Plausibly Realistic Sociotechnical Simulation with Aspect Orientation

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

Table of Contents

Todo list

Chapter 1

Introduction

1.1 A primer on aspect orientation

Explain aspect orientation briefly. Some useful notes for the explanation:

- AO originated in xerox parc, first described in [Kic+97]. There are lots of weaving mechanisms for regular, static aspect orientation, and there's a good early survey of them all (and implementation in a custom OO language specifically for this!) in [MK03].
- AO has some forebears: metaobject protocols, subject-oriented programming, adaptive programming, composition filters. The latter three are described by [CBB19] as being alternative *kinds* of AO — I disagree, but they're certainly attempting similar things.
- The original and still most widely used AO implementation is AspectJ, which comes with its own aspect language. It's grown over the years and is used sometimes in industry [citation needed...]. A smaller alternative would be Spring AOPfind a citation for spring AOP

1.2 Prior Work

Write a section for the introduction describing the work done on pdsf before this, to delineate

where we're starting from and avoid any claims of plagiarism. This can be short, the first sec of the lit review is a proper discussion, but the tool should be mentioned here. See **??** for what already exists.

1.3 Terms & definitions

Complete the glossary in section 1.3.

Decide whether terms like BPMN, simulation & modelling, etc also belong in the glossarysection 1.3.

Aspect

Advice

Joinpoint

Pointcut

Weaving

AspectJ The original aspect orientation framework, with language extensions to describe pointcuts and aspects.

Target The procedure an aspect is applied to via a join point, to affect advice.

PyDySoFu

Chapter 2

Relevant Literature

This thesis presents an aspect-oriented approach to simulation and experimental design, tooling to support these endeavours, some empirical assessment of both the paradigm and its application in the domain of simulation and modelling. In particular, the presented research makes use of aspects to model hypotheses and the complexities of variable and unpredictable behaviour in the simulation of socio-technical systems.

Relevant literature in this topic typically comes from a variety of fields which do not overlap significantly, meaning that this review must cover segments of several parly-related fields. The presented material is eclectic in nature as a result. We therefore present context for unfamiliar readers as well as a motivating case for the work to follow in three distinct areas, which are, in order:

Section 2.1, which introduces aspect orientation and gives a background on the field;

Section 2.2, which details some existing approaches to the dynamic weaving of aspects;

Section 2.3, which discusses the use of aspect orientation in simulation and modelling; and

Section 2.4, which outlines literature on the modelling of process variance, particularly in simulation and modelling.

2.1 Aspect Orientation

Often, software engineers have to repetitively handle an issue in some codebase, where the issue is pervasive across many parts of the codebase and is necessarily interwoven through its core functionality. Common examples of this are guarding against unsafe concurrency usage, memory management, and logging. Maintainability is considered a key trait of maintainable, flexible and legible programs[Par72]. Modern design techniques often center around the structure of a program to increase its modularity, with object-oriented programming being the standard approach to designing with modularity in mind in many industrially-relevant languages today. Citation needed?

Approaches to modularity typically section codebases into units of functionality. Concerns such as logging and memory management happen in effectively all areas of a codebase, as a result of engineering needs and the properties of a project's language and environment. As a result, common devices employed with the aim of increasing modularity are unable to strip these "cross-cutting concerns" into some separate, modular unit. Programmers separating these concerns into additional modules are expected to see two key benefits:

① A reduction of *"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 A reduction of *"scattering"*, where program logic for cross-cutting concerns is strewn throughout a codebase, making maintainance of this code more difficult.

In other words, cross-cutting concerns are expected to make maintainance of both ancillary program logic and a program's core logic more difficult. Addressing this, Kiczales et al. introduced the notion of aspect-oriented programming [Kic+97]. Aspects are best described through their component parts:

- A *"join point"* defines some point in a program's execution (usually the moment of invocation or return of some function or method).
- *"Advice"* defines some behaviour, such as emitting a logline, 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 cut"*s: 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 definition of join points and advice, or how weaving occurs, is a matter left for aspect orientation frameworks and 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 maintainance and comprehension), and reminaing program logic can be understood and maintained without the overhead imposed by the previously tangled cross-cutting concerns.

In **??**, Kiczales et al. see these engineering concepts as universal throughout business logic, motivating the aspect-oriented approach for the first time. The authors present an implementation of AOP in Lisp, and compare implementations by way of e.g. SLOC count in an emitted C program to a comparable, non-AOP implementation, with two examples (its use in image processing and document processing). They find the idea — which they note is "young" and note 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 implemenation. It is noted that measuring the benefits of their approach quantitatively is challenging.

This approach was later developed into more robust tooling by Kiczales et al. [Kic+01], in which more technical detail is discussed. The need for the tool is born from the growing community, and their need for a tool with which to demonstrate their paradigm in case studies. The tool is intended to serve as "the basis of an empirical assessment of aspect-oriented programming". The library makes use of standard AOP 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 AOP techniques; the authors are explicit about their approach being distinct from metaprogramming in, say, Smalltalk or Clojure.

2.1.1 Criticisms of aspect-oriented programming

The growing aspect-oriented programming research community collected both proponents and detractors of the paradigm. The developments in aspect-orientation pertinent to the research presented in this thesis are discussed in later sections of this literature review. However, common criticisms of aspect-oriented programming are important to present in two regards:

① Discussions of advancements in the field pertinent to the work presented in this thesis should be understood within the context of some perceived weaknesses in the field, which helps to frame an understanding of literature reviewed in this chapter,

② The presented work addresses some criticisms of aspect-oriented programming, meaning that the criticisms of the paradigm writ large and properties of work published in awareness of those weaknesses will motivate some research presented in later chapters.

An early piece of scepticism in the aspect orientation community is [CSS04], in which Constantinides, Skotiniotis, and Stoerzer see AOP's core concepts as having significant similarities to `GO-TO` statements, which have historically been the subject of some derision in the literature. [CSS04] is, in spirit, a child of Dijkstra's "Letters to the editor: go to statement considered harmful". The authors note that the notion of unstructured control flow makes reasoning about a program complicated — disorientating a programmer by way of "destroying their coordinate system", leaving them unsure about both a program's flow of execution and the states at different points of that flow — and discuss whether aspect orientated programs can have a consistent "coordinate system" for developers. They note that, while Go To statements are at least visible in disrupted code, the AOP concept of obliviousness makes such reasoning even more difficult than Go To statements, as even the understanding of where and how flow is interrupted is not represented structurally within an aspect-oriented program. They compare aspects to a Come From statement, noting that the concept is a literal April Fools'

joke for programming language enthusiasts who claim they've found an improvement over Go To statements. The authors conclude that existing techniques, specifically Dynamic Dispatch in OOP, provide similar benefits without the tradeoff in legibility of a program's intended execution.

A similar, but more thorough critique of the aspect oriented paradigm is Steimann's work in [Ste06]. The concern in this paper is that the popularity of aspect oriented programming — which was nearly 10 years old at time of publication — was founded on a perception that it assisted in engineering more than it was proof that such assistance viable in practice. The author notes that most papers are theoretical in their discussion on tooling, that examples were typically repetitive, and that the community's discussion concerned more what aspect orientation is good for than what it actually is in practice. AOP is compared against OOP, AOP's claimed properties and principles are examined in detail, and the impact on software engineering is reasoned about from a skeptical perspective, comparing claims such as improved modularity against classic papers on the subjects (such as Parnas' work on the same). The paper presents a philosophical examination of aspect orientation, assessing the paradigm against its purported merits and discussing whether we should expect, rationally, that the claims made by the AOP research community would hold true. The paper ends noting some benefits of AOP that do hold true under rational scruitiny, and notes that the true utility of AOP may be very different to those purported by the community. Overall, the paper is a philosophical and critical reflection on the state of AOP research and the community's zeitgeist at the time, where what it is isn't necessarily consistent with claims around what it's useful for. In particular, the author sees AOP's promise of unprecedented modularisation as unfulfillable.

Similar sentiments are shared in [Prz10] by Przybylek, who looks to examine aspect oriented programming within 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". The paper discusses whether the modularity offered by aspect orientation actually makes code more modular. In particular, they distinguish between lexical separation of concerns and the separation of concerns originally discussed by Dijkstra and Dijkstra in "On the role of scientific thought". They assess the principles of modularity — modular reasoning, interface design, and a decrease in coupling, for example — and find that from a theoretical perspective, there are many reasons to believe that the aspect-oriented paradigm can

detrimentally impact the expected benefits of proper modularisation in a program. They conclude that the benefits touted by AOP are a myth repeated often enough to be believed, but point to many papers which suggest improvements to the standard AOP approach which might reduce it's negative impact or make it more practically viable. Przybylek presents a critical review of aspect orientation literature, but often hints at others' solutions to the problems identified too.

2.2 Dynamic Aspect Weaving

One implementation of dynamic weaving is PROSE [PGA02; PAG03], a library which achieves dynamic weaving by use of a Just-In-Time compiler for Java. The authors saw aspect orientation as a 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 pointcut used is correct. If not, a developer could use dynamic weaving to remove a mis-applied aspect, rewrite the pointcut, and weave again without recompiling and relaunching their project.

Indeed, the conclusion Popovici, Alonso, and Gross provide in [PAG03] indicates 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.

PROSE 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 software making use of aspects does not constitute a product.

The performance issues noted by Popovici, Alonso, and Gross are explored in more detail by Chitchyan and Sommerville in [CS04]. Chitchyan and Sommerville present a review of early dynamic aspect orientation techniques. The paper reviews AspectWerkz, JBoss, Prose, and Nanning Aspects through the lens of the authors' prior work on dynamic reconfiguration of software systems and their generalisation of dynamic aspect orientation 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".

Because of the paper's focus on software reconfiguration (rather than the mechanics and design of dynamic aspect weaving specifically), the analysis of the tools presented in the paper is of less relevance to the work presented in this thesis than their generalisation of dynamic weaving. The tradeoffs of the three generalised philosophies are discussed. 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. It's explicitly noted that, in practice, one could use many of the systems they describe. Though the paper is an early publication in the field, no tool the authors review offers all three, and none offers collective weaving alongside either kind of hook weaving.

In contrast, Gilani and Spinczyk note 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 available memory. Gilani and Spinczyk therefore propose a framework through which weavers can be assessed for suitability in a given domain, or generated from a set of possible features (where, presumably, features would be enabled and disabled as per an environment's needs). Their families of weavers are defined by the similarities of the requirements in domains they are applied to, and specifically defined by their tradeoff between dynamism and resource use (asserted to be broadly proportional). It is unclear that a carefully crafted "actual hook weaver", or JIT-compiled "collective weaving", in the parlance of Chitchyan and Sommerville (see [CS04]), would be meaningfully less efficient than static weaving in all but the extreme application areas outlined in the paper (embedded systems with resources in the range of 30kb memory). Aspect oriented programming's criticism can often be that it doesn't know what it "aims to be good for" , and its application in such extreme environ-

ments is arguably mistaken from the off. The families outlined in Gilani and Spinczyk's publication are unneccesary if dynamic aspects are not required in their target environments. Steimann's critique of aspect-oriented programming, contrasted against these families, presents an interesting question. If the goals of dynamism and resource efficiency are at odds, and Steimann's stance that aspect-oriented programs do not earn its proponents' plaudits in practice, what can dynamic aspect weaving be appropriately applied to? In what environment does the tradeoff presented by dynamic weaving not necessitate a theory like Gilani and Spinczyk's in the first place? Arguably, that environment is not found in low-resource systems, and a take-away of [GS04] could be that researchers should seek other contexts in which to apply aspect oriented programming. [1]

et al. propose a new aspect-oriented invocation mechanism, which they call "Bind". Bind's design is motivated by perceived opportunities to improve modularity from a design perspective. The impact of "scattering" and "tangling" in a codebase after weaving in some aspect orientation implementations leads to a more complicated post-weave codebase, which in turn leads to increased difficulty including the compilation of aspect-oriented code and the development and execution of unit tests on said code. In order to demonstrate Bind's approach to simplifying post-weave codebases, the design of "Nu", an aspect-orientation framework in .NET supporting Bind, is explained and an implementation presented. They present Bind as an alternative to the weaving of aspect hooks (for load-time registration) into target code, in the style of PROSE (see [PGA02; PAG03]), or the weaving of calls directly, in the style of AspectJ (see [Kic+01]). Bind's model for aspect orientation is to apply or remove implementations of crosscutting concerns to arbitrary sets of join points at a time of a developer's choosing. Nu's model for this is designed with the aim of granularity of join point specification. What results is a flexible model for aspect orientation which is demonstrated to satisfactorially emulate many other models for aspect oriented programming, such as the models of AspectJ, HyperJ, and Adaptive Programming. It is noted that it is "very common in aspect-oriented programming research literature to provide language extensions to support new properties of aspect-like constructs", and note that their work is similar to (yet distinct from) weaving approaches in run-time & load-time weaving, support for aspect orientation directly in a language's virtual machine, and work towards general models of aspect oriented programming (models which can represent a variety of existing approaches). Their approach is flexible and considered enough to warrant impact in the introduction of aspect orientation within

[1] As discussed alongside [GK99] in Section 2.3, simulation & modelling might present a more appropriate field.

virtual machines (because they require no direct support), and in their ability to represent different weaving approaches, arguably *because* their approach is general enough in design to approach the general model worked towards, which qualifies their satisfaction of their motivation to provide a model distinct to the approaches initially discussed. In line with the complaints of AOP's critics, this does not seem to qualify the sastisfaction with which they achieve their practical engineering goals.

In [DR10], Dyer and Rajan explain in more depth than in [**rajan2006nu**] 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. Notably, the implementation discussed is a Java implementation, rather than the .Net implementation presented in [**rajan2006nu**]. Many aspect orientation frameworks are language-specific; the existence of Nu's implementation on multiple platforms highlights the work's most interesting facet being the design of the Bind primitive, rather than the framework itself. In a research area where tooling papers are common but the lack of design philosophy & analysis of case studies is frequent fodder for critics, the novelty of the Bind mechanism distinguishes this series of papers.

2.3 Aspect Orientation in Simulation & Modelling

Aspect orientation as applied to simulating systems, and building models of systems, has been researched from several approaches.

A very early example of aspect orientation in simulation & modelling is [GK99]. Gulyás and Kozsik observes that, in the study of complex systems through software models, the software developed typically serves two purposes: the experimental subject, and the observational apparatus used to conduct the experiment itself. Arguing that separting these roles ought to make both the implementation of an experimental system and its later analysis simpler, Gulyás and Kozsik proposes the use of aspect orientation as a means of separating what they perceive as cross-cutting concerns of systems modelling. They present their Multi-Agent Modelling Language, a language implemented in Objective-C via the Swarm simulation package and designed for aspect-oriented simulation of agent-based models. Their aspect orientation effectively makes use of Observer patterns to measure a pre-constructed system

under simulation, without the observations being an intrinsic component of the simulated system. They find that AOP provides an intuitive and straightforward method by which experimental simulated systems can be composed, and that MAML's simplicity and its philosophy on modelling are more "satisfactory" than Swarm's standard approach, though the paper betrays that its 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 [GK99], Gulyás and Kozsik present not only tooling for aspect oriented simulation, but some reasoning & philosophy on the potential benefit of using aspect orientation in these endeavours that extends further than the conclusions of modularity through separation of concerns 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 many aspect orientation papers reviewed in this chapter. Many papers describe the expected benefits by simply drawing from existing literature such as [Kic+97]. The fact that a rare example of detailed reasoning about the appropriateness of aspect oriented programming in a particular domain is highlighted because the domain in question is simulation & modelling; the subject of this thesis. That aspect orientation might be well suited to separating observer and experiment motivates, in part, this thesis' work showing the plausible realism of a simulation in which behaviour is modified by aspects. Put another way: 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", a premise this thesis shares.

2.4 Variance & Aspect Orientation in Simulation & Modelling

Chapter 3

Rewriting PyDySoFu

Check the most up-to-date pdsf implementation — is it actually on that backup drive?

The work undertaken in this thesis required in improved implementation of old tooling. When previously used, PyDySoFu was a proof of concept which could feasibly produce scientific simulations, but was implemented in a manner which was not optimised for speed (making it a burden for large simulations), lacked granularity in the application of its aspect hooks (hooks could only be applied to entire classes), and most importantly, did not work with Python3 (Python2 support officially ended during this PhD).

This chapter briefly outlines the new implementation of PyDySoFu, discusses improvements made to design and performance, and explains some contributions made to the design of aspect orientation frameworks which addresses some core issues raised with the paradigm. Consider adding references to the sections through this PDSF chapter, depending on how beefy it becomes…

3.1 Requirements for Change

As time wore on with PyDySoFu's original implementation, it became increasingly clear that a rewrite was required. PyDySoFu grew out of an undergraduate project, and accrued technical debt as a result of being written under extreme time constraints with little experience. On revisiting, and

on reflecting on other aspect orientation frameworks (as discussed in section 2.2 and [CS04]) and the use previously found for PyDySoFu (see [WS18a; WS18b]), it was clear that there were a series of improvements which could be made in the process of rewriting the tool:

- Before this body of work, 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 read-only wrapper class that made the replacement of `__getattribute__` impossible via the previous route.

- PyDySoFu's original implementation made no serious accomodations for efficiency. It could be seen as the "total weaving" described by Chitchyan and Sommerville in [CS04], and it was not possible to provide additional options to ensure that aspects could be as efficiently woven as possible at runtime given a particular use-case.

- The original PyDySoFu implementation wove onto a *class*, meaning that even properties of the class which were not considered join points were still affected by the weaving, even if in a minor way. Because `__getattribute__` retrieves all attributes including special builtin attributes and non-callable attributes, these are also returned via the modified implementation of `__getattribute__`, incurring an overhead, albeit small, for all attribute resolutions instead of a desired subset.

- The original PyDySoFu implementation 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 is 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 [Kic+01], join points could not be specified by pattern. Instead, each individual join point must be supplied as a Python object. This means that, while the target attributes are still oblivious to the advice applied to them, the application of that advice could not be written obliviously.

As a large number of requirements were left unfulfilled by the original implementation of PyDySoFu, a new implementation satisfying them was deemed necessary.

3.2 Python3 Specific Implementation

Replacing `__getattribute__` on the class of a targeted method was no longer viable in Python 3. A replacement method therefore had to be found. For clarity: replacing `__getattribute__` allowed for hooks to be woven (at runtime) into likely future targets for advice. These hooks would then discover and manage the execution of advice around each target. Because advice can be run before and around a target, and dynamic weaving implies that advice could be supplied or removed at any time, we look to intercept the calling of any target, and manage advice immediately before execution. So, the task at hand is to find a method of attaching additional work to the calling of any potential target, before that target is executed. We refer to code woven around a target which manages applied advice as *aspect hooks*.

3.2.1 Abandoned techniques

Rather than "monkey-patching"[1] a new version of `__getattribute__` with hooks for weaving aspects, the rewritten method could be patched to the object itself at a deeper level than used in the original PyDySoFu implementation. 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 [con21]. Efforts were made to add the required functionality to ForbiddenFruit — patching `__getattribute__` directly on the object, or "cursing" it in ForbiddenFruit jargon — but this was abandoned as the underlying mechanism is particularly unsafe, Python API changes could render the work unusable in future versions easily, 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__` which did not work,

[1] Making on-the-fly changes to object behaviours / definitions by taking advantage of scripting languages' typically flexible object structures, such as objects literally being maps from string attribute / method names to the associated underlying value. Monkey-patching makes use of these simple structures and changes object behaviour by replacing values such as the function object mapped to by the original function's name in the dictionary. This is the method by which PyDySoFu originally replaced `_getattribute__` on a class object.

and attempts proved challenging, indicating that this would also be complicated to maintain over time. 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 function call*.

Other approaches involved making use of existing Python functionality for interrupting method calls. As PyDySoFu wraps method calls at execution time, what is required is to add functionality to the beginning and end of the execution of a method. Python has built-in functionality for implementing debuggers, profilers, and similar development tools, which provides exactly this functionality, as debuggers must be able to — at any point during execution marked as a breakpoint — pause a running program and inspect call stacks, the values of variables, and so on. As a result, the method `settrace()` allows a developer to specify a hook providing additional functionality to a program. Making use of this also has issues in our case. Most significantly, `settrace()` catches myriad events in the Python interpreter which PyDySoFu may not need to concern itself with, incurring significant overhead. In addition, use of the function overrides previous calls to it, meaning that any debuggers used by a user of PyDySoFu would be replaced with PyDySoFu's functionality, which was deemed untenable. However, it is worth noting that the technique could work in theory, and if future versions of Python allow for multiple trace handlers being managed by `settrace()`, this could provide an interesting approach when implementing future dynamic aspect orientation frameworks.

3.2.2 A viable technique: import hooks

A final available technique was to continue to monkey-patch hooks to discover and weave aspects, via an alternative method which did not make use of `__getattribute__`. This approach would change the use of PyDySoFu slightly to make a comprimise between performance and obliviousness of aspect application: when *importing* a module targeted for aspect weaving, methods which are potential weaving targets are invisibly monkey-patched with a wrapper method with a reference to the original[2] and hooks to detect and run dynamically supplied advice.

An important note for discussing the implementation of PyDySoFu is that almost all Python

[2]Necessary to run the originally targeted method.

functionality operates by use of its "magic methods"[3], which has the affect of making the language an ideal environment to implement dynamic aspect orientation. Our method of adding hooks to modules at import time is an example of this. Python's built in importing functionality 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.

Monkey-patching `builtins.__import__` is as simple as replacing the function object with a new one, which has the effect of changing 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, our intent is not necessarily to manipulate *all* modules, but a subset of imports specified by a modeller as suitable for manipulation. If all imported modules were affected, this would include all invocations of `import`, including those made recursively by package implementations, for example. Therefore, it is important to have a mechanism to enable and disable the weaving of aspect hooks on each import (effectively, to enable and disable PyDySoFu's modified import logic).

Decide whether this needs to be more thoroughly broken up / structured…

Include a discussion of *what* gets hooks added using this method…

This can be done 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.

3.2.3 Implementing import hooks

We are interested in manipulating `builtins.__import__` only when imports are made which should have aspect hooks woven. We enable this new import behaviour with a syntax of the form:

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

[3] "Magic methods" are methods beginning and ending with two _ characters. The Python language documentation specifies sets of magic methods and their required function signatures which are used internally to implement functionality — 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 support more than operator overloading. 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 (not unlike a function). More can be found in the Python documentation[VD09], although more focused guides exist in the Python community [con16].

```
class AspectHooks:
    def __enter__(self, *args, **kwargs):
        self.old_import = __import__
        import builtins
        builtins.__import__ = self.__import__

    def __import__(self, *args, **kwargs):
        # ...replacement import logic for performing weaving...

    def __exit__(self, *args, **kwargs):
        builtins.__import__ = self.old_import
```

Figure 3.1: Magic methods used to enable the `with` keyword usage for PyDySoFu

... which would weave aspect hooks into all functions and (non-builtin) class methods within the `mymodule` module object added to the local namespace of the importing stack[4]. Less formally: importing `with AspectHooks()` applies aspect hooks to all potential targets of advice in the `mymodule` package. The behaviour of Python's `with` keyword is defined by more magic methods: any object with `__enter__()` and `__exit__()` defined can be used here, where `__enter__()` is run at the beginning of the enclosed block, and `__exit__()` when leaving the block.

PyDySoFu caches the original `builtins.__import__` object in an instance of the class, and replaces it with `AspectHooks.__import__`, in its `__enter__()` method. This is reversed by replacing `builtins.__import__()` with the cached object in its `__exit__()` function. The resulting implementation for weaving aspect hooks is satisfyingly uncomplicated, as can be seen in fig. 3.1.

3.2.4 Strengths and weaknesses of import hooks

As a technique for weaving aspect hooks, this new method provides multiple benefits. Application of aspect hooks is straightforward from the perspective of a modeller using PyDySoFu, whose code clearly applies aspect hooks and does so in a legible way for future maintainers, i.e. there is no confusion as to where aspect hooks might be applied. Aspect hooks can be applied to specific modules or every module depending on the use of the supplied `with` statement, allowing for total weaving or actual hook weaving [CS04] depending on their preferences. Further, performance is optimised at least in comparison to the previous implementation of PyDySoFu, as hooks are weave-able at a more granular

[4] Python's use of the stack namespace in its importing system means that careless re-importing a module can lead to multiple copies of it in different function stacks, meaning that the same name resolution (such as resolving a class by its name in a module) might, after applying aspect hooks in PyDySoFu, change the behaviour of procedures depending on where they are called. Scenarios where this might arise are deemed unlikely enough that the risk of this design decision becoming troublesome are considered negligable. Still, it would be remiss not to make note of the fact.

level (on the level of procedures such as functions or methods, rather than all attributes of a class).

However, there are also caveats of this approach that are necessary to address. As aspect hooks are woven in the new implementation of PyDySoFu via Python's import functionality, any procedure not imported from a module cannot have aspect hooks attached.Consider adding local namespace weaving to pdsf3: should be easy to implement as a cheeky little monkey-patch… However, as aspect orientation is primarily concerned with a separation-of-concerns approach to software architecture, targets are expected to exist in other modules, and we do not consider this to be a significant limitation.

A more significant limitation of the import hook approach is that the object with aspect hooks woven exists in the namespace of the function *importing* the function. In other words, this method makes it impossible for a module to make use of aspect hooks that are woven in an unrelated piece of code. We therefore have a "semi-oblivious" property to our aspect orientation approach: targets of advice are unaware of any adaptations made, but *any code making use of those adaptations must be aware enough to at least apply aspect hooks*[5].

In a manner of speaking, this can be considered to alleviate some concerns with aspect orientation as a paradigm. Aspect Orientation is criticised for making reasoning about programs more difficult [Prz10; CSS04; Ste06]. One cause of this is that aspects separate logic from where it is run; Constantinides, Skotiniotis, and Stoerzer 's comparison with the jokingly proposed `come from` statement [Cla73; CSS04] is a reminder that it can be effectively impossible to understand how a program will execute if the path of execution is not at least linear or clearly decipherable from source code. Aspect orientation as a paradigm inherantly violates this linearity. However, import hooks as implemented in fig. 3.1 present code which can be interpreted in one of only two ways:

① Looking at the original implementation of a procedure, its intended execution is clear. A programmer can make use of this directly and it is guaranteed to behave as expected.

② Any program making use of a procedure imported from a module will see, when the procedure is imported, whether it has had aspect hooks applied. In this case its behaviour is unknown — falling prey to the design flaws discussed in the aspect orientation literature **??????** — but this unpredictability is at least highlighted to the programmer.[6]

[5] Note that once aspect hooks are applied, advice can still be supplied from anywhere in the codebase.
[6] It is worth noting that a third case technically exists, where a procedure is imported from a module which imports that

As a result, while import hooks are somewhat limited in that they are applied specifically to imported code and break the traditional AOP concept of obliviousness in at least a weak manner, these two facts combine to arguably fix a latent issue in the design of the aspect oriented paradigm. The original PyDySoFu implementation was able to modify any procedure in a more traditional, oblivious manner. While this new implementation is clearly more limited as a result, we consider these limitations an overall benefit to the design of the tool, and a contribution to aspect orientation framework design.

3.2.5 Weaving process

Describe the improved process of weaving in PDSF3

Find the newest copy of PDSF3, make a repo for it, cite the new impl

3.3 Discussion

The new implementation of PyDySoFu makes a few contributions, particularly in comparison to the previous version:

- Its new technique of weaving aspect hooks on import, making use of Python's `with` keyword, improves aspect orientation framework design by trading a degree of obliviousness for clarity
- Aspect hooks can be applied with more precision than the previous implementation of PyDySoFu, meaning:
  - Users of the framework can better delineate between total and actual hook weaving
  - Unnecessary overheads from checking dynamically applied aspects at each join point are reduced.
-

Despite this, there is room for improvement in the design of the framework:

procedure from another module. If the latter module contains the implementation and the former applies aspect hooks when it imports, then any program making use of the former module will be importing a procedure with aspect hooks applied implicitly. However, these situations are still visible through simple inspection of these chained imports, where other aspect orientation frameworks might apply an aspect to any join point at any time, without this being obviously discoverable by a programer.

- Caching of applied aspects to join points could be implemented. If between two invocations of a target no changes have been made to the applied aspects, a function object containing the composed aspects from earlier invocations should be run. This would permit runtime aspect weaving with less overhead, as searching for applied aspects need not be performed at every target invocation. Targets should have "changedness" flags which are set every time an aspect is applied or removed from it.

- 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 [Tha+17].

Chapter 4

RPGLite: A Mobile Game for Collecting Data

RPGLite is a game designed with some special qualities: Kavanagh and Miller have produced PRISM models which can be model-checked to identify ideal play strategies in all game states[KM20]maybe add more kavanagh citations early on!. Some experiments were conducted around RPGLite to answer the question: "Over time, do players converge on an ideal strategy of play?"Fix formatting and correct wording of William's hypothesis, and mine below.

An alternative question to answer would be, "what strategy of play do players typically adopt", and the related question, "do all players adopt the same strategies?" These are not scientific hypotheses, but interesting questions to ask of a game where "correct" and "incorrect" actions can be categorised. Moreover, Kavanagh and Miller's work can identify the *cost* of an action, allowing for even richer datasets and analyses. It would not be possible to perform actual analyses of player behaviour without real-world player data, however.

To that end, a collaboration was undertaken with Kavanagh and Miller to develop and release a mobile implementation of RPGLite which would collect player data for later analysis. Kavanagh and Miller would get to demonstrate the utility of their model checking in an empirical scenario. We would get to develop models representing player behaviour, and check these models against the collected data. This represents an ideal opportunity to make use of aspect orientation in a new context: a model of naive RPGLite play would be produced which represented random playThe naive version *does* do random play...right?, and aspects could be written which augment the naive model with guesses as to

player behaviour. If the data augmented models generate correlates with empirical data more closely than the naive data, we can dismiss naive play as "realistic", and assume the augmented behaviour. Many aspects can be written representing different styles of play, which might be adopted by different players, a concrete benefit of aspect orientation in modelling & simulation. This chapter discusses the design and implementation of RPGLite for data collection purposes, allowing for discussions of actual experiments — and a more detailled examination of the application of aspect orientation — in chapterscref the chapters on specific experiments at the beginning of the RPGLite chapter..

Add an outline of the RPGLite chapter here.

4.1 An Overview of RPGLite

RPGLite is a simple two-player game played in turns. Each player selects characters independent of the other, with each character having a unique set of abilities and properties, which are generally health, chance of success on attack, and damage dealt on a successful attack. The abilities of some characters necessitate additional properties. Each player selects an "alive" character (one with health greater than 0) to perform their action against a chosen "alive" target (or occasionally targets). A successful attack — randomly determined by chance of successful attack for the selected attacking character — results in that character's unique ability being inflicted on their target[s]. A random player is chosen to take a first move, players may always skip their turn as a valid action, and players continue to take alternating turns until a victor is left with the only "alive" characters.

Eight characters are available for selection, with the following abilities:

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 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.

Specific details of each character — their health, chance to hit, and damage on hit as well as character-specific details (such as the threshold for additional Barbarian or Rogue damage, for instance) — are defined as a *"configuration"* of RPGLite. 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 and Miller found that model-checking a configuration of the game could discover the relative strengths of characters and character pairs when played optimallyCite the correct paper for the game blaancing!.

RPGLite's design has two objectives it must meet. First, that it is interesting to players, which requires that it is approachable and complex enough not to be immediately solvable. This is necessary for real-world data collection, and to demonstrate a design representative of something that could conceivably be a real-world game with an active playerbase. Second, RPGLite's design must be sufficiently simple for model-checking. Model checking is a necessary requirement of design because of our need to identify optimal moves: analysis of player behaviour rests on our understanding of how close to "ideal" players are, and whether players approach ideal strategies over time. This is the crux of the work found in []Cite William's PhD thesis for explaining RPGLite design., which relies on a reduced

state space in order to calculate optimal moves, character pairings, and the like.

4.1.1 RPGLite's Design Implications

The state space of RPGLite makes it unusually well-suited to analysis through formal methods. Because of this, a the datasets produced through simulation of RPGLite can be compared against two other datasets: one of real-world play, and another of what can be mathematically shown to be "correct" player behaviour.

To demonstrate this state space, note that RPGLite games can have their states described by a set of values: the healths of characters on each team, plus a stunned character. With eight characters having a maximum health value of w, x, y, z[1], two players, and an indicator of which character is stunned.[2]. The entropy of a game's state is therefore $log_2(w \times x \times y \times z \times 3 \times 2)$, where the multiplications by 3 and 2 represent the stunnedness indicator for the current player (either character, or neither), and the player whose turn it is to play, respectively. The maximum health for a character is 10 hit points, which makes the maximum entropy of a game state $log_2(10^4 \times 3 \times 2\}) = log_2(60000) \approx 16.87 bits$. Each player picks 2 of 10 characters, and can choose to attack either opponent character with either of their own, or skip their turn, for a maximum of 6 possible actions. We can therefore see that the total entropy of the entire RPGLite game is no greater than $log_2(10^4 \times 6 \times 2 \times \binom{10}{2} \times 6 \approx 24.36 bits$.[3]

Iterating through these states allows us to map the entire state space of RPGLite. As the state space the game defines is relatively small — for comparison, mapping valid positions in chess takes about 136 bits[Nie77], and this figure does not account for valid *moves* within the game, which our calculation for RPGLite does — it is feasible to analyse every possible game state. Note that the figure of $\approx 24.36 bits$ includes movements *between* states as well as the states themselves, meaning that it is feasible to map

[1] Maximum health values are dependent on RPGLite's configuration.

[2] Note that stunnedness is valid for exactly one character for one turn, meaning that only one character may be stunned at any time, and the status effect immediately resets, meaning there are only three possible states for stunnedness: either character belonging to the player taking a turn, or neither.

[3] The actual figure is smaller:

- Players may choose two characters with special abilities that prevent them from attacking both opponent characters at once (this accounts for 9 out of 10 characters), giving 5 possible actions in a turn, rather than 6.
- The "metagame", which refers to the percieved "best" strategies at any given point in time, would impact the chance of a player selecting making certain moves or choosing to play with certain characters. RPGLite is designed to be slightly "unbalanced" in the parameters of different characters such as health, attack damage, or potency of special abilities, meaning long-term players are expected to learn effective playstyles and adjust accordingly.

Players' behaviour is therefore less uniform than this calculation would imply, but the calculation provides a *maximum* entropy of the game.

the potential progressions through all possible games of RPGLite using formal methods. Further, this allows us to understand the chances of a given player winning given transitions between different states, for example by representing moves in the game as transitions in a decision diagram where nodes are the game's state. We can calculate exactly how "good" a move is, by comparing chances of success making a given move in a given state against the chances of success when making the calculably-optimal move.

In this way, RPGLite's design allows it to be understood formally, yet it also draws on common game design elements and is sufficiently "interesting" to generate data from a real-world playerbase.[4] This yields some properties that are interesting for the purposes of aspect-oriented simulation:

① Simulated moves can be selected naively, i.e. at random, but can also be made perfectly according to the known-correct move in a given game state, or made with some calculated "cost" as to the chance of winning.

② Real-world players' behaviour can also be analysed according to the same metrics: for example, moves made by players can be analysed to understand 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,[5] the biases of players may be represented as aspects which are applied only to specific actors within the simulation.

④ Should aspects be suitable as a manner of accurately representing biased play, aspects offer a separation of concerns within the simulation: any nuance found within the playstyle of specific real-world players would be replicated and applied not to the model itself, but to specific simulated players. Playstyles 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

[4] Data collected from several thousand completed games can be found at [KWM20].
[5] The concept of play correlation is introduced in chapter 5.

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, the system is closed insofar as all interactions within the system are known and all game elements are precicely understood, and all interactions take place between experiment participants for data collection purposes, allowing for a large dataset to be collected without information being removed due to players not consenting to their data being collected and disseminated for science. In short, RPGLite's design constitutes a system where all aspects are well-understood, no interference is anticipated from system components which are unknown or outside of experimental control, and lots of data can be collected for analysis. Therefore, the data generated by gameplay is suitable for comparison against datasets outputted through aspect-oriented simulation. This can be used to assess whether simulated players with aspect-affected behaviour accurately reflect the playerbase, and so can help to assess whether aspect orientation is suitable for realistic simulation.

4.2 Implementation of RPGLite

Knowing ideal play is useful, but to understand how real-world players would 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. Play constituted engagement with an experiment for data collection, and after several months, a database logging player behaviour presented a dataset which could be used to simulate real-world player behaviour.

This section briefly describes the details of RPGLite's implementation as a data collection tool. Some lessons learned after reflection on the implementation process were documented for the benefit of others' avoiding our errors[Wal+21], might produce a more complete overview of the development.

4.2.1 Mobile app

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[6]. Most assets

[6]Meaning that there are technically also versions of RPGLite playable on, say, a games console or web browser.

were developed in GIMP, with character designs contributed by a commissioned artist online. Unity allowed for a "WYSIWYG" or what-you-see-is-what-you-get 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 William Kavanagh, the collaborator on the project and original RPGLite designer. Therefore, in an attempt not to take credit for this work, see Cite William's thesis here for client-side development notes for full details.

Beta testing required user engagement. Apps were deployed to Android and iOS devices of colleagues, who played a series of games to check that game logic was robust enough (and graphic design adequate enough) for final distribution of the game. Beta tests were iterated for 2-3 months, until the game behaved correctly in all edge cases and a final design was settled upon.

Insert client-side photos here, maybe William's photos of the design changing over time...?

4.2.2 API & Server-side Logic

As the data collected ought to be empirical, RPGLite was developed as an online game. This required a server and API for a client to communicate with.

A REST API was developed with Python's *Flask* framework. Endpoints were created for almost all in-game actions, allowing for player search, matchmaking, player profile design, game history and statistics analysis, ranking calculations, login and password reset, implement mutexes on sensitive information, and other in-game activities. The API would also allow moves to be made, and reject erroneous game states or unauthorised input from any malicious input. The API would also send push notifications to an opponent's device when moves were made, which beta testing showed improved engagement significantly.

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 send data directly to be inputted into our 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 easy analysis of the games' results. The API was also hosted on the same virtual machine. A combination of port access rules and hardening of the database itself prevented unauthorised access to the database, ensuring that the data remained untampered-with.

MORE HERE?!

4.3 Empirical Play and Data Collected

In total, players produced a dataset used in this PhD comprising around 4,000 gamesFind the exact number of games analysed entirely completed[7]. It also includes around 1,000,000 datapoints generated by gameplay or player interaction with the client, such as players checking their history or rolling a dice, although these datapoints are not used in the simulations presented in this simulation. The data is drawn directly from the MongoDB database used to run the game.

Completed games drawn from the MongoDB instance contain many fields, including:

- The history of moves made, and the times those moves were made Confirm moves made includes timestamps, I'm sure it does
- The players involved (by username), and the winning player
- The ELO scores of the players before and after playing the game Confirm that games include both ELO before and after the game is played.
- The characters chosen by each player
- The "score" of each player[8]

[7] "Entirely completed" here means games that ended in a win or lose, not abandoned by players or left unfinished by a player who abandoned the app.

[8] RPGLite's mobile app presented users with a naive scoring mechanism used to rank users on a leaderboard, which some users then used to identify other players of a similar notional skill.

Chapter 5

Simulation Optimisation with Aspect Orientation

With a game deployed to experiment participants and a dataset of empirical play collected, it was possible to determine optimal play in any game state. This entirely separate body of work is documented in another student's PhD thesisCite William's PhD thesis. This dataset leads to further research. If we understand how players *should* play, and we have data to indicate how they *do* play, we can investigate how real-world players might be modelled.

5.1 Aims

Aspect orientation's use in previous simulation and modelling efforts have typically focused on the use of aspects to compose model or simulation detailsThere must be tons of good citations for aspects being used to compose together a simulation / model. Critics of aspect orientation note that the act of process composition makes visually understanding codebases difficult, and so ensuring that a simulation properly models real-world behaviour is made trickier with the introduction of aspect orientation. However, aspect orientation might instead be used to *augment* an existing model, by rethinking what aspects are used to represent.

An alternative use of aspects would be to first build a non-aspect-oriented model of *expected* behaviour, and separately build aspects which describe deviations from this. For example, one might more realistically simulate safety procedures by first producing an idealised, "naive" model of what employees are expected to do, and separately model alterations to prescribed behaviour as an employee's boredom, expectation that checks and balances are unneccesary wastes of their time, and so on — effectively, separating out models of degraded modes[JS07].

Previous research on the use of aspect orientation to model degraded modes adopted the traditionally claimed benefit of aspect orientation: separation of cross-cutting concerns, allowing for a greater reusability of codebases[WS18a]. A repository of cross-cutting concerns in socio-technical simulation such as boredom was developed as a library to be applied to any future models[SW16]. However, aspects used in simulation have no intrinsic need to represent concerns that are cross-cutting. Indeed, whether they can be accurately used to represent cross-cutting concerns in simulation is the topic addressed in Add a cross-reference to the chapter on cross-cutting concern simulation accuracy when it exists. Aspects might instead be used to represent *amendments to processes* which deviate from an expected norm, in this case represented by the idealised model aspects are applied to.

To more concretely relate this to the experiment at hand: play of RPGLite can be modelled as players matchmaking, picking characters, and then mutually taking turns until one player's characters are entirely expired. Once a player's characters are dead, new matches can be made. This can continue indefinitely. Lacking a heuristic to select next moves or characters, players might be modelled as picking random moves. However, heuristics for move selection can be added to the naive model of play by way of augmenting the processes already defined through aspects. This approach can be of significant utility in both modelling player behaviour and accurately modelling different players:

① Different players might use their own unique heuristics to model play. Each player's behaviour is therefore well described by separating what play "looks like" to what makes a given player play differently to their peers.

② Different players might lean more heavily on different heuristics, or mixes thereof. Play might be characterised by reliance on experience, on recent games, on knowledge of an opponent, and so on; these different variables can be expected to be weighted differently by each player, adding

complexity to the code which models this individualised play.

③ A modeller might discover a new idea for a heuristic long after developing an original concept for a model. The easiest methods for amending the original model should require the least rewriting of original code. Due to the impact of ②, ideal architectures for an approach such as this should require these heuristics to be defined entirely separately to the base model.

Considering ①, ②, and ③, architectures and paradigms which enable separation of concerns are well-suited to defining alternative approaches to play. Some architectural approaches such as mixins or plugin design patterns might support this structure well, but they typically rely on language features (in the case of mixins) or knowledge of software engineering (in the case of design patterns). Aspect orientation is typically provided to developers as a framework or runtime in a language (such as AspectJ[Kic+01] or PROSE[PGA02]) and can require minimal architectural understanding to use: concepts are simple, and the effort of composition is alleviated by the supporting framework or runtime.

The approach makes little use of aspect orientation's significant contribution — cross-cutting concerns — as whether behaviour cross-cuts different parts of a codebase is not of interest in this use case. Instead, aspect orientation is treated as a composition mechanism with a reasonably low degree of technical knowledge required.

5.1.1 PyDySoFu Suitability

Some aspect orientation frameworks do not adequately achieve this requirement. For example, the most influential framework, AspectJ, requires the use of language extensions to define integrate aspect orientation[Hil+00], and similar additional complexity is added in seemingly every alternative framework, through the use of bespoke virtual machines, compilers, translators, or languages[**rajan2006nu_towardsAO_invocation**; PAG03; SL07; BH02].

PyDySoFu, however, requires very little additional knowledge to use. Its design prioritises simplicity and a shallow learning curve that makes its adoption by researchers without a software engineering background feasible: maybe cut this list of reasons PyDySoFu is fantastic...

→ PyDySoFu is implemented as a pure-python library, meaning that it can be installed through

Python's package manager (pip) and imported like any other Python library. No additional supporting infrastructure is required.

- Aspects in PyDySoFu are simple functions which take as arguments whichever pieces of information are pertinent for the function's use as an aspect[1].
- To weave a PyDySoFu aspect requires only a method call, which returns a `callable` which unweaves that aspect.
- Defining PyDySoFu pointcuts requires only a regular expression matching a method name. This can apply to a wide range of join points if required, but where method names are provided directly, the join point is made clear.
- Additional clarity over where aspects *can* be woven is introduced by PyDySoFu's transparent weaving of aspect hooks, mitigating some of aspect orientation's most prominent criticisms.

PyDySoFu therefore satisfies the requirements of this work well: it offers composition of procedures outside of the scope of an original codebase, makes what is being composed where clear to a programmer, and makes no significant changes to Python as a language (thereby requiring users to specialise in fewer tools).

5.1.2 Proposed Experiment

Aspect orientation's use as a composition tool for model components makes sense in principle, but it is unclear whether the addition of behaviours to a naive model would make the model more "realistic". It is unclear whether the changes made would properly represent what might be empirically observed, and while PyDySoFu's design makes understanding *what* is being composed simpler than other aspect orientation frameworks, a composed model under this paradigm is still split across multiple areas of a codebase, making a visual assessment of whether a model accurately reflects the intended behaviour impractical.

We can confirm whether aspects can realistically represent changes to a naive understanding of the real world by comparing their output against empirical data. For example, if a such a model of

[1] For example, an "encore" aspect which is woven after a target procedure returns will be provided that target's return value.

behaviour in a system outputs data which correlates poorly against empirically collected data, a change to that system would make it more realistic if it improved this correlation, and could be said to be realistic if the generated data appeared sufficiently "close" to the empirical dataset — which here means that the correlation between the two is of statistical significance. Such a change can be aspect-oriented. Therefore, we can see the application of aspects as the application of packages of potential improvements to a base model, which can be verified by way of comparison to known-good datasets.

This is the basis of the experiments in this thesis.

With datasets collected empirically on RPGLite's play, we can build a naive model of play and aspects to apply that should realistically model data from players. This can be used to answer the question:

> *" Can aspect-oriented models be said to exhibit realism?"*

To answer this question, we will produce a model of play, and develop aspects which encapsulate different play styles so as to compare the aspect-augmented datasets and naive datasets against the empirically sourced data. The following subsections detail the naive model developed and aspects applied to this model respectively.

5.2 Naive Model

A naive model of play was developed by separating each stage of the actions taken by players in the client-side app, and separating them into individual procedures. To facilitate the retrieval of most information about a simulation in an applied aspect, the model was written so as to contain the entire simulation state as mutable function arguments. The model was written as a workflow, and state of workflow execution was separated into three components: the actor a function invocation (or "step") repesents activity from; the context of that step in the execution of a workflow; and the context of that workflow's execution in a broader environment. Incidentally, we found this structure to allow a flexible and natural implementation of a procedural simulation, which should translate easily to existing simulation frameworks such as SimPy[Mat08]:

Actor — allows the function to identify the actor performing the activity defined by the function. This argument is any object uniquely identifying an actor.

Context — allows the function 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 are often 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. Because all actors share access to a global environment, this also provides a message passing space, or a space where actors can set values and flags other actors might look to, should those details be more general than their specific thread of work at a given point in time. [2]

Each simulation step suitable as a join point receives these three arguments at a minimum. Aspects applied to these therefore have access to the entire state of the simulation.

The naive model of RPGLite follows a simple workflow mimicking player interaction with the client-side application used by real-world players. Graphically, it would be represented as a flowchart like so:

TODO FINISH WRITING ABOUT THE NAIVE MODEL'S STEPS

[2] This is different to environments in some other simulation frameworks, such as SimPy[dev21], where the environment controls scheduling and execution: this structure imposes no constraints such as models of time, and anticipates that any such functionality should be implemented by the programmer. However, an environment such as SimPy's might satisfy a programmer's needs when using this particular pattern.

5.3 Aspects Applied

Players can be expected to select stronger characters as they understand gameplay more. Aspects were therefore developed to track player experience with characters. As players became more familiar with the characters they play, we can hypothesise that they will better understand how to play with those characters, and more often select characters they have success with.

Write about *both* aspects so as to answer the hypothesis : Can aspects be used to generate models of alternative behaviours?

Chapter 6

A chapter title here for the experiment moving aspects to new systems, or systems with some changes

Chapter 7

Future Work

The focus of this thesis is in developing a state-of-the-art aspect-oriented framework, producing a suitably constrained experimental environment to demonstrate its effectiveness, and using that environment to investigate whether aspect orientationis suitable for simulation purposes? It's about showing that we *can* use aspect orientation appropriately in simulation environments, and that aspect orientation can also lead us to realistic and nuanced simulations, too. Go back through the pdsf and lit review chapters to make this argument properly.. As we have found that aspect orientation is appropriate in this context, successfully produced this well-constrained environment for simulation, and produced a novel aspect orientation framework which demonstrates novel and powerful weaving concepts, lots of opportunities for research outwith this thesis' scope present themselves.

This chapter describes some possibilities for the presented research to be extended in the future.

Rewrite the chunks in each future work section for this chapter, into their own subsecs with proper explorations, citations, and so on. This chapter is currently a scapbook of ideas!

7.1 Future Work pertaining to PyDySoFu

7.1.1 Aspect-Oriented Metaprogramming

The combination of metaprogramming and aspect-orientation introduces powerful new possibilities in the realm of aspect-orientation. In traditional aspect-oriented work, aspects treat their targets as black boxes. This leads to some limitations:

- Traditional aspects cannot add their behavioural modifications interspersed within the work being done by their target. 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 more mainstream programming styles may wish to insert logging behaviour *within* their business logic, rather than *around* it. Aspect-oriented metaprogramming makes this possible, as the target can have logging logic interspersed through otherwise decoupled business logic when woven. As even in aspect-orientation's most famous example there are benefits to the introduction of aspects woven within their targets, a study of the broader utility of the aspect-oriented metaprogramming approach should be conducted.

- Traditional aspects cannot make decisions based on reflection on specific properties of the code they are being applied to, such as calculations made, data accessed, computational complexity, and so on. There are many scenarios where one can anticipate this reflective behaviour to be useful. For example, compilation of Python code for efficient performance on a GPU (redirected from the CPU), as in cite the thesis here of Jeremy's student who worked in Python. He was very nice — can't remember his name for the life of me., seems to decouple from the rewritten logic nicely in concept, but relies on an examination of iteration logic and specifics of the code being recompiled. Aspects with metaprogramming directly support reflection as access to the target's AST is trivial to achieve. There are many possible applications for this technology, and likely in a diverse set of domains; there is therefore research to be done to demonstrate the utility of this new approach.

- Would be nice to have a third reason as to why PyDySoFu introduces new research possibilities.

7.2 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, many analyses are yet to be explored:

- Why were games abandoned? Are there patterns that can be identified which lead players to abandon games?
- Players likely formed cliques, where they would play against people they knew (perhaps in person) rather than relying on RPGLite's matchmaking features to find new opponents. The existence of cliques of players may have implications for the playstyles of players, how they learned "better" strategies over time, and the players' general dedication to playing RPGLite (and therefore producing a greater wealth of data for analysis and dissemination to the community)
- RPGLite's dataset contains information about players' interactions with the application itself; as the game made available some features typical of modern games (leaderboards, matchmaking, achievements, graphical customisation), an analysis of the features most commonly used can shed light on the more effective aspects of modern game design in both the general playerbase and more dedicated players[1]
- RPGLite's playerbase was recruited informally and there is scope for a larger and longer-term data collection effort to be made. A re-release of the application in major mobile app stores with a concerted effort to release new seasons of the game and maintain player interest for an extended period of time — perhaps 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 inherant in RPGLite's design) would enable a richer analysis, and broader utility to the games research community.

While investigations into these questions warrant further study, they remain outwith the scope of this thesis, which focuses on simulation technologies more than it does game design. There are many opportunities available for the game design research community to investigate. Publications in the

[1] For example, are some features heavily used, but only by a dedicated subset? Do all players use other features a moderate amount, showing mild but general appeal?

field from co-creators of RPGLite reflect further on the design and future improvements of the game; see cite William's PhD here.

7.3 Future Work pertaining to Aspect-Oriented Simulation

Aspect-orientation's goal of separation of concerns, and the possibility of using its trait of obliviousness to augment naive models in ways the original creator did not anticipate, presents research opportunities that are also outwith the scope of this thesis.

- Myriad models exist which have been provided accurate results in past research, but could not account for unforseen modern situations. For example, models of world health over time could not account for the Covid19 pandemic, and models of the world economy could not incorporate real-world data from the recession caused by responses to the pandemic, or the 2008 financial crisis. The World3 model is an example of one which has provided accurate predictions for decades, but could not account for incidents in modern times when constructed. Models such as these work from prior data which requires some adjustment as simulated time progresses to account for events of a large enough scale to disrupt their simulated system (here, global population, industry, food, resources, and pollution). An alternative to adjusting the models directly — adding cases at the relevant points in time to introduce "blips" in simulated data — is to construct aspects which represent global events such as pandemics, economic crises, and others such as war or famine. These can be modelled on real-world data, which we have shown in this research to produce realistic simulations. A proof-of-concept of the approach as applied to pre-existing models would start this work, and the augmentation of existing models to improve their accuracy can follow.

- Relatedly, aspects can represent anticipated future states so as to model their potential impact without modifying a known-good model of the world today. Future health and economic crises can be constructed as prospective changes to a model in an aspect, and applied to investigate the possible effects. A potential benefit of this approach as opposed to the simple modification of an existing model would be that many potential crises can be applied in any combination. For example, 10 aspects representing unpredictable future events yield 1024 possible combinations:

there are 2^{10} possible combinations of these aspects being applied or omitted from an execution of a simulation. Work to develop aspect-oriented models of speculative futures therefore gives an exponential number of predicted futures, which one could analyse to predict possible future trends. With a successful proof-of-concept of the augmentation of existing models to represent past events, this further step could anticipate future events and take advantage of aspect orientation's unique properties as a tool for simulation and modelling.

→ Combinations of traits in human modelling where a player might not fit to one specific trait well, but fits to a combination applied weakly.

7.4 Discussion

This section is not intentionally left blank.

Bibliography

[+06] hridesh rajan hridesh et al. *nu: towards an aspectoriented invocation mechanism*. Tech. rep. technical report 414, iowa state university, department of computer science, 2006.

[BH02] Jason Baker and Wilson C. Hsieh. "Maya: Multiple-Dispatch Syntax Extension in Java". In: *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*. PLDI '02. Berlin, Germany: Association for Computing Machinery, 2002, pp. 270–281. ISBN: 1581134630. DOI: 10.1145/512529.512562. URL: https://doi.org/10.1145/512529.512562.

[CBB19] 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).

[Cla73] R Lawrence Clark. "LINGUISTIC CONTRIBUTION TO GOTO-LESS PROGRAMMING". In: *Datamation* 19.12 (1973), pp. 62–63.

[con16] RafeKettler & github contributors. *magicmethods (A guide to Python's magic methods)*. https://github.com/RafeKettler/magicmethods/tree/65cc4a7bf4e72ba18df1ad17a377d 2011 — 2016.

[con21] Clarete & github contributors. *ForbiddenFruit*. https://web.archive.org/web/20210515092416/https://github.com/clarete/forbiddenfruit. 2021.

[CS04] Ruzanna Chitchyan and Ian Sommerville. "Comparing dynamic AO systems". In: 2004.

[CSS04] Constantinos Constantinides, Therapon Skotiniotis, and Maximilian Stoerzer. "AOP Considered Harmful". In: *In Proceedings of European Interactive Workshop on Aspects in Software (EIWAS)*. 2004.

[DD82] Edsger W Dijkstra and Edsger W Dijkstra. "On the role of scientific thought". In: *Selected writings on computing: a personal perspective* (1982), pp. 60–66.

[dev21] SimPy developers. *SimPy 4.0.1 documentation*. https://simpy.readthedocs.io/en/4.0.1/. 2021.

[Dij68] Edsger W Dijkstra. "Letters to the editor: go to statement considered harmful". In: *Communications of the ACM* 11.3 (1968), pp. 147–148.

[DR10] 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.

[GK99] 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.

[GS04] Wasif Gilani and Olaf Spinczyk. "A family of aspect dynamic weavers". In: *Dynamic Aspects Workshop (DAW04)*. 2004.

[Hil+00] 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.

[JS07] 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.

[Kic+01] G. Kiczales et al. "An Overview of AspectJ". In: *ECOOP*. 2001.

[Kic+97] Gregor Kiczales et al. "Aspect-oriented programming". In: *European conference on object-oriented programming*. Springer. 1997, pp. 220–242.

[KM20] 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.

[KWM20] William Kavanagh, Tom Wallis, and Alice Miller. "RPGLite player data and lookup tables". In: (2020).

[Mat08] 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.

[MK03] Hidehiko Masuhara and Gregor Kiczales. "A modeling framework for aspect-oriented mechanisms". In: *Proceeding ECOOP'03*. 2003.

[Nie77] 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.

[PAG03] 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.

[Par72] 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.

[PGA02] 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.

[Prz10] Adam Przybylek. "What is Wrong with AOP?" In: *ICSOFT (2)*. Citeseer. 2010, pp. 125–130.

[SL07] Olaf Spinczyk and Daniel Lohmann. "The design and implementation of AspectC++". In: *Knowledge-Based Systems* 20.7 (2007), pp. 636–651.

[Ste06] 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.

[SW16] Tim Storer and William Wallis. *FUZZI-MOSS*. https://web.archive.org/web/20201018121800/https://github.com/twsswt/fuzzi-moss. 2016.

[Tha+17] 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.

[VD09] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009. ISBN: 1441412697.

[Wal+21] William Wallis et al. "Designing a mobile game to generate player data - lessons learned THIS IS A DUPLICATE FIXME THIS IS A DUPLICATE FIXMETHIS IS A DUPLICATE FIXMETHIS IS A DUPLICATE FIXME THIS IS A DUPLICATE FIXME THIS IS A DUPLICATE FIXME THIS IS A DUPLICATE FIXME". In: *CoRR* abs/2101.07144 (2021). arXiv: 2101.07144. URL: https://arxiv.org/abs/2101.07144.

[WS18a] 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.

[WS18b] 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.