != goose2 - 4 &
    goose4 != goose2 - 4) ->
138         0.5 : (goose2' = goose2 - 4) & (turn' = 0) +
139         0.5 : (goose2' = goose2 - 3 ) & (turn' = 0);
140 [g2_rl_] turn = 1 & (goose2 = 9 | goose2 = 10 | goose2 = 11 |
141     goose2 = 17 | goose2 = 18 | goose2 = 19 |
142     goose2 = 25 | goose2 = 26 | goose2 = 27) &
143     (fox != goose2 - 5 & goose1 != goose2 - 5 & goose3 != goose2 - 5 &
    goose4 != goose2 - 5) &
144     (fox != goose2 - 4 & goose1 != goose2 - 4 & goose3 != goose2 - 4 &
    goose4 != goose2 - 4) ->
145         0.5 : (goose2' = goose2 - 4) & (turn' = 0) +
146         0.5 : (goose2' = goose2 - 5) & (turn' = 0);
147
```

```
148 [g3_r] turn = 1 & (goose3 = 8 | goose3 = 9 | goose3 = 10 | goose3 = 11 |
149     goose3 = 16 | goose3 = 17 | goose3 = 18 | goose3 = 19 |
150     goose3 = 24 | goose3 = 25 | goose3 = 26 | goose3 = 27) &
151     fox != goose3 - 4 & goose1 != goose3 - 4 & goose2 != goose3 - 4 &
    goose4 != goose3 - 4 ->
152         (goose3' = goose3 - 4) & (turn' = 0);
153 [g3_r_] turn = 1 & (goose3 = 4 |goose3 = 5 | goose3 = 6 |
154     goose3 = 12 | goose3 = 13 | goose3 = 14 |
155     goose3 = 20 | goose3 = 21 | goose3 = 22 |
156     goose3 = 28 | goose3 = 29 | goose3 = 30) &
157     fox != goose3 - 3 & goose1 != goose3 - 3 & goose2 != goose3 - 3 &
    goose4 != goose3 - 3 ->
158         (goose3' = goose3 - 3) & (turn' = 0);
159 [g3_l] turn = 1 & (goose3 = 9 | goose3 = 10 | goose3 = 11 |
160     goose3 = 17 | goose3 = 18 | goose3 = 19 |
161     goose3 = 25 | goose3 = 26 | goose3 = 27) &
162     fox != goose3 - 5 & goose1 != goose3 - 5 & goose2 != goose3 - 5 &
    goose4 != goose3 - 5 ->
163         (goose3' = goose3 - 5) & (turn' = 0);
164 [g3_l_] turn = 1 & (goose3 = 4 |goose3 = 5 | goose3 = 6 | goose3 = 7 |
165     goose3 = 12 | goose3 = 13 | goose3 = 14 | goose3 = 15 |
166     goose3 = 20 | goose3 = 21 | goose3 = 22 | goose3 = 23 |
167     goose3 = 28 | goose3 = 29 | goose3 = 30 | goose3 = 31) &
168     fox != goose3 - 4 & goose1 != goose3 - 4 & goose2 != goose3 - 4 &
    goose4 != goose3 - 4 ->
169         (goose3' = goose3 - 4) & (turn' = 0);
170 [g3_rl] turn = 1 & (goose3 = 4 | goose3 = 5 | goose3 = 6 |
171     goose3 = 12 | goose3 = 13 | goose3 = 14 |
172     goose3 = 20 | goose3 = 21 | goose3 = 22 |
173     goose3 = 28 | goose3 = 29 | goose3 = 30) &
174     (fox != goose3 - 3  & goose1 != goose3 - 3  & goose2 != goose3 - 3
    & goose4 != goose3 - 3 ) &
175     (fox != goose3 - 4 & goose1 != goose3 - 4 & goose2 != goose3 - 4 &
    goose4 != goose3 - 4) ->
176         0.5 : (goose3' = goose3 - 4) & (turn' = 0) +
177         0.5 : (goose3' = goose3 - 3 ) & (turn' = 0);
178 [g3_rl_] turn = 1 & (goose3 = 9 | goose3 = 10 | goose3 = 11 |
179     goose3 = 17 | goose3 = 18 | goose3 = 19 |
180     goose3 = 25 | goose3 = 26 | goose3 = 27) &
181     (fox != goose3 - 5 & goose1 != goose3 - 5 & goose2 != goose3 - 5 &
    goose4 != goose3 - 5) &
182     (fox != goose3 - 4 & goose1 != goose3 - 4 & goose2 != goose3 - 4 &
    goose4 != goose3 - 4) ->
183         0.5 : (goose3' = goose3 - 4) & (turn' = 0) +
184         0.5 : (goose3' = goose3 - 5) & (turn' = 0);
185
```

```
[g4_r] turn = 1 & (goose4 = 8 | goose4 = 9 | goose4 = 10 | goose4 = 11 |
    goose4 = 16 | goose4 = 17 | goose4 = 18 | goose4 = 19 |
    goose4 = 24 | goose4 = 25 | goose4 = 26 | goose4 = 27) &
    fox != goose4 - 4 & goose1 != goose4 - 4 & goose2 != goose4 - 4 &
   goose3 != goose4 - 4 ->
        (goose4' = goose4 - 4) & (turn' = 0);
[g4_r_] turn = 1 & (goose4 = 4 |goose4 = 5 | goose4 = 6 |
    goose4 = 12 | goose4 = 13 | goose4 = 14 |
    goose4 = 20 | goose4 = 21 | goose4 = 22 |
    goose4 = 28 | goose4 = 29 | goose4 = 30) &
    fox != goose4 - 3 & goose1 != goose4 - 3 & goose2 != goose4 - 3 &
   goose3 != goose4 - 3 ->
        (goose4' = goose4 - 3) & (turn' = 0);
[g4_l] turn = 1 & (goose4 = 9 | goose4 = 10 | goose4 = 11 |
    goose4 = 17 | goose4 = 18 | goose4 = 19 |
    goose4 = 25 | goose4 = 26 | goose4 = 27) &
    fox != goose4 - 5 & goose1 != goose4 - 5 & goose2 != goose4 - 5 &
   goose3 != goose4 - 5 ->
        (goose4' = goose4 - 5) & (turn' = 0);
[g4_l_] turn = 1 & (goose4 = 4 |goose4 = 5 | goose4 = 6 | goose4 = 7 |
    goose4 = 12 | goose4 = 13 | goose4 = 14 | goose4 = 15 |
    goose4 = 20 | goose4 = 21 | goose4 = 22 | goose4 = 23 |
    goose4 = 28 | goose4 = 29 | goose4 = 30 | goose4 = 31) &
    fox != goose4 - 4 & goose1 != goose4 - 4 & goose2 != goose4 - 4 &
   goose3 != goose4 - 4 ->
        (goose4' = goose4 - 4) & (turn' = 0);
[g4_rl] turn = 1 & (goose4 = 4 | goose4 = 5 | goose4 = 6 |
    goose4 = 12 | goose4 = 13 | goose4 = 14 |
    goose4 = 20 | goose4 = 21 | goose4 = 22 |
    goose4 = 28 | goose4 = 29 | goose4 = 30) &
    (fox != goose4 - 3  & goose1 != goose4 - 3  & goose2 != goose4 - 3
   & goose3 != goose4 - 3 ) &
    (fox != goose4 - 4 & goose1 != goose4 - 4 & goose2 != goose4 - 4 &
   goose3 != goose4 - 4) ->
        0.5 : (goose4' = goose4 - 4) & (turn' = 0) +
        0.5 : (goose4' = goose4 - 3 ) & (turn' = 0);
[g4_rl_] turn = 1 & (goose4 = 9 | goose4 = 10 | goose4 = 11 |
    goose4 = 17 | goose4 = 18 | goose4 = 19 |
    goose4 = 25 | goose4 = 26 | goose4 = 27) &
    (fox != goose4 - 5 & goose1 != goose4 - 5 & goose2 != goose4 - 5 &
   goose3 != goose4 - 5) &
    (fox != goose4 - 4 & goose1 != goose4 - 4 & goose2 != goose4 - 4 &
   goose3 != goose4 - 4) ->
        0.5 : (goose4' = goose4 - 4) & (turn' = 0) +
        0.5 : (goose4' = goose4 - 5) & (turn' = 0);

```

```
224 endmodule
225
226 label "fox_wins" = fox > 27 | (fox > goose1 & fox > goose2 & fox >
       goose3 & fox > goose4);
```

Appendix B

Chained Strategy Generation Model

```
// Author:  William Kavanagh, University of Glasgow
// Created: 2021-01-13 15:41:29
// File:  CSG auto-generated model
// Characters:  AW vs KA

mdp

// Configuration D:
const int Knight_health = 9;
const int Knight_damage = 3;
const double Knight_accuracy = 0.7;

const int Archer_health = 7;
const int Archer_damage = 2;
const double Archer_accuracy = 0.8;

const int Wizard_health = 7;
const int Wizard_damage = 2;
const double Wizard_accuracy = 0.85;

module game
  attack  : [0..9]; // Action decision: 0 - NONE, 1 - p1c1>p2c1, 2 - p1c1>p2c2, 3 - p1c2>p2c1, 4 - p1c2>p2c2, 5 - p2c1>p1c1, 6 - p2c1>p1c2, 7 - p2c2>p1c1, 8 - p2c2>p1c2, 9 - NEXT
  turn  : [0..2]; // Player to act
// Health and is_stunned variables
  p1c1  : [health_floor..health_ceiling] init 7;  // player 1 character 1 health value
  p1c2  : [health_floor..health_ceiling] init 7;  // player 1 character 2 health value
  p1_stun : [0..2];           // 0 - Neither character stunned, 1 - character 1 stunned, 2 - character 2 stunned
```

```
28   p2c1  : [health_floor..health_ceiling] init 9;  // player 2 character
      1 health value
29   p2c2  : [health_floor..health_ceiling] init 7;  // player 2 character
      2 health value
30   p2_stun : [0..2];           // 0 - Neither character stunned, 1 -
      character 1 stunned, 2 - character 2 stunned
31
32   [flip_coin] turn = 0 -> 0.5 : (turn' = 1) + 0.5 : (turn' = 2);
33   [next_turn] attack = 9 & turn > 0 & (p1c1 > 0 | p1c2 > 0) & (p2c1 > 0
      | p2c2 > 0) -> (attack' = 0) & (turn' = 3 - turn);
34
35 // Action decision for P1, free strategy
36   [p1_turn_1] attack = 0 & turn = 1 & p1c1 > 0 & p1_stun != 1 & (p2c1 >
      0 | p2c2 > 0) -> (attack' = 1) & (p1_stun' = 0);
37   [p1_turn_3] attack = 0 & turn = 1 & p1c2 > 0 & p1_stun != 2 & p2c1 > 0
       -> (attack' = 3) & (p1_stun' = 0);
38   [p1_turn_4] attack = 0 & turn = 1 & p1c2 > 0 & p1_stun != 2 & p2c2 > 0
       -> (attack' = 4) & (p1_stun' = 0);
39   [p1_turn_skip]  attack = 0 & turn = 1 & ( (p1_stun = 1 & p1c2 < 1) | (
      p1_stun = 2 & p1c1 < 1) ) -> (attack' = 9) & (p1_stun' = 0); // skip
      if forced
40 // Action decision for P2, stochastic strategy
41   [p2_turn_9] p2c1 = -2 & p2c2 = -2 & p2_stun = 0 & p1c1 = -2 & p1c2 =
      -2 -> (attack' = 9) & (p2_stun' = 0);
42   [p2_turn_9] p2c1 = -2 & p2c2 = -2 & p2_stun = 1 & p1c1 = -2 & p1c2 =
      -2 -> (attack' = 9) & (p2_stun' = 0);
43   [p2_turn_9] p2c1 = -2 & p2c2 = -2 & p2_stun = 2 & p1c1 = -2 & p1c2 =
      -2 -> (attack' = 9) & (p2_stun' = 0);
44   [p2_turn_9] p2c1 = -2 & p2c2 = -2 & p2_stun = 0 & p1c1 = -2 & p1c2 =
      -1 -> (attack' = 9) & (p2_stun' = 0);
45   [p2_turn_9] p2c1 = -2 & p2c2 = -2 & p2_stun = 1 & p1c1 = -2 & p1c2 =
      -1 -> (attack' = 9) & (p2_stun' = 0);
46 // ...
47 // 62199 lines skipped
48 // ...
49   [p2_turn_7] p2c1 = 9 & p2c2 = 9 & p2_stun = 0 & p1c1 = 9 & p1c2 = 9 ->
       (attack' = 7) & (p2_stun' = 0);
50   [p2_turn_7] p2c1 = 9 & p2c2 = 9 & p2_stun = 1 & p1c1 = 9 & p1c2 = 9 ->
       (attack' = 7) & (p2_stun' = 0);
51   [p2_turn_5] p2c1 = 9 & p2c2 = 9 & p2_stun = 2 & p1c1 = 9 & p1c2 = 9 ->
       (attack' = 5) & (p2_stun' = 0);
52
53 // Action resolution player 1
54   [p1c1_p2c1] attack = 1 & p2c1 > 0 -> Archer_accuracy: (p2c1' = p2c1 -
      Archer_damage) & (attack' = 2) + 1 - Archer_accuracy: (attack' = 2);
55   [p1c1_p2c1] attack = 1 & p2c1 < 1 -> (attack' = 2);
```

```
56   [p1c1_p2c2] attack = 2 & p2c2 > 0 -> Archer_accuracy: (p2c2' = p2c2 -
      Archer_damage) & (attack' = 9) + 1 - Archer_accuracy: (attack' = 9);
57   [p1c1_p2c2] attack = 2 & p2c2 < 1 -> (attack' = 9);
58   [p1c2_p2c1] attack = 3 & p2c1 > 0 -> Wizard_accuracy: (p2c1' = p2c1 -
      Wizard_damage) & (attack' = 9) & (p2_stun' = 1) + 1 - Wizard_accuracy
      : (attack' = 9);
59   [p1c2_p2c2] attack = 4 & p2c2 > 0 -> Wizard_accuracy: (p2c2' = p2c2 -
      Wizard_damage) & (attack' = 9) & (p2_stun' = 2) + 1 - Wizard_accuracy
      : (attack' = 9);
60 // Action resolution player 2
61   [p2c1_p1c1] attack = 5 & p1c1 > 0 -> Knight_accuracy: (p1c1' = p1c1 -
      Knight_damage) & (attack' = 9) + 1 - Knight_accuracy: (attack' = 9);
62   [p2c1_p1c2] attack = 6 & p1c2 > 0 -> Knight_accuracy: (p1c2' = p1c2 -
      Knight_damage) & (attack' = 9) + 1 - Knight_accuracy: (attack' = 9);
63   [p2c2_p1c1] attack = 7 & p1c1 > 0 -> Archer_accuracy: (p1c1' = p1c1 -
      Archer_damage) & (attack' = 8) + 1 - Archer_accuracy: (attack' = 8);
64   [p2c2_p1c1] attack = 7 & p1c1 < 1 -> (attack' = 8);
65   [p2c2_p1c2] attack = 8 & p1c2 > 0 -> Archer_accuracy: (p1c2' = p1c2 -
      Archer_damage) & (attack' = 9) + 1 - Archer_accuracy: (attack' = 9);
66   [p2c2_p1c2] attack = 8 & p1c2 < 1 -> (attack' = 9);
67 endmodule
68
69 label "p1_wins" = (p1c1 > 0 | p1c2 > 0) & p2c1 < 1 & p2c2 < 1;
70 label "p2_wins" = (p2c1 > 0 | p2c2 > 0) & p1c1 < 1 & p1c2 < 1;
71 formula health_ceiling  = max(Knight_health, Archer_health,
      Wizard_health);
72 formula health_floor  = 1 - max(Knight_damage, Archer_damage,
      Wizard_damage);
```

Appendix C

RPGLite 2 KA-optimality Generator

```
// Author:  William Kavanagh, University of Glasgow
// Created: 2021-01-28
// File:    CSG auto-generated model
// Comment: This file is a generator for a later strategy for KA

// Configuration DELTA9:

smg
const int Knight_health = 9;
const int Knight_damage = 4;
const double Knight_accuracy = 0.63;

const int Archer_health = 7;
const int Archer_damage = 2;
const double Archer_accuracy = 0.97;

const int Wizard_health = 7;
const int Wizard_damage = 3;
const double Wizard_accuracy = 0.7;

const int Rogue_health = 6;
const int Rogue_damage = 3;
const int Rogue_execute = 6;
const double Rogue_accuracy = 0.66;

const int Healer_health = 7;
const int Healer_damage = 2;
const int Healer_heal = 2;
const double Healer_accuracy = 0.8;

player p1
[p1_K_K], [p1_K_A], [p1_K_W], [p1_K_R], [p1_K_H], [p1_A_K], [p1_A_KA], [
```

```
   p1_A_KW], [p1_A_KR], [p1_A_KH], [p1_A_A], [p1_A_AW], [p1_A_AR], [
   p1_A_AH], [p1_A_W], [p1_A_WR], [p1_A_WH], [p1_A_R], [p1_A_RH], [
   p1_A_H], [p1_W_K], [p1_W_A], [p1_W_W], [p1_W_R], [p1_W_H], [p1_R_K],
   [p1_R_Ke], [p1_R_A], [p1_R_Ae], [p1_R_W], [p1_R_We], [p1_R_R], [
   p1_R_Re], [p1_R_H], [p1_R_He], [p1_H_KK], [p1_H_KA], [p1_H_KW], [
   p1_H_KR], [p1_H_KH], [p1_H_AK], [p1_H_AA], [p1_H_AW], [p1_H_AR], [
   p1_H_AH], [p1_H_WK], [p1_H_WA], [p1_H_WW], [p1_H_WR], [p1_H_WH], [
   p1_H_RK], [p1_H_RA], [p1_H_RW], [p1_H_RR], [p1_H_RH], [p1_H_HK], [
   p1_H_HA], [p1_H_HW], [p1_H_HR], [p1_H_HH], [p1_skip]
endplayer

player p2
[choose_RH], [choose_WH], [choose_WR], [choose_AH], [choose_AR], [
   choose_AW], [choose_KH], [choose_KR], [choose_KW], [choose_KA], [
   p2_K_K], [p2_K_A], [p2_K_W], [p2_K_R], [p2_K_H], [p2_A_K], [p2_A_KA],
    [p2_A_KW], [p2_A_KR], [p2_A_KH], [p2_A_A], [p2_A_AW], [p2_A_AR], [
   p2_A_AH], [p2_A_W], [p2_A_WR], [p2_A_WH], [p2_A_R], [p2_A_RH], [
   p2_A_H], [p2_W_K], [p2_W_A], [p2_W_W], [p2_W_R], [p2_W_H], [p2_R_K],
   [p2_R_Ke], [p2_R_A], [p2_R_Ae], [p2_R_W], [p2_R_We], [p2_R_R], [
   p2_R_Re], [p2_R_H], [p2_R_He], [p2_H_KK], [p2_H_KA], [p2_H_KW], [
   p2_H_KR], [p2_H_KH], [p2_H_AK], [p2_H_AA], [p2_H_AW], [p2_H_AR], [
   p2_H_AH], [p2_H_WK], [p2_H_WA], [p2_H_WW], [p2_H_WR], [p2_H_WH], [
   p2_H_RK], [p2_H_RA], [p2_H_RW], [p2_H_RR], [p2_H_RH], [p2_H_HK], [
   p2_H_HA], [p2_H_HW], [p2_H_HR], [p2_H_HH], [p2_skip]
endplayer

player sys
[coin_flip]
endplayer

module game
  turn  : [0..2];
  p1K : [0..Knight_health]    init Knight_health; // P1 Knight
  p1A : [0..Archer_health]    init Archer_health; // P1 Archer
  p1W : [0..Wizard_health]    init 0; // P1 Wizard not used
  p1R : [0..Rogue_health]   init 0; // P1 Rogue not used
  p1H : [0..Healer_health]    init 0; // P1 Healer not used
  p1_stun : [0..5];       //0 - none, 1 - Knight stunned, 2 - Archer
    stunned, 3 - Wizard stunned .. etc
  p2K : [0..Knight_health]    init Knight_health; // P2 Knight
  p2A : [0..Archer_health]    init Archer_health; // P2 Archer
  p2W : [0..Wizard_health]    init Wizard_health; // P2 Wizard
  p2R : [0..Rogue_health]   init Rogue_health; // P2 Rogue
  p2H : [0..Healer_health]    init Healer_health; // P2 Healer
  p2_stun : [0..5];       //0 - none, 1 - Knight stunned, 2 - Archer
    stunned, 3 - Wizard stunned .. etc
```

```
57
58 // Choose opposing material
59   [choose_KA] p2K > 0 & p2A > 0 & p2W > 0 & p2R > 0 & p2H > 0 ->
60     (p2W' = 0) & (p2R' = 0) & (p2H' = 0);
61   [choose_KW] p2K > 0 & p2A > 0 & p2W > 0 & p2R > 0 & p2H > 0 ->
62     (p2A' = 0) & (p2R' = 0) & (p2H' = 0);
63   [choose_KR] p2K > 0 & p2A > 0 & p2W > 0 & p2R > 0 & p2H > 0 ->
64     (p2A' = 0) & (p2W' = 0) & (p2H' = 0);
65   [choose_KH] p2K > 0 & p2A > 0 & p2W > 0 & p2R > 0 & p2H > 0 ->
66     (p2A' = 0) & (p2W' = 0) & (p2R' = 0);
67   [choose_AW] p2K > 0 & p2A > 0 & p2W > 0 & p2R > 0 & p2H > 0 ->
68     (p2K' = 0) & (p2R' = 0) & (p2H' = 0);
69   [choose_AR] p2K > 0 & p2A > 0 & p2W > 0 & p2R > 0 & p2H > 0 ->
70     (p2K' = 0) & (p2W' = 0) & (p2H' = 0);
71   [choose_AH] p2K > 0 & p2A > 0 & p2W > 0 & p2R > 0 & p2H > 0 ->
72     (p2K' = 0) & (p2W' = 0) & (p2R' = 0);
73   [choose_WR] p2K > 0 & p2A > 0 & p2W > 0 & p2R > 0 & p2H > 0 ->
74     (p2K' = 0) & (p2A' = 0) & (p2H' = 0);
75   [choose_WH] p2K > 0 & p2A > 0 & p2W > 0 & p2R > 0 & p2H > 0 ->
76     (p2K' = 0) & (p2A' = 0) & (p2R' = 0);
77   [choose_RH] p2K > 0 & p2A > 0 & p2W > 0 & p2R > 0 & p2H > 0 ->
78     (p2K' = 0) & (p2A' = 0) & (p2W' = 0);
79
80 //who goes first (deterministic)
81   [coin_flip] turn = 0 & p2K*p2A*p2W*p2R*p2H = 0 ->
82     0.5 : (turn' = 1) + 0.5 : (turn' = 2);
83
84 // Actions for p1
85   [p1_K_K] turn = 1 & p1K > 0 & p1_stun != 1 & p2K > 0 ->
86     Knight_accuracy : (p2K' = max(0, p2K - Knight_damage)) & (turn' = 2)
       & (p1_stun' = 0) +
87     1 - Knight_accuracy : (turn' = 2) & (p1_stun' = 0);
88   [p1_K_A] turn = 1 & p1K > 0 & p1_stun != 1 & p2A > 0 ->
89     Knight_accuracy : (p2A' = max(0, p2A - Knight_damage)) & (turn' = 2)
       & (p1_stun' = 0) +
90     1 - Knight_accuracy : (turn' = 2) & (p1_stun' = 0);
91   ! .. 189 lines skipped ..
92   [p1_skip] turn = 1 ->
93     (turn' = 2) & (p1_stun' = 0);
94
95 // Actions for p2
96   [p2_K_K] turn = 2 & p2K > 0 & p2_stun != 1 & p1K > 0 ->
97     Knight_accuracy : (p1K' = max(0, p1K - Knight_damage)) & (turn' = 1)
       & (p2_stun' = 0) +
98     1 - Knight_accuracy : (turn' = 1) & (p2_stun' = 0);
99   [p2_K_A] turn = 2 & p2K > 0 & p2_stun != 1 & p1A > 0 ->
```

```
100     Knight_accuracy : (p1A' = max(0, p1A - Knight_damage)) & (turn' = 1)
        & (p2_stun' = 0) +
101     1 - Knight_accuracy : (turn' = 1) & (p2_stun' = 0);
102   ! ... 197 lines skipped ...
103   [p2_skip] turn = 2 ->
104     (turn' = 1) & (p2_stun' = 0);
105
106 endmodule
107
108 formula p1_sum = p1K+p1A+p1W+p1R+p1H;
109 formula p2_sum = p2K+p2A+p2W+p2R+p2H;
110 label "p1_wins" = p1_sum > 0 & p2_sum = 0;
111 label "p2_wins" = p1_sum = 0 & p2_sum > 0;
```

Appendix D

RPGLite Medals

Medal	**Description**(Bronze, Silver, Gold)
Knight Enthusiast	Play games with the Knight (10, 20, 40)
Knight Veteran	Win games with the Knight (5, 10, 25)
Archer Enthusiast	Play games with the Archer (10, 20, 40)
Archer Veteran	Win games with the Archer (5, 10, 25)
Rogue Enthusiast	Play games with the Rogue (10, 20, 40)
Rogue Veteran	Win games with the Rogue (5, 10, 25)
Healer Enthusiast	Play games with the Healer (10, 20, 40)
Healer Veteran	Win games with the Healer (5, 10, 25)
Wizard Enthusiast	Play games with the Wizard (10, 20, 40)
Wizard Veteran	Win games with the Wizard (5, 10, 25)
Barbarian Enthusiast	Play games with the Barbarian (10, 20, 40)
Barbarian Veteran	Win games with the Barbarian (5, 10, 25)
Monk Enthusiast	Play games with the Monk (10, 20, 40)
Monk Veteran	Win games with the Monk (5, 10, 25)
Gunner Enthusiast	Play games with the Gunner (10, 20, 40)
Gunner Veteran	Win games with the Gunner (5, 10, 25)
Frequent Flyer	Log in on consecutive days (5, 10, 25)
Fearsome Warrior	Increase your skill level (250, 750, 2000)
Goliath Slayer	Win against someone with higher skill than you (250, 500, 1000)
David Vanquisher	Win against someone with lower skill than you (250, 500, 1000)
Streaker	Win games in a row (3, 5, 10)
Cream of the Crop	Be in high percentiles for skill (50, 80, 95)
Friend to Many	Use all characters several times (5, 10, 20)
Addict	Play games (20, 50, 100)
Firm Favourite	Play with a single character (50, 75, 100)
Untouchable	Win games with a character at full health (8, 15, 30)
Close Call	Win games with only 1 health remaining (8, 15, 30)
Speedster	Win games in under 1 hour (20, 50, 100)
Challenge Accepted	Accept challenges (10, 25, 50)
Community Outreach	Match with active users (25, 50, 100)
Patient Practitioner	Matchmake using the waiting list (25, 50, 100)
Challenging Individual	Send out challenges (10, 25, 50)
Eager Beaver	Respond to an opponent's move in under a minute (500, 750, 1000)
Vengeance	Get your revenge on your enemies (10, 25, 50)
Fast Learner	Win against somebody with a higher S2 skill than you (250, 500, 1000)
Low-Hanging Fruit	Win against somebody with a lower S2 skill than you (250, 500, 1000)
Player For All Seasons	Increase your seasonal skill level (250, 500, 1000)

Table D.1: Medals in RPGLite application during season 2 shown with their description and values required for bronze, silver and gold variants (S2 refers to season 2).