Aspect-Oriented Modelling

Tom Wallis

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

Declaration of Originality

I certify that the thesis presented here for examination for a PhD degree at the University of Glasgow is solely my own work, other than where I have clearly indicated that it is the work of others. The thesis has not been edited by a third party beyond what is permitted by the University's PGR Code of Practice.

Printed name	*Signature*
Tom Wallis	

Acknowledgements

Regrettably, I haven't had time to write this acknowledgements section. I care about it and want to do it right — so I'm saving time to edit the contents of the thesis, and will write this properly this during the wait for my viva.

Thanks to Obashi Technology Ltd, who funded this work.

Abstract

Aspect-oriented programming is a software engineering paradigm used to modularise parts of a program which are difficult to separate using other means. It does this using aspects: combinations of program modifications and places to apply them. While it has received some academic interest, aspect orientation has seen lukewarm adoption from industry and its practical benefits are not well demonstrated. The paradigm lacks a use-case for which aspects are particularly well-suited.

One such use-case may be in producing and maintaining codebases for research purposes. In many fields, research is conducted with the aid of software models. Changes to these models are delicate: they may invalidate results, add complexity to a codebase, or absorb researchers' time. These changes *could* be represented as aspects, but the paradigm is yet to be applied to codebases for scientific models. We propose that aspects are particularly well-suited to describe these changes and that aspect-oriented modelling may ease model maintenance.

This thesis investigates the viability of aspect-oriented modelling for simulation purposes. An aspect-oriented programming framework is implemented which addresses criticisms of the paradigm, and contributes new kinds of aspects which are useful for describing changes to models. With this tool, a case study of aspect-oriented modelling is constructed using a model of a real-world mobile game and its players' activity. This forms the foundation of three experiments. They demonstrate that aspects can be used to successfully augment models, can add new behaviours and parameters to models, and can be reused across different models in some cases. As these contributions invite new research in many fields, the thesis also enumerates the possibilities enabled for others researching aspect orientation, simulation & modelling, and research software engineering, as well as the methodological implications for researchers whose hypotheses are encoded within software models.

Table of Contents

Chapter 1

Introduction

This thesis is about two distinct areas of study and what happens when they are married together. First, it is concerned with aspect-oriented programming: a software engineering paradigm which has received some academic interest, but comparatively little industry use or practical demonstration of its benefits. It applies some extra program logic (called "advice") to places where that program logic is required (called "join-points"); the combination of each is an "aspect". Integral to the paradigm's design is its property of "obliviousness", where the implementation of a join-point does not indicate whether an aspect applies to it. Obliviousness simplifies parts of the codebase changed by aspects, as they can be written without including special aspect-oriented programming logic. Aspect orientation has been well researched, but a specific context for which it is better suited than its competitors is yet to be found. Its tooling and theory are well established, but it isn't the "tool of choice" for any particular problem. Critics cite obliviousness as a potential reason for this: reasoning about a program is difficult if it could be altered by any other part of the program. However, a use-case where aspect-oriented programming *is* well-suited would allow the existing theory and tooling to be put to use.

Second, it is concerned with how researchers write programs to support their work. Many fields rely on software models — programs which represent or emulate a subject of study

— when conducting an investigation or experiment. This software is subject to unique requirements, such as the ability to produce reproducible results, to represent hypothetical systems or behaviours which may not have been represented before, and an architecture supporting future studies (potentially conducted by other research teams) which may look to extend that software in ways which are hard to predict, as they encode novel ideas for research. The design and maintenance of these codebases is unique as a result of these requirements.

Later chapters motivate and demonstrate that aspect-oriented programming is well-suited to be used as a tool for maintaining models, particularly in a research context. We suggest that aspects can represent units of change to an existing model. This allows research software engineers to develop experimental models as descriptions of the differences between a system under study and a hypothesised one, by encoding the differences between them as aspects. As later chapters discuss, the properties which make the paradigm difficult to apply in industrial codebases can be a strength in the context of research software engineering: as a model is oblivious to the application of aspects, removing those aspects is trivial. One can imagine that different models could be constructed by composing a "base" model with different sets of aspects. Failed experiments would not incur technical debt, or necessitate the careful removal of code related to that experiment.

Aspect-oriented modelling may be the application for which aspect-oriented programming is uniquely well-suited, because obliviousness is a unique property of the paradigm. However, aspects have not been used to date to represent units of change to scientific models, and so the viability of the technique is unknown. This thesis shows that aspect-oriented modelling is feasible by demonstrating that aspects can be used to represent change to a modelled system. It investigates the use of aspects to introduce new model parameters and actor behaviours, and shows that aspects as units of change can be successfully applied to different systems in certain cases.

This introductory chapter proposes the problem to be solved in research software engineer-

ing more thoroughly in Section 1.1, and proposes an aspect-oriented solution in Section 1.2. The latter section also describes aspect-oriented programming in more detail, and narrows the scope of the types of models within the scope of this thesis. Section 1.3 notes the research questions and key contributions found in later chapters, and the structure of the rest of the thesis is described in Section 1.4.

1.1 The Problem to Solve

Consider a research software engineer, responsible for the development and maintenance of a well-adopted model of a system involving human behaviour, such as a public transport system, the spread of a disease in a pandemic, or a software development team's day-to-day work. As the model's use spreads and users' requirements broaden, how might our engineer manage the maintenance burden of their codebase? What tools and techniques might they make use of to ease their task? Their tool might have originally modelled the interactions of individual developers in a team (to extend the latter example), and offered parameters for development methodologies or team sizes; subsequent researchers might look to run simulations of the team with biased behaviour such as illness or tiredness, or different experience levels in the team, or communication breakdowns between individuals, or the impact of a change in management or project direction. What then?

Typically, if a required change would alter the model significantly, researchers looking to adopt the tool for new purposes would fork it (assuming the source is available to them under a permissive enough licence) and would make whichever modifications suited their needs. Other researchers might make their own forks. If the change is small, or if the new behaviour can be enabled in configuration, then it might be merged into the original tool. The contribution is thus disseminated to the research community who adopted the original tool without the need to migrate to a fork. However, if a future team wanted to research a combination of factors — say, simulating the impact of a change of management on teams

with different levels of experience and communication quality — they would repeat the process, producing another fork with their own implementation of this particular combination. As modifications are built on top of each other, the logic for each new behaviour must be interwoven with that of the original model in any scenario where those changes cannot be developed as independent modules for the original codebase.

Such a codebase would be increasingly difficult to maintain. For example, the abstractions used to modularise it may have elegantly separated different concerns into different modules at first, but abstractions are often domain-specific. As the model becomes increasingly general-purpose, the abstractions used to separate concerns accumulate technical debt in the face of broader use-cases. In addition, the behaviours added by different teams may not make sense when enabled at the same time. In the example of a model of a software development team, a change representing remote work and a change representing the spread of a virus in different office settings are at odds conceptually and both are unlikely to be required in most cases. As a result, the tool's source code risks becoming confusing and unwieldy.

There is therefore an opportunity to improve the design of software models used in research contexts. We identify a need for a technique which allows a model to be altered in an arbitrary fashion so that any changes required of the tool can be represented as separate, optional modules — even when that change introduces or alters logic in the middle of an existing process, and therefore would not modularise using other mechanisms. It should also permit a developer to select the changes they need for their study, effectively composing a model with whichever behaviours they require.

1.2 A Proposed Solution

We identify aspect-oriented programming as a tool which is well-suited to the needs of research software engineers. This section starts with a definition of relevant terms and an

explanation of the scope of the research: the type of system being modelled, "socio-technical systems", are described in Section 1.2.1. This thesis concerns models and simulations of socio-technical systems. Specific definitions of "model" and "simulation" for the purposes of this thesis are given in Section 1.2.2.

Having defined the relevant terms, a high-level overview of aspect-oriented programming is given in Section 1.2.3. Motivations for the construction of aspect-oriented models can then be discussed, as the necessary background has been given; this follows in Section 1.2.4. To give the reader context as to the current research opportunities in the field of aspect-oriented programming — as well as the work required to build an aspect orientation framework suitable for aspect-oriented modelling — some of aspect-oriented programming's current limitations are discussed in Section 1.2.5. A more thorough discussion of the aspect-oriented paradigm and its literature is given later in Section 2.1, and literature on its shortcomings are also described in more detail in Section 2.1.7.

1.2.1 What are Socio-Technical Systems?

The research in this thesis applies aspect-oriented programming to the modelling & simulation of socio-technical systems. A socio-technical system is one composed of people, technology, and the interactions between the two. The term originates in the study of work and organisations undertaken by Trist and Bamforth [139]. Socio-technical systems research continues to focus on organisations and workplaces [111, 5], though the interaction of people and technology is also studied more broadly in areas such as degraded modes [70], resilience engineering [63], and responsibility modelling [90].

The involvement of human behaviour in the system's dynamics makes these systems useful subjects for aspect-oriented modelling: variations in the behaviour of an individual or an entire group is may be considered a concern which cuts across different actors (or actors in models of different systems if the behaviour is exhibited in more than one model). To this

end, models of learning are developed in Chapter 6 which represent learning as a cross-cutting concern. In addition, datasets of users' interactions with computer systems can be collected to empirically verify socio-technical models of their behaviour. This is also useful for the research in later chapters: Chapter 5 describes the design, implementation, and data collected from the release of a mobile game, another example of a socio-technical system.

The technique of creating models using aspect orientation to modularise particular behaviours or dynamics is not uniquely useful in the domain of socio-technical systems modelling. We suspect that it can be generalised to the behaviours and dynamics of other systems, too. However, the paradigm is demonstrated as a socio-technical modelling technique for the purposes of setting bounds on our aims. It is not feasible to verify the technique's appropriateness in *every* type of system, but it is feasible to investigate the technique as applied to a *particular* type of system. Also, aspect-orientation has received attention in the business process modelling research community [15, 13, 14], setting a precedent for its broader use in modelling other socio-technical systems too.

1.2.2 Simulations & Models

The field of simulation & modelling concerns the building of models of some system and the simulation of that system using the model. We have found (anecdotally) that the difference between the two can be vague; some definitions are provided to avoid confusion.

In this thesis,"model" will be used to refer to some representation of a system or subject of study, or an abstraction of it. In this sense, a model can be a concept, a physical model, a diagram, and so on — in this thesis in particular, "model" will be used with the meaning: *"a software representation of a system or other subject of study"*. An example of such models are OBASHI models [146], which are software representations of how data flows through socio-technical systems. Another is CovidSIM [124], which models the spread of COVID-19 through a population by representing proportions of that population at different points of

a pandemic who are susceptible to infection, exposed to the virus, presently infected, or recovered from infection.

The term "simulation" also requires definition, and throughout this thesis will generally refer to the execution of a model. More formally put, "simulation" is used to refer to *"the emulation of the processes within a system or other subject of study"*. A simulation typically generates data, such as a record of system states over time or a log of actions taken by some actor within that system. One example of such a simulation could be the execution of a model of stock trading where agents mimic the activity of real-world traders to study effective trading behaviour [25]. The software artifact represents the process of trading; executing that process mimics trading itself, and so simulates the activity by emulating real-world behaviours.

As the research in later chapters is primarily concerned with the simulation & modelling of socio-technical systems, the subject of a simulation or model should be assumed to be a socio-technical system unless otherwise specified. When models are discussed in the context of the research conducted in later chapters, they refer specifically to models constructed for the purposes of simulation. Models designed for purposes other than simulation may also prove to be suitable for the use of aspect-oriented modelling, such as non-executable models of business processes or dataflow [146, 13, 16]. However, models designed for simulation purposes are the focus of later chapters to limit the scope of the research at hand.

1.2.3 What is Aspect-Oriented Programming?

Aspect-oriented programming originated in Kiczales et al.'s work at Xerox Parc [81]. The motivation for the paradigm was the modularisation of "cross-cutting concerns": parts of a program which are not directly related to the fulfilment of its requirements but are common throughout it. Cross-cutting concerns generally don't relate to their neighbouring logic semantically. Rather, they fulfil an ancillary task, such as logging. This suggests it is a suitable candidate for modularisation: the single responsibility principle [96] suggests that software

with multiple concerns should be refactored into separate modules. However, traditional techniques are unsuitable for the creation of separate modules in these situations.

Aspect-oriented programming introduces a new pattern which modularises cross-cutting concerns. This is achieved by separating them from a program's core logic and rewriting them as modules ("advice"), the combination of some implementation (an "aspect") and places in a program to apply it ("join-points", a set of which are defined by a "pointcut") — which are later "woven" back into their intended positions by ensuring the aspect's logic is included somehow at its corresponding join point. This allows high-level behaviours such as logging, synchronisation, or memory safety to be factored away from a program's logic and into separate units. Unlike traditional modules, which are included by the code which requires them, a "weaver" composes advice and the code they should be applied to, with neither requiring knowledge of the other. This separation is core to aspect-oriented programming's design, and is referred to as "obliviousness".

Aspect-orientation allows for existing codebases to be structured in new ways, but its notion of modularising cross-cutting concerns is general and can be applied in other fields too. For example, business process modelling languages have been adapted to accommodate behaviours or processes modelled using advice [13, 129, 15]. Some researchers suggest using aspect-oriented programming to separate concepts in experimental software, such as experiment and observation, to mirror the setup of a traditional experiment in research-specific codebases [55]. Concepts originating in aspect-oriented programming have also seen use outside of software engineering [21, 13, 129, 15].

This thesis explores modelling changes to behaviour using advice, and is applied to the modelling of socio-technical systems in particular. Socio-technical systems partly consist of the behaviour of human agents within a system, which can exhibit variations contingent on environmental factors and other states. This variance might be expressed differently in different agents in the system. In later chapters, we demonstrate that these behavioural

variations can be separated from a generic model into advice, with the advantage that the generic model's codebase is simplified and any particular behaviour of interest in a particular study can be modelled by weaving those behaviours at runtime. Advice, as demonstrated in later chapters, is a flexible and useful tool for composing models.

1.2.4 Motivations for Aspect-Oriented Modelling

A researcher who maintains a model of some system they are studying would have many potential use-cases for aspect-oriented programming. Some are discussed here to motivate the use of aspect-oriented programming in a research software engineering context.

First, a researcher looking to study new behaviours for actors in their model would need to implement those behaviours: however, the original behaviour of the model may be useful as a control or point of comparison in their study. They may "fork" their codebase to produce two models (one unmodified, and one with their new behaviour), but this requires them to maintain another codebase every time a new behaviour is studied. A single codebase containing behavioural changes as optional modules would reduce the overhead of maintaining that codebase: changes to the original model could be made once, instead of in every fork.

Some changes do not modularise using existing techniques [81] and opportunities to do so depend on codebase being changed. For example, an epidemiological model which represents interactions between people in a bar would yield different possible changes depending on the implementation of the bar's physical layout, distances between people, places where they may interact, and the nature of those interactions. The software itself would yield different opportunities to implement changes as modules depending on the programming language, paradigm, and abstractions used.

However, *modifications represented as aspects can be applied independently of these factors*. This is a consequence of aspect orientation's unique property of obliviousness, which is that

aspects can be applied to an existing codebase regardless of its implementation, because it does not need to include any special structure or logic for aspects to be applied[39]. This means that the epidemiological model mentioned earlier could represent new types of interactions between modelled agents regardless of whether the codebase was designed to easily support new kinds of interactions, by encoding them as aspects.

A second scenario where researchers may look to represent model changes as advice would be in the instrumentation of that model for experimental observation [55]. For example: researchers may develop a model of a city's transport infrastructure to measure commute times which outputs measurements of simulated commuters' journey duration. Other researchers — or future projects conducted by the same research team — may look to extend that model to also measure the *expense* of those commutes, to contrast the time and money spent commuting from different suburbs. While simulated commuters' behaviour may not require modification, the measurements made of their commutes would. These observations can be made by weaving aspects which record additional information about modelled commutes without requiring modification of the original codebase, which is potentially onerous (as described in this subsection). This design for an experiment implemented as software mirrors experiment best practices in other fields, where observational apparatus is separated from a subject under study so as not to bias results [55].

A third scenario where aspect orientation is useful relates to the second: instrumenting several models in the same way to study how those models compare. Aspect-oriented programming's obliviousness allows researchers to apply advice to *any* codebase, including those which were not developed with aspect orientation in mind. For example, comparisons of multiple models could be performed in the same way by applying the same aspects to each codebase to observe their states during execution. Previous studies which compared software models of physical phenomena such as the formation of galaxies [83] or the radiation emitted by them [125] contrast the outputs of those models but could not observe their state during execution. More detailed measurements could be made by instrumenting each model for

observation during their execution. This could be achieved by applying aspects to model codebases to take the same measurements of every model being compared regardless of their architecture, taking advantage of aspect-oriented programming's property of obliviousness.

Another motivating use-case for aspect-oriented modelling is that researchers could use aspect-oriented programming to design experiments. For example, consider a study which seeks to compare the impact of several political policies on a country's economy, environment, and health to find a set of policies which optimise them. Policies could be represented as changes to a model of a country as it presently exists, such as World3 [98]. By implementing n policies as aspects, up to 2^n models can be produced by applying every combination of aspects to the country's model. The optimal set of policies can be discovered by comparing the outlook of the country with each combination of policies applied. This has several benefits over a study where a model is developed with policies included directly in the model codebase. For example, the study can be applied to other countries by changing the underlying country without risk of compromising the implementation of any policy, as their implementations are independent. Extending the study is also simple: adding policies to the study would only require writing new aspects, rather than maintaining a potentially large codebase. This also avoids the risk of introducing bugs to the implementations of other policies, as well as avoiding the introduction of bugs to the model of the country itself.

These motivating cases are all made possible due to aspect-oriented programming's unique property of obliviousness [39]. These results therefore cannot be achieved using other modularisation techniques. Researchers who build, maintain, or modify models may have a need for aspect-oriented modelling in any of the scenarios described, and aspect-oriented programming may be a useful tool for research software engineers.

1.2.5 Challenges in Aspect Orientation Today

The aspect-oriented paradigm has some drawbacks. Identifying these means they can be addressed when applying aspect-oriented programming to a new domain, and add context which motivates research presented in later chapters. While they are discussed in detail in Section 2.1.7, a quick summary is given here as background for the ease of reading later chapters and for the discussion of the contributions made in this thesis, which follows in Section 1.3.

The paradigm's philosophy of obliviousness can make codebases difficult to comprehend. When reading the implementation of a function or feature, it's not possible to know whether advice is being applied to it elsewhere by inspecting the relevant source code in isolation. If aspects are applied to it, the behaviour of that code might be altered. Accurately understanding a program's control flow can therefore be challenging as a consequence of aspect-oriented programming's fundamental design properties [28, 116].

Other limitations of aspect-oriented programming concern aspects themselves. Aspect orientation tooling typically allows for advice to be woven before join-points, after them, or around them (in effect, both before and after). However, this requires aspects to treat their join-points as black boxes: no changes can be made *inside* the join-point. This could prevent some use-cases of aspect-oriented modelling, such as the instrumentation of several models to compare them (discussed in Section 1.2.4). This would only be feasible if the join-points available in a codebase allowed aspects to collect the information required. If those join-points are treated as black boxes and the state to observe only exists inside them, the instrumentation suggested would not be possible. Therefore, aspect-oriented modelling frameworks would ideally support new kinds of join-point which advice can be applied inside.

Lastly, while a variety of toolsets exist for aspect-oriented programming, few studies appear to have been undertaken confirming that its expected benefits actually materialise [117]. Some

scepticism around its practical benefits have been raised [132, 116, 28]. An application of aspect orientation to a new domain ought to demonstrate that the technique is of practical benefit, rather than a theoretical curiosity.

1.3 Contributions

1.3.1 Research Questions

The research questions investigated in the following chapters are:

> **RQ1** *Can models of systems more accurately reflect their subjects by weaving aspects which represent improvements?*
>
> **RQ2** *Can advice be used to faithfully introduce behaviours or parameters into a model which were not originally present in it?*
>
> **RQ3** *Can advice be used as a portable module, such that aspect-oriented improvements to one model can be woven into another without loss of performance?*

These are motivated following a review of relevant literature in Section 2.4.1. Clarifications of specific terms and phrases are also provided in Section 2.4.1 as it contains a thorough discussion of the research questions.

1.3.2 Summary of Contributions

In developing tooling for aspect-oriented modelling and investigating the technique's feasibility, this thesis makes multiple contributions to the relevant research communities, including but not limited to answering the above research questions. These are summarised as:

Tooling for Aspect-Oriented Modelling This thesis presents a redesigned and re-implemented version of a tool, PyDySoFu[1], originally prototyped for some prior work,[2] which contributes both a tool which can be employed for aspect-oriented modelling and a weaving technique designed for legibility when applying aspects to models. It can be applied to existing Python code without modifying the codebase it is applied to and introduces minimal dependencies when added to a project.

Dataset describing Socio-Technical System Interaction A dataset describing 370 players' interactions with a mobile game released for research purposes called RPGLite. The design, implementation, and data obtained from the game is presented, and the dataset collected via RPGLite is used to design experiments presented in this thesis.

Demonstration of Aspect-Oriented Model Enhancement Aspects are used to augment a model of RPGLite gameplay and demonstrate that aspect-oriented enhancements can alter a model to improve it. This is achieved by measuring the similarity of simulated players' preference of characters in the game with that of real-world players by correlating the frequency with which simulated and real players select different characters. In particular, we show that a model can be augmented to synthesise data with the properties of the real-world dataset.

Using Aspect-Oriented Enhancements to Identify Hypothesised Player Behaviour A model is presented which represents how players learn which characters are likely to win games of RPGLite. We demonstrate that this can be tuned to specific real-world players' behaviour using the dataset collected, resulting in player-specific models of learning which characters are likely to win games. Properties of learning of specific real-world players are identified by fitting a learning model to their RPGLite gameplay data.

Investigation into Aspect Portability We investigate whether aspect-oriented changes to a model can be ported from one system to another, taking advantage of aspect-orientation's

[1] PyDySoFu is an acronym for *PYthon DYnamic SOurce FUzzer*.
[2] See Chapter 3 for a discussion on related research undertaken prior to starting this PhD.

modular nature.

Exploration of Opportunities Enabled by Aspect-Oriented Modelling New research opportunities are yielded by this novel technique in simulation & modelling. As in other PhD theses where contributions carry potential for many pieces of future work [94], the avenues for future research are broad enough that identifying and enumerating them is a substantial piece of work and constitutes an additional contribution.

1.4 Thesis Structure

The rest of the thesis is structured as follows.

Chapter 2 surveys the project's relevant literature and identifies specific research questions in the field which the thesis addresses. Some earlier work precedes the research presented in this thesis. To delineate this from the contributions presented in later chapters, Chapter 3 surveys the state of the research project before this PhD began.

Having surveyed the literature, identified research questions, and established the starting point of the research, Chapter 4 describes the re-design and implementation of the aspect-oriented programming framework PyDySoFu, named PDSF3. Other technical contributions follow in Chapter 5, which describes the design and implementation of RPGLite, a mobile game developed for research purposes, as well as the data collected from it.

Later chapters explore the application of aspect-oriented programming to simulation & modelling codebases. Three experiments are constructed using a model of RPGLite to answer the research questions mentioned in Section 1.3.1. The experiments involved share the majority of their technical foundations and methodology, but are used in different ways to research different facets of aspect-oriented programming as applied to simulation & modelling. For this reason, their common foundations and methodology are explained in Chapter 6. The specifics of each experiment — and the results of those experiments — are described in their

own chapter, Chapter 7, alongside answers to the research questions they address.

Finally, the encoding of behaviours and model properties as aspects yields novel research approaches, a discussion of which is omitted in all relevant literature reviewed to date. We therefore investigate the possibilities enabled by aspect-oriented modelling in Chapter 8 thoroughly, in the same vein as other theses making similar contributions [94].

Chapter 2

Relevant Literature

This thesis presents an aspect-oriented approach to simulation & modelling and to experiment design, with tooling to support this. An empirical assessment of the paradigm's application in the domain of simulation & modelling is also contributed in later chapters. This chapter motivates those contributions, through a review of related literature.

The chapter starts by discussing aspect-oriented programming in Section 2.1, providing an overview of aspect-oriented programming's history, design philosophy, and some aspect orientation frameworks of note. Criticisms of and alternatives to aspect-oriented programming are also discussed. Existing applications of the paradigm to simulation & modelling are also particularly relevant to the research in later chapters, and this is described in Section 2.2. Also relevant to the contributions made in later chapters is the representation of variations to processes. This is because aspects are used to represent individual units of change to a model, which may introduce changes to behaviours and the processes which simulated actors follow. A review of literature representing changes to processes is given in Section 2.3. The chapter concludes with a brief discussion in Section 2.4, where the research questions which later chapters address are motivated.

2.1 Aspect-Oriented Programming

2.1.1 Motivations & Philosophy Underlying Aspect Orientation

Modularity is considered a key trait of maintainable, flexible, and legible programs [110, 31]. Modern modular design techniques are often concerned with segmenting a program's logic or data into separate units. The standard approach to designing with modularity in mind in many industrially-relevant languages today is object-oriented programming, where data structures and selected program logic are modularised as classes.

Some concepts in software engineering are not easily modularised by techniques such as object orientation. Guarding against unsafe concurrency usage, manual memory management, and logging are all examples of program components which are not readily modularised using an object-oriented or similar approach, because they occur alongside the logic fulfilling a program's requirements. These parts of a program are common across many modules; however, this common logic does not form traditional modules. Such concerns of a program are termed "cross-cutting concerns" in aspect-oriented programming parlance. Programmers looking to separate cross-cutting concerns into separate modules seek to address two problems they give rise to [81]:

① *"Tangling"*, where program logic essential for a program's intended purpose is intermixed with ancillary code addressing cross-cutting concerns, thereby making essential logic more difficult to maintain;

② *"Scattering"*, where program logic for cross-cutting concerns is strewn throughout a codebase, making maintenance of this code more difficult.

The existence of cross-cutting concerns is expected to make maintenance of both ancillary program logic and a program's core logic more difficult because of these traits. To address them, Kiczales et al. [81] introduced the notion of aspect-oriented programming. The paradigm is

simple to define through its unique software concepts:

- A *"join point"* defines some point in a program's execution (usually the moment of invocation or return of some function or method) where additional logic is required.
- *"Advice"* implements some behaviour such as logging, which which can conceptually happen anywhere in program execution (i.e. what's defined would typically represent behaviour which cuts across many parts of a codebase).
- An *"aspect"* is constructed by composing this advice with *"point cuts"*: sets of join points that define all moments in program execution where associated advice is intended to be invoked.
- An *"aspect weaver"* then adds the functionality defined by each aspect by adding the functionality defined by its advice at each join point defined by its point cut.

The implementation of join points, and advice, or weaving is a matter left for aspect orientation frameworks and aspect-oriented languages to define. In employing the technique, aspect-oriented programming aims to separate cross-cutting concerns into aspects, removing the aforementioned repetitive code from the logic implementing a program's functional behaviour so that additional pieces of functionality — logging, authentication, and so on — can be maintained in only one place in a codebase (thereby simplifying their maintenance and comprehension), and remaining program logic can be understood and maintained without the overhead imposed by the previously tangled cross-cutting concerns.

Kiczales et al. [81] see these engineering concepts as universal throughout business logic, motivating the aspect-oriented approach for the first time. The authors present an implementation of aspect-oriented programming in Lisp, and compare implementations by way of SLOC count in an emitted C program to a comparable, non-aspect-oriented implementation, with two examples (its use in image processing and document processing). They find the idea — which they note is "young" and describe many areas where research might help it to grow —

can successfully separate systemic implementation concerns such as memory management in a way that reduces program bloat and simplifies implementation. It is noted that measuring the benefits of their approach quantitatively is challenging.

Tooling followed the theoretical work presented by Kiczales et al. [81] with a demonstration and subsequent technical description of AspectJ, a Java extension for aspect oriented programming [61, 79]. AspectJ was introduced to satisfy the research community's need for a tool with which to demonstrate the aspect-oriented paradigm in case studies. The tool is intended to serve as "the basis of an empirical assessment of aspect-oriented programming" [79]. The library makes use of standard aspect-oriented concepts: Pointcuts, Join Points, and Advice, bundled together in Aspects. They define "dynamic" and "static" cross-cutting, by which they refer to join points at specific points in the execution of a program, and join points describing specific types whose functionality is to be altered in some way. Their paper describes only "dynamic" cross-cutting, but presents tooling support, architectural detail of its implementation, and the representation & definition of pointcuts in AspectJ. AspectJ is compared to other tools for aspect orientation and related decomposition-focused paradigms, and the authors are explicit about their approach being distinct from metaprogramming[1] in, for example, Smalltalk or Clojure.

Filman, Friedman, and Norvig [39] isolate properties of aspect orientation which they assert are definitive of the paradigm. Specifically, they claim that aspect-oriented programming should be considered in terms of "quantification" and "obliviousness":

> AOP can be understood as the desire to make quantified statements about the behaviour of programs, and to have these quantifications hold over programs written by oblivious programmers. […] We want to be able to say, "This code realises his concern. Execute it whenever these circumstances hold."

[1] Metaprogramming is the practice of writing programs which write or re-write other programs (or themselves) by treating code as data and performing transformations on it. An example of metaprogramming is the mutation of programs for testing purposes, termed "mutation testing". Another is the "macro system" feature of such as Racket or Common Lisp, which allows developers to write a procedure which takes a part of a program as input and returns another program fragment which replaces the original one.

These concepts became core concepts in aspect orientation literature alongside "tangling" and "scattering". Filman, Friedman, and Norvig give no concrete definition of the terms in their original paper and cite no sources for the terms. This thesis therefore adopts the definitions:

Quantification: the specification of points in a program in which that program should change;

Obliviousness: the property of a codebase that it contains no lexical or conceptual reference to advice which might be applied to it, and of the programmer of a target program that their code may be amended by an aspect programmer.

Filman, Friedman, and Norvig write about aspect orientation as a paradigm independent of a particular application or implementation. They are therefore able to arrive at conclusions about the paradigm in the abstract and can identify concerns for future researchers in the field and design goals for developers of aspect-oriented tooling. They note:

> Better AOP systems are more oblivious. They minimise the degree to which programmers (particularly the programmers of the primary functionality) have to change their behaviour to realise the benefits of AOP. It's a really nice bumper sticker to be able to say, "Just program like you always do, and we'll be able to add the aspects later." (And change your mind downstream about your policies, and we'll painlessly transform your code for that, too.) [39]

Here, Filman, Friedman, and Norvig theorise about the advantages they expect aspect orientation to bring software engineers. However, the notion that obliviousness is a straightforward advantage of aspect-oriented programming has been contested in more recent literature. This is discussed further in Section 2.1.7.

Aside from the benefits of aspect-oriented programming's obliviousness, claims made by Filman, Friedman, and Norvig such as the "painless transformation" of code when a developer changes their mind during the development process — another theorised benefit of

obliviousness — is incompatible with earlier writing on modularity. Yourdon and Constantine [156] assert:

> The more that we must know of module B in order to understand module A, the more closely connected A is to B. The fact that we must know something about another module is a priori evidence of some degree of interconnection even if the form of the interconnection is not known.

Aspect orientation's critics describe similar incompatibilities with existing best-practices [116, 28], as well as the lack of empirical evidence for the benefits of obliviousness [132]. Claims about "better" aspect-oriented systems being more oblivious should therefore be regarded as *suggestions* from the literature. While obliviousness and quantification are useful concepts in discussing research in the field, they also give context for the research community's perspective that obliviousness and quantification are design goals for those using aspect-oriented programming [131, 76, 14] (though Leavens and Clifton [87] suggest they may be best applied in moderation).

Aspect-oriented programming is thus a contentious paradigm which addresses limitations of existing methods using a novel approach, but its design philosophy also attracts sceptics. More specific criticisms of aspect-oriented programming are reviewed in Section 2.1.7; specific implementations of aspect-oriented programming language extensions and frameworks are reviewed first.

2.1.2 Approaches to Weaving Aspects

The design of the weavers in different aspect orientation frameworks and language extensions is pertinent to the research in this thesis, because the research introduced

To review and compare different aspect-oriented programming frameworks, it is important to differentiate their weavers. Many designs for weavers exist. Some weavers introduce aspects when code is compiled, calcifying the program's behaviour and disallowing extension at a

later time. Others weave advice as a program executes. To flexibly support the needs of a developer of a scientific model, this thesis focuses on the latter type of weaver. These weavers are referred to as "dynamic".

Chitchyan and Sommerville [20] present a review of aspect orientation frameworks with dynamic weavers. They compare AspectWerkz [9], JBoss [40], Prose [113], and Nanning Aspects [26] through the lens of the authors' prior work on dynamic reconfiguration of software systems. By comparing different implementations of dynamic weaving, Chitchyan and Sommerville contribute a categorisation of the tools' approaches:

① "Total hook" weaving, where aspect hooks are woven at all possible points;

② "Actual hook" weaving, where aspect hooks are woven where required;

③ "Collective" weaving, where aspects are woven directly into the executed code, "collecting the aspects and base in one unit".

As Chitchyan and Sommerville focus on software reconfiguration rather than the mechanics and design of dynamic aspect weaving, their analysis of the reviewed tools is of less relevance to the work presented in this thesis than their generalisation of dynamic weaving. However, their review discusses the trade-offs of the three approaches identified. Chitchyan and Sommerville propose that total hook weaving allows flexibility in the evolution of a software product, at the expense of the performance of that product; this contrasts collected weaving, which shifts overhead out of the codebase and into the maintenance effort. Actual hook weaving is positioned as a compromise between the two, offering the best approach for none of their criteria but never compromising so much as to offer the worst, either. This suggests merit in a tool designed to flexibly offer any weaving approach appropriate for the task at hand.

Chitchyan and Sommerville note that one could use many of the systems they describe in practice. Though the paper is an early publication in the field, no tool the authors review

offers all three dynamic weaving approaches, and none offers collective weaving alongside either kind of hook weaving.

2.1.3 AspectJ

Following their seminal work on aspect orientation, Kiczales et al. [79] published an aspect-oriented programming extension to Java called AspectJ, as discussed earlier in Section 2.1.1. AspectJ is still actively developed, and is currently maintained by the Eclipse organisation. It is also the most well-adopted tool for aspect-oriented programming today, and its language for specifying aspects is adopted by other tools such as Spring AOP [123]. AspectJ can weave aspects at compile time or when join points are loaded, resolving the trade-offs between collective weaving and total hook weaving by allowing developers to select the weaving method appropriate for their use-case.

The motivations behind developing AspectJ were those of aspect-oriented programming, as the paradigm lacked an implementation at the time. Later aspect orientation research efforts have since been folded into the project, such as AspectWerks' dynamic weaving mechanisms, or have been reproduced within it, such as support within the JVM [50] and late binding of aspects [49]. The popularity of the framework has led it to see some industry adoption [62], although its use in real-world settings has been found to introduce a trade-off between the legibility of a codebase and the ease with which that codebase can be updated [117]. AspectJ's design has influenced other aspect orientation frameworks: for example, the SpringBoot Java framework provides Spring AOP, an aspect-oriented programming implementation which is compatible with the domain-specific language AspectJ uses to describe aspects [123].

2.1.4 PROSE

Another implementation of aspect-oriented programming with dynamic weaving is PROSE [113, 112], a library which achieves dynamic weaving by use of a Just-In-Time compiler for Java.

Popovici, Gross, and Alonso motivate the project be identifying aspect orientation as a possible solution to software's increasing need for adaptivity. Mobile devices, for example, could enable a required feature by applying an aspect as a kind of hotfix, thereby adapting over time to a user's needs. Other uses of dynamic aspect orientation they identify are in the process of software development: as aspects are applied to a compiled, live product, the join points being used can be inspected by a developer to see whether the correct pointcut is used. If not, a developer could use dynamic weaving to remove a misapplied aspect, rewrite the pointcut, and weave again without recompiling and relaunching their project.

Popovici, Alonso, and Gross [112] indicate that some performance issues may prevent dynamic aspect orientation from being useful in production software, but that it presented opportunities in a prototyping or debugging context. The PROSE project explores dynamic weaving as it could apply in a development context, but the authors do not appear to have investigated dynamic weaving as it could apply to simulation contexts, or others where aspect-oriented software does not constitute a product.

Their observation that aspect-oriented programming could be used for the purpose of adaptation or software prototyping instead of modularisation is an example of an alternative use case for aspect orientation suggested in Steimann's critique [132] discussed in Section 2.1.7.

2.1.5 Nu

Rajan et al. propose a new aspect-oriented invocation mechanism, which they call "Bind" [119]. Bind's design is motivated by opportunities to improve modularity from a design perspective: Rajan et al. assert that scattering and tangling can be introduced into a codebase *after* weaving with some weaver implementations. This complicates the use of compiled aspect-oriented code, and the development and execution of unit tests on such a codebase. Bind alleviates this issue by allowing a developer to choose when aspects are applied. This new mechanism is presented as an alternative to the weaving of aspect hooks for load-time registration into

target code in the style of PROSE [113, 112] and the direct weaving of aspect invocations in the style of AspectJ [79].

In order to demonstrate Bind's approach to simplifying post-weave codebases, Rajan et al. [119] present "Nu", an aspect orientation framework written in .NET supporting Bind. Nu's design is explained and its implementation presented, which aim to promote granularity in join point specification. What results is a flexible model for aspect orientation which is demonstrated to satisfactorily emulate many other paradigms and tools, such as aspect orientation in AspectJ and subject-orientation in HyperJ.

Dyer and Rajan [36] give a more thorough technical explanation of the design and implementation of the Bind mechanism and the implementation of the Nu framework. A more technical discussion is presented, in particular on implementation details including optimisation and benchmarking, largely against AspectJ. The implementation discussed is a Java implementation, rather than the .Net implementation presented by Rajan et al. [119]. Many aspect orientation frameworks are language-specific; the existence of Nu's implementation on multiple platforms highlights the work's main contribution in the design of the Bind primitive, rather than the framework itself.

Rajan et al. note that it is "very common in aspect-oriented programming research literature to provide language extensions to support new properties of aspect-like constructs". They look to provide similar extensions to virtual machines, noting that their mechanism is a suitable candidate to introduce aspect orientation directly in a language's virtual machine. They position the project as a general model of aspect-oriented programming which can flexibly represent a variety of existing approaches. Bind fulfils this aim by providing a single mechanism which can be used to achieve many weaving techniques; however, case studies demonstrating that Bind can be used to make aspect orientation more practically effective for software developers does not appear to have been published.

2.1.6 Dynamic Weaving in Embedded Systems

Gilani and Spinczyk [48] observe that while there are different approaches to dynamically weaving aspects, no approach is suitable for an embedded environment. This is due to these systems' low power and limited memory. Gilani and Spinczyk propose a framework for these situations through which weavers can be assessed for their suitability in a given environment, or generated from a set of desired features.

Gilani and Spinczyk define families of weavers by grouping the environments they can be suitably applied to, separating them in particular by their trade-off between dynamism and resource use. There is an implication that dynamism and resource use are broadly proportional, presumably because even a carefully crafted "actual hook weaver" or JIT-compiled "collective weaver" in the parlance of Chitchyan and Sommerville [20] carries runtime overhead by virtue of the weaving mechanism used. A "collective weaver" embeds aspect invocations directly into their join points, which eliminates the need for an intermediary hook invocation or additional compilation. The embedded systems of interest to Gilani and Spinczyk have memory in the range of ~30kb, where these overheads could represent significant resource use.

Aspect oriented programming's criticism is sometimes that it doesn't know what it "aims to be good for" [132], and the dynamic weaving of aspects may not be good for resource-constrained environments. As Gilani and Spinczyk identify a trade-off between dynamism and resource economy, a highly constrained environment would not be able to take advantage of dynamic weaving by definition. Additionally, because the anticipated benefits of aspect-oriented programs are not observed in practice and others have noted the inefficiencies introduced by dynamic weaving [36, 20], the paradigm is difficult to apply to any use-case where conservative resource usage is a concern. We might observe that researchers should seek other contexts in which to apply aspect oriented programming.

Support for this observation can be found in criticisms of aspect-oriented programming from Steimann [132], whose position that alternative use-cases of aspect orientation should be explored is discussed in Section 2.1.7. Examples of alternative uses include Popovici, Gross, and Alonso's proposal of using aspect orientation for software prototyping, the proposal presented by Gulyás and Kozsik [55] that simulation & modelling is a more appropriate field, or Cieslak et al. [21]'s use of aspect orientation in modelling plant growth using L-systems. These proposed uses of aspect orientation involve a different resource economy than that imposed by embedded systems. A discussion of aspect orientation's use in simulation & modelling can be found in Section 2.2.

2.1.7 Criticisms of Aspect Orientation

As referenced in Section 2.1.1, the design philosophy underpinning aspect-oriented programming has its detractors. The opposition to aspect orientation is of particular relevance to the work presented in this thesis, which should be understood within the context of some perceived weaknesses of the paradigm. The following chapters introduce contributions which address some criticisms of aspect-oriented programming, so broad criticisms of the paradigm — and the work published in awareness of those weaknesses — motivates some research presented in later chapters.

Constantinides, Skotiniotis, and Stoerzer published an early critique of aspect-oriented programming [28] which notes similarities between the paradigm's core concepts and `GO-TO` statements. Constantinides, Skotiniotis, and Stoerzer uses the comparison to demonstrate fundamental issues with the paradigm's philosophy: use of `GO-TO` statements is widely accepted to be bad practice [31], and their infamy has become an in-joke for academics and language enthusiasts [23]. They note that `GO-TO` statements disorientate a programmer by way of "destroying their coordinate system" — referring to a developer's technique for navigating and understanding code — which results in uncertainty about both a program's

flow of execution and the state of a program at different points of its flow. Constantinides, Skotiniotis, and Stoerzer uses the comparison to `GO-TO` statements to question whether aspect-oriented programs can have a consistent coordinate system for developers: `GO-TO` statements lack obliviousness as they are visible in disrupted code, wheras aspects are not represented structurally within a program due to their oblivious design. This complicates a developer's understanding of where and how flow is interrupted. They draw comparison between aspects and `COME-FROM` statements, an April Fools' joke where a claimed improvement over `GO-TO` is developed by removing the latter's structural component [23], and conclude that existing engineering techniques provide similar benefits without a trade-off in program legibility. In particular, Dynamic Dispatch is identified as a preferred alternative.

Steimann makes a similar but more thorough critique of aspect-oriented programming [132]. They express concern that the popularity of aspect-oriented programming — which was nearly 10 years old at time of publication — was founded on the perception that it would solve real-world engineering problems, yet no proof existed that it was effective in practice. Steimann notes that most papers are theoretical in their discussion on tooling, that examples were typically repetitive, and that the community's discussion was concerned more with what aspect orientation could be used for than whether it worked in practice. They present a comparison between aspect orientation and object-orientation, where aspect orientation's claimed properties and principles are examined in detail, and the impact on software engineering is reasoned about from a skeptical perspective. They compare the paradigms' specific claims that they both support improved modularity against classic papers on the subjects.[2]. Steimann presents a philosophical examination of aspect orientation and assesses the paradigm against its purported merits, discussing whether we should accept the claims made by the aspect-oriented programming community. Aspect orientation's promise of unprecedented modularisation is presented as unfulfilled, and Steimann reflects critically on the state of aspect orientation research at the time. However, they conclude by proposing that the aspect-oriented approach

[2] In particular, they compare against literature by Parnas [110].

could have a legitimate alternative use-case — other than as a *general-purpose* technique for modularising cross-cutting concerns — where it could be shown to be effective empirically.

Similar sentiments are shared by Przybylek [116], who examines aspect-oriented programming in the context of language designers' quest to achieve maintainable modularity in system design. They frame the design goals of aspect orientation as being to represent issues that "cannot be represented as first-class entities in the adopted language". Przybylek questions whether the modularity offered by aspect orientation can really be said to make code more modular. In particular, they distinguish between lexical separation of concerns and the separation of concerns originally discussed by Dijkstra [32]. They assess principles of modularity — modular reasoning, interface design, and a decrease in coupling — and find that the aspect-oriented paradigm can detrimentally impact the expected benefits of proper modularisation in a program by "violating basic software engineering principles" such as information hiding and structured programming. They suggest that aspect orientation's claimed benefits are a myth repeated often enough to be believed.

2.1.8 Alternatives to Aspect Orientation

Aspect-oriented programming's goal of modularising cross-cutting concerns is shared by other paradigms. As discussed in Section 2.1.1, the work first introducing aspect orientation by Kiczales et al. [81] makes note of similarities to reflection, metaprogramming, and program transformation. They also observe that other disciplines have introduced "aspectual decomposition" independently. Some modelling & decomposition techniques which are related to aspect orientation are discussed here.

System Diagrams

The example of pre-existing aspectual decomposition by way of diagramming given by Kiczales et al. [81] is in physical engineering. To give a concrete example from their description,

differing types of diagrams when engineering a system such as thermal and electric diagrams of a heater are described as "aspectual" because of the modular nature of the diagrams; though there might be many diagrams of different kinds, they compose together to give an overview of the system being designed.

Similar diagramming techniques have independently arisen in other domains since. The Obashi dataflow modelling methodology[146] by Wallis and Cloughley models all possible paths of dataflow through "B&IT" (business and IT) diagrams, where business-specific concerns (people, locations, groups, and business processes such as payroll, stock-check or budgeting) are modelled alongside IT concerns such as applications supporting business processes and the software and hardware infrastructure supporting them. Modelling dataflows in this way allows for a comprehensive understanding of assets and business processes. However, in order to understand how data flows between specific assets within a B&IT, sub-graphs ("DAVs", or Dataflow Analysis Views) denote specific pathways through which data flows between source and sink assets. Alternatively, a B&IT can be viewed as a composition of all possible DAVs within an organisation. Dataflows are therefore broken into different diagramming techniques and specific business concerns can be described independently of others, even if these concerns interact in their dataflow pathways (and, therefore, cutting across each other). Obashi therefore allows for the aspectual decomposition of business processes, through the description of an organisation by individual dataflow analysis views, which compose into an overall model of a system in a B&IT diagram. Obashi models are an instance of aspect orientation which were designed for simplicity and comprehension[146, 126], but trade this for domain-specificity.

Other diagrams of systems, called "bigraphs", were produced by Milner [101]. Bigraphs are a category-theoretic process calculus able to simulate petri nets, π-calculus, and λ-calculus [102] but are also used in the simulation & modelling of real-world systems such as networks [12] and ubiquitous computing [7]. A key property of bigraphs are their "ports", which denote possible points of composition a given model can have with others. Bigraphs' compositional

property is reminiscent of the use of aspect-orientation for composition [55, 21]. While the existence of a port means bigraphs cannot be composed obliviously, they are a powerful tool for formal modelling and are used similarly.

Metaprogramming

Metaprogramming is identified as a precursor to aspect orientation by the original paper on the paradigm [81]. Research into the use of metaprogramming to simplify the composition of software modules was undertaken at a similar time by Keller and Hölzle [77], who introduce a technique they name "Binary Component Adaptation" (BCA). Their research into BCA the difficulties involved in the integration of software components, particularly considering their evolution over time where components are re-used with differing requirements. By modifying binaries directly, incompatibilities between a program and one of that program's dependencies can be resolved by way of mutating either after compilation. Their implementation defines a representation for the modification of pre-existing Java class binaries, the output of which can be verified as also being valid Java class binaries. Keller and Hölzle claim that BCA allows for dynamic modification of programs with little overhead. They believe BCA is unique in its combination of features, which include engineering concerns such as typechecking code which is subject to adaptation and its obliviousness to source implementation, as well as guarantees that modifications are valid even for later iterations of the program subject to adaptation.[3]

Metaobject protocols describe the properties of an object's class (including, for example, its position within a class hierarchy) in an adaptable manner [80]. Espák [38] note that this technique can be used to implement aspect orientation, therefore providing at a minimum the same functionality, though they achieve this through reflective programming techniques and are designed with metaprogramming as a primary goal as opposed to modularisation of

[3] BCA shares concepts with aspect orientation, but is also a promising technology for the introduction of process variance; see Section 2.3 for a discussion.

cross-cutting concerns [80, 137].

Engineering Techniques with Related Aims

Multiple-dispatch, where methods on objects are chosen to be run based on the properties of the parameters passed at point of invocation, allows for oblivious decomposition without the need for a weaver [35], although this does not support the goals of aspect orientation in totality. For example, a programmer might want their program to exhibit differing behaviour when methods are called with differently-typed arguments, which is supported by multiple dispatch. However, they might instead want their program to exhibit some additional behaviour whenever a method is invoked, such as logging, but might not want to implement logging alongside the rest of their method implementation for clarity or length reasons. Multiple dispatch therefore offers comparable but different functionality to a software engineer.

Engineering patterns such as decorators provide similar functionality to aspects [46], in that cross-cutting concerns can be separated into their own module, but they differ in their approach to obliviousness: decorators annotate areas of a codebase they are applied to, and therefore do not offer obliviousness as aspects do. An example of a decorator and its application to a simple function is given in Fig. 2.1. Decorators allow for additional functionality to be written as a separate module and applied as a wrapper around a function definition. A function with a wrapped definition is replaced by a function returned by the decorator, which takes the original definition as an argument [47]. Additional logic can therefore be simply applied before or after a function. While this achieves a similar effect to aspect orientation — as logic is added before and/or after the original logic of a function — its design principles are different as annotating the function's definition directly marks it as altered, and so the original definition cannot be oblivious to the change.

```python
# Decorators take functions as arguments and
# return a function to replace the one they are passed.
def print_invocation(f):
    # Construct the function to return, which will print
    # when it invokes `f` and when `f` returns.
    def wrapper(*args, **kwargs):
        print(f"Executing {f}")
        print(f"\tArgs: {args}")
        print(f"\tKeyword args: {kwargs}")
        return_val = f(*args, **kwargs)
        print(f"Returning from {f}")
        print(f"\tReturn value: {return_val}")
        return return_val
    # Return a new function wrapping `f`
    return wrapper

# When the decorator is applied, the function it is applied to is
# passed as its argument. The decorator's return value replaces the
# function which would otherwise be defined here.
@print_invocation
def add(a, b):
    return a + b

add(5, 6)
# Prints:
"""
Executing <function add at 0x104978f40>
        Args: (5, 6)
        Keyword args: {}
Returning from <function add at 0x104978f40>
        Return value: 11
"""
```

Figure 2.1: A code snippet showing the definition and use of a decorator in Python. Decorators follow a design pattern which provides similar functionality to aspects, but do not conform to aspect-oriented programming's philosophy of obliviousness.

2.2 Aspect Orientation in Simulation & Modelling

This thesis is concerned with the use of aspect orientation in simulation & modelling codebases; it is therefore necessary to review related research in simulation and modelling. This section reviews literature contributing aspect-oriented tooling and the philosophy of aspect-oriented experiment design in the simulation & modelling community.

2.2.1 Suitability of Aspect Orientation in Experimental Codebases

Gulyás and Kozsik [55] observed that, in the study of complex systems through software models, the codebase produced serves two purposes: the experimental subject, and the observational apparatus used to conduct the experiment itself. They use this framing to identify that the observation of a program's state constitutes a cross-cutting concern. In order to reduce scattering and tangling in experimental codebases, Gulyás and Kozsik theorise that aspect-oriented programming may separate the logic of observation from the core logic of a simulation or model.

The approach is demonstrated via their Multi-Agent Modelling Language (MAML). MAML was designed to enable aspect-oriented simulation of agent-based models and was implemented using the SWARM simulation system [60], a domain-agnostic framework for agent-based simulation. While SWARM takes the form of a collection of C libraries, MAML is implemented as a domain-specific modelling language, allowing it to support aspect-oriented programming as a language feature.

MAML's aspect orientation effectively makes use of Observer patterns to measure a simulation's state. This enables a researcher to observe an experiment without the necessary logic being scattered or tangled in their domain model. Gulyás and Kozsik find that aspect-oriented programming provides an intuitive and straightforward method by which simulated experimental systems can be composed. They note that MAML's simplicity and its philosophy on

modelling are more “satisfactory” than Swarm’s standard approach. The team report that MAML’s implementation was more complex than initially conceived: the `patch` unix tool was intended for use as their weaver, though the team eventually developed a transpiler from MAML to Swarm instead (which they name `xmc`.). The deciding factors for the development of a custom transpiler are not discussed.

In addition to presenting tooling for aspect oriented simulation, Gulyás and Kozsik [55] theorise about the potential benefit of applying aspect orientation to simulation & modelling. They observe that there may be benefits beyond improvements to modularity and a reduction of tangling & scattering. In particular, their work discusses specific scenarios in which the *type* of separation of concerns offered by aspect orientation is desirable, and the engineering approach to achieving the aim reasonable. This distinguishes the work in comparison to most other research on aspect orientation. Many papers describe the expected benefits by simply drawing from existing literature and the claims made in Kiczales et al. [81]’s first paper on the subject, rather than drawing on empirical evaluation [132, 116, 117] as discussed in Section 2.1.7.

This is a rare example of philosophy on aspect oriented programming’s suitability in a particular use-case. Gulyás and Kozsik’s work is of particular importance in this review, because the domain they identify as particularly suitable is simulation & modelling, which is the subject of this thesis. That aspect orientation might be well suited to separating observer and experiment partially motivates research in later chapters, which investigates the use of aspects to *modify* simulated behaviour rather than simply observing it. Further discussion follows in Section 2.4.

2.2.2 Aspect Orientation in Discrete Event Simulation

Aspect-Orientated Implementations of Simulation Frameworks

Chibani, Belattar, and Bourouis [17] observed that simulation frameworks and the codebases built upon them can exhibit cross-cutting concerns such as event handling, resource sharing, and the restoring the state of a simulation run. They investigated the introduction of aspect orientation to Discrete Event Simulation (DES) frameworks to address tangling and scattering that may arise from the cross-cutting concerns they identified. They contribute a discussion of aspect-oriented programming's potential application to DES codebases, and detail the avenues available for research in the field. Chibani, Belattar, and Bourouis [17] identify Japrosim [1, 6] — a DES framework previously developed by the research team, which was designed to model domains through process interaction in Java — as an example of an existing framework which exhibits the cross-cutting concerns they describe.

Later, Chibani, Belattar, and Bourouis [19] identified opportunities for the use of aspect orientation in simulation tooling, aiming to increase "modularity, understandability, maintainability, reusability, and testability" by applying the paradigm [19]. They present a case study of an application of aspect orientation to simulation tooling by identifying cross-cutting concerns in Japrosim, a discrete event simulation framework, and propose an aspect-oriented redesign of the tool using AspectJ. Chibani, Belattar, and Bourouis describe Japrosim's existing object-oriented design, followed by aspect oriented variations of some design elements, including concurrent process management and in Japrosim's graphical animation features. A similar survey of areas in which Japrosim's source might benefit from the application of aspect orientation is presented by Chibani, Belattar, and Bourouis in earlier work [18]. In both cases, the main contribution noted is the design itself. Evaluating the main improvements between the presented aspect-oriented design and the existing object-oriented one is left to future work.

Chibani, Belattar, and Bourouis [19]'s later work presented an implementation of their proposal and provided a quantitative evaluation of that implementation. The quantitative evaluation provides measurements based on Martin [95]'s object-oriented design metrics and demonstrates a greater independence of packages in their aspect oriented version of Japrosim than in the original. However, the intended aim of aspect orientation is not to decouple existing packages, but to isolate those packages' cross-cutting concerns into new ones. It is therefore unclear that their quantitative evaluation achieves its improvements as a result of aspect orientation. No further discussion of their results is provided, and it is possible that the improvement is due to the rewriting necessary in their maintenance of the Japrosim source, rather than due to their use of aspect orientation specifically.

Similarly to Chibani, Belattar, and Bourouis [19, 17, 18], Aksu, Belet, and Zdemir observe that there are advantages in adopting aspect orientation when developing a simulation framework [3]. Examining the DES framework Simkit [51], they motivate two different applications of aspect orientation: a refactoring of the framework itself to better manage cross-cutting concerns within its codebase; and aspect-oriented tooling for use by modellers who represent cross-cutting concerns within their models. Opportunities for improvements in production and development are discussed, and some implementation notes are detailed, although no concrete implementation or evaluation is provided; the work instead proposes design alterations, and the authors "leave it as a future work *[sic]* to explore the usability and efficiency" of aspect orientation used idiomatically alongside Java's existing reflection offerings. The existence of multiple attempts to refactor differing simulation packages with aspect orientation indicates potential for modellers in the use of aspect-oriented patterns, but the real-world utility of the techniques are omitted. As is common in aspect orientation literature, Chibani, Belattar, and Bourouis and Aksu, Belet, and Zdemir both defer to the general claims that aspect orientation improves modularity of cross-cutting concerns and can eliminate code smells such as tangling and scattering. Chibani, Belattar, and Bourouis do present some quantitative evaluation, but this is flawed as previously described.

Research projects applying aspect orientation to the implementation of simulation frameworks [19] fail to provide a case study which evaluates their technique with real-world examples. However, case studies do exist which evaluate aspect orientation's suitability in maintaining model source code.

Ionescu et al. identify an increased demand for computational power in simulation execution on supercomputers [67]. Existing known-good models might be unsuitable for the extreme requirements of code efficiency modellers contend with, but running the code in different environments requires modifications to make the code suitable in the environment. These modifications are subject to regulations and introduce risk of a reduction in quality during maintenance. The authors propose an aspect-oriented solution to the problem, where aspects modify the simulation codebase with minimal overhead. An implementation of a real-world model for disaster prevention is presented, and assessed both by comparison against an equivalent non-aspect-oriented codebase and by assessment of the aspect-oriented variant's scalability and reliability in both cluster and multi-cluster environments. They find that a comparative analysis of generated code and of their simulations in various configurations both indicate that their simulation's aspect-oriented implementation is suitable for use in disaster prevention, implying that aspect orientation could be suitable in scenarios with comparable requirements.

This work provides evidence that aspect-oriented programming is suitable for supporting modifications to simulations and models. This contrasts the lack of evidence for the paradigm's success in improvements to modularity and maintainability [116, 28], and supports Steimann [132]'s suggestion that aspect orientation might be suitable for other uses — as well as the suggestion made by Gulyás and Kozsik [55] that the paradigm is well suited to the requirements of simulation & modelling codebases.

As it is a rare example of aspect-oriented case studies, the methodology employed by Ionescu et al. is important to highlight. Their evaluation measures code quality improvements, which are claimed using similar measurements in early aspect orientation research [81]. Their code analysis makes use of significant lines of code as a core metric, which doesn't reliably reflect code quality. As Rosenberg [122] explains:

> […]the best use of SLOC is not as a predictor of quality itself (for such a prediction would simply reduce to a claim about size, not quality), but rather as a covariate adjusting for size in using another metric.

Although Ionescu et al. [67] evaluate code quality, the methodology employed to measure improvements is unreliable.

Improvements in code quality are those which have come under scrutiny by the critical papers reviewed in Section 2.1.7. The results presented by Ionescu et al. do not satisfy critics' requests for empirical evidence of improved code quality. This does not impact their aspect-oriented models' viability: their study demonstrates that their models can be augmented to support new supercomputing environments without lack of performance. The models described in this work satisfy that aim: their models are also evaluated in their performance. Model performance is a priority in supercomputing contexts, where execution time is financially expensive and energy-intensive. Quantitative evaluation of their models' execution time shows less than 5% slowdown compared to a non-aspect-oriented implementation. Ionescu et al. deem this a reasonable trade-off for the engineering improvements they observe.

Ionescu et al.'s application of aspect orientation to supercomputing & disaster prevention simulations meet their performance requirements and demonstrate a modelling technique which adapts existing models for use in new environments without directly modifying pre-existing source code. This result is notable with regards the contributions presented in this thesis, which similarly aim to augment a pre-existing model without directly modifying its source code — though this thesis' targets model reuse and design simplification rather than

for avoiding the regulatory overhead and financial cost of maintaining models which run on supercomputers.

2.2.3 Aspect-Oriented L-Systems

Aspect orientation is also applied in other simulation paradigms. Cieslak et al. [21] investigated the use of aspect orientation in L-system based simulations. An L-system[89] is defined by a set of symbols, an initial string composed of these symbols, and a set of rules for rewriting substrings. While being a powerful tool for representing fractal structures, they were originally conceived of for modelling in botanical research.

Cieslak et al. note that some details of plant modelling are actually cross-cutting concerns against many plants or families of plants, such as carbon dynamics, apical dominance and biomechanics. To represent these, they introduce a new language to describe plant models which makes use of aspect orientation to represent these cross-cutting concerns. They successfully test the approach by representing carbon dynamics, apical dominance and biomechanics as cross-cutting concerns that are integrated into a previously published model of kiwifruit shoot development. Cieslak et al. hope that these cross-cutting concerns might work in other models too, but this is untested. The use of an aspect developed for use in one model and reused in another seems untested in the community's literature and is a noted omission in the conclusion of this particular work.

2.2.4 Aspect Orientation & Business Process Modelling

Several projects within the business process modelling research community make use of aspect orientation to design modelling languages which produce less monolithic business process models [13, 129] and simplify the composition of models [15]. Business process modelling research is relevant to this thesis' contributions, as business processes are inherently socio-technical and later chapters present tooling for and results in the modelling and simula-

tion of socio-technical systems using aspect-oriented techniques. In addition, some research conducted prior to this thesis developed software engineering processes that are conceptually similar to business process modelling (see Chapter 3). There also exists interest in modelling behavioural variance within the business process modelling community (see Section 2.3), which is relevant to this thesis' concern with the aspect-oriented representation of changes to processes and modelled behaviours.

Charfi and Mezini observed opportunities integrating BPEL, an executable business-process modelling language, with aspect-oriented concepts [15]. This is because when BPEL systems are composed together the logic being composed can lack the flexibility required by BPEL's use cases. The specific use-case examined is web service definition, where changes affecting composition of multiple component parts can affect many areas of a final result, making modification error-prone. The authors specifically seek to support dynamic workflow definitions — "adaptive workflows" — which BPEL's existing extension mechanisms do not sufficiently support but which aspect orientation literature does discuss (see the aspect-oriented programming implementations discussed in Section 2.1). Therefore, they look to construct an aspect-oriented BPEL extension. Using the case study of modelling a travel agency's web services, they create an aspect-oriented extension by first defining how such an extension would be represented graphically in BPEL's workflow diagrams. Further detail is added to arrive at a technical definition with XML representations, weaving mechanics, and eventually the construction of a BPEL dialect, AO4BPEL. The authors find that their pointcut system (which describes join points on both processes and BPEL messages), support for adaptive workflows, and aspect-oriented approach to workflow process modelling make AO4BPEL unique at the time of publication, though related AOP implementations exist in each individual area of their contributions. The work is weakened by brittle semantics around pointcuts, join points, and the temporal nature of workflow modelling. For example, they note that defining contingent behaviour — only applying an aspect conditionally, based on a trace through a simulation of a modelled system — would allow weaving advice only

when model state deems this appropriate. They also call for more generally theoretical AOP research, which mirrors the requests some critics of aspect orientation research make (as noted in Section 2.1.7). The contingent application of model adaptation is a motivating case for some work presented in this thesis; see Chapter 3 for a discussion.

Charfi and Mezini [14] described AO4BPEL in detail and presented a generalisation of the notation developed for it which applies to any graphical workflow modelling language. Accompanying this are some examples of its use building a framework for enforcing certain requirements of BPEL models, and use of that framework to develop aspect-oriented frameworks for enforcing security and reliability within AO4BPEL models.

In later work, Charfi, Müller, and Mezini defined a similar aspect-oriented dialect of BPMN they name AO4BPMN [16], after asserting that the concerns addressed by AO4BPEL [14, 15] in the field of executable process languages also apply to business-process modelling languages, and can be solved similarly. The generalised notation of aspectual workflow models presented in Charfi and Mezini's thesis [14] are applied to BPMN to produce an aspect-oriented language specifically for process modelling, as opposed to executable business process modelling.

Cappelli et al. also note that cross-cutting concerns exist in business process models, and are specifically motivated by monolithic design approaches common in business process modelling languages. Like Kiczales et al., they claim that a lack of modularity in business process models leads to cross-cutting concerns scattered throughout a model [13]. To alleviate the issue, they propose a meta-language, AOPML, which incorporates aspect orientation in a metamodel of business process modelling languages, and instantiate it within their own dialect of BPMN. Using a model of a steering committee as a case study, and separating cross-cutting concerns such as logging, the paper proposes reducing complexity and repetition graphically, thereby in a manner more in keeping with the language design philosophies of popular business process modelling languages, the design and use of which are typically graphical [108, 34, 109]. They note that this is in contrast to other applications of aspect orientation in

business process modelling — specifically AO4BPMN — where aspect definitions written in XML concern not only the advice to be applied but also their relevant join points, as in general programming aspect orientation implementations such as AspectJ. In this way, AOPML exhibits the design philosophy of business process modelling more stringently than does Charfi and Mezini's notation for aspect-oriented workflow modelling.

The difference between Charfi, Müller, and Mezini's approach in designing AO4BPMN [16] and Cappelli et al.'s approach in designing AOPML [13] highlights design decisions taken when introducing aspect orientation in a new domain. There is an opportunity for a domain-specific aspect orientation framework to align its design with the traditions and idioms already present in models within that domain, but doing so may break the traditions and idioms which already exist in aspect-oriented approaches in other domains. Comparing the approaches of Charfi, Müller, and Mezini and Cappelli et al. demonstrates that there may be no clear "best" design approach when blending pre-existing modelling paradigms, such as the graphical modelling languages used in business-process modelling and the abstract concepts of aspect orientation. The discussion around whether it is more desirable to adapt existing design elements of aspect-oriented frameworks to a given domain or adapt that domain's existing modelling traditions, idioms, conventions or syntax to incorporate aspect orientation as it is used elsewhere is outside the scope of this thesis.

New concepts within the design of aspect orientation frameworks are addressed in the business process modelling community. Jalali, Wohed, and Ouyang note that aspect oriented modelling frameworks often do not explicitly model the precedence of aspect application [68]. They address this limitation by defining a mechanism to be used in capturing multiple concerns as aspects, where the invocation of advice must follow a certain precedence. The aim of the work is not to propose tooling around the precedence of aspect application so much as to contribute to aspect oriented design theory, providing a notation for precedence which is broadly applicable. The precedence model is, put simply, that a mapping exists for each application of advice to join point such that the mapping defines an ordering on advice for

that join point. The definition defines "AOBPMN", a formalised dialect of BPMN supporting aspect orientation with precedence. A case study is provided where AOBPMN is instantiated within a coloured Petri net. Their study expands on existing work by research teams led by Capelli [13, 129] and by Charfi [15], in that it develops a mature formalism for and model of aspect orientation as applied to business process modelling. However, Jalali, Wohed, and Ouyang note that their case study is limited in scale. No tooling or evaluation of the practical benefit of their approach is provided.

2.3 Process Variance in Simulation & Data Generation

Research presented in later chapters concerns the application of aspect-oriented programming to the codebase of a model to represent changes within it. In particular, we are interested in representing changes to simulated behaviour. This section reviews techniques where variations on already-modelled behaviour are represented in some way.

2.3.1 Discussion of Variation & Motivations for Variations in Process Models

Difficulties arise when obtaining real-world datasets for many reasons. For example, large empirical datasets are typically produced by organisations which would prefer some level of secrecy around their operations, making publishing those operations for the investigation of research teams unlikely. Researchers collecting these datasets describe a "lengthy process" [33] and explain that traces of real-world processes are hard to obtain because "higher management [can be] worried about the risks" of publishing such datasets [33]. Another factor contributing to the difficulty of collecting empirical datasets is that it is often infeasible to do so, either because there is a need to study the process before it can be put into practice, making synthetic datasets the only option available to researchers. The process may also not yet be fully understood, in which case simulation of many variants of that process can be

useful in aiding understanding. Data may also be difficult to collect because the subject of study is a rare deviation from the codified process: for example, a security violation, which is a deviation to a process which is hoped to exist rarely if ever in the real-world system that would exhibit it [133]. The systems under study may also exist within industries which are typically unwilling to publish datasets describing sensitive internal processes [33] or are unable to harvest that data in a complete, well-formatted manner [66]. In other cases, the dataset itself is of interest to researchers rather than the real-world system that produced it: an example is the generation of datasets to evaluate process mining techniques [140, 2]. The generation of datasets with specific properties is its own field of research; the literature of interest specifically concerns techniques for the modification of a model or simulation to generate those datasets.

The reviewed techniques allow a programmer to include some (separately defined) modification of a process, or represent the modifications of a varied process within simulated output [133, 134]. The benefit of this is that possible changes to a process can be described once and applied to that process where appropriate, resulting in datasets which are affected by the modification. Changes to processes might describe attempts to circumvent security protocols, laziness or confusion in a human actor within the model, or random "noise" to produce synthetic log traces containing aberrations which mimic those found in noisy empirical datasets. In all cases, behavioural variations can be described as some alteration to a process and applied to either a model or the product of that model (datasets or log traces) to represent the same alteration introduced at an arbitrary point in the simulation.

This decouples the expected behaviour encoded in the original model from simulated behaviour, which is obtained by composing the model and behavioural variation using a given technique's method for doing so. This approach to modelling behavioural variation allows the same altered behaviour, which would otherwise be described in many disparate points in a model, to instead be written once and introduced wherever required. The observation that the same variation might appear in many areas of a model, and that the variation can be separated

from the model and introduced where necessary, frames the modelling of these variations in the same way as aspect orientation frames cross-cutting concerns. The work presented in this thesis explicitly applies changes to processes and simulated behaviour as aspects in the same manner. Therefore, although this aspectual connection is not made explicit in much of the literature to date, it is important to review literature on simulation and modelling which modularises these variations; this section reviews that literature. The work reviewed is highly relevant to the contributions in this thesis, in particular because the core motivations of this field are shared by the research presented in later chapters. For this reason, the section leads by discussing those motivations and their relationship to aspect orientation in detail.

2.3.2 SecSY

Research undertaken by Stocker and Accorsi [133] aims to synthesise process logs which are representative of attackers' efforts to compromise the security of a modelled system. Their research project, named "SecSY", is an attempt to address issues arising from the difficulty of retrieving representative log traces for security-critical systems in which attacker activity is present. Logs are developed by process simulation through "well-structured" models, a mathematical property on which transformations were previously defined by Vanhatalo, Völzer, and Koehler [143]. The authors develop a tool for the simulation of a process using well-structured process models, and apply transformations to both the model before execution and the log it produces through the trace of a simulation.

They conclude that their tool is performant and verify it can produce logs representing security violations by way of analysis through PROM, a popular framework for process mining, and pre-defined security constraints on their models. They also note some limitations: log traces cannot be interleaved (due to a lack of parallel simulation of processes), may be incomplete (missing violations), and that mutated models and traces are not guaranteed to be sound by construction. However, they see their proposal as a necessary step in realistic

data generation for business processes. A further weakness of the work is that model and trace modification techniques are limited: processes can be added or removed, but complex graph transformations are only permissible when representable through the composition of the mutation primitives they provide, on which there are only three for processes: swapping AND and XOR definitions of process gateways, and swapping process order. Mutations cannot be applied contingent on the state of a simulation run, for example, representing a decision taken by an attacker based on what had already happened.

In later work, Stocker and Accorsi detail the technical aspects of SecSY, their tool for implementing the generation of synthetic logs which use their technique [133] to represent security violations of security-critical business processes. A Java implementation of SecSY is described, which simulates well-structured models and applies mathematically-defined transformations on the model being simulated (before simulation occurs) and the logs obtained through simulation traces. An improvement on earlier work is that custom transformers can be written. However, a limitation of the original work remains, which is that users cannot easily dictate the degree to which variations are applied.

2.3.3 Cross-Organisational Process Mining

Pourmasoumi et al. also address the need for access to variations on business processes, though for the development of a research field, "cross-organisational process mining" [115]. Process mining can require many process logs, as does the benchmarking and evaluation of process mining techniques. Traces from business processes which are similar but not identical can produce log traces which reflect that similarity, but also reflect the variations in different instances of those processes. These log traces exhibiting variation can be used in the training and analysis of process mining tooling and techniques, which must contend with natural variation present in the execution of real-world traces. To support the field, log trace generation from a variety of process models is therefore required. Such logs are not in adequate supply, as

explained in Section 2.3. The authors' approach to the problem is to present an algorithm for the mutation of business processes, such that simulation against variations of the business process can produce process logs reflecting those variations. Their algorithm makes use of structure tree representations of processes, which models processes as trees and permits conversion to BPMN models and Petri nets [11]. Pourmasoumi et al. make use of this constraint to demonstrate that their models are block-structured, a mathematical constraint on model structure which 95% of models have been shown to comply with [88]. Their contribution is a set of transformations on structure trees and block-structured models, and an algorithm applying these transformations to process models, and a tool which implements it built on PLG, a process log generation tool. They conclude that tools such theirs can be used to generate log traces representing process variation, in such a manner as to satisfy the requirements of the process mining research community.

Pourmasoumi et al. describe a list of model transformations they explain is "not intended to be comprehensive" [115], which makes its full potential unclear. Additionally, processes their tool applies to must be block-structured. The importance of this requirement is that it limits their model transformation technique to business process models. It is not demonstrated that models of processes in other domains satisfy the condition, such as the flow of data [146] or behaviour of human or technical actors in socio-technical systems [150]. Finally, the tool is limited by its lack of capacity to represent variations which are applied contingent on system state. A use case for modelling behavioural variance is to model changes which are impossible to anticipate from the vantage of a modeller, such as a socio-technical system's human actors' mistakes, security exceptions and violations, or corrective actions to mitigate undesirable system state. Pourmasoumi et al.'s tool produces variations on a process model, but modelling behaviour which is expected to vary within iterations of the *same* model is outwith the scope of their project.

2.3.4 Executable BPMN Models

Mitsyuk et al. are motivated by the field of process mining's requirement for datasets of process logs made from well-understood process models, defined in a high-level manner [103]. They demonstrate a technique for generating event logs from BPMN models by introducing algorithms for the direct simulation of BPMN models and the collection of traces from those simulations. While their approach does not support the simulation of all BPMN concepts, notably message passing, they provide a tool which produces log traces for a BPMN model through PROM, a standard tool within the process mining community [141]. This results in their technique providing high-level model simulation through already-standard tooling, meaning adopters of the technique need not rely on dedicated tooling which may not be compatible with other researchers' process mining techniques.

The algorithms presented by Mitsyuk et al. simulate processes described by BPMN models, but don't include any provisions for representing variance. However, the technique could be combined with aspect orientation techniques for BPMN as discussed in Section 2.2.4 [16, 13] to represent alternate behaviour applied contingently. Demonstrating the viability of this approach is an avenue of research beyond the scope of this thesis. However, the motivation of the work mirrors that of other research projects reviewed in this section: a need for synthetic datasets of traces through a process, for use in scenarios where empirical datasets are difficult to obtain.

2.3.5 BPMN Extensions for Process Variation

Machado et al. note that there are operational costs to the inefficient modelling of business processes. Specifically, they note that costs can be replicated across automated processes, and failure to identify such scenarios give rise to these operational costs. This work's core motivation is that the representation of variation in process models would allow for the capture of a replicated process once, with instances of similar processes described as deviations from

that captured blueprint. On this basis, Machado et al. extend BPMN to support the notion of individual processes as transformations of an underlying process, i.e. that a given process can sometimes be expressed as a deviation from some pattern, and is therefore define-able as the composition of that pattern and a variation upon it. Their approach is illustrated through two small, broadly similar business processes initially modelled in BPMN and then represented in Haskell, allowing the authors to demonstrate their representation of variability as process deviation with realistic examples.

While the work presented makes no empirical evaluation of their technique, Machado et al. note that their industrial partner responded positively to the research presented in this publication and that further technical improvements are to be made (support for around advice, and for quantification). They also express an intent to conduct a real-world evaluation in the HR domain. While we are unaware of any real-world evaluation undertaken by this research time to date, some formal proofs that the transformations their tool supports are always well-formed have been contributed [91].

2.4 Discussion

Steimann [132] suggested that aspect-oriented programming *"doesn't know what it aims to be good for"*. This thesis draws on the idea that, as Gulyás and Kozsik [55] suggest, the paradigm applies well in a simulation & modelling context. This discussion concludes the literature review by summarising the observations made and motivating the work presented in future chapters.

This thesis draws on the idea that, in response to Steimann asking what aspect orientation is good for, Gulyás and Kozsik would seem to answer, "simulation & modelling". This discussion concludes the literature review by summarising the observations made and motivating the work presented in future chapters.

Aspect-oriented programming is designed to permit highly modular software engineering in scenarios where cross-cutting concerns are identified, by isolating them as separate modules [81]. The aim in employing aspect oriented programming is to reduce tangling, where a cross-cutting concern is intertwined within a program's main concern, and scattering, where the same cross-cutting concern is re-written at many points in a program's source. Aspects which modularise that concern can be written once, separately from a program's main concern, and re-introduced to each point in a program to which the aspect relates by way of weaving. An aspect orientation framework must therefore be able to quantify the points at which the aspect applies [39]. Aspects specify both advice (the implementation of its associated cross-cutting concern) and the join point defining where in a target program that advice should be woven. Aspect orientation therefore implies that the source aspects are applied to are oblivious to their application [39].

The design of the paradigm is intended to increase modularity in the software applying it [81, 39, 117], and its proponents often claim this modularity as a benefit of the aspect-oriented approaches of their research [48, 15, 13, 68, 17]. However, its critics question the reasoning around these benefits, and note that there is little empirical study into whether aspect oriented programs truly benefit as a result of this modularity [28, 132, 116, 117].

One appropriate application area may be in the representation of behavioural variation in simulation & modelling. The application of aspects to models is already well-studied [3, 17], particularly within aspect-oriented business process modelling [15, 13, 68], where modelling behavioural variation has also seen some prior research [92, 133, 115, 103]. Outside of business process modelling, aspect orientation would reasonably be expected to support the development and observation of models themselves [55].

Research opportunities at the intersection of aspect orientation and behavioural variation in modelling occur because behavioural variation is an example of a cross-cutting concern. Changes to expected behaviour such as laziness, boredom, confusion or misunderstanding can

impact many parts of a process in a socio-technical system, but modelling the variation caused by any one of these requires similar alterations to behaviour in many disparate parts of the model they occur within. Behavioural variations are therefore both scattered and tangled, and constitute a cross-cutting concern which might be well suited to modularisation in aspects. They may also cut across models, requiring a re-implementation for each codebase if not modularised.

Existing aspect orientation techniques and behavioural variation modelling techniques are ill-equipped to take advantage of this alignment. That behaviour changes when it varies is tautological; however, changes supported by existing aspect orientation techniques weave advice before, after, or around their join points, and therefore do not alter the definition itself. In some systems, some variations may be representable as additions inserted before and/or after some other behaviours, but such techniques are unsuitable for representing modifications of the target behaviour itself, or behaviours which should be omitted instead of added. Additionally, join points available to an aspectual programmer may not be granular enough to permit representing the changes they require in such as system, and aspect orientation's principle of obliviousness opposes the modification of target code to make new join points available. Techniques which would directly rewrite target source are typically extremely low-level, and therefore ill-suited to most modelling applications [77]. Other techniques which permit defining changes to processes at a high level may allow a modeller to *describe* intended changes (such as the high-level annotations supported in AOPML [13]), but such techniques are intended for human interpretation, not machine execution for simulation purposes. These techniques permit describing behavioural variation within another process, but only by virtue of the flexibility of natural-language annotation, and are therefore unsuitable for simulation and modelling purposes.

Such techniques also lack executable notion of "state". Real-world behavioural variance can often be contingent on the environment an actor exists within. While variations such as security breaches might be predictable (by identifying weaknesses in existing processes,

for example), variance in socio-technical systems often occurs in the behaviour of human actors. This might be in response to a degraded mode [70], where behaviour naturally drifts to a functional — but undesirable — state, or due to an individual making a mistake, forgetting procedure, or being in a state which alters their behaviour, such as tiredness or drunkenness [107]. A framework for modelling behavioural variation using aspects should therefore apply that aspect to a simulated system contingent on the state of that system at a given point in time. High-level modelling technologies such as BPMN and OPM are executable [103, 34], but it is not trivially evident that executable versions of these technologies are compatible with aspect-oriented modifications of their modelling language [16, 13]. As noted, low-level program transformation technologies are also unsuitable. Techniques for applying variations to models exist, but are unsuitable for simulation (and therefore cannot represent application based on temporal state) [133], produce individual models representing each possible instance of a variation [115], do not support the dynamic weaving of aspects for contingent behaviour [92], or attempt to represent the changes one would expect in the output of a simulation by executing an unmodified simulation and amending its output directly [128]. None of these techniques are suitable for representing a behavioural change which is applied contingent on state. Incidentally, these techniques for modelling behavioural variation also lack support for the alteration of a process definition, or changes "inside" a process definition, as discussed earlier, which also makes them unsuitable for simulated behavioural variation.

Finally, it should be noted that Ionescu et al. [67] produce a case study of using aspect-oriented programming to model a system. While their evaluation makes use of unreliable metrics, their research alters models without significant loss of performance. The modification made to their model does not change the dynamics of the model itself, but ports an existing model to a different platform. Their success supports the suggestion made by Gulyás and Kozsik [55] that aspect-oriented programming may be useful in the development of simulations & models.

2.4.1 Research Questions

A tool which dynamically weaves definitions of change to a model and which is capable of expressing changes *within* a join point rather than before or after it prompts a demonstration of the benefits achieved by that tool. Application of behavioural variation, and the ability to define changes to processes specifically, would fulfil the opportunity to be found in marrying aspect orientation and simulation & modelling. However, the introduction of a modification to a model may break its representation of its real-world analogue, making such a model difficult to reason about. Additionally, the proposed tool is made more complex to reason about due to its capacity to rewrite any join point's definition. Though aspect-oriented programming literature often lacks case studies demonstrating the benefits of the paradigm, it is particularly important to investigate whether such a tool could produce *realistic* models, and whether the expected benefits of aspect orientation as applied to the model hold in practice. Further, the research should demonstrate that units of model change written as aspects are re-usable when modelling other systems. If so, aspects would represent units of modifications to models which cut across different model codebases.

For these reasons, this literature review motivates the following research questions, which are addressed in later chapters:

> **RQ1** *Can models of systems more accurately reflect their subjects by weaving aspects which represent improvements?*
>
> **RQ2** *Can advice be used to faithfully introduce behaviours or parameters into a model which were not originally present in it?*
>
> **RQ3** *Can advice be used as a portable module, such that aspect-oriented improvements to one model can be woven into another without loss of performance?*

It is important to clarify some terms used in these research questions. The "accuracy" of a model is its similarity to its subject in the dimensions by which the subject is being modelled. For example, consider a model of the rates of infection of a virus developed to study the proportion of a population which is likely to be infected with that virus at a given point

in time. If the model's predictions of rate of infection are reflected in those measured in the real world, it would be said to accurately model rates of infection for that virus. While in this case the model might not account for other concerns — such as the geographical spread of the virus — it nonetheless models its intended subject accurately and is therefore an accurate model of infection rates. This concept is distinct from a model's faithful representation of its subject. A faithful representation is used here to mean that what is represented accurately reflects what is *intended* to be represented. For example, consider a scenario where researchers develop a mathematical model of infection rates which correctly reflects their theory of the spread of a virus, but their theory does not reflect how that virus spreads in the real world. In this case, their model faithfully represents the theory as intended but does not accurately model infection rates. The model would not faithfully represent their theory if a mistake was made in developing the model, such that it does not represent what it was intended to. Faithful models may not accurately represent a system under study, and accurate models may not faithfully represent what they are intended to represent.

To address these, new tooling is produced in Chapter 4 which offers new types of aspects. These aspects weave their advice *within* their targets, rather than before or after them. A system is presented in Chapter 5 which is a suitable subject to construct models of in the pursuit of investigating aspect-oriented simulation & modelling. That model — and the aspects applied to it — is described in Chapter 6. As the experiments which investigate the research questions described earlier use the same model, aspects, and methodology, these common foundations are explained together in Chapter 6. The specific designs of those experiments and their results are then discussed in individual sections within Chapter 7. Before discussing these responses to the literature, related work which predates the contributions of later chapters is explained. Chapter 3 distinguishes contributions in this thesis from some related work which precedes it.

Chapter 3

Prior Work

An implementation of the main tool developed and used in this thesis, predates the research presented in later chapters. The original tool, named PyDySoFu, is re-implemented with a new design as part of the work in later chapters. As context for the contributions in this thesis, this chapter will describe the state of PyDySoFu before its successor, PDSF3, was developed. Motivations for the original development of PyDySoFu are described in Section 3.1, which are followed by its design and implementation in Section 3.2, and that of related tooling for experiments and simulations in Section 3.3. The chapter concludes with a descriptien of the research undertaken using these tools in Section 3.4, as some results in the representation of behavioural variance using aspect orientation were contributed using these tools which predate the research presented in later chapters and offer important background for the research undertaken in it.

3.1 Motivation for Original Implementation

PyDySoFu [150] was developed with different motivations than those outlined in Chapter 2. These motivations are clarified as context for its design and development.

PyDySoFu was developed for the representation of behavioural variance in socio-technical systems, and was first produced as a proof-of-concept. It was developed with a focus on Python simulations by virtue of the language's widespread use and its flexibility in its modelling of data and process.

The tool was to be applied to models of behaviour in socio-technical systems written in Python, where individual actions of an agent in a system were represented as functions. Actions which could be decomposed further into more granular actions could be defined as functions with sequential calls to the more fine-grained actions. Invocations of low-level behaviours would implement some change to an environment in the model which its modelled behaviour would be expected to incur. Invocations of high-level behaviours, containing the invocations of lower-level behaviours they compose in the model, would therefore apply the combined effect of the collected behaviours they represent. A benefit of this approach to modelling behaviour was that high-level behaviours could implement the "flow" of a behaviour. For example, a behaviour which would be modelled in a flowchart as having some loop could be modelled analogously in the method described through use of primitive control flow operators in Python, such as `for` and `while` loops.

Another benefit of this approach is that the behaviours modelled have a predictable structure which is amenable to metaprogramming. A low-level behaviour's affect could be changed by changing the function definition; more structural changes could be made by altering the flow of less granular behaviours. A simple high-level behaviour containing a series of function invocations (modelling an ordered list of steps in the socio-technical system) can be represented as a literal list of function calls. The contents of such a list is trivially modifiable. Removing an item from a list or truncating it at a certain length, for example, are both achievable in a trivial manner using high-level languages such as Python. Notably, many behaviours can be conceived of which could be represented as high-level behaviours but would not be amenable to a simple list of more granular behaviours, such as a behaviour with a looping quality.

With a mechanism to rewrite either an implementation of a behaviour or a collection of behaviours (in the less granular functions mentioned), modelling in such a fashion could therefore lend itself to semantically simple metaprogramming that could represent real-world variations in behaviour. However, for reasons discussed in Section 2.3, the use of metaprogramming to represent realistic behavioural variations in socio-technical simulations should be able to take advantage of system state. Many real-world behaviours are contingent on environmental state. Real-world actors in socio-technical systems might become tired after lots of work, or proportionally to time of day within a simulation. Therefore, the metaprogramming as described should be performed during runtime, for which no suitable candidate was available. PyDySoFu was developed to fulfil this requirement, so that behavioural variance in socio-technical simulation could be modelled as described and subsequently studied.

3.2 PyDySoFu Implementation

The implementation of PyDySoFu is described here. This is to disambiguate the improvements made to PyDySoFu's implementation in its successor — PDSF3— described throughout Chapter 4. This also lays the foundation for discussion of PDSF3's design, as it retains some design decisions with PyDySoFu.

3.2.1 Weaving Mechanism

PyDySoFu made use of a weaving mechanism which could be categorised as "total weaving" in the parlance of Chitchyan and Sommerville [20]: hooks to apply advice are woven into every possible join point. The library was implemented in Python, which offers a flexible object model PyDySoFu is able to take advantage of in order to weave its hooks. The weaving mechanism of PyDySoFu was eventually factored out into another library, Asp [153]. However, PyDySoFu predates this refactoring — it was rewritten early in this project, and case studies

using PyDySoFu use the older version of the tool (discussed in Section 3.4). For this reason the weaving mechanism *initially* used by PyDySoFu will be described.

Python's object model has three key properties which PyDySoFu takes advantage of:

① Everything in Python is an object, including types, functions, and classes. Properties of Python's objects which can be used for implementing aspect-oriented programming are useful as a result.

② Objects are, in essence, implemented as a dictionary (Python's name for what other languages might call a `map` or `hashmap`) with string keys. All attributes of an object — such as a method on an instance of a class — are values in this dictionary, and their identifiers are the string keys of the dictionary. This means that the expression `someObject.val` is *notionally* equivalent to `someObject.__dict__['val']`, though the subtleties of this mechanism will be explained later.

③ Certain operations on objects such as attribute lookup, addition, or comparison are handled by "magic methods", reserved method names surrounded by double underscores which Python calls on an object to invoke the behaviour of the operation associated with the magic method. For example, the expression `objA == objB` is interpreted by Python as `objA.__eq__(objB)`. Many such magic methods exist, and a deeper explanation is given in Section 4.3.2. These methods can be overridden or specified by a programmer.

PyDySoFu weaves aspect hooks into classes by taking advantage of these three properties of Python. At a high level, PyDySoFu operates by replacing the `__getattribute__` method of a class object with a custom one. `__getattribute__` is the magic method responsible for retrieving an attribute from an object's underlying dictionary, by taking a string argument as an identifier and returning the object associated with that identifier if it exists. The logic of the replaced `__getattribute__` method is diagrammed in Fig. 3.1.

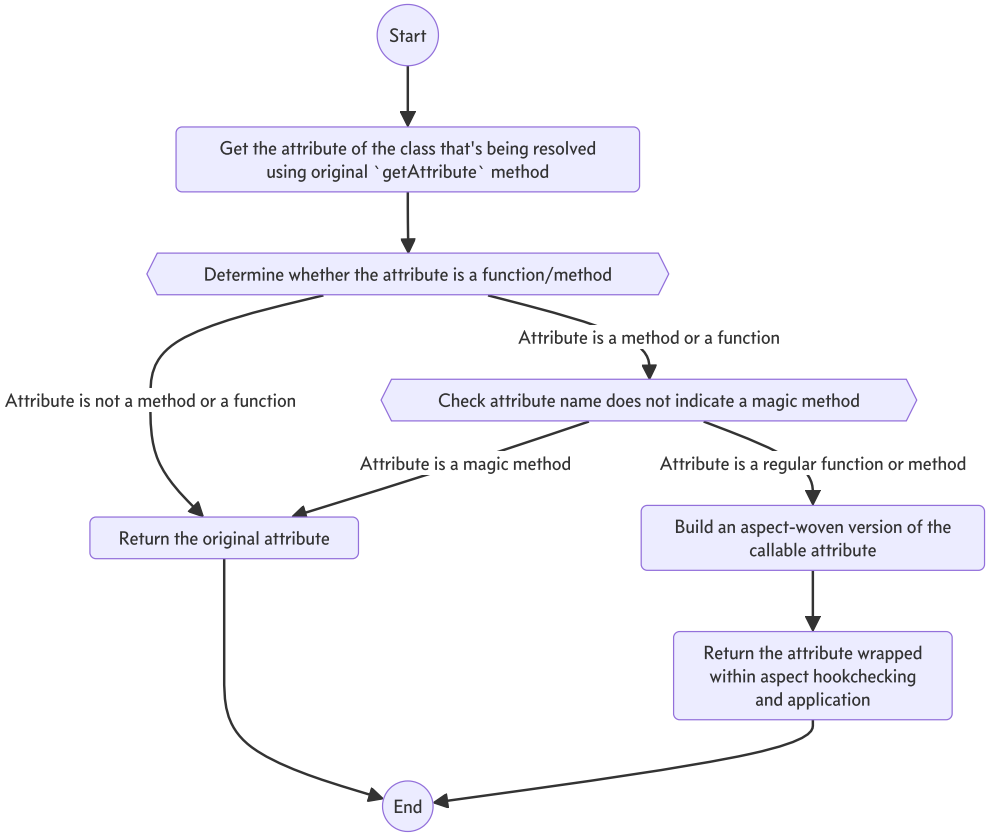

Figure 3.1: UML Activity Diagram of the process implemented by the replacement `__getattribute__` method as implemented by PyDySoFu.

The replacement `__getattribute__` method injected by PyDySoFu also looks up attributes in the relevant object's underlying dictionary, but in addition to retrieving an object's attribute it searches for advice to be applied when performing these lookups, and applies any advice it finds. The replaced attribute lookup logic implements the aspect hook woven by PyDySoFu.

Hooks are applied to a class by way of an invocation to a function, `fuzz_clazz`, which takes a class as a parameter and weaves aspect hooks into that class [152, 153]. `fuzz_clazz` replaces the `__getattribute__` method of the class with a new function object which it constructs. The replacement function object discovers aspects which have been woven, invokes them, and also invokes the target of any woven aspects at the correct time. Without any woven aspects, this has the effect of simply executing a target function; however, in following this process the replaced function object also implements aspect hooks.

The implementation of the function constructing the replaced attribute lookup logic as well as the aspect hook implementation is given in Fig. 3.2, which is edited slightly for legibility. The replacement `__getattribute__` method first makes a call to the class' original `__getattribute__` method to retrieve an attribute when required. If this attribute is not a function or method, it is returned by the woven `__getattribute__` function and the affected class behaves as if it was never altered. However, if an attribute to be retrieved is a method or function, a new function is constructed and returned. This function contains the aspect hooks described earlier. When it is invoked, the function retrieves woven advice, invokes it at the appropriate points, and invokes the attribute it replaced at the appropriate time.

3.2.2 Applying Process Mutations

The development of PyDySoFu was intended to support simulation & modelling research by providing a way of applying program modifications at runtime. It sought to support models where a (potentially non-deterministic) change to the behaviour was required, and enable it to apply that change dynamically during execution in an aspect-oriented fashion. Aspect orientation libraries typically support advice woven before, around, or after a join point; modifying the join point itself essentially allows changes inside its definition, introducing a fourth type of weaving. PyDySoFu achieves this through a special type of "before"-style aspect termed a "fuzzer".

Fuzzers implement transformations on abstract syntax trees. They are implemented as functions which receive a list of AST objects representing the body of a function definition which matches a fuzzing advice's pointcut, and return another list of AST objects, which replace the target function's definition. Any transformation resulting in a valid AST is permitted. A code snippet demonstrating the implementation of this process is shown in Fig. 3.3, edited lightly for legibility.

```
def weave_clazz(clazz, advice):
    """
    :param clazz : the class to weave.
    :param advice : the dictionary of method reference->aspect mappings to apply
        for the class.
    """
    if clazz not in _reference_get_attributes:
        _reference_get_attributes[clazz] = clazz.__getattribute__

    if clazz in advice_cache:
        advice_cache[clazz].update(advice)
    else:
        advice_cache[clazz] = advice

    def __weaved_getattribute__(self, item):
        attribute = object.__getattribute__(self, item)
        if item[0:2] == '__':
            return attribute
        elif inspect.isfunction(attribute) or inspect.ismethod(attribute):
            def wrap(*args, **kwargs):
                advice = advice_cache[clazz]
                reference_function = attribute
                advice_key = reference_function

                if inspect.ismethod(attribute):
                    reference_function = attribute.im_func
                    advice_key = getattr(attribute.im_class, attribute.func_name)

                aspect = advice.get(advice_key, identity)
                try:
                    if hasattr(aspect, 'prelude'):
                        aspect.prelude(attribute, self, *args, **kwargs)

                    if hasattr(aspect, 'around'):
                        result = aspect.around(attribute, self, *args, **kwargs)
                    else:
                        func_args = args
                        if inspect.ismethod(attribute):
                            func_args = (self,) + args

                        result = reference_function(*func_args, **kwargs)

                    if hasattr(aspect, 'encore'):
                        aspect.encore(attribute, self, result)

                    return result
                except Exception as exception:
                    if hasattr(aspect, 'error_handling'):
                        return aspect.error_handling(attribute, self, exception)
                    else:
                        raise exception

            wrap.func_name = attribute.func_name
            wrap.func_dict = attribute.func_dict
            return wrap
        else:
            return attribute
    clazz.__getattribute__ = __weaved_getattribute__
```

Figure 3.2: The implementation of `fuzz_clazz`, a function which replaces `__getattribute__` on Python 2 objects to return callable attributes wrapped within an aspect hook implementation.

```
class FuzzingAspect(IdentityAspect):

    def __init__(self, fuzzing_advice):
        self.fuzzing_advice = fuzzing_advice

    def prelude(self, attribute, context, *args, **kwargs):
        self.apply_fuzzing(attribute, context)

    def apply_fuzzing(self, attribute, context):
        # Ensure that advice key is unbound method for instance methods.
        if inspect.ismethod(attribute):
            reference_function = attribute.im_func
            advice_key = getattr(attribute.im_class, attribute.func_name)
        else:
            reference_function = attribute
            advice_key = reference_function

        fuzzer = self.fuzzing_advice.get(advice_key, identity)
        fuzz_function(reference_function, fuzzer, context)


def fuzz_function(reference_function, fuzzer=identity, context=None):
    reference_syntax_tree = get_reference_syntax_tree(reference_function)

    fuzzed_syntax_tree = copy.deepcopy(reference_syntax_tree)
    workflow_transformer = WorkflowTransformer(fuzzer, context)
    workflow_transformer.visit(fuzzed_syntax_tree)

    compiled_module = compile(fuzzed_syntax_tree, inspect.getsourcefile(
        reference_function), 'exec')

    reference_function.func_code = compiled_module.co_consts[0]
```

Figure 3.3: A code snippet from PyDySoFu's codebase [152], implementing `Fuzzing` aspects and applying fuzzing to a function definition.

As can be seen in Fig. 3.3, fuzzing aspects are implemented by way of "prelude" advice (PyDySoFu nomenclature for advice run *before* a target invocation, inspired by early work on Theatre_AG, which is discussed in Section 3.3.2). Using prelude advice, the `apply_fuzzing` method is called before a target is invoked. This method ensures that the correct function object and key to look up advice are selected (which requires different logic depending on whether the target is a method or a function), retrieves any advice woven onto to the target, and calls the function `fuzz_function` on it. Fuzzing is dynamically applied, as advice is retrieved and applied only when a target is invoked and its prelude advice is executed.

`fuzz_function` retrieves the abstract syntax tree (*"AST"*) of the target, a tree representation of the Python code implementing the target. This is cached in case fuzzing is applied to a target multiple times, resulting in a changed AST every time. A copy is made, and a `WorkflowTransformer` recursively visits nodes of the AST and applies the transformations defined by any discovered fuzzers to function definitions within the AST. The implementation of a workflow transformer is given in Fig. 3.4. After transforming the AST, it is compiled into a Python `code` object and used to replace the code object containing the target's implementation. This has the effect of changing the behaviour of the target function, which now executes the program represented by the transformed AST when it is invoked. As fuzzing is implemented as prelude advice, this invocation is guaranteed to occur presently, meaning that the transformation is guaranteed to occur right before the function is run. A fuzzer transforming an AST can therefore make its transformations conditionally based on the state of the program with little chance that the program state will change afterward.[1]

3.2.3 Limitations

PyDySoFu demonstrated the feasibility of runtime metaprogramming via aspect-oriented programming using a model of software development as a case study. This introduced a novel

[1]Later prelude advice or the initial stages of around advice could theoretically also change program state, meaning programmers must still be cautious about assumptions of program state.

```python
import ast

class WorkflowTransformer(ast.NodeTransformer):
    def __init__(self, fuzzer=lambda x: x, strip_decorators=True, context=None):
        """
        :param fuzzer: a function that takes a list of strings (lines of program
            code) and returns another
        list of lines. :param strip_decorators: removing decorators prevents
        re-mutation if a function decorated with a mutator is called
        recursively.
        """

        self.strip_decorators = strip_decorators
        self.fuzzer = fuzzer

        self.context = context

    def visit_FunctionDef(self, node):
        """
        Applies this visitor's mutation operator to the body of the supplied node
            .
        """

        # Renaming is necessary so that we don't overwrite Python's object
            caching.
        node.name += '_mod'

        if self.strip_decorators:
            node.decorator_list = []

        # Perform visit before applying mutation, to avoid recursive mutations.
        result = self.generic_visit(node)
        node.body = self.fuzzer(node.body, self.context)
        return result
```

Figure 3.4: A code snippet from the PyDySoFu codebase [152], implementing the `WorkflowTransformer` which visits AST nodes and applies fuzzing to them.

type of join-point which could apply advice within a join-point, rather than treating that join-point as a black box. However, its design presents limitations.

There is an overhead involved in running the wrapped `__getattribute__` function for every invocation of a fuzzed class, and also in running aspect hooks for all possible join points even when those hooks are not targets of advice at a given moment. When an attribute is the target of advice, aspects are discovered and applied. However, aspects are discovered by lookup within the scope of the function creating a replacement `__getattribute__` method. This design requires multiple instances of advice weaving to create multiple replacement `__getattribute__` calls, all of which are invoked on any attribute lookup on a target class. A single target of advice which has multiple pieces of advice woven therefore incurs a performance penalty for every piece of advice applied, which must be incurred when any attribute is looked up on the target's class. If the join point defining the target may apply to many classes, each class must incur the same penalty, even if none of their attributes are targets of advice in practice. The original library was developed to demonstrate the feasibility of the idea underlying PyDySoFu (runtime metaprogramming), but the weaving mechanism implemented left room for improvement. A robust implementation with attention paid to reducing this overhead is introduced in Chapter 4.

Similarly, an additional overhead is incurred by a lack of caching of the modifications fuzzers make to function definitions. One can envisage a need for runtime metaprogramming which produces different function definitions at different times: an example could be modelling different degrees of degraded modes introduced to an actor's behaviour in safety-critical systems research [70]. One can also envisage no such need: an example could be minor temporary modifications otherwise permanently made in program maintenance, such as to constants within a function definition, to the format of a function's return value, or adding control flow which exits a function early on a termination condition. The requirement is a product of the tool's use case in different scenarios. In scenarios where the same modification is to be made every invocation, a fuzzer need only be run once; optimisations enabling the

caching of fuzzing aspects' effects would provide better performance in use cases where such a feature is appropriate.

Other aspect orientation frameworks offer support for other types of advice. Handi-wrap and AspectJ both support features related to the processing of exceptions thrown by a program [4, 79] and these features have inspired work into improved exception handling in object-oriented systems [100]. However, PyDySoFu offers no direct support for exception handling. Opportunities to support the feature were therefore available for future revisions of the library to capitalise on; such a revision is also presented in Chapter 4.

A final limitation is that the weaving technique used by PyDySoFu is incompatible with Python3, as replacing `__getattribute__` is not possible in Python's newer version. It was determined that a tool which was of practical use to the simulation and modelling community should be produced which would remain useful to future researchers making modern models; as the existing weaving technique lacked performance, an opportunity presented itself for a complete redesign. The resulting new design is presented in Chapter 4 which makes use of a new weaving technique.

3.3 Additional Simulation Machinery

Other related projects developed tooling for sociotechnical simulation & modelling. FuzziMoss was a project collecting a library of standardised behavioural modifications for use in sociotechnical simulation & modelling; Theatre_AG was a project offering a model of time against which actors within sociotechnical simulations & models could act. While these projects ultimately were not used in producing the contributions of this thesis, they are outlined here as relevant to PyDySoFu as they were originally developed as a suite of simulation & modelling tools to be employed together.

```
def missed_target(random, pmf=default_distracted_pmf(2)):
    """
    Creates a fuzzer that causes a workflow containing a while loop to be
        prematurely terminated before the condition
    in the reference function is satisfied.  The binary probability distribution
        for continuing work is a function of
    the duration of the workflow, as measured by the supplied turn based clock.
    :param random: a random value source.
    :param pmf: a function that accepts a duration and returns a probability
        threshold for an
    actor to be distracted from a target.
    :return: the insufficient effort fuzz function.
    """

    def _insufficient_effort(steps, context):

        break_insertion = \
            'if not self.is_distracted() : break'

        context.is_distracted = IsDistracted(context.actor.clock, random, pmf)

        fuzzer = \
            recurse_into_nested_steps(
                fuzzer=filter_steps(
                    fuzz_filter=include_control_structures(target={ast.While}),
                    fuzzer=recurse_into_nested_steps(
                        target_structures={ast.While},
                        fuzzer=insert_steps(0, break_insertion),
                        min_depth=1
                    )
                ),
                min_depth=1
            )

        return fuzzer(steps, context)

    return _insufficient_effort
```

Figure 3.5: Implementation of Fuzzi-Moss' "missed target" fuzzer

3.3.1 Fuzzi-Moss

Fuzzi-Moss[2] was a library of standard behavioural variations written as fuzzers to be applied by PyDySoFu [136]. It was primarily created for use in a model of the impact of inconsistency in teams' executions of software engineering methodology, which is discussed further in Section 3.4.

Fuzzi-Moss contained utilities and fuzzers such as:

- Two probability mass functions representing the chance that behaviour is to be altered given a length of time and an actor's...:

[2] A backronym for "Fuzzing Models of sociotechnical simulations"

- conscientiousness, a lack of which would increase chance of behavioural adaptation due to lack of effort;
- concentration, a lack of which would increase chance of behavioural adaptation due to distraction

- A `missed_target` fuzzer, which terminated a `while` loop early if activated via a probability mass function of an actor's propensity for negligence. This is shown in Fig. 3.5 as an example of a Fuzzi-Moss fuzzer.
- An `incomplete_procedure` fuzzer, which truncated the steps[3] taken by an actor if activated via a probability mass function representing an actor's propensity for distraction

Stubs also exist in the PyDySoFu codebase for fuzzers which represent a model of confusion and its effects on behaviour, but this remains unimplemented at time of writing and specific details of this model have not been defined. A discussion around the revival of this project in the context of the development of PDSF3 presented in Chapter 4 is given in Section 8.7.

3.3.2 Theatre_AG

Theatre_AG is a project defining a model of time against which actors in sociotechnical models & simulations can act [135]. In the project's overview, it describes itself as…:

"Theatre_AG" is a workflow oriented agent based simulation environment. Theatre_AG is designed to enable experimenters to specify readable workflows directly as collections of related methods organised into Plain Old Python Classes that are executed by the agents in the simulation. All other simulation machinery (critically task duration and clock synchronization) is handled internally by the simulation environment.

The central metaphor underlying Theatre_AG's model of timing is theatrical: actors in a

[3] Represented by lines of code

simulation or model are members of a "cast" (a collection of actors) who enact a "workflow" (simulation steps) in a "scene" (domain model within which the actors interact). Central to the library is its clock: tasks are given durations, and a clock which synchronises all agents' position in time ticks to complete different tasks. The theatrical model Theatre_AG introduces is the context for PyDySoFu's nomenclature for its types of advice[4]: "prelude" advice happens before a task and "encore" advice is invoked afterward as a prelude and encore would be in a literal theatre.

Theatre_AG has been used as the environment in models of TCP/IP, algorithmic trading, the spread of disease [107], and the impact of behavioural variation in software engineering methodologies as described in Section 3.4.

3.4 Example Studies using PyDySoFu for Behavioural Simulation

The viability of encoding behavioural variations as aspects using PyDySoFu has been demonstrated in earlier studies [150, 107]. The study modelled software engineers working to different methodologies of software engineering: waterfall, in which requirements are gathered, software is developed to meet requirements, quality assurance steps are undertaken, and the resulting software is delivered to customers; and TDD, where the development of tests for quality assurance precedes the development of features. The study sought to investigate whether, when software engineers were working suboptimally, there was a difference in the rate of bugs introduced to a program developed under each methodology.

The study began with a "naive" model of software engineers following each paradigm, developed in Python using PyDySoFu, Theatre_AG, and Fuzzi-Moss. Engineers would produce "chunks" of code, which could contain bugs. In quality assurance, engineers were modelled as attempting to identify bugs in different areas of the codebase, fixing them if they were discovered. Developers could commit chunks of code toward features identified through

[4]This nomenclature is also retained by PDSF3.

requirement engineering, which were eventually completed, but could potentially contain undiscovered bugs within chunks of code.

This model was then augmented aspectually using PyDySoFu. Distraction was represented through the truncation of functions representing workflow steps. Developers were modelled with different levels of distraction, affecting a probability mass function (PMF) which would activate when a developer was modelled as being distracted in a given moment. If the PMF activated, the workflow step invoked at that moment was truncated using PyDySoFu. The model showed that developers following the TDD methodology could successfully complete a larger number of features on average than those following waterfall, concurring with the prevailing consensus on the two methodologies.

In replicating the community understanding of the model, the paper demonstrated the feasibility of aspectually augmenting modelled behaviour: the simulation took a naive model with no capacity for analysing errors, and introduced new features of the model supporting an avenue of investigation otherwise impossible with the methodologies represented by the naive model. That the resulting simulation matched the expectations existing within the community gave confidence that the tool could be used to build realistic simulations where some features of a model could be separated from its core codebase.

3.5 Discussion

The existing case study employing PyDySoFu demonstrated that realistic model features could be separated from their core codebase, and gave credence to PyDySoFu's use as a tool for aspectually augmenting models with behavioural variance. It also left many research questions unanswered and tooling flaws unsatisfied, however:

- Aspects were believed to be "realistic" as they represented the expected outcomes of the simulation. However, no real-world data was used to corroborate the claim, and it was

unclear that aspectually augmented behaviour could capture the variations present in real-world human behaviour.

- These aspects also demonstrated variations which were identifiable in the emergent properties of a system (for example, mean time to failure of a software system under development, or successfully completed features). The variations applied to individual developers might have poorly modelled individual behaviour, but produced accurate emergent properties of the system individual developers acted within.

- As discussed in Section 3.2.3, PyDySoFu was designed to be a proof-of-concept which, while successfully demonstrating the potential of aspect-oriented runtime metaprogramming, was also inefficient, feature-incomplete, and lacked compatibility with modern software engineering tooling.

- Models of distraction were adopted from the common library provided by Fuzzi-Moss. However, this model was not applied to other codebases. It remains unclear that Fuzzi-Moss' model of distraction is broadly applicable in other projects: different models of distraction might be required by different researchers. Further, a model of distraction which realistically represents the behaviour of an individual (rather than the emergent properties of the system that individual acts within) might not apply to other systems the individual acts within. Briefly put, The portability of aspectually modelled behavioural simulation has not been investigated, and literature within the aspect orientation community lacks evidence to support a belief in their portability [116, 28, 132].

This leaves opportunities to improve both the tooling offered for aspect-oriented runtime metaprogramming, and the evidence supporting its use to encode behavioural variations in sociotechnical systems. Improvements to the tooling follow in Chapter 4; later chapters propose — and discuss the implementation of — real-world systems which are suitable for modelling using PyDySoFu in Chapter 5, and a study of those systems using aspectually

augmented models in Chapter 6 and Chapter 7.

Chapter 4

PDSF3: A Tool for Aspect-Oriented Modelling

4.1 Introduction

4.1.1 Chapter Outline

The work undertaken in this thesis required an improved implementation of old tooling. PyDySoFu was a proof of concept which could feasibly produce scientific models & simulations but was implemented in a manner which was not optimised for speed, making it a burden for large simulations. It also lacked granularity in the application of its aspect hooks: hooks were applied to entire classes, which forced overhead to be introduced to all attribute lookups, even when those attributes were never intended to be used as join-points. Most importantly, it was not compatible with Python3. PyDySoFu manipulates Python2 objects, but Python3's objects have a changed structure which replaces their underlying dictionaries with special classes. These enforce read-write protections on attributes which PyDySoFu relied on. Python2 lost official support shortly after the work described in Chapter 3 was undertaken, and was not a suitable platform to build tooling on.

A new version of PyDySoFu with support for weaving aspects in Python3 was therefore needed; its revised implementation is named PDSF3. Section 4.2 motivates in detail the de-

velopment of PDSF3 by identifying opportunities to improve PyDySoFu's design. Section 4.3 describes approaches which were considered in the implementation of PDSF3. The implementation of import hook weaving is discussed in Section 4.4. Other features of PDSF3 which were developed to improve PyDySoFu's maturity as a tool are given in Section 4.5.

4.1.2 Contributions

Implementing PDSF3 provided an opportunity to address limitations of aspect-oriented programming and of PyDySoFu, thus producing a more mature tool which is appropriate for use in research software engineering practice. In particular, PDSF3 implements a new design for the weaving of aspect hooks into prospective join-points — "import hook weaving" — which improves the legibility of an aspect-oriented codebase by making the application of advice more clearly visible to a reader of that codebase. This design addresses criticism of aspect orientation which observes that obliviousness can make programs difficult to read, reason about, and maintain [28, 132, 116, 117][1].

Import hook weaving is the most significant contribution of PDSF3. It makes advice application more legible than other aspect-oriented programming frameworks by restricting the ability of one area of a codebase to affect other areas, and takes advantage of conventional Python programs' layout to place uses of aspect-oriented programming at the top of a file. Using these design improvements, a developer can reason about any usage of aspect-oriented programming through "hints" that they should be aware of aspect-oriented programming in the codebase. As these hints are embedded in the source of a program using PDSF3, they are provided to software engineers without need for special tooling or static analysis.

PDSF3 also contributes a new type of optimisation to aspect-oriented programming frameworks, termed "cached fuzzing". This is an improvement to PyDySoFu's unique type of join-point, "fuzzers" where advice is able to change the definition of the target it is applied to.

[1] These criticisms are discussed in Section 2.1.7.

A limitation of PyDySoFu is that fuzzers are applied to their targets on every invocation of the corresponding join-point. While this allows for dynamic fuzzer behaviour, it also forces a user of the feature to repeatedly re-compile the target. If the result of this recompilation is the same every time, unnecessary overhead is incurred. To alleviate this, PDSF3 offers an optimisation which caches the result of a fuzzing aspect on its first invocation, and re-uses its cached value to avoid unnecessary re-compilation. This is discussed in more detail in Section 4.5.4.

Improvements over PyDySoFu's original design are also addressed. For example, PDSF3 only weaves aspect hooks into callable values stored within modules such as methods and functions, avoiding the overhead introduced by PyDySoFu's aspect hooks when resolving non-callable attributes. PDSF3 also offers granular application of aspects by making use of Python's import functionality — if only specific functions within a module are intended to be used as join-points, importing only those functions from the module using PDSF3's import hook weaving applies hooks only to those functions. This further reduces overhead from unused join-points.

Features of other aspect-oriented programming frameworks were also missing from PyDySoFu, which have been added to PDSF3, such as join-points which handle raised `Exception`s and the ability to specify the ordering of advice invocation. This improves the tool's maturity and brings it closer to feature parity with other aspect-oriented programming frameworks. Exception handling aspects are discussed in Section 4.4.5; ordering advice invocation is discussed in Section 4.5.3. Other optimisations are discussed throughout Section 4.5.

4.2 Requirements for Change

After developing a study using PyDySoFu [150], it became clear that an iteration on its design was required. PyDySoFu grew out of an undergraduate project, and accrued technical debt as a result of being written under time constraints with little experience. On revisiting

its design and reflecting on other aspect orientation frameworks reviewed in Section 2.1 and the use previously found for PyDySoFu [150, 151] it was clear that there were improvements which could be made to the tool:

- PyDySoFu made use of techniques for applying aspect hooks which did not translate to the changes Python 3 made to its object model. In particular, Python 3 changed its underlying object model, using a wrapper class imposing constraints on writing to some attributes of callable objects. This impacted the implementation of fuzzers, and no viable way to reimplement fuzzers using the old technique could be found.

- PyDySoFu made no accommodations for efficiency. It could be seen as the "total weaving" described by Chitchyan and Sommerville [20] as aspect hooks were implemented within the `__getattribute__` method of a class, with the effect that even properties of the class which were not intended to be used as join-points were processed through PyDySoFu's aspect hooks. This incurred an overhead to their performance. In Python, the `__getattribute__` method retrieves *all* attributes of an object, including special built-in attributes and non-callable ones. Aspect hooks were therefore invoked in almost any scenario where a program made use of an instance of any class which had aspect hooks applied. As `__getattribute__` is used by Python when retrieving any attribute of an object, it was not possible to limit the impact of PyDySoFu's aspect hooks to only affect potential join-points.

- PyDySoFu made no accommodations for scenarios where fuzzing of source code was applied in a "static" manner. That is to say, where a deterministic modification to source was woven as advice, instead of dynamically modifying source code, the same modification would still be made every time the target attribute was executed, unless caching of results was specifically managed by the aspect applying the change. No optimisations were made pertaining to this, but compilation and abstract syntax tree editing have the potential to be PyDySoFu's most expensive operations.

- Unlike other aspect orientation frameworks such as AspectJ [79], pointcuts could not be specified. Instead, individual join-points were be supplied as a Python object. This deviated from aspect-oriented programming conventions but only limited a user of PyDySoFu: specifying a set of join-points was not supported by the library. Users of PyDySoFu did not benefit from this lack of functionality.

As several requirements were left unfulfilled by PyDySoFu, a new implementation satisfying them was deemed necessary.

4.3 Python3-Compatible Weaving Techniques

Replacing `__getattribute__` on the class of a targeted method was no longer viable in Python 3. A replacement method therefore had to be found. As discussed in Section 3.2.1, PyDySoFu's technique of replacing `__getattribute__` allowed for hooks to be woven at runtime into possible targets of advice. These hooks would then discover and manage the execution of advice around each target. Because advice can be run before, after and around a target dynamically, an alternative technique must also intercept the calling of any target and manage advice immediately before execution.

This section describes a search to discover alternative techniques to dynamically weave and manage aspects which are suitable for PDSF3's implementation. We refer to code woven around a target which manages applied advice as *"aspect hooks"*.

The techniques of interest are specifically those which can be used to implement aspect-oriented programming without introducing domain-specific languages (as in the case of AspectJ, surveyed in **????**) or a special language runtime (as in the case of PROSE [113] or Nu[119]). PDSF3 is instead implemented in Python, requires no additional dependencies, and is used by writing native Python code. This design is chosen for two reasons. Firstly and most significantly, PDSF3 would be easier to maintain in the future if it avoids requiring a

change to the Python runtime or the development and maintenance of a domain-specific language. Changes to the language's runtime are avoided because the Python runtime is routinely updated and changes would need to be rebased against new Python versions. A domain-specific language is avoided because an implementation of a new language is likely to be more complicated to implement than a Python package: such a package is simple to implement by virtue of Python's flexible design [65, 22] (described further in Sections 4.3.1 and 4.3.2). Secondly, it is expected that the ability to use PDSF3 without learning new languages or adopting unofficial (and therefore potentially less reliable) language runtimes may be appealing to potential users of aspect-oriented programming. This expectation could be verified by a survey of researchers who are curious about aspect-oriented modelling to investigate whether these design properties make PDSF3 more appealing than other aspect-oriented programming frameworks. This is left as future work, as there was insufficient time remaining in this project to conduct such a survey.

4.3.1 Abandoned Techniques

PyDySoFu relied on "monkey-patching": the practice of making on-the-fly changes to objects by "patching" them during program execution [65]. This is often achieved by taking advantage of language properties such as flexible object structures [65]. Common examples of these structures are maps from string attribute / method names to the associated underlying value, which is the model used to represent all objects in both Python and JavaScript. Monkey-patching makes use of these structures by replacing values such as the function object mapped to by the original function's name in the dictionary, effectively changing its behaviour. An example of monkey-patching in Python is given in Fig. 4.1 This is the method by which PyDySoFu replaced `_getattribute__` on a class object.

Rather than relying on monkey-patching a new version of `__getattribute__` containing aspect hooks, the rewritten method could be patched to the object itself at a deeper

```python
from functools import partial
class Person:
    def __init__(self, name: str):
        self.name = name

    def say_hello(self):
        print(f"Hello! I'm {self.name}.")


arthur = Person("Arthur Dent")
ford = Person("Ford Prefect")

# Prints "hello, I'm " + name of person for both Arthur and Ford.
arthur.say_hello()
ford.say_hello()

# Monkey-patch the `Person` class, changing `say_hello` for all instances.
def italian_hello(person: Person):
    print(f"Ciao! Sono {person.name}.")
Person.say_hello = italian_hello

# Arthur and Ford now both greet in Italian.
arthur.say_hello()
ford.say_hello()

# Monkey-patching can affect individual instances of classes too, rather than
# all instances of the class.
# Monkey-patch Ford's `say_hello` method, making their greeting Norwegian.
# (Python fills in say_hello's first argument automatically; this is replicated
# here using `functools.partial`. This is an implementation detail made
# necessary by Python and is unrelated to the concept of monkey-patching.)
def norwegian_hello(person: Person):
    print(f"Hei! Jeg er {person.name}.")
ford.say_hello = partial(norwegian_hello, ford)

# Arthur now greets in Italian, but Ford greets in Norwegian.
arthur.say_hello()
ford.say_hello()
```

Figure 4.1: An example of monkey-patching in Python, showing both changes to all instances of the `Person` class and changes to individual instances.

level than was available through PyDySoFu's underlying mechanism. This would make use of Python's `ctypes` API to patch the underlying object. Similar work has been done in the python community in a project called ForbiddenFruit [22]. ForbiddenFruit uses the `ctypes` API to interact with the Python runtime and alter the in-memory representation of the attributes of class objects, "cursing" those objects in ForbiddenFruit's jargon.

This modification is highly dependent on the specific details of the modified attribute. While many attributes are supported by ForbiddenFruit, cursing `__getattribute__` is not supported. Efforts were made to add `__getattribute__` cursing, but this was abandoned as the underlying mechanism is unsafe, Python API changes could render the library unusable in future versions of the language, and the implementation would only work with particular implementations of Python (for `ctypes` to exist, the Python implementation must be written in C). Community patches existed for cursing `__getattribute__`, but these also could not be made to work. There are also efficiency concerns with this technique depending on its use: weaving advice around a function would mean monkey-patching the built-in class of functions, which would incur an overhead from running aspect hooks on every invocation of every function as PyDySoFu did.

Another approach for dynamic weaving which was attempted involved making use of existing Python functionality for interrupting method calls. As PyDySoFu wrapped method calls dynamically, what was required was to add program logic to the beginning and end of the execution of a method. Python has features providing this functionality which were designed for the implementation of debuggers, profilers, and similar development tools. The built-in function `settrace` allows a developer to specify a function which is invoked on certain events in Python's runtime. The intended use case for `settrace` is to track function invocations and exceptions being raised — however, the existence of a function invocation event suggested that PDSF3 could dynamically apply aspects by using this functionality.

Making use of `settrace` also has issues. Most significantly, `settrace` catches myriad

events in the Python interpreter which PDSF3 is unconcerned with, which incurs significant overhead. In addition, only one callback can be set using `settrace`, meaning that any programmer would have to choose between using a debugger and using PDSF3. This was deemed untenable: debuggers are an important development tool and a weaving approach which prevents their use was not realistically useful. However, future Python versions may change their design of `settrace` in a way which makes this weaving approach feasible. If future versions of the language allow for multiple trace handlers, this could provide a promising alternative approach to the implementation of future dynamic aspect orientation frameworks.

4.3.2 Import Hooks

A final available technique is to continue to monkey-patch hooks to discover and weave aspects, via an alternative method which does not make use of `__getattribute__`. This approach would change the use of PDSF3 to make a compromise between performance and obliviousness of aspect application: when importing a module targeted for aspect weaving, potential target methods are monkey-patched with a wrapper method with a reference to the original — necessary to run the targeted method — and hooks to detect and run dynamically supplied advice.

Many behaviours of objects in Python are defined through their "magic methods" (as mentioned in Section 3.2.1). Magic method identifiers begin and end with two underscore (_) characters and have special meanings within Python. The Python language documentation specifies sets of magic methods and their required function signatures which are used internally to implement functionality [142]. For example, any object with the method `__eq__` defined can be compared against using the == operator, and the `__eq__` magic method is run to determine the outcome of the operator. Magic methods are also used for more complex Python functionality: they determine whether they satisfy the conditions of some Python's

categories of objects. For example, anything which defines `__len__` and `__getitem__` is treated as an immutable container, and adding `__setitem__` and `__delitem__` makes that container mutable. Any class defining `__call__` is treated as a callable object akin to a function: when the object is called, the `__call__` magic method is executed. More can be found in Python's documentation [142], although more focused guides exist in the Python community [118].

Magic methods make Python ideal for implementing import-based weaving of aspect hooks, particularly dynamically, as they can be overridden. Python's functionality for importing modules is managed by `builtins.__import__`, which receives module names as strings and handles package resolution; by monkey-patching the import system, modules can be modified during the process of importing. As this technique allows for control over where aspect hooks are applied, PDSF3 can target only function and method objects to apply aspect hooks to, avoiding the overhead PyDySoFu introduced when applying hooks to all attribute lookups including non-callables, such as variables or `Class` objects. [2]

Monkey-patching `builtins.__import__` is as simple as replacing the `__import__` function object with a new one, which changes the behaviour of Python's `import` keyword. Because all Python functionality relies on magic methods implicitly, its behaviour can be altered in this way. However, the intent is not necessarily to manipulate *all* modules, but a subset of imports specified by a modeller as suitable for manipulation. If all invocations of `import` wove hooks into modules, including those made while *already* in the process of importing packages, an unnecessary overhead would be introduced when invoking any module and another overhead would be incurred executing the aspect hooks on any callable imported by any module. For this reason it is important to have a mechanism to enable and

[2] A successor to PDSF3 could hypothetically be extended to apply advice to attributes instead of functions or methods, as was possible with the previous version's weaving technique using `__getattribute__`. However, it is unclear how this could be implemented as the new weaving technique relies on target *invocation*, meaning only callables are viable targets. The additional research overhead of devising a technique to apply advice to aspects when their values are resolved was deemed too great to justify the investment given this functionality was not required for the work at hand. Python's `@property` decorator turns methods of classes into properties of those classes; as this would allow properties to be represented internally as methods, and so as possible join-points with PDSF3, an investigation into this decorator could yield a viable approach. This is suggested as potential future work on the PDSF3.

disable the weaving of aspect hooks for a given `import` statement, requiring a mechanism to enable and disable PDSF3's modified import logic.

Monkey-patching `builtins.__import__` can be achieved through another use of magic methods in a manner which also makes clear to a modeller exactly where aspect hooks are being applied: making use of Python's `with` keyword, which uses the `__enter__` and `__exit__` magic methods to define a block of code where an arbitrary resource such as a file, mutex, or network connection is safely managed. This behaviour is exapted[3] by PDSF3 to manage the behaviour of the import keyword, creating a block of code where aspect hooks can be injected into modules as they are imported. As these are a core component of PDSF3's weaving implementation, a deeper explanation of Python's `with` keyword follows in Section 4.4.2 following a technical discussion of the new weaving process which gives context to its design and implementation.

4.4 PDSF3's Import Hook Weaving

Section 4.3 surveyed mechanisms to implement dynamic aspect hook weaving which did not rely on a domain-specific language or changes to language runtimes. Some potential mechanisms were described in Section 4.3.1 which were ultimately deemed infeasible to adopt for performance reasons, maintenance burden, and the difficulty of using those mechanisms with other developer tools. In Section 4.3.2 it was determined that weaving aspect hooks as modules are imported was viable. The implementation of import hook weaving and its use in implementing an aspect-oriented programming framework is discussed in this section.

[3] Gould and Vrba [53] coined the term "exaptation" to refer to a trait of a species which evolved in response to one need, but were later co-opted to satisfy another.

4.4.1 High-Level Description of Weaving Process

Weaving in PDSF3 [147] takes place via monkey-patching of aspect hooks, described in detail in Section 4.4.2. Aspect hooks replace executable targets within a module at the moment the module is imported. When the target is invoked, the wrapping aspect hook is executed in lieu of the original target object. The wrapping aspect contains the target function within its closure, allowing it to execute the original target; however, it was also created by the `AspectHooks` class, and so has reference to aspects registered against it.

The high-level process of using aspect-oriented programming with PDSF3's import hook weaving is as follows (visualised in Fig. 4.2):

① A module which will be the target of later aspect application is imported using `AspectHooks`. This replaces callable objects in the module with wrappers of those objects which contain aspect hooks. The process involved in this step is explained in detail when discussing implementation details in Section 4.4.2. A simple example is shown in Fig. 4.3 on lines `1-3`.

② An aspect is registered against `AspectHooks`. A simple example is shown in Fig. 4.3 on line `8`. More detailed examples are given in Section 4.4.6 following an explanation of their implementation. A pointcut is specified by providing a regular expression matching the names of intended join-points as the first argument to an aspect registering method of `AspectHooks`; their second arguments are the advice to apply. Available aspect registration methods are described in Table 4.1. Note that…:

- Invoking the above aspect hook registration methods compiles and caches the regular expression provided, improving the efficiency of using the regular expression to identify join-points in later target invocations.
- Each aspect registration method returns a callback which de-registers the registered aspect. This facilitates the dynamic application of aspects.

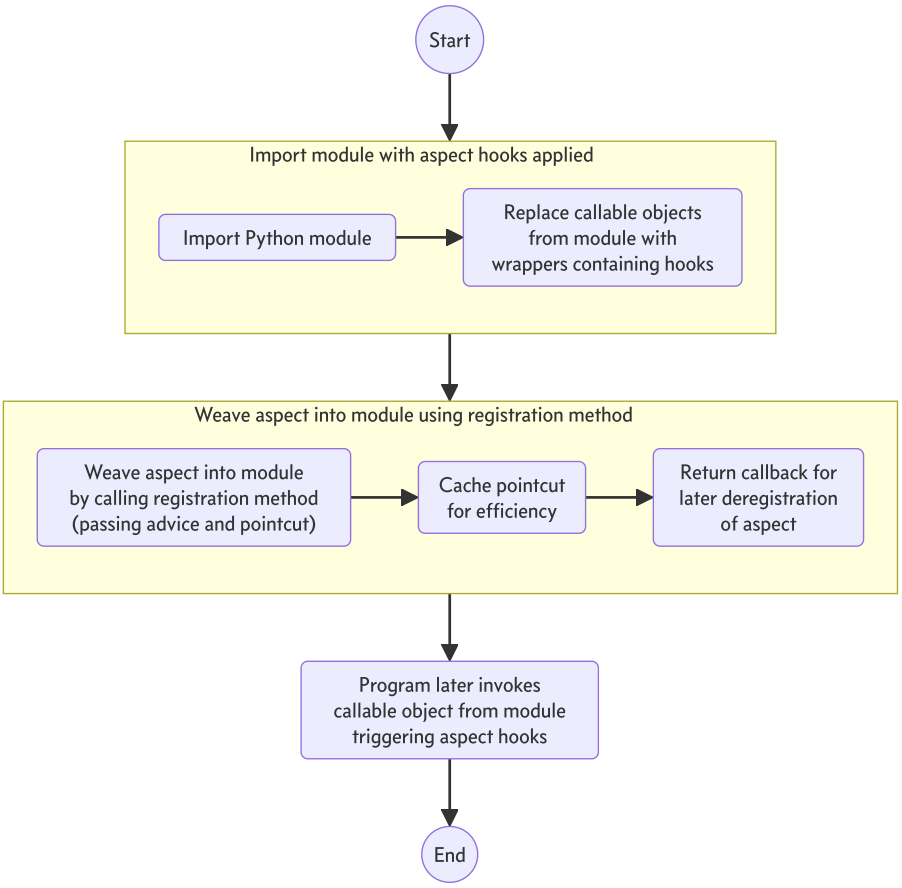

Figure 4.2: Flowchart showing the high-level process of using aspect-oriented programming through PDSF3's import hook weaving, as described in Section 4.4.1. This involves weaving aspect hooks into a module, applying advice to point-cuts, and invoking a join-point within the specified point-cut to trigger the advice.

③ An invocation of a function or method within that module triggers PDSF3's aspect hooks, which dynamically discover aspects registered against themselves using aspects' pointcuts. These pointcuts are described by the regular expressions they were registered with. Any aspects which are discovered are executed appropriately, as described later in Section 4.4.5.

PDSF3 advice is implemented as callable objects (usually functions). Different types of advice have different function signatures; the arguments expected by each type of advice are given in Table 4.2. In the arguments for each type of advice listed in the table: `target` is the target an aspect is applied to; `target_return_val` is the value returned by a target after it was run; `next_around` is what an `around`-style advice runs when it has finished its pre-target component, and intends to run the target itself; `ast_steps` is a list of AST

```
from pdsf import AspectHooks
with AspectHooks():
    import some_module

def prelude_advice(target, *args, **kwargs):
    print(f"Invoking {target.__name__}.")

AspectHooks.add_prelude("*some_func*", prelude_advice)
```

Figure 4.3: A simple example of aspect weaving, showing a prelude aspect definition, the injection of aspect hooks into a module, and the registration of advice against those aspect hooks, with pointcuts defined as regular expressions. Specific details of each step are given in later subsections as appropriate.

Method Name	*Description*
`add_prelude(rule, aspect)`	Registers advice to be run before a target is invoked
`add_encore(rule, aspect)`	Registers advice to be run after a target is invoked
`add_around(rule, aspect)`	Registers advice to wrap an aspect invocation, effectively providing the functionality of `prelude` and `encore` advice with a single aspect
`add_error_handler(rule, aspect)`	Registers advice to catch and process exceptions raised by a join-point
`add_fuzzer(rule, aspect)`	Registers advice to modify a target before it is invoked, effectively providing aspect application within a join-point, and permits arbitrary modifications of the target at the level of the abstract syntax tree.

Table 4.1: Methods of the `AspectHooks` class which are used to register different types of aspect to be dynamically woven. All methods take a string-represented regular expression describing the aspect's pointcut, and a callable object as the aspect to apply to that pointcut. The aspect should have the appropriate function signature for the type of advice being woven, as described in Table 4.2. Each method also accepts an optional `urgency` integer parameter, enabling optional additional features of PDSF3. This optimisation is explained after implementation details are discussed; more details are given in Section 4.5.3.

PDSF3 Type of Advice	*Common Name In Literature*	*Arguments Accepted by Aspect Implemented in PDSF3*
`prelude`	"before"-style	`target, *args, **kwargs`
`encore`	"after"-style	`target, target_return_val, *args, **kwargs`
`around`	"around"-style	`next_around, target, *args, **kwargs`
`error\_handler`	error handler	`target, handled_exception, *args, **kwargs`
`fuzzer`	"within"-style	`ast_steps, target, *args, **kwargs`

Table 4.2: Different types of advice supported by PDSF3 and the arguments they expect

```
1 @classmethod
2 def add_around(cls, rule, advice, urgency=0):
3     rule_tuple = (re.compile(rule), advice, urgency)
4     AspectHooks.around_rules.append(rule_tuple)
5     return lambda : AspectHooks.around_rules.remove(rule_tuple)
```

Figure 4.4: Implementation of the `add_around` method of the `AspectHooks` class, used to associate some advice (3^{rd} argument) to a pointcut, specified by a regular expression (2^{nd} argument). The `urgency` argument is related to an optimisation in the tool, and is described in Section 4.5.3. Its first argument is an ancillary implementation detail related to the `classmethod` decorator.

objects representing the definition of the target a fuzzing aspect is applied to; and `*args` , `**kwargs` is a Python idiom collecting a function's positional and keyword arguments respectively which were passed into the function when it was invoked but was not specified within its signature. These are used in the signatures of advice to collect arguments passed to a target, and can be inspected by any aspect, and passed into the target if the aspect invokes it.

Most types of advice are named using the metaphor of a play, drawing on early Theatre_AG work as described in Section 3.3.2: `prelude`, `encore`, and `around` advice runs before, after, and before & after the target respectively. Also following the naming scheme used by PyDySoFu are `fuzzer` aspects, which operate within their targets. The common names of these types of aspects in other literature, also shown in Table 4.2, make clear how their invocation relates to the invocation of the target they are applied to. The term "within" is introduced to conform to the positional naming convention used by other tools ("before", "after", and "around"). A type of advice not present in PyDySoFu is `error_handler` advice, which catches and attempts to handle exceptions raised by a target.

```
1 with AspectHooks():
2     import mymodule
```

Figure 4.5: Example of importing a module named `mymodule` using PDSF3's import hook design. This code snippet weaves aspect hooks into all functions and non-builtin class methods within the `mymodule` module object, and this modified module is added only to the local namespace of the importing stack.

`AspectHooks` methods which register methods are all implemented similarly. The implementation of the `add_around` method is given in Fig. 4.4. It requires two arguments: advice to apply, and a string-formatted regular expression describing the pointcut which this advice should be applied to. Line `3` constructs an aspect from the advice and pointcut passed as arguments, and line `4` adds this aspect to a list of around aspects, which is searched as required when potential join-points are invoked. This is described in further detail in Section 4.4.4. It returns a lambda which deregisters the aspect. Detailed examples using these methods to register aspects are given in Section 4.4.6.

4.4.2 Controlling Import Behaviour using Python's `with` Blocks

To inject aspect hooks into modules when importing them, PDSF3 must modify `builtins.__import__` *only* when modules containing potential join-points are imported. A developer might import many modules which do not require aspect hooks to be introduced: introducing hooks to all modules would incur some overhead from the hooks themselves when invoking any callable in any module. To enable this, PDSF3 uses Python's `with` keyword, as demonstrated in Fig. 4.5. The use of `with AspectHooks()` in Fig. 4.5 monkey-patches all potential targets of advice in the `mymodule` package to inject aspect hooks into them.

The behaviour of Python's `with` keyword can be explained through the magic methods which enable it: when entering a `with` block, the `__enter__` method on the object following the `with` keyword is invoked. When exiting the block, its `__exit__` method is invoked. It is only valid to enter a `with` block by using an object with both magic methods defined. The `AspectHooks` class uses these methods to override Python's import functionality when enter-

```
import sys
import copy

class AspectHooks:
    def __enter__(self, *args, **kwargs):
        self.old_import = __import__
        self.old_modules = copy.copy(sys.modules)
        self.imported = set()
        import builtins
        builtins.__import__ = self.__import__

    def __import__(self, *args, **kwargs):
        if args[0] in sys.modules:
            del sys.modules[args[0]]
        mod = self.old_import(*args, **kwargs)
        self.imported.update([args[0]])

        def apply_hooks(target_object):
            # ...implementation snipped

        apply_hooks(mod)
        return mod

    def __exit__(self, *args, **kwargs):
        builtins.__import__ = self.old_import
        for i in self.imported:
            if i in self.old_modules:
                sys.modules[i] = self.old_modules[i]
            else:
                del sys.modules[i]
        self.imported = set()
```

Figure 4.6: PDSF3 implementation of the magic methods used by a Python `with` block to alter import behaviour.

ing the block by replacing `builtins.__import__`, and reverts to the original functionality when exiting the block. The implementation of these methods is shown in Fig. 4.6.

When entering the block, the original import function is stored and replaced by a method of the `AspectHooks` instance. This is shown on lines `6`, `9` and `10` of Fig. 4.6. The original function is used to import modules on line `15`, but the module it returns is altered by PDSF3 by injecting aspect hooks on line `21`. The mechanism for applying these hooks is explained in Section 4.4.4. The original implementation of `builtins.__import__` is restored on line `25`, ensuring that import behaviour is only altered within the `with` block.

4.4.3 Locally Injecting Aspect Hooks

Before discussing how aspect hooks are injected into a module object, an additional piece of functionality shown in Fig. 4.6 must be explained.

PDSF3 ensures that modules only have aspect hooks where the module is imported with `AspectHooks`, rather than every time the module is imported. Additional work is necessary for this behaviour, because Python's default import logic caches modules and uses the cached module objects when importing a second time [44]. This cache is stored in `sys.modules`. If hooks are applied to a module within this cache, *every* part of a program which imports that module will include hooks, as every time the module is imported the same object is returned. This would lead to previous imports of a module which is imported using `AspectHooks` to have altered behaviour, even though aspect hooks were injected in an unrelated part of a program. This is similar to the behaviour of other aspect-oriented programming approaches, such as AspectJ [61]: aspects can be applied to a pointcut anywhere in a program. However, this makes code difficult to reason about [132, 116]. PDSF3 improves on the legibility of aspect-oriented programs by *only injecting hooks where PDSF3 has been invoked*, affecting the program locally rather than globally. To achieve this, Python's cache of previously imported modules must be managed.

Management of this cache is explained using the implementation shown in Fig. 4.6.

When entering a `with` block using an instance of the `AspectHooks` class, the instance's `__enter__` method is called. This method stores a copy of the modules imported when entering the block on line `7` of Fig. 4.6. This is used later to ensure that the cache contains the versions of any imported module which do not contain aspect hooks when exiting the `with` block. On line `8`, a set of all modules imported while inside the block is initialised. This is used to identify which modules' states must be reset when leaving the block.

Before leaving the block, PDSF3 modifies Python's import logic to inject aspect hooks. The `if` statement on `13` of Fig. 4.6 checks whether the module being imported already exists in the cache. If it does, it is removed from the cache, so that when Python's usual import function is later invoked it re-imports the module rather than relying on the cache. The usual import function is invoked on line `15`. This returns a new module object, and also adds this module object to the cache. The rest of this function injects aspect hooks into the module.

When exiting the `with` block, the cache should be returned to its expected state. Any modules which were altered by PDSF3 exist in the cache of imported modules, and should be replaced with their expected values: if they previously existed in the cache, the previous object without aspect hooks should replace the one currently residing in the cache. If it did not previously exist, the object should be removed from the cache entirely, so that later import statements re-import the module and construct a new module object which lacks aspect hooks. This check is performed by the `for` loop on line `26` of Fig. 4.6. Previously existing modules replace those constructed by PDSF3 on line `28`, and new entries in the cache are deleted on line `30`.

Following these steps, the state of Python's module cache is managed to avoid modules imported elsewhere without PDSF3 from containing aspect hooks.

```
1  from inspect import isfunction, ismethod, isclass
2  from functools import filter
3
4  def build_wrapper(callable_object):
5      # ...implementation snipped.
6      # Builds a wrapper around `callable_object` containing aspect hooks
7      # and returns the wrapped object.
8
9  def apply_hooks(target_object):
10     nonprivate = lambda p: isinstance(p, str) and len(p) > 1 and p[:2] != "__"\\
11     for item_name in filter(nonprivate, dir(target_object)):
12         item = getattr(target_object, item_name)
13
14         from_mod = getattr(item, "__module__", None) == mod.__name__
15         if from_mod or AspectHooks.deep_apply:
16             if isfunction(item) or ismethod(item):
17                 setattr(target_object, item_name, build_wrapper(item))
18             elif isclass(item):
19                 apply_hooks(item)
```

Figure 4.7: The implementation of the `apply_hooks` function used on line `21` of Fig. 4.6, which injects aspect hooks into any function or method within a module object or the classes it contains. Edited slightly for legibility.

4.4.4 Injecting Aspect Hooks Into Modules

Figure 4.7 shows the implementation of the `apply_hooks` function, referenced on line `21` of Fig. 4.6. This function searches an object for functions and methods which are possible targets of advice, injecting aspect hooks into each one. Any classes contained within the object are searched recursively.

Line `10` of Fig. 4.7 defines the condition used to filter attributes of `target_object`, which is a module object when `apply_hooks` is first called, but a class object on recursive invocations, as explained later. The condition on which an attribute of an object is checked as a possible join point is that it is a string with at least two characters and where the first two characters are not underscore (_) characters. This ensures that protected attributes, which are idiomatically prepended with two underscore (_) characters, are not accidentally used as targets for advice. These methods typically have special meanings within Python and could introduce unexpected side-effects and significant performance overheads if their behaviours are changed by advice. This condition is used to filter out attributes of `target_object` which would be unsuitable join-points on line `11`. Attributes which are suitable join-points are retrieved from the object on line `12`.

Line 14 of Fig. 4.7 checks whether the prospective join point belongs to the same module as the one being imported. The variable mod is defined elsewhere, as it refers to the module being imported: the code shown in Fig. 4.7 is the snippet truncated from line 19 of Fig. 4.6. This functionality is related to the deep hook weaving optimisation explained later in Section 4.5.1 — lines 14 and 15 will be explained as needed when discussing this optimisation.

Lines 16–19 of Fig. 4.7 inject aspect hooks into target_object if appropriate. If the object is a function or method, it is replaced within the imported module. Its replacement is a new function or method built by the build_wrapper function, which wraps target_object with aspect hooks. However, if target_object is a class, it is not callable (and so cannot be a join-point) but may *contain* functions or methods which would be valid join-points. To inject aspect hooks into these, apply_hooks is called recursively on the class object.

Calls to apply_hooks modify the objects passed as arguments directly. As a result, they need not return any value. All functions and methods which are suitable join-points have been replaced with wrappers implementing aspect hooks at this point, and so the work required to be done on the imported module object is done, and aspect hooks are injected into it at all appropriate points.

4.4.5 Building Wrappers With Aspect Hooks

Figure 4.8 shows the implementation of the build_wrapper function, which takes a function or method as an argument and returns that argument wrapped inside a function which invokes advice. The build_wrapper function is used by the apply_hooks function as shown in Fig. 4.7 to inject aspect hooks into functions and methods modules and classes they contain.

This code snippet is large, but there are six areas of particular importance to the discussion of PDSF3's implementation. Lines 2–13 construct some foundations used by other parts of

```
def build_wrapper(target):
    old_target_code = copy.deepcopy(target.__code__)
    class CouldNotFuzzException(Exception):
        pass

    @wraps(target)
    def wrapper(*args, **kwargs):
        pre, around, post, error_handlers, fuzzers = self.get_rules(target.__name__,
                                                                    AspectHooks.manage_ordering)

        def reset_code_to_previous():
            if not isinstance(target, FunctionType):
                target.__func__.__code__ = old_target_code
            else:
                target.__code__ = old_target_code

        try:
            # Apply fuzzers
            t = target
            if fuzzers is not None and fuzzers != []:
                cache_key = tuple([str(fuzzer) for fuzzer in fuzzers] + [str(target)])

                if self.cache_fuzz_results and self._cached_fuzzer_applications.get(cache_key):
                    compiled_fuzzed_target = self._cached_fuzzer_applications[cache_key]
                else:
                    code = dedent(inspect.getsource(t))
                    target_ast = ast.parse(code)
                    funcbody_steps = target_ast.body[0].body

                    for fuzzer in fuzzers:
                        non_inline_changed_steps = fuzzer(funcbody_steps, *args, **kwargs)
                        if non_inline_changed_steps:
                            funcbody_steps = non_inline_changed_steps

                    target_ast.body[0].body = funcbody_steps
                    compiled_fuzzed_target = compile(target_ast, "<ast>", "exec")
                    if self.cache_fuzz_results:
                        self._cached_fuzzer_applications[cache_key]=compiled_fuzzed_target

                if not isinstance(t, FunctionType):
                    t.__func__.__code__ =  compiled_fuzzed_target.co_consts[0]
                else:
                    t.__code__ = compiled_fuzzed_target.co_consts[0]

            # Run prelude advice
            [advice(t, *args, **kwargs) for advice in pre]

            # Build a wrapper for the target using around advice
            def nest_around_call(nested_around, next_around):
                return partial(next_around, nested_around)
            nested_around = reduce(nest_around_call,
                                   around[::-1],
                                   self.final_around)
            ret = nested_around(t, *args, **kwargs)

            # Run post advice and return the final return value
            for advice in post:
                post_return = advice(t, ret, *args, **kwargs)
                ret = ret if post_return is None else post_return
            reset_code_to_previous()
            return ret

        except Exception as exception:
            reset_code_to_previous()
            prevent_raising = False
            for handler in error_handlers:
                prevent_raising = prevent_raising or handler(t, exception, *args, **kwargs)
            if not prevent_raising:
                raise exception

    return wrapper
```

Figure 4.8: The implementation of `build_wrapper`, which takes a function or method as an argument and returns that object wrapped with logic implementing aspect hooks. Edited slightly for legibility.

```
function build_wrapper(target):
    original_target_code ← get_code_for(target)
    return wrapper(target, original_target)

function wrapper(target, original_target):
    preludes, arounds, encores, error_handlers, fuzzers ← aspects_for(target)
    try:

        Fuzzing aspect implementation
        if fuzzers is not empty:
            if target in fuzzer cache:
                target ← cache⟨target⟩
            else:
                steps ← ast of target
                foreach fuzzer in fuzzers:
                    steps ← fuzzer(steps)
                if caching enabled:
                    cache⟨target⟩ ← compile steps
                target ← compile steps

        Prelude advice implementation
        foreach prelude in preludes:
            prelude(target)

        Around advice implementation: wrap target in around advice
        foreach around in arounds:
            target ← partial around(target)

        Invoke target and around advice, by invoking wrapper built above
        return_value ← target()

        Encore advice implementation
        foreach encore in encores:
            new_return_value ← encore(target, return_value)
            if new_return_value is not None:
                return_value ← new_return_value

        Reset changes from fuzzer
        target ← original_target

        Return value originally returned by target
        return return_value

    Exception handling advice implementation
    catch Exception as error:
        target ← original_target
        prevent_raising ← False
        foreach handler in error_handlers:
            if handler(target, error):
                prevent_raising ← True
        raise error if not prevent_raising
```

Figure 4.9: Pseudocode explaining how `build_wrapper` works. The implementation is given in Fig. 4.8.

the implementation. Lines 17–41 implement "within"-style aspects (fuzzers). Prelude advice is handled by line 44. Lines 50–57 invoke "around"-style aspects, and also invoke the target of any advice applied. Lines 60–64 invoke "after"-style aspects, and handle the return value of the target. Finally, lines 66–72 handle exceptions raised by the target or any advice applied, using exception handling advice. These will be explained individually.

Initial Setup

Lines 2–13 of Fig. 4.8 contain setup for later parts of the `build_wrapper` function.

A copy of the original code object of the target being invoked is stored on line 2. This is required to restore its state if fuzzers are applied. This state is restored by calling a function, `reset_code_to_previous`, implemented on lines 9–13. An exception to be raised if an error is encountered during fuzzing is implemented on lines 3–4. The aspects which apply to the target being invoked are retrieved by a call to another function, `get_rules`, on line 8.

Fuzzers

Lines 17–41 of Fig. 4.8 implement PDSF3's "within"-style aspects, or fuzzers. A pseudocode representation of the process is given in Fig. 4.10. Lines 19–22 implement a static fuzzing optimisation, which is discussed later in Section 4.5.2. Their explanation is omitted here, and included in Section 4.5.2 instead.

The target being invoked is fuzzed on lines 24–36. The original source code of the target is retrieved using Python's built-in `inspect` library, and indentation is removed to avoid errors when recompiling. An AST is constructed from this source code, and the part of that AST representing the body of the function implementation is stored in the `funcbody_steps` variable. Lines 28–30 contain a `for` loop which calls every fuzzer applied to the target. Each fuzzer takes an AST as an argument and returns a replacement AST with any transforma-

```
1  fuzzers ← get fuzzers for target
2  if fuzzers is not empty:
3      if target in fuzzer cache:
4          target ← cache⟨target⟩
5      else:
6          steps ← ast of target
7          foreach fuzzer in fuzzers:
8              steps ← fuzzer(steps)
9          if caching enabled:
10             cache⟨target⟩ ← compile steps
11         target ← compile steps
```

Figure 4.10: A pseudocode representation of the process used to apply fuzzers to the targets of advice. This is a simplification of the implementation given in Fig. 4.8.

tions it applies. By the end of this `for` loop, the AST stored in `funcbody_steps` has been transformed by every fuzzer woven onto the target.

Lines `33-34` recompile this AST, producing a new function for which the definition has been transformed by every fuzzer applied. Lines `38-41` replace the code object holding the target's compiled implementation with the code object stored within the new function object which was just compiled. Every Python function or method contains a code object, which stores the compilation of the function performed by the Python runtime. By replacing this object, the implementation of the target is effectively changed. An appropriate technique is used to replace the target's code object depending on whether the target is a function or a method. Having replaced the code object and so the target's implementation, fuzzers are successfully applied to the target, and their effects will occur when the target is invoked.

Prelude Advice

Prelude advice is PDSF3's terminology for "before"-style advice, following the terminology established by PyDySoFu and Theatre_AG. This is invoked before around advice or encore advice, and is implemented by line `44`. This line contains a list comprehension which iterates through every prelude advice retrieved on line `2` and invokes each one. As every prelude advice is called, no further work is required to implement prelude aspect hooks.

```
def multiply(a, b):
    return a*b

def construct_partial(func, val):
    def partial(*args, **kwargs):
        return func(val, *args, **kwargs)
    return partial

double = construct_partial(multiply, 2)
```

Figure 4.11: An example of a partial function. A function multiplying two numbers is given the value 2 for its first argument, producing a partial function which doubles its remaining argument.

Around Advice

Lines 50–57 of Fig. 4.8 implement around advice.

Partial functions are used to create wrappers for around advice using other around advice. A partial function is one where a subset of its arguments are given values, but the function is not invoked. Figure 4.11 contains an example where a function which multiplies two numbers is given the value of 2 for its first argument to create a partial function which doubles its remaining argument.

Partial functions can be used to implement around advice because of their function signatures. All around advice takes the arguments:

```
next_around, target, *args, **kwargs
```

…and each around aspect must make the invocation…

```
next_around(target, *args, **kwargs)
```

…somewhere, to ensure that all around advice is properly invoked. All around advice contains some logic to happen before a target is invoked and other logic to happen after a target is invoked. As many around advices may be applied to a join-point, each around advice must invoke its successor. This is achieved using partial functions: before being invoked, every around advice is given its successor as its first argument. This means that when `next_around` is invoked, its own first argument is already given the value of the

```
1 def final_around(self, target, *args, **kwargs):
2     return target(*args, **kwargs)
```

Figure 4.12: The implementation of the `final_around` method on an instance of the `AspectHooks` class, used on lines `52-54` of `build_wrapper` as shown in Fig. 4.8

successor to that around advice. The successor to this is also a partial function containing its own successor as its first argument, and the successor to that advice is a partial function constructed in the same manner, and so on for every around advice. This ensures that every around advice contains a reference to its successor. Without this design, the first around advice would require a reference to *every* around advice to ensure every one could be invoked, of which an indefinite number are applied. These partial functions are constructed using the `nest_around_call` function on lines `50-51`.

While there exists an indefinite amount of around advice to apply, there must be a finite amount, which means that the final around advice has no successor. However, its function signature must be the same as any other advice, and so the `next_around` argument must be given a value. The `final_around` method of the `AspectHooks` class is designed to be used as the final successor in the chain of around advice, and is used to end the sequence of partial functions with references to each other. Its implementation is given in Fig. 4.12. Its signature contains only the target of the advice and its arguments. The `final_around` function acts as a wrapper for the target: it invokes the target with the correct arguments and returns any value returned by the target. Pseudocode illustrating the high-level steps executed in building and invoking around advice is given in Fig. 4.13.

Lines `52-54` of Fig. 4.8 construct a single function which uses the partial function logic explained above and the `final_around` function to create a single function which invokes all around advice *and* the target. This function is stored as `nested_around`. Its construction is most easily explained recursively. If there is no advice to apply, `nested_around` is equal to `final_around`, and so is a wrapper around the target which invokes it and returns any value returned by the target. This will be referred to here as ⓞ. If one around advice is woven,

```
Wrap target in final_around so function signatures match around advice
target ← partial(final_around, target)

Give every around advice the next one in the list as its first argument
The first around advice takes the final_around-wrapped target instead
foreach around in reversed(arounds):
    target ← partial(advice, target)

Call the "target", invoking the first around advice, which calls the
second, which calls the third...until finally calling final_around,
which calls target.
return_value ← target()
```

Figure 4.13: Pseudocode demonstrating how a wrapper is constructed which invokes both around advice and the target of aspect application.

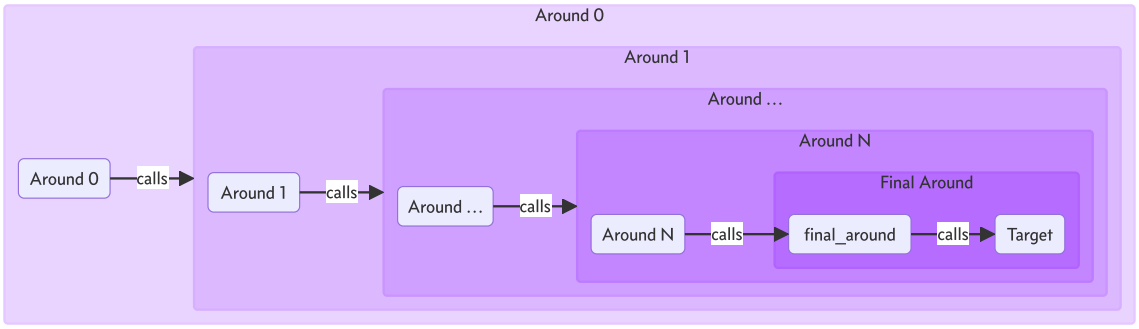

Figure 4.14: An informal flowchart showing how around advice nests. Each box shows a level of nesting of partial functions, and represents the closure of the partial function constructed at that level of nesting. Nodes in the graph are functions implementing either the target of advice, `final_around`, or a piece of advice woven against the target.

`final_around` is a partial function which is the same as that around advice but for which the first parameter is `final_around`, or ⓪. As a result, when it invokes its successor, the target is invoked instead (via its wrapper). This value of `nested_around` is a function which executes both the around aspect and its target and is referred to as ①. If a second around advice is woven, `final_around` is a partial function constructed from *this* around advice, but its successor parameter is ①. This would execute the second around aspect with ① as its successor, applying its own logic before and after the invocation of the other advice and the target at the correct points. This is referred to as ②. If a third is woven, the same occurs, and its successor parameter is ②. As with ②, this advice is executed with the correct ordering. This continues for every n^{th} advice applied, which is constructed as a partial function where its successor parameter is given the value $(n-1)^{th}$ version of `nested_around`. By induction, this is correctly constructed for every n. The levels of nesting are shown in Fig. 4.14.

Line 57 invokes nested_around, which in turn invokes every around advice applied as well as the target. The return value of the target, possibly modified by around advice, is stored in the variable ret.

Encore Advice and Returning from Aspect Hooks

Encore advice, invoked after a target returns, is implemented on lines 60–62 of Fig. 4.8. Any advice which runs after the target returns is able to inspect and modify the target's return value if required; line 62 replaces the copy of the target's return value with that returned by any advice if the advice returns a non-None value.

Line 63 resets the code object implementing the target using reset_code_to_previous , ensuring that any fuzzers that were invoked had no side-effects lasting after the wrapper function returns. No further work is required of PDSF3, and so the wrapper function returns the return value of the target (or its modified value if changed by any encore advice).

Error Handling Advice

The final type of advice supported by PDSF3 is error-handling advice. If a target raises an exception, it is intercepted by PDSF3, which will continue to raise the exception unless error handling advice is applied.

Line 66 of Fig. 4.8 intercepts a thrown exception. The underlying code object of the target is reset on line 67, in case it was modified by a fuzzer, to avoid side-effects remaining after the aspect hooks return. Each error handler applied to the target is executed on lines 69–70. If any "truthy" value is returned by any error handling advice, the exception is not raised and the caller of the target can continues to execute, oblivious to the exception. A "truthy" value is any value which is interpreted as True in a boolean expression in Python — this is any value which is not None, 0, or False.

```
1  def prelude_log_invocation(target, *args, **kwargs):
2      '''
3      Prelude advice to print invocations of a target.
4      '''
5      print(f"Invoking {target.__name__}")
6
7  def encore_log_return(target, return_val, *args, **kwargs):
8      '''
9      Encore advice to print a target's return value
10     '''
11     print(target.__name__ + " returned value: " + str(return_val))
12
13 def around_logger(next_around, target, *args, **kwargs):
14     '''
15     Around advice equivalent to applying both
16     prelude_log_invocation and encore_log_return.
17     '''
18     print("Invoking " + target.__name__)
19     return_value = next_around(target, *args, **kwargs)
20     print(target.__name__ + " returned value: " + str(return_val))
21     return return_value
22
23 from pdsf import AspectHooks
24 with AspectHooks():
25     from some_module import some_func
26
27 AspectHooks.add_prelude("some_func", prelude_log_invocation)
28 AspectHooks.add_encore(".*", encore_log_return)
```

Figure 4.15: Complete examples of prelude, encore, and around advice. Prelude and encore advice is registered against pointcuts described by string-represented regular expressions. The around advice is not registered, but is provided to demonstrate the similarities between around advice and the combination of prelude and encore advices.

4.4.6 Using PDSF3

An example of an aspect woven into a join-point is shown in Fig. 4.15.

Lines `1-11` define functions which can be used as advice. The first, `prelude_log_invocation`, is intended to be run as prelude advice and so has the function signature `target`, `*args`, `**kwargs`. It prints the name of a function before it is invoked. The second, `encore_log_return`, is intended to be run as encore advice and so has the function signature `target`, `return_val`, `*args`, `**kwargs`. It also logs the name of the target function, as well as the value that function returned.

Lines `23-25` import a function, `some_func`, from a module `some_module` using `AspectHooks`. The hooks are active within the body of the `with` block. The function object imported as `some_func` is the original function wrapped in the aspect hook implementation shown in

```
1  import ast
2
3  def fuzzer_invocation_logger(ast_steps, target, *args, **kwargs):
4      '''
5      Fuzzer advice which prints the name of the invoked target.
6      '''
7      print_invoked_func = f"Invoking {target.__name__}"
8      print_invocation_ast = ast.parse(print_curr_func)
9      return print_invocation_ast.body + ast_steps
10
11 from pdsf import AspectHooks
12 with AspectHooks():
13     from some_module import some_func
14
15 AspectHooks.add_fuzzer("some_func", fuzzer_invocation_logger)
```

Figure 4.16: Complete example of logging function invocation using a fuzzer instead of the prelude advice shown in Fig. 4.15.

Fig. 4.8. Line 27 applies prelude advice to the `some_func` function by associating `prelude_log_invo` with the pointcut `"some_func"`. Only functions with the name `some_func` will be invoked with this advice applied. Line 28 applies the encore aspect `encore_log_return` to the pointcut `".*"`, which matches any function identifier — as a result, *any* function invocation with aspect hooks injected will be run with this encore advice applied. As a result, an invocation of the function `some_func` would print its name before invocation as well as its return value after invocation.

The around advice defined on lines 13-21, `around_logger`, is not woven but is shown to demonstrate how prelude and encore advice can be represented as around advice.

Figure 4.16 shows a fuzzer which logs function invocations similar to the example in Fig. 4.15. On line 7, a line of Python is stored as a string which prints the name of the target. This code is parsed into an AST on line 8, and the parsed code is prepended to the target's body on line 9. The first thing the target will do once recompiled by PDSF3 is to execute the line of code shown on line 7, after which the target's implementation is defined as it was previously. This fuzzer is woven into the target in the same way as in Fig. 4.15 in the remainder of the example.

4.4.7 Comparison to Other Weaving Mechanisms

Import hook weaving is a novel technique for weaving aspect hooks. This section compares PDSF3 against other aspect-oriented programming frameworks to determine its novelty and contribution. As it is not feasible to compare PDSF3 against *all* alternative aspect-oriented programming frameworks, it will be compared against AspectJ [61] and Spring AOP [93] as these are actively maintained aspect-oriented programming frameworks which see some industry adoption [117]. As such, we consider these frameworks representative of the aspect-oriented programming ecosystem at time of writing. Any claims of novelty made in relation to these frameworks hold in comparison against other frameworks too; AspectJ and Spring AOP are selected for illustrative purposes.

PDSF3's implementation of aspect-oriented programming weaves aspect hooks into modules which may contain join points. This allows join-points to be determined at runtime. Advice is implemented in the form of Python functions. Weaving is performed when modules are imported. Only a module which imports join-points with `AspectHooks` will observe aspect hooks woven into those join-points; other modules within the same program which do not import with `AspectHooks` will observe no aspect-oriented behaviour when invoking those join-points. Fig. 4.17 contains a Python program made of several modules which illustrates this. Weaving is dynamically performed at runtime, and can happen whenever a package is imported. As Python supports re-importing packages, PDSF3 can weave aspect hooks even when a package is already loaded. This allows PDSF3 users to determine whether they require aspect hooks to be woven at any point in program execution. Also, PDSF3's application of aspect hooks — and weaving of aspects — appears only in the parts of a program it effects. As PDSF3 weaves aspect hooks when packages are imported and Python packages are conventionally imported at the top of a file, PDSF3 users can easily clarify whether any aspect-oriented behaviour is expected in their program, and where that behaviour occurs.

AspectJ offers multiple mechanisms for weaving aspect hooks: compile-time weaving is

main.py

```
import moduleA
import moduleB

moduleA.runExample()
moduleB.runExample()
```

joinpointModule.py

```
def joinPoint():
    print("Inside joinPoint")
```

aspects.py

```
def log_invocation(target, *args, **kwargs):
    print(f"Inside prelude before invoking {target.__name__}")
```

moduleA.py

```
from pdsf import AspectHooks
import aspects
with AspectHooks():
    import joinpointModule

# Prints "Inside prelude before invoking joinPoint", then "Inside joinPoint".
# The first line is printed by the prelude advice woven using AspectHooks.
def runExample():
    AspectHooks.add_prelude(aspects.log_invocation, 'joinPoint')
    joinpointModule.joinPoint()
```

moduleB.py

```
import joinpointModule

# Only prints "Inside joinPoint" even though aspects were woven into the module
# by moduleA, because joinpointModule was not imported with AspectHooks
# in _this_ module.
def runExample():
    joinpointModule.joinPoint()
```

Figure 4.17: Example Python project with multiple modules, which demonstrate how PDSF3 isolates aspect-oriented behaviour to affect only modules which import join-points with `AspectHooks`. Here, `moduleA` weaves an aspect into `joinpointModule`, but `moduleB` does not observe any aspect-oriented behaviour because it did not import `joinpointModule` with `AspectHooks`.

offered through rewriting binary `.class` files [42] and runtime weaving[4] is offered through modified classloaders [43]. Neither mechanism supports weaving aspect hooks into already-loaded classes. The application of aspects is controlled via `.xml` configuration files in both cases. Aspects woven into a class are observed by all users of that class. Aspects are written in Java, using a superset of the Java language[5], so an extension to Java is required in order to use it. As AspectJ join-points are specified in configuration files, users of AspectJ must check any configuration files for their program to understand whether any part of their program will exhibit aspect-oriented behaviour. The negative impact on program legibility this causes is often cited as a limitation of aspect-oriented programming [28, 132, 116, 117].

Spring AOP is designed with AspectJ compatibility in mind [123], and so supports the AspectJ superset of the Java language and AspectJ's `.xml` configuration format. However, Spring AOP also offers its own aspect hook weaving technique: proxy objects which act as adapters between join-points and other parts of a program that invoke them. Proxy objects act as proxies of join-points. This allows fine-grained weaving of aspect hooks, as a program only needs to construct proxy objects for any join-points in use to support aspect-oriented programming. It also allows for flexible weaving of aspect hooks, as the construction of a proxy object can occur at any point in a program's execution. However, this weaving mechanism complicates code comprehension: any part of a program using a non-proxied reference to a join-point will see no aspect-oriented behaviour. For example, this occurs when the proxied object invokes methods on itself, as an object's `this` reference is not proxied through Java's proxy API. Overcoming this limitation requires a join-point to implement additional logic to call its own methods through a proxy if it exists, which breaks aspect-oriented programming's philosophy of obliviousness [39]. Spring AOP supports using AspectJ to weave aspect hooks and apply aspects instead of or in addition to its own aspect weaving, which may complicate program comprehension by adding additional complexity to determining whether any part of

[4] AspectJ refers to runtime weaving as "load-time weaving", because its implementation specifically weaves hooks when classes are loaded.

[5] The superset of the Java language used to implement aspects is called AspectJ.

a program exhibits aspect-oriented behaviour. This is because both weaving techniques must be checked to form a complete understanding of program behaviour.

Import hook weaving offers improved code comprehension compared to other weaving mechanisms, because it offers simple semantics around aspect-oriented behaviour: modules which import other modules with `AspectHooks` observe aspect-oriented behaviour, and any other modules do not. Confirming whether a program exhibits aspect-oriented behaviour for conventionally written Python programs is as simple as checking the top of a file for references to `AspectHooks`. This compares favourably to other techniques, which rely on program configuration to identify join-points and complicate the invocation of aspect-oriented program behaviours with wrapper mechanisms such as proxy objects. Import hook weaving therefore improves on existing aspect hook weaving mechanisms by offering a novel solution to the problem of aspect-oriented programs' legibility.

4.4.8 Strengths and Weaknesses of Import Hooks

As a technique for weaving aspect hooks, this new method provides multiple benefits.

Local Aspect Weaving. Application of aspect hooks is straightforward from the perspective of a modeller using PDSF3, whose code clearly applies aspect hooks and does so in a legible way for future maintainers. This is guaranteed by PDSF3's weaving of aspect hooks only in the scope of the program where they are applied: hooks are woven into the module where it is imported using `AspectHooks`, but not in other parts of the program. Import hooks' explicit application of hooks to modules makes clear the specific areas of a program where advice could be applied.

Legible Obliviousness. While join points are oblivious to potential aspect application, callers of join points are responsible for weaving the advice they want to apply. This leaves signs of aspect orientation in helpful places within a codebase for its maintainers. Aspect

hooks can be injected into specific functions imported from a module or the entire module depending on the way modules are imported, allowing for total weaving or actual hook weaving depending on their preferences. This is because, in Python, `import X` will import a module named `X`. However, the syntax `from X import A, B, C` imports only the symbols `A`, `B`, and `C` from the module `X`. Using this, aspect hooks can be injected into subsets of a module, rather than the entire module.

As well as allowing greater flexibility for the convenience of a developer, the performance of import hooks is improved in comparison to replacing the `__getattribute__` method on a class. In comparison to PyDySoFu, which used the latter technique, the new implementation's import hooks are weave-able at a more granular level. This is because import hooks introduce additional program logic to procedures such as functions and methods, rather than all attributes of a class as did PyDySoFu's old mechanism. Less overhead is expected as a result.

Requires No Special Accommodations. As with PyDySoFu, PDSF3's import hooks are implemented entirely in Python and require no special support from language runtime modifications. Other aspect orientation frameworks often require special languages for describing aspects, such as AspectJ [61], or runtime plugins, such as PROSE [113]. PDSF3's import hooks are written entirely in native Python, and can be used in existing software projects without requiring any special accommodations. Minimising the difficulty of adopting PDSF3 is a desirable property of its design: if it can be used without modifying existing codebases, researchers' pre-existing models & simulations can adopt PDSF3 with minimal engineering effort.

There are also caveats to this approach. As aspect hooks are woven in PDSF3 via Python's import functionality, any procedure which is not imported from a module cannot have aspect hooks attached. However, as aspect orientation is primarily concerned with a separation-of-concerns approach to software architecture, targets are expected to be scattered across many

modules, so this does not appear to be a significant limitation. The notion that targets reside in other modules than those where weaving takes place holds in other aspect-oriented simulation & modelling codebases: for example, those from Wallis and Storer [150] as described in Chapter 3.

4.5 Optimisations

Additional features were implemented in PDSF3 to make it more useful for research software engineering, and to optimise performance where necessary. The optimisations introduced are deep hook weaving, non-dynamic weaving, priority ordering of aspect application and cached fuzzing. They are explained individually in the following subsections.

4.5.1 Deep Hook Weaving

Ordinarily PDSF3 weaves aspect hooks into an imported module, but not modules imported by that module. This is to avoid overheads incurred by patching commonly-used libraries with aspect hooks in scenarios where they are not intended to be join-points. However, this behaviour may be desirable at times. Deep hook weaving enables weaving of aspect hooks into all potential join-points by not only applying hooks to functions and methods which were imported by the original module, but also to any modules which the original module imports. The feature is enabled by toggling a flag set on the `AspectHooks` class, as shown in Fig. 4.18.

```
1 from pdsf import AspectHooks
2 AspectHooks.deep_apply = True
```

Figure 4.18: Code snippet enabling deep hook weaving

Deep hook weaving introduces performance overhead due to additional checks for aspects which were woven dynamically. However, it is possible that the modules which contain desired join points are not imported directly by a developer, but are available to them indirectly

```
1 from_mod = getattr(item, "__module__", None) == mod.__name__
2 if from_mod or AspectHooks.deep_apply:
3     if isfunction(item) or ismethod(item):
4         setattr(target_object, item_name, build_wrapper(item))
5     elif isclass(item):
6         apply_hooks(item)
```

Figure 4.19: The implementation of PDSF3's "deep weaving" option, which allows aspect hooks to be injected into recursive imports or only into the first module being imported, reducing overhead if a specific join-point of advice is targeted by a developer. Edited slightly for readability.

through another package they make use of. It is also possible that a developer may look to instrument the entirety of a call stack without attaching a debugger to a process, for example to examine call stacks, to perform security checks, for fuzz testing, or for custom logging within third-party and built-in modules. These use-cases require aspect hooks to be woven more deeply than to only one module. In these examples, PDSF3's default behaviour is insufficient. Deep weaving is provided to support these use cases.

The implementation of deep weaving is shown in Fig. 4.19, a snippet of the `apply_hooks` function which monkey-patches module objects during import to inject aspect hooks. The feature is implemented as an `if` statement which ensures that the condition checking whether a module or class object originated from the original import always evaluates to `True`, as seen on line `2` in the figure. This check can effectively be disabled by ensuring the condition always evaluates to `True`, meaning that all imported modules and classes would be woven into, even if they did not belong to the first module object imported.

PDSF3 requires an option for deep hook weaving because it weaves aspect hooks when modules containing possible join-points are imported rather than when pointcuts are specified. Similar functionality is offered by aspect-oriented programming frameworks such as AspectJ and PROSE, but concepts in other frameworks differ by virtue of their different weaving mechanisms. For example, AspectJ weaves aspect hooks into every Java class by default but can configure this to be more specific through the configuration of its weaver [**empty citation**]. An example AspectJ configuration which weaves aspect hooks selectively and controls the recursive weaving of hooks shown in Fig. 4.20. AspectJ disallows the weaving of advice

```
<aspectj>
    <weaver>
        <!-- Weave hooks within the foo.bar or com.example namespaces -->
        <include within="foo.bar.*"/>
        <include within="com.example.*"/>

        <!-- Do not weave types within the "bar" namespace -->
        <exclude within="bar.*"/>
    </weaver>
</aspectj>
```

Figure 4.20: An example aspectj configuration which weaves aspect hooks rec

into delegated classloaders when weaving advice at runtime [43], which prevents classes repeatedly weaving advice into others they import or load (which is the use case for AspectJ most analogous to Python modules importing other Python modules). PROSE weaves aspects as join-points are called using a just-in-time (JIT) compiler [105]. Callbacks are woven into join-points when aspects are applied, alleviating the need to weave aspect hooks into classes and so avoiding the need for deep hook weaving by design. Deep hook weaving is therefore a novel solution to the recursive weaving of modules, but is not a *unique* solution to that problem.

4.5.2 Static Weaving

PDSF3's runtime weaving of aspects them to be applied dynamically. However, programs may be written with the intention of applying aspects once — for example, as program modifications to be introduced without direct manipulation of a codebase. Once defined, a set of applied aspects would remain unchanging in this scenario, so dynamic weaving of PDSF3 introduces unnecessary overhead: aspects will never be *unwoven*, so repeated searches for advice which applies to a given target are unnecessary.

```
from pdsf import AspectHooks
AspectHooks.treat_rules_as_dynamic = True
```

Figure 4.21: Code snippet enabling dynamic weaving

To avoid this overhead in scenarios where it is not required, an optimisation can be introduced where aspects are woven statically. PDSF3 can cache the aspects applied when

any callable wrapped by an aspect hook is invoked for the first time. On its first execution in this mode, the aspect hook stores the set of aspects it matched to the invoked target, and future invocations retrieve the set of aspects to apply from the cache. This avoids expensive regular expression matches, which fail in all cases but those where an invoked target is to be augmented by the application of an individual aspect the regular expression is paired with.

For performance reasons, the default behaviour of PDSF3 is to use static weaving. Dynamic weaving is enabled by toggling the `treat_rules_as_dynamic` flag on the `AspectHook` class, as similarly to enabling or disabling deep hook weaving (see Section 4.5.1). A code snippet demonstrating this is shown in Fig. 4.21.

Static weaving is similar to the ability to turn on or off dynamic behaviour in other aspect-oriented programming frameworks which offer dynamic aspect weaving. Static weaving therefore brings PDSF3 closer to feature parity with other aspect-oriented programming frameworks. An example of an analogous feature in other frameworks is AspectJ's compile-time loading of aspects by injecting advice into join-points by rewriting the `.class` files corresponding to each join-point [42].

4.5.3 Priority Ordering of Aspect Application

As dynamic weaving allows for the conditional application of aspects, it may be that the order in which aspects are woven is not the order in which they are intended to be *invoked*: different aspects may have different priorities. To support these use cases, aspects can be applied with a priority, which is used to sort them when aspects are dynamically searched for and invoked on target invocation. This feature is disabled by default, but can be enabled using the code snippet shown in Fig. 4.22.

```
1 from pdsf import AspectHooks
2 AspectHooks.manage_ordering = True
```

Figure 4.22: Code snippet enabling priority ordering of aspect application in PDSF3.

```
<aspectj>
    <aspects>
        <concrete-aspect name="com.thesis.ExampleAdviceClass"
                         precedence="logInvocation, checkSecurityProperties, *"/>
    </aspects>
</aspectj>
```

Figure 4.23: An example AspectJ configuration which specifies precedence of advice application [41].

As mentioned in Section 4.4.1, when aspects are registered against the `AspectHooks` class an optional `urgency` parameter is available. This parameter is an integer representation of the aspect's "priority" with the same semantics as in a priority queue. Higher numbers represent more urgent application, so high-urgency aspects are applied before low-urgency aspects. Aspects with no urgency applied default to `urgency=0`.

Other aspect-oriented programming frameworks also offer priority ordering of aspect application, although the feature is relatively novel in the community [68]. For example, AspectJ allows aspects to be declared with a precedence ordering which explicitly notes which aspects should run in which order for a given join-point [41]. An example AspectJ configuration which specifies precedence of advice application is given in Fig. 4.23.

4.5.4 Cached Fuzzing

Invoking fuzzers requires parsing Python ASTs and compiling Python code objects, which introduces overhead in their implementation. However, some use-cases for fuzzers will produce the same AST every time they fuzz their target. Examples include the function invocation logger given as an example in Fig. 4.16 and the advice written to implement experiments in Chapter 6 such as Fig. 6.8, both of which inject the same AST step nodes at the same point in their target on every invocation. To alleviate this overhead, the compiled result of applying fuzzers to an aspect can be cached. This is enabled using the `cache_fuzz_results` attribute of the `AspectHooks` class can be set, as shown in Fig. 4.24.

Caching fuzzer output allows PDSF3 to compile the output of any applied fuzzer on their

```
from pdsf import AspectHooks
AspectHooks.cache_fuzz_results = True
```

Figure 4.24: Code snippet enabling priority ordering of aspect application

```
1  if fuzzers is not None and fuzzers != []:
2      cache_key = tuple([str(fuzzer) for fuzzer in fuzzers] + [str(target)])
3
4      if self.cache_fuzz_results and self._cached_fuzzer_applications.get(cache_key
           ) is not None:
5          compiled_fuzzed_target = self._cached_fuzzer_applications[cache_key]
6      else:
7          code = dedent(inspect.getsource(t))
8          target_ast = ast.parse(code)
9          funcbody_steps = target_ast.body[0].body
10
11         for fuzzer in fuzzers:
12             non_inline_changed_steps = fuzzer(funcbody_steps, *args, **kwargs)
13             if non_inline_changed_steps:
14                 funcbody_steps = non_inline_changed_steps
15
16         target_ast.body[0].body = funcbody_steps
17         compiled_fuzzed_target = compile(target_ast, "<ast>", "exec")
18         if self.cache_fuzz_results:
19             self._cached_fuzzer_applications[cache_key]=compiled_fuzzed_target
20
21     if not isinstance(t, FunctionType):
22         t.__func__.__code__ =  compiled_fuzzed_target.co_consts[0]
23     else:
24         t.__code__ = compiled_fuzzed_target.co_consts[0]
```

Figure 4.25: Implementation of aspect hooks for fuzzers, which includes the optimisation for cached fuzzing. Fuzzed targets are cached on line `19`, and used on line `5` instead of re-running fuzzing advice if cached fuzzers are enabled by setting the `cache_fuzz_results` attribute of the `AspectHooks` class.

first invocation. Its implementation is shown in Fig. 4.25. On line `2`, a unique key for the advice applied to an invoked target is constructed. This key changes if dynamic fuzzing is enabled (see Section 4.5.2), so a cache miss occurs if fuzzing aspects are added or removed between target invocations. Lines `4-5` circumvent the implementation of fuzzer aspect hooks and retrieve a cached fuzzed target instead if the key constructed on line `2` exists in the cache of fuzzed targets. If it does not, fuzzing continues as normal, and the output of running any applied fuzzers is added to the cache on line `19` to be used when the target is next invoked.

As fuzzers are a unique feature of PyDySoFu and PDSF3, cached fuzzing does not exist in other aspect-oriented programming frameworks. It is an improvement over PyDySoFu and thus a novel contribution in the design of aspect-oriented programming frameworks.

4.6 Summary

PDSF3 improves over both existing aspect-oriented programming frameworks and its own previous incarnation, PyDySoFu. It introduces a new technique of weaving aspect hooks when importing modules, improving its design over a typical aspect orientation framework by making use of Python's `with` keyword when weaving hooks. This improves the legibility of aspect-oriented programs through more explicit application of aspect hooks. In addition, many optimisations are introduced PyDySoFu did not offer: aspect hooks can be injected only into a module being imported, or recursively into every module imported in the process of importing the first one; aspects can be woven dynamically or statically, reducing overheads when developers do not require dynamic behaviour; and the order of application of aspects can be made explicit by developers when this functionality is required.

PDSF3 also provides opportunities for improvements and for future work. Our intended use case for aspect orientation for simulation & modelling is in scientific codebases specifically: direct integration with the scientific package ecosystem (which is vibrant in Python's community) should be made. A good initial project would be integration of aspect application in sciunit tests [138], which represent experiments on a model as unit tests. The potential to encode hypotheses as advice (discussed in Section 8.8) or as alternative behaviours within a model (described in Chapter 6 and evaluated in Chapter 7) has lots in common with SciUnit's design goal of formalising hypotheses as unit tests. As both projects are written in Python and have similar use-cases, there is seemingly some potential for integration between these projects and collaboration with the SciUnit team. A discussion on potential use cases of PDSF3 together with existing research software engineering technologies is provided in Section 8.8.

Chapter 6 and Chapter 7 use PDSF3 to design experiments which investigate aspect-oriented simulations & models, and evaluate their results.

4.6.1 Contributions

PDSF3 contributes import hook weaving to aspect-oriented programming framework design. This novel type of weaver introduces aspect hooks to join-points without making changes to a language runtime or weaving aspect hooks directly into a program before or after compilation.

Import hooks address common criticisms of the aspect-oriented paradigm by improving the legibility of an aspect-oriented programming program. This is achieved by limiting the scope of a program which aspect hooks are woven into. This has the effect of reducing the number of points in a program that *could* weave aspect hooks into a join-point, so developers who need to understand a program only need to follow a function's call stack to observe whether that function could have aspects applied to it. This is because PDSF3's aspect hooks are woven in predictable parts of a program that are easily reviewed: import statements, which are written at the top of a Python program by convention.

PDSF3 also contributes new optimisations to aspect-oriented programming framework design: cached fuzzers. These concern fuzzing aspects, introduced by PyDySoFu, which make changes to their target when invoked. PyDySoFu was previously the only aspect-oriented programming framework which offered these join-points, but its implementation of fuzzing aspects caused targets to be recompiled on every invocation, regardless of whether the fuzzer's output was the same on every invocation. To avoid this overhead, PDSF3 offers a new type of optimisation which caches the output of fuzzing advice on its first invocation and re-uses its output on future invocations. This is a novel optimisation for aspect-oriented programming frameworks, as no other framework implementing fuzzing aspects offers it to date.

Finally, PDSF3 improves on PyDySoFu's design to make the framework more appropriate for use in research software engineering practice. This is achieved through the introduction of new join-points such as exception handlers and the ability to specify the ordering of aspect

application. PDSF3 also changes the underlying mechanisms used to weave aspect hooks to make the framework compatible with versions of Python which are still maintained.

Chapter 5

RPGLite

This chapter introduces the foundations of contributions in Chapters 6 and 7. It supports those contributions through a research software engineering effort which contributes an implementation of a mobile game as well as its distribution and maintenance. This chapter also contributes a dataset describing real-world gameplay [74] to support other research efforts [71, 73] and a discussion of best practices which research software engineers may fall victim to [154] to aid other researchers' engineering efforts in this domain.

5.1 Introduction

To investigate whether models can be augmented using aspect-oriented programming, PDSF3 can be employed to produce aspect-oriented models. However, a system to model is required. To address the research questions introduced in Section 2.4.1, a system is required which can be modelled, and for which the accuracy of that model's representation of the system can be measured. This is required because the aspectually augmented model must be compared to the model without aspects woven to discern which is more accurate. This chapter introduces a suitable system, RPGLite, a game for which play can be modelled simply and augmented using aspect-oriented programming. RPGLite was implemented and deployed

on iOS and Android to collect real-world play data to compare simulation output against. Techniques for performing these comparisons are introduced in Chapter 6 and the results of these comparisons are presented in Chapter 7. This chapter describes the design of RPGLite, its implementation as a mobile game, and the data collected from its use.

RPGLite is a game designed to mimic existing popular games, while being structured to permit exploring all game states via formal methods. Kavanagh and Miller [72] have produced formal models using a model-checker, PRISM, which identifies optimal play strategies in all states a game can reach.[1] Some experiments were conducted using RPGLite by Kavanagh [71] to investigate whether players' interactions with the game converged on optimal play.

RPGLite gameplay data invites many research questions. Kavanagh and Miller's original RPGLite design as a formal model was intended to allow the calculation of the *"cost"* of an action [72, 71] (the degree to which a player is less likely to win having made a given move instead of the optimal one), allowing for rich analysis of gameplay datasets. However, another possibility invited by these datasets is that they enable the development of models of individuals' gameplay. This offers an opportunity to investigate the effectiveness of aspect-oriented simulation and modelling: the style of play of different individuals may be best encoded by different advice (or parameters for that advice).

RPGLite is of interest in other ways than the study of gameplay and game design. The game is a socio-technical system which can produce useful datasets to support multiple avenues of research in different disciplines. [2] A dataset of real-world RPGLite play would support future work in a variety of fields as a non-commercial dataset for research, including at least game design and gameplay research [71], formal methods [72], sociotechnical simulation & modelling research as in the following chapters, and research software engineering tooling demonstration also in the following chapters.

[1] Game states are explained in Section 5.2.1.

[2] Later contributions in this thesis are supported by datasets produced by RPGLite, as were the contributions of Kavanagh's PhD thesis [71].

For this reason, this research includes a collaboration with Kavanagh to develop and release a mobile implementation of RPGLite which collects gameplay data for research purposes. This dataset enabled Kavanagh to demonstrate the utility of their model checking in an empirical scenario [71]. With regards the research presented in later chapters it enables the analysis of models representing player behaviour. The mobile game contributes this by supporting the comparison of these models' output against the collected data, and providing the basis of a model which can be augmented using aspect-oriented programming. The model, introduced in Chapter 6, represents random play. Aspects are written which augment the model to better represent real-world play, the results of which are presented in Chapter 7. If the aspectually augmented models generate gameplay data which correlates with empirically sourced data more closely than that generated by the unmodified model, we can dismiss unmodified play as "unrealistic", and understand the aspectually augmented behaviour as "more realistic" — however, the mobile game is required to provide the real-world data enabling this comparison.

5.1.1 Contributions and Chapter Outline

This chapter describes research software engineering work undertaken to support the contributions of Chapters 6 and 7. Its purpose is not to introduce contributions constituting of specific results or answers to the research questions posed in Section 2.4.1. However, two other contributions are produced as a result of this work:

① A dataset describing real-world RPGLite gameplay, which has been published as an open dataset for the benefit of other researchers [74] and has already supported others' research in game design and analysis [71, 73]. This is described in detail in Section 5.3.6.

② Insights into the pitfalls some research software engineers may fall prey to when developing a mobile game. This was published as a separate piece of work [154] but is summarised in Section 5.3.4. The discussion of design principles when developing a game for research purposes is intended to aid other researchers who have a need to

collect real-world data through a mobile application or game but have little experience of developing similar projects. This is intended to mitigate setbacks in the work of others and improve the quality of software developed for research purposes.

This chapter focuses only on the design of RPGLite, its implementation as a mobile game, and the dataset collected from real-world play [154]. It starts with an overview of RPGLite itself in Section 5.2, and progresses to discuss the game's implementation in Section 5.3 including an application for iOS and Android (Section 5.3.1), the server and API powering its online functionality (Section 5.3.2), and the recruitment of players to use the game (Section 5.3.3). Some insights from the game development process are given in Section 5.3.4, which summarises already-published contributions [154]. The game's multiple configurations are described in are described in Section 5.3.5 followed by a description of the dataset produced by players of the game in Section 5.3.6 before the chapter concludes with a brief summary (Section 5.4).

5.2 An Overview of RPGLite

RPGLite is a two-player game and is played in turns. Each player selects two characters to play with independently of their opponent. Each character has a unique set of abilities and properties, which are health, chance of success on attack, and damage dealt on a successful attack as well as special abilities and properties for some characters. Eight characters are available for selection. Characters are differentiated by the special effect they may use when attacking an opponent character. They are defined alongside their special abilities in Table 5.1. Specific parameters for each character — their health, chance to hit, and damage on hit as well as character-specific details depending on their special abilities — are defined as a *"configuration"* of RPGLite. Two configurations were released as different *"seasons"* of the RPGLite mobile game, which are discussed in Section 5.3.5.

To act on their turn, each player selects an "alive" character (one with health greater than

Character	*Special Ability when Attacking*
Knight	Deals damage to an opponent character on a successful hit
Archer	Deals damage to two opponent characters on a successful hit
Wizard	Deals damage to an opponent character on a successful hit, disabling (or *"stunning"*) them for the duration of the opponent's next turn
Healer	Deals damage to an opponent character on a successful hit, and heals themselves or, optionally, the other player character instead (assuming that character is still alive)
Barbarian	Deals damage to an opponent character on a successful hit, dealing additional damage if their own health is low when attacking
Rogue	Deals damage to an opponent character on a successful hit, dealing additional damage if the target's health is low when attacked
Monk	Deals damage to an opponent character on a successful hit, and immediately takes another turn, until their attack is unsuccessful
Gunner	Deals damage to an opponent regardless of success, dealing additional damage on a successful hit

Table 5.1: RPGLite characters and their unique special abilities.

0) to attach a chosen "alive" target of their opponent, with caveats for some characters' special abilities. A successful attack is randomly determined by the chance of a successful attack for the character selected by the active player. If successful, the attack deals damage, and also results in that character's unique ability being activated. When starting a game, a random player is chosen to take a first move. Players may always skip their turn as a valid action. Players continue to take alternating turns until one player has no "alive" characters. They lose, and their opponent is the victor.

Different configurations change the game's *"balance"*, a term referring to the relative strengths of different characters or character pairs. For example, if a configuration leaves many characters with initial health values close to a Barbarian's threshold for additional damage, then they become a very powerful character due to their ability to inflict additional damage. If the Monk's chance to hit is high, the repeated turns it offers can be very advantageous. Character skills can work in concert with each other: choosing a Barbarian and Healer such that the barbarian can be kept at low health for additional damage, but the healer can be used to keep them alive, may be an effective strategy depending on the game's configuration. Kavanagh et al. found that model-checking a configuration of the game could discover the relative strengths of characters and character pairs when played optimally [75].

5.2.1 Game States & Playing Optimally

RPGLite's design is sufficiently simple that it supports mathematical analysis while being interesting enough to attract a community of active players over many months [74]. An explanation of the simplicity of its design is given here and the implications for its use in simulation & modelling are discussed.

A game of RPGLite can be described as being in a particular state at any point in time. For example, characters can have different amounts of remaining health, characters may be stunned, and different players can be active at different points in time. The set of possible

states a game can reach is its "state space". The state space of RPGLite is small by design, making it well-suited to analysis through formal methods. A calculation determining the size of this state space follows. Because of the game's small number of possible states, it is feasible to search the state space and discern which moves in each state are optimal. Simulations can take optimal play into account, and real-world play can also be compared against optimal moves to discern whether players play "correctly" or exhibit biases or errors.

It is possible to calculate the size of this state space by identifying which properties of the game uniquely distinguish a state. RPGLite games can have their states uniquely identified by the healths of characters on each team, the player whose turn is next, and which character is stunned (if any). Two characters are selected for each player and each has their own maximum health value, which is no more than 10 for any character in RPGLite's publicly released configurations; there are therefore 10^4 different health states. Only one of two players may take their turn at any moment, meaning that every game state has two possible values for this parameter. There are three possible values for stunned-ness: the active player's first character can be stunned, or their second character, or neither. As status of a character resets when a turn finishes, stunned-ness is only relevant to the active player's characters.

These values uniquely determine the state of an RPGLite game [71]. There are therefore a maximum of:

First Player's Health		Second Player's Health				Stunned-ness		Active Player		Max States Per Game
10	×	10	×	10	×	10	×	3	×	2	=	60,000

…possible states for any RPGLite game, assuming maximum health values of 10. However, this number only quantifies the number of states a *single* game of RPGLite can have. Players pick two of eight character pairs, so there are $\binom{8}{2} = 28$ choices of character pairs a player can make. The total number of possible states across all possible games is therefore: $28 \times 28 \times$

$60000 = 47,040,000$. This calculation disregards that some character pairs do not include wizards and so cannot lead to stunned states — it therefore sets an upper bound on the size of the state space.

This upper bound is significantly lower than that of other popular games. The information content of an RPGLite space can be calculated as its entropy [127], which can be calculated as the logarithm of the size of its state space [57]: $log_2(47,040,000) \approx 25.5 bits$. By comparison, mapping valid positions in chess takes $\approx 136 bits$ [106]. This property allows RPGLite to be analysed formally, as the game's state space is small enough to explore computationally. It also draws on common game design elements and is sufficiently "interesting" to generate data from a real-world community of players, demonstrated through data collected from 9,693 completed games [74]. This yields some useful properties for the purposes of aspect-oriented simulation:

① Simulated moves can be selected at random, but can also be made perfectly by selecting the provably correct move in a given game state. They can also be made with some calculated "cost" to the chance of winning by deliberately selecting non-optimal moves [71].

② Real-world players' behaviour can also be analysed according to the same metrics: for example, moves made by players can be analysed to discover bias, whether players learned to play "better" moves over time, or whether they selected known-strong characters more frequently than those who can be formally shown to have a relatively low chance of winning games.

③ As the actions taken when playing RPGLite are consistent — such as deciding the target character of an attack, or a character to use in an attack — random play can be simulated as a "naive" play style, which can be compared against real-world players. Where player behaviour does not correlate to naive play,[3] the biases of players may be represented as aspects which are applied only to specific actors within the simulation.

[3] Play correlation is discussed in Section 6.6.3.

④ Should aspects be suitable as a technique for the accurate representation of biased play, they offer a separation of concerns within the simulation: any nuance found within the style of play of specific real-world players would be replicated and applied not to the model itself, but to specific simulated players. Styles of play might also be mixed with the application of multiple aspects.

Whether aspect orientation is suitable for the realistic simulation of RPGLite gameplay is the topic of the remainder of this thesis. However, the design of RPGLite allows for a controlled system where a clear notion of "good" and "naive" behaviours can be defined. Additionally, the system is closed insofar as all interactions within the system are known, and all game elements are precisely understood. All interactions take place between experiment participants for data collection purposes, allowing for a large dataset to be collected and disseminated to the research community. RPGLite's design constitutes a system where all of its properties are well-understood, no interference is anticipated from system components which are unknown or outside experimental control, and lots of data can be collected for analysis. The data it produces is therefore valuable for the purpose of constructing a realistic model, as the system producing that data is well suited to research use, no external factors affecting player behaviour are anticipated, and players' decisions can be modelled as optimal rather than random when "realistic" decisions are difficult to model and players are known to make few errors. This property is used to model players' move selections, discussed in Section 6.4.1.

5.3 Implementation of RPGLite

To understand how players interact with RPGLite, empirical data needed to be collected. To produce this, a mobile, online, multiplayer version of the game was developed for data collection purposes. After several months of play the database used by RPGLite's server was used to create a dataset describing players' interactions with the game [74]. This section

describes details of RPGLite's implementation as a data collection tool.

This involved consent to anonymous gameplay data being collected and published for research purposes (Section 5.3.3), the development of a mobile client for the game (Section 5.3.1), and the development of a server for that client to interact with (Section 5.3.2). A mobile game ready to be used for data collection purposes was constructed as described in the aforementioned sections; lessons learned from that development process are discussed in Section 5.3.4. The configurations of RPGLite deployed in the mobile game are discussed in Section 5.3.5. Data collected is described in Section 5.3.6.

5.3.1 Mobile Client Application

As a mobile game, RPGLite's user-facing component was an application, distributed through the Google Play Store on Android and the Apple App Store on iOS. This was developed in Unity, a framework for developing games in C# which can be distributed to almost any platform. Most assets were developed in GIMP with character designs contributed by a commissioned artist online. Unity allowed for a "What You See is What You Get" (*"WYSIWYG"*) interface builder, with event handlers defined in C# code which would "hook" into events signalled by interface element interactions. User-facing components of the game were largely produced by Kavanagh [71], the collaborator on this project and original RPGLite designer. Readers are referred to their notes on the development process for full details.

Beta testing required user engagement. Apps were deployed to colleagues' Android and iOS devices. Beta testing followed a design science methodology [69], where an artefact is repeatedly released and improved upon following feedback from users. Beta tests comprised of iterations on the artefact of the game and its server-side component, discussed in Section 5.3.2. Testing involved playing a series of games to check that the implementation was sufficiently robust and graphic design adequate for public distribution of the game. Early builds of the game were released to testers as `.apk` files for Android devices and through Apple's TestFlight

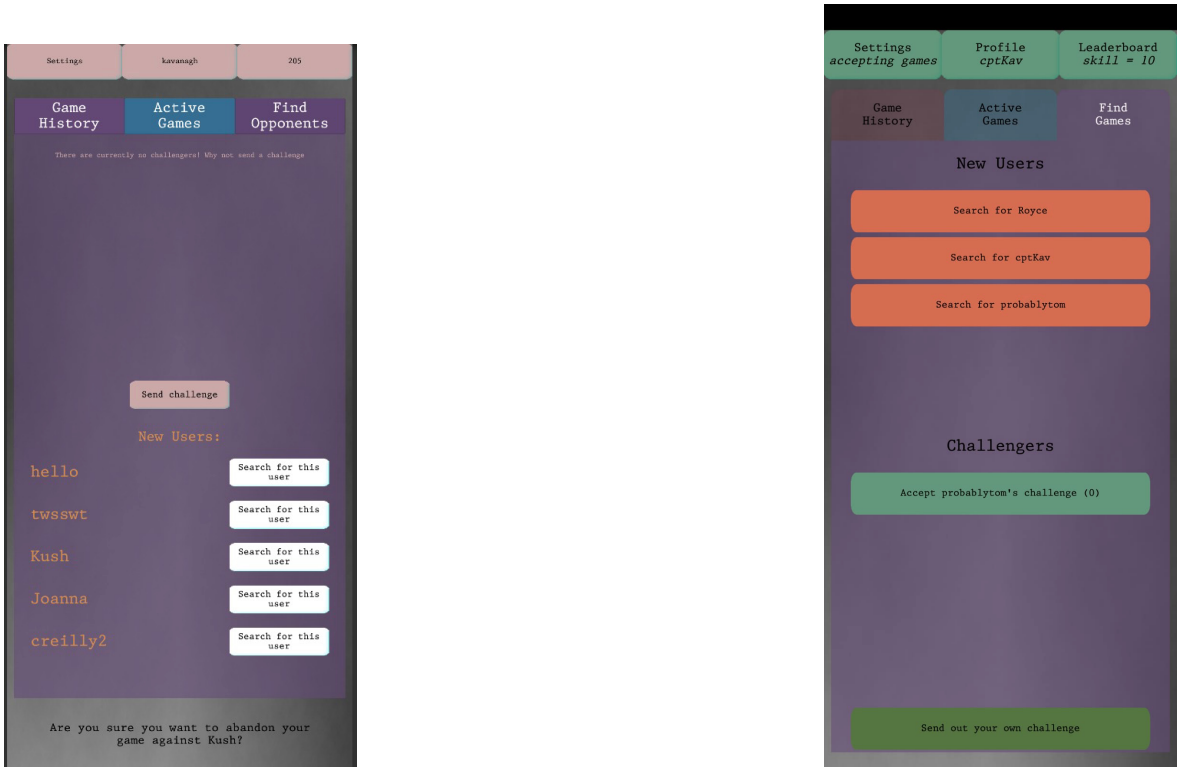

Figure 5.1: Evolution of the "Find users" matchmaking screen. Prototype left, final design right.

system for iOS devices. Informal feedback and enhancement requests were sought on each iteration until no major bugs were reported and no major complaints about interaction or aesthetics were reported, at which point the game satisfied criteria for public distribution.

An example of the game's visual evolution is given in Fig. 5.1, where colourful buttons replace a tabular, text-heavy interface. Another example is the evolution of the game's main screen, "Active Games", which players were presented with upon logging into the app and showed users active games they were involved in. The visual identity and colour palette of this screen was refined iteratively, as can be seen in Fig. 5.2. Other features which were developed following feedback from beta testers include the implementation of a notification system and a leaderboard showing a player's experience relative to their peers. These are shown in Fig. 5.3.

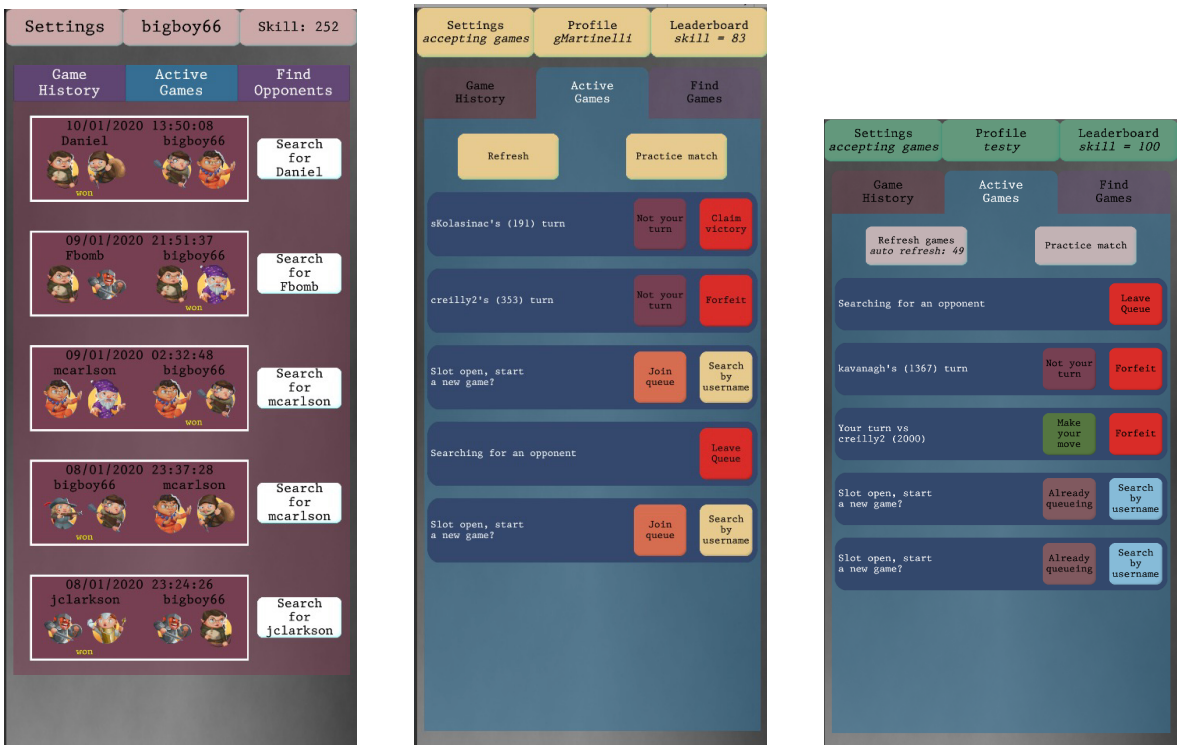

Figure 5.2: Evolution of the "Active Games" screen, which allowed RPGLite users to see and interact with games they were playing. An early prototype is shown to the left; a refinement through beta testing in the centre; and the final version released to the public to the right, with an improved colour palette.

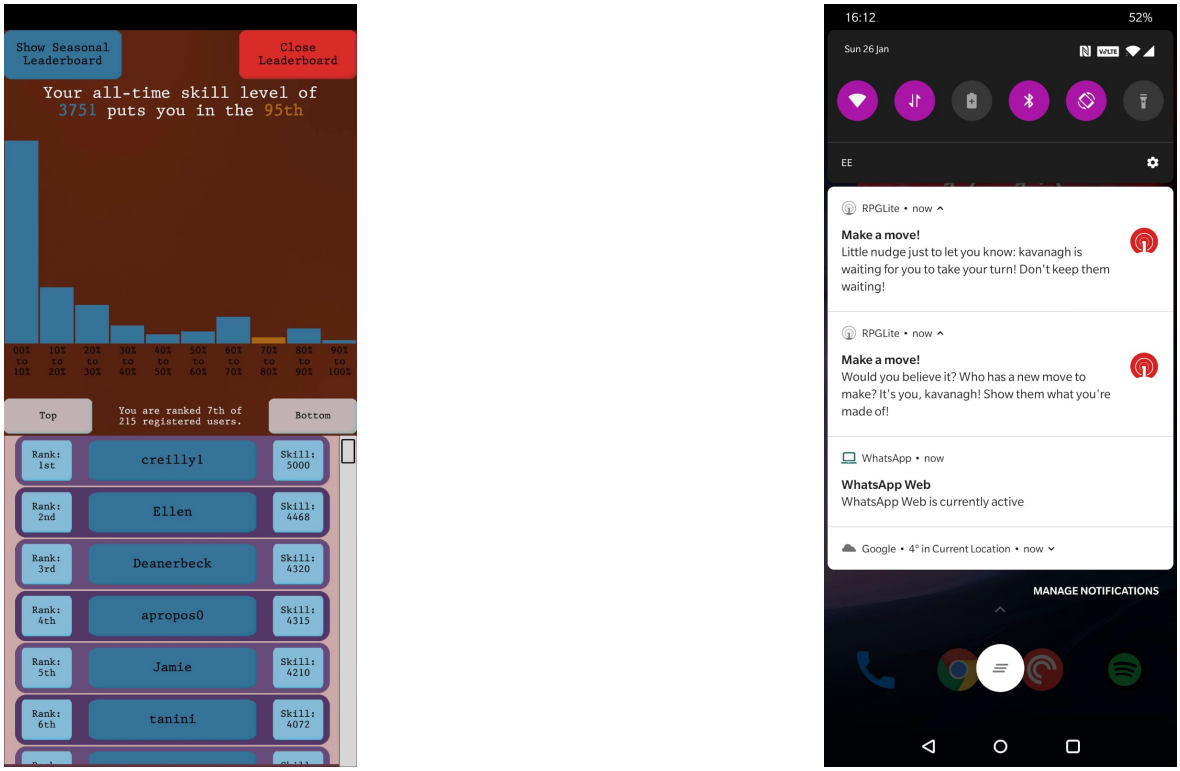

Figure 5.3: Features developed following requests from beta testers. A leaderboard of player engagement is shown to the left; examples of notifications when an opponent has acted are shown to the right.

5.3.2 API & Server-Side Logic

To develop the mobile game in such a way as to allow communication between two RPGLite players remotely, a server and API for the game was required.

A REST API was developed with Python's *Flask* framework. Endpoints were created to support all in-game actions requiring centralised state (or a recording of that state), allowing for player search, matchmaking, player profile design, game history and statistics analysis, ranking calculations, login and password reset, synchronisation of access to sensitive information, and other in-game activities. The API also allowed moves to be made, and rejected erroneous game states or unauthorised input from any malicious actor, to ensure that the data collected, analysed, and published was not manipulated. The API also sent push notifications to an opponent's device when moves were made. This was a feature requested during beta testing, which was reported to increase engagement with the game.

On each of these actions, data was collected about the action performed, and logged in a database. In addition, in-game activities which required no server-side input but were considered to have potential in later analysis would log data points through the API into the same database.

A MongoDB database instance was installed and managed on a University of Glasgow Computing Science virtual machine. The no-SQL nature of the database permitted flexible structuring of the data, and simplified analysis of the games' results after data collection was completed. The API was also hosted on the same virtual machine. A combination of port access rules applied via Nginx and hardening the security of the database itself through MongoDB's account features and controlled permissions within the server prevented unauthorised access to the database, ensuring that the data remained untampered.

Game state was initially interpreted client-side. The intention in this design choice was to unburden a centralised service we were responsible for maintaining, moving functionality

such as calculating game states after moves were created to clients on players' devices. This left the server-side logic mostly concerned with executing database transactions and ensuring the integrity of game states was uncompromised. However, some difficulties arose from this decision: the lessons learned and corrective action taken are explained in Section 5.3.4.

5.3.3 Player Recruitment & Collection of Consent

Players were recruited using a number of methods. Lecturers at the University of Glasgow advertised the game to their students through email. Word of mouth was used to advertise the game through our social and professional circles, and contacts were encouraged to share the game in their own communities. The Scottish branch of the International Game Developers Association advertised the game to their group. A University of Glasgow newsletter for Science & Engineering undergraduate students was used to advertise the game. Figure 5.4 shows player acquisition over time. It is also reflected upon in Section 5.3.4 with regards the factors which impacted successful player recruitment.

Players gave consent to participate in a scientific study as a part of creating an account to play RPGLite. Players were required to explicitly scroll through an information sheet and consent agreement, and were required to agree to both to make an account and play the game. Copies of both were made available to download for player reference in both the game and RPGLite's website [148]. Players were instructed to contact the involved researchers to withdraw participation, and were instructed that they could do so at any time. Email addresses to contact were noted in both the app and on the game's website [148].

The study, information sheet, consent form, and data collected were reviewed and approved by the University of Glasgow Science & Engineering ethics committee before the game was released.

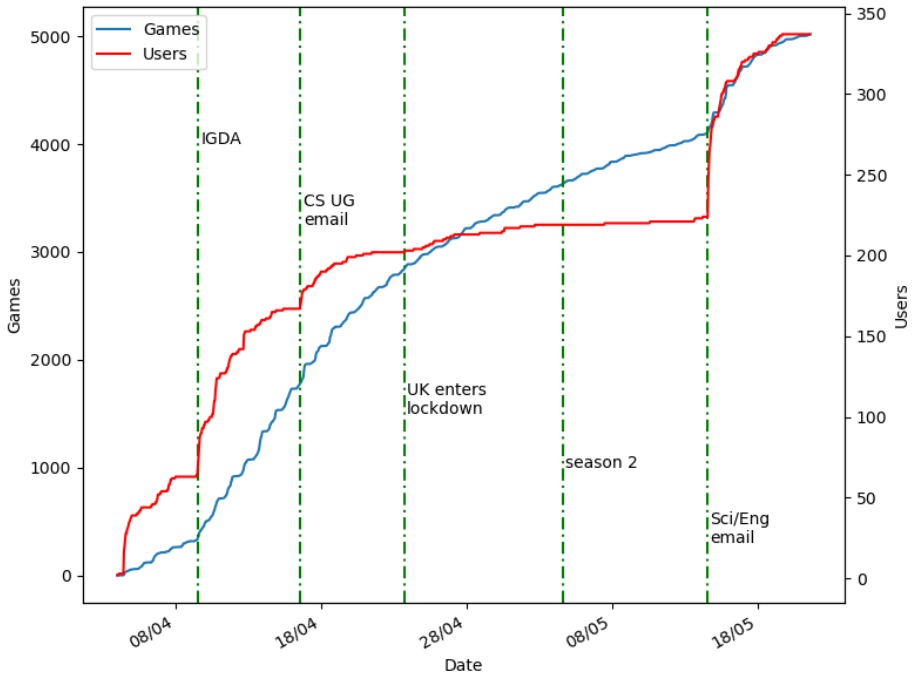

Figure 5.4: Rate of user acquisition in the weeks following RPGLite's release. Important events are marked in the graph and include the promotion of the game via the Scottish branch of the International Game Developers Association, an email to Computing Science undergraduates at the University of Glasgow, the start of the UK's first COVID-19 lockdown, the release of a major update to RPGLite, and an email to all Science and Engineering undergraduates at the University of Glasgow.

5.3.4 Lessons Learned from RPGLite Implementation

Some observations were made about the mistakes made when producing RPGLite, published separately to this thesis [154]. These were documented to support other researchers in their own development processes who were also new to game development, to avoid the need to "learn the hard way".

RPGLite's design was influenced heavily by beta testers, as mentioned in Section 5.3.1. Comments from beta testers formed a feedback loop where rapid iteration on design and features could take place. This was important because the game's release was timed to coincide with Glasgow University's study break for exams, as students would have more time to play when the teaching season ended and students were expected to constitute the majority of RPGLite's community. Faster feedback from beta testers allowed for quicker iterations on designs, and so allowed progress to be accelerated, helping to meet the timeline originally

agreed to.

Recruitment of players was an important part of RPGLite's success, as a diverse community of active players makes the published dataset more useful. The rate of player acquisition and specific events expected to have influenced the game's adoption is shown in Fig. 5.4.

A lesson which was not immediately apparent when developing the game but is significant in hindsight is that it was essential to make use of our personal professional networks to advertise the game and build a community. The game was shared by the chair of the Scottish branch of the International Game Developers Association, which saw a large increase in player count. The largest increase appears to have been advertising the game to all Science & Engineering students at the University of Glasgow. Other events were expected to cause an increase in RPGLite's community: we believed that the UK entering its first COVID-19 lockdown would encourage players to suggest the game to their friends, but no significant increase in player account appears to have been caused by this. The release of RPGLite's second season was also expected to create some excitement in the community which might have encouraged the game to be shared in players' social circles. However, no discernable increase in player count appears to be attributable to the release of the new season. The most significant factor in RPGLite's accumulation of players therefore appears to be active promotion through our available networks, rather than players promoting the game within their own social circles.

RPGLite was designed to validate game state client-side, and used the server it communicated with largely as a mechanism to communicate updates to game state. This minimised the work involved in developing RPGLite's server and API. However, the Unity-based mobile game proved more difficult to debug than the Python-based server, and logic was duplicated across both codebases as the server had to validate game state — a task originally intended for the mobile client — to prevent players cheating by crafting requests to the server which would update a game with an inauthentic state. Updates to the server were also easier to

control than updates to the mobile game, as deploying a new version of the server was simple but deploying a new version of the app required all players to update, which was outwith our control as developers. As the server was ultimately responsible for RPGLite's security, had to include all game logic, was easier to develop than the mobile game, and additions to the game were more easily deployed through the server than through the mobile client, development effort shifted toward the server and away from the mobile client as we gained real-world experience in developing mobile games.

5.3.5 RPGLite Seasons

RPGLite was played through two "seasons": after an initial 3,000 games played, the game was updated and a second configuration was released. This meant that, having learned a strategy to play with the initial configuration (such as a preference of characters) changes were made which may invalidate what players had learned. The dataset therefore contains two conceptual subsets: data collected from a system with the same mechanics but tweaked parameters, resulting in different optimal character pairs and moves. Parameters for characters in season 1 are given in Table 5.2, in Table 5.3 for season 2, and the differences between the two configurations are summarised in Table 5.4.

Character	*Health*	*Hit Accuracy*	*Damage*
Knight	10	0.60	4
Archer	8	0.85	2
Wizard	8	0.50	2
Rogue	8	0.75	3
Healer	10	0.85	2
Barbarian	10	0.75	3
Monk	7	0.80	1
Gunner	8	0.80	4

Table 5.2: Configuration for RPGLite in season 1.

Character	*Health*	*Hit Accuracy*	*Damage*
Knight	10	0.80	3
Archer	8	0.85	2
Wizard	8	0.50	2
Rogue	8	0.70	3
Healer	9	0.90	2
Barbarian	9	0.70	3
Monk	7	0.75	1
Gunner	8	0.70	4

Table 5.3: Configuration for RPGLite in season 2.

In the second configuration, most character pairs have their hit accuracy reduced by 0.05 percentage points. Only the knight's damage is reduced, from 4 points to 3. Health is set to 9 for the Archer, Healer, and Barbarian. This change alters RPGLite's metagame, as making

certain characters weaker and others stronger incentivises different strategies of play: different character pairs and moves are optimal as a result of the changed configuration. Player response to changed configuration is analysed using formal methods by Kavanagh [71].

Parameter	*Character*	*Old Value*	*New Value*
Hit Accuracy	Knight	0.60	0.80
	Archer	0.85	0.80
	Rogue	0.75	0.70
	Healer	0.85	0.90
	Barbarian	0.75	0.70
	Monk	0.80	0.75
	Gunner	0.75	0.70
Damage	Knight	4	3
Health	Archer	8	9
	Healer	10	9
	Barbarian	10	9

Table 5.4: Altered values of different parameters when moving from season 1 to season 2.

5.3.6 Data Collected

In total, 360 players produced a dataset of 9,323 games, 170 of which were in progress when the dataset was created from a snapshot of the MongoDB database used to develop the game. This snapshot was taken in November 2020, around 8 months after the game was initially released. The dataset includes 1,069,595 data points generated by gameplay or player interaction with the client, such as players checking their position on a leaderboard, searching their game history, or rolling to hit as part of an attack. These data points are ancillary to the games themselves and mostly record interactions players made with the mobile game. These were initially collected for diagnostic purposes, but were published as part of the dataset to support future research, should they be useful for other research projects.

This data is published and available in the format of a JSON object [74]. The object has three attributes: `"games"`, which contains a record of the 9,323 games played by RPGLite players; `"page_hits"`, containing data points describing interactions with the RPGLite app; and `"players"`, which contains every username registered to play RPGLite.

String identifier for "kind"	*Contents of data point specified by "kind"*	*# data points*
`rolls_fastforwarded`	Player fast-forwarded the animation of a roll determining the outcome of an attack	187,797
`home_to_gameplay`	Player viewed a game from their home screen	100,071
`hs_refresh_pressed`	Player refreshed their list of active games to check for updates	24,798
`tabbed_to_find_opponents`	Player used the "Find games" tab to find a new opponent to challenge to a game	15,531
`homescreen_to_leaderboard`	Player moved from their home screen to the game's leaderboard	7,394
`homescreen_to_profile`	Player checked their own profile from their home screen	3,005
`game_abandoned`	Player abandoned a game	2,018
`leaderboard_user_zoom`	Player investigated another player discovered from the leaderboard	1,385
`challenge_sent`	User challenged another user to a game	1,291
`customise_tag`	User customised the tag displaying their username in a game	114
`notifications_toggled`	User toggled notifications on or off	71

Table 5.5: Some types of data included in the 1,069,595 "page hit" data points, which mostly describe interactions with the RPGLite app.

The `"players"` attribute of the dataset contains only a list of usernames, represented as strings.

The `"page_hits"` attribute contains a list of "page hit" objects. The "page hit" name for these attributes is a historical artefact of the game's development, and — while esoteric — is not a description of the data itself, which mostly concerns app interaction. The specific interaction described by each data point (rather than the history of a specific game) is denoted by a "kind" field within each data point. 63 such kinds are included in the dataset. Some examples are described in Table 5.5.

The abridged list of different page hit data points described in Table 5.5 contains some kinds of particular interest. Refreshing the list of games on the home screen occurred frequently: with 24,798 data points logged, it was the eight most common interaction with the game. This is surprising, as the list of games refreshed automatically when game states changed (i.e. when an opponent made a move), and players were sent notifications if they were not logged into the game. Future research could investigate whether players who checked on their games' statuses were particularly likely to engage with RPGLite for a long period of time, or whether engagement with their list of games correlates with a player's degree of optimal play. These events also reveal information about how players engaged with the application's features: players moved to the "Find games" tab around twice as often as they moved to the leaderboard. Notifications were toggled only 71 times, meaning that a maximum of 19.72% of players could have disabled their notifications, which were on by default. As some players likely re-enabled notifications, the proportion is expected to be lower than this estimate. This either implies that most players did not mind receiving notifications from RPGLite, or that they managed their notifications some other way (such as through their phones' operating system settings) or did not play enough to notice the notifications. More investigation is required to determine insights about RPGLite's community and their interactions with the application from these data points.

Finally, the `"games"` attribute of the RPGLite dataset contains a list of JSON objects describing the state of a game. Each object contains similar attributes describing the entire history of the RPGLite game played; attributes of particular interest are highlighted in Table 5.6.

Moves stored in game objects are encoded in a string format similar to chess move notation. A move notated as `p1Gp2A_65` reads as, *"Player 1's Gunner attacked Player 2's Archer, rolling a 65 to hit"*. Values for rolls range from 1 to 100 and are compared against the attacking character's chance to hit in the configuration being used to determine whether the attack is successful: an attack is a success if $roll > 1 - (acc \times 100)$, where *roll* is the value rolled to hit

Attribute	*Description of attribute*
`usernames`	A list of usernames for the players in the game
`winner`	Only exists for completed games. A number indicating which user in the list of usernames won the game, where 1 is the first username and 2 is the second.
`Moves`	A list of moves made in the game. Moves are written in a chess-like notation explained in Section 5.3.6.
`p1c1`	Player 1's first selected character
`p1c2`	Player 1's second selected character
`p2c1`	Player 2's first selected character
`p2c2`	Player 2's second selected character
`p1_abandon`	A boolean representing whether p1 forfeited the game
`p2_abandon`	A boolean representing whether p2 forfeited the game
`elo_scores_at_end`	A mapping of usernames to floating point ELO scores, a player ranking metric common in games like Chess [37]
`balance_code`	A string value identifying the RPGLite configuration (or "season") the game refers to. Season 1 games have no such attribute, and so are identified by the lack of a balance code. Season 2 games have the balance code `"1.2"`. No other seasons of RPGLite were released.

Table 5.6: Examples of attributes common to records of RPGLite games as published in the RPGLite gameplay dataset [74].

and *acc* is the accuracy of the attacking character. The notation is composed of the following segments:

Active Player	Attacking Character	Opponent	Opponent's Targeted Character		Roll To Hit
p1	G	p2	A	_	65

The Healer and Archer characters produce variations on this notation due to their special abilities. The notation is extended in each case. The Archer can attack two characters when they move, and rolls for each target must be recorded. An example of the notation describing each of an Archer's rolls is `p2Ap1G_41p1R_25`, which reads similarly to the original example, but contains a second target at the end. The healer includes both the targeted opponent character to deal damage to and the character belonging to the attacking player which the Healer will heal if the attack is successful. An example of the Healer's notation is `p2Hp1Gp2A_48`, which describes a Healer attacking an opponent Gunner and healing the active character's Archer. The notation extends similarly for other characters.

5.4 Summary

A mobile game, RPGLite, was implemented and distributed to collect data for research purposes. It followed an existing game design to support formal analysis through model checking [75]. The dataset produced by the game [74] was released to the public after around 8 months of play from 360 players who participated in 9,323 games. Players were recruited using the University of Glasgow's outreach channels to advertise the game to undergraduate students in particular. The data collected from these players' games and interactions with the application include specific moves made within games, the characters chosen, players' use of application features such as its leaderboard and challenge system, and players' ELO ratings (a popular metric to rank players of games, originating in chess [37]).

5.4.1 Contributions

While this chapter contains some research contributions, its main purpose is to describe the research software engineering required support the contributions of Chapters 6 and 7.

Contributions presented in this chapter (aside from the game itself) are twofold. First, the development of RPGLite yielded a dataset of real-world play produced by its players [74]. This describes real-world interaction with RPGLite, which is useful in comparing the game's formally verifiable properties with players' assumptions or learnings. This has supported both the research in Chapters 6 and 7 and research in the game design community [71, 73]. It is hoped that the dataset will continue to support future work related to RPGLite and game design generally. In addition, this chapter also contributes some observations on the development of mobile games for research purposes as outlined in Section 5.3.4 and discussed in more depth in a separate publication [154].

However, these contributions are a result of the chapter's focus and main contribution: the development, maintenance and distribution of a mobile game to collect datasets for research purposes, in particular the game's server-side component, networking, and hosting. This research software engineering effort is a prerequisite for validating models of RPGLite players learning to prefer certain character pairs over others. Further engineering efforts are presented throughout Chapter 6, which explains the process by which aspect-oriented models of RPGLite play are designed and developed. Results contributed by the validation of those models are given in Chapter 7.

Chapter 6

Designing and Implementing Aspect-Oriented Models

This chapter is the first of two chapters describing a set of three experiments which investigate the use of aspect-oriented programming in a simulation & modelling context. These experiments share a technical foundation which alone constitutes a significant contribution. Therefore, contributions related to these experiments are split into two chapters: the first focused on research software engineering, and the second discussing the experiments that engineering enabled and the results produced by those experiments. Details of both these experiments and of the structure of this chapter are given in Section 6.1.

This chapter presents contributions in the form of research software engineering. It explains the design and implementation of a model of RPGLite play. This is followed by the design and implementation of aspects which instrument that model for experimental observation, correct simulated play in some edge cases, and implement a model of learning which is used to direct simulated players' selection of character pairs. The second, Chapter 7, describes individual experiments which make use of the aspect-oriented models designed and implemented in this chapter. Together, these two chapters detail all contributions produced

by the work performed to conduct the experiments.

6.1 Introduction

This chapter describes the design and implementation [149] of experiments which investigate using advice to augment behaviour in a model. The experiments simulate RPGLite gameplay. Using these simulations, the experiments investigate the utility of aspect-oriented programming in a simulation & modelling context. Advice which uses data about players' gameplay [74] is applied to a basic model of RPGLite play. The advice changes the behaviour of simulated players in an attempt to reflect realistic interactions with the game. The behaviour modified by the advice is players' selection of character pairs. Character pairs selected by simulated players are compared against those from a player's gameplay data and their correlation is measured. If the datasets correlate, then players are successfully simulated. This approach is used to answer the research questions proposed in Chapter 2:

> **RQ1** *Can models of systems more accurately reflect their subjects by weaving aspects which represent improvements?*
>
> **RQ2** *Can advice be used to faithfully introduce behaviours or parameters into a model which were not originally present in it?*
>
> **RQ3** *Can advice be used as a portable module, such that aspect-oriented improvements to one model can be woven into another without loss of performance?*

To answer these, two approaches to modifying player behaviour are developed: one alters the distribution from which character pairs are selected to reflect the distribution found in a player's gameplay data, and the other models the player learning to play RPGLite and developing preferences for certain character pairs in the process.

The experiments share a significant proportion of their design and implementation. As their common elements require a significant amount of explanation, a chapter outlining both the setup *and* results of these experiments would be extremely long, and their shared

foundation means that they are most naturally explained together. For this reason, the design and implementation of *all* experiments is explained in this chapter, and their results follow in Chapter 7.

6.1.1 Contributions

This chapter contributes research software engineering in the form of the design of aspect-oriented models. These models include aspects which implement the observation of experiment state as suggested by [55], corrections to modelled behaviour to account or edge cases unresolved in RPGLite's design, and changes to modelled behaviour including a model of learning.

This chapter should be understood as the first of a pair of chapters investigating the applicability of aspect-oriented programming to models for research purposes. It describes the engineering and design required to support three experiments, which are discussed in Chapter 7. The chapters are separated for the purposes of legibility and document structure; because of this, they effectively share contributions and should be understood as two halves of one research effort.

6.1.2 Chapter Outline

The common aspects of design and implementation in this chapter are explored as follows. To give context as to how different pieces of the experiments' foundations are used, an overview of the models constructed and the experiments employing them is given in Section 6.2. The model of RPGLite play which these experiments weave aspects into is described in Section 6.3. The design of the more complex of the two models of behaviour change — a model of learning — is explained in Section 6.4. Aspects implementing the learning model, other models of behavioural change, and additional apparatus required to conduct experiments are described in Section 6.5. Having described details of the aspects used by experiments, details

of the implementation of the experiments themselves are given in Section 6.6. Following this, Chapter 7 explains the design, implementation, results and evaluation of each experiment individually, as all context necessary to understand these designs and evaluations are given in the aforementioned sections.

6.2 Experiment Design

To answer the proposed research questions, two models are made which change simulated players' character pair selections: one which reflects the choices made by the player being simulated, and another which simulates that player learning over time. These are used to examine three things about aspect-oriented simulations & models: whether aspects can be woven to improve a model's accuracy; whether it is feasible to use aspects to introduce new behaviours and parameters in a model; and whether the same aspects can be successfully woven across different models.

The complete designs for the first, second, and third experiments are explained in Section 7.2, Section 7.3, and Section 7.4 respectively as it is useful to have discussed the implementation of the RPGLite model, aspects, and statistical methodologies before the experimental design is explained. However, some context as to how those foundations are used is important when explaining them too. For this reason, a brief description of the behavioural modifications and the experiments which use them are given here, and more detailed descriptions are given in the following chapter.

6.2.1 Changes to Behaviour

Simulated players' behaviour is changed using aspects when they select characters. The rationale for altering character selection in particular is explained in Section 6.4. The model they alter — described in Section 6.3 — is "naive", meaning that it makes all decisions a

player could make randomly rather than strategically or optimally. Two changes to character pair selection are created: the first selects characters with the same distribution as found the gameplay data of a player being simulated, called the "prior distribution" model; the second selects characters by modelling players learning which characters are most likely to win games and selecting character pairs according to what they learned, called the "learning model".

Prior Distribution Model The prior distribution model selects character pairs with a distribution calculated from players' gameplay data.[1] As the distribution is already known, simulated games can reproduce the distribution. This makes little change to the behaviour of simulated players, as no new activity is modelled on their part: no additional actions are taken and simulated players do not behave in a "new" way. However, the emergent properties of simulated play should reflect the dataset used by the prior distribution model. This is expected to have the effect of synthesising new datasets with the same emergent properties and requires little modification to the model. The implementation of the prior distribution model is given in Section 6.5.3.

Learning Model The learning model selects character pairs by preferring pairs which previously won games. This model requires an additional model of confidence to support it (discussed in Section 6.4.4) and special measures to be in place when generating datasets to ensure the space of possible character pairs is explored properly (discussed in Section 6.6.4). As a result this explanation is superficial and is provided as necessary context for the introduction of experiments following in Section 6.2.2. A thorough discussion of the learning model is given in Section 6.4.

The learning model selects character pairs using a distribution defined by the pairs' number of observed wins. If a player has never seen a character pair winning a game, that pair is unlikely to be selected; if a player sees a pair winning an overwhelming number of games, the pair will be selected proportionally to its win rate. This model introduces new behaviours for simulated players, as their decisions are based on historical observations

[1] The distribution is known ahead of the simulation being run, hence "prior".

which are not present in the original model and emerge from a player's interactions with others as opposed to being defined when the model is first executed. New parameters are also introduced to control how players learn, including a parameter controlling their arrogance, a parameter scaling the number of games required for them to rely on their knowledge, and parameters controlling how bored they are and how likely they are to stop playing RPGLite. The introduction of new behaviours and parameters to the naive model allows the latter two research questions to be answered.

6.2.2 Brief Explanations of Experiments

Experiment #1: Improving a Model The first experiment answers the research question: *Can models of systems more accurately reflect their subjects by weaving aspects which represent improvements?* Aspect-oriented programming can only be used to augment models if modifications to a model can be encoded in aspects, woven into a model, and become provably more accurate as a result. Similar research was conducted in prior work [150], but this study represented changes to a model which were not verified as "accurate" or "realistic". Beyond producing a change that *looked* accurate, it did not simulate any real-world system and so could not be verified as rigorously as a model of RPGLite gameplay can be.

To verify that a model is changed in the intended way through the weaving of advice, three datasets of completed games are used for each player simulated. The first is that player's gameplay data collected from their interactions with the mobile game in RPGLite's first season, the second is produced by the naive model, and the third is produced by the naive model with the prior distribution model woven. The distribution of chosen character pairs from each model is compared with the player's distribution of chosen character pairs using a correlation metric explained in Section 6.6.3. The random distribution of chosen pairs produced by the naive model is expected to correlate poorly with the player's distribution of choices, as the player is expected to be biased

toward certain character pairs as they continue to play RPGLite. The prior distribution model should produce datasets with the same distribution as the player exhibited when interacting with the mobile game, so the dataset produced with the prior distribution model woven into RPGLite should correlate closely to the empirically sourced dataset. If this behaviour is seen, then the aspects woven into the model induced a change in simulated players' behaviours and so successfully augmented the model.

A more thorough description of this experiment is given in Section 7.2 as the reader has been presented with the details of the experiment's foundation at that point.

Experiment #2: Extending Behaviours in a Model The first experiment investigates whether advice can alter a model and introduce changes to existing behaviours. Another experiment follows to investigate whether advice can introduce *new* behaviours to a model and new parameters which alter modelled behaviour. This aims to answer the research question: *Can advice be used to faithfully introduce behaviours or parameters into a model which were not originally present in it?*

The learning model is woven into the naive model of RPGLite play to add new behaviours and parameters to the model. For each player, parameters are found using a technique explained in Section 7.1.2 which simulate their learning most accurately. This technique determines reproducible results which show statistically significant correlation with the player's gameplay data from the RPGLite mobile game, extending the statistics used in the first experiment (explained in Section 6.6.3). If the learning model can reproduce the play style of some real-world players, then the additional behaviour and parameters successfully model their play of RPGLite and the experiment is a success. To do this, datasets are generated by weaving the learning model parameterised in different ways. The correlation of these datasets is measured against subsets of a player's season 1 gameplay data. If the parameters can produce data which correlates with the player's gameplay data consistently, the player is accurately simulated by the learning model with these parameters and so the research question is answered.

Note that not all players are expected to be simulated accurately by the model of learning. For example, players might have biases toward or against some characters, might play in cliques which limit their exploration of possible character pairs, or might learn differently to the way the model assumes such as exhibiting temporal discounting [54]. As these experiments aim to investigate the feasibility of applying aspect-oriented programming in a simulation & modelling context, building a formalism of learning which is universally applicable to all players is beyond the project's scope. The experiment only seeks to demonstrate that new behaviours *can* be accurately represented by advice, and that the technique can be used in future work to augment models in whatever manner a research team requires.

A more thorough description of this experiment is given in Section 7.3, as the reader has been presented with the details of the experiment's foundation at that point.

Experiment #3: Behaviours as Cross-Cutting Concerns Having conducted the first two experiments, the utility of aspects as units of behaviour change are established. The results in Section 7.2 and Section 7.3 are positive. One more research question remains to be addressed. The final research question is: *Can advice be used as a portable module, such that aspect-oriented improvements to one model can be woven into another without loss of performance?*

To establish this, season 2 of RPGLite is used to represent a system which is subtly changed. RPGLite's seasons are defined by game configurations, which means that the behaviours of players do not change but the strengths and weaknesses of different characters do. To play strategically, players must learn which changes have affected their ordinary strategies and re-evaluate their preferred character pairs. The prior distribution model and learning model should work independently of the RPGLite season they are applied to, and so — given positive results are found for the first and second experiments — the models ought to work when woven into a model of the second season too.

Data is generated using the prior distribution model and learning model in the same

manner as in earlier experiments, but applied to player's gameplay data from RPGLite's second season. Parameters which yielded statistically significant results for the learning model in season 1 are used in season 2, as these parameters ought to represent how an individual player learns. The learning model may also work to simulate season 2 players, but with different parameters; to investigate this, model parameters specific to season 2 are found which yield statistically significant datasets using the same technique as in the second experiment. If these aspectually-augmented models successfully simulate players then the experiment establishes the portability of aspects representing behavioural change. Correlation is measured in the same ways as in previous experiments to evaluate the models' accuracy in simulating players.

A more thorough description of this experiment is given in Section 7.4 as the reader has been presented with the details of the experiment's foundation at that point.

6.3 Naive Model

A naive model of play was developed by separating each stage of the actions taken by players and separating them into individual procedures. The model was written as a workflow in Python, and state of workflow execution was separated into three components: the actor that a function invocation (or "step") represents activity from; a "context" in the parlance of languages like Golang, representing the state of a game being played; and the environment in which that game is played.[2] The naive model of RPGLite follows a simple workflow mimicking player interaction with the mobile game deployed. A graphical representation is provided in Fig. 6.1. Figure 6.2 contains a diagram showing how the naive model is used to produce synthetic data by simulating repeated gameplay.

Two randomly-selected players repeatedly select characters to play from the pool of 8

[2]Incidentally, this structure allowed a flexible and natural implementation of a procedural simulation containing concepts common in software engineering (such as contexts) and environments (found in simulation frameworks). We imagine that it is easily adopted in existing simulation frameworks such as SimPy[97]. Some additional detail is included in Appendix A.

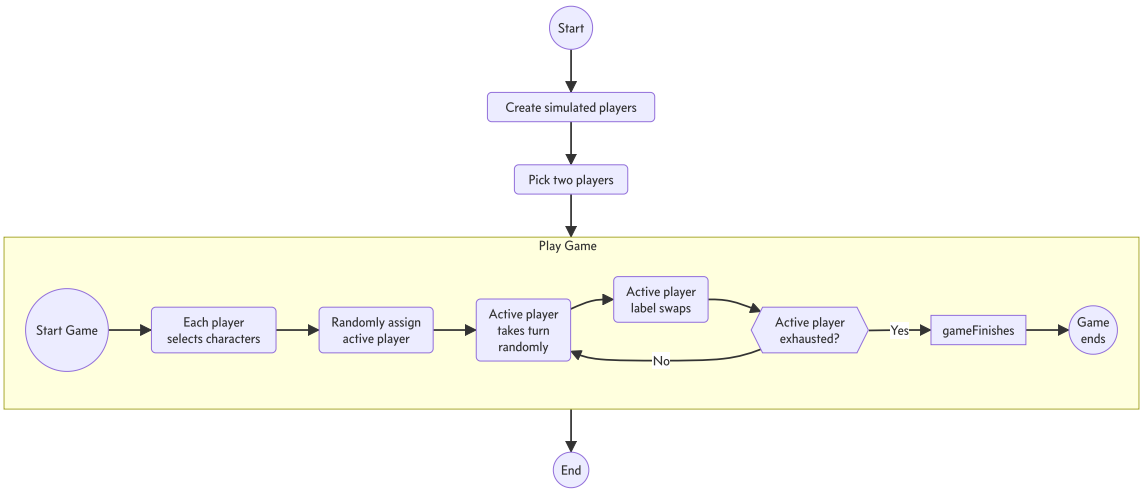

Figure 6.1: A UML activity diagram of the "naive model" of RPGLite play used in experiments.

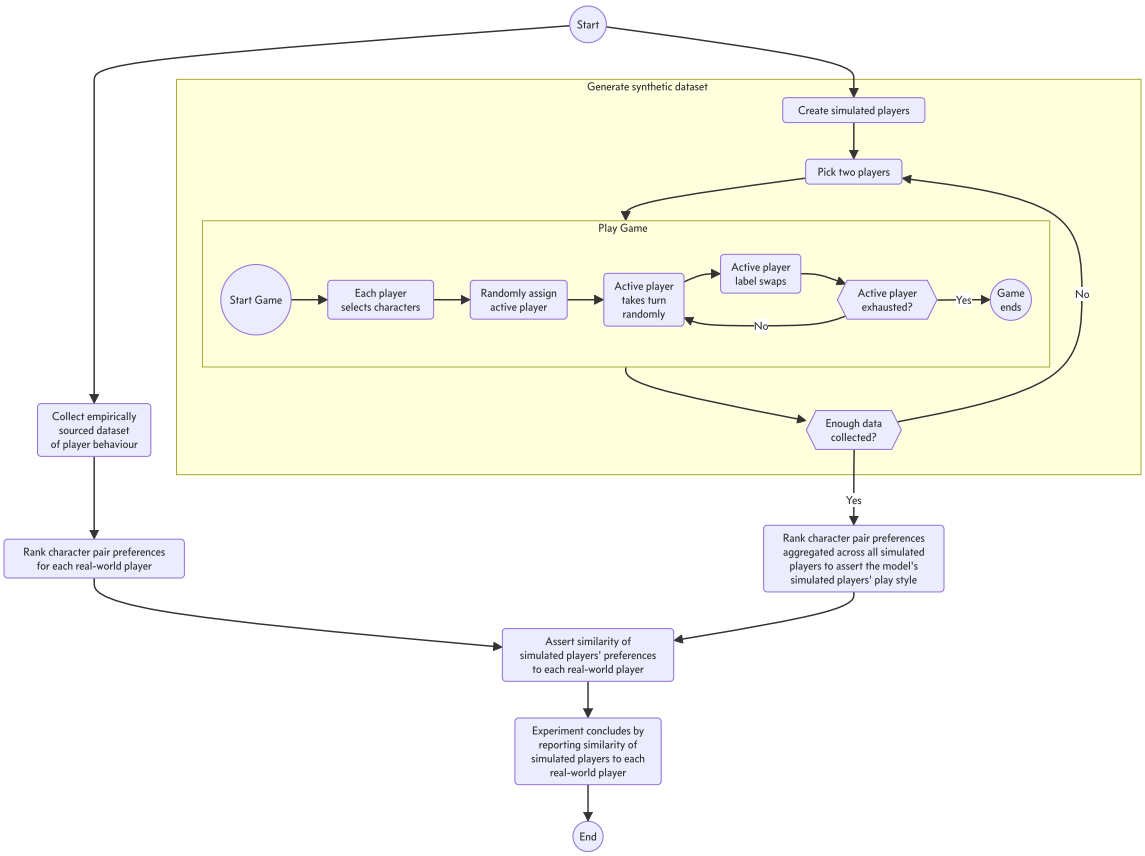

Figure 6.2: A UML activity diagram illustrating how the naive model is used to generate datasets used in experiments.

characters described in Chapter 5, and a player is chosen to play first at random, referred to as the "active" player. That player selects a random valid move to make. The active player alternates, and the process repeats, until such time as an active player starts their turn with both of their characters fully depleted of health. The player with remaining characters is the victor. When used to generate datasets for analysis, another game is started by picking another pair of random players and starting a game between them; this continues until a predetermined number of games has been played. After a sufficient number of games are played, analysis of the generated datasets is performed. [3] Decisions made by players in the model are random; this is because the model is designed to avoid informed decisions where possible. This model illustrates the steps taken by players and disregards whether the simulated players' choices reflect realistic ones. Informed decisions are woven as aspect-oriented changes to behaviour.

6.4 Modelling Learning

The naive model makes choices randomly. Informed decisions are made by weaving advice into the model which changes player behaviour. Experiments are evaluated through the distribution of character pairs found in gameplay data, so aspects are required which change that distribution of character pairs. To change the distribution of character pairs chosen, an aspect-oriented model of learning is created which alters player behaviour to select character pairs at the beginning of a game by inferring strong characters from those observed to win games previously. This section describes its design.

6.4.1 What Players Learn

Players learn many things when playing a game. Other than learning about characters and their strengths and weaknesses, players also learn how to take successful turns by making moves strategically. Character pair selection is the focus of these experiments because they

[3] Dataset analysis is explained in Section 6.6.

are simpler and have a smaller state space for players to understand: the "metagame" for character pair selections is the simpler of the two.

A metagame is a community's perception of "good" gameplay at a given point in time. For example, if certain characters are popular, players may select other characters which are not likely to win against *most* characters but are likely to win against *currently popular* ones. As a result, players may select weak characters strategically, and strategically selected characters change over time in response to changing popular choices in the community. Similar reasoning applies to move selection. For a more thorough discussion of the concept of a metagame, see the survey of literature and definitions by Kokkinakis et al. [84] or the original but more theoretical work on the topic by Howard [64].

A design consideration of RPGLite is that its metagame is "solvable", meaning that there exists an objectively optimal choice for a player to make in any state [71]. As the space of possible moves is much larger than the space of possible character pair choices players are expected to learn optimal choices more quickly, which would lead to an unchanging ("stable") metagame. In practice, the character selection metagame converged on optimal character selections, whereas players consistently made errors when selecting optimal moves [73]. As players' choices of character pairs is therefore easier to model (and more consistent between players), a model of learning is applied to their character pair selections instead of their move selections. Player behaviour is altered to select objectively optimal moves in every state instead of this being learned over time; this is described in Section 6.5.1.

6.4.2 Literature regarding Learning

Different people learn in different ways. Indeed, no universally-accepted definition of learning appears to exist. This is presumably because it is convenient to define what it means to learn differently in the context of different pieces of work. For example, cognitive models of learning can be useful when considering mental processes specifically, whereas functional

models of learning lend a more empirically applicable perspective. What it means to learn is outwith the scope of this research; however, the experiments presented in Chapter 7 include a model of learning. To justify our model, we consider a functional approach to learning, as this is more closely linked to the empirically focused work of simulating real-world behaviour than cognitive alternatives.

Lachman [86] summarises standard definitions of learning as "[…] a relatively permanent change in behavior as a result of practice or experience". They observe that these definitions have practical shortcomings such as a focus on behavioural change (as learning may not change behaviour) or conflating learning's process and its product (the process by which we learn is not obviously identical to its result, of which behavioural change is an example). They suggest learning might be better defined as:

> […] the process by which a relatively stable modification in stimulus-response relations is developed as a consequence of functional environmental interaction via the senses […] rather than as a consequence of mere biological growth and development [86].

They note that their definition distinguishes learning from phenomena such as injury, changes to one's maturity, or sensory adaptation, incorporates stimulus-response relationships the research community consider as learned, and differentiates learning's process and product. Their model is inherently functional, making it useful for the purposes of simulation and modelling, although they offer only a definition of learning and a brief comparison to the standard textbook definition they introduce. The work presented is not intended to demonstrate its improved model of learning empirically, only to discuss its semantic merit. However, the models proposed in this thesis require only a theoretically informed, sound basis for their model of learning, and a lack of empirical justification is not a barrier to the relevance of the model Lachman proposes.

De Houwer, Barnes-Holmes, and Moors [29] propose a functional definition of learning

which is primarily concerned with providing a definition of learning which is both accurate and useful for the purposes of cognitive learning research. Doing so attempts to provide a model around which some consensus can be reached; learning is a central concept in psychology, and they describe their definition as supportive of cognitive work without requiring a cognitive model. They introduce their definition as:

> Our definition consists of three components: (1) changes in the behavior of the organism, (2) a regularity in the environment of the organism, and (3) a causal relation between the regularity in the environment and the changes in behavior of the organism.

This model of learning contains more nuance than the "textbook definitions" of learning they paraphrase as "a change in behaviour that is due to experience" but does not stray far from the core concept of an environmental stimulus impacting behaviour in a causal fashion. The introduction of "regularity" to their definition refers to the presence of the stimulus with some form of repetition, either through multiple instances of a stimulus at different times or the same stimulus occurring concurrently. De Houwer, Barnes-Holmes, and Moors [29] observe that their model is straightforward without the sweeping inclusivity of the simple model mentioned earlier and is easily verified (although, as in the work of Lachman [86], empirical verification is omitted in favour of semantic analysis).

Aside from other benefits more particular to their research community, these benefits are especially useful from the perspective of modelling learning in the case of RPGLite. A simple functional definition can be captured in a software model and introduces fewer opportunities for misunderstanding or misapplication than a more complex or theoretical model. It also introduces concepts such as regularity and causality other definitions do not. We therefore adopt this definition as a basis for our model of learning.

6.4.3 Modelling Learning Character Pair Selection in RPGLite

We use De Houwer, Barnes-Holmes, and Moors [29]'s definition to define an aspect-oriented model of learning that can be applied to the naive model of RPGLite. Their definition of learning gives criteria that this model must meet: players' behaviour must be influenced by their learning to meet the definition's first criterion; repeated experiences in successive games must influence the direction of players' learning is required by the second; and the third criterion requires that this must happen in a causal manner. To meet these, the learning model's design should model a causal relationship between a player's observation of successful character pairs and their future choices of character pairs.

To fulfil these requirements, a model might draw on previously successful character pairs to determine future ones. One approach is to model learning as consistently playing the character pair which most *recently* was observed to win a game. A completed game must have a winning pair,[4] and we can select this pair when playing future games until a different pair is observed to win instead. However, this does not align with one's intuition of how players *would* engage with a game in the real world. A player seems unlikely to be deterred from a strategy they believe is ideal when RPGLite's random nature gives them an outcome they could percieve as unlucky. We can expect players to understand that *perfect* play might not be *winning* play: in some games, the right moves might not lead to a successful outcome due to moves randomly missing opponent characters. Equally, players may take time to become confident in a strategy. We would expect a player to explore character choices before settling on a preferred pair early in their experience, and would expect experienced players to choose characters based on what they have learned through their experience rather than continuing to explore their options. From these observations, we can see that:

① There are scenarios where players can be expected to observe wins/losses *without* incurring behavioural change.

[4] Or be forfeited, in which case the previously winning pair could be substituted.

② Players' confidence in what they have learned can affect their inclination to rely on that knowledge when making decisions.

③ What players learn in successive games would have a small impact in their early experiences, but an increasingly significant impact proportional to their experience in the game.

The model of learning used to simulate RPGLite players can be explained following a similar structure: ① implies that players' observations regarding winning characters is separate from behaviour change; ② implies some mechanism determining players' inclination to use their knowledge when choosing characters (rather than exploring their options); and ③ implies that their inclination to rely on their knowledge instead of exploring the state space is proportional to the amount of experience players have.

To fulfil requirement ① and separate win/loss observation from behaviour change, a player's assessment of how likely a character pair is to win a game is represented as a probability mass function (*"PMF"*) updated through its own aspect (which is described in more detail in Section 6.5). The PMF maps character pairs to their chance of being selected by a player, and initially tracks all character pairs as having an equal chance of being selected. After every game, the chance of selecting the winning character pair increases, and the chance of selecting any other pair decreases. The sum of probabilities of being selected across all character pairs always sums to 1 (100%). This is implemented as a record of the winning character pairs observed by a player: many ways of producing a PMF from a sequence of wins exist, but for the purposes of explanation, one such method is to take the proportion of wins for every character pair as their probability of being selected. This method produces a valid PMF because the sum of those proportions accounts for 100% of observed wins and losses, so the sum of all probabilities must also be 100%.

Requirements ② and ③ are fulfilled through a separate mechanism to control whether players use this PMF to make decisions. Players are expected to explore possible options early in

their experience, and rely on their observations when they have completed many games. This is because Lachman [86] identifies that the experience of a learning agent draws from "regularity" in their environment, and requiring many games to be played before players' behaviour changes ensures that a regular experience is present.[5] The mechanism controlling whether players will make decisions based on previous experiences is referred to as "confidence", referring to their confidence in their experiences at a point in time. A model of confidence fulfils requirement ②, and ③ is fulfilled by confidence increasing proportionally to players' experience.

Confidence is modelled in these experiments as a monotonically increasing function mapping experience (quantified as games played) to confidence as a percentage chance 0% and 100% that a player determines their character choice based on their observations of wins and losses rather than exploring the space of possible choices. Players are "confident" when making a decision with the probability determined by their confidence model.

If the player is not confident, their behaviour is unchanged and they select character pairs randomly as in the naive model. If they are confident, their behaviour is instead informed by their experiences and they select a character pair from the distribution defined by the PMF that their historical observations define. As the PMF affords higher probabilities to repeatedly winning character pairs, player behaviour is causally affected by the regularity of their experience. This implements a realistic model of learning in fulfiling the requirements of the functional model proposed by Lachman [86].

6.4.4 Modelling Player Confidence

To model confidence we require a function mapping the number of games a player has completed to a probability between 0 and 1, fitting the criteria described in Section 6.4.3. A

[5]Note that the regular experience might be one of the character pair choice not having an impact at all; if the player observes all character pairs winning an equal number of times, the PMF would reflect this and player behaviour would effectively remain random. However, the player would choose each character pair at an equal rate because they would have learned that the choice was inconsequential, so even in this scenario learning occurs.

sigmoid curve fulfils these requirements: it rises monotonically and produces values between 0 and 1. It also notionally conforms to an intuition around "confidence" as a behavioural trait: like a sigmoid, players' confidence starts low and remains so until it reaches some inflection point, after which one's confidence increases more significantly, with the rate of this increase tapering as experience continues to increase. However, not all real-world players might express the same traits in their growth of confidence, and this intuition is not universally applicable: the shapes of players' confidence sigmoids differ in the real world. To answer the proposed research questions, an ideal model of confidence is not required, but one which reflects *some* real-world players *is*. To account for this, the confidence model uses a sigmoid which can be parameterised to alter its shape.

A sigmoid curve is suitable for modelling confidence where other curves are not, because we require a period where players lack confidence and explore their options before growing in confidence and remaining at high confidence thereafter, implying the shape of a sigmoid. They are also commonly used in simulation & modelling. Sigmoid curves such as the logistic [144] or Gompertz [52] are widely used when modelling systems [155], but while they fulfil the role of a monotonically increasing curve with asymptotically low and high initial and final states, the shape of such curves is not trivially modified to fit different players' learning styles. To fulfil the confidence model's requirements it is necessary to find an alternative: different players are expected to exhibit more bullish or timid styles of play, so the curve should be parameterised to account for the behaviours of those individuals.

More flexible asymptotic curves were developed by Richards [121] drawing on growth curves developed by Von Bertalanffy [145], which afford a natural pattern of growth. Richards amends this curve to offer a parameterised growth rate. This can be made equivalent to other curves, including the logistic and Gompertz [45]. This curve allows for a parameterised rate of growth but lacks parameters controlling the points at which growth occurs most rapidly. The relative rate of confidence gain is a separate concern to the point at which such growth occurs: a player might cautiously grow in their confidence until they are already very experienced,

or might bullishly grow in confidence yet plateau early, taking longer to reach complete confidence in themselves than they did to garner an initial increase, regardless of their relative growth in confidence.

The flexibility of a parameterised relative growth rate appeals to the notion that different players would gain confidence at different rates, but the point at which confidence accelerate most must also be controlled. We therefore employ the Birch curve, proposed by Birch [8] for its increased flexibility as compared to the Richards curve combined with its additional parameter used to control the curve's shape. [6] Different players might exhibit different rates of growth in their confidence, and might grow maximally in their confidence at different points in their experience. It is defined by the equation:

$$\frac{dy}{dt} = \frac{ay(K-y)}{K-y+cy}$$

Where c is its curve parameter, K is its upper asymptote,[7] a is its relative growth rate (RGR)[8], and y is the value of the curve at a given point in time. The Birch curve can represent other curves through its shape parameter: at $c = 0$ the curve is exponential, and at $c = 1$ the curve is logistic [8]. As the Birch curve models the properties of different players' confidence through its shape parameter it is a suitable candidate for the model of confidence required by the model of learning described in Section 6.4.3. Its implementation in the aspects altering the naive model is given in Section 6.5.2.

[6] Birch [8] refers to shape to mean the point of inflection of a curve. The point of inflection of an exponential rise to a limit is at its initial point; the point of inflection of the logistic curve is in the exact midpoint of the curve's growth.

[7] This is fixed at 1 for the confidence model, as confidence should never exceed 100%.

[8] The RGR is a common parameter of sigmoid curves and defines the rate at which the sigmoid increases near its inflection point.

6.5 Aspects Applied

To generate datasets for experiments the naive model is augmented by weaving aspects. These aspects fulfil different requirements and can be categorised as either: aspects which instrument the model to simplify experimental observation and prepare the model for use in experiments; aspects which instrument the model to observe its state, assisting the implementation of aspects which change behaviour; and aspects which alter players' behaviour.

Aspects which alter the model to make it appropriate for use in the experiments are described in Section 6.5.1. These include altering the moves made to more closely resemble real-world move selection and handling edge cases which that change introduces. Aspects which instrument the model to observe its state are described in Section 6.5.2. Aspects implementing behaviour change including the learning model are described in Section 6.5.3. A diagram of a game of RPGLite's naive model annotated with aspects and their join points is presented in Fig. 6.3. As the source code for the model is made available separately [149] and can be lengthy, code snippets are provided selectively as examples of aspect implementations where it is helpful to do so. Complete implementations for all aspects can be found in the model's source code [149].

6.5.1 Aspects for Model Improvement

Ensuring the Best Move is Played

Experiments model players' character selection rather than move selection as explained in Section 6.4; however, in the naive model players randomly select moves. This is liable to place players in unrealistically weak positions, as players are unlikely to make obviously poor moves such as skipping a potentially useful turn. This is a concern when modelling players learning to select characters because the learning model relies on a causal relationship between what is observed (characters which most reliably win games) and behavioural change

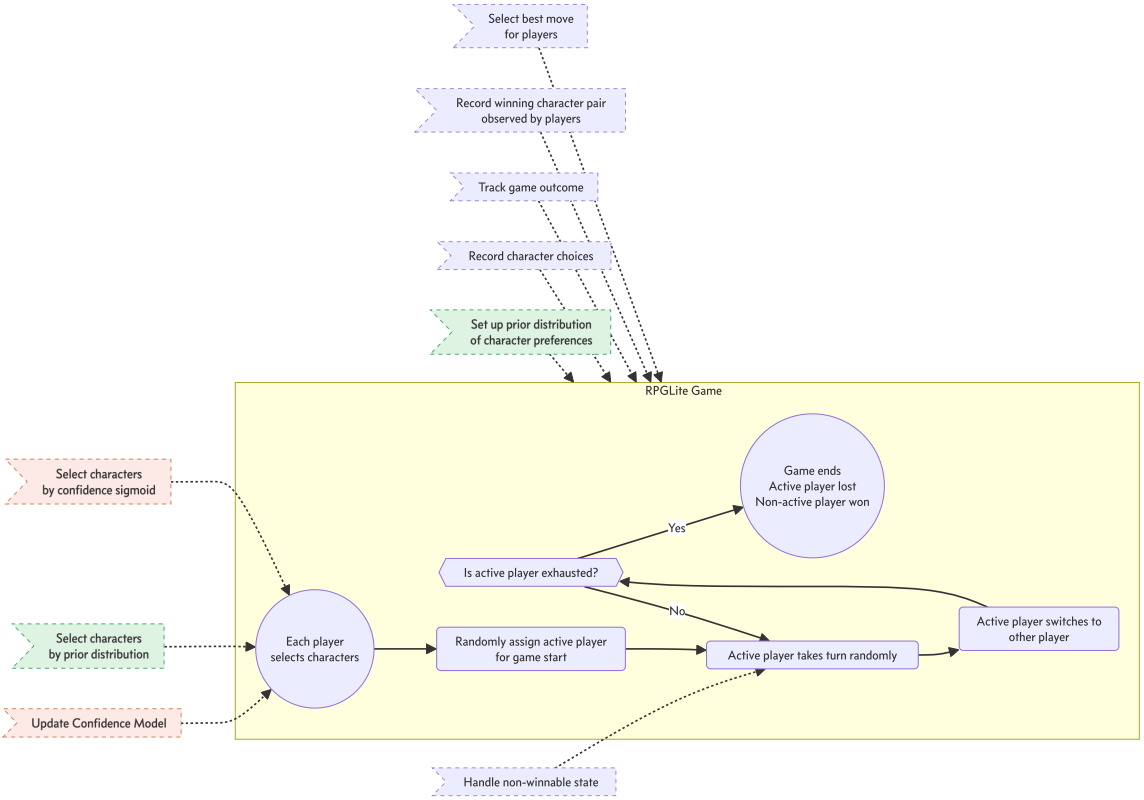

Figure 6.3: A UML activity diagram describing a simulated game of RPGLite, and all aspects woven into the game to implement the various models of learning. Aspects are given dashed borders and applied with a dotted line in the same manner as found in AOPML [13]. Green nodes are used exclusively in the model which selects character pairs using a prior-known distribution (described in Section 6.2.1). Red nodes are used exclusively in the model which selects character pairs using a model of learning (described in Section 6.4).

(players choosing those characters). If selecting random moves causes simulated players to lose games when they would have won them when selecting moves realistically then move selection would affect character selection. Realistic character selection and move selection are therefore related.

Players of RPGLite usually selected moves optimally. In the majority of cases Kavanagh and Miller [73] found that players chose the best move available to them: *"[…] 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."*[9] Move selection can be realistically modelled in around $\frac{3}{4}$ of cases by selecting the best move at every opportunity. This behaviour is implemented in an aspect by performing a lookup on the dataset of action costs defined by Kavanagh and Miller [73] and selecting the known-optimal move in every case. This is woven into the naive

[9]Different seasons of RPGLite are discussed in more depth in Section 5.3.5.

model's `get_moves_from_table` function, which looks up moves from the table of valid moves produced by Kavanagh [71]. It is invoked after the function runs and alters its return value to include only the optimal move, rather than all possible moves. Its source is included in Fig. 6.4 as an example of the aspects implemented for this thesis.

```
def choose_best_moves(target, ret, *args, **kwargs):
    '''
    Replaces a set of possible moves from base_model.get_moves_from_table with
    the single best move, forcing that to be taken. Args:
        target: base_model.get_moves_from_table ret: the list of best moves to
        be taken at the game's current state *args: args for the function
        **kwargs: keyword args for the function

    Returns: a list containing only the best move of all moves in ret
    '''
    gamedoc = args[0]

    if list(map(str, gamedoc.get('moves', [None, None])[-2:])) == ['skip', 'skip'
        ]:
        return ret

    sorted_moves = sorted(ret.items(), key=lambda move: -move[1])
    return {sorted_moves[0][0]: sorted_moves[0][1]}
```

Figure 6.4: An example of the implementation of an aspect for developing experiments around the naive model. This aspect ensures that the optimal move is selected by replacing the return value of `get_moves_from_table` with the optimal move the function returned.

The aspect selecting optimal moves handles another concern of move selection. As simulated players' behaviours are augmented to select universally optimal moves, an anomaly in the dataset was identified. In some games, optimal play results in infinite loops: players can arrive in a state where they would skip turns mutually and indefinitely.

This occurs because barbarians deal more damage once they lose a sufficient number of hit points. As a result, dealing damage to a barbarian can result in their having an opportunity to win in one move rather than two in certain health states. Both players can concurrently exist in this state. Therefore, both players' optimal move is to skip their turn. Kavanagh and Miller note that such states were reached in 64 cases in real-world play but real-world players never skipped their turn. To better mimic real-world play, the aspect returns random moves if the previous two moves in a game skipped. Such a state indicates that the game reached a state where neither player would deal damage to the other when making optimal moves.

Handle Game States with no Viable Moves

A consequence of playing games using only optimal moves is that unexpected states can occur: the dataset of possible moves produced by Kavanagh [71] maps game states to moves a player may make, but not all states exist in the dataset. This is because the dataset was produced through model checking with the aim of identifying the cost of an action regards its impact on a player's chance of winning. If a loss is guaranteed, all moves have a 0% chance of winning; the model checker which produced the dataset of available moves therefore identified that the outcome of the game is already determined in those cases, and produced no possible moves for them.

The winning player in this situation is the player who made the previous move, as the now-active player has no moves with a chance of winning above 0%. This logic is added to the model through an aspect which handles exceptions raised by move selection, catching errors caused by an invalid table lookup. After identifying that the exception indicates that the state does not exist (in this case the relevant exception is Python's `KeyError`), the aspect handles the exception by assigning the losing player's characters 0 health and swapping active players. The game proceeds with the losing player taking the following turn. As this turn starts with the active (losing) player having 0 health, the game ends as expected.

6.5.2 Aspects for Instrumentation

Update Model of Confidence

As described in Section 6.4.4, confidence is modelled as a Birch curve. This aspect updates a player's level of confidence using the Birch curve when a game ends. It is woven as a prelude to the naive model's `choose_character_pair` function, so that confidence is updated for each player before it is used for character pair selection.

The aspect updates a value in the model's environment which represents the confidence of the player selecting a character pair. The aspect uses the model's environment to find parameters such as players' initial confidence or the value of the curve's shape parameter c. The implementation of the aspect updating the confidence model is given in Fig. 6.5. This implementation is edited for readability, as the version used when running experiments contained debugging code and additional logic used by other experiments which were attempted but abandoned.

```
def update_confidence_model(_target, player, _game, environment):

    sigmoid_initial_confidence = environment['special vals']['sigmoid initial
        confidence']

    if 'confidence' not in environment:
        environment['confidence'] = dict()
    y = environment['confidence'].get(player, sigmoid_initial_confidence)

    k = 1  # upper asymptote
    a = environment['special vals']['rgr']

    if environment['special vals']['sigmoid type'] == "birch logistic":
        c = 1  # for logistic curve
    elif environment['special vals']['sigmoid type'] == "birch exponential":
        c = 0  # for exponential curve
    elif environment['special vals']['sigmoid type'] == "birch controlled":
        # the curve value set specifically
        c = environment['special vals']['birch c']
    else:
        # birch, but not specified which one
        raise Exception("Birch equation used, but no specific equation requested"
            )

    # Update value of player's confidence using Birch (1999)
    environment['confidence'][player] = y + a * y * (k - y) / (k - y + c * y)

    return environment['confidence'][player]
```

Figure 6.5: An example of the implementation of an aspect tracking model state. This aspect updates the value of a player's confidence model and is woven as prelude to the naive model's `choose_character_pair` function.

Record Prior Distribution of Character Preferences

To implement the prior distribution model, the character pair preferences of the player being simulated must be calculated. The prelude aspect implementing this is shown in Fig. 6.6. Computing this distribution involves loading, parsing, and filtering the entire dataset of RPGLite gameplay data. However, the program running experiments must load this dataset

to construct the parameters controlling the simulation — in particular `training_data` and `testing_data`, introduced later in Section 6.6.1 — so to avoid duplicating this work, the values are retrieved from the stack frames of the function which constructed the model parameters. The players being simulated and games played by them are identified by their variable names in the stack frame, and their distribution is calculated through a convenience function, `find_distribution_of_charpairs_from_players_collective_games`, the implementation of which is given in Fig. 6.7.

```
def prelude_before_learning_by_prior_distribution(target, actor, gamedoc, env, *
    args, **kwargs):
  # Before we fuzz, we need to make sure the prior distribution is set up in the
      game's environment.
  # Calculate and inject it if it's not already there.
  if 'simulation prior distribution' not in env:
      # Grab players and games from an earlier stack frame
      # (avoids weaving an aspect to collect the info)
      frame_to_find_name = "generate_charpair_distributions"
      for frameinfo in inspect.stack():
          if frameinfo.function == frame_to_find_name:
              break
      games = frameinfo.frame.f_locals['games']
      players = frameinfo.frame.f_locals['players']

      from experiment_base import
          find_distribution_of_charpairs_from_players_collective_games
      env['simulation prior distribution'] =
          find_distribution_of_charpairs_from_players_collective_games(players,
          games)
```

Figure 6.6: The prelude aspect calculating the distribution of character pairs selected by the real-world player being simulated, which is reproduced by simulated players using the prior distribution model.

This allows the calculation of the distribution to be separated from the change to players' character pair selection. Separating these into two different aspects allows them to use different join points: this aspect is a prelude and operates before character pairs are chosen, whereas the change made to character selection is performed using a "within"-style fuzzer from PDSF3. The `if` statement at the beginning of the aspect, as shown in Fig. 6.6, avoids calculating the distribution more than once.

```python
def find_distribution_of_charpairs_from_players_collective_games(players, gameset
    ):
    distribution = dict()
    for player in players:
        player_distribution =
            find_distribution_of_charpairs_for_user_from_gameset(player, gameset)
        for charpair, count in player_distribution.items():
            distribution[charpair] = distribution.get(charpair, 0) + count

    return distribution

def find_distribution_of_charpairs_for_user_from_gameset(player, gameset):
    charpair_distribution = dict()
    for game in gameset:
        # Game is from RPGLite's real-world mongodb if it has a `_id` field.
        if '_id' in game:
            game = convert_gamedoc_to_tom_compatible(game)

        if player in game:
            char1 = game[player]['chars'][0][0]
            char2 = game[player]['chars'][1][0]
            pair = char1 + char2 if char_ordering[char1] < char_ordering[char2]
                else char2 + char1
            charpair_distribution[pair] = charpair_distribution.get(pair, 0) + 1

    return charpair_distribution

def convert_gamedoc_to_tom_compatible(gamedoc):
    '''
    We use this to convert a game document from MongoDB to the kind generated by
        the simulation, for ease of comparison.
    '''
    new_gamedoc = deepcopy(gamedoc)

    new_gamedoc['players'] = gamedoc['usernames']
    new_gamedoc[gamedoc['usernames'][0]] = dict()
    new_gamedoc[gamedoc['usernames'][1]] = dict()
    new_gamedoc[gamedoc['usernames'][0]]['chars'] = [gamedoc['p1c1'], gamedoc['
        p1c2']]
    new_gamedoc[gamedoc['usernames'][1]]['chars'] = [gamedoc['p2c1'], gamedoc['
        p2c2']]

    return new_gamedoc
```

Figure 6.7: The implementation of a helper function which calculates the distribution of character pairs from the games a player completed.

Record Character Pair Choices

It is important to keep track of the characters chosen by simulated players, as this data is required to analyse the outcome of the experiments outlined in Section 6.2.2. However, the naive model makes no special accommodation for the collection of character choices made by players. Character choices could be calculated by iterating through the list of completed games after a simulation is complete, filtering for games relating to a specific player. However, the collection of character selection data can be simplified by tracking what was played at the end of a game. This is achieved with with an aspect which observes the character pairs played in a completed game, and appends each players' chosen pairs to their own list. This aspect is applied after its join point executes (an "encore" in PyDySoFu parlance, as discussed in Section 3.3.2).

There are many ways to collect this data, aspectually or otherwise; however, the simplicity of collecting information mid-process without requiring modification of a base model demonstrates the flexibility of an aspectual augmentation of models and simulations as an approach. Collection of chosen character pairs mid-simulation is an opportune example of this convenience.

Track Detailed Outcomes of Games

Similarly to the recording of character pair choices, the outcomes of games must be tracked. This information is important as an input the learning model, as successes and failures observed by players for each character pair constitute what they learn as "good" and "bad" character pair selections.

As with character pair choices, the naive model was not engineered with the intention of providing this information specifically. However, the model is easily instrumented to collect such information at many suitable points. As with recording a player's chosen character pairs,

an "encore" aspect was implemented which records wins and losses observed for character pairs on game end. Specifically, the aspect models players observing character pairs which won at the end of a game, lost at the end of a game, and also pairs which were uninvolved in the game. This requires a list of outcomes (wins, losses, and neither) for every player, for every character pair. The lists of observed outcomes for a given character pair record `True` for a winning pair, `False` for a losing pair, and `None` for a character which was not involved in the game's outcome. Each player observes the relevant state for every character at the end of any game they play. Provisions were made in the implementation of this aspect for a player only paying attention to their own outcomes (i.e. only recording whether their own pair won or lost, without consideration for their opponent's outcome), though in practice all simulations were performed with players observing both their own outcomes and those of their opponents.

Record Winning Pair observed by Players on Game End

The simple model of learning makes use of a distribution of winning teams to simulate learning (the implementation of which is explained in Section 6.5.3). Rather than coalescing the more complex dataset collected as described in Section 6.5.2 to achieve the desired format, another aspect can be applied to collect the data in a simpler format directly.

An aspect was implemented which models players observing winning character pairs on game end and collects the data in a simple list of character pairs which won games, rather than a mapping of character pairs to game end states as described in Section 6.5.2. Character pairs which lose games or are not involved are not recorded for the purpose of the alternative model of learning.

6.5.3 Aspects Implementing Models of Learning

The aspects described at this point mostly correct move selection, handle exceptions which can arise from this, and instrument the model for experimental observation. What

```
def fuzz_learning_by_prior_distribution(steps, target, actor, env, *args, **
    kwargs):
    get_choices_from_prior_dist_injected_code = '''
players = _context['players']
games = _env['games']

distribution = _env['simulation prior distribution']
possible_choices = list()
for pair, count in distribution.items():
    for _ in range(count):
        possible_choices.append(pair)

char_class_map = {
    "K": Knight,
    "A": Archer,
    "R": Rogue,
    "W": Wizard,
    "H": Healer,
    "B": Barbarian,
    "M": Monk,
    "G": Gunner
}
chars_chosen = [char_class_map[char] for char in choice(possible_choices)]
'''
    to_inject = ast.parse(get_choices_from_prior_dist_injected_code)
    return [steps[0]] + to_inject.body + [steps[2]]
```

Figure 6.8: The implementation of the prior distribution model's change to player behaviour, implemented as a PDSF3 fuzzer which inserts character pair selection logic into the `choose_chars` function of the naive model to select character pairs using the distribution taken from RPGLite gameplay data in an earlier aspect.

remains is to describe the implementation of aspects which encode behavioural changes: the prior distribution model, which selects character pairs by drawing from a distribution of character pair choices known before the simulation runs, and the learning model, which selects character pairs using a character's observations of which pairs typically win or lose games.

Character Selection using Prior Distribution

The prior distribution model alters character selection by changing the distribution of character pairs selected from when simulating RPGLite play. The distribution they choose from is changed to reflect the distribution selected by the player being simulated. Another aspect, explained earlier in Section 6.5.2, calculates that distribution and stores it in the simulation's environment. The role of this aspect is to use that distribution when selecting character pairs at the beginning of a game.

```
def choose_chars(_actor, _context, _env):
    if _actor not in _context: # First "step", or steps[0]
        _context[_actor] = dict()

    chars_chosen = sample(characters, 2) # Second "step", or steps[1]
    _context[_actor]['chars'] = [c() for c in chars_chosen] # Third "step", or
        steps[2]
```

Figure 6.9: The naive model's `choose_chars` function, which selects character pairs for a player to play a game with randomly.

The aspect implementing the prior distribution's behaviour change is shown in Fig. 6.8. It is implemented using PDSF3's fuzzers (or "within"-style aspects) to insert additional logic within character pair selection rather than prepending or appending logic to it. Logic is added by modifying the target program's abstract syntax tree (*"AST"*), a native representation of Python code. The steps of the fuzzer's target — provided by PDSF3 as the fuzzer's first argument — are nodes of the AST representing the function definition of the target. The target in this case is `choose_chars`. An implementation of this function is given in Fig. 6.9, edited to show how the lines in the function definition map to values in the `steps` argument of the fuzzer.

Python code to select character pairs using the desired distribution is written in a multi-line string and parsed into an AST. This code:

- Takes the distribution of character pairs from the game's environment, represented as a map of two-letter strings (for example, "KA", representing the pair `Knight`, `Archer`) to the number of times a player selected the pair represented by that string,
- Constructs a list, `possible_choices`, which contains the two-character string identifying a character pair as many times as the character pair was chosen by a player,
- Creates a map of letters to classes implementing the characters those letters represent (for example, "K" maps to the `Knight` class),
- Sets the `chars_chosen` variable to a list of two classes, by selecting a character pair from the `possible_choices` list at random and adding the class implementing each

character in that pair to the `chars_chosen` list.

This leaves `chars_chosen` in the same state as it would have been without the application of the prior distribution model: a list of two character-implementing classes. However, those classes were selected using the distribution produced from RPGLite gameplay data. The AST nodes produced by parsing this code are used to replace the second step in `choose_chars`. The rest of the function logic remains the same. The changed AST nodes are returned by the fuzzer and compiled by PDSF3, ensuring that the modified function definition is executed and so changing player behaviour.

Character Selection using confidence sigmoid

The first model of learning applied through aspect orientation draws on character pairs a simulated player observed to have won games (as recorded by the relevant instrumenting aspect, discussed in Section 6.5.2). The record of previously winning character pairs defines a probability mass function by selecting a character with equal probability to their rate of appearance in the history of winning characters.

Selecting a character based on this distribution is gated by a confidence model as described in Section 6.4.4. If a simulated player's confidence model indicates insufficient experience to found decisions on, players will instead select characters randomly, effectively exploring their space of possible choices. By doing so, players have the time and opportunity to observe many matchups between different character pairs. Time to observe matchups is important because some character pairs may only present a strong choice if played against specific alternatives. We can imagine a character pair which is extremely effective against 50% of pairs, but extremely likely to lose games played against the remaining 50% of possible opponents. Without exploring possible matchups, a player may lack observations which would inform them about whether a character pair is effective in general, or in a narrow set of circumstances. A confidence model encouraging early exploration promotes the experience of a wide variety

of matchups, avoiding this issue.

Another concern early exploration mitigates is that simulated players need opportunities to build well-rounded priors: if a character pair is never selected by any player, experienced players basing their character pair choices on those which previously win games cannot include pairs which were never selected, because they will never have had the opportunity to win games at all.

This aspect was implemented as an "around" aspect in PyDySoFu's parlance, allowing it to apply additional logic before and after its join point. It could equally appropriately have been implemented as an "encore", applying logic only after its join point had executed; the aspect discards the random selection made by the base model unless the simulated player lacks confidence. Either implementation can return a different character pair choice to the join point's caller, allowing the chosen character pair to effectively be overridden, and so allowing the aspect-applied RPGLite model to proceed executing as it would otherwise; the only difference being the pair of characters "chosen" by the simulated player whose behaviour was augmented.

6.6 Model Implementation Details

The aspects described in Section 6.5 are used to investigate whether aspect-oriented programming can be used to augment models, add new behaviours and parameters to them, and successfully do so across different models. The results of these investigations are presented in Chapter 7; some details about the implementation of those experiments are given first in this section.

The parameters used to control experiments are introduced in Section 6.6.1. The representation of experiments' results is given in Section 6.6.2. The statistical methods used to evaluate those results are explained in Section 6.6.3. Finally, the methods used to control sim-

ulated players' interactions with their environments to produce reliable datasets are explained in Section 6.6.4.

6.6.1 Model Parameters

Many parameters are used to control simulated play of RPGLite. They affect the impact of the learning model and the scaffolding which runs simulations, for example by determining the number of simulated players. Parameters which control an experimental run are collected in the `ModelParameters` class, shown in Fig. 6.10. Defining these values in a class helps to identify them in the codebase, and defines all variables which are used when optimising the model.

As there are a significant number of parameters and many have a straightforward impact on simulated play, a complete description is given in Appendix B in the interest of readability. A description is given here for parameters which have a non-obvious effect, or are the focus of the experimental methodology in Section 7.1, or are central to the results given in Section 7.3 and Section 7.4. Variables used in these ways are:

starting confidence The initial value of the model's confidence curve.

assumed confidence plateau A corresponding high value for the model's confidence curve. As the curve's growth rate is calculated to align with the number of games completed by the real-world player being simulated, and the curve asymptotically approaches 1, a high value is required which represents the expected confidence of a real-world player having played a significant number of games.

curve inflection relative to numgames The proportion of games played by the real-world player being simulated at which the player's confidence curve should reach the parameter

```python
from dataclasses import dataclass

@dataclass
class ModelParameters:
    c: float
    curve_inflection_relative_to_numgames: float
    prob_bored: float
    boredom_enabled: bool
    training_data: list
    testing_data: list
    assumed_confidence_plateau: float
    starting_confidence: float
    iteration_base: int
    number_simulated_players: int
    advice: list[tuple[str, str, str|callable]]
    players:list[str]
    args:list[any]
    kwargs:dict[str:any]
    boredom_period: int = 25
    initial_exploration: int = 28

    @property
    def boredom_period(self) -> int:
        '''
        Number of games to play before checking player boredom.
        Attempts to allow every player combo to play each other once on average
            before checking again.
        '''
        return int(self.number_simulated_players**2)/2

    def active_dataset(self, testing) -> list:
        return self.testing_data if testing else self.training_data

    def iterations(self, testing) -> int:
        '''
        The number of games to play when generating a synthetic dataset
        '''
        if self.boredom_enabled:
            return self.iteration_base
        return int(self.number_simulated_players**2 * len(self.active_dataset(
            testing)) / 2)

    def rgr(self, testing) -> float:
        '''
        RGR for the parameterised c value, number of games to play, and start/end
             confidences.
        '''
        if not hasattr(self, '_rgr_cache'):
            self._rgr_cache = dict()
        if self._rgr_cache.get(testing) is None:
            num_games_to_confidence = len(self.active_dataset(testing)) * self.
                curve_inflection_relative_to_numgames
            self._rgr_cache[testing] = \
                curveutils.rgr_yielding_num_games_for_c(
                    num_games_to_confidence,
                    self.c,
                    start=self.starting_confidence,
                    limit=self.assumed_confidence_plateau)
        return self._rgr_cache[testing]
```

Figure 6.10: A class defining the parameters of the model of learning RPGLite which vary when optimising for a given player. For layout & space reasons, some methods of minor significance are removed.

`assumed_confidence_plateau`. This allows for control over the growth rate of the curve while linking the rate of growth to the number of games played by the real-world player being simulated;

C The curve parameter of the confidence model's underlying Birch curve;

prob bored The probability that a player with confidence above `assumed_confidence_p` becomes "bored" and stops playing RPGLite. These players are removed from the simulated playerbase and replaced with new players. This mechanism is discussed in further detail in Section 6.6.4.

advice A list of tuples defining advice to weave when generating data. Pieces of advice are uniquely defined by a tuple containing the type of aspect to weave (such as `"prelude"` or `"error_handler"`), a string-represented regular expression defining the join point of the advice, and a `callable` to use as an aspect when weaving the advice. As `ModelParameters` instances are serialised to disk to persist experiment results and Python functions are not seralisable, this may also be a string value; strings are IDs of aspects, and are replaced with their corresponding aspect on deserialisation.

training data The set of real-world games used as a training fold when optimising parameters for simulating a player. This is used when annealing toward optimal parameters for the learning model, and so is explained in more detail in Section 7.1.

testing data The set of real-world games used as a testing fold when optimising parameters for simulating a player. This is used when annealing toward optimal parameters for the learning model, and so is explained in more detail in Section 7.1.

As these determine all variables affecting data generation, individual experimental runs[10] can therefore be reproduced using the class. The convenience method `run_experiment()`, shown in Fig. 6.11, is provided to simplify reproducing experimental runs with `ModelParameters` objects. These parameters are varied in some experimental runs to optimise the model and represent individual players' learning to find a combination of parameters which best represent a given player. The strategy for this optimisation is described in Section 7.1.

6.6.2 Representing the Result of an Experimental Run

As referenced in Fig. 6.11, a `Result` class is defined to collect the results of an experimental run and simplify analysis of the distribution of character pair selection produced. The implementation of the `Result` class is given in Fig. 6.12. This class is used in the analysis of the success of an experimental run when selecting ideal experimental parameters for the simulation of a real-world player.

A `Result` is not the result of an entire experiment, or the ideal parameters to simulate a given player. Instead, it represents the result of attempting to generate realistic data for a player, which is defined by a `ModelParameters` attribute. Several `Result` objects can be compared against correlation statistics and probabilities of circumstantial correlation (the `statistic` and `pval` properties respectively) using the `within_acceptable_bounds` method to select an optimal set of parameters to simulate an RPGLite player. A technique which uses these properties to determine optimal parameters is described in Section 7.1.

6.6.3 Quantifying Similarity of Character Pair Selection

To measure the results of experiments in Chapter 7, many datasets are generated (with differing model parameters in the case of the learning model) and are compared against a

[10] Throughout this chapter, "experimental run" is used to refer to the process of producing a simulated dataset for comparison against training or testing folds taken from the dataset of completed RPGLite games. Many experimental runs with different parameters are used to optimise a model.

```
class ModelParameters:

    # <snipped additional attributes and methods for space>

    def run_experiment(self, testing, correlation_metric):
        real, synthetic = generate_charpair_distributions(\
            rgr_control=self.rgr(testing=False),
            iterations=self.iterations(testing=False),
            games=self.training_data,
            players=self.players,
            birch_c=self.c,
            sigmoid_initial_confidence=self.starting_confidence,
            boredom_confidence=self.assumed_confidence_plateau,
            num_synthetic_players=self.number_simulated_players,
            boredom_period=self.boredom_period,
            prob_bored=self.prob_bored,
            advice=self.advice,
            *self.args,
            **self.kwargs)
        return Result(self, real, synthetic, correlation_metric, testing)
```

Figure 6.11: The `run_experiment()` method on the `ModelParameters` class, which can be used to reproduce an experimental run from its parameters.

```
from dataclasses import dataclass
@dataclass
class Result:
    params:ModelParameters
    real_distribution:list[float]
    simulated_distribution:list[float]
    correlation_metric:Optional[callable]
    testing: bool

    @property
    def pval(self) -> float:
        return self.correlation_metric(self.real_distribution, self.
            simulated_distribution).pvalue

    @property
    def statistic(self) -> float:
        return self.correlation_metric(self.real_distribution, self.
            simulated_distribution).statistic

    def within_acceptable_bounds(self, pval_threshold:float, statistic_threshold:
        float) -> bool:
        return self.pval < pval_threshold and self.statistic >
            statistic_threshold
```

Figure 6.12: The `Result` class used to collect results of experimental runs and evaluate their success.

player's completed games to determine which generated dataset is most similar to the empirical one. The distribution of the naive model should exhibit no bias as character pairs are chosen randomly; simulated play with advice woven is expected to exhibit more "realistic" character pair preferences than the naive model as a result of the advice applied. The similarity between a player's character pair preferences and those of simulated players are determined using a rank correlation measurement.

Many rank correlation measurements exist. A non-paremetric measurement is required for measuring the correlation of character pair distributions. Non-parameteric correlation measurements make no assumptions about the distribution of data; the alternative is to use a parametric measurement, which would be selected based on the distribution in the data being compared. The distribution of character pair preferences is unknown, so to avoid assuming a distribution which is universally applicable to all players' interaction with RPGLite a non-parametric correlation measurement is chosen.

Two common non-parametric rank-correlation measurements are Spearman's ρ [130] and Kendall's τ [78]. The differences between these methods can affect measurements of character pair preference. Spearman's ρ accounts for the magnitude of difference between ranks, and is calculated as $\rho = 1 - \frac{6\sum(d_i^2)}{n(n^2-1)}$, where d_i is the difference in the rank of the i^{th} observation in each dataset and n is the total number of datapoints in a dataset. Kendall's τ accounts only for their ordering. It can be calculated as $\tau = \frac{C-D}{C+D}$, where C is the count of pairs with the same rank (*"concordant"*), and D the count of pairs with differing ranks (*"discordant"*).

To choose an appropriate measure for comparing character pair distributions, they can be related to potential patterns in those distributions. As a player is likely not to select many character pairs once they have developed preferences (if at all), a slight increase in selecting uncommon character pairs within a simulation could incur a significant increase in the rank of those pairs. This would have a more significant impact on ρ than on τ, as the larger difference in rank is squared in the calculation. Character pairs' rankings may also have small differences

between the two distributions but similar positions relative to each other. For example, most character pairs could have ranks only 1 position apart between the distributions. This would have an outsized impact on τ, as relative positioning affects concordance, but the square of a small difference in rank remains relatively small, meaning that ρ would be less affected by this pattern. A choice of correlation metric is thus a choice of what correlation means: it can reflect preference of character pairs relative to each other, or "forgive" small relative preference deviations and impose a greater cost on the scale of disagreement on a given rank instead.

To prioritise the preservation of the order of a ranking character pair preferences are compared using τ. Unfavoured character pairs' ranks might be affected greatly by a small difference in the number of games played, because unfavoured character pairs play few games. A small difference in selection count is therefore more likely affect the rank of those pairs. Minimising rank difference is sensitive to RPGLite's random nature for this reason. Preserving the ordering of character pair selection is also important when considering favoured pairs, as most games are likely to be played with a small number of favoured pairs. Much as large differences in unfavoured pairs' ranks can be caused by random chance, small differences in the ranks of favoured pairs are indicative of a player's preferences. An appropriate correlation measurement must take this into account. Small differences in the ranks of favoured pairs would have a less signifiant impact on ρ than on τ: ρ accounts for the square of the difference in rank, which remains small for small differences. Experimental results are calculated using τ in Chapter 7 for this reason: τ prioritises accurately simulating a player's preferences when selecting character pairs, which is the aim of the experiments, and so is the more appropriate measurement.

6.6.4 Controlling State Space Exploration

The model of confidence grows monotonically as described in Section 6.4.3, which affects simulated players' exploration of possible character pairs. The learning model uses confidence as a probability that character pairs are selected based on experience. As character pairs which are not played cannot win games, pairs which are not selected for play cannot be chosen by the learning model as it relies on a player's historical observations, which could lead to inaccurate simulation if the unchosen pair is the player's preference. To counter this possibility, two mechanisms are introduced to control state space exploration: "initial exploration" and "boredom".

Initial exploration is a threshold number of games a simulated player must complete before the confidence model determines whether aspects should change player behaviour. The earning model's impact on player behaviour is gated on both a player's level of confidence and the number of games played. This allows players to enough games that later decisions directed by the learning model can make "informed" decisions, having explored the possible options. The length of the initial exploration phase for any player is given as an integer number of games by the `initial_exploration` parameter when generating data.

Boredom is a mechanism which solves a problem introduced by some combinations of model parameters: if the confidence model's growth rate and curve parameter are high, player confidence raises sharply, with the effect that simulated players are unlikely to explore the state space. A situation can arise where the simulation's metagame falls into a local optimum where the entire playerbase collectively observe many historical wins for some character pairs because of a disproportionately high rate of random selection when players explore the space of possible character pairs. This can be overcome by introducing new players into the playerbase over time. New players are guaranteed to explore the state space due to the initial exploration phase. The randomly selected character pairs played in this time mean that veteran players will play games with character pairs which otherwise could not have been selected. This

reinforces existing preferences when novel character pairs are weaker than veteran players' preferences, and weakens existing preferences if veteran players are exposed to novel character pairs which are likely to win games played against their preferred pairs. By disrupting the simulation's metagame boredom mitigates potential issues with random play while allowing simulated players to learn over time.

Players are eligible to be removed from the playerbase when their model of confidence reaches a threshold defined by the `assumed_confidence_plateau` parameter. Each player is removed with probability `prob_bored` once this threshold is reached, allowing some confident players to remain in the playerbase for a long time. Any player who is removed is immediately replaced with a new one, so the playerbase has a constant size. The `boredom_period` parameter determines the number of games between these checks, and the mechanism can be disabled entirely using the `boredom_enabled` parameter.

6.7 Contributions

This chapter describes the construction of an aspect-oriented model of RPGLite play. This model's foundation is a "naive" model of random play, to which aspects are applied which:

- Instrument the model to make measurements necessary for the evaluation of experiments;
- Compensate for edge-cases present in RPGLite's design;
- Implement models of confidence and learning;
- Use those models of confidence and learning to alter modelled players' behaviour.

This chapter contributes the research software engineering required to design aspect-oriented models of confidence and learning and implement those models using PDSF3. Having

contributed these aspect-oriented models, this chapter supports the contributions of **??**. The latter chapter presents the results of three experiments which address the research questions posed in Section 2.4.1:

> **RQ1** *Can models of systems more accurately reflect their subjects by weaving aspects which represent improvements?*
>
> **RQ2** *Can advice be used to faithfully introduce behaviours or parameters into a model which were not originally present in it?*
>
> **RQ3** *Can advice be used as a portable module, such that aspect-oriented improvements to one model can be woven into another without loss of performance?*

This chapter should be understood as the first of two halves of a single research effort which investigates the application of aspect-oriented programming to models for research purposes. These two halves are separated for structure and legibility reasons. Chapter 7 describes the remaining contributions of this effort.

Chapter 7

Results of Experiments concerning Aspect-Oriented Modelling

The naive model of RPGLite and the aspect-oriented models of learning & confidence which are described in Chapter 6 are designed to answer the following research questions:

> **RQ1** *Can models of systems more accurately reflect their subjects by weaving aspects which represent improvements?*
>
> **RQ2** *Can advice be used to faithfully introduce behaviours or parameters into a model which were not originally present in it?*
>
> **RQ3** *Can advice be used as a portable module, such that aspect-oriented improvements to one model can be woven into another without loss of performance?*

To investigate each research question, relevant advice is woven into the naive model, and datasets are generated of recorded simulated gameplay. To answer the proposed research questions, these datasets are compared against the empirically sourced datasets described in Chapter 5. Different experiments require different comparisons and yield different contributions, but all make use of the same technical foundations described earlier in Chapter 6.

This chapter explores the results of the experiments which are enabled by the previous chapter's foundations. Section 7.1 describes how synthetic datasets are generated which to

yield statistically significant correlation when simulating a given player, and in particular how parameters are chosen for the learning model to yield those results. Following this are three sections each focused on an experiment to answer a research question. These sections are composed of an explanation of the experiment's design, followed by a presentation of results and an evaluation of those results with regards the research question of interest. Section 7.2 describes an experiment answering the first research question, concerning the use of advice to alter pre-existing modelled behaviour. Section 7.3 discusses an experiment answering the second research question, concerning the use of advice to introduce new parameters and behaviours to a model. Finally, Section 7.3 presents an experiment answering the third research question, concerning the portability of advice as individual modules to new systems, or to changed instances of the same system. The chapter's contributions are briefly reviewed in its conclusion, Section 7.5.

7.1 Methodology concerning the Interpretation of Results

7.1.1 K-Fold Validation

RPGLite exhibits randomness, and its random nature affects the datasets generated when running experiments; this may cause correlation to appear by chance, biasing results. As discussed in Section 6.6.4, efforts are made to mitigate the impact of random play such as periodically refreshing the player pool and ensuring many games are played. Efforts are also made when interpreting results to control for the influence of random chance.

Simulations are run several times, and the datasets they generate are compared against subsets of RPGLite player data to measure correlation, minimising the impact of randomness when evaluating results. This is achieved using k-fold validation, and "Leave-One-Out Cross-Validation" (LOOCV) in particular. In k-fold validation, the dataset being compared against is divided into k equally sized partitions ("folds"), which can be combined in different ways

to produce datasets for training an algorithm or finding optimal parameters for an algorithm ("training folds") and corresponding datasets for testing the results of each optimisation ("testing folds") [120]. LOOCV creates k pairs of training and testing folds, where testing folds are individual partitions, and their corresponding training folds are the union of all other partitions.

All experiments in this chapter make use of LOOCV to minimise bias in results. RPGLite player data is partitioned, and sets of training and testing folds are produced from these partitions. Experiments generate multiple datasets, comparing the results against different testing folds, and optimising model parameters against different training folds where appropriate. In all experiments $k = 5$ to avoid small training & testing folds and to measure correlation many times, which helps to identify individual biased results.

7.1.2 Identifying Model Parameters yielding Optimally Significant Results

Section 7.2 and Section 7.3 describe experiments where parameters for the model of learning are optimised. As this optimisation occurs across many folds, it is possible that many parameters produce statistically significant datasets when compared against a given player. These parameters may also vary between folds, as comparison against different testing folds may show correlation with different datasets. A technique to identify model parameters which produce statistically significant results across many folds is therefore required.

Identifying optimal parameters requires searching across two dimensions of statistical significance reported in experiment results: Kendall's τ correlation metric produces a statistic of correlation (which is also referred to as τ in the context of results) and a p-value describing the probability that the correlation statistic would arise in the case that the null hypothesis for a given experiment is true, as described in Section 6.6.3. Statistically significant results must demonstrate a sufficiently high correlation statistic and a sufficiently low p-value; "sufficiently" high or low is defined below.

To identify parameters for a player which reliably produce statistically significant results across folds, a threshold is set for both values of τ and their corresponding p-values. Parameters producing results which meet both thresholds across $> 50\%$ of folds are considered to be significant parameters for simulating a player, as they are more likely than not to produce datasets correlating to real-world play. If no such parameters exist, thresholds are weakened, and the search repeats. Thresholds continue to be weakened until doing so would indicate statistically insignificant results, such as large p-value or a low τ value. If no parameters exist which reliably produce statistically significant results for a player, the player is not accurately represented by the model.

Correlation coefficient thresholds are selected at 0.5, 0.4, 0.3, and 0.2. In behavioural science research, a correlation coefficient of 0.5 is considered strong in practice, 0.3 of medium strength, and anything below 0.2 weak; these values are both suggested as guidelines [27] and found in the distribution of published results in metastudies [59]. As the model of learning used in these experiments predicts human behaviour, typical bounds on high and low correlation from the behavioural science community are adopted for these experiments, and a correlation coefficient below 0.2 is discarded in all cases as insignificant. P-value thresholds are 0.01, 0.02, 0.035, and 0.05. A maximum value of 0.05 is adopted by convention in many research communities [56]; other p-value thresholds are selected arbitrarily for incrementally more significant results.

To determine which pairs are searched first, they are ordered by their Manhattan distance from the most restrictive pair, $\tau > 0.5$ and $p < 0.01$. This is calculated by summing the number of steps away from the most restrictive threshold for each measurement: a threshold pair $\tau > 0.4$ and $p < 0.2$, with a Manhattan distance of $1 + 1 = 2$, is considered stronger than a pair $\tau > 0.5$ and $p < 0.05$, which has a Manhattan distance of $0 + 3 = 3$, and therefore is used first when searching for significant results.

Experiments in Section 7.2 and Section 7.3 use this technique to find “optimal” parameters

to model each player with statistical significance, if such parameters exist for a player. The algorithm implementing this search is given in Appendix C for reference.

7.2 RQ1: Altering Model Behaviour using Aspect Orientation

7.2.1 Experimental Design

To demonstrate the viability of altering a model by using aspects to describe a change in behaviour, the advice applying the already-known character pair distribution to character pair selection is applied. This forms the first of three experiments in this chapter, addressing the research question:

> *Can models of systems more accurately reflect their subjects by weaving aspects which represent improvements?*

The research question yields a null hypothesis: *models of systems cannot be altered using advice to reflect their subjects more accurately*. If this were the case, there would be no discernable difference between the real world data's correlation against data from the naive model and data from a model with advice woven. Simulated players in the naive model select character pairs randomly, meaning that — if patterns such as personal preference exist in the real-world data, or the data is not randomly distributed — we would not expect much correlation between the two. If weaving advice produces datasets which do correlate with the real-world data, then the advice would have affected the model to more accurately reflect the player being simulated. We would therefore discount the null hypothesis, and answer the research question affirmatively. If no correlation could be produced as a result of weaving aspects, then the null hypothesis would have been demonstrated instead, answering the research question negatively.

Figure 7.1 illustrates the advice woven into the naive model to alter player behaviour to

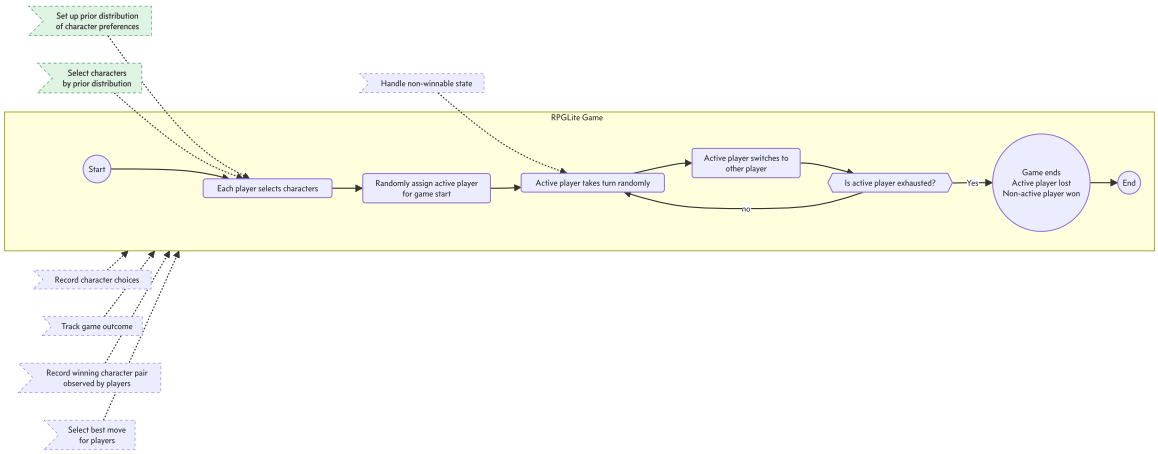

Figure 7.1: A UML activity diagram showing aspects woven into the naive model to adopt the distribution used to select character pairs from real-world data. Aspects are given dashed borders and applied with a dotted line in the same manner as found in AOPML [13]. Aspects uniquely used by this model are coloured green.

select character pairs with the same distribution as the real-world data exhibits. As this advice causes simulated players to select character pairs with the same distribution as real-world players, the datasets generated using them ought to correlate strongly with the real-world data: rather than being random, the distribution is expected to be the same. This therefore tests our hypothesis that advice can be used to alter model behaviour to be more accurate. [1]

7.2.2 Results

Results are generated through k-fold validation as described in Section 7.1.1. As no additional parameters are added to the model, no searching for optimal parameters is required. A drawback of the lack of searching for optimal parameters is that there is no convenient way to summarise results across folds. However, results are *consistent* across folds; the result of the first fold is given in this section's result tables, described below. Complete tables with data from all folds is given in Appendix D in the interest of transparency. Results are shown for players who completed at least 100 games in season 1 of RPGLite, and the experiment is conducted against season 1 gameplay data.

The results of the experiment are shown in Table 7.1 for datasets produced by the naive

[1] Later experiments will investigate the addition of specific behaviours, rather than reproducing properties of an already-present dataset.

model, and Table 7.2 for datasets produced by applying advice to the naive model which alters the distribution from which character pairs are selected.

Username	*p-value*	τ *statistic*
apropos0	0.930	-0.013
basta	0.182	0.205
creilly1	0.352	0.141
creilly2	0.764	-0.046
cwallis	0.042	0.309
Deanerbeck	0.881	-0.023
ECDr	0.370	0.135
elennon	0.683	0.062
Ellen	0.417	-0.120
Etess	0.291	0.165
Fbomb	0.169	-0.220
Frp97	0.452	0.119
georgedo	0.185	0.205
Jamie	0.944	-0.0106
kubajj	0.704	-0.058
l17r	0.646	-0.073
Nari	0.330	-0.152
Paddy	0.294	0.166
sstein	0.529	0.101
tanini	0.213	0.185
timri	0.160	-0.220

Table 7.1: Correlation of real-world character pair selection and those generated by an unmodified naive model.

Username	*p-value*	τ *statistic*
apropos0	6.070×10^{-10}	0.964
basta	6.984×10^{-9}	0.975
creilly1	6.984×10^{-10}	0.961
creilly2	1.154×10^{-8}	0.984
cwallis	2.514×10^{-9}	0.970
Deanerbeck	4.742×10^{-8}	0.979
ECDr	8.455×10^{-10}	0.959
elennon	3.963×10^{-9}	0.973
Ellen	2.538×10^{-9}	0.950
Etess	1.113×10^{-8}	0.994
Fbomb	3.117×10^{-8}	0.996
Frp97	2.440×10^{-8}	1
georgedo	4.719×10^{-8}	0.970
Jamie	5.760×10^{-9}	0.985
kubajj	5.728×10^{-9}	0.966
l17r	1.056×10^{-7}	0.994
Nari	1.965×10^{-8}	0.985
Paddy	1.171×10^{-8}	0.984
sstein	5.017×10^{-8}	0.988
tanini	1.539×10^{-9}	0.952
timri	2.582×10^{-8}	0.990

Table 7.2: Correlation of real-world datasets of character pair selection and those generated by the naive model with advice woven to bias the characters chosen.

Briefly summarised, the only player showing statistically significant results from the data produced by the naive model is from *cwallis*, from whom a p-value of 0.042 and a τ correlation coefficient of 0.309 is produced, indicating medium-strength correlation with a low chance of being a coincidence. As noted in Section 7.1.2, a τ statistic above 0.2 paired with a p-value below 0.05 is the threshold set for statistical significance. With a 4.2% chance of being coincidentally generated and 21 players simulated, this outcome is not unexpected. Correlation results from *cwallis'* other folds show no significant correlation, as seen in the complete table presented in Appendix D. The results of simulating all players with advice woven show strong correlation and low p-values, meeting the criteria for statistical significance as described in Section 7.1.2 in every case.

7.2.3 Answering the First Research Question

The data shown in Table 7.1 demonstrates that — without any advice woven into the model to alter its selection of characters — the naive model selects character pairs which do not correlate with those found in real-world datasets at all. As the naive model acts randomly with regards character pair selection, this aligns with expectations.

The data shown in Table 7.2 contains the correlation of real-world datasets for a player with the naive model, with advice woven to select character pairs from the distribution found in that player's empirical dataset. Every simulated player showed an extremely strong correlation statistic and p-value for their simulated dataset's correlation with their empirical equivalent. While this correlation is extreme, it also matches expectations. Advice which selects character pairs using a known distribution ought to result in that distribution being represented in simulated games.

This demonstrates the hypothesised effect: player behaviour can be altered though the weaving of aspects to produce a model which more closely matches real-world observations in some manner. The research question can therefore be answered affirmatively.

A curiosity of this experiment is that the altered behaviour is not a cross-cutting concern as they are typically defined. Modularisation of these parts of a program is aspect-oriented programming's original use-case [81], but this experiment shows that the technique can be successfully used in other ways (as suggested by Gulyás and Kozsik [55] and Steimann [132]). The success of advice as a mechanism to introduce change to a program, rather than being used as a mechanism to refactor or more elegantly design it, suggests that aspect-oriented programming could be well-suited to exaptation [53] as a tool to introduce new behaviours in a model.

7.3 RQ2: Adding Model Behaviours & Parameters using Aspect Orientation

7.3.1 Experimental Design

The behaviour added to the naive model in Section 7.2 impacts how accurately the model reflects a player, but does not model new player *behaviours*. While this is sufficient to demonstrate that aspect-oriented programming can be successfully used to improve a model, it does not answer the second research question:

> *Can advice be used to faithfully introduce behaviours or parameters into a model which were not originally present in it?*

The experiment discussed in Section 7.2 changes the distribution used to select characters, but does not parameterise its changes and adds no new activity on the part of simulated players which would require a new parameter. It therefore demonstrates that the software engineering technique can be applied in the context of simulation & modelling, but does not demonstrate that aspect-oriented programming meets the needs of researchers who seek to apply advice which add or make changes to the behaviours of actors in their simulations rather than tweaking existing ones. As a result, answering this research question requires a further experiment, which the aspect-oriented models of confidence and learning introduced in Chapter 6 yield.

Aspect-oriented models of confidence and learning are therefore woven into the naive model to produce another in which players learn over time. A player's choice of characters is not selected randomly from a distribution in this model, but informed by their observations of wins and losses in previous games. The advice used is that of the models of confidence and learning described in Chapter 6. A diagram of the woven advice is given in Fig. 7.2.

The null hypothesis associated with the research question this experiment addresses is, *advice cannot be used to accurately introduce behaviours or parameters into a model in which they*

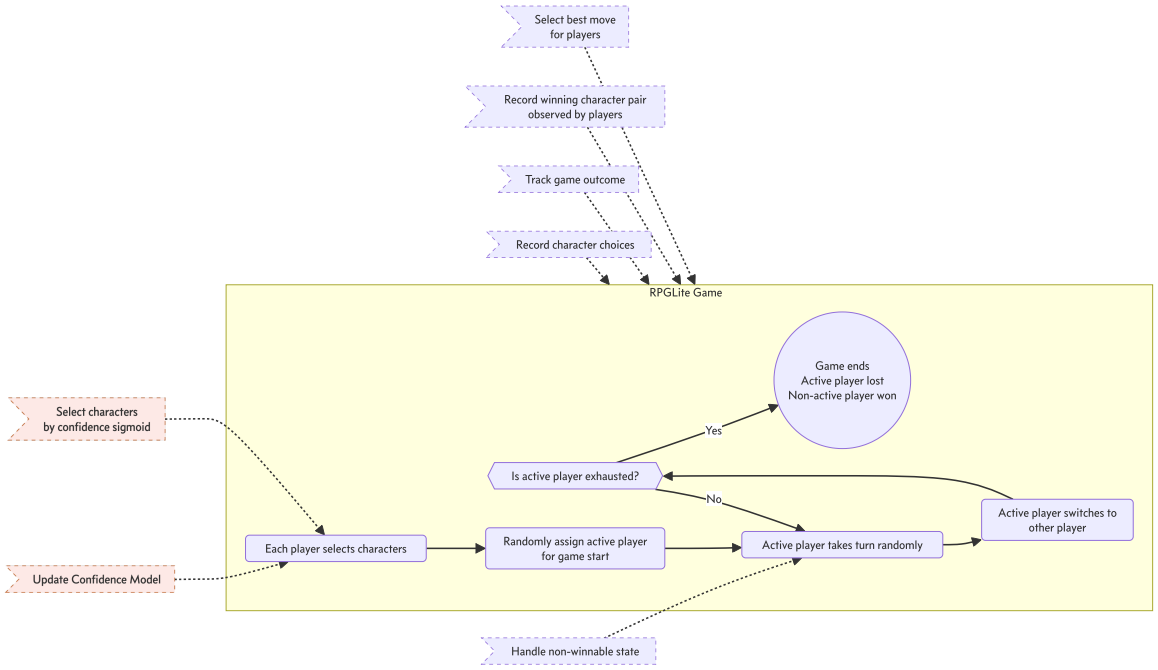

Figure 7.2: A UML activity diagram showing aspects woven into the naive model to amend simulated players' behaviour to include learning over time, and to track the relevant parameters for the related model of confidence. Aspects are given dashed borders and applied with a dotted line in the same manner as found in AOPML [13]. Aspects uniquely used by this model coloured red.

are not already present. If this were true, the datasets produced by the aspect-oriented model shown in Fig. 7.2 would exhibit no more correlation with real-world data than the naive model does. If a learning behaviour can be successfully introduced, some players' simulations would exhibit consistently realistic behaviour which would correlate with their real-world data.

Unlike the experiment described in Section 7.2, this experiment uses no real-world data to inform the actions simulated players make. As a result, if parameters can be found for a player which consistently generates datasets correlating with their empirical dataset, the simulated player's behaviour must have been successfully altered to learn in an accurate manner. This would discount the null hypothesis; the research question could therefore be answered affirmatively.

7.3.2 Results

Results were generated by running the naive model with advice woven which implements a model of learning. Results were generated using k-fold validation and model parameters searched for using the technique described in Section 7.1.2. The parameters modified when running simulations were:

- The confidence model's Birch curve shape parameter, which could have any of the values $c \in \{\frac{1}{50}, \frac{1}{10}, \frac{1}{5}, 1, 5, 10, 50\}$.
- The coefficient applied to scale the relative growth rate of the confidence model's Birch curve, which could have any of the values *rgr coeff.* $\in \{0.1, 0.3, 1, 3\}$
- The probability that a simulated player would become "bored" and replaced in the player pool for the simulation, as described in Section 6.6.4. The probability that a player would become bored could have any of the values *prob. bored* $\in \{disabled, \frac{1}{64}, \frac{1}{16}, \frac{1}{4}, 1\}$

These parameters were selected arbitrarily. The practice of identifying appropriate parameters is termed "parameter recovery" in the cognitive modelling community, and is its own field of research [58]. Unfortunately, remaining time on this project did not support a rigorous parameter recovery study for these parameters; to approximate the process of a parameter recovery study, Birch curves were drawn using different parameters to determine how extreme the effect of different parameter values was. Appropriate values were selected by visually inspecting those diagrams. A future study could include a parameter recovery study to select sets of parameters which more closely align with the behaviours of real-world players. An alternative approach which future studies could adopt would be the use of a search algorithm which treats the space of possible parameters as continuous instead of discrete, such as simulated annealing [82].

Table 7.3 contains the results of the search for parameter sets which would accurately simulate each player. For each player, model parameters which produced accurate simulations

Username	*p-value* <	τ >	*Confidence curve value*	*Confidence RGR modifier*	*Prob. bored*	*# folds*
apropos0	0.01	0.4	0.2	3	0.062	3
apropos0	0.01	0.4	10	0.1	*disabled*	3
cwallis	0.01	0.4	0.02	0.1	0.25	4
cwallis	0.01	0.4	0.1	0.3	0.25	4
cwallis	0.01	0.4	1	0.1	0.25	4
cwallis	0.01	0.4	10	0.1	0.25	4
elennon	0.035	0.3	50	0.1	0.062	3
Ellen	0.01	0.3	10	0.3	0.016	3
georgedo	0.02	0.3	50	3	0.016	3
georgedo	0.02	0.3	0.1	3	0.016	3
georgedo	0.02	0.3	0.2	3	0.062	3
georgedo	0.02	0.3	0.2	1	0.062	3
georgedo	0.02	0.3	0.02	1	0.062	3
Jamie	0.01	0.4	0.2	1	1	3
Jamie	0.01	0.4	1	3	0.25	3
Jamie	0.01	0.4	5	0.1	0.25	3
Jamie	0.01	0.4	0.02	1	0.016	3
basta				*N/A*		
creilly1				*N/A*		
creilly2				*N/A*		
Deanerbeck				*N/A*		
ECDr				*N/A*		
Etess				*N/A*		
Fbomb				*N/A*		
Frp97				*N/A*		
l17r				*N/A*		
kubajj				*N/A*		
Nari				*N/A*		
Paddy				*N/A*		
sstein				*N/A*		
tanini				*N/A*		
timri				*N/A*		

Table 7.3: Correlation of empirically sourced datasets and those generated by applying the aspect-oriented model of learning.

are shown. Results for players for whom parameters could not be found which produced correlation at any threshold of statistical significance have their results shown as *N/A*. For some players, many parameters could be found which produced correlating data at some threshold of statistical significance. In these cases, Table 7.3 contains all such parameters. Results are shown for players who completed at least 100 games in season 1 of RPGLite, and the experiment is conducted against season 1 gameplay data.

Simulations of player *cwallis* yielded 26 parameter combinations producing statistically

significant results, far more than any other player. For legibility, only parameters which produced statistically significant correlation in 4 folds are shown in Table 7.3. A complete table with all results is provided in Appendix E.

7.3.3 Answering the Second Research Question

Table 7.3 shows that six players are accurately modelled by the aspect-oriented model of learning shown in Fig. 7.2. The answer to the research question is therefore affirmative: advice can be used to accurately introduce new behaviours and parameters to a model which were not previously present.

Not all players are accurately simulated using this model of learning. This result is to be expected. One reason for this is that different players learn in different ways; the model of learning developed in this thesis is not intended to represent some universal model of learning (if such a model can exist) but to represent some players' behaviours, thereby demonstrating that complex behavioural models can be tractably modelled as advice. Another reason is that, as these models draw from historically observed data, learning may be sensitive to a player's personal biases: for example, they may not understand how to usefully employ a certain character, skewing their observations of wins and losses. Finally, real-world players may simply dislike certain characters for superficial reasons, such as artwork or a lack of interest, which would skew their distribution of character pair selections in a manner which the model of learning would be unable to account for as written.

The accurate simulation of $\approx 28.9\%$ of the sampled playerbase demonstrates two successes of this experiment. First, the fact that model did not accurately simulate *all* players is a positive result: if it did, this would indicate that the experiment was incorrectly conducted in some manner, as biases unrelated to learning are to be expected in players' interactions with RPGLite, making a 100% success rate indicative of an error in the simulation. Second, the research question is answered affirmatively: advice *can* be employed to represent new

behaviours within – and add new parameters to – an existing model.

7.4 RQ3: Applying Aspects to New Models

7.4.1 Experimental Design

Advice representing change to a model may only be applied to a single join-point, as is the case for the aspects described in Section 6.5, but that advice could be applied to future models. Seen though this lens, the behaviour encoded in a model may not be cross-cutting *within* a model, but cuts *across* models, with each potentially including its own implementation of the behaviour tangled within implementations of other behaviours. Its representation as advice could allow the behaviour to be implemented once, but woven into every model that can make use of it. This is the concept underlying prior research [150] which modelled software engineering teams working under different software methodologies, and used aspect-oriented programming to implement behavioural changes such as distractedness as cross-cutting concerns. However, this research did not confirm its findings against real-world data, and so could not verify that models of behaviours as cross-cutting concerns were accurate.

Chapter 2 proposed the following research question to verify that aspects can cross-cut models rather than modules of a program:

> *Can advice be used as a portable module, such that aspect-oriented improvements to one model can be woven into another without loss of performance?*

As RPGLite was played in multiple seasons, variants of the underlying system were used to collect player data. Changes in game seasons constitute changes to the configuration of the game by strengthening some characters and weakening others. As this alters RPGLite's metagame [71], players are expected to react by re-evaluating their preferred character pairs. This has the effect of creating a second model which differs minimally from the first, which

can be used to investigate the accuracy of an aspect cross-cutting two models.

A final experiment is described in this section, in which the model drawing from a known distribution of character pair choices and the model of learning are each applied to RPGLite's second season. The accuracy of these models was examined in Section 7.2 and Section 7.3 respectively. Should they remain accurate when applied to the second season, we can affirmatively answer the final research question. The research question also yields a null hypothesis: *aspects which are accurate when applied to one model cannot be ported to another*, which can be discounted if and only if the experiment demonstrates aspects being accurately woven into a model of the second season as well as the first.

When an aspect is reused, it is unclear whether values for its parameters which produced accurate data in one model should also be reused. As the learning model introduces more parameters to the model, two experiments can be run using it to investigate parameter reuse. First, the learning model is run using parameters which produced statistically significant data for each player in season 1. Second, the learning model is run using the same methodology as used in Section 7.3 to anneal toward optimal parameters using season 2 player data. The methodology for the latter is the same as used for season 1 in Section 7.3. In the case of the former, parameters which produced statistically significant data in season 1 are re-run and the datasets produced are compared against relevant player's gameplay data from season 2.

7.4.2 Results

The results of running the model with a change to the distribution used for character pair selection are shown in Table 7.4. Results are presented in the same manner as with the analogous experiment in Section 7.2: results from the first fold are presented in the table for each player, with a complete table available in Appendix F.

The results of applying the learning model to season 2, parameterised using a player's

Username	*p-value*	τ *statistic*
aaaa	7.629×10^{-9}	0.962
apropos0	4.944×10^{-9}	0.969
basta	4.244×10^{-9}	0.934
Becccca	2.711×10^{-9}	0.959
creilly1	1.232×10^{-7}	1.0
DavetheRave	5.239×10^{-9}	0.949
Deanerbeck	1.032×10^{-9}	0.974
DX13	8.022×10^{-11}	0.958
ECDr	2.035×10^{-7}	1.0
Ellen	6.935×10^{-10}	0.963
Ezzey	5.702×10^{-8}	0.995
Frp97	1.232×10^{-7}	1.0
Jhannah	3.117×10^{-8}	0.996
l17r	4.265×10^{-8}	1.0
Luca1802	3.930×10^{-9}	0.982
Martin	7.137×10^{-9}	0.943
Nari	6.937×10^{-9}	0.992
sstein	7.316×10^{-8}	0.999
timri	7.316×10^{-8}	0.999

Table 7.4: Correlation for simulation using simple character pair distribution model on RPGLite Season 2

Username	*Season 1 p-value <*	*Season 1* $\tau >$	*Confidence curve value*	*Confidence RGR modifier*	*Prob. bored*	*Season 2 p-value*	*Season 2* τ *value*
apropos0	0.01	0.4	0.2	3	0.062	0.597	0.075
apropos0	0.01	0.4	10	0.1	*disabled*	0.440	0.109
elennon	0.035	0.3	50	0.1	0.062	1.0	0.0
Ellen	0.01	0.3	10	0.3	0.016	0.008	0.369
georgedo	0.02	0.3	50	3	0.016	0.419	0.118
georgedo	0.02	0.3	0.1	3	0.016	0.470	0.106
georgedo	0.02	0.3	0.2	3	0.062	0.085	0.253
georgedo	0.02	0.3	0.2	1	0.062	0.097	-0.242
georgedo	0.02	0.3	0.02	1	0.062	1.0	0.0
cwallis				*No Season 2 Data*			
Jamie				*No Season 2 Data*			

Table 7.5: Correlation for simulation using an aspect-oriented model of learning applied to RPGLite season 2, with parameters identified as optimal for season 1 in Section 7.2.

Username	*p-value* <	τ >	*Confidence curve value*	*Confidence RGR modifier*	*Prob. bored*
Ellen	0.01	0.4	0.02	0.1	*disabled*
aaaa			*N/A*		
apropos0			*N/A*		
basta			*N/A*		
Becccca			*N/A*		
creilly1			*N/A*		
DavetheRave			*N/A*		
Deanerbeck			*N/A*		
DX13			*N/A*		
ECDr			*N/A*		
Ezzey			*N/A*		
Frp97			*N/A*		
Jhannah			*N/A*		
Luca1802			*N/A*		
l17r			*N/A*		
Martin			*N/A*		
Nari			*N/A*		
sstein			*N/A*		
timri			*N/A*		

Table 7.6: Correlation for simulation using aspect-oriented model of learning applied to RPGLite Season 2, annealing for parameters specific to season 2.

best-performing model parameters for playing season 1 (found during the second experiment) are shown in Table 7.5. The results of applying the learning model to season 2, and annealing to discover optimal parameters, are found in Table 7.6.

Results are shown for players who played at least 100 games in season 2, except for the model of learning run with parameters discovered in the season 1 experiment. As the parameters were found for players who were active in season 1, those players were re-used in season 2. Some players were active in season 1 but played no games; their results are marked as *No Season 2 Data* in the relevant results table, Table 7.5. As in Table 7.3, results for players for whom no parameters could be found which produce accurate data in season 1 are marked *N/A* when annealing toward new parameters using season 2 data, as shown in Table 7.6.

7.4.3 Answering the Third Research Question

The third research question can be answered affirmatively from the experimental results: Table 7.4 shows a high degree of correlation when used to simulate season 2 gameplay, and its success is comparable to that of simulating season 1 gameplay as shown in Table 7.2. Aspects used in the model applying a known distribution of character pairs to players' choices therefore port to other models of RPGLite. As these aspects are written as fuzzers which weave changes within their targets, this result extends to the novel type of advice introduced in PDSF3 as well as pre-existing types.

While the research question can be affirmatively answered, the results generated using the learning model in Table 7.5 and Table 7.6 show little correlation and so add nuance to these findings. Only one player, *ellen*, was successfully simulated using the same parameters in seasons 1 and 2. The same player was also the only one for whom parameters could be discovered using season 2 data which yielded statistically significant results across a majority of folds. As many fewer players are successfully modelled using these aspects in season 2 than in season 1, these results do not demonstrate that aspects which successfully introduce new behaviours or parameters to one model can be reused across different models. Several factors may contribute to these results.

These results may be influenced by the design of the learning model: it may more accurately represent the behaviour of a player who is new to a game than one who is adjusting to a change in that game. It may be that players respond to a change in RPGLite's metagame by using old strategies and adjusting them as needed, rather than discovering new strategies. The model of learning makes an assumption that simulated players have an initial lack of confidence which impacts how character pairs are selected, forcing players to explore possible character pairs and build preferences over time. If real-world players do not lose much confidence in their ability to select winning character pairs when seasons change then the assumption of the model would be incorrect, which could account for the learning model's

loss of accuracy in simulation season 2 players.

An alternative factor which may bias simulation accuracy may be habit. Players may build habits in season 1 which prove maladaptive in later seasons. If players were previously successful and did not realise that they may need to adapt to a changed metagame, their gameplay may be biased. They may also have grown accustomed to a particular strategy when playing, and either regard an increased rate of lost games in the new season as a result of RPGLite's random nature or simply refuse to change their character pair preferences out of stubbornness. The learning model does not account for any of these biases, which may impact its effectiveness when applied to simulations of season 2. Further research is needed to demonstrate that more complex augmentations of models such as additions of behaviours and parameters can be reused across models without a loss of model accuracy; some suggestions are given in Section 7.5.

Finally: advice implementing improvements to the model or instrumenting it to collect results have been successfully reused. If this were not so, the outcomes of games could not be recorded, and no players would be successfully modelled using the model of learning, as the confidence model would be unimplemented. As results are collected by instrumentation advice and user *ellen* is successfully simulated using the learning model in season 2, we infer that these pieces of advice have been successfully reused. Instrumentation for data collection is an example of logic typically included in model codebases but ancillary to their main concerns [55], making this an example of a cross-cutting concern in simulation & modelling which these experiments implement as aspects. These results add further evidence that aspects which augment models can be successfully reused, and so constitute portable modules in the context of simulation & modelling.

7.5 Summary

The experiments in this chapter contribute an investigation into the viability of using aspect-oriented programming for simulation & modelling and its practical utility in terms of model accuracy and its viability as a technique.

The experiment in Section 7.2 explores whether aspect-oriented changes to models can be realistic. It finds that altering minor properties of a model to improve its accuracy is feasible, and applies advice to change a distribution in RPGLite to produce player-specific gameplay simulations. This confirms that aspect-oriented tooling for simulation & modelling *can* be used to encode changes to models, but does not introduce changes on the scale of adding new behaviours or parameters to a model; it investigates the tooling's viability, but not its utility.

Section 7.3 contributes an investigation into the utility of aspect-oriented simulation & modelling by applying a model of novel behaviour with new parameters to the model of RPGLite. The model is accurate for 6 players out of 21. Considering additional factors such as or players' misunderstandings of the game or possible character selection criteria other than their success in games, this is a successful result. This model constitutes a study into the practicality of representing changes to a model as advice, and determines that advice can be used to amend models and improve their representation of a system under study.

Finally, Section 7.4 presents an experiment which studies the modular nature of aspects representing changes to models. It is anticipated that aspects can be used to encode behaviour which cuts across different models; to investigate this, the experiments presented in Section 7.2 and Section 7.3 are run again against gameplay data collected from RPGLite's second season. The experiments show that some aspects *can* be applied to other models, in particular simple modifications to a model which do not require parameterisation or add additional behaviour. However, Section 7.4 does not conclusively demonstrate this for advice which introduces new behaviours and parameters to a model. These findings may be biased by players' season 1

habits around character pair selection being carried into their season 2 behaviour. If players chose characters habitually rather than responding to RPGLite's changed metagame, no learning would be present in their gameplay data. Relatedly, players may not respond to the change of season with any behavioural change at all. This would mean that the learning model's assumption that players explore possible character pairs when they are inexperienced would not be appropriate: players would effectively begin the game with a high degree of initial confidence, which the model of learning does not represent. This would cause real-world gameplay data to compare less favourably to datasets produced by the simulation.

One player — *ellen* — was modelled accurately in season 2 in both experiments using the learning model. Also, the model applying a player-specific distribution of character pair selections to simulated players' choices produced successful results. Advice which instrumented the model or modified it to mitigate the impact of stalemates was also successfully reused in the experiments described in Section 7.4. For these reasons, the final research question can be answered affirmatively — however, further research is needed to examine whether more complex changes to models suffer a loss of performance. This can be investigated by creating aspects which represent behavioural change found in many systems, such as the models of distraction implemented by Fuzzi-Moss[136] and used in prior research [150] as discussed in Chapter 3. Applying these aspects across different case studies may yield positive results. This research should take care to avoid factors biasing behaviours such as habits or mistaken assumptions.

Chapter 8

Future Work

The focus of this thesis is to develop a state-of-the-art aspect-oriented framework, to produce a suitable experimental environment to demonstrate its effectiveness, and to use that environment to investigate whether aspect orientation is a suitable tool for the augmentation of simulation & modelling codebases. Chapter 6 and Chapter 7 showed that aspect orientation can be used to create a realistic and nuanced model using PDSF3 to model RPGLite. Having successfully achieved the aims of the project, these contributions enable opportunities which are outwith this thesis' scope.

This chapter describes some of those opportunities. It illustrates both the possibilities to extend the research presented in this thesis, and the other research projects adjacent to this one which are enabled by this thesis' contributions.

8.1 Aspect-Oriented Metaprogramming in Real-World Software Engineering

The combination of metaprogramming and aspect orientation introduces new possibilities in aspect-oriented programming. In traditional aspect orientation frameworks, aspects treat their targets as black boxes. This leads to limitations which aspect-oriented metaprogramming

such as PDSF3's fuzzers can address.

Traditional aspects cannot intersperse their behavioural modifications with the work being done by their target, as they apply their logic before or after the target's execution. The "textbook" use case for aspect-orientation is logging: aspects can separate logging from the business logic they are applied to. However, a programmer in mainstream programming paradigms may wish to insert logging behaviour *within* their business logic rather than *around* it. Aspect-oriented metaprogramming enables this as the target can have logging logic woven within business logic, which remains decoupled within the codebase. To achieve this end goal without "within"-style aspect application would require a refactoring of business logic to create join points which traditional aspects could apply against; this somewhat weakens the "obliviousness" property of the aspect-oriented philosophy discussed in Section 2.1.1 in that target code must be modified to apply advice. If the code must be modified anyway, that modification could introduce new logic rather than refactoring what already exists so that an aspect can do the same.

To demonstrate the improved pragmatism of aspect-oriented programming in real-world software engineering, existing codebases should be augmented using aspects. This work could take many forms. An industrial partner could adopt aspect-oriented programming in an existing project with consultation from a research team, who discuss the industrial team's experiences after some period of aspect-oriented development is completed. Alternatively, pull-requests in open-source projects could be reimplemented by a research team using aspect orientation, and the two implementations could be compared by the project's maintainers, who could be interviewed by the research team to discern whether aspect-oriented programming could be of practical use to them. The team may see aspect-oriented programming as a means improve the codebase's modularity and maintainability, or to deploy hotfixes in their code and features hidden behind feature flags which can be disabled during runtime by making use of dynamic aspect weaving. The research team may make use of aspect-oriented metaprogramming such as PDSF3's fuzzers to better support the team's application of aspect-

oriented programming to an existing industrial codebase. Some promising related work has been undertaken by Przybyłek [117], who compared student comprehension of aspect-oriented programs but did not evaluate the paradigm's appeal to industrial teams or its effectiveness as perceived by engineers in industry.

In either of the research projects suggested, the academic team seek to answer a research question such as:

> *Can aspect-oriented programming be used in existing industrial codebases to improve modularity and maintainability of the project as perceived by that project's team of maintainers?*

8.2 Augmentation of Pre-Existing Models

Models which predict the future state of some system may be accurate in the general case, but cannot account for unforseeable events such as financial collapses, pandemics, or unpredictable weather events. Pre-existing models cannot account for these one-off shifts in system state due to the random nature of these events. For example, models of world health over time could not account for the Covid19 pandemic, and models of local or global economies could not incorporate real-world data from the recession resulting from the pandemic. The 2008 financial crisis also impacted local and global economies, but could not be predicted before the event. An example of a model predicting both health and economic outcomes is the World3 model, which has provided accurate high-level predictions of the state of various global systems over many decades [10], but predictions for recent years fail to account for these unforseeable events [104].

If the modification is represented as a special case within the model, a research software engineer must introduce their corrective code alongside the model's core logic, thereby tangling[1] the two. It therefore has the properties of a cross-cutting concern [81, 39], and can be

[1] For an overview on the tangling of cross-cutting concerns, see Section 2.1.1.

factored into an aspect as such.

Research investigating the use of aspects to correct for one-off events could aim to show that an aspect can "nudge" a system's state in-line with real-world data when exceptional circumstances affect a modelled system but cannot be elegantly represented within the model's logic. If this technique proves successful, it may also lead to other novel ways of simulating systems. For example, future unforeseeable events could be compensated for by predicting them at different points in a simulated system's future and observing how their impact changes the simulation over time. This is discussed further in Section 8.4.

While there is potential for lots of related research in the engineering of models, a first study should be conducted to demonstrate that the concept is sound and that it is useful in practice. A research question this early study could answer is:

> *Can aspect orientation be used to introduce special cases to real-world systems to correct a model's predictions in when a system under research is altered by unforeseeable "freak events"?*

8.3 Aspect Orientation's Utility for Research Software Engineers

A corollary of the research opportunities for aspect orientation's use in software engineering discussed in Section 8.1 is that there is an opportunity for research software engineers to benefit from the adoption of aspect oriented programming. However, while aspect orientation's use in industrial software engineering has drawn criticism [132, 116, 28] its use within research codebases is a special case where it may be more suitable. The results presented in Chapter 7 show that aspect-oriented programming *can* be successfully employed in research codebases to represent changes to models. Related suggestions for the use of aspect orientation in research codebases were proposed by Gulyás and Kozsik (as discussed in Section 2.2.1). However, the benefits proposed in earlier work concerned the design of the software itself. Aside from aspect-orientation's utility in software design in a research context, there are

potential benefits for the *practice* of developing these codebases which may be of professional interest to research software engineers ("RSEs").

The resource constraints and stringent requirements for accuracy in research codebases present challenges which aspect-orientation could assist with. Compensating for a special case in a pre-existing model would require maintenance of the codebase, which takes time. Care must be taken not to inadvertently alter its behaviour. Time is a scarce resource in research environments, and undesired changes to a model's behaviour can invalidate research results. An alternative to adjusting the models directly is to construct aspects which represent large and unpredictable events in real-world systems such as pandemics, economic crises, war and famine. These can be modelled on real-world data for accuracy, which can produce realistic simulations as Chapter 7 demonstrates. The use of aspect orientation in simulation and modelling can be further investigated by creating a proof-of-concept of the approach as applied to pre-existing models.[2] Studies can also be conducted to investigate whether an aspectually-augmented model is quicker to construct and easier to maintain in future than a codebase with "patches" written into its original logic.

Researchers investigating this technique's application to existing models could also investigate the difficulties of augmenting a model originally implemented without any intention of weaving advice in the future. The construction of advice requires appropriate join points to be specified, and codebases which are structured in a way which doesn't yield convenient join points might be more complex to augment aspectually. These cases raise another use-case of PDSF3's "within"-style weaving through runtime metaprogramming: where other aspect orientation frameworks force aspects to treat the targets they are invoked on as black boxes, PDSF3 can make modifications within them. PDSF3's fuzzers may make aspect-oriented programming more useful than the aspects offered by other frameworks for maintainers of research codebases. This possibility requires future investigation.

[2]This approach is similar to PyDySoFu's initial proof-of-concept study [150] as discussed in Chapter 3.

If the research described successfully shows that aspectually-augmented simulations are easier and quicker for a research software engineer to maintain and deliver than direct maintenance of the model's codebase, then augmentation of existing models to improve their accuracy can follow in the community. Researchers investigating this may address research questions such as:

> ① *Can existing models developed* without *aspectual augmentation in mind be made more accurate through aspect-oriented "patches"?*
>
> ② *Can research software engineers use aspect-oriented programming to more easily maintain existing codebases without making invasive changes to experimental source code in the process?*

This is differentiated from the contributions in Chapters 6 and 7 by its application to pre-existing codebases. It is also related to the proposed benefits to research software engineers discussed in Section 8.2.

8.4 Hypothesising Possible System Dynamics via Aspects

Unpredictable events can cause discrepancies between simulated system state at a given time and the real-world system it models, as discussed in Section 8.2. While this technique presents promising research opportunities, researchers face other kind of model uncertainty which aspect orientation could also counter. In some scenarios it is difficult or impossible to make predictions about a system's future states, because its dynamics are actively being researched. Standard scientific practice is to create a model to create synthetic datasets which indicate accuracy if their predictions align with what is empirically observed [114]. Some aspects of the system under study may be well-understood.

Rather than creating models of an existing system which encode its hypothesised behaviour, a naive model can be created which operates as the scientific consensus understands it. Hypothesised behaviour takes the form of aspects altering the model in any manner the hy-

pothesis requires. Within-style aspects enable arbitrary modification to simulated behaviour, which increases aspect-oriented programming's potential in this use-case. Data produced by each model can be compared to empirically sourced data, and their similarity quantified, as demonstrated in Chapter 6 and Chapter 7. An experiment's null hypothesis would be that the naive model's similarity to empirically sourced data is greater than that of the aspectually augmented model; the experiment's hypothesis would be that the behavioural change applied as aspects is more representative of the system under study than that of the community consensus (encoded in the naive model).

This technique has a satisfying property: the hypothesis in a given experiment is completely encoded by its aspectual representation, and if an experiment is successful then the augmented model can be adopted in future research. In this way, experimental design and scientific process are directly represented by the structure of the codebase, and the community's progression to increasingly accurate models of a system is represented by the progressive adoption of aspects as "patches" to an original theory.

Hypotheses can also be created compositionally in this model. Researchers might develop a series of potential system properties or behaviours, but are unable to investigate all reasonable combinations in a timely manner. However, sets of aspects representing each can be composed to produce 2^N models of potentially realistic behaviours from N hypothesised behaviours by applying each possible combination. The results produced by each can be compared to an empirical dataset to identify which combination of behaviours most closely resembles that of the real-world system under study.

Finally: hypothesised properties of a system might interact, meaning their discovery could involve the fitting of multiple models of possible behaviours to discover the properties of a real-world system. The use of aspects to encode — and discover the presence of — hypothesised behaviour is discussed in Section 8.11 as the RPGLite dataset presents convenient opportunities for doing so. The optimisation of multiple aspectual models of behaviour is discussed in

Section 8.9.

8.5 Aspect-Oriented Models to Support the Investigation of Scientific Progress

The technique discussed in Section 8.4 for developing experimental model codebases has another desirable property: it simultaneously reflects different philosophies of the scientific process in its encoding of hypotheses and the research community's acceptance of a perspective in their fields. These philosophies are those of Kuhn and Popper. Kuhn [85] explains the scientific process as inherently social: it starts with a paradigm which is accepted as broadly true, and accumulates an increasing number of exceptions until the paradigm itself is deemed unfit, and a new basis for a field's research is adopted. Popper [114] explains the scientific process as an approximation towards truth, with incremental progress made with each result achieved by a research community. The proposed technique for developing experimental model codebases demonstrates features of both.

To illustrate this, consider the original model a paradigm initially selected by community consensus. The successful application of aspects would be equivalent to Popper's incremental movements toward truth as successful experiments are conducted; Kuhn's exceptions to the agreed model can be identified as experiments which demonstrate weaknesses in the original model. In this case, each successive new model adopted by a community is adopted in a Popplarian manner: improvements are objectively measured, incremental, and would trend towards truth as a model's behaviour fits empirical observations increasingly closely. However: over a sufficient period of time, the incremental patching of an original model would produce an accepted community model which contains a relatively large amount of discovery and complexity encoded in aspects, as compared to the original model they are applied to. One would expect the research community to rewrite the base model to simplify future aspect application and to more elegantly encode recent research findings; effectively discarding the original paradigm in favour of a new one. This process is Kuhn's "paradigm shift", where

paradigms are dropped once a generation of researchers determine that an originally accepted theory on a topic is unfit for purpose as evidenced by mounting exceptions in the literature; a new paradigm is to be accepted by the community, as a new base model would have to be written and adopted.

The relevant philosophy of science is more nuanced than its brief explanation here, and the suitability of the approach for the development of research codebases is to be investigated; the work involved is outwith the scope of this thesis, but is suggested as future work. A basis for the incremental improvement of models via aspects is effectively demonstrated in Chapter 6 and Chapter 7, but the feasibility of the approach as a basis of a community's scientific process and relation to philosophy of science warrants further investigation.

8.6 Standards for Aspect Orientation in Research Codebases

The possibility of a research community sharing their research as aspect-oriented changes to model involves model logic written as advice. The community developing these aspects have the responsibilities of maintaining a codebase as well as the added complexity that research software engineering introduces. These codebases may be used for many years, and may be iterated on in a series of future experiments. The legibility of these codebases and their long-term maintenance are areas of criticism in the software engineering community [28, 132, 116]. The research community must therefore mitigate these weaknesses of the aspect-oriented paradigm when adopting the techniques discussed in this chapter for simulation and modelling.

To address the concern of the visibility of advice being woven, researchers may already take advantage of improvements to tooling produced by the aspect orientation research community, including IDE integration [24] and runtime inspection [99], should make clear to engineers what advice is being woven in a codebase and assist with debugging aspect-oriented programs

respectively. In addition, aspect orientation frameworks specifically designed to clarify to an engineer the aspects being woven should allow for less friction on the part of a maintainer who inherits a codebase from another developer and must reason about its behaviour. This is particularly important if the maintainer aims to weave more aspects into the codebase, and so must understand its existing behaviour before augmenting it further. Adopting weaving patterns such as import hook weaving — described in Chapter 4 — should make a program clearer to a developer regardless of the tooling they have access to.

The impact of framework design on a codebase's maintenance should assist a developer even in the absence of tooling, but the success of import hook weaving in this regard is untested. An appropriate research question which arises is therefore:

> *Do aspect orientation frameworks with weaving techniques designed to simplify a developer's understanding of a program affect a codebase's long-term maintainability?*

8.7 Standard Aspect-Oriented Model Features

Researchers who build aspect-oriented models and extend others' aspect-oriented codebases must be able to collaborate at least as easily as they currently do in a culture without aspect orientation. One way aspect orientation might improve researchers' ease of collaboration is with standardised libraries for aspect construction. Similar libraries were developed when developing a case study for PDSF3's viability [150, 136], but additional opportunities to complete and expand the tool remain as discussed in Section 3.3.1.

The original library with this aim, Fuzzi-Moss [136], was originally designed to provide standardised aspects to represent behavioural variance in socio-technical systems [150]. The aspects developed in Fuzzi-Moss were not developed with a notion of being fitted to real-world data. Instead, they are simple models of behavioural variances such as distraction, which are parameterised to allow users of the library to use these simple models in whatever manner

is appropriate for their use-case. A broader collection of these behavioural variations could simplify the use of aspect-oriented behavioural variation in the research community writ large, by removing researchers' burden to develop these themselves. Early construction of a library with Fuzzi-Moss' goals would also support researchers in sharing models or extending others', as they would be familiar with a common set of tools. The development of such a library and the production of case studies demonstrating its effectiveness would answer the research question:

> *Can researchers using aspect-oriented behavioural variance share a common set of tools to support and simplify the use of the technique in their codebases?*

Other tooling to support researchers in the development of aspectually augmented simulations and models could also be developed. For example, a library of fuzzers which make changes to an abstract syntax tree could be constructed. Such a library would not model specific behaviours, but would allow researchers to build models performing within-style aspect weaving without writing code which contained no metaprogramming logic. Instead, this logic would be encapsulated in utility functions provided by the proposed library. This would also reduce the work required of researchers looking to use the techniques demonstrated in this thesis. The library could also support the development of a Fuzzi-Moss-like set of socio-technical behavioural variances. However, such a library does not currently exist. A summary of the contribution to the research community which the proposed library would make is:

> *The development of a library of metaprogramming operations which simplify the construction of within-style aspects, supporting its research use and demonstrated in case studies.*

The proposed library need not be a separate codebase to the aspect orientation framework it is designed alongside. Instead, it could be developed with interoperability in mind to permit reuse across different implementations of aspect-oriented programming. This would enable

researchers who develop their models as aspects to share their work with others who use other frameworks. Precedent for interoperability between frameworks has already been set by AspectJ, for which the accompanying DSL has been adopted by SpringBoot's implementation of aspect-oriented programming [123]. To achieve this, some standardisation around the design of aspect orientation frameworks would be required, so that aspects can be developed against a common foundation.

8.8 Testing Frameworks to Detect Model Incorrectness

Many of the uses of aspect orientation in simulation & modelling research discussed in this chapter affect the construction and maintenance of a codebase. However, aspect orientation also has potential in the instrumentation of an experiment. This was theorised by Gulyás and Kozsik [55] and put into practice in this thesis to instrument the naive model of RPGLite to observe its state (see Section 6.5.2 for a description of the relevant aspects). Instrumentation has potential to be used in aspects of research software engineering for purposes other than the observation of system state: for example, aspects may be useful to alert researchers to impossible states visited by a simulation by instrumenting them to observe a simulation's state and assert that it is within expected bounds.

Many models construct a version of a real-world system which cannot achieve certain states. For example, a model of collisions might never be expected to see an increase in the total energy present within the model; simulations of socio-technical systems might expect bounds on the hours worked by a simulated workforce, or the amount of output the workforce creates; researchers might study a system to observe an emergent property which is bounded by laws around its growth (for example, a linear increase in some property over time as opposed to exponential growth). It is important that any model of a real-world system accurately reflects the system's physical limits and underlying mechanics. Models may fail to reflect these mechanics for many reasons, including:

- The model may be developed with bugs which are not detected, i.e. it is conceptually incorrect.
- The model may be verified and trusted, but later maintenance introduces undetected errors, i.e. it contains bugs.
- The model may be conceptually correct, but given unexpected inputs.
- The model may be conceptually correct, but a bug may exist in a dependency, on certain hardware, or in some other detail specific to the environment it runs in as opposed to its implementation.
- The model may be conceptually correct when initially implemented, but fail to meet researchers' expectations at a later date due to a shifting perspective within the research community. This may not be apparent when reading the model's source code, but may be an emergent property arising from interactions within the system. Non-deterministic interactions leading to emergent properties would be difficult to verify through static analysis.

Software engineering practices such as unit testing are common practice and may catch these bugs through identifying incorrect behaviour in model components. Thorough test suites may also include integration tests which observe the behaviour of many model components working together. However, there are limitations of both testing techniques:

- Test suites as described may fail to identify non-deterministic errors in a model, which arise only in a small number of cases;
- Traditional test suites also may fail to identify emergent state errors, where individual components operate correctly but their repeated integration results in an incorrect emergent property of the system; [3]

[3] Emergent properties are often the state under study, as in PyDySoFu's original case study [150].

- Test suites often fix values such as random seeds or input data to remove non-deterministic behaviour. This may hide errors arising from non-determinism in a modelled system or may not exhibit the properties of real-world data, thus exhibiting different behaviour under test than when an experiment is conducted, making it difficult to verify in conditions which differ from those of an experimental run.

Chapters 6 and 7 showed that aspects can be used to instrument models to assert a system state within expected bounds, both as components of a test suite and as assurance that a model was conceptually correct at any time it is used as part of an experiment. All of these limitations can be alleviated by a test suite which is able to assert the correctness of model state at different points in a program's execution. Aspects can be used to instrument a model and observe its behaviour while a simulation executes as part of an experimental run, ensuring that the results of the experiment were not impacted by any incorrectness tested for. These states may also be deterministically caused by experimental input data which is not reflected by the inputs used by a test suite. Testing for erroneous state during an experimental run ensures that incorrect states were not reached through unexpected inputs, meaning they could not affect results and so alleviating this issue. Emergent properties of a system may also be tracked over time, and aspects may measure emergent properties while a simulation executes to ensure they are valid at all times. For example, these aspects could ensure expected growth curves for the property in question and check that values are within physically possible bounds.

Aspects can also be used as regression tests to detect the recurrence of bugs after they have been identified and fixed. If a simulated system enters an incorrect state during development, and the underlying bug is fixed, an aspect observing the system's erroneous property can be constructed which alerts developers if the bug is re-introduced.

Aspects instrumenting a model throughout its use in an experiment would also serve to guarantee peer reviewers of a study that the model was correct with regards the properties

instrumented by aspects. Functionally speaking, a suite of aspects which observe bounds on and characteristics of system properties is a declarative set of known-good behaviours of a model. Peer reviewers who sought to check that a model was correct should be able to extend this list of properties and observe any assertions when reproducing a study's findings.

Within-style aspects are particularly appropriate for this task, as the measurement desired of a program might exist *inside* an existing function or block of code. It is conceivable that observing system state before and after a function executes would not be sufficiently granular to assert correct system state in some cases, particularly if the model's codebase does not include suitable join-points to weave onto. Aspectual instrumentation of research codebases is therefore feasible for many more models using PDSF3's within-style fuzzing.

To demonstrate the efficacy of this approach, a library of tools to observe boundaries on system states could be developed, similarly to the library of tools suggested in Section 8.7. Case studies augmenting research codebases could then be constructed. These should include new codebases, to demonstrate the technique's utility during development, but should also include the instrumentation of old models and the detection of any irregularities in existing, published work. Such a project would answer the research question:

> *Can aspect orientation be used to instrument a model and assert correctness by observing that the model's properties remain within expected bounds?*

8.9 Optimisation of Multiple Aspects

Experiments in Chapter 6 and Chapter 7 made use of models which were fitted to real-world data. However, they only fitted parameters of learning models, and only some parameters were fitted. This was partly due to the computational cost involved in fitting these parameters and time constraints on this research.

Optimisation is an independent research field with a broad range of techniques outwith the

scope of this thesis to explore; however, a study of optimisation algorithms and technologies as applied to the fitting of aspects to real-world system models is a topic for future research. Models of real-world systems can be computationally expensive to complete, meaning that measuring the accuracy of a model with a variety of parameters could be computationally expensive. A survey of optimisation techniques suitable for aspectually augmented simulations and models would answer the research question:

> *Which optimisation techniques are best suited to fitting parameters of aspects woven into models of real-world systems, with regards the accuracy of the aspectually augmented model produced?*

8.10 Future Work pertaining to RPGLite

RPGLite's dataset was analysed for the purposes it was collected for in this thesis: to aid in the realistic simulation of a well-controlled socio-technical system. However, some pieces of analysis have not been completed. This section discusses future work enabled by the dataset of real-world play which this thesis contributes.

8.10.1 Causes of Game Abandonment

The games included in the dataset produced by RPGLite finish in varying states: some are incomplete (as some games were ongoing when the snapshot of the game's database was taken), and others are finished. A third category of games are ones where a player opted to abandon a game they were participating in. This feature of RPGLite was developed to allow players who were playing inactive opponents to free a game slot. It also allowed players who felt they had no hope of winning — or were bored with a particular game — to forfeit and start a new one.

As many data points are included in the published dataset, it is possible to observe how

active players were (for example, by measuring the frequency with which they would open RPGLite), and engagement with the app might be quantified by players' exploration or frequent use of leaderboards and customisation features. Correlations might exist between players' activity, engagement, or experience with their propensity to abandon a game in different states. One hypothesis could be that particularly active players frequently forfeited games against inactive players out of boredom, or that players who were unlikely to win a game (or were inexperienced players of RPGLite paired against competent players) might be more likely to forfeit. These results might be of interest to the broader game design community in industry and research. Research investigating reasons for abandonment, for example, could answer the research question:

> *Can players who are likely to forfeit a game be identified through patterns of RPGLite play?*

8.10.2 Patterns of Play within Cliques of Players

RPGLite's application implemented matchmaking systems which allowed players to advertise themselves as open to new games, challenge players discovered on a leaderboard, or rematch against players they had previously played against through a game history. However, two factors might have impacted how players chose their opponents:

① The recruiting methods used mean that people might have known each other before playing RPGLite and so might be biased toward playing each other rather than other members of the community,

② Players who find others they are well-matched against might repeatedly re-match rather than continuing to discover players they might have less satisfying games with.

These factors would imply the formation of cliques of players: small groups who often created games against each other much more frequently than they did with the game's broader

community. If so, this could have implications for uses of the PRGLite play dataset. For example, in the experiments discussed in Chapter 6 and Chapter 7 no structure is imposed on the matching of simulated players, meaning they are are uniformly exposed to their simulated community. As these experiments simulate learning from the outcomes of games, exposure to the entire community would expose players to learnings from all games played; cliques of players have a limited set of games to learn from, and so could affect what is learned by that clique. The broader community would also be insulated from what was learned within the clique, meaning that a loss of information affects all players.

Future research making use of the RPGLite dataset should be aware of any underlying patterns which could affect the dataset's emergent properties, such as learning styles. Clique formation is an example of such a pattern; others might yet present themselves. Future research making use of the dataset of RPGLite play may look to address research questions such as:

> *Are patterns such as cliques of players present in the RPGLite dataset? If so, what patterns exist, and what is likely to have driven the formation of those patterns?*

8.10.3 Large-Scale Data Collection

There is scope for a larger and longer-term RPGLite data collection effort to be made than this thesis had scope for. RPGLite's player base was recruited informally, only two seasons of the game were run, and it was not heavily advertised to prospective players once it was clear that sufficient data would be collected for the research that required it. A re-release of the application in mobile app stores with a concerted effort to release new seasons of the game and maintain player interest for an extended period of time would enable a richer analysis, and broader utility to the games research community. New RPGLite data collection efforts could also take the opportunity to expand the game's implementation with additional features, such as in-game chat, favourites lists of previous opponents, or match replay and analysis with

suggestions for improved play backed by the formal methods inherent in RPGLite's design

RPGLite's dataset contains information about players' interactions with the application itself. As the game made available some features typical of modern games such as leaderboards, matchmaking, achievements, and graphical customisation, player data containing interaction data on features which are used commercially could be used to shed light on the more effective aspects of modern game design for engagement purposes. There are many opportunities available for the game design research community to investigate. Unfortunately, they remain outwith the scope of this thesis, which focuses on simulation technologies more than it does game design. However, Some already-published work reflects further on the design and possible future improvements of the game [75, 154, 72, 71, 73].

A large-scale user study of RPGLite could also be used in pursuit of other research questions suggested in this chapter. Players could be classified by their likelihood of abandoning games using the patterns identified in Section 8.10.1 to verify the patterns empirically. A larger dataset might be useful in testing the discovery of system behaviours using hypothesised behaviours encoded in aspects. This could be used for the aspectual encoding of an experimental setup, as discussed in Section 8.4, the discovery of features of player behaviour suggested in Section 8.11, the discovery of dataset properties such as player cliques including the real-time prediction of clique formation to during data collection for experimental verification of a clique classification process discussed in Section 8.10.2, or the fitting of multiple aspects to models as described in Section 8.9. Relevant research questions are discussed in their respective sections of this chapter.

8.11 Aspect-Oriented Discovery of System Properties

The RPGLite gameplay dataset presents opportunities for research in aspectually augmented simulation & modelling which is related to the future work described in Section 8.4,

which discussed the use of aspect orientation to encode hypothesised behaviour. Researchers may be able to use aspects to discover unknown properties of a real-world system. This would be achieved by writing aspects which alter a simulation to discern some pattern in it, or in the behaviours which produced a dataset that simulation uses. Rather than writing aspects to encode a scientific hypothesis, they could also be used as an investigative tool.

The existence of some biases which could affect RPGLite player behaviour is unknown. However, these may be discernable by encoding the behaviours as aspects and observing whether player behaviour is made realistic when they are woven. Examples of possible biases include: whether cliques were formed by RPGLite players; the behavioural impact of player interaction with ancillary game features such as leaderboards or profile customisation; whether players' early aptitude for RPGLite may cause them to lose interest relatively quickly due to a lack of challenge or spur them to climb leaderboards or continue to win games; whether players more likely to challenge others with more experience learn RPGLite quickly and be more likely to continue playing, or quickly lose interest due to repeated loss. These may be identified by writing aspects which encode them and observing whether simulated play produces realistic data.

While these behaviours can be hypothesised, many might have an impact on each other. Players might find experienced peers through the leaderboard, so their progression in or engagement with the game could be due to their propensity to explore (by using the leaderboard) or their exposure to more advanced play (through games with experienced peers). This can be investigated through aspects fitted to real-world data, with parameters describing the strength of their effect.[4] The *relative* importance of each concern could be discovered by determining accurate values for the parameters controlling the strength of every one, similarly to the experimental design used in Chapter 7. This would discern the existence and relative impact of real-world behaviours.

[4] Research into relevant techniques is proposed in Section 8.9.

This proposal is an extension of the research discussed in Section 8.4, but the RPGLite dataset is a suitable testing ground for the technique which already exists. Larger-scale data collection as described in Section 8.10.3 could provide opportunities to test this more rigorously, as the discovery of emergent properties of RPGLite play might require a larger dataset than this thesis had the scope to contribute. Future work demonstrating this technique could address the research question:

> *Can potentially interacting emergent properties in a real-world system be identified through the fitting of aspectual models representing hypothesised properties to empirical datasets which may exhibit them?*

8.12 Discussion

The contributions presented in earlier chapters motivate more research efforts spanning multiple fields, and has implications for experimental design as well as software engineering and simulation & modelling techniques. This chapter contributes an investigation into this future work by identifying specific opportunities yielded by earlier contributions and proposing research questions to precisely communicate the future work enabled by those contributions.

It is important to examine new research results. Contributions can impact existing results, and the theories they are founded upon. Kuhn [85] notes (emphasis added):

> […] new inventions of theory [are not] the only scientific events that have revolutionary impact upon the specialists in whose domain they occur. The commitments that govern normal science specify not only what sorts of entities the universe does contain, but also, by implication, those that it does not. It follows […] that a discovery like that of oxygen or X-rays does not simply add one more item to the population of the scientist's world. Ultimately it has that effect, but not until the professional community has re-evaluated traditional experimental procedures, altered its conception of entities with which it has long been familiar, and, in the process, shifted the

> network of theory through which it deals with the world. *Scientific fact and theory are not categorically separable, except perhaps within a single tradition of normal-scientific practice. That is why the unexpected discovery is not simply factual in its import and why the scientist's world is qualitatively transformed as well as quantitatively enriched by fundamental novelties of either fact or theory.*

The ideas presented in this thesis — in particular the aspectual augmentation of simulations & models — are not new "theories" in the sense that they contribute any profound or revolutionary new paradigms. However, they do contribute a new approach to simulation & modelling, and do so through new aspect orientation techniques. There are therefore contributions in both fields in which the "scientific fact and theory" Kuhn refers to are affected: in both fields new research opportunities and experimental practices are made feasible. These contributions concern both research *facts* such aspect-oriented runtime metaprogramming, and *theories* forming the foundation of those facts such as the use of new aspect orientation technologies in experimental design.

New capabilities of aspect orientation frameworks are suggested, built, and demonstrated in experimental case studies through PDSF3. Their design and the extension of the tooling around these concepts present opportunities for future work as discussed in Section 8.6 and in Section 8.7, but the implications of their use in software engineering requires further study, as discussed in Section 8.1. There are specific software engineering implications for research software engineering, as discussed in Sections 8.2, 8.3, 8.8 and 8.9, Further, the methodologies demonstrated here yield metascientific research opportunities, as discussed in Sections 8.4, 8.8, 8.9 and 8.11. This thesis contributes not only the ideas presented in earlier chapters, but a discussion of their implication for Kuhn's "scientific fact[s] and theor[ies]" in this chapter for the relevant fields.

The dataset produced by RPGLite also offers opportunities for further study. Research opportunities presented by the game design and formal methods are explored by Kavanagh [71], but the dataset this thesis helped to contribute also offers opportunities in the mining of player behaviour and the dataset's use in future aspectually augmented modelling research,

particularly if a large-scale data collection study is undertaken. These possibilities were discussed in Section 8.10.

Chapter 9

Closing Discussion

The discussion in this chapter summarises the findings made in earlier chapters, and provides some concluding remarks.

First, the contributions made in earlier chapters are summarised to provide an overview of the research as a whole. Next, some limitations of the findings and methodology are acknowledged in the spirit of scientific integrity. The chapter then ends by reviewing the core concept of the thesis: that parts of scientific models can be written as — or refactored to be — advice, using aspect-oriented programming. This concept is examined through the lens of earlier chapters' investigations & results.

9.1 Summary of Contributions

Several contributions are made in earlier chapters. This section summarises them individually to provide an overview of the presented research and its conclusions. These contributions were made in response to opportunities found in Chapter 2's literature review, and are distinct from the contributions of earlier work as outlined in Chapter 3.

A new aspect weaving technique aiming to improve aspect-oriented code's legibility The principle of obliviousness which forms part of aspect-oriented programming's design philosophy [39, 76, 14], but critics note that it makes the intended behaviour of a program difficult to ascertain, and should therefore be used in moderation [87] or not at all [116, 28]. PDSF3 applies aspect hooks to modules as they are imported, meaning that modules which *can* be altered by aspects are identified not only when weaving is applied, but when any module is used. As this design preserves obliviousness, function implementations still contain no reference to any potentially applied aspects — but any other module which invokes that function must import it with hooks applied.

Advice may still be woven anywhere within the module calling the function, but the entire module does not need to be searched for a call to PDSF3's weaver: developers can simply find the module's import to check whether it can be influenced by aspects. This reduces the possible parts of the codebase one must check to identify whether arbitrary changes could be made to a function's behaviour from anywhere at all to a single, well-defined point in the program, addressing the limitation of obliviousness raised by critics such as Leavens and Clifton [87] or Constantinides, Skotiniotis, and Stoerzer [28] without sacrificing the design principle itself.

Development of a tool which is suitable for use in aspect-oriented modelling The application of advice to existing codebases requires some technique to overcome a lack of useful join points in a pre-existing model. In addition, aspect-oriented programming has been subject to criticism that obliviousness limits program legibility, an important quality of software developed for research purposes. To address this, a redesign of PyDySoFu was developed which provided compatibility with modern versions of Python and improved on the design of its hook weaver to address criticisms of aspect-oriented programming's legibility by adding aspect hooks at import time, thus preserving obliviousness while improving legibility.

Demonstration that a model's behaviour can be augmented to accurately reflect a system under study using advice

The review of relevant literature in Chapter 2 observes that many research projects in aspect-oriented programming produce tooling, but do not confirm that tooling's hypothetical contribution empirically, and few case studies exist which demonstrate the benefits of aspect-oriented programming empirically. It is therefore particularly important that PDSF3 is demonstrated to successfully represent changes to a model as advice.

PDSF3 was used to apply advice which augmented a naive model of RPGLite play; character selection was altered to reflect the characters selected by real-world individuals, as observed in the RPGLite gameplay dataset [74]. This advice successfully produced synthetic gameplay datasets which correlated with the individuals being modelled by the advice. Advice is therefore demonstrated to be a viable mechanism for the encoding of changes to a model, producing the expected datasets when applied. PDSF3 is also shown to be used successfully in a simulation & modelling setting.

A model of RPGLite play, and corresponding dataset of player interactions Supporting the earlier contributions is a model of RPGLite play, paired with a dataset of RPGLite player interactions [74]. Both are publicly available in support of other researchers' future work.

An exploration of the possibilities which aspect-oriented modelling yields The demonstration that one can successfully represent changes to models as aspects yields novel research opportunities which are largely undocumented in the literature. Having demonstrated that model behaviours can be woven as advice rather than calcifying the changes in a model's source code, a large body of future work can be produced investigating more specific uses of the technique and applying it in novel, ways.

The future work illustrated in Chapter 8 follows the example of Marsh [94], whose thesis introduces a formalism of trust and also yields a large number of research opportunities. Marsh notes that there exists such a large number of avenues of research which existing literature does not identify in any fashion that their thorough discussion of now-feasible

opportunities constitutes a contribution in its own right; "future work" in the context of their results doesn't only refer to improvements on their own findings, nor to more advanced versions of their own formalism, but to entirely new research projects across a variety of fields. Similarly, the application of aspect orientation to simulation & modelling supports novel research in applying the technique to existing codebases, using the technique to develop rigorous methodologies for the acceptance of changes to models, alternative forms of collaboration for research teams, and other "future work" possibilities. These possibilities are enumerated in Chapter 8 as a thorough exploration of earlier contributions' significance. Where possible, all future work identified is accompanied by a specific research question, to qualify the particular contribution that work would yield and to simplify other researchers' engagement with the topic.

9.2 Limitations

As with all research, the contributions summarised in Section 9.1 have some limitations. These are acknowledged here in the spirit of openness, and because identifying weaknesses in any piece of work helps to improve anything building on it.

Import Hook Weaving does not solve all problems with Obliviousness A weakness of aspect-oriented programming which its critics identify [132, 28, 116] is that aspect-orientation's principle of obliviousness makes a program more difficult to reason about. Obliviousness — that the join-point of some advice is unaware that a weaver might change it — complicates reasoning about a program, because it's unclear from reading its source code that additional logic may be included when it is run, where that logic is to be woven, and what it is to do. PDSF3's ability to weave changes *within* its join-point as opposed to before or after it introduces new ways for a program to exhibit unexpected logic. The only way to identify whether a program been affected by PDSF3 is to identify how it is used, by observing where it is imported and whether PDSF3 is utilised there.

Other aspect-orientation frameworks provide tooling to facilitate easier inspection of aspect-oriented programs; at present, this is not provided for PDSF3.

Only one instance of aspect-oriented modelling is investigated The experiments described in Chapter 6 and Chapter 7 investigate the use of aspect-oriented programming in a simulation & modelling context, but only apply aspect orientation to one model, and specifically one which is socio-technical in nature. As advice is a feasible mechanism to represent changes to models, changes which do not relate to behavioural variance should be considered also. Promising related work includes research by Cieslak et al. [21] which represents details of plant growth in aspect-oriented L-systems. The technique could be used to model more socio-technical systems such as degraded modes [70] and epidemiology [107] or tackling modelling problems in other fields, such as diverging behaviours in astronomical models [83, 125] or cross-cutting concerns in business-process modelling [13, 129] and botany [21]. Future work should prove aspect-oriented programming's utility in a broader array of contexts.

Further research is required into the portability of aspects across models Results for the third experiment, in Section 7.3, investigated the portability of aspects across models and found mixed results. The model of learning appears to successfully model players in season 2 of RPGLite but is unable to model as many players as it could for season 1 regardless of how the model is used. One explanation of this result is that biased player and assumptions within the learning model impacted the aspect's suitability to modelling a second season of RPGLite. Aspects implementing the prior distribution model are successful in simulating players across seasons; therefore, the approach seems viable, but future research should clarify the weaknesses of the learning model as applied to season 2 of RPGLite.

9.3 Aspect-Oriented Modelling

This thesis presents research into the application of aspect-oriented programming to the development of models and simulations in a research setting. The use of aspects to represent changes to a model is novel. Core to the approach is the notion that parts of a model such as behavioural traits, additional parameters (and their impact on a model), and behaviour contingent on environmental factors can be suitably modularised from it as cross-cutting concerns. Similarly, minor changes to a model such as altering existing model properties can be represented as advice rather than as a change to the model's source, making the change easily enabled or disabled without adding complexity to the model itself.

Verification that the technique is viable — and a demonstration of it — required the development of new tooling and the application of that tooling to investigate the benefits augmenting a model using aspects. PyDySoFu was rewritten to support Python 3 and to weave aspect hooks at import time, allowing for dynamic and flexible weaving of advice. The result was a more mature aspect-oriented modelling tool, PDSF3, which introduced new techniques for weaving aspect hooks — "import hook weaving — which addresses criticisms of aspect-oriented programming as a paradigm. A system suitable for simulation — RPGLite — was designed, implemented as a mobile game, and released to collect data to support modelling efforts. A model of RPGLite was constructed, and a formalism of learning and confidence was defined, implemented, and used as the foundation of experiments investigating aspect-oriented modelling. Three experiments were designed and conducted to investigate whether aspect-oriented programming could be used to change models, to compose a model with more complex behaviours, and to build single units of model change which are applicable across multiple models. Positive results from these experiments indicate that the technique may apply to other fields and use cases, and the breadth of these opportunities was explored in its own chapter.

Aspect-oriented programming is shown to successfully modularise changes to models.

Lots more can be done to explore how it can be used and to identify its limitations; the work presented shows that it is feasible and promising.

References

[1] Bourouis Abdelhabib and Belattar Brahim. "JAPROSIM: a Java framework for process interaction discrete event simulation". In: *Journal of Object Technology* 7.1 (2008), pp. 103–119.

[2] Rakesh Agrawal, Dimitrios Gunopulos, and Frank Leymann. "Mining process models from workflow logs". In: *Advances in Database Technology—EDBT'98: 6th International Conference on Extending Database Technology Valencia, Spain, March 23–27, 1998 Proceedings 6*. Springer. 1998, pp. 467–483.

[3] AU Aksu, Faruk Belet, and B Zdemir. "Developing aspects for a discrete event simulation system". In: *Proceedings of the 3rd Turkish Aspect-Oriented Software Development Workshop*. Bilkent University. 2008, pp. 84–93.

[4] Jason Baker and Wilson Hsieh. "Runtime aspect weaving through metaprogramming". In: *Proceedings of the 1st international conference on Aspect-oriented software development - AOSD '02*. ACM Press, 2002. DOI: 10.1145/508386.508396. URL: https://doi.org/10.1145%2F508386.508396.

[5] Gordon Baxter and Ian Sommerville. "Socio-technical systems: From design methods to systems engineering". In: *Interacting with computers* 23.1 (2011), pp. 4–17.

[6] Brahim Belattar and Abdelhabib Bourouis. "Yet Another Java Based Discrete-Event Simulation Library." In: *J. Softw.* 9.1 (2014), pp. 82–88.

[7] Steve Benford et al. “On Lions, Impala, and Bigraphs: Modelling Interactions in Physical/Virtual Spaces”. In: *ACM Trans. Comput.-Hum. Interact.* 23.2 (May 2016). ISSN: 1073-0516. DOI: 10.1145/2882784. URL: https://doi.org/10.1145/2882784.

[8] Colin PD Birch. “A new generalized logistic sigmoid growth equation compared with the Richards growth equation”. In: *Annals of botany* 83.6 (1999), pp. 713–723.

[9] Jonas Bonér. “AspectWerkz–dynamic AOP for Java”. In: *Invited talk at 3rd International Conference on Aspect-Oriented Software Development (AOSD)*. Citeseer. 2004.

[10] Gaya Branderhorst. “Update to limits to growth: Comparing the World3 model with empirical data”. PhD thesis. 2020.

[11] JCAM Buijs. “Flexible evolutionary algorithms for mining structured process models”. In: *Technische Universiteit Eindhoven* 57 (2014).

[12] Muffy Calder et al. “Real-time verification of wireless home networks using bigraphs with sharing”. In: *Science of Computer Programming* 80 (2014), pp. 288–310.

[13] Claudia Cappelli et al. “An aspect-oriented approach to business process modeling”. In: *Proceedings of the 15th workshop on Early aspects - EA ’09*. ACM Press, 2009. DOI: 10.1145/1509825.1509828. URL: https://doi.org/10.1145%2F1509825.1509828.

[14] Anis Charfi and Mira Mezini. “Aspect-Oriented Workflow Languages”. In: *OTM Conferences*. 2006.

[15] Anis Charfi and Mira Mezini. “Ao4bpel: An aspect-oriented extension to bpel”. In: *World wide web* 10.3 (2007), pp. 309–344.

[16] Anis Charfi, Heiko Müller, and Mira Mezini. “Aspect-Oriented Business Process Modeling with AO4BPMN”. In: *Modelling Foundations and Applications*. Ed. by Thomas Kühne et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 48–61. ISBN: 978-3-642-13595-8.

[17] Meriem Chibani, Brahim Belattar, and Abdelhabib Bourouis. “Toward an aspect-oriented simulation”. In: *International Journal of New Computer Architectures and their Applications (IJNCAA)* 3.1 (2013), pp. 1–10.

[18] Meriem Chibani, Brahim Belattar, and Abdelhabib Bourouis. “Practical benefits of aspect-oriented programming paradigm in discrete event simulation”. In: *Modelling and Simulation in Engineering* 2014 (2014).

[19] Meriem Chibani, Brahim Belattar, and Abdelhabib Bourouis. “Using aop in discrete event simulation: A case study with japrosim”. In: *International Journal of Applied Mathematics, Computational Science and Systems Engineering* 1 (2019).

[20] Ruzanna Chitchyan and Ian Sommerville. “Comparing dynamic AO systems”. In: *Dynamic Aspects Workshop (DAW04)*. 2004.

[21] Mikolaj Cieslak et al. “Towards aspect-oriented functional–structural plant modelling”. In: *Annals of Botany* 108.6 (July 2011), pp. 1025–1041. DOI: 10.1093/aob/mcr121. URL: https://doi.org/10.1093%2Faob%2Fmcr121.

[22] Clarete and Github Contributors. *ForbiddenFruit*. https://web.archive.org/web/20210515092416/https://github.com/clarete/forbiddenfruit. 2021.

[23] R Lawrence Clark. “Linguistic Contribution To GOTO-Less Programming”. In: *Datamation* 19.12 (1973), pp. 62–63.

[24] Andy Clement, Adrian Colyer, and Mik Kersten. “Aspect-oriented programming with AJDT”. In: *ECOOP Workshop on Analysis of Aspect-Oriented Software*. Vol. 10. 2003.

[25] Dave Cli. “Minimal-intelligence agents for bargaining behaviors in market-based environments”. In: *Hewlett-Packard Labs Technical Reports* (1997).

[26] CodeHaus. *Nanning Aspects (Github Repo)*. https://github.com/codehaus/nanning. 2005.

[27] Jacob Cohen. *Statistical power analysis for the behavioral sciences*. Academic press, 2013, pp. 79–80.

[28] Constantinos Constantinides, Therapon Skotiniotis, and Maximilian Stoerzer. "AOP Considered Harmful". In: *In Proceedings of European Interactive Workshop on Aspects in Software (EIWAS)*. 2004.

[29] Jan De Houwer, Dermot Barnes-Holmes, and Agnes Moors. "What is learning? On the nature and merits of a functional definition of learning". In: *Psychonomic bulletin & review* 20 (2013), pp. 631–642.

[30] SimPy developers. *SimPy 4.0.1 documentation*. `https://simpy.readthedocs.io/en/4.0.1/`. 2021.

[31] Edsger W Dijkstra. "Letters to the editor: go to statement considered harmful". In: *Communications of the ACM* 11.3 (1968), pp. 147–148.

[32] Edsger W Dijkstra. "On the role of scientific thought". In: *Selected writings on computing: a personal perspective* (1982), pp. 60–66.

[33] Boudewijn van Dongen. "BPI Challenges: 10 years of real-life datasets". In: *IEEE Task Force on Process Mining Newsletter #2* (May 14, 2020). URL: `https://www.tf-pm.org/newsletter/newsletter-stream-2-05-2020/bpi-challenges-10-years-of-real-life-datasets` (visited on 05/14/2020).

[34] DOV DORI. "Object-process Analysis: Maintaining the Balance Between System Structure and Behaviour". In: *Journal of Logic and Computation* 5.2 (Apr. 1995), pp. 227–249. ISSN: 0955-792X. DOI: `10.1093/logcom/5.2.227`. eprint: `https://academic.oup.com/logcom/article-pdf/5/2/227/6244720/5-2-227.pdf`. URL: `https://doi.org/10.1093/logcom/5.2.227`.

[35] Adrian Dozsa, Tudor Girba, and Radu Marinescu. "How lisp systems look different". In: *2008 12th European Conference on Software Maintenance and Reengineering*. IEEE. 2008, pp. 223–232.

[36] Robert Dyer and Hridesh Rajan. "Supporting dynamic aspect-oriented features". In: *ACM Transactions on Software Engineering and Methodology* 20.2 (Aug. 2010), pp. 1–34. DOI: 10.1145/1824760.1824764. URL: https://doi.org/10.1145%2F1824760.1824764.

[37] Arpad E Elo and Sam Sloan. *The rating of chessplayers: past and present*. Bronx, NY: Ishi Press International, 1978. ISBN: 9780923891275.

[38] Miklós Espák. "Aspect-Oriented Programming On Lisp". In: *International Conference on Applied Informatics*. 2004.

[39] Robert E Filman, Daniel P Friedman, and Peter Norvig. "Aspect-oriented programming is quantification and obliviousness". In: *Workshop on Advanced Separation of Concerns, OOPSLA* (2000).

[40] Marc Fleury and Francisco Reverbel. "The JBoss extensible server". In: *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer. 2003, pp. 344–373.

[41] Eclipse Foundation. *Advice Precedence Documentation*. https://eclipse.dev/aspectj/doc/released/progguide/semantics-advice.html#advice-precedence. 2024.

[42] Eclipse Foundation. *Bytecode Weaving, Incremental Compilation, and Memory Usage Documentation*. https://eclipse.dev/aspectj/doc/released/devguide/bytecode-concepts.html. 2024.

[43] Eclipse Foundation. *Load-Time Weaving Documentation*. https://eclipse.dev/aspectj/doc/released/devguide/ltw-configuration.html. 2024.

[44] Python Software Foundation. *The Import System | The Module Cache*. https://docs.python.org/3.12/reference/import.html#the-module-cache. 2024.

[45] James France, John HM Thornley, et al. *Mathematical models in agriculture*. Butterworths, 1984.

[46] Daniel Friesel, Markus Buschhoff, and Olaf Spinczyk. “Annotations in Operating Systems with Custom AspectC++ Attributes”. In: *Proceedings of the 9th Workshop on Programming Languages and Operating Systems*. 2017, pp. 36–42.

[47] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0201633612.

[48] Wasif Gilani and Olaf Spinczyk. “A family of aspect dynamic weavers”. In: *Dynamic Aspects Workshop (DAW04)*. 2004.

[49] Ryan M Golbeck, Peter Selby, and Gregor Kiczales. “Late binding of AspectJ advice”. In: *Objects, Models, Components, Patterns: 48th International Conference, TOOLS 2010, Málaga, Spain, June 28–July 2, 2010. Proceedings 48*. Springer. 2010, pp. 173–191.

[50] Ryan M Golbeck et al. “Lightweight virtual machine support for AspectJ”. In: *Proceedings of the 7th international conference on Aspect-oriented software development*. 2008, pp. 180–190.

[51] Fabian Gomes et al. “SimKit: A High Performance Logical Process Simulation Class Library in C++”. In: *Proceedings of the 27th Conference on Winter Simulation*. WSC ’95. Arlington, Virginia, USA: IEEE Computer Society, 1995, pp. 706–713. ISBN: 0780330188. DOI: 10.1145/224401.224714. URL: https://doi.org/10.1145/224401.224714.

[52] B. Gompertz. *On the Nature of the Function Expressive of the Law of Human Mortality, and on a New Mode of Determining the Value of Life Contingencies.* Jan. 1815. DOI: 10.1098/rspl.1815.0271. URL: https://doi.org/10.1098/rspl.1815.0271.

[53] Stephen Jay Gould and Elisabeth S Vrba. “Exaptation—a missing term in the science of form”. In: *Paleobiology* 8.1 (1982), pp. 4–15.

[54] Leonard Green and Joel Myerson. “Exponential versus hyperbolic discounting of delayed outcomes: Risk and waiting time”. In: *American Zoologist* 36.4 (1996), pp. 496–505.

[55] László Gulyás and Tamás Kozsik. "The use of aspect-oriented programming in scientific simulations". In: *Proceedings of Sixth Fenno-Ugric Symposium on Software Technology, Estonia*. 1999.

[56] Allan Hackshaw and Amy Kirkwood. "Interpreting and reporting clinical trials with results of borderline significance". In: *BMJ* 343 (2011). ISSN: 0959-8138. DOI: 10.1136/bmj.d3340. eprint: https://www.bmj.com/content/343/bmj.d3340.full.pdf. URL: https://www.bmj.com/content/343/bmj.d3340.

[57] Ralph VL Hartley. "Transmission of information 1". In: *Bell System technical journal* 7.3 (1928), pp. 535–563.

[58] Andrew Heathcote, Scott D Brown, and Eric-Jan Wagenmakers. "An introduction to good practices in cognitive modeling". In: *An introduction to model-based cognitive neuroscience* (2015), pp. 25–48.

[59] James F Hemphill. "Interpreting the magnitudes of correlation coefficients." In: *American Psychologist* (2003). DOI: 10.1037/0003-066X.58.1.78. URL: %5Curl%7Bhttps://doi.org/10.1037/0003-066X.58.1.78%7D.

[60] David Hiebeler et al. "The swarm simulation system and individual-based modeling". In: *Decision Support-2001*. Santa Fe Institute Santa Fe, NM, USA. 1994.

[61] Erik Hilsdale et al. "AspectJ: The Language and Support Tools". In: *Addendum to the 2000 Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (Addendum)*. OOPSLA '00. Minneapolis, Minnesota, USA: Association for Computing Machinery, 2000, p. 163. ISBN: 1581133073. DOI: 10.1145/367845.368070. URL: https://doi.org/10.1145/367845.368070.

[62] Uwe D.C. Hohenstein and Michael C. Jäger. "Using Aspect-Orientation in Industrial Projects: Appreciated or Damned?" In: *Proceedings of the 8th ACM International Conference on Aspect-Oriented Software Development*. AOSD '09. Charlottesville, Virginia, USA: Association for Computing Machinery, 2009, pp. 213–222. ISBN: 9781605584423.

DOI: 10.1145/1509239.1509268. URL: https://doi.org/10.1145/1509239.1509268.

[63] Erik Hollnagel, David D Woods, and Nancy Leveson. *Resilience engineering: Concepts and precepts*. Ashgate Publishing, Ltd., 2006.

[64] Nigel Howard. *Paradoxes of rationality: theory of metagames and political behavior*. MIT Press, 1971. ISBN: 9780262582377.

[65] John Hunt. "Class Slots". In: *Advanced Guide to Python 3 Programming*. Cham: Springer International Publishing, 2023, pp. 15–21. ISBN: 978-3-031-40336-1. DOI: 10.1007/978-3-031-40336-1_3. URL: https://doi.org/10.1007/978-3-031-40336-1_3.

[66] Jon Espen Ingvaldsen and Jon Atle Gulla. "Preprocessing support for large scale process mining of SAP transactions". In: *International Conference on Business process management*. Springer. 2007, pp. 30–41.

[67] Tudor B Ionescu et al. "An Aspect-Oriented Approach for Disaster Prevention Simulation Workflows on Supercomputers, Clusters, and Grids". In: *2009 13th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications*. IEEE. 2009, pp. 21–30.

[68] Amin Jalali, Petia Wohed, and Chun Ouyang. "Aspect oriented business process modelling with precedence". In: *International Workshop on Business Process Modeling Notation*. Springer. 2012, pp. 23–37.

[69] Paul Johannesson and Erik Perjons. *An introduction to design science*. Vol. 10. Springer, 2014.

[70] Chris W Johnson and Christine Shea. "A comparison of the role of degraded modes of operation in the causes of accidents in rail and air traffic management". In: *2007 2nd Institution of Engineering and Technology International Conference on System Safety*. IET. 2007, pp. 89–94.

[71] William Kavanagh. “Using probabilistic model checking to balance games”. PhD thesis. University of Glasgow, 2021.

[72] William Kavanagh and Alice Miller. “Gameplay Analysis of Multiplayer Games with Verified Action-Costs”. In: *The Computer Games Journal* 10.1-4 (Dec. 2020), pp. 89–110. DOI: 10.1007/s40869-020-00121-5. URL: https://doi.org/10.1007%2Fs40869-020-00121-5.

[73] William Kavanagh and Alice Miller. “Gameplay analysis of multiplayer games with verified action-costs”. In: *The Computer Games Journal* 10 (2021), pp. 89–110.

[74] William Kavanagh, Tom Wallis, and Alice Miller. *RPGLite player data and lookup tables*. Released through University of Glasgow. 2020. DOI: 10.5525/gla.researchdata.1070. URL: %5Curl%7Bhttps://researchdata.gla.ac.uk/1070/%7D. *NB: The authors “Tom Wallis” and “William Wallis” are the same person.*

[75] William Kavanagh et al. “Balancing turn-based games with chained strategy generation”. In: *IEEE Transactions on Games* 13.2 (2019), pp. 113–122.

[76] Stephen Kell. “A survey of practical software adaptation techniques.” In: *J. Univers. Comput. Sci.* 14.13 (2008), pp. 2110–2157.

[77] Ralph Keller and Urs Hölzle. “Binary component adaptation”. In: *European Conference on Object-Oriented Programming*. Springer. 1998, pp. 307–329.

[78] M. G. Kendall. “A New Measure of Rank Correlation”. In: *Biometrika* 30.1/2 (1938), pp. 81–93. ISSN: 00063444. URL: http://www.jstor.org/stable/2332226 (visited on 12/16/2023).

[79] G. Kiczales et al. “An Overview of AspectJ”. In: *ECOOP*. 2001.

[80] Gregor Kiczales, Jim Des Rivieres, and Daniel G Bobrow. *The art of the metaobject protocol*. MIT press, 1991.

[81] Gregor Kiczales et al. “Aspect-oriented programming”. In: *European conference on object-oriented programming*. Springer. 1997, pp. 220–242.

[82] Scott Kirkpatrick, C Daniel Gelatt Jr, and Mario P Vecchi. “Optimization by simulated annealing”. In: *science* 220.4598 (1983), pp. 671–680.

[83] Alexander Knebe et al. “nIFTy cosmology: comparison of galaxy formation models”. In: *Monthly Notices of the Royal Astronomical Society* 451.4 (June 2015), pp. 4029–4059. ISSN: 0035-8711. DOI: 10.1093/mnras/stv1149. eprint: https://academic.oup.com/mnras/article-pdf/451/4/4029/3857619/stv1149.pdf. URL: https://doi.org/10.1093/mnras/stv1149.

[84] Athanasios Kokkinakis et al. “Metagaming and metagames in Esports”. In: *International Journal of Esports* (2021).

[85] Thomas S Kuhn. *The structure of scientific revolutions*. University of Chicago press, 2012.

[86] Sheldon J Lachman. “Learning is a process: Toward an improved definition of learning”. In: *The Journal of psychology* 131.5 (1997), pp. 477–480.

[87] Gary T Leavens and Curtis Clifton. “Multiple concerns in aspect-oriented language design: a language engineering approach to balancing benefits, with examples”. In: *Proceedings of the 5th workshop on Software engineering properties of languages and aspect technologies*. 2007, 6–es.

[88] Chen Li. “Mining process model variants: Challenges, techniques, examples”. PhD thesis. University of Twente, The Netherlands, 2010.

[89] Aristid Lindenmayer. “Mathematical models for cellular interactions in development I. Filaments with one-sided inputs”. In: *Journal of theoretical biology* 18.3 (1968), pp. 280–299.

[90] Russell Lock, Tim Storer, and Ian Sommerville. “Responsibility modelling for risk analysis”. In: *Reliability Engineering and System Safety* (2009).

[91] Giselle Machado, Vander Alves, and Rohit Gheyi. “Formal Specification and Verification of Well-formedness in Business Process Product Lines”. In: *6th Latin American*

Workshop no Aspect-Oriented Software Development: Advanced Modularization Techniques, LAWASP. Vol. 12. 2012.

[92] Idarlan Machado et al. "Managing variability in business processes". In: *Proceedings of the 2011 international workshop on Early aspects - EA '11*. ACM Press, 2011. DOI: 10.1145/1960502.1960508. URL: https://doi.org/10.1145%2F1960502.1960508.

[93] Spring Maintainers. *Aspect Oriented Programming with Spring Documentation*. https://docs.spring.io/spring-framework/docs/4.3.15.RELEASE/spring-framework-reference/html/aop.html. 2024.

[94] Stephen Paul Marsh. "Formalising trust as a computational concept". PhD thesis. University of Stirling, 1994.

[95] Robert Martin. "OO design quality metrics". In: *An analysis of dependencies* 12.1 (1994), pp. 151–170.

[96] Robert Cecil Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.

[97] Norm Matloff. "Introduction to discrete-event simulation and the simpy language". In: *Davis, CA. Dept of Computer Science. University of California at Davis. Retrieved on August* 2.2009 (2008), pp. 1–33.

[98] D.H. Meadows and Club of Rome. *The Limits to Growth: A Report for the Club of Rome's Project on the Predicament of Mankind*. Universe Books, 1972. ISBN: 9780876631652.

[99] Katharina Mehner and Awais Rashid. "Towards a standard interface for runtime inspection in AOP environments". In: *Workshop on Tools for Aspect-Oriented Software Development at OOPSLA*. Vol. 2. 2002.

[100] Richard Millham and Evans Dogbe. "Aspect-Oriented Security and Exception Handling within an Object Oriented System". In: *2011 IEEE 35th Annual Computer Soft-*

ware and Applications Conference Workshops. July 2011, pp. 321–326. DOI: 10.1109/COMPSACW.2011.60.

[101] Robin Milner. "Bigraphs and their algebra". In: *Electronic Notes in Theoretical Computer Science* 209 (2008), pp. 5–19.

[102] Robin Milner. *The space and motion of communicating agents*. Cambridge University Press, 2009.

[103] Alexey A. Mitsyuk et al. "Generating event logs for high-level process models". In: *Simulation Modelling Practice and Theory* 74 (May 2017), pp. 1–16. DOI: 10.1016/j.simpat.2017.01.003. URL: https://doi.org/10.1016%2Fj.simpat.2017.01.003.

[104] Arjuna Nebel et al. "Recalibration of limits to growth: An update of the World3 model". In: *Journal of Industrial Ecology* (2023).

[105] Angela Nicoara and Gustavo Alonso. "Dynamic AOP with PROSE." In: *CAiSE Workshops (2)*. Citeseer. 2005, pp. 125–138.

[106] Jurg Nievergelt. "Information Content of Chess Positions". In: *SIGART Bull.* 62 (Apr. 1977), pp. 13–15. ISSN: 0163-5719. DOI: 10.1145/1045398.1045400. URL: https://doi.org/10.1145/1045398.1045400.

[107] Aran O'Leary. "Simulating Human Behaviour in a Micro-epidemic using an Agent-based Model with Fuzzing Aspects". In: *Unpublished masters thesis* (2020).

[108] Object Management Group (OMG). *Business Process Model and Notation, Version 2.0*. OMG Document Number formal/2011-01-03 (https://www.omg.org/spec/BPMN/2.0). 2011.

[109] Object Management Group (OMG). *Unified Modelling Langauge, Version 2.5*. OMG Document Number formal/2015-03-01 (http://www.omg.org/spec/UML/2.5). 2015.

[110] D. L. Parnas. “On the criteria to be used in decomposing systems into modules”. In: *Communications of the ACM* 15.12 (Dec. 1972), pp. 1053–1058. DOI: 10.1145/361598.361623. URL: https://doi.org/10.1145%2F361598.361623.

[111] William Pasmore et al. “Reflections: sociotechnical systems design and organization change”. In: *Journal of Change Management* 19.2 (2019), pp. 67–85.

[112] Andrei Popovici, Gustavo Alonso, and Thomas Gross. “Just-in-Time Aspects: Efficient Dynamic Weaving for Java”. In: *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*. Association for Computing Machinery, 2003, pp. 100–109.

[113] Andrei Popovici, Thomas Gross, and Gustavo Alonso. “Dynamic Weaving for Aspect-Oriented Programming”. In: *Proceedings of the 1st International Conference on Aspect-Oriented Software Development*. AOSD ’02. Enschede, The Netherlands: Association for Computing Machinery, 2002, pp. 141–147. ISBN: 158113469X. DOI: 10.1145/508386.508404. URL: https://doi.org/10.1145/508386.508404.

[114] Karl Popper. “The Logic and Evolution of Scientific Theory”. In: *All Life is Problem Solving*. Trans. by Patrick Camiller. Routledge, 2013, pp. 3–22.

[115] Asef Pourmasoumi et al. “On Business Process Variants Generation.” In: *CAiSE Forum*. 2015, pp. 179–188.

[116] Adam Przybylek. “What is Wrong with AOP?” In: *ICSOFT (2)*. Citeseer. 2010, pp. 125–130.

[117] Adam Przybyłek. “An empirical study on the impact of AspectJ on software evolvability”. In: *Empirical Software Engineering* 23.4 (2018), pp. 2018–2050.

[118] RafeKettler and Github Contributors. *magicmethods (A guide to Python’s magic methods)*. https://github.com/RafeKettler/magicmethods/tree/65cc4a7bf4e72ba18df1ad17a377d879c9e7fe41. 2011 — 2016.

[119] Hridesh Rajan et al. *Nu: Towards an AspectOriented Invocation Mechanism*. Tech. rep. Technical Report 414, Iowa State University, Department of Computer Science, 2006.

[120] Payam Refaeilzadeh, Lei Tang, and Huan Liu. "Cross-Validation". In: *Encyclopedia of Database Systems*. Ed. by LING LIU and M. TAMER ÖZSU. Boston, MA: Springer US, 2009, pp. 532–538. ISBN: 978-0-387-39940-9. DOI: 10.1007/978-0-387-39940-9_565. URL: https://doi.org/10.1007/978-0-387-39940-9_565.

[121] F. J. Richards. "A Flexible Growth Function for Empirical Use". In: *Journal of Experimental Botany* 10.2 (June 1959), pp. 290–301. ISSN: 0022-0957. DOI: 10.1093/jxb/10.2.290. eprint: https://academic.oup.com/jxb/article-pdf/10/2/290/1411755/10-2-290.pdf. URL: https://doi.org/10.1093/jxb/10.2.290.

[122] Jarrett Rosenberg. "Some misconceptions about lines of code". In: *Proceedings fourth international software metrics symposium*. IEEE. 1997, pp. 137–142.

[123] Chris Schaefer et al. "Introducing Spring AOP". In: *Pro Spring: Fourth Edition* (2014), pp. 161–239.

[124] Kristan A Schneider et al. "The COVID-19 pandemic preparedness simulation tool: CovidSIM". In: *BMC infectious diseases* 20 (2020), pp. 1–11.

[125] Federico Sembolini et al. "nIFTy galaxy cluster simulations – II. Radiative models". In: *Monthly Notices of the Royal Astronomical Society* 459.3 (Apr. 2016), pp. 2973–2991. ISSN: 0035-8711. DOI: 10.1093/mnras/stw800. eprint: https://academic.oup.com/mnras/article-pdf/459/3/2973/8107345/stw800.pdf. URL: https://doi.org/10.1093/mnras/stw800.

[126] Simon Seow. "OBASHI White Paper". In: *APMG-International, High Wycombe, UK* (2011).

[127] Claude Elwood Shannon. "A mathematical theory of communication". In: *The Bell system technical journal* 27.3 (1948), pp. 379–423.

[128] Ivan Shugurov and Alexey A Mitsyuk. “Generation of a set of event logs with noise”. In: *Proceedings of the Spring/Summer Young Researchers’ Colloquium on Software Engineering*. 2014.

[129] Hercules Sant Ana da Silva Jose et al. “Implementation of Aspect-oriented Business Process Models with Web Services.” In: *Bus. Inf. Syst. Eng.* 62.6 (2020), pp. 561–584.

[130] C. Spearman. “The Proof and Measurement of Association between Two Things”. In: *The American Journal of Psychology* 15.1 (1904), pp. 72–101. ISSN: 00029556. URL: http://www.jstor.org/stable/1412159 (visited on 12/16/2023).

[131] Olaf Spinczyk and Daniel Lohmann. “The design and implementation of AspectC++”. In: *Knowledge-Based Systems* 20.7 (2007), pp. 636–651.

[132] Friedrich Steimann. “The paradoxical success of aspect-oriented programming”. In: *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications - OOPSLA ’06*. ACM Press, 2006. DOI: 10.1145/1167473.1167514. URL: https://doi.org/10.1145%2F1167473.1167514.

[133] Thomas Stocker and Rafael Accorsi. “Secsy: Security-aware synthesis of process event logs”. In: *Proceedings of the 5th International Workshop on Enterprise Modelling and Information Systems Architectures, St. Gallen, Switzerland*. Citeseer. 2013.

[134] Thomas Stocker and Rafael Accorsi. “SecSy: A Security-oriented Tool for Synthesizing Process Event Logs.” In: *BPM (Demos)*. 2014, p. 71.

[135] Tim Storer. *Theatre_AG*. https://web.archive.org/web/20230728153002/https://github.com/twsswt/theatre_ag. 2023. *NB: The authors “Tom Wallis” and “William Wallis” are the same person.*

[136] Tim Storer and William Wallis. *Fuzzi-Moss*. https://web.archive.org/web/20201018121800/https://github.com/twsswt/fuzzi-moss. 2016. *NB: The authors “Tom Wallis” and “William Wallis” are the same person.*

[137] Gregory T Sullivan. "Aspect-oriented programming using reflection and metaobject protocols". In: *Communications of the ACM* 44.10 (2001), pp. 95–97.

[138] Dai Hai Ton That et al. "Sciunits: Reusable Research Objects". In: *2017 IEEE 13th International Conference on e-Science (e-Science)*. IEEE, Oct. 2017. DOI: 10.1109/escience.2017.51. URL: https://doi.org/10.1109%2Fescience.2017.51.

[139] Eric Lansdown Trist and Kenneth W Bamforth. "Some social and psychological consequences of the longwall method of coal-getting: An examination of the psychological situation and defences of a work group in relation to the social structure and technological content of the work system". In: *Human relations* 4.1 (1951), pp. 3–38.

[140] Wil MP Van der Aalst and Anton JMM Weijters. "Process mining: a research agenda". In: *Computers in industry* 53.3 (2004), pp. 231–244.

[141] Boudewijn F Van Dongen et al. "The ProM framework: A new era in process mining tool support". In: *Applications and Theory of Petri Nets 2005: 26th International Conference, ICATPN 2005, Miami, USA, June 20-25, 2005. Proceedings 26*. Springer. 2005, pp. 444–454.

[142] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009. ISBN: 1441412697.

[143] Jussi Vanhatalo, Hagen Völzer, and Jana Koehler. "The refined process structure tree". In: *Data & Knowledge Engineering* 68.9 (2009), pp. 793–818.

[144] PF Verhulst. "La loi d'accroissement de la population". In: *Nouveaux Memories de l'Académie Royale des Sciences et Belles-Lettres de Bruxelles* 18 (1845), pp. 14–54.

[145] Ludwig Von Bertalanffy. "A quantitative theory of organic growth (inquiries on growth laws. II)". In: *Human biology* 10.2 (1938), pp. 181–213.

[146] Paul Wallis and Fergus Cloughley. *The OBASHI Methodology*. The Stationery Office, 2010.

[147] Tom Wallis. *PDSF3 source code*. https://github.com/RPGLite/analysis/blob/2f1bebfc643d59900bc98fc8dafad19804c8e506/tom_models/pdsf.py. 2023. *NB: The authors "Tom Wallis" and "William Wallis" are the same person.*

[148] Tom Wallis and William Kavanagh. *RPGLite Website*. https://rpglite.app. 2018. *NB: The authors "Tom Wallis" and "William Wallis" are the same person.*

[149] Tom Wallis and William Kavanagh. *RPGLite Analysis Repo (with aspect-oriented modelling source in the `tom_models` folder*. https://github.com/RPGLite/analysis/tree/2f1bebfc643d59900bc98fc8dafad19804c8e506. 2023. *NB: The authors "Tom Wallis" and "William Wallis" are the same person.*

[150] Tom Wallis and Tim Storer. "Modelling realistic user behaviour in information systems simulations as fuzzing aspects". In: *International Conference on Advanced Information Systems Engineering*. Springer. 2018, pp. 254–268. *NB: The authors "Tom Wallis" and "William Wallis" are the same person.*

[151] Tom Wallis and Tim Storer. "Process Fuzzing as an Approach to Genetic Programming". In: *Proceedings of the SICSA Workshop on Reasoning, Learning and Explainability*. Ed. by Kyle Martin, Nirmalie Wiratunga, and Leslie S. Smith. Vol. 2151. CEUR Workshop Proceedings. Aberdeen, UK: CEUR-WS.org, June 2018. URL: http://ceur-ws.org/Vol-2151/Paper_S3.pdf. *NB: The authors "Tom Wallis" and "William Wallis" are the same person.*

[152] Tom Wallis and Tim Storer. *PyDySoFu*. https://github.com/twsswt/pydysofu. 2018. *NB: The authors "Tom Wallis" and "William Wallis" are the same person.*

[153] Tom Wallis and Tim Storer. *ASP*. https://web.archive.org/web/20230730164214/https://github.com/probablytom/asp. 2023. *NB: The authors "Tom Wallis" and "William Wallis" are the same person.*

[154] William Wallis et al. "Designing a mobile game to generate player data - lessons learned". In: *CoRR* abs/2101.07144 (2021). arXiv: 2101.07144. URL: https://arx

iv.org/abs/2101.07144. *NB: The authors "Tom Wallis" and "William Wallis" are the same person.*

[155] AR Werker and KW Jaggard. "Modelling asymmetrical growth curves that rise and then fall: Applications to foliage dynamics of sugar beet (Beta vulgarisL.)" In: *Annals of Botany* 79.6 (1997), pp. 657–665.

[156] Edward Yourdon and Larry L Constantine. "Structured design. Fundamentals of a discipline of computer program and systems design". In: *Englewood Cliffs: Yourdon Press* (1979).

Appendix A

Details of Pattern Used when Designing RPGLite's Naive Model

This appendix contains additional details on the actor, context, and environment variables used as arguments to the functions implementing RPGLite's workflow steps. Each simulation step receives these three arguments at a minimum. Because steps of the model are functions, and therefore valid join points, aspects applied to these have access to the entire state of the simulation.

Actor — allows the function to identify the actor performing the activity defined by the function. This argument is any object uniquely identifying an actor. While RPGLite's naive model did not use this variable to store actor-specific data, the object identifying the actor could be used to encapsulate actor-specific information across different workflows or different invocations of the same workflow.

Context — allows a workflow step to determine details of the current thread of work being undertaken by the actor. This is necessary because in some simulations, the same actor might pause and resume multiple occurrences of the same activity — for example, they might concurrently play three different matches in RPGLite. As a result, it is necessary

to understand the context of the action being performed by the actor in question. This argument can be any object uniquely identifying the context of a piece of work, but should be mutable (such as a class or dictionary-like object) to permit the communication of information across invocations of different action-representing functions.

Environment — an actor's actions can be determined by the global environment they act within. There may be ancillary details to the actor's actions and the context of their particular thread of work which they are undertaken within which are used to determine behaviour, such as a landscape they traverse or other actors they might choose to interact with. An actor may also alter the state of that environment. Because all actors share access to a global environment, it can also function as an area of memory used for message passing or a space where actors can set values and flags to alter the environment of other actors.

This concept is related to environments in some other simulation frameworks, such as SimPy[30], where the environment controls scheduling and execution. However, other frameworks' environments impose model properties such as a model of time which may not be of interest to a researcher; this structure imposes no such constraints and its flexibility offers a general-case design pattern rather than the structure of a pre-existing framework, as it requires nothing more than a mutable function argument common throughout a model's implementation.

Each simulation step receives these three arguments at a minimum. Because steps of the model are functions, and therefore valid join points, aspects applied to these have access to the entire state of the simulation. This happens to be a useful property for aspect-oriented simulation & modelling, though not a necessary one.

Appendix B

Complete Description of Model Parameters

The `ModelParameters` class defines the parameters which generate data from an experiment. It is introduced in Section 6.6.1, and its implementation is shown in Fig. 6.10 and Fig. 6.11. However, many parameters control the

starting confidence assumed confidence plateau The initial value of the model's confidence curve. A corresponding high value for the model's confidence curve. As the curve's growth rate is calculated to align with the number of games completed by the real-world player being simulated, and the curve asymptotically approaches 1, a high value is required which represents the expected confidence of a real-world player having played a significant number of games.

curve inflection relative to numgames The proportion of games played by the real-world player being simulated at which the player's confidence curve should reach the parameter `assumed_confider` . This allows for control over the growth rate of the curve while linking the rate of growth to the number of games played by the real-world player being simulated;

C The curve parameter of the confidence model's underlying birch curve;

prob bored The probability that a player with confidence above `assumed_confidence_plateau` becomes "bored" and stops playing RPGLite. These players are removed from the simulated playerbase and replaced with new players. This mechanism is discussed in further detail in Section 6.6.4.

boredom enabled Whether the boredom mechanism discussed in Section 6.6.4 is to be enabled.

training data The set of real-world games used as a training fold when optimising parameters for simulating a player. This is used when annealing toward optimal parameters for the learning model, and so is explained in more detail in Section 7.1.

testing data The set of real-world games used as a testing fold when optimising parameters for simulating a player. This is used when annealing toward optimal parameters for the learning model, and so is explained in more detail in Section 7.1.

iteration base The number of games to play when generating a synthetic dataset. If `boredom_enabled==False`, the number of games is instead calculated to ensure that all synthetic players play the same number of games as were completed by the played being simulated, (calculated from the training or testing fold as appropriate).

number The size of the simulated playerbase.

simulated players advice A list of tuples defining advice to weave when generating data. Pieces of advice are uniquely defined by a tuple containing the type of aspect to weave (such as `"prelude"` or `"error_handler"`), a string-represented regular expression defining the join point of the advice, and a `callable` to use as an aspect when weaving the advice. As `ModelParameters` instances are serialised to disk to persist experiment results and Python functions are not

seralisable, this may also be a string value; strings are IDs of aspects, and are replaced with their corresponding aspect on deserialisation.

players The player being simulated. This parameter is a list of player usernames to support simulating groups of players if required, though this functionality is not used in the experiments presented in Chapter 7.

args Any additional arguments to pass when running a simulation.

kwargs Any additional keyword arguments to pass when running a simulation.

boredom period The number of games completed by the simulated playerbase between checks of players' confidence and random selection for removal (default 25).

initial exploration A value which controls the number of games individual players complete before their confidence & learning models control their character pair selection (default 28, to allow sufficient games to select every available pair exactly once).

Appendix C

Search for Significant Results

This algorithm searches for statistically significant results within p-value and τ correlation coefficient thresholds across a majority of folds from k-fold validation, as described in Section 7.1.

```
def analyse_result_file(filepath:str, season:int=1, viability_budget:int=1) -> (str, list[Result], float, float):
    '''
    Analyses results in a file at `filepath` and …returns:
        - username
        - all params that are viable for all folds & against the total dataset
        - and the pval threshold at which we find those params
        - the stat threshold at which we find those params.
    viability_budget is the number of pval + stat combinations which yield significant results to return. In other words, when the budget is 1, The first pval + stat combo yielding statistically significant results will be the only one for which results will be returned; if the budget is 2, a second pair will be found before returning; if 3, another still; and so on. A budget of -1 indicates that all results should be returned.
    '''
    # Unpickle results and pop them into a dict, where keys are players and values are relevant results.
    all_results = list()
    with open(filepath, 'rb') as result_file:
        try:
            all_results = pickle.load(result_file)
        except EOFError:
            print(f"could not read [possibly empty?] result in file {filepath}")
            all_results = []
        except Exception as e:
            print("ERR!")
            print(e)
            raise e
```

```
22 …
23     pvals = [0.01, 0.02, 0.035, 0.05]
24     stats = [0.5, 0.4, 0.3, 0.2]
25     pval_stat_index_map = [(pval_index, stat_index) for pval_index in range(len(
           pvals)) for stat_index in range(len(stats))]
26     sorted_pval_stat_indices = sorted(pval_stat_index_map, key=sum)
27     search_param_combos = list(map(lambda indices: (pvals[indices[0]], stats[
           indices[1]]), sorted_pval_stat_indices))
28 
29     username = filepath.split('/')[-1].split('-')[0]
30 
31     all_games = None
32 
33     for pval, stat in search_param_combos:
34         print(username, pval, stat)
35         within_threshold = list()
36         combinations_seen = set()
37         result = None
38         globally_viable_params = list()
39         params_to_mutate = None
40         for fold in all_results:
41             within_threshold_for_fold = list()
42             for result in fold:
43                 if all_games is None:
44                     all_games = result.params.training_data + result.params.
                           testing_data
45                 if params_to_mutate is None:
46                     params_to_mutate = copy(result.params)
47                 if result.within_acceptable_bounds(pval, stat):
48                     within_threshold_for_fold.append((result.params.c, result.
                           params.curve_inflection_relative_to_numgames, result.
                           params.prob_bored))
49                     combinations_seen.add((result.params.c, result.params.
                           curve_inflection_relative_to_numgames, result.params.
                           prob_bored))
50             within_threshold.append(within_threshold_for_fold)
51 
52         commonality = dict() # c, rgr_coeff combo mapped to how many folds they
               appeared above threshold
53         for combination in combinations_seen:
54             commonality[combination] = reduce(lambda acc, fold_res: acc + (1 if
                   combination in fold_res else 0), within_threshold, 0)
55         ranked_params_for_player = sorted(commonality.items(), key=lambda x: x
               [1], reverse=True)
56         passing_params_for_player = filter(lambda x: x[1]>=3,
               ranked_params_for_player)
```

```
57 …
58         # Find a param set which passes more than it fails on testing folds, and
               also passes on the dataset as a whole.
59         globally_viable_params = list()
60         for possible_viable_param_set, _ in passing_params_for_player:
61             params_to_mutate.c = possible_viable_param_set[0]
62             params_to_mutate.curve_inflection_relative_to_numgames =
                   possible_viable_param_set[1]
63             params_to_mutate.prob_bored = possible_viable_param_set[2]
64             params_to_mutate.boredom_enabled = params_to_mutate.prob_bored !=
                   0.0001
65             params_to_mutate.testing_data, params_to_mutate.training_data =
                   all_games, all_games
66             test_against_all_player_games = params_to_mutate.run_experiment(
                   testing=True, correlation_metric=kendalltau, season=season)
67             if test_against_all_player_games.within_acceptable_bounds(pval, stat)
                   :
68                 globally_viable_params.append(test_against_all_player_games)
69
70         if len(globally_viable_params) > 0:
71             viability_budget = viability_budget - 1
72             if viability_budget == 0:
73                 break
74
75     return username, globally_viable_params, pval, stat
```

Appendix D

Complete Result Table answering RQ1

Table D.1: Correlation of all folds of real-world character pair selection and those generated by an unmodified naive model

Username	*fold #*	*p-value*	τ *statistic*
apropos0	1	0.9295560829453566	-0.013223592098145723
apropos0	2	0.543551087147816	-0.09089184624256738
apropos0	3	0.24967847077719418	0.1740513323160178
apropos0	4	0.9137449869231443	0.01604642463551945
apropos0	5	0.9647628029421551	0.006602422212782976
basta	1	0.1816194374918113	0.20462566640533308
basta	2	0.18248946612827355	0.2059628159428695
basta	3	0.07612323582082255	-0.272855015888954
basta	4	0.8577156891579766	-0.027709044747234814
basta	5	0.8661911974793254	0.026015295118650823
creilly1	1	0.35240946540386875	0.1408415780640769
creilly1	2	1.0	0.0
creilly1	3	0.12963600753361082	0.2286190426597633

Table D.1: *Continued on next page*

Username	*fold #*	*p-value*	τ *statistic*
creilly1	4	0.8772351814797427	-0.022972314182022596
creilly1	5	0.23450470832759351	0.17684896380472018
creilly2	1	0.7644615028216746	-0.04586638580669498
creilly2	2	0.3738292876597282	-0.13638531407402904
creilly2	3	0.17362238220159643	-0.21433767861360697
creilly2	4	0.328310729802638	0.14811336858631574
creilly2	5	0.9652366286655247	-0.0065855064974466625
cwallis	1	0.0415013132764738	0.3093048452693846
cwallis	2	0.3097043535187982	-0.15154965050057037
cwallis	3	0.9156021858411141	-0.01584867655605092
cwallis	4	0.9659578395732228	0.0063394706224203685
cwallis	5	0.689920709902269	0.06039860389867787
Deanerbeck	1	0.881413966884111	-0.023469523137789664
Deanerbeck	2	0.004650160066661944	-0.4401133171167231
Deanerbeck	3	0.9228899037702708	-0.01512874020122304
Deanerbeck	4	0.7933653799467553	0.04143965365710999
Deanerbeck	5	0.617370080862484	-0.07781714997961811
ECDr	1	0.36980925964418543	0.13489559954637306
ECDr	2	0.3439505882395243	-0.1413306545515153
ECDr	3	0.4344070895785651	0.11690221930822196
ECDr	4	0.5123199775611744	0.09847971421781773
ECDr	5	0.38565298773701395	0.12963938295936053
elennon	1	0.6831590303385897	0.06225988751474009
elennon	2	0.8578928320061399	-0.02749806661015982
elennon	3	0.7094791911120704	-0.056959703582473833

Table D.1: *Continued on next page*

Username	*fold #*	*p-value*	*τ statistic*
elennon	4	0.8770639630397704	0.023931519038060085
elennon	5	0.31841443329320684	-0.15169528069375557
Ellen	1	0.4173609577782533	-0.11952108728582653
Ellen	2	0.5641844970478596	0.08530901495354998
Ellen	3	0.13921559643746703	0.21826031618958391
Ellen	4	0.1642969752311254	0.20782616765906128
Ellen	5	0.5442947581369362	0.08956801917846717
Etess	1	0.291201011176352	0.16483767372268998
Etess	2	0.6942338351677287	-0.06105840084617214
Etess	3	0.431484384718381	0.1222120131676966
Etess	4	0.046612898050594605	-0.3075889702362141
Etess	5	0.5323480476649404	0.09594614036147774
Fbomb	1	0.16921401929250612	-0.2198877878822063
Fbomb	2	0.5803180551036174	0.08720827961883659
Fbomb	3	0.34901258267821544	0.14955808305450605
Fbomb	4	0.1405016420954379	0.23354024746948013
Fbomb	5	0.5525244548993048	0.09469610989712465
Frp97	1	0.45156575491741413	0.11887614636188644
Frp97	2	0.9523646984875704	-0.0094262053632344
Frp97	3	0.3865494332628323	0.13478934738386475
Frp97	4	0.9778079547585447	0.004378139362159142
Frp97	5	0.7253387704489176	-0.05492984459933149
georgedo	1	0.1851060401540079	0.20487048828935067
georgedo	2	0.07564136825095882	-0.27471958553261244
georgedo	3	0.8390968137012565	0.03170390147558708

Table D.1: *Continued on next page*

Username	*fold #*	*p-value*	τ *statistic*
georgedo	4	0.1203247115520158	0.24061753448870887
georgedo	5	0.12420626686270571	0.23473636763429323
Jamie	1	0.9443745624098961	-0.010563118354805768
Jamie	2	0.4847370503668401	-0.10644772056493343
Jamie	3	0.28413758436715064	0.16281579310997882
Jamie	4	0.5015708584486456	-0.10205393726282892
Jamie	5	0.8416509694265051	-0.030619219822267146
kubajj	1	0.7049453722291028	-0.05773003318743752
kubajj	2	0.7832851410075442	-0.04233212173460013
kubajj	3	0.7801854145662819	-0.042367134344875504
kubajj	4	0.4541561566755933	-0.11381205114579031
kubajj	5	0.23964707182973377	0.18161750637060775
l17r	1	0.6463837866590947	-0.07301576789300605
l17r	2	0.2801163044252959	-0.17047719783200246
l17r	3	0.5917784227616225	-0.08651471844197936
l17r	4	0.4350024282886019	0.12319642336105148
l17r	5	0.1664772014231095	0.22197284567390435
Nari	1	0.3306282216863503	-0.152456608934384
Nari	2	0.039896701204242956	-0.320121430661928
Nari	3	0.27984406128855255	0.17117066726234803
Nari	4	0.713223902979671	-0.05751633284923023
Nari	5	0.8815679241977081	-0.023186944788008413
Paddy	1	0.294674479189036	0.16598367734530717
Paddy	2	0.47885099520247654	0.10998533626601498
Paddy	3	0.6973750999146054	0.06081670409998015

Table D.1: *Continued on next page*

Username	*fold #*	*p-value*	*τ statistic*
Paddy	4	0.3988438321963125	0.13169331414131036
Paddy	5	0.7425619922201162	0.05170534143767003
sstein	1	0.5291782404289969	0.10103980549567479
sstein	2	0.34193365348492155	0.15131899415399777
sstein	3	0.6466648998216489	0.0725193938999425
sstein	4	0.45762296710401484	0.11897387338047426
sstein	5	0.23777258107163413	-0.1888651688872934
tanini	1	0.21382336458709483	0.18518069231484682
tanini	2	0.3664479227452031	-0.13182000244322564
tanini	3	0.9639432667791659	0.006790711498249916
tanini	4	0.7371912487644987	-0.049667410648660835
tanini	5	0.9827301253433335	-0.003205688090246685
timri	1	0.16090126850038755	-0.21973655363972666
timri	2	0.8241999300915683	-0.03470528041969024
timri	3	0.4215584983269117	-0.1290629353181705
timri	4	0.5958958837249172	0.08444944900452496
timri	5	0.6979115214262599	-0.06118748109105733

Table D.2: Correlation of all folds of real-world datasets of character pair selection and those generated by the naive model with advice woven to bias the characters chosen

Username	*fold #*	*p-value*	τ *statistic*
apropos0	1	$6.069\,589\,430\,124\,279 \times 10^{-10}$	0.9642436861538818
apropos0	2	$8.079\,029\,120\,240\,886 \times 10^{-10}$	0.9698244380089833
apropos0	3	$1.100\,193\,426\,691\,723\,7 \times 10^{-9}$	0.982403317924857
apropos0	4	$7.194\,122\,983\,312\,375 \times 10^{-10}$	0.9883036912035246
apropos0	5	$1.659\,168\,211\,024\,309\,6 \times 10^{-9}$	0.9854310631217637
basta	1	$6.984\,416\,886\,646\,488 \times 10^{-9}$	0.9751373529625749
basta	2	$1.254\,321\,413\,956\,981\,1 \times 10^{-8}$	0.9534625892455924
basta	3	$1.219\,848\,016\,158\,136 \times 10^{-8}$	0.9637888196533972
basta	4	$7.626\,421\,141\,995\,402 \times 10^{-9}$	0.9523532664857335
basta	5	$1.129\,950\,973\,298\,532\,2 \times 10^{-8}$	0.9480678693136386
creilly1	1	$6.984\,424\,740\,718\,803 \times 10^{-10}$	0.9609023536933049
creilly1	2	$8.088\,739\,467\,118\,352 \times 10^{-10}$	0.9681333740581184
creilly1	3	$1.292\,242\,772\,833\,327\,3 \times 10^{-9}$	0.9702535449004767
creilly1	4	$1.217\,149\,603\,922\,523\,7 \times 10^{-9}$	0.9804286358922405
creilly1	5	$1.292\,242\,772\,833\,327\,3 \times 10^{-9}$	0.9702535449004767
creilly2	1	$1.154\,297\,389\,956\,216\,1 \times 10^{-8}$	0.9839131599805843
creilly2	2	$3.334\,603\,473\,776\,998\,5 \times 10^{-9}$	0.9865162369237952
creilly2	3	$2.243\,342\,049\,839\,241 \times 10^{-8}$	0.9630868246861536
creilly2	4	$6.027\,332\,602\,847\,420\,5 \times 10^{-9}$	0.9752492558885195
creilly2	5	$3.929\,678\,299\,475\,491 \times 10^{-9}$	0.9820613241770825
cwallis	1	$2.513\,900\,528\,386\,046\,4 \times 10^{-9}$	0.9704949588309458
cwallis	2	$1.976\,414\,853\,123\,879\,5 \times 10^{-9}$	0.9371259937730507
cwallis	3	$1.558\,093\,273\,009\,808\,4 \times 10^{-9}$	0.9683640522700839

Table D.2: *Continued on next page*

Username	*fold #*	*p-value*	τ *statistic*
cwallis	4	$1.865\,113\,605\,022\,339\,5 \times 10^{-9}$	0.9744805811819829
cwallis	5	$2.503\,434\,431\,199\,603 \times 10^{-9}$	0.9625334218796218
Deanerbeck	1	$4.741\,611\,516\,573\,964 \times 10^{-8}$	0.9793792286287206
Deanerbeck	2	$5.016\,702\,308\,111\,833 \times 10^{-8}$	0.9879271228182775
Deanerbeck	3	$2.323\,229\,622\,171\,393 \times 10^{-8}$	0.9819805060619656
Deanerbeck	4	$3.117\,067\,728\,180\,191\,4 \times 10^{-8}$	0.9959919678390986
Deanerbeck	5	$5.016\,702\,308\,111\,833 \times 10^{-8}$	0.9879271228182775
ECDr	1	$8.455\,080\,743\,459\,683 \times 10^{-10}$	0.9591663046625438
ECDr	2	$4.736\,270\,844\,907\,838 \times 10^{-9}$	0.9051392757105036
ECDr	3	$2.914\,095\,334\,565\,709\,6 \times 10^{-9}$	0.9302605094190632
ECDr	4	$1.172\,069\,372\,010\,927\,7 \times 10^{-9}$	0.9539392014169457
ECDr	5	$3.109\,857\,683\,466\,065\,3 \times 10^{-9}$	0.956948752938691
elennon	1	$3.963\,369\,503\,544\,85 \times 10^{-9}$	0.9727697526833043
elennon	2	$1.047\,076\,476\,119\,670\,5 \times 10^{-8}$	0.966091783079296
elennon	3	$1.356\,940\,893\,104\,937\,3 \times 10^{-8}$	0.933447121915197
elennon	4	$3.890\,138\,222\,908\,016 \times 10^{-9}$	0.9727697526833043
elennon	5	$7.626\,421\,141\,995\,402 \times 10^{-9}$	0.9523532664857335
Ellen	1	$2.537\,912\,953\,036\,465\,6 \times 10^{-9}$	0.9499679070317291
Ellen	2	$2.969\,396\,487\,692\,385\,4 \times 10^{-10}$	0.982423938239031
Ellen	3	$8.169\,101\,159\,404\,917 \times 10^{-10}$	0.9863408219546951
Ellen	4	$9.452\,655\,162\,863\,045 \times 10^{-10}$	0.9843740386976972
Ellen	5	$4.678\,753\,424\,418\,981 \times 10^{-10}$	0.9491726884870106
Etess	1	$1.112\,969\,177\,936\,710\,5 \times 10^{-8}$	0.9940297973880048
Etess	2	$2.439\,595\,382\,030\,009\,8 \times 10^{-8}$	0.9999999999999999
Etess	3	$1.375\,777\,098\,018\,077\,5 \times 10^{-8}$	0.9812063786211384

Table D.2: *Continued on next page*

Username	*fold #*	*p-value*	τ *statistic*
Etess	4	$2.582\,210\,122\,313\,199 \times 10^{-8}$	0.989743318610787
Etess	5	$9.369\,742\,817\,562\,114 \times 10^{-9}$	0.9780192938436515
Fbomb	1	$3.117\,067\,728\,180\,191\,4 \times 10^{-8}$	0.9959919678390986
Fbomb	2	$7.316\,467\,015\,672\,806 \times 10^{-8}$	0.9999999999999998
Fbomb	3	$1.231\,558\,254\,354\,619 \times 10^{-7}$	1.0
Fbomb	4	$5.702\,069\,557\,398\,809\,5 \times 10^{-8}$	0.9950859653474028
Fbomb	5	$5.702\,069\,557\,398\,809\,5 \times 10^{-8}$	0.9950859653474028
Frp97	1	$2.439\,595\,382\,030\,009\,8 \times 10^{-8}$	0.9999999999999999
Frp97	2	$3.976\,207\,311\,697\,647\,7 \times 10^{-8}$	0.9919677414109794
Frp97	3	$2.582\,210\,122\,313\,199 \times 10^{-8}$	0.989743318610787
Frp97	4	$4.741\,611\,516\,573\,964 \times 10^{-8}$	0.9793792286287206
Frp97	5	$4.741\,611\,516\,573\,964 \times 10^{-8}$	0.9793792286287206
georgedo	1	$4.718\,509\,821\,360\,782 \times 10^{-8}$	0.9697815168769666
georgedo	2	$7.629\,394\,231\,426\,926 \times 10^{-9}$	0.9620913858416693
georgedo	3	$9.369\,742\,817\,562\,114 \times 10^{-9}$	0.9780192938436515
georgedo	4	$1.112\,969\,177\,936\,710\,5 \times 10^{-8}$	0.9940297973880048
georgedo	5	$1.254\,321\,413\,956\,981\,1 \times 10^{-8}$	0.9534625892455924
Jamie	1	$5.759\,794\,374\,505\,349\,5 \times 10^{-9}$	0.9854006762498304
Jamie	2	$1.047\,076\,476\,119\,670\,5 \times 10^{-8}$	0.966091783079296
Jamie	3	$1.185\,709\,491\,670\,605\,4 \times 10^{-8}$	0.9441784091034143
Jamie	4	$3.360\,116\,517\,000\,501\,6 \times 10^{-9}$	0.9843215373488934
Jamie	5	$4.219\,204\,633\,189\,228 \times 10^{-9}$	0.9902910322235658
kubajj	1	$5.727\,633\,416\,979\,75 \times 10^{-9}$	0.9663768920667863
kubajj	2	$6.793\,717\,497\,068\,974\,4 \times 10^{-9}$	0.9829463743659808
kubajj	3	$9.888\,967\,082\,391\,682 \times 10^{-9}$	0.986612515286092

Table D.2: *Continued on next page*

Username	*fold #*	*p-value*	τ *statistic*
kubajj	4	$2.755\,529\,578\,835\,524\,5 \times 10^{-8}$	0.9498537899183339
kubajj	5	$4.204\,686\,852\,490\,938 \times 10^{-9}$	0.9531132715605782
l17r	1	$1.055\,971\,659\,135\,280\,6 \times 10^{-7}$	0.99356906512808
l17r	2	$9.899\,858\,585\,023\,514 \times 10^{-8}$	0.9900495037128094
l17r	3	$5.702\,069\,557\,398\,809\,5 \times 10^{-8}$	0.9950859653474028
l17r	4	$5.702\,069\,557\,398\,809\,5 \times 10^{-8}$	0.9950859653474028
l17r	5	$4.265\,109\,899\,393\,854\,4 \times 10^{-8}$	1.0
Nari	1	$1.964\,917\,496\,486\,691\,2 \times 10^{-8}$	0.9850066473065526
Nari	2	$4.265\,109\,899\,393\,854\,4 \times 10^{-8}$	1.0
Nari	3	$1.698\,564\,245\,713\,669\,6 \times 10^{-8}$	0.99659283506935
Nari	4	$3.117\,067\,728\,180\,191\,4 \times 10^{-8}$	0.9959919678390986
Nari	5	$4.265\,109\,899\,393\,854\,4 \times 10^{-8}$	1.0
Paddy	1	$1.170\,929\,917\,772\,374\,8 \times 10^{-8}$	0.9839131599805843
Paddy	2	$2.323\,229\,622\,171\,393 \times 10^{-8}$	0.9819805060619656
Paddy	3	$1.698\,564\,245\,713\,669\,6 \times 10^{-8}$	0.99659283506935
Paddy	4	$5.016\,702\,308\,111\,833 \times 10^{-8}$	0.9879271228182775
Paddy	5	$1.375\,777\,098\,018\,077\,5 \times 10^{-8}$	0.9812063786211384
sstein	1	$5.016\,702\,308\,111\,833 \times 10^{-8}$	0.9879271228182775
sstein	2	$1.001\,055\,644\,467\,893\,8 \times 10^{-7}$	0.985184366143778
sstein	3	$1.231\,558\,254\,354\,619 \times 10^{-7}$	1.0
sstein	4	$1.034\,780\,942\,526\,623 \times 10^{-7}$	0.9653873777482519
sstein	5	$7.316\,467\,015\,672\,806 \times 10^{-8}$	0.9999999999999998
tanini	1	$1.538\,655\,754\,928\,303\,2 \times 10^{-9}$	0.9518614754555286
tanini	2	$7.194\,122\,983\,312\,375 \times 10^{-10}$	0.9883036912035246
tanini	3	$8.210\,940\,108\,367\,392 \times 10^{-10}$	0.9882579099091766

Table D.2: *Continued on next page*

Username	*fold #*	*p-value*	τ *statistic*
tanini	4	$1.100\,193\,426\,691\,723\,7 \times 10^{-9}$	0.982403317924857
tanini	5	$2.757\,229\,316\,465\,029\,4 \times 10^{-10}$	0.9746794344808962
timri	1	$2.582\,210\,122\,313\,199 \times 10^{-8}$	0.989743318610787
timri	2	$5.702\,069\,557\,398\,809\,5 \times 10^{-8}$	0.9950859653474028
timri	3	$2.582\,210\,122\,313\,199 \times 10^{-8}$	0.989743318610787
timri	4	$5.702\,069\,557\,398\,809\,5 \times 10^{-8}$	0.9950859653474028
timri	5	$3.117\,067\,728\,180\,191\,4 \times 10^{-8}$	0.9959919678390986

Appendix E

Complete Result Table answering RQ2

Table E.1: Untruncated data from the experiment in Section 7.3, generated using the aspect-oriented model of learning and applied to season 1.

Username	*p-value* <	τ >	*Confidence curve value*	*Confidence RGR modifier*	*Prob. bored*	*# folds*
apropos0	0.01	0.4	0.2	3	0.0625	3
apropos0	0.01	0.4	10	0.1	*disabled*	3
basta	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*
creilly1	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*
creilly2	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*
cwallis	0.01	0.4	0.02	0.1	0.25	4
cwallis	0.01	0.4	0.1	0.3	0.25	4
cwallis	0.01	0.4	1	0.1	0.25	4
cwallis	0.01	0.4	10	0.1	0.25	4
cwallis	0.01	0.4	0.02	0.1	0.0625	3
cwallis	0.01	0.4	0.02	0.3	0.25	3
cwallis	0.01	0.4	0.02	1	0.015625	3

Table E.1: *Continued on next page*

Username	*p-value <*	*τ >*	*Confidence curve value*	*Confidence RGR modifier*	*Prob. bored*	*# folds*
cwallis	0.01	0.4	0.02	1	0.0625	3
cwallis	0.01	0.4	0.02	1	1	3
cwallis	0.01	0.4	0.02	3	0.015625	3
cwallis	0.01	0.4	0.02	3	0.25	3
cwallis	0.01	0.4	0.1	0.1	*disabled*	3
cwallis	0.01	0.4	0.1	0.1	0.0625	3
cwallis	0.01	0.4	0.1	0.3	0.0625	3
cwallis	0.01	0.4	0.1	1	1	3
cwallis	0.01	0.4	0.2	3	0.25	3
cwallis	0.01	0.4	0.2	1	*disabled*	3
cwallis	0.01	0.4	0.2	1	1	3
cwallis	0.01	0.4	1	0.3	0.25	3
cwallis	0.01	0.4	1	3	*disabled*	3
cwallis	0.01	0.4	1	3	0.25	3
cwallis	0.01	0.4	5	3	0.015625	3
cwallis	0.01	0.4	5	3	0.0625	3
cwallis	0.01	0.4	10	1	0.0625	3
cwallis	0.01	0.4	10	1	0.25	3
cwallis	0.01	0.4	50	0.3	0.25	3
Deanerbeck	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*
ECDr	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*
elennon	0.035	0.3	50	0.1	0.0625	3
Ellen	0.01	0.3	10	0.3	0.015625	3
Etess	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*

Table E.1: *Continued on next page*

Username	*p-value <*	*τ >*	*Confidence curve value*	*Confidence RGR modifier*	*Prob. bored*	*# folds*
Fbomb	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*
Frp97	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*
georgedo	0.02	0.3	50	3	0.015625	3
georgedo	0.02	0.3	0.1	3	0.015625	3
georgedo	0.02	0.3	0.2	3	0.0625	3
georgedo	0.02	0.3	0.2	1	0.0625	3
georgedo	0.02	0.3	0.02	1	0.0625	3
l17r	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*
Jamie	0.01	0.4	0.2	1	1	3
Jamie	0.01	0.4	1	3	0.25	3
Jamie	0.01	0.4	5	0.1	0.25	3
Jamie	0.01	0.4	0.02	1	0.015625	3
kubajj	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*
Nari	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*
Paddy	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*
sstein	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*
tanini	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*
timri	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*

Appendix F

Complete Result Table for RQ3 using Prior Distribution Model

Table F.1: Untruncated data from the experiment in Section 7.4 which applies the prior distribution model of character pair selection to the model of RPGLite season 2.

Username	*Fold #*	*p-value*	τ *statistic*
aaaa	1	$7.629\,394\,231\,426\,926 \times 10^{-9}$	0.9620913858416693
aaaa	2	$5.291\,130\,983\,031\,646\,7 \times 10^{-8}$	0.9479191551169758
aaaa	3	$7.626\,421\,141\,995\,402 \times 10^{-9}$	0.9523532664857335
aaaa	4	$7.071\,834\,710\,138\,214 \times 10^{-8}$	0.8842393310811687
aaaa	5	$2.423\,076\,414\,547\,952\,5 \times 10^{-8}$	0.9125528002479737
apropos0	1	$4.943\,866\,078\,073\,06 \times 10^{-9}$	0.9685125342119326
apropos0	2	$2.362\,740\,616\,794\,553 \times 10^{-9}$	0.9607689228305228
apropos0	3	$3.360\,116\,517\,000\,501\,6 \times 10^{-9}$	0.9843215373488934
apropos0	4	$1.646\,556\,894\,454\,968 \times 10^{-9}$	0.9874734094849511
apropos0	5	$2.909\,943\,990\,006\,779 \times 10^{-9}$	0.9865765724632495
basta	1	$4.244\,305\,440\,694\,443 \times 10^{-9}$	0.9342963657194947

Table F.1: *Continued on next page*

Username	*Fold #*	*p-value*	τ *statistic*
basta	2	$2.018\,076\,324\,638\,670\,7 \times 10^{-9}$	0.9645739631583109
basta	3	$1.947\,707\,312\,746\,609\,3 \times 10^{-9}$	0.9645739631583109
basta	4	$2.693\,168\,060\,799\,769\,3 \times 10^{-9}$	0.9703784010607079
basta	5	$2.795\,605\,711\,599\,785 \times 10^{-9}$	0.9499679070317291
Becccca	1	$2.711\,275\,692\,905\,026 \times 10^{-9}$	0.9588607403653148
Becccca	2	$5.797\,919\,688\,404\,042 \times 10^{-9}$	0.9564144928693535
Becccca	3	$9.266\,862\,632\,910\,002 \times 10^{-10}$	0.9663321941277055
Becccca	4	$9.054\,062\,499\,720\,839 \times 10^{-10}$	0.9574271077563379
Becccca	5	$5.289\,446\,640\,726\,799 \times 10^{-9}$	0.9120173797327435
creilly1	1	$1.231\,558\,254\,354\,619 \times 10^{-7}$	1.0
creilly1	2	$2.034\,554\,614\,544\,656\,8 \times 10^{-7}$	1.0
creilly1	3	$7.316\,467\,015\,672\,806 \times 10^{-8}$	0.9999999999999998
creilly1	4	$1.055\,971\,659\,135\,280\,6 \times 10^{-7}$	0.99356906512808
creilly1	5	$1.231\,558\,254\,354\,619 \times 10^{-7}$	1.0
DavetheRave	1	$5.239\,159\,390\,172\,232\,6 \times 10^{-9}$	0.9492622930986467
DavetheRave	2	$2.891\,038\,904\,159\,347 \times 10^{-9}$	0.9588607403653148
DavetheRave	3	$3.622\,737\,888\,087\,755 \times 10^{-9}$	0.9462929690509494
DavetheRave	4	$8.949\,262\,834\,930\,154 \times 10^{-9}$	0.9314482373247217
DavetheRave	5	$4.289\,862\,666\,946\,785 \times 10^{-9}$	0.9426037036635698
Deanerbeck	1	$1.032\,243\,064\,364\,038 \times 10^{-9}$	0.9740215340114144
Deanerbeck	2	$1.406\,783\,830\,738\,999\,3 \times 10^{-9}$	0.9784499686163167
Deanerbeck	3	$2.486\,701\,750\,939\,000\,4 \times 10^{-9}$	0.9811501422614944
Deanerbeck	4	$1.406\,783\,830\,738\,999\,3 \times 10^{-9}$	0.9784499686163167
Deanerbeck	5	$5.297\,448\,992\,699\,553 \times 10^{-10}$	0.9833783437888832
DX13	1	$8.021\,880\,434\,586\,157 \times 10^{-11}$	0.958190302064658

Table F.1: *Continued on next page*

Username	*Fold #*	*p-value*	τ *statistic*
DX13	2	$6.769089021532447 \times 10^{-11}$	0.9765041584740065
DX13	3	$1.672345294150902 4 \times 10^{-10}$	0.9489914621963873
DX13	4	$2.192595885738179 3 \times 10^{-10}$	0.9253937492549975
DX13	5	$7.846209991089 25 \times 10^{-11}$	0.966402573430935
ECDr	1	$2.034554614544656 8 \times 10^{-7}$	1.0
ECDr	2	$1.231558254354619 \times 10^{-7}$	1.0
ECDr	3	$2.034554614544656 8 \times 10^{-7}$	1.0
ECDr	4	$2.034554614544656 8 \times 10^{-7}$	1.0
ECDr	5	$2.034554614544656 8 \times 10^{-7}$	1.0
Ellen	1	$6.935148266306 3 \times 10^{-10}$	0.9625078725300248
Ellen	2	$1.086397214066052 5 \times 10^{-9}$	0.982403317924857
Ellen	3	$1.271878588822927 9 \times 10^{-9}$	0.9625868410556143
Ellen	4	$9.266862632910002 \times 10^{-10}$	0.9663321941277055
Ellen	5	$1.198710413441469 5 \times 10^{-9}$	0.9521904571390466
Ezzey	1	$5.702069557398809 5 \times 10^{-8}$	0.9950859653474028
Ezzey	2	$5.702069557398809 5 \times 10^{-8}$	0.9950859653474028
Ezzey	3	$3.117067728180191 4 \times 10^{-8}$	0.9959919678390986
Ezzey	4	$7.316467015672806 \times 10^{-8}$	0.9999999999999998
Ezzey	5	$3.117067728180191 4 \times 10^{-8}$	0.9959919678390986
Frp97	1	$1.231558254354619 \times 10^{-7}$	1.0
Frp97	2	$7.316467015672806 \times 10^{-8}$	0.9999999999999998
Frp97	3	$4.741611516573964 \times 10^{-8}$	0.9793792286287206
Frp97	4	$7.316467015672806 \times 10^{-8}$	0.9999999999999998
Frp97	5	$3.117067728180191 4 \times 10^{-8}$	0.9959919678390986
Jhannah	1	$3.117067728180191 4 \times 10^{-8}$	0.9959919678390986

Table F.1: *Continued on next page*

Username	*Fold #*	*p-value*	τ *statistic*
Jhannah	2	$2.243\,342\,049\,839\,241 \times 10^{-8}$	0.9630868246861536
Jhannah	3	$1.964\,917\,496\,486\,691\,2 \times 10^{-8}$	0.9850066473065526
Jhannah	4	$2.582\,210\,122\,313\,199 \times 10^{-8}$	0.989743318610787
Jhannah	5	$3.117\,067\,728\,180\,191\,4 \times 10^{-8}$	0.9959919678390986
l17r	1	$4.265\,109\,899\,393\,854\,4 \times 10^{-8}$	1.0
l17r	2	$5.702\,069\,557\,398\,809\,5 \times 10^{-8}$	0.9950859653474028
l17r	3	$5.702\,069\,557\,398\,809\,5 \times 10^{-8}$	0.9950859653474028
l17r	4	$5.702\,069\,557\,398\,809\,5 \times 10^{-8}$	0.9950859653474028
l17r	5	$7.316\,467\,015\,672\,806 \times 10^{-8}$	0.9999999999999998
Luca1802	1	$3.929\,678\,299\,475\,491 \times 10^{-9}$	0.9820613241770825
Luca1802	2	$5.161\,154\,063\,622\,142 \times 10^{-9}$	0.9775252199076786
Luca1802	3	$4.219\,204\,633\,189\,228 \times 10^{-9}$	0.9902910322235658
Luca1802	4	$8.285\,638\,099\,074\,469 \times 10^{-9}$	0.9893045053244826
Luca1802	5	$2.136\,204\,808\,827\,512\,7 \times 10^{-9}$	0.9910712498212336
Martin	1	$7.137\,415\,944\,888\,333 \times 10^{-9}$	0.9434563530497266
Martin	2	$3.929\,678\,299\,475\,491 \times 10^{-9}$	0.9820613241770825
Martin	3	$5.797\,919\,688\,404\,042 \times 10^{-9}$	0.9564144928693535
Martin	4	$3.360\,116\,517\,000\,501\,6 \times 10^{-9}$	0.9843215373488934
Martin	5	$8.285\,638\,099\,074\,469 \times 10^{-9}$	0.9893045053244826
Nari	1	$6.936\,798\,375\,486\,612 \times 10^{-9}$	0.9919891900577794
Nari	2	$4.219\,204\,633\,189\,228 \times 10^{-9}$	0.9902910322235658
Nari	3	$2.136\,204\,808\,827\,512\,7 \times 10^{-9}$	0.9910712498212336
Nari	4	$4.219\,204\,633\,189\,228 \times 10^{-9}$	0.9902910322235658
Nari	5	$3.595\,765\,153\,442\,547 \times 10^{-9}$	0.9927271762054323
sstein	1	$7.316\,467\,015\,672\,806 \times 10^{-8}$	0.9999999999999998

Table F.1: *Continued on next page*

Username	*Fold #*	*p-value*	τ *statistic*
sstein	2	$7.316\,467\,015\,672\,806 \times 10^{-8}$	0.9999999999999998
sstein	3	$1.055\,971\,659\,135\,280\,6 \times 10^{-7}$	0.99356906512808
sstein	4	$1.055\,971\,659\,135\,280\,6 \times 10^{-7}$	0.99356906512808
sstein	5	$1.055\,971\,659\,135\,280\,6 \times 10^{-7}$	0.99356906512808
timri	1	$7.316\,467\,015\,672\,806 \times 10^{-8}$	0.9999999999999998
timri	2	$1.231\,558\,254\,354\,619 \times 10^{-7}$	1.0
timri	3	$7.316\,467\,015\,672\,806 \times 10^{-8}$	0.9999999999999998
timri	4	$5.702\,069\,557\,398\,809\,5 \times 10^{-8}$	0.9950859653474028
timri	5	$7.316\,467\,015\,672\,806 \times 10^{-8}$	0.9999999999999998