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

Chapter 1

Introduction

Some good things here

also some text

1.1 The first section header

this thesis has many features, such as content and citations, such as Wallis et al. [2020]

Chapter 2

Relevant Literature

Some notes here on what this chapter contains, and a brief description of its layout. Two snappy paras.

Some notes he
and a brief de
paras.

2.1 Early work on PyDySoFu

PyDySoFu[1] is a Python library **??** built for altering the source of a Python function as it is called, and before it is executed, while the original function definition remains oblivious to the changes being made. It was originally developed as an honours-level dissertation, which was built upon and detailled in a subsequent paper **??**. This thesis furthers that original work. To be clear about the work this thesis contains, the state of the project *before* this work began is briefly discussed here.

The goals of "changing a function's behaviour" and maintaining "obliviousness" in the original definition of that function speak to the goals of the aspect oriented programming paradigm**??**. Quoting their original definitions:

> Components are properties of a system, for which the implementation can be cleanly encapsulated in a generalized procedure. Aspects are properties for which the implementation cannot be cleanly encapsulated in a generalized procedure. Aspects and cross-cut components cross-cut each other in a system's implementation. […] The key difference between AOP and other approaches is that AOP provides component and aspect languages with different abstraction and composition mechanisms.

[1] Or "PDSF" for short.

Generally, aspect orientation is percieved to be a technique for separation of concerns. Any cross-cutting concerns can be separated from their components into aspects applied where that concern arises. The strength of aspect orientation lies in its compositional nature: developers can write short, maintainable implementations of a procedure's core purpose (for example, business logic) and ancillary concerns such as logging or security can be woven into this implementation as preprocessing, compilation, or at runtime. This compositional nature is what gives rise to aspect orientation's "obliviousness", as the procedure targetted by a piece of advice is written without regard to that fact.

The original PyDySoFu implementation was an aspect orientation library focusing on separating a function's definition from *potential changes to it*. This was used to model "contingent behaviour" — behaviour sensitive to some condition — as an original, "idealised" definition of that behaviour, plus some possible alterations. These changes might apply to many different behaviours in the same manner, and therefore represent concerns which separate cleanly into an aspect. An example would be the behaviour of a worker whose job requires focus on allocated tasks. A lack of focus could be represented as steps of the worker's tasks being executed in duplicate, out-of-order, or skipped. Assuming aspects as described by Kiczales et al. are able to edit the definition or execution of a procedure[2], such contingent behaviours are well modelled as aspects.

To achieve this, a model was presented in **??** wherein aspects were developed which could change function *definitions* on each invocation of that function, contingent on program state. This allowed behavioural adaptation to be simulated in an aspect-oriented fashion. In addition, a library of behavioural adaptations called FUZZI-MOSS was developed which implemented many cross-cutting, contingent behaviours in procedural simulations of sociotechnical systems.

One important contribution of this work is that PDSF aspects are effectively able to operate *inside* a target. In typical aspect orientation frameworks such as AspectJ**??**, aspects operate by effectively prepending or appending work to a target, referred to as "before" or "after" pointcuts respectively. To do both is referred to as "around". By manipulating procedures within Python directly, PDSF is able to manipulate its target from a new perspective, adding (or removing) work during the target's execution[3]. Moreover, because weaving is performed dynamically, every execution of a function may perform different operations.

[2] As opposed to simply wrapping it with additional behaviour before and/or after execution
[3] Similarly to **??**, but in an aspect oriented manner.

2.1.1 Opportunities presented by PDSF

PDSF presented several oppportunities for future research. Some salient properties of the original work include:

-

Do we need a tation is here?

2.2 Dynamism in AOP

Aspect orientation frameworks have supported "dynamic behaviour" in different ways for a long time. This is largely through a technique referred to as dynamic– or runtime–weaving.

2.2.1 Dynamic and static weaving

Dynamic weaving integrates advice into a target program during its execution, as opposed to during compilation or a pre-processing step. The advantage of this is flexibility: dynamic aspect-oriented approaches have been proposed for deploying hotfixes in safety-critical scenarios where software systems cannot be taken offline to apply patches, and in adaptive mobile scenarios where software may need to alter its properties in response to its environment**??**, or when debugging code to apply potential patches without reloading an entire software system**??**.

CITECITECIT

To meet these needs, software systems need to check for available aspects to weave at any join point, as it is always possible that the set of applied advice has changed since the program last encountered this point. The technique therefore presents a tradeoff compared to traditional (static) aspect weaving, as illustrated in **??**. Chitchyan and Sommerville generalise this tradeoff by describing different mechanisms used to implement aspect orientation into three main categories[4], each with their own strengths:

"Total hook weaving" alters all join points where advice may be applied before runtime, so that during execution each join point "watches" for applied advice. The benefit of this approach is that aspects can be applied at any point at runtime, but this flexibility is bought at the cost of maximum overhead: at all points where weaving *may* be possible, checks for applied advice must be made.

[4]Drawing from **????** where "PROSE", a particularly influential dynamic aspect orientation library, is detailled.

"Actual hook weaving" weaves hooks only to join points that are expected to be in use. This limits overhead from watching for applied advice, at the cost of flexibility: during program execution, advice may be applied or retracted *only at specific points within the system*.

"Collected weaving" weaves aspects directly into code at compilation / preprocessing, so as to collect advice and target codebase into a single unit. This provides exactly the necessary amount of overhead, and in many cases may result in requiring no "watching" for applied advice at all, but this limits a developer's ability to amend advice supplied at runtime.

There is an almost direct tradeoff between the number of potential join points actively checking for applied advice at runtime, and the overhead of dynamism in any aspect oriented framework, with "total hook weaving" providing complete adaptability at the expense of checking at all possible points whether advice is applied.

Another tradeoff could be seen to be the clarity of dynamically woven aspect oriented code.

2.2.2 PROSE

One implementation of dynamic weaving is PROSE**????**, 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 et al. provide in **??** indicates that the performance issues generalised by Chitchyan and Sommerville in **??** 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*.

2.2.3 Handi-Wrap

Handi-Wrap?? is a Java library allowing for dynamic weaving via a third-party language designed for metaprogramming, called Maya. At the time of development Handi-Wrap's dynamic aspect weaving feature was novel: the aspect orientation library of note, AspectJ, wove only statically[5], and Handi-Wrap's purpose was to show that DSLs for metaprogramming could pave a way to dynamic weaving.

Do I want a ci
worth revisitin

Baker and Hsieh implemented an aspect orientation framework which is reasonably performant, weaves dynamically, and allows for aspect orientation features to be implemented natively for greater control as compared to Handi-Wrap's then competitor, AspectJ. As a tool, Handi-Wrap demonstrated a promising approach to dynamic weaving, but the project appears to have enjoyed less attention than similar work (such as PROSE, described in section 2.2.2).

The technique used to implement Handi-Wrap (implementation via a metaprogramming-specific DSL, Maya) is familiar, in that it shares a perspective on dynamic weaving with early PyDySoFu work. The fuzzers used in ?? applied transformations to abstract syntax trees, not unlike a LISP-style macro. Quoting ?? by way of contrasting, ``Maya generalizes macro systems by treating grammar productions as generic functions…'' . The two approaches have clear differences. Most notably, PyDySoFu's entire implementation *and use* is performed in Python directly, and Maya's intended purpose is metaprogramming in a more general sense. It is possible that, while Maya provided a useful foundation to explore the dynamic weaving of aspects, its lack of adoption as a language limited handi-wrap's reach; nevertheless, it is encouraging to see another use of metaprogramming for weaving aspects at runtime.

2.2.4 BCA

Binary Component AdaptationKeller and Hölzle [1998] (BCA) is a technique for performing adaptations on software components after compilation. Though it works on already-compiled code it does provide dynamic behaviour: the technique can adapt software components via rewriting before or during the loading of its target. Like some aspect orientation techniques, BCA adapts a Java class loader to make its adaptations, but unlike aspect oriented approaches it does not require access to the original source of the software. For scientific simulation purposes, it could therefore be appealing in

which?!

[5] AspectJ now supports what it calls "load-time weaving" — that is, weaving aspects as classes are loaded into the JVM — but not weaving to things that are *already* loaded, meaning AspectJ still allows for only a particular flavour of dynamic behaviour.

situations where adaptations are made to another researcher's simulations — assuming the original source code is not published — or in security settings investigating trust in compilers and runtimes**??**. In the present context of developing sociotechnical simulations however, this does not appear to be an advantage, particularly at a time when the source code of software components of research projects are more routinely published.

2.3 Aspect Orientation & Modelling

2.4 Aspect Orientation & Simulation

Surprisingly little literature exists pertaining to aspect-oriented simulation. Aspect orientation is often applied in the context of *modelling*, composing together a perspective of the world which isn't necessarily executable or able to produce data.

Early in the history of aspect orientation as an emerging paradigm, there was some interest in its use for scientific simulation. **??** discuss that computer simulations require code for both observation of a simulation and the simulation itself, and that misuse of this could cause what is in effect a kind of Hawthorne Effect, where the inclusion of observation code intertwined with simuilation code might influence the outcome of an experiment. They suggest that improving simulation technologies could combat this approach. Aspect Orientation, being developed specifically with obliviousness in mind, is an ideal candidate which Gulyás and Kozsik identify.

Much of the literature concerning aspect-oriented programming and simulation focus on tooling support for aspect-oriented simulation, rather than investigations into its efficacy. For example, attempts have been made to integrate aspect orientation into new tools **??** , or into existing ones **??**.

Some experiments specifically using aspect orientation in the implementation of process-based simulations also exist**??** . For example, Ionescu et al. apply aspect orientation in a nuclear disaster prevention simulation. Their motivation is that code can become complex to maintain over time and changes to the scientific zeitgeist or to regulatory requirements become costly as technical debt mounts. Aspect orientation therefore allows developers to separate functionality into distinct modules more easily, without disturbing the underlying codebase.

2.5 Dynamism in Simulation and Modelling

2.6 Opportunities

Bibliography

Jason Baker and Wilson Hsieh. Runtime aspect weaving through metaprogramming. In *Proceedings of the 1st international conference on Aspect-oriented software development - AOSD '02*. ACM Press, 2002. doi: 10.1145/508386.508396. URL https://doi.org/10.1145%2F508386.508396.

Ruzanna Chitchyan and Ian Sommerville. Comparing dynamic ao systems. In *Proceedings of the 2004 Dynamic Aspects Workshop (DAW04)*, pages 120–134. Citeseer, 2004.

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.

Tudor B. Ionescu, Andreas Piater, Walter Scheuermann, Eckart Laurien, and Alexandru Iosup. An aspect-oriented approach for disaster prevention simulation workflows on supercomputers, clusters, and grids. In *2009 13th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications*. IEEE, 2009. doi: 10.1109/ds-rt.2009.35. URL https://doi.org/10.1109%2Fds-rt.2009.35.

Ralph Keller and Urs Hölzle. Binary component adaptation. In *European Conference on Object-Oriented Programming*, pages 307–329. Springer, 1998.

Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European conference on object-oriented programming*, pages 220–242. Springer, 1997.

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*, AOSD '03, page 100–109, New York, NY, USA, 2003. Association for Computing Machinery. ISBN 1581136609. doi: 10.1145/643603.643614. URL https://doi.org/10.1145/643603.643614.

William Wallis, William Kavanagh, Alice Miller, and Tim Storer. Designing a mobile game to generate player data-lessons learned. 2020.