THE UNIVERSITY OF HULL

Middle-Out Domain-Specific Aspect Languages and Their Application in Agent-Based Modelling Runtime Inspection

being a thesis submitted for the Degree of
Doctor of Philosophy
in the University of Hull

by

Craig Ashley Maddra, BSc(Hons)

School of Engineering and Computer Science

April 2019

Abstract

Domain-Specific Aspect Languages (DSALs) are a valuable tool for separating cross-cutting concerns, particularly within fields with endemic cross-cutting practices. Agent-Based Modelling (ABM) runtime inspection, which cuts across the core concern of model development, serves as a prime example. Despite their usefulness, DSALs face multiple adoption issues: the literature regarding their development and use is incohesive, coupling to a weave target hinders re-use, and available tooling is immature compared to Domain-Specific Languages (DSLs). We believe these issues can be aided by furthering DSL middle-out techniques for DSALs.

We first define the background of what a DSAL is and how they may be used, moving onto how we can use DSL techniques to further DSALs. We develop a middle-out semantic model approach for developing domain-level DSALs with transparent aspect orientation using adaptions of DSL techniques. We have implemented the approach for model-specific DSALs for the in-house framework Animaux, and as middleware-specific DSAL for agent messages in the JADE framework, which can be specialised to models using extension DSALs. We give illustrative result cases using our implementations to provide a base of the user development costs and performance of this approach.

In conclusion, we believe the adoption of these technologies aids ABM applications and encourage future work in similar fields. This thesis has given a base philosophy toward DSLs, a novel approach for the development of middle-out DSALs and illustrative cases of this approach.

Acknowledgements

I would like to thank my supervisor Professor Ken Hawick for introducing me to the world of research, and more specifically the topic of domain-specific languages. I would also like to thank the members of my supervisory panel: Dr Nina Dethlefs, Dr Martin Walker and Dr Mike Brayshaw, for their guidance throughout the creation of this thesis.

I am also grateful to all those in the School of Engineering and Computer Science, who have provided me with the facilities and community to make my research possible. A special thank you to my former Swale house peers for forming a great research group, and more importantly, a great friendship group.

On a work-related note; thanks to Dr James Walker, Nick Fletcher and Phininder Balaghan for being 'the team' in my first research assistant post, and subsequently, Professor John Murray for joining phase two of the same project.

Last but not least, the support I have received from my family throughout the time I have spent at the University of Hull has been invaluable, thank you most of all.

Contents

List of Acronyms

ABM Agent-Based Modelling

AOP Aspect-Oriented Programming

DSAL Domain-Specific Aspect Language

DSJP Domain-Specific Join Point

DSL Domain-Specific Language

FIPA Foundation for Intelligent Physical Agents

GPAL General-Purpose Aspect Language

GPJP General-Purpose Join Point

GPL General-Purpose Language

IDE Integrated Development Environment

JP Join Point

OOP Object-Oriented Programming

List of Figures

List of Listings

List of Tables

Chapter 1

Introduction

This chapter introduces our motivating problem statement, research aims and contributions. Our research aims are defined through a set of core hypotheses, and each chapter has an associated question which directs its purpose.

1.1 Problem Statement

Computational science uses software as scientific apparatus to perform and observe experiments 'in silico'. Through this thesis, we focus on Agent-Based Modelling (ABM), a popular computational science tool used to simulate the actions and interactions of autonomous agents with the intention of assessing the system as a whole. The field relies on many agents with simple rules causing complex behaviour at the macro scale to re-create or predict the appearance of complex phenomena in the modelled system.

The use of ABM programs as scientific apparatus brings forth the software concern of runtime inspection of models for experimentation. The observing code must be reliable to retain scientific integrity, malleable to a range of experiments and maintainable for the scientist. In an ideal scenario control of inspection will be separate from the model code.

Using conventional object-oriented methodologies, runtime inspection becomes a cross-cutting concern during ABM development. A cross-cutting concern is a concern which cannot be modularised using a program's dominant decomposition method. In practice this results in inspection code scattered and tangled throughout solutions with a large amount of shared code across experiments and even models.

This shared code is referred to as boilerplate code, a phrase used across many industries referring to unoriginal, repetitive text which is used as a filler for larger documents. The use of libraries, frameworks and Domain-Specific Languages (DSLs) can provide a division of labour between a core implementation of 'that type of program' and specific variants of it by allowing the re-use of boilerplate code across implementations by packaging it into components or notations. We specifically focus on the DSL approach to reducing boilerplate through this thesis.

This leaves the problem of scattering and tangling of the cross-cutting runtime inspection code, which is challenged by the software decomposition method Aspect-Oriented Programming (AOP). AOP allows for cross-cutting concerns to be separated into aspects. These aspects can then be controlled and developed separately to the core concerns of a system.

Domain-Specific Aspect Languages (DSALs) are aspect-oriented DSLs which allow the expression of cross-cutting concerns using domain-specific abstractions. A DSAL provides domain-specific abstractions to the representation of, identification of

or effect at a set of Join Points (JPs) in the runtime of a program. Proper adaption of DSALs for the runtime inspection of ABMs could allow ideal separation of concerns with domain expert level code for writing experiments.

DSAL adoption is hampered, firstly because of a lack of knowledge of the combination of the domain-specific and aspect-oriented paradigms. Secondly, DSALs are more difficult to develop than a standard DSL because of the addition of an external weaving target. This difficulty is exacerbated by the inability to use an off-the-shelf DSAL for problems as is commonly possible with a DSL. Where a General-Purpose Aspect Language (GPAL) is general to all programs written in a certain language, a DSAL is not meaning small differences in weave target can lead to incompatibility for weaving or unsuitability of notation.

We believe that the use of DSALs is becoming more important as the software complexity of systems is rising, and computational time is becoming a readily available commodity to perform these 'in-silico' experiments with. As such, the bottleneck for scientific development is the scientist's productivity rather than the computer's execution times. This problem has been addressed in part by movements such as the research software engineer position, where a software engineer supports a scientist's coding work. The creation of notations such as DSLs and DSALs is extremely useful in this area because it allows clear separation of simulation implementation expert and domain expert roles.

This thesis aims to present the philosophies of these paradigms to aid their informed use, present a development methodology for DSALs which allows proper division of labour between implementation and use of model experiments and give implementations using these methods with quantitative results.

The summary of our problem statement is:

DSALs are a suitable tool for ABM runtime inspection yet they have a high barrier to entry. This is because custom-built implementations are generally required while their implementation difficulty is higher than that of a DSL compounded by a lack of knowledge about their implementation patterns. Are there ways to practically improve the productivity of development and use of custom DSALs for ABM runtime inspection?

1.2 Objectives of the Thesis

We now discuss the objectives of this thesis in terms of questions and contributions.

We can define our thesis statement as:

DSL middle-out techniques can be adapted to create domain expert level DSALs with transparent aspect orientation. This technique reduces the barrier to entry for using a mature method of reducing scatter and tangle using domain-level code.

The driving hypothesis which we visit throughout the thesis are:

- Applying a middle-out approach utilizing a semantic model to DSAL development and use allows separation between domain-application and weaving implementation. This separation allows multiple sets of aspects, with differing weaving implementations to be written using the same middle layer across multiple instances of one or more models.

- Using custom-built middle-out DSALs allows sets of runtime inspection experiments to be run on ABMs with reduced scatter, tangle and boilerplate code compared to inline object-oriented methods.
- A direct GPAL implementation of Domain-Specific Join Points (DSJPs) will provide similar to inline performance when there is a direct mapping to DSJP available, yet poor performance when not. Dependency injection of General-Purpose Join Points (GPJPs) into target code may be used to allow better performance, with the disadvantage of more scattering throughout the base code.

1.3 Thesis Structure and Contributions

This chapter has presented our problem topic, research questions and hypothesis. The rest of the thesis is laid out as chapters which have accompanying questions and contributions.

Chapter 2: Foundations

This chapter presents the background of our work and the definitions which we use throughout this thesis. The main contribution to be found in this chapter is our definition of what a DSL is.

The question of this chapter is:
How can we define and view DSLs, AOP, DSALs and ABM through this thesis?

This question grounds our research, giving clear meaning to our intended direction. Grounding is necessary because accepted definitions of these terms are not consistent throughout the literature, and our definitions make this thesis a contained entity.

Chapter 3: Towards Modularised Runtime Inspection of Agent-Based Models

This chapter reviews the literature around our problem statement and the literature around middle-out DSALs. This literature review sets the scope of our work and leads us to our two main contribution chapters.

The question of this chapter is:
What difficulties face ABM runtime inspection which are reduced by the use of a DSAL, what are the barriers to DSAL adoption and what existing research has challenged these barriers?

This question sets up our initial literature review of research towards allowing the profitable implementation of DSALs, then brings us to our research direction of a novel approach to implementing DSALs for ABM runtime inspection which forms the rest of the thesis. We specifically focus towards improvements of DSALs by moving closer to the DSL experience because this is a well-grounded and practical method of improvement.

Chapter 4: AnimauxRI - Runtime Inspection Targeting Specific Models

This chapter is our first investigation into our middle-out DSAL with a semantic model concept. The chapter focuses on Sugarscape and Kawasaki models within our in-house framework called Animaux. The main contribution to be found in this chapter is the development of a semantic model based middle-out process for DSALs.

The question of this chapter is:
How can we implement a middle-out DSAL for specific models without domain-oriented interfaces?

This question brings the original work into defining a novel method for developing middle-out DSALs for a set of pre-defined models within the Animaux framework. This work then becomes the basis for the next chapter's research.

Chapter 5: JADERI - Runtime Inspection Targeting Middleware

This chapter follows from the previous, furthering the concept of a middle-out DSAL with a semantic model by allowing the use of a core DSAL across a generic cross-cutting concern which can then be extended using an internal DSAL for specific models without re-implementation of weave logic. The main contribution to be found in this chapter is the laying of middle-out processes together to allow for specific DSAL development without re-implementing an aspect orientated bottom layer.

The question of this chapter is:
How would we implement a middle-out DSAL for a generic cross-cutting concern within a framework using domain-oriented interfaces, which could then be specialised to specific models without re-implementing weave logic?

This question forwards the previous questions contribution by applying the middle-out concept to allowing multiple extension DSALs to be created from a core DSAL which performs the weaving logic. This moves the development of model-specific DSAL tasks closer to a DSL developer rather than an aspect-oriented language developer.

Chapter 6: Synthesis of AnimauxRI and JADERI

This chapter discusses, contrasts and compares the approaches taken in AnimauxRI and JADERI. This states that the base approach underpinning both chapters is the same, and choices made in either chapter and may be applicable to the other. This chapter is an especially useful discussion for choosing language type and weaving target specificity forwarding future research within this area.

Chapter 7: Conclusion

This chapter discusses our work with relation to our original aims, hypotheses and contributions set out in this chapter. We conclude with a discussion of future research.

Chapter 2

Foundations

This chapter lays out the background of the four main fields which underpin our research, giving us a clear framework to base the rest of the thesis.

The primary research question answered by this chapter is:

How can we define and view Domain-Specific Languages (DSLs), Aspect-Oriented Programming (AOP), Domain-Specific Aspect Languages (DSALs) and Agent-Based Modelling (ABM) through this thesis?

2.1 Introduction to Domain-Specific Languages

Firstly, we will introduce the foundations of the base research area throughout this thesis. We use domain-specific techniques throughout the thesis, and our definitions of why we use DSLs show the motivations for our work.

2.1.1 Motivation

DSLs have been popular since the beginning of computer science, often without people even recognising they are writing a DSL. The classical use case of a DSL can be shown by one of the many small Unix languages such as grep or awk performing a single small but common task as is discussed in Bentley (1986), which calls on people to treat these interfaces as languages. This stems from the idea of reducing boilerplate code in tasks, especially tasks which are performed often and by many people as the biggest time savers available to us are the things which we do very often. This reduces the burden of the large start-up costs of creating a language because many uses benefit. The application domain in question is also a relevant issue, with different tasks being more amenable to domain-specific benefits (Sprinkle et al., 2009). In Unix times a prime example was text processing, meaning DSLs such as sh and bash became the primary user interface of a computer. This benefit was further forwarded by writing programming languages which generate code of other DSLs when working in very specific domains such as typesetting graphs as in GRAP which generated PIC code to outsource image creation (Bentley and Kernighan, 1986). The creation of DSLs has been forwarded as a worthy design pattern by Gamma et al. (1995).

In a modern context, DSLs are used within projects as extensions for what general-purpose Object-Oriented Programming (OOP) cannot do well. Some are stand-alone languages bundled as libraries which take a string as input for a standard task, such as regular expressions; some are extensions within a language allowing domain-specific input style, such as LINQ in C#; some are sets of languages with vast frameworks allowing full projects to be based around them, such as in the RePast suite.

One of the most important features of a DSL is removing boilerplate from what the user has to write or read. The expression boilerplate is borrowed from the legal profession where it refers to statements which are very often present and can be used as the basis to create a new legal document. This leverage of known boilerplate code in areas and its automatic generation from known language context allows for greater efficiency in areas which have been dealt with before. The downside of this approach is the opposite of the benefit; the generated code is hidden from the programmer meaning they may be mistakes in the code they cannot be aware of or could only be aware of from reading the documentation on the semantics of the language if such documentation exists. This movement of responsibility from the programmer to the implementation of the language must be taken into account when choosing to use a DSL, mature DSL software is more likely to have comprehensive documentation than side hobby project languages for example. The process of using a DSL to reduce the creation of boilerplate code for a problem can be explained through its mapping from problem to solution space as shown in Figure 2.1. A DSL is generally implemented as a problem-oriented language which has a mapping to a solution-oriented language, this solution-oriented language is generally a General-Purpose Language (GPL) which in turn mappings towards actual implementation. The choice of implementing a DSL through an intermediary GPL or hosting the DSL within a GPL is generally taken because of the quality of mapping between these languages and many implementation targets. In the same way that DSLs provide suitable mappings for a domain, GPLs provide suitable mappings for a virtual architecture to many physical architectures. The use of GPLs as implementation targets for a DSL reduces the workload required for implementing the DSL because cross-platform implementations are provided by the general-purpose base.

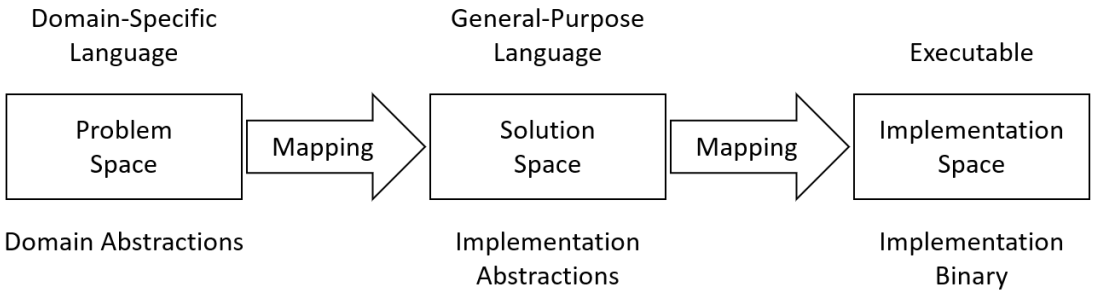

Figure 2.1: The mapping of domain-specific code to machine implementation inspired by Czarnecki and Eisenecker (2000).

We can explain this concept of mapping from problem space to solution space by comparing programming languages to transport maps. Transport maps are abstractions over an environment to allow planning of movement through an environment at speed and ease greater than traversing the environment physically. Much like maps, high-level programming languages are abstractions over a computing environment to allow the expression of programs with benefits such as speed of writing, safety, useful features not available out of the box at machine level. Machine level programming languages are generally explained as representations of computer operations, although even these operations are becoming more and more complex mappings to even lower level operations. Higher level languages allow us to traverse through many of these operations at an abstraction level of some higher-level virtual architecture such as Java or C rather than a specific machine architecture. This is much like standard roads maps allow journey planing at the abstraction level of routing and distances without knowing the specifics of implementing a drive through that route, which is not appropriate to consider thoroughly at planning time. Domain-specific high-level languages allow us to traverse through specific problems at the task level rather than requiring implementation details. This is shown in transport maps by how train route maps sacrifice geographical accuracy

for clear, complete maps of stops because the passengers are trying to decide which train they need to be on and for how many stops rather than see the geographical 'implementation' of this.

2.1.2 What is a DSL

Given this motivation, we now investigate what constitutes a DSL. We place emphasis on this because we believe it is important to ground our work around a solid philosophical framework. We will begin by looking through the literature definitions of DSL, then moving into the framework we follow in this work. We include similar terms in our umbrella of DSL such as little languages and application-oriented languages as well as field-specific names such as probabilistic programming languages which are DSLs for machine learning.

There are two main schools of defining DSLs within the literature. By far the most popular method is through a list of necessary and sufficient features as in classical categorisation such as domain focus, limits on expressiveness and language fluency as found in Thibault et al. (1999); Mernik et al. (2005); Michaelson (2016). A secondary method generally found through interdisciplinary research is through qualitative assertions about the elegance or productivity boost from DSLs as found in Hawick (2011, 2013). These two methods share complimentary problems where the objective focus of classical categorisation does not translate well into real life as it causes fuzzy edges, and the qualitative assertions while generally uncontroversial are also generally without proof or elaboration of the claims.

These types of definitions are well suited for scientific articles where people are already familiar with the concept of a DSL, and there is limited space for background and research methodology of the work. For the purposes of this thesis, we believe these are not sufficient definitions and give the philosophy and framework for our work through the rest of this chapter.

The main contention points in the literature are around the definitions of the narrowness of the domain, exclusivity of the specificity and nature of the language. The question of how narrow a domain must originate with the first high-level languages such as FORTRAN, COBOL and LISP which had definite application areas but were sufficiently general to perform in many domains without issue. Sammet (1969) refers to these languages as problem-oriented languages because they are higher level than assembly code making them closer to the problem than the machine implementation, reserving application-oriented languages for those with facilities and notations which are useful primarily in a single application area. Sammet moves onto saying how this definition between problem and application is somewhat relative depending on the width of the application area in question. More recent work by Czarnecki and Eisenecker (2000) states that all programming languages have some degree of domain specificity, but there is some arbitrary cut-off point where a language becomes domain-specific. The question of how broad a domain can be while still being specific has recently became poignant for languages such as Ant. Ant is a software build automation language whose domain has gone through a rapid change since Ant's introduction in 2000 resulting in the Ant programming language 'sliding into generality' because of what programmers expect within the domain with the increased automation brought by integrated development environments and ubiquitous internet access Fowler (2010).

Given that we can decide what constitutes an acceptable domain size the next question is what does specific mean in this context. Does the language need to be exclusively used within a domain? If so using a DSL in a different domain would strip it of the DSL definition, especially so if the language was found to be Turing complete and thus the very definition of a general programming language. Taking the example of postscript which has a single definite use of page description, would we remove the DSL definition of it because of its Turing completeness? Fowler (2010) includes in his definition 'limited expressiveness focused on a particular domain', given this description can a Turing complete language be referred to have limited expression? Alternatively, are all programming languages of limited expressiveness because they cannot express solutions to non-computable problems? We believe this limited expression is a valid comment given the expected use a human can give to a language because Turing completeness is not the only factor in limiting programming languages. Furthermore, we believe the use of a DSL in a separate domain or coincidental usefulness in other domains does not strip the DSL of its domain specificity. Although we do not consider it impossible for a DSL to be Turing complete, especially when this Turing completeness is difficult to manage; we agree with the general best practices set out in Völter (2009) that external DSLs should avoid being Turing complete leaving Turing complete problems which require domain specificity for APIs and internal DSLs mixed with the host language.

The final common point of contention is: does a DSL have to be executable as is claimed in the seminal survey of DSL research by van Deursen et al. (2000), or as stated in Mernik et al. (2005) and Wile (2001) can they be specifications, definitions or descriptions for domain expert involvement which may or may not be to be programming languages. Völter (2010) brings the notion that DSLs must be

processable rather than executable which we believe is a better definition. Through the definitions in this thesis, we include these non-executable languages in our definition of DSLs.

To satisfy these contentions, we treat the categorisation of DSLs using prototype theory, a method of categorisation which is popular in cognitive psychology and linguistics. Prototype theory originated with Rosch (1978) building upon Wittgenstein's (1953) work on family resemblance. This moves away from necessary and sufficient feature classifications and towards how categories are used in the real world. This approach views prototypical categories as experiential rather than objective, envisaging a relationship between concepts and the world rather than the objective features of a category. The main reason we believe DSLs require an experiential classification rather than an objective set of features is the inherent subjectivity in the use of programming languages. Although a set of programming languages may be able to solve a problem, there are qualitative and quantitative differences between the resulting approaches when considered in relation to a human programmer. We believe DSLs exist in relation to the environment they are used in, while a DSL cannot offer more computational capability than a Turing complete language, it may offer many things in relation to the use of the language such as domain expert understanding, reduced lines of code and less boilerplate code required. It should be noted classical feature semantics is not replaced by prototype theory but augmented where necessary, for example, shapes such as triangles can be categorised exhaustively using classical feature semantics as 3 sided polygon with internal angles adding up to 180°. We will now explain why prototype theory is appropriate for classifying DSLs using the four common qualities of prototype theory research discussed in Geeraerts (2016).

- **Prototypical categories cannot be defined using a single set of necessary and sufficient attributes**

  As we have seen from definitions of DSLs from the literature, although the core of each definition alludes to the same base of improving software economics or involving domain experts there is contention around what features a DSL must or should have. This means that any given definition, somebody will come up with a counterexample. This could either be something included by the definition which others would not consider DSLs or something excluded by a definition which others do consider a DSL.

  Furthermore, these sets of features vary depending on the intent of use within the definitions field because the focus of using a DSL is as a tool through a process rather than the requirement of ticking boxes. This ties into work by Coleman and Kay (1981) where a thing or event is a psychological object or process rather than an objective set of necessary and sufficient conditions. We can illustrate this using the famous psychological experiment of categorising a vessel as a cup or a bowl depending on the prescribed use of an object from Labov (1973). This begins by showing images with a neutral framing context; when the same drawing of a cup is stretched horizontally alike in Figure 2.2 it becomes less likely people will refer to it as a cup, and once it has reached a certain threshold people begin referring to it as a bowl. Interestingly when framing context is added of the vessel being used for food the threshold of people beginning to categorise the vessel as a bowl requires far less horizontal stretching. Although this is an excellent illustration of how framing effects categorisation there are limitations to using laboratory experiments focused solely around the applicability of words as discussed by Wierzbicka (1985) who favours methodical introspection towards the structure of the concept which

Figure 2.2: Illustrations of vessels with scaling width similar to those used for the Labov (1973) experiment.

underlies and explains the applicability of a word. As such, our definition of a DSL should constitute the philosophy which is being driven forward through a DSL rather than descriptions of the features which happen because of this.

This is especially related to the variety of XML subset DSLs which are written in a general-purpose modelling language, yet only using a small subset of the features and self-enforcement of pseudo keywords the programmer could be classified as using a DSL because the use of the language is in line with the internal structure of how a DSL is intended to be used even though the language base which is used is objectively general-purpose.

- **Prototypical categories exhibit a family resemblance structure**

  The core idea of family resemblance comes from Wittgenstein (1953) where a rigid definition cannot be made because there's no element that everything in a category has in common, but they all share something and likely many things with other members of the category. This is in reference to the visual resemblance between members of a family; the children can exhibit many traits from either parent such as eyes, height, nose but the children themselves do not need to have any single characteristic in common. In regards to prototype theory Lakoff (1987) defines this sort of group as a radial category of clustered and overlapping meanings.

DSLs fit into this type of structure because a majority share the same major goals of improving software productivity and increasing domain expert involvement, although there are so many different methods of achieving this creating a list of necessary and sufficient features is an impractical task. Furthermore, as we have previously mentioned this method of categorisation would not sufficiently answer the question of what concepts experientially underlie and explain the applicability of DSLs (Wierzbicka, 1985).

- **Prototypical categories exhibit degrees of category membership**
  Progressing from a binary classical feature semantics, the members within a category are given a graded membership through the semantic features they hold, how they meet these features and the importance of these features in regards to the prototype of this category. The scale of which a graded membership if operated is defined operationally by people's judgements of the goodness of membership in the category, with the prototype being the clearest case of category membership (Rosch, 1978), for example, an owl may be a better representative of the bird category than a penguin is. The people involved in these judgements vary from discipline to discipline, and as such, different features will receive different weightings depending on the discipline involved. For example, DSL researchers may consider more languages DSLs than general software engineers because they are looking for research potential in features which are taken for granted by a general developer such as fluent interfaces on APIs.

  This phenomenon is informally mentioned throughout the DSL literature specifically as domain specificity being a matter of degree or the target of a language being somewhat relative Czarnecki and Eisenecker (2000); Sammet (1969). A concrete example of this is a fluent interface DSL such as the Python

family tree DSL in Maddra and Hawick (2016) may be given a weak grading in domain-specificity, when on the other hand an external DSL such as NetLogo (Wilensky, 1999) may be given a high grading of departing from traditional general-purpose code.

An example of an arbitrary communities grading of DSLs may be like is shown in Figure 2.3. The prototypical DSLs are languages such as SQL, regex and NetLogo because they have clear domains and no general-purpose features away from their domain. Moving away from this we have languages like HTML which may not be considered a 'programming language' but are very domain-specific. Ant and Bash are intended towards a domain, but their domains are large. Fluent interface DSLs may be called an API or a DSL depending on their use. Unity is a game engine rather than a language, although is used as a DSL for games with its scripting APIs so may be a borderline case which could be called a DSL in some contexts. On the other hand, the ABM framework MASON is used to reduce boilerplate just like Unity is although provides no scripting feature out of the box so we would not consider it a DSL, although extensions to Mason may be. Finally, languages such as FORTRAN and COBOL do have an intended domain although the domains are generic and general-purpose features are in the languages, so they would be considered general-purpose.

- **Prototypical categories are blurred at the edges**

  Finally, the concept of fuzzy edges arises because a member may lie in-between categories by sharing features of multiple categories. This causes blending of definitions that cannot be done in the strict sets given by a classical feature semantics. This allows for the creation of rich categories especially useful for areas which have evolved through time leaving interesting outliers to groups. An example of this could be categorising a mudskipper fish which possesses many

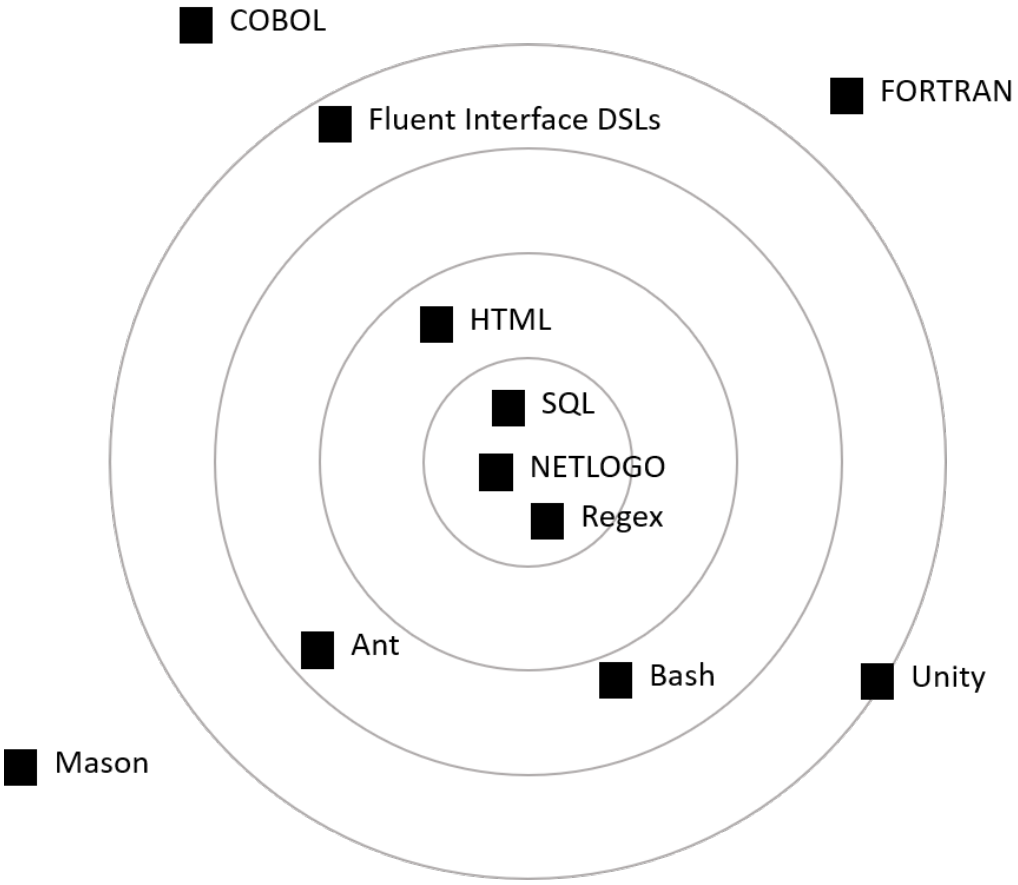

Figure 2.3: An example group of programming languages in an illustration of their graded membership to the category of DSL.

of the semantic features of a fish but also spends large portions of time outside of the water which would leave it out of many conventional fish definitions. The preciseness of the definitions we give using prototypes has been criticised by Hofstadter and Sander (2013) because the same fuzziness affecting the categories we discuss effects the words we use to define the semantic features of a category. For example, does domain-specific apply at language design, develop or use time? Moreover, when this does apply is it an exclusive specific or is the intent of specificity enough. The ambiguity of words used in the definitions is beyond the scope of this thesis, but we have mentioned it for completeness.

A DSL which is Turing complete may be considered a DSL or a GPL depending on the context of the classification. This is in contrast to a binary definition where domain-specific is a direct antonym to general-purpose, which we believe does not capture the underlying concept which gives real-world applicability to the term domain-specific. There is more to the definition of DSL than a forced specificity towards a domain, and as such, its categorisation should not depend upon this sole feature.

We primarily consider DSLs as tools for forwarding different fields, especially scientific fields by moving the level of abstraction to the most suitable for the task at hand. It is agreed throughout the literature that DSLs are a step past frameworks for software maturity (Mernik et al., 2005; Michaelson, 2016) and through this thesis, we treat them as such. We look to use DSLs as tools facilitating confluences between fields have library and framework technology in place, and furthering DSL techniques can aid the work (Walker, 2016). As such, our definition of DSL focuses on the potential to improve the economics of software development or improve domain expert involvement.

In closing, we have given this detailed definition as it is useful for presenting our research framework but through general use of DSLs we believe straightforward use of definitions alike that of the agent definition from Russell and Norvig (2003) where the DSL definition is meant to be a tool for forwarding programming language research and use, not as an absolute characterisation dividing DSLs and non-DSLs. In the same way, we could refer to a calculator as an agent, but it would not be interesting; we could remove Turing complete languages from the DSL definition, but it could remove a valid line of inquiry. A DSL is a programming language designed or used towards a domain-specific goal.

2.1.3 Purposes of DSLs and their Implementation

Now we have discussed what motivates a DSL use we move into the purpose and implementation of DSLs in the literature. In Fowler's (2010) DSL book, Brad Cross creates a separation between a computational and compositional DSL.

- Computational: performs actions on a semantic model. Generally, programming an alternative computing architecture such as a state machine.

  For example, JMock expectations are set up using a mix of Java and declarative JMock statements which manipulate the semantic model's expectations as shown in Listing 2.1.

Listing 2.1: Example JMock computational DSL code.

```
// create expectation
context.checking(new Expectations() {{
   oneOf (subscriber).receive(message);
}});

// perform task
publisher.publish(message);

// verify expectation is met
context.assertIsSatisfied();
```

- Compositional: describes composite structures. Generally, data types for a specific application such as user interfaces.

  For example, GEDCOM defines the individuals and relations between individuals in a family tree to form a graph structure as shown in Listing 2.2.

Listing 2.2: Example GEDCOM compositional DSL code.

```
0 @I1@ INDI
1 NAME /Last/ First
1 BIRT
2 DATE 1 Jan 1970
1 FAMC @F1@
1 FAMS @F2@
```

Through this thesis we use Fowler's (2010) definition of a semantic model as the backbone of DSL implementation. A semantic model is a representation of the subject the DSL describes, for example, an in-memory object model, state machine or database tables. In simple applications, a semantic model may be used as a domain model which executes or generates code, in more complex scenarios a domain model would be a framework for creating applications within a domain with the semantic model containing the data which drives specific implementations using this

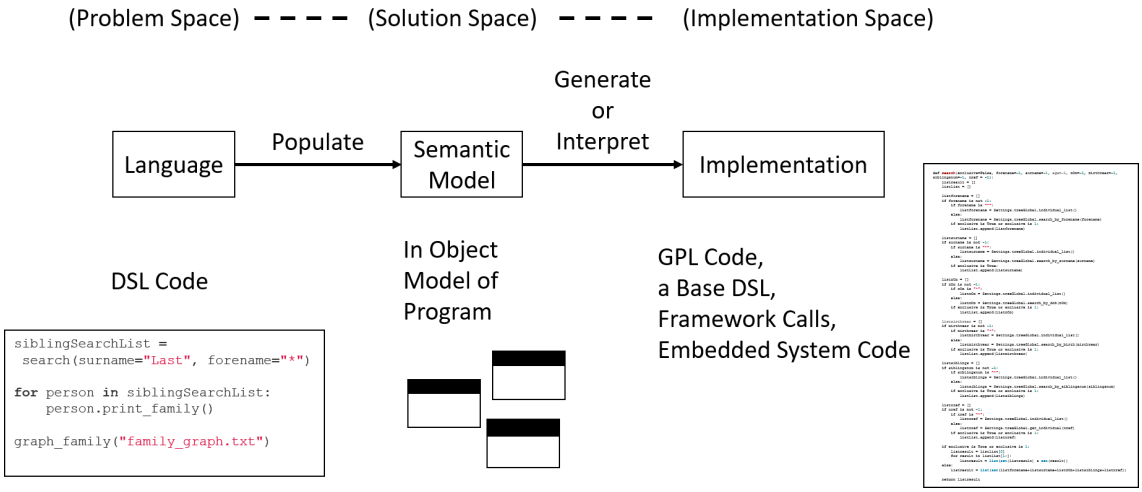

Figure 2.4: The semantic model concept from problem to solution to implementation.

framework. Semantic models have two interfaces: population and operation. The population interface is used to create instances of the semantic model and should only be used by the parser and tests of the model. The operation interface is used to operate an already populated semantic model.

An illustration of the semantic model idea is a DSL for machine learning such as DEFIne (Dethlefs and Hawick, 2017) whose intention is to allow the quick population of complicated machine learning frameworks for experiments. The semantic model for this will hold parameters specificity the training, optimising, evaluating and visualising deep learning models and the domain model will be able to operate this model to fulfil the DSLs purpose. As shown in Figure 2.4 the DSL will be parsed or executed to populate the semantic model, which can then be operated by the domain model to start up the specified experiments.

A semantic model is generally stored in a way suited towards its use rather than an abstract syntax tree because it is concerned with business logic rather than DSL implementation. Separating this allows the semantic model to be used separately from the DSL, and the DSL's testing to be split into parsing the DSL text and

executing the semantic model. This also allows the use of multiple DSLs or parsers to populate the same semantic model; this is referred to as a network of domains by Czarnecki and Eisenecker (2000). For external DSLs, this also reduces the burden of parsing DSL texts directly to host language text which is error prone especially for non-expert compiler writers.

This idea is similar to Parnas's (1979) concept of 'virtual machines' which in modern terminology could be referred to as frameworks which virtually extend the language's architecture. He states that the convenient programs accessible through the higher-level machines allow the writing of simpler software, even though going higher level reduces expressibility and adds computational constraints. Semantic models are like this because they act as an architecture of the domain model to be programmed by the DSLs population.

The implementation of a DSL is generally split into external and internal. An external DSL is a stand-alone language, whereas an internal DSL is based within a host language. Through this thesis, we focus on textual DSLs although we also discuss non-textual at the end of this segment for completeness.

- **Textual External**

  External DSLs are stand-alone languages implemented as a separate entity from any host language although generally using some high-level language as a compilation or interpretation target. This gives freedom in implementation techniques, especially in limiting expression as the language features can be selected carefully which is more difficult in internal DSLs as they may be mixed with host language features. External DSLs can either be implemented as stand-alone languages to perform a task, components for use within general-purpose frameworks or strings for parsing within libraries. The benefit of an

external DSL is they have full control over their syntax and the code they generate. They are especially useful for projects where the main GPL does not have many extension features such as C or in stand-alone tasks which follow an unconventional programming structure such as text manipulation.

There are several ways to implement a textual external DSL:

- Regular language or ad-hoc parser in an interpreting program, suited to naive implementations of simple data formats alike CSV and domain-specific limitations on XML. This is the default for very simple DSLs, for CSV or XML based DSLs there may be libraries which can aid this parsing. Generally, these languages will not provide rich Integrated Development Environment (IDE) support or error checking as a full language would be expected to.
- Traditional language development tools using context-free grammars, suited to creating complex languages. This requires knowledge of parsing languages and auxiliary features such as IDE support will have to be created separately if required.
- Language workbenches providing a full language creation suite, suited to creating supported complex languages. This requires vendor buy-in but gives great usability of end language with the option to add features such as IDE support with little configuration.

An example of an external DSL used widely in industry and academia is Neo4j's graph query language Cypher; an external DSL implemented using a custom parser written in Scala. Being external allows Cypher to constrain operations on the underlying database, at the cost of expressibility in fringe use cases such as implementing graph algorithms. An example of fringe use is Cypher being

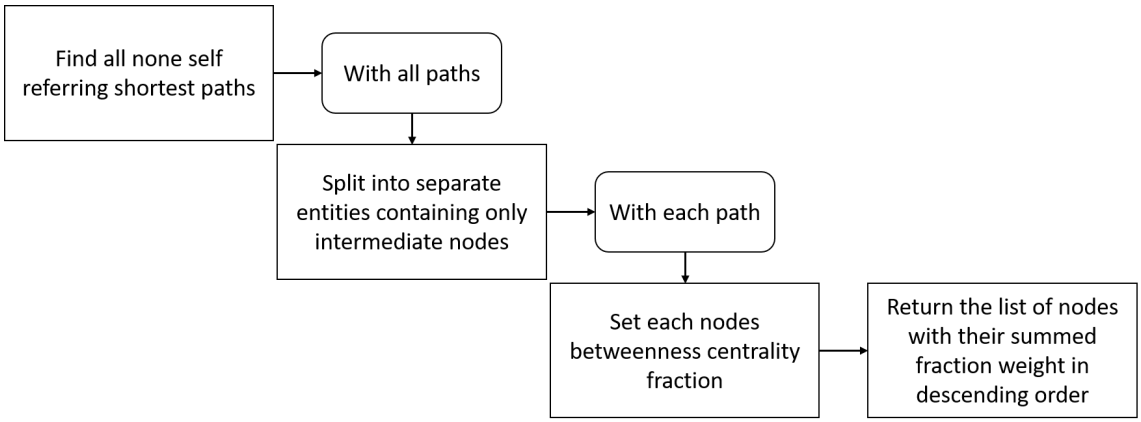

Figure 2.5: The Cypher betweenness centrality algorithm flow diagram.

used for finding the betweenness centrality of a graph using a greedy heuristic adapted from Mattiussi et al. (2011) in Balaghan et al. (2017). This fringe case requires advanced use of Cypher's syntax. The algorithm is shown in Listing 2.3 and illustrated in Figure 2.5.

There has also been work on creating an internal DSL for Neo4j with JCypher from Schützelhofer (2016), an internal fluent interface DSL in Java which generates Cypher code without the user leaving the Java ecosystem.

Listing 2.3: Cypher code to implement a betweenness centrality greedy heuristic from Balaghan et al. (2017).

```
MATCH p=allShortestPaths((a)-[*]->(b))
WHERE a <> b
   WITH a,b,collect(p) AS allpaths, count(p) AS allcount
   UNWIND allpaths AS p
   UNWIND nodes(p)[1..-1] AS n
      WITH n,a,b,1/tofloat(allcount) AS fraction
      RETURN ID(n), sum(fraction)
      AS betweenness centrality
      ORDER BY betweenness centrality DESC
```

- **Textual Internal**

  Internal DSLs are languages within host languages, implemented using existing language features. They are languages which allow writing at the domain-level of abstraction, rather than directly using a set of libraries at the implementation-level. These are especially popular in the functional community because of the inherent extensibility of functions. The main use of internal DSLs is improving software productivity within a commonly occurring problem for the host language user or problems not suited to the host languages style. There are several ways to implement a textual internal DSL such as (van Deursen et al., 2000; Veldhuizen and Gannon, 1998; Czarnecki et al., 2000):

  – Embedding the language within the host as a fluent interface to a semantic model, allowing the use of the host language's IDE features to populate semantic models without any modification. This is the simplest form of DSL, and there is contention to its status as a DSL. We consider fluent interfaces to be a DSL technique, further discussion of this can be found in Section 2.1.2.

  – Embedding the language within the host such as a library using the provided features, limiting possibilities of extension to the host language's features but retaining the host's environment. For example, creating an internal DSL in C++ allows rich additions through templates and operator overloading.

  – Adding pre-processing or textual macros to a host language, allowing the creation of pseudo-external DSLs in a fraction of the time but without semantic checks. This is common in C code because of its lack of DSL support otherwise.

- Meta programming or runtime code generation within the background of the library can allow complex compiler like results, but errors can only be found at runtime. This is especially exciting because domain-specific optimisations can be produced without extending the toolchain.
- Extending the host compiler or interpreter, bringing opportunities for compile-time semantic checks and domain-specific partial evaluation. This is only possible if the host language is extensible or the compiler for the host language allows this.
- Extending the programming environment to provide domain-specific tooling inside an environment using specific tools, this relies on the hooks provided the chosen extension tool. This has been done by Blitz++ using Tau profiling (Shende et al., 1998) where profiling information is based on domain-specific user-level routines instead of the hidden implementation-level. This allows coherence between written code and profile output.

An example of an internal DSL is our Python family tree DSL (Maddra and Hawick, 2016). This DSL is motivated by the rise in the availability of digitised family history data, with the intent of searching, outputting and analysing GEDCOM (Genealogical Data Communication) files without the need of a GUI. GEDCOM itself is a compositional external DSL designed to pass genealogical data between parties. GEDCOM is widely supported by genealogy software and has a simple reference-based lineage-linked structure to link families together. Order within the file itself is insignificant as the people and families are identified using reference tags, this makes the format difficult to read without an interfacing program. Example code using the DSL can be found in Listing 2.4.

The DSL takes advantage of IDLE, Python's Integrated Development and Learning Environment to allow interactive programming as is discussed by Martin (1985), where programming can be considered a monologue or a dialogue. With a domain such as a genealogy file traversing, interactive dialogue between the program and the programmer allows real-time exploration of datasets with feedback of results and conflicts as they happen. Non-interactive execution is unaffected by this, and pre-existing scripts can be called from within the interpreter. To avoid cluttering of irrelevant domain jargon, the DSL's Python modules can be imported and removed seamlessly, as such, the semantic model can be populated by multiple DSLs depending on the user's needs.

Providing a development environment similar to this for an external DSL would require high implementation costs, highlighting the benefit of internal DSLs being able to utilise the host language's ecosystem. Python is a particularly strong candidate for this because of the large amounts of scientific libraries and frameworks available.

Listing 2.4: Python internal family tree DSL example code.

```
tree << "OldTree.ged"

father = search(surname="Last", forename="First")

child = new_person().\
      set_sex("male").\
      set_name("Brand", "New").\
      set_birth("1 JAN 1970", "ENGLAND"))

father.add_children(child)

print(father.name())

father.graph_family()

tree >> "NewTree.ged"
```

DOT is used as an output language for visualisation of the family tree from arbitrary root nodes. This is a re-use of an existing external DSL to reduce our implementation burden. As DOT is a lightweight DSL which just requires a simple compiler to produce images this fits seamlessly with our other forms of output. This benefit of being able to re-use DSLs in different projects should be considered when choosing to create or use a DSL.

An additional definition of 'active libraries' can be made for DSLs which although they are treated as internal DSLs they play a role in the compilation or execution of their code allowing for compile or runtime domain-specific code generation, optimisation, debugging, profiling or testing to be performed (Czarnecki et al., 2000). Active libraries allow the extension of languages in cases where the host language is insufficient or not practically suited towards doing a task, but the task as a whole is suited to the language. An example of this is in Husselmann's

(2014) SOL language towards data-parallel structural optimisation in ABM through the use of a multi-stage programming language Terra embedded within LUA.

- **Visual and Projectional**

  In addition to the traditional textual DSLs there are visual and projectional DSLs which offer interesting techniques for certain scenarios and are interesting research for the future of DSAL development, throughout this thesis, we focus on textual DSLs. Traditionally textual formats have been preferred because of their better integration with existing tooling and users learned experience of coding, with graphical representations being used more at the planning and modelling stages of development although this is changing with advances in technology (Völter et al., 2014).

  Graphical editing is generally suited to compositional DSLs because it shows the connections and positioning of objects. Often there is a textual format behind the graphical interface, which the user may edit independently of the visual representation and keep changes from both such as in Eclipse Modelling Framework (EMF) projects and Android UI design. Displaying large numbers of entities can cause problems for performance and usability. For example, in graphical family tree software large families may become incomprehensible, requiring filtering to reduce the amount of entities to display.

  An example of a computational visual DSL is Repast Simphony's flowchart mechanism for describing agent behaviours. This is suited to non-traditional programmer audiences and displaying agent behaviours in articles to domain experts. The visual format allows the gist of behaviours to be effectively communicated and edited without knowledge of the underlying implementations.

Projectional editing brings forward the opportunity for the mixed-type writing of programs such as tables to populate 2D arrays and selective hiding of non-relevant parts of abstract syntax trees. Examples of projectional language workbenches are MPS (Campagne and Campagne, 2014), MetaEdit+ (Tolvanen, 2016) and Intentional software (Simonyi et al., 2006). Examples of projectionally edited languages include Mbeddr (Völter et al., 2012) and Cedalion (Rosenan, 2010; Lorenz and Rosenan, 2011).

Although some visual DSLs may be considered to be more of a GUI rather than a DSL, we still consider visual interfaces which are designed towards populating a model through domain-specific parameters. We think this is similar to the concept of a punch card and a good example of this type of language is Simulation Program Generator (SPG) found in Hawick (2011). Before electronic development environments were commonplace punch cards were considered a method of programming, the shift towards textual languages is just a sign of the usefulness of the format with the advent of keyboards and electronic text editors. The move towards more visual methods for non-programmers and specific scenarios is a further move towards the new graphical interfaces we are all used to today. The main issues for these visual languages is competing with the portability, compatibility and free input speed of text.

2.1.4 Summary

In summary, we have defined DSLs as a tool to improve software economics and domain expert involvement by changing notations and automating boilerplate creation. This can be used as an enhancement to predominantly object-oriented projects either as external languages or internal extensions to a base language. This frame of use ties into our next sections introduction of AOP.

2.2 Introduction to Aspect-Oriented Programming

We now introduce AOP as a base for separating cross-cutting concerns before moving into DSALs. We concentrate on an object-oriented target with the pointcut advice model which is used by the industry standard AspectJ throughout this thesis.

2.2.1 Motivation

AOP gives developers another option in how they would like to structure their code by introducing aspects, a new unit of modularisation specifically for cross-cutting concerns. Over time the software paradigms have moved towards more effective software economics. Procedural programming brought subroutines to improve reuse; structured programming then placed constraints on program flow, OOP then combined data and behaviour into single conceptual entities which may be extended through hierarchies. These techniques provide a way to package the core concerns of an application effectively although still leave some axillary concerns scattered and tangled across modules.

AOP is a programming paradigm which can be used to augment the capabilities of object-oriented techniques to deal with cross-cutting concerns. This has been included in movements such as post-object programming and advanced separation of concerns where the primary method of software decomposition is object-oriented with techniques such as AOP and DSLs being used where they can improve software economics.

The concept behind AOP is stated in Filman and Friedman (2000) as the desire to make statements in the form: in program P, whenever condition C arises, perform action A over a conventionally coded program P. This means the core concerns of a program can be written as program P without being responsible for the cross-cutting concerns of a program such as the runtime inspection, logging, caching, transaction management and authorisation. These cross-cutting concerns could then be specified as separate entities will be weaved into this program P using the conditions required to fulfil the cross-cutting concerns.

This solves issues of scattered and tangled code where common tasks can be performed from a single place saving time and reducing the risk of mistakes such as leaving a module without the concern or an old version of a concern. Tasks may include adding a cross-cutting concern to an existing project, modifying a cross-cutting concern, toggling of a cross-cutting concern or testing a cross-cutting concern.

2.2.2 What is AOP

Now we established that the goal of AOP is to allow separation of cross-cutting concerns within software which cannot be sufficiently captured using procedural or object-oriented techniques, we can describe the concept behind its use.

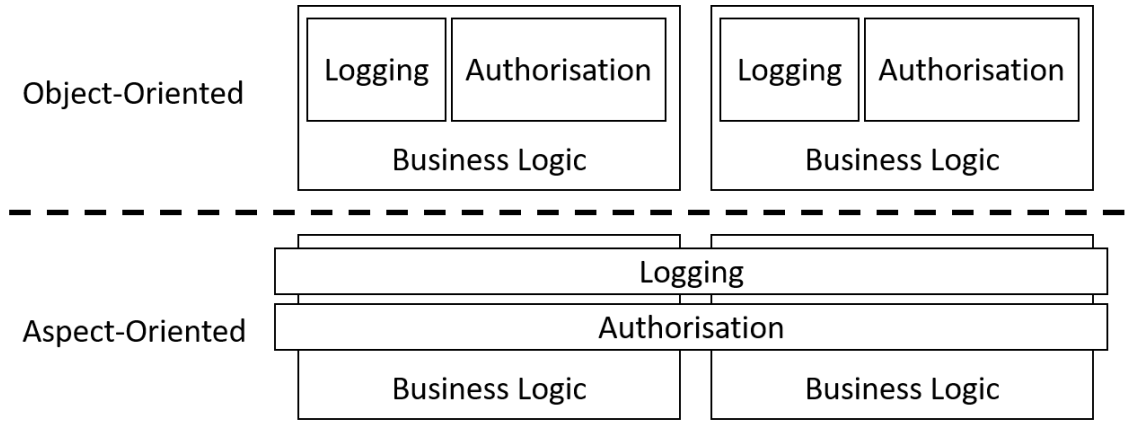

Figure 2.6: Comparison of object-oriented and aspect-oriented methods of separating concerns.

AOP is a paradigm which changes the view of a software system around the concept of a cross-cutting concern. Cross-cutting concerns have been generally defined to be properties that affect the performance or semantics of a system's components in systemic ways. Practically this means a cross-cutting concern is a concern which cannot be encapsulated within the dominant decomposition method because it is scattered across many dominant modules and tangles implementation with other concerns (Elrad et al., 2001). This is illustrated in Figure 2.6 where the cross-cutting concerns are viewed as cutting across the dominant decomposition method in AOP whereas in object-oriented they are scattered and tangled throughout the system as properties of objects. We can explain these changes by how paradigms treat nouns and verbs: procedural programming requires nouns and verbs to be defined separately, object-oriented allows the encapsulation of nouns and verbs within individual objects which may form hierarchies with other objects, aspect-oriented allows the implicit addition of verbs to existing nouns or the addition of nouns through inter-type declarations.

Cross-cutting concerns are separated from primary concerns by adding the new unit of modularisation, the aspect. Kiczales et al. (1997) defines component and aspect as follows:

- A component: a unit of a system which can be cleanly separated using objects, methods, procedures or APIs.
- An aspect: properties that affect the performance or semantics of components in systematic ways which cannot be cleanly separated into components.

Aspects can be symmetrical or asymmetrical, where symmetrical allows aspects to affect core code and other aspects and asymmetrical gives a unidirectional distinction of aspects affecting core code. It is generally accepted that asymmetrical aspects are easier to debug because the graph which could be formed from symmetrical aspects can become increasingly complex compared to the well-defined asymmetrical unidirectional aspects (Murphy et al., 2001; Lippert and Lopes, 2000). We focus on asymmetrical aspects because they are the industry standard and suit the purposes of runtime inspection.

The core idea behind this is that a system can be viewed to allow the primary decomposition method to fulfil the primary concerns and cross-cutting concerns to be separated into aspects and weaved into the solution at compile or runtime. This is a fundamentally different way of looking at these cross-cutting concerns as an extension of the object-oriented mindset, an object's class no longer contains all features of an object because cross-cutting features such as logging and authorisation may be stored as aspects.

We can explain this in relation to the paradigm of literate programming from Knuth (1984) where the highly coupled concerns of documentation and programming which are normally dealt with separately are moved into the same source. In literate

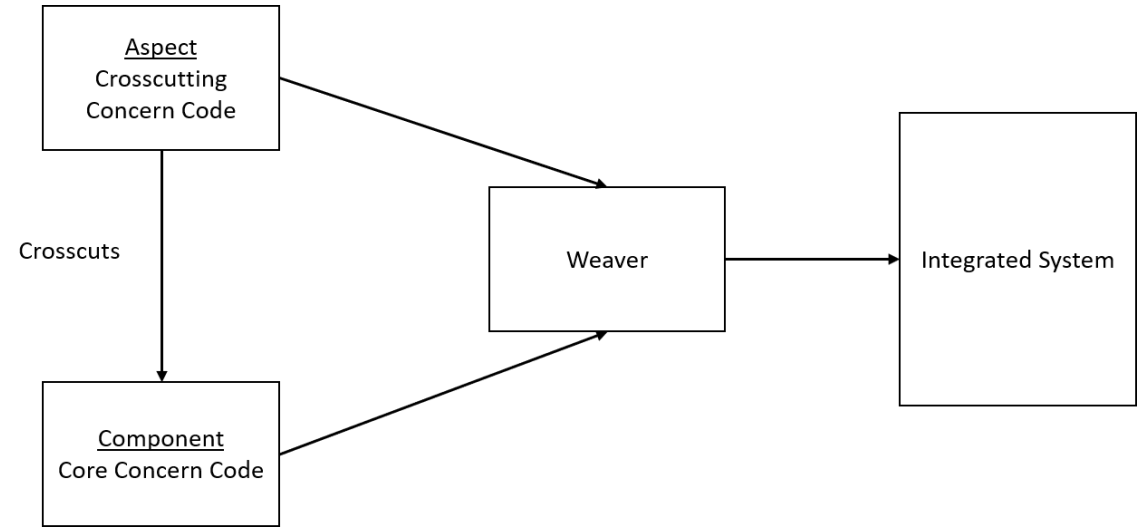

Figure 2.7: The aspect-oriented process weaving aspects and components adapted from Kumar et al. (2016).

programming, the documentation and coding are written in a WEB file which is then weaved to produce documentation or tangled to produce a program. This line of thinking has followed on in modern languages with tools such as Javadoc where documentation is written in Java comments with the related implementations.

Where literate programming brings two coupled concerns into the same source to be separated through weaving and tangling, AOP focus on separating systematic concerns into separate sources which are to be joined through weaving. The likeness of these two approaches in opposite directions can be seen in Figures 2.7 and 2.8.

Compared to ubiquitous and well-understood object-oriented practices this may seem to be over-engineering or difficult to comprehend for developers. This is explored in Constantinides et al. (2004) where AOP is compared to the arbitrary use of go to and come from statements. However, the weaving process is still a structured approach which follows the aim from Dijkstra (1968) of shortening the conceptual gap between the static program and the dynamic process through a trivial correspondence between the program source and execution process. We believe although the use

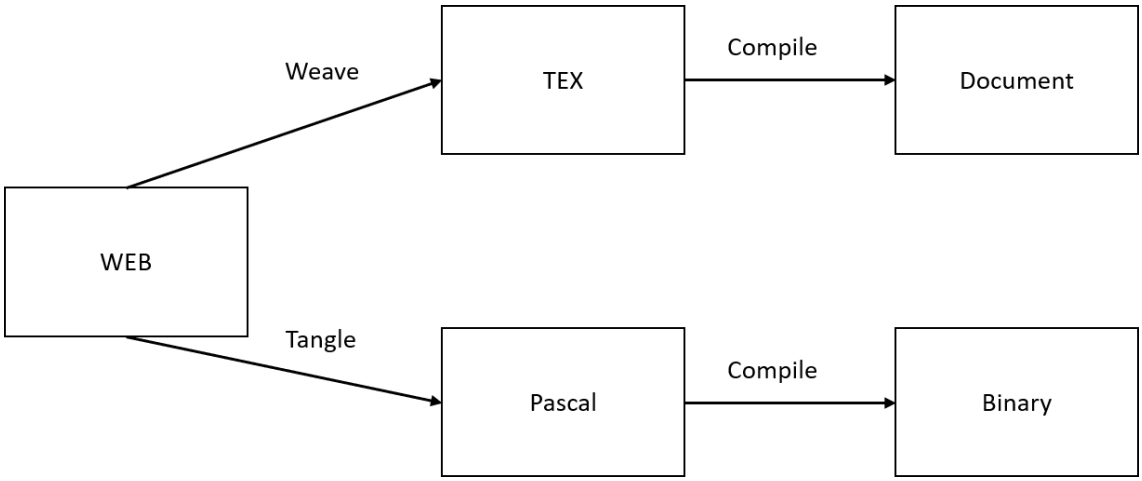

Figure 2.8: The literate programming process adapted from Knuth (1984).

of aspects being weaved from external sources from may increase the complexity of forming a meaningful set of coordinates to describe the process of a program, this is traded off against the benefit of refactoring previously scattered and tangled cross-cutting concerns into structured aspects.

The addition of aspect orientation to a project should change the philosophy of development within the system, where objects contain the core concerns of a system; axillary cross-cutting concerns are dealt with using aspects. This means that a programmer should not have to concern themselves with any changes to the coordinates used at runtime as aspects will provide implementations for cross-cutting systematic concerns without affecting the flow of their code. This weaving of aspects into oblivious objects could be considered a disruption of object-oriented encapsulation, yet when viewed from the aspect-oriented lens of them being part of the classes which they affect, aspects do not break the encapsulation of classes (Elrad et al., 2001). Furthermore, the user can be aided in viewing the progress of a program through the use of IDE extensions such as AspectJ development tools which display Join Points (JPs) and potential pointcut matches of aspects.

In summary Filman and Friedman (2000) have defined AOP's distinguishing features over object-oriented as quantification + obliviousness. The quantification represents the breadth of the JP model allowing separate statements within aspects to affect a program in multiple appropriate places with appropriate potential for effect. The obliviousness represents the ability to weave programs without the target requiring to prepare for receiving these enhancements. As such, the obliviousness does not mean a complete lack of knowledge that aspects will be applied but reasonable freedom from accommodating the aspects as is required in interfaced programming or dependency injection approaches.

Moving onto how aspect-oriented systems are implemented, the ability to supporting cross-cutting concerns requires several components as discussed in (Ubayashi et al., 2004; Kumar et al., 2016; Elrad et al., 2001):

- The JP representation. The points of reference that aspects can select and effect, these may be lexical (static) as in method calls or dynamic as in method executions.
- A means of identifying JPs. Such as on certain method calls or program state changes.
- A means of effecting at JPs. Such as executing code at JPs.
- A means of altering the static structure of a program. Such as inserting new variables for logging.
- A means of packaging cross-cutting concerns. Such as an aspect class.

The first 3 of this list form the language's JP model in the Ubayashi and Tamai (2001) definition, as this forms the basis of possible cross-cutting support.

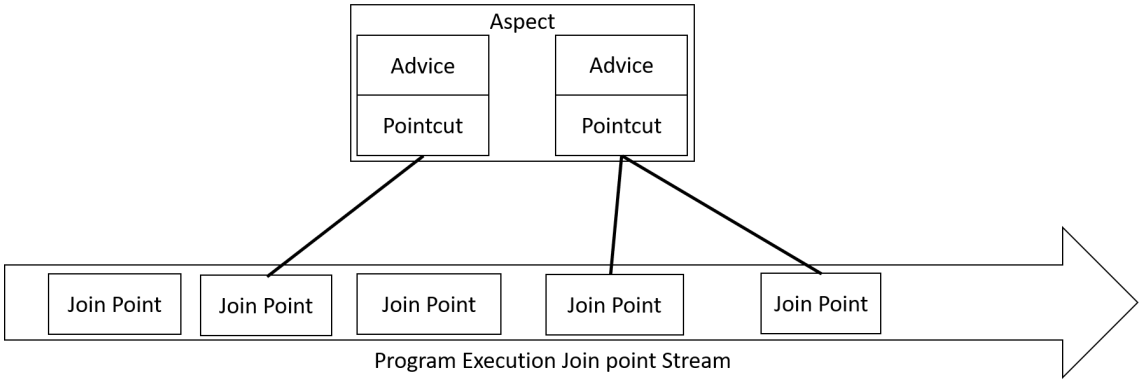

Figure 2.9: Aspect-oriented join point stream through program execution with an illustrative aspect identifying join points.

A JP representation realises itself through execution as a stream of points through the execution of a program where aspects may identify or effect. An illustration of this is shown in Figure 2.9, through the flow of the program JPs arise which may match an aspects identification and have effect executed on it.

2.2.3 Categories of AOP and their Implementation

Through this thesis, we predominantly focus on the pointcut advice model of AOP although Chapter 5 takes inspiration from a composition filter approach. We will now discuss proxy-based weaving through inversion of control frameworks, deploy-time weaving, compiler weaving and the composition filters approach.

Proxy-based

The simplest form of AOP is proxy based; this is where an invocation of control framework wraps methods with proxies which perform the identification checks for JPs. This method limits the potential JPs to method call JPs which can be extended using other methods of weaver implementation. This approach is used by Spring AOP,

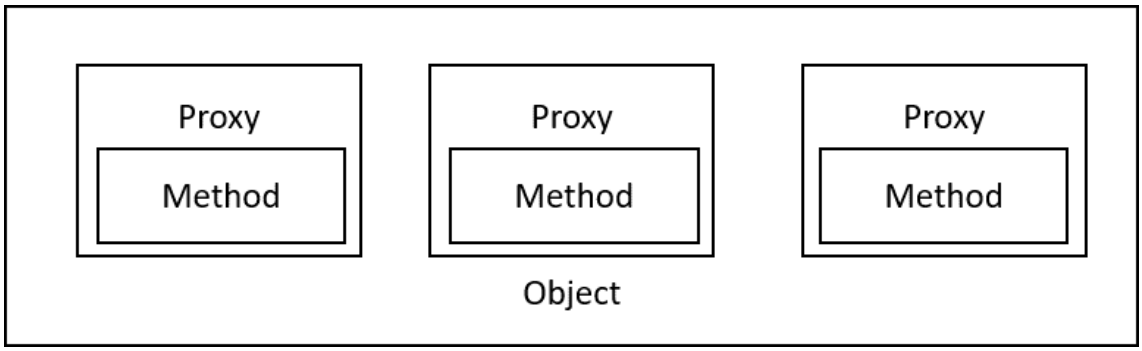

Figure 2.10: The proxy-based AOP method, each method which may have join points attached to it is wrapped in a proxy method.

a part of the larger Spring framework which does not claim to provide a full aspect-oriented solution. The Spring framework is a lightweight framework for enterprise applications, common enterprise tasks such as logging and authorisation are suited to using Spring AOP. Spring AOP allows the use of the AspectJ compiler-weaver instead of or in addition to Spring AOP if they require a complete aspect-oriented solution. Through the Spring IDE Eclipse plug-in, users can visualise their aspects cross-cutting. An illustration of the proxy-based AOP method is shown in Figure 2.10.

Deploy-time weaving is an approach used by Cohen and Gil (2004) where an aspect program is implemented through method overriding during the compilation stage allowing the user to extend, enhance and replace the standard services provided by JavaBeans containers. This means that the program's object structure exists as written without aspects in the executable allowing for debugging and profiling of unaltered core code. This form of weaving has essentially the same expressive power as proxy approaches because it is essentially weaving in proxy classes during compilation time. This approach is useful for middleware approaches where the methods which will require aspect orientation are known.

Compiler-Weaver

AspectJ is the most popular General-Purpose Aspect Language (GPAL); it is implemented through an extended Java compiler and a runtime library. It has also been the base for aspect-oriented research such as the Aspect Bench Compiler (ABC) from Allan et al. (2005).

AspectJ offers three types of JP weaving which produce the same class files regardless of method:

- Compile time weaving, which can be used when the source code for aspects and target is available. The AspectJ compiler will take both source files to create a woven standard Java class file for use with the JVM.
- Post-compile weaving, which may also be referred to as binary weaving is used to weave pre-compiled classes or JAR files.
- Load time weaving, which is binary weaving but deferred until the JVM loads a class. This requires one or more weaving class loaders to be running on the JVM to weave classes before they are defined in the JVM.

AspectJ does not have any explicit support for runtime weaving although toggling of aspects can be provided using simple boolean checks at runtime. An illustration of the resulting implementation from a compiler-weaver method is shown in Figure 2.11.

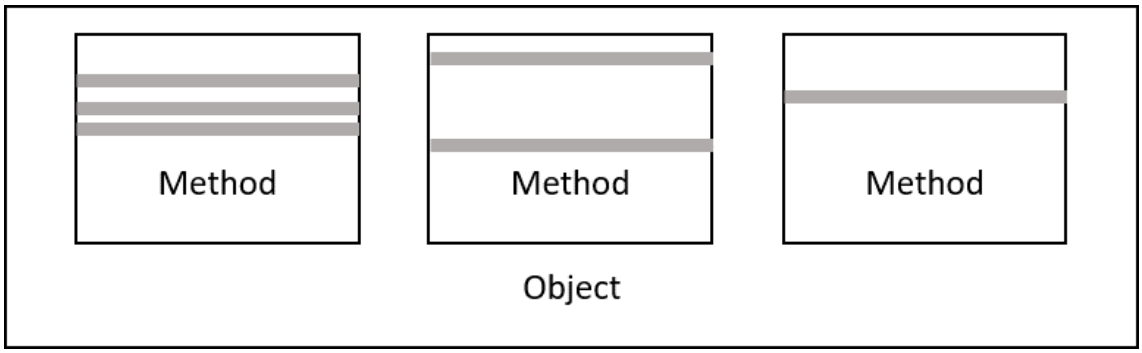

Figure 2.11: The compiler-weaver AOP method, cross-cutting concerns are directly placed into the implementation classes at compile, post-compile or load time.

Composition Filter Approach

A modular extension of the OOP model focused towards composability allowing for aspect-oriented development through declarative filter specifications. The core concept of composition filters is enhancing OOP by manipulating sent and received messages between objects. This allows behavioural changes because externally visible behaviour in OOP is manifested by the communication between objects (Bergmans, 2004).

This approach shows usability for middleware approaches where common filterable actions are known for a large set of programs and may be modified slightly to fit different cross-cutting concerns. This inspires some of the work we do in Chapter 5, as this filters approach is suited towards agent-oriented message passing.

This approach is designed around modification at runtime as it binds to objects at runtime and filters may be altered as program execution goes because they are just data which is operated on. The implementation idea behind this method of extending an object with layers of filters is illustrated in Figure 2.12.

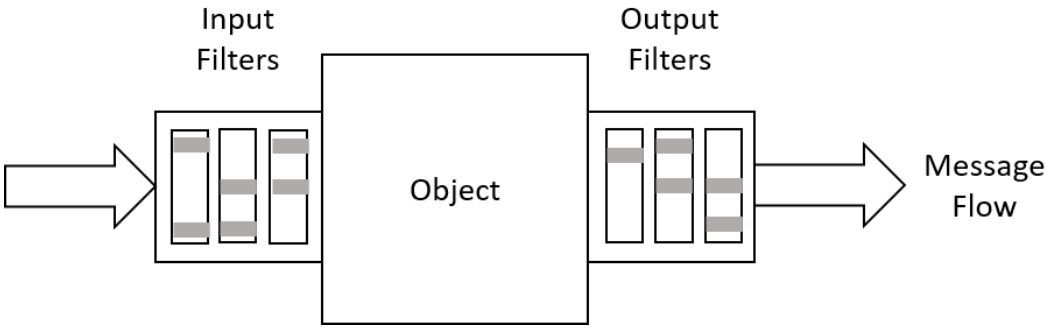

Figure 2.12: The composition model AOP method, cross-cutting concerns are expressed by manipulating incoming and outgoing messages.

2.2.4 Summary

In summary, we have defined AOP as an enhancement to predominantly object-oriented projects, especially in areas with endemic cross-cutting concerns. This definition is in line with our definition of DSLs which can be used in predominantly object-oriented projects for specific tasks which have consistent boilerplate code. These two paradigms are complementary and are combined in the field of DSALs.

2.3 Introduction to Domain-Specific Aspect Languages

We now discuss our thesis's core field of DSALs which brings the idea of domain specificity to managing a set of cross-cutting concerns. This type of language is the basis for our work in agent-based runtime inspection and is used through our contribution Chapters 4 and 5.

2.3.1 Motivation

DSALs are aspect-oriented languages which are domain-specific in one or more of the axis of the language's JP model (Fabry et al., 2015). The previously mentioned axis of a language's JP model are representation, identification and effect. This is a combination of our previously mentioned topics of DSLs and AOP. This combination provides an opportunity to combine existing mature DSL and AOP technologies to form exciting research directions in practical application domains.

The comparison of a DSAL to a GPAL is similar to the difference between a DSL and GPL. Where a DSL is intended to remove boilerplate code from a program, a DSAL is intended to reduce the boilerplate forming step associated with aspect mining when using a GPAL. Aspect mining is a step in the aspect-oriented process where cross-cutting concerns are found before the implementation or refactoring of the concern (Kumar et al., 2016). A DSAL greatly reduces or removes the mining step because it is already aimed towards a set of cross-cutting concerns in either representation, identification or effect.

A hindrance of DSAL implementation, when compared to DSLs, is they require a weave target, which inherently means coupling to external sources. While a DSLs implementation will be tied to some semantic model generally under the control of the language developer, the weave target's external source may be a static completed program, a program under current development or a version of some middleware component. The weave target problem is specific for DSALs rather than GPALs because a GPAL's weave target is generally target language features which will remain static regardless of base program. For example, a GPAL may target method calls or variable changes in general as opposed to specific behaviour calls or state changes which depend on some specific application implementation for a DSAL. This

is where pure-DSL techniques and pure-AOP techniques clash. We must take into account not only the generation of implementation from a problem domain-level DSL but also coordinate this implementation within the host program.

DSALs are especially suited to projects where there is a stable domain and stable weave target. An example scenario showing the usefulness of a DSAL is a software engineer can create languages for domain experts to use when dealing with cross-cutting concerns on some base program. For example, changing many runtime inspection tasks for experiments upon pre-written simulation models. In this case, the base weave target remains the same which is ideal for the DSAL developer, and the domain experts would not need to concern themselves with the weaving and boilerplate implementation of their cross-cutting experiments.

An interesting direction for DSAL research is how we can reduce the start-up costs for creating and using a DSAL, with a target of nearing the difficulty of the implementation of DSLs. Without appropriate methods and tools for creating DSALs, there is an increased risk of ad-hoc solutions being chosen even when a DSAL could provide benefit in the long term, especially when deadlines are tight.

2.3.2 What is a DSAL

Our definition of what can be classed as a DSAL relies on our previous definitions of what constitutes aspect-orientation and domain-specificity. Although a DSAL is a DSL, we separate the definitions through this thesis to avoid requiring specifying aspect-oriented DSL and non-aspect-oriented DSL throughout. Using the definition from Fabry et al. (2015), a DSAL is an aspect-oriented language with domain specificity applied to one or more of the axis of aspect orientation: representation, identification and effect. We will discuss the concept of domain-specific JP

representation, identification and effect within this section. This concept forms a three-dimensional graph to give a rough estimation of how domain-specific a DSAL is, where coordinates are assigned in the same way as specified in the graded membership definition of how we can define a DSL from Section 2.1.2. For example, a DSAL which is only domain-specific in effect would resemble a GPAL which uses a DSL for effect, whereas a DSAL which is domain-specific in representation and identification would resemble a GPAL with declarative, domain-specific pointcuts provided. We will now examine how each of these axes could be made to be domain-specific.

Representation

A DSAL is based around its representation which provides the JPs which can be targeted by the identification and effect axis. General-Purpose Join Points (GPJPs) refer to places in the execution (dynamic) or syntax (lexical) of the implementation of a program; Domain-Specific Join Points (DSJPs) refer to JPs at a domain-specific level of abstraction at either the dynamic or lexical level.

Through this thesis we use the specialise, aggregate and create model which is defined in Fabry et al. (2015) and shown in Figure 2.13. This model defines the implementation of a DSJP as the specialisation of a single GPJP, the aggregation of many GPJPs or the creation of a new JP augmenting the original program's flow of GPJPs.

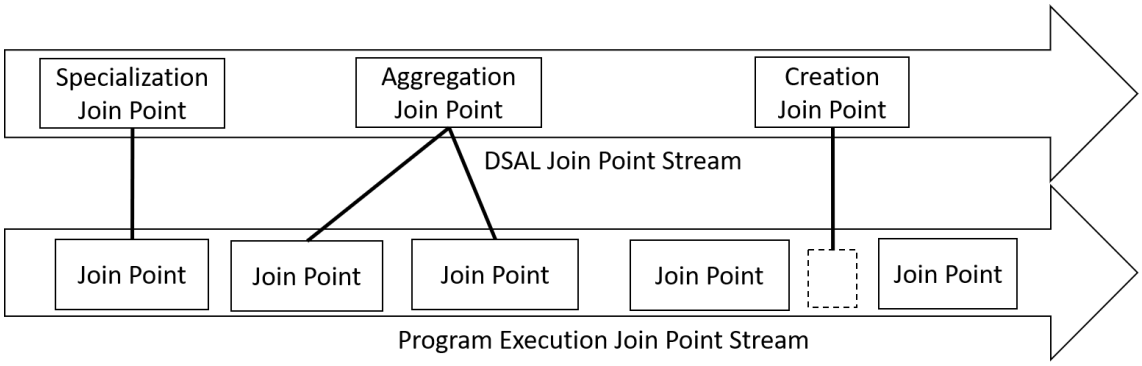

Figure 2.13: DSJP in relation to GPJP through specialisation, aggregation and creation.

Identification

Given a JP representation, identification allows selection of the JPs to be affected. The domain specificity of the identification could be an extension of a domain-specific representation or a domain-specific layer which forms declarative statements over a general-purpose representation.

Effect

Effect is placed at an identified JP, the considerations for a domain-specific effect are the language used and parameters available to the programmer. The parameters available affect the domain specificity of an effect language by providing the leverage the programmer has on the context around the JP. For example, an effect could have no context, a GPJP description giving information about program state or a domain-specific object which provides declarative calls for tasks which are important for this domain. The domain specificity of an effect language can be judged as a DSL would be, with the most domain-specific options being full declarative DSLs with domain-specific IDE support taking into account JP context. On the other

hand, the effect of a DSAL could be a GPL if the goal is to shield from program implementation for inserting cross-cutting concerns although requiring full leverage to implement the concern.

The domain specificity of effect code could also take into account the limits of a domain, for example, a DSAL designed for the verification of models may provide an effect language which may record information for verification purposes but cannot affect the state of a simulation.

2.3.3 DSAL Development Approaches

The development approaches of DSALs has been covered in a review from Fabry et al. (2015) which highlights the lack of direction on how to develop a DSAL through the literature. Furthermore, any DSL literature methods which apply to DSALs must be adapted to consider the external weave target, aspect composition methods and implementation JP context mapping to domain-specific context. The main implementation methods highlighted by Fabry et al. (2015) are interpreters, compilers, embedding, hybrids and DSAL infrastructures. This list is inspired by the list in Mernik et al. (2005) with DSAL infrastructures added. We can relate the DSAL infrastructures to language workbenches which were not prominently used at the time of the Mernik et al. (2005) survey.

When considering the implementation method for a DSAL, it is important to consider the community around a method's tooling, the learning curve of a method and re-usability of the method across other problems. This is a problem for DSAL development more than DSL development because DSLs have mature parser generators such as ANother Tool for Language Recognition (ANTLR) and language workbenches such as Xtext which have large communities around mature consistently

updated software. Using mature software in the implementation of languages goes towards ensuring updates and bug fixes are available across the implementations of the DSAL project which may not be available for hobbyist or academic projects. As such, movements to increase DSAL adoption cannot just focus on creating new research tooling but also methods to develop and use emerging tooling, preferably leveraging pre-existing mature software.

We now look at the different types of implementation for DSALs.

Interpreters

Creating a DSAL by making a JP interpreter on an instrumented program is the simplest form of creating a DSAL. This method allows for dynamic JPs but not lexical JPs as everything is done at runtime.

The implementation of this method involves instrumenting the application with JPs which can be checked using the runtime interpreter and advice can be fired directly if there is a match. This approach will generally be developed using the host language by adding the DSAL classes, JPs may be added using a GPAL or directly called within base code.

This method allows rapid development times and is extendible even at runtime because the DSALs weaving interpreter is held in memory rather than weaved into the source code. This means that prototyping new DSAL ideas and creating DSALs for short projects is an ideal use case for this method.

The disadvantage of this approach, like all interpreter approaches is the runtime overhead of matching JPs in an interpreter and no access to semantic checks at compile time which a compiler would have. The runtime slowdown can be somewhat reduced by using techniques such as partial evaluation of JPs meaning only possible JPs are passed through the full interpreter.

Compilers

Creating a DSAL by compiling targets and aspects together to source or binary is an efficient way of implementing a DSAL because only necessary JPs need to be added into the source code. This method is less open to runtime changes because changes are done in the source of the program being weaved into. To change aspects at runtime would require the reloading of affected classes or aspects to be pre-weaved in but toggled off until needed. Both lexical and dynamic JPs may be implemented using this method, and semantic checks can be provided at compile time.

This method is more time consuming to implement than the interpreting method, and the implementation requires knowledge of code generation. The generator could weave into the target source code or generate semantically equivalent code in a GPAL to delegate the task of weaving. This could be implemented by creating a compiler from scratch, extending a compiler or adding a pre-processing step. If the weave target is a DSL, the weaving may be easier compared to a GPL because of simpler keyword matching.

Creating a compiler weaver for a DSAL is different to a GPAL in that the mapping from domain-specific concept to implementation may change over time, whereas for a GPAL mapping to language is consistent. This problem remains whether weaving directly to source or generating GPAL code because the implementation changes will still require changes.

Embedding

Creating a DSAL by packaging host language features into a library is an emerging way of implementing DSALs as aspect-oriented techniques become available in modern multi-paradigm languages. This is analogous to implementing an internal DSL where the host language defines the difficulty and opportunity for language development. As such, this approach's difficulties compared to other methods depends on the host language and type of aspect orientation required.

Simple AOP can be performed using proxy AOP which only requires GPL features to wrap each target method with a proxy method which will perform all weaving at runtime. This requires target code to have a network of proxies, these proxies could be ad-hoc or structured using a framework such as Spring AOP.

Further AOP can be performed in languages with post-object-oriented paradigm features such as Scala with traits and Groovy with its meta-object protocol, all without leaving the host language's feature set. Languages which allow meta-programming such as Java and C# allow the same sort of internal DSAL to be created yet with a considerable performance cost by modifying objects at runtime.

This approach shows great promise for the future as more languages take this multi-paradigm approach and developers begin to use AOP as part of their normal development without requiring extra tooling in their build process. Depending on the host language features used this may be done at compile time or runtime, forcing performance cost to be calculated on a case by case basis.

Hybrids

Given the differing potentials of these approaches, in many cases, it may be wise to use a combination to alleviate issues while retaining benefits. These hybrids may be because of requirements such as embedding the identification and effect measures into a host language but requiring a separate JP interpreter because the language lacks appropriate aspect-oriented features. On the other hand, a hybrid may be done to take advantage of different approaches such as using a mature GPAL weaver to capture GPJPs then to interpret them as DSJPs at runtime. An example of making a production level DSAL using a hybrid approach would be a language which has separate implementation styles depending on the situation, where runtime changes are required an interpreter is used, yet with static JPs a compilation approach is taken to increase performance.

This approach requiring knowledge of different approaches may seem to increase development time, but on the other hand, using appropriate methods to alleviate the pitfalls of choosing a single method can save time. For example, the compilation approach is efficient at runtime but time-consuming to create a compiler for each DSAL, using a GPAL to compile a partially evaluated set of JPs which are then interpreted at runtime can give a middle ground between fast language development and appropriate runtime. This approach has the most power out of the above

methods because it can combine each of them to achieve the required results. The disadvantage of this method is scattering and tangling within the DSAL itself must be managed carefully because of the dependencies between different methods.

Infrastructures

The DSAL infrastructure approach is a move towards making language workbenches for DSALs, providing a cohesive environment designed towards the DSAL development and deployment process. This is to solve the problems of requiring multiple heterogeneous pieces of software tied together to develop DSALs.

Developing an infrastructure from scratch is a gargantuan task which requires a large intended user base to make development worthwhile. While DSLs have large projects such as Eclipse Xtext and JetBrains MPS for creating DSL infrastructure, most DSAL infrastructures are academic projects which are focused towards research. This brings into question if it is worth using an academic project to create DSALs over the mature DSL and AOP tools which give re-usable skills and a wider community of users.

The main considerations when choosing an infrastructure are the completeness of its feature coverage, the learning curve required to create solutions, the frequency of software updates and community which relies on the software. The vendor or developer of the infrastructure is also a consideration because once a project has chosen to use an infrastructure, it is tied into this decision.

A common downside of large frameworks is although example projects are easy to follow and show useful features, moving onto creating bespoke solutions requires a deep understanding of the features and semantics of the infrastructure. These

downsides are alleviated with industry standard mature projects which have active community boards filled with help topics and development from many sources keeping updates flowing.

One of the most notable infrastructures for DSALs is ABC from Allan et al. (2005) which is focused around extending the AspectJ compiler. The code produced by this infrastructure is production quality and has a relatively large user base. This infrastructure can also be used for general AOP extensions, for example, Harbulot and Gurd (2006) created an AspectJ extension allowing JPs at loops. The addition of loop JPs in AspectJ allows the language to perform more implementation invasive AOP, which is discouraged by the general audience but may be useful in specific scenarios such as creating a DSAL generating AspectJ code to implement a weaver.

Another example of extending the AspectJ compiler is the language-oriented modularity project from Hadas and Lorenz (2017) which extends the AspectJ compiler to allow for semantic preservation of DSAL concepts in AspectJ + metadata implementations. This workbench takes the existing mature AspectJ compiler and adds means to insert metadata which is relevant for DSAL development.

As most AOP research is done with the assumption that Java will be a base language, The Pluggable and OPen Aspect Run-Time (POPART) from Dinkelaker et al. (2009) is a DSAL infrastructure for weave targets with a DSL as a base language. POPART allows the creation of DSALs using internal DSLs within Groovy to create plugins for the JP representation (referred to as JP model in the article), pointcut language and advice language. POPART is implemented as a library in groovy with AspectJ for instrumenting GPJP; they do not consider aspect composition in their approach.

2.3.4 Summary

In summary, we believe DSALs provide an important tool for cross-cutting concerns using domain-specific techniques. We believe these are especially useful where a domain expert can change a cross-cutting concern without requiring implementation details. Runtime inspection, visualisation and verification of simulations are prime examples. DSALs suffer from a longer implementation time and less mature tooling compared to a standard DSL which opens up research for their development techniques and tools.

2.4 Introduction to Agent-Based Modelling

Finally, we introduce the foundations of our application area throughout this thesis. We concentrate on lattice-based ABMs of complex systems in Chapter 4 and the communications between intelligent agents in Chapter 5.

2.4.1 Motivation

ABM is a popular complex systems research tool used to simulate the actions and interactions of autonomous agents with the intention of assessing the system as a whole. The field is based around emergence which is the process of many agents with seemingly simple rules at the microscale causing complex behaviour at the macroscale to re-create or predict the appearance of complex phenomena in the system being modelled (Bonabeau, 2002). An agent has been defined as anything which perceives its environment through sensors and acts upon its environment through actuators (Russell and Norvig, 2003). This is a broad definition which allows many things to

fall into the category of agent, because of this the definition is to be used as a tool for analysing systems rather than an absolute characterisation. The term agent is used through this thesis where it is profitable through our scientific lens to view the item as an agent. For example, a polling method for our visualisation could be viewed as an agent although it does not contain interesting properties for our purposes, so we are not to be concerned with it. Further definitions of agent types are given later.

ABM is especially useful for the study of complex systems where agent-agent and agent-environment processes dominate the behaviour of the system and thus influence the examined data's relationships and potential for future action in critical ways (Marshall and Galea, 2015; McLane et al., 2011). Furthermore, the flexibility and repeatability of in-silico experiments within computational science allows ABM to modify existing models and examine previously examined behaviours with high re-use of existing systems (Hawick, 2012b). This is especially useful in areas where it is expensive, complicated or dangerous to perform experiments in the field, such as military and offshore environments. Two models discussed in this thesis are the Sugarscape and Kawasaki models, visual examples of these models can be found in Figure 2.14. The Sugarscape model is an ant-based model where choices are dictated by environmental factors. Kawasaki based models are exchange or spin based where choices are dictated by interacting with neighbouring agents.

Throughout this thesis, we focus on ABM rather than agent-oriented artificial intelligence although there is significant cross-over between the ideas and applicability of our work between these fields. As our focus is the runtime inspection of how agents act, the internal structure of the agents is of little concern to us. The agents we focus on can generally be described using Russell and Norvig's (2003) skeletal taxonomy of agents as simple or model based reflex agents as they appear in complex physical systems rather than goal or utility based agents which are more common in complex

adaptive systems. We hold scope to investigate the use of this technology further especially as big-data analytics and machine learning open the door for further agent inspection.

Emergence refers to the collective phenomena or behaviours in complex systems at the macroscale which arise from microscale interactions. In short, the whole of a complex system is more than the sum of its parts. Common emergent properties of systems are self-organisation into patterns such as flocking birds, chaotic behaviours such as small changes in initial conditions producing large changes in later conditions, fat-tailed behaviours such as rare events happening far more often than expected and adaptive interaction such as agents modifying strategies through experience (Holland, 2014).

A direct example of this is water's wetness, no individual molecule can be assigned the property of wetness, yet many individual interactions of these water molecules create the emergence of wetness. The whole is larger than the sum of its parts. One explanation for such unexpected behaviour comes from Simon's (1996) explanation of evolution not as a set of agent tournaments for the occupation of a fixed set of niches, but the proliferation of niches coming from the act of evolution itself. The environment to which agents adapt is formed mainly of the other agents in the system rather than the physical environment; which is, in turn, changed because of the agent population.

ABM has gained increased use in fields such as economics, social science, military, biology and public policy as a tool for the investigation, explanation and prediction of complex phenomenon (Macal, 2016). Traditionally models have been used in only the context of a theory which they mechanically describe. The field of complex systems has led to simulation models not only used as a bookend for a well-accepted theory

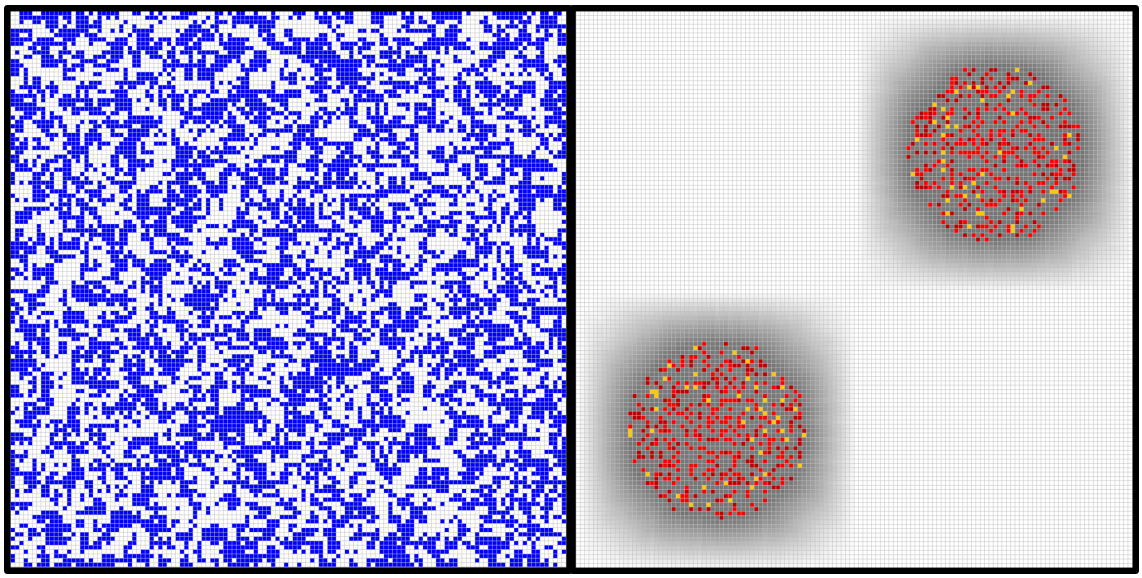

Figure 2.14: Examples of Kawasaki (left) and Sugarscape (right) models.

but used to actively and autonomously mediate between theory and the real system (Morgan and Morrison, 1999; Janssen, 2012). This change allows us to consider how techniques of interacting with models can be used to forward the frontiers of a field rather than just describing theories within it.

The purpose of an ABM depends on the underlying conceptual knowledge of the area and the objectives of the study being performed. In a survey of ABM practices Heath et al.'s (2009) defines three roles of a model which lie on a spectrum dependent on conceptual knowledge of a system. The basic premise of the scaling goes from a low conceptual knowledge being examined by implementing models to generate potential behaviours to a high conceptual knowledge being used to examine predictions of the real system as is shown in 2.15.

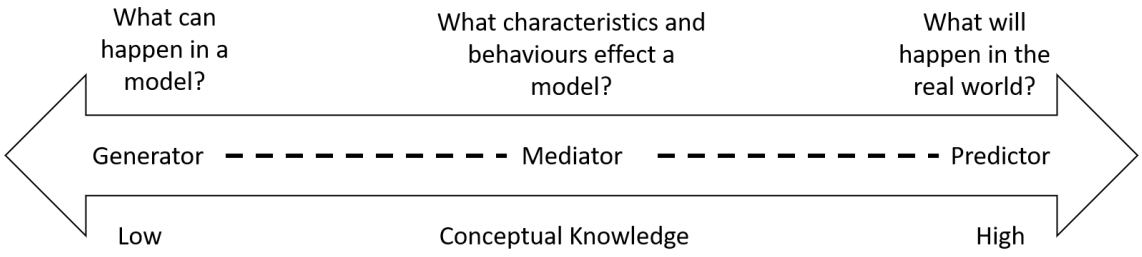

Figure 2.15: The 3 roles of ABMs adapted from Heath et al. (2009).

2.4.2 Categories of ABM

ABM simulations have been separated into four types by Macal (2016) because of the difficulty in giving such a broad field a single encompassing definition. The definitions essentially scale in the complexity of computation from simple agents performing isolated behaviours to intelligent agents adapting as the model runs. These definitions are useful in defining what an ABM is, what it does, how it works and what it can do.

- **Individual ABMS**

  Agents are treated individually and have a diverse set of characteristics. This is a baseline of what an ABM is; using simple agents with simple scripted behaviours. The benefits of this type of model is simple agents scale well for ultra-large simulations.

- **Autonomous ABMS**

  Agents are autonomous individuals who act upon changes in the model on their own accord. This takes the autonomy of an agent as the fundamental distinction of what makes an agent. The benefits of this type of models is the agent's actions can be based upon their inner state rather than being dictated to them by an observer, this type of model also scales well for large simulations.

- **Interactive ABMS**

  Agents are autonomous individuals involved in non-trivial interactions between other agents and the environment which allows for higher modelling capability at the cost of another degree of computational complexity. This type of model requires special care for scalability as agents become more intelligent and rely on synchronising actions with other agents and the environment.

- **Adaptive ABMS**

  Agent are autonomous individuals involved in non-trivial interactions between other agents and the environment while adapting their behaviour individually or changing group behaviour throughout the simulation. These agents are aimed towards being more intelligent than in previous definitions, and as such, are significantly more computationally complex and difficult to simulate in reasonable time on the large scale.

Although ABM has multiple definitions to what constitutes agent simulation and roles within studies, they are used in a standard flow which has been covered by many both in general (Macal and North, 2006) and in specific fields such as pharmacology (Cosgrove et al., 2015).

The ABM process is described in Figure 2.16. The base of the process is hypothesising before, and after each iteration of model development, this base allows for the model to be first created and then improved at each step. The core process of model creation contains first the conceptual development, followed by an operational implementation of this conceptual model. Once an operational model has been created this must be verified against the conceptual model and validated

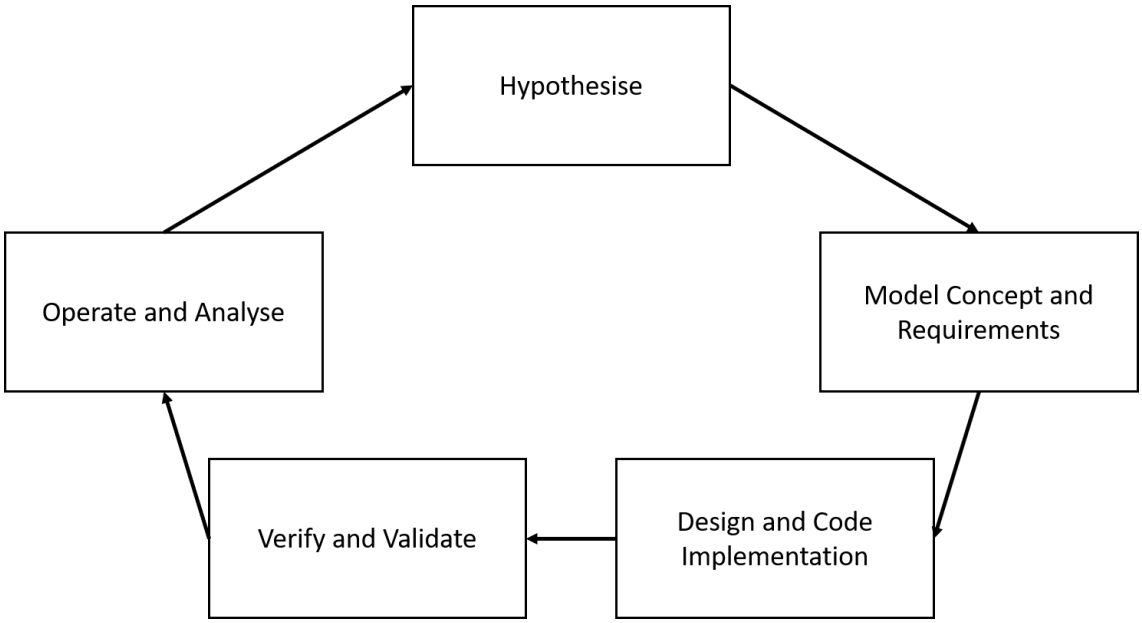

Figure 2.16: The basic ABM process adapted from processes described by Macal and North (2006), Husselmann (2014) and Mitre (2014).

against the target system, following this, the model should be operated and analysed with the validation and verification results in mind. The process will then begin again with the hypothesising informed by the results of previous model steps.

2.4.3 Popular Frameworks and Languages

The purpose of advancing the field of simulation leads to the technology being used to implement the model being the object of observation. ABM has seen a great deal of effort into improving the tooling and methods available for its users specifically in the conceptual process, domain-specific frameworks and analysis tools. ABM is a prime candidate for software re-use because simulation environment boilerplate code requirements are similar across models, and inter-disciplinary use has driven the requirement for tools to aid the development process for domain experts. It has been highlighted in the literature that the improvement of ABM tooling by computer

scientists is a beneficial step forward especially with the community of digitally literate members of other fields emerging to benefit from the inter-disciplinary work (Cegielski and Rogers, 2016).

There is an active community of researchers working on ABM frameworks using domain-specific and aspect-oriented techniques. The first notable toolkit to gain traction in the community was Swarm (Minar et al., 1996) aimed at creating complex models without having to write the boilerplate framework; this has become an inspiration for a class of GPL framework based ABM toolkits. This is in contrast to stand-alone languages such as NetLogo (Wilensky, 1999) and seSAm (Klügl et al., 2003) which allows for the creation of models through simplified frameworks for faster development time at the expense of runtime speed.

A majority of the state of the art frameworks are built on top of object-oriented general-purpose programming languages (Bandini et al., 2009) which opens up the possibility for augmenting framework capabilities with currently available libraries and internal DSLs which are not made directly available by the framework. Major frameworks for object-oriented ABM include:

- Multi-Agent Simulator Of Neighbourhoods or Networks (MASON) from Luke et al. (2005) is a micro-kernel approach to ABM which is developed with the intention of being extended by domain-specific packages. MASON's simplicity and execution speed means it is a primary candidate for creating large custom purpose simulation suites. D-MASON from Cordasco et al. (2012) is a parallel version of MASON designed to harness unused PCs to increase the performance of models. The goal of this framework is to introduce distribution at the framework level so domain experts with limited knowledge of computer programming and systems can be unaware of the distribution.

- REcursive Porous Agent Simulation Toolkit (Repast) is a full suite approach to ABM which has a Simphony framework (North et al., 2013) and HPC framework (Collier and North, 2013) which both have ReLogo modes of input for simplified Logo inspired input. Repast Simphony is an easy to learn Java framework for use on standard computing hardware. Repast for HPC is an expert focused C++ framework designed for use on large clusters and supercomputers. Repast is an interesting case because it has suites suitable for all programmers from domain experts using the ReLogo groovy DSL up to expert high-performance programmers using a lean C++ framework.

- Flexible Large-scale Agent Modelling Environment (FLAME) from Holcombe et al. (2006) is an ABM framework designed around templates which map formal descriptions of agents into simulation code, allowing simulations to run on systems from laptops to high-performance computers. This is done by focusing on the inherent parallelism of ABM, unlike other frameworks which treat agent actions on a one-by-one basis (Coakley et al., 2012). Models are described using XML and agent functions are implemented in C. The model specifications are then compiled to C code by the xparser which can then be compiled with agent functions to produce a simulation. FLAME GPU from Richmond et al. (2010) builds upon the flame framework by targeting graphical processing units which are under-utilised by other frameworks while hiding implementation boilerplate for GPU data storage and agent communication. Both these frameworks allow for very large, efficient models.

- Java Agent DEvelopment framework (JADE) from Bellifemine et al. (2003) is a Java framework which supports agent-based development by providing middleware for Foundation for Intelligent Physical Agents (FIPA) compliant messaging and graphical tools to aid in debugging ABM. This is a multi-agent

development framework rather than an ABM framework although can be used to create ABMs if the user creates their own scheduler. JADE concerns itself with the creation and communication of agents, leaving the rest for the user to develop themselves. JADE has been the target of DSL activity by Bergenti et al. (2017) creating JADEL, a language for agent-oriented development of real-world modern driven projects. This language limits its domain specificity by focusing on the definition of agents rather than the creation of a full simulation; the author refers to this as an agent programming language rather than a multi-agent programming language.

AOP techniques have been brought to ABM by Amiguet et al. (2004) which uses the Aalaadin meta-model (Ferber and Gutknecht, 1998) for separating agents for AOP on top of the MadKit multi-agent platform created by the same authors as Aalaadin (Gutknecht and Ferber, 2000). The Multi-Agent Modelling language is a macro-language over swarm which uses the aspect-oriented separation of concerns to allow computer scientists who cannot model social systems and social scientists who cannot implement models to be able to each deal with their side of a project (Gulyás et al., 1999). Established frameworks have also been augmented with aspect-oriented features to explore the potential for AOP on top of ABM. Notable examples include HLA-ACTOR-REPAST which allows the distribution of RePast models using asynchronous message passing actors in a theatre architecture using aspect-oriented techniques Cicirelli et al. (2009) and AspectNetLogo which allows the use of aspect-oriented techniques to be used on Netlogo (de S. Braga et al., 2012).

2.4.4 Concerns Within ABM

ABM is an especially suitable target domain for frameworks and languages which aid the separation of concerns because of the division between the conception, implementation and operation stages of modelling; high domain expert involvement; high quantifiability of autonomous components and consistent boilerplate code. Common concerns of ABM identified during the development and use of the Animaux framework are (Hawick, 2012a; Preez et al., 2012):

- The model's agent behaviour
- The simulation geometry
- The specification of parameters both statically and dynamically
- The dynamic visual rendering of the whole system
- The runtime inspection of the system of agents and their environment

Core concerns of ABM can be successfully expressed using existing popular frameworks and languages which provide general boilerplate code for ABM, as outlined in Section 2.4.3. Furthermore, the active DSL community for ABM has created very domain-specific frameworks and languages to facilitate model creation by domain experts in a specific sub-field. For example, ENVISION from Bolte et al. (2007) is a spacial multi-paradigm modelling framework for the analysis of scenario-based community and regional integrated planning and environmental assessment, which has been used to model forest management outcomes using an ABM by Spies et al. (2017). This type of tool is becoming more important as 'in-silico' experiments are spreading across interdisciplinary research in fields such as systems pharmacology and public health as covered in reviews by Cosgrove et al. (2015); Tracy et al. (2018) respectively.

While these tools provide excellent reduction in boilerplate code and abstraction level of language, cross-cutting concerns such as runtime inspection are still scattered and tangled throughout projects even in highly domain-specific frameworks. Runtime inspection of ABM is an especially interesting concern for DSAL research because of its importance for agent-based modellers and cross-cutting involvement of the other concerns. The mindset of ABM being from the microscopic level up poses interesting questions for the quantification of concerns and runtime inspection of models (Bonabeau, 2002); it gives the potential for not only the actions and interconnections of the microscopic units but also the macroscopic emergent phenomenon which these simple actions produce.

A real-world field which relies heavily on runtime inspection of models is validation and verification where the emphasis is on ensuring the model is an appropriate representation of a real system given a set of objectives (Heath et al., 2009). This is an important step because models are inevitably imperfect representations of the real system, but we can take steps to prove they are conceptually sound, appropriate for purpose and correctly implemented (Stanislaw, 1986). Verification and validation has also been referred to as the sanctioning of simulation models and assessing credibility of models rather than the binary claim of verified or validated because of the inherent subjectivity involved in choosing appropriate objectives to base completion criteria from (Winsberg, 1999; Schlesinger et al., 1979).

2.4.5 Summary

In summary, we have defined our target domain of ABM as a tool scientists can use as apparatus for 'in-silico' experiments of many simple agents producing emergent behaviour at a macro level. This field is pre-disposed to the use of domain-specific

and aspect-oriented techniques because of: many sub-domains, domain expert involvement, high quantifiability and consistent boilerplate code. This thesis is especially concerned with research opportunities for the runtime inspection of ABM for both general and domain-specific using DSALs. We now move into a literature review of DSALs for the runtime inspection of ABM.

Chapter 3

Towards Modularised Runtime Inspection of Agent-Based Models

This chapter follows our problem statement through the literature towards our two main contribution chapters; this sets the purpose, significance and boundaries of the following chapters. This chapter contains the literature required to lead to our contribution chapters, further literature with the associated discussion is found within the body of our contribution chapters.

The primary research question answered by this chapter is:

What difficulties does Agent-Based Modelling (ABM) runtime inspection have which are reduced by the use of a Domain-Specific Aspect Language (DSAL), what are the barriers to DSAL adoption and what existing research has challenged these barriers?

3.1 Implementing ABM Runtime Inspection

Runtime inspection is one of the core concerns of scientific simulation, it is the means of viewing the data which can be used to generate results, mediate between theories and predict real systems as we use simulations as scientific apparatus. The use of proper apparatus can save a scientists time, allow a grounding for others to understand results from and increase ease of reproducibility (Minar et al., 1996). Although conceptually inspecting a model is a core concern for the scientist, in implementation runtime inspection is generally considered a cross-cutting concern which must be scattered and tangled across many modules amongst the concerns of the model itself at implementation-level abstraction. Providing facilities to aid the verification and validation of models is an important next research step by North et al. (2013) for the Repast Frameworks. As such, we now look at the literature to investigate the potential for new techniques which can be added to ABM frameworks or used as auxiliary libraries is a much-needed research direction.

The simplest method of runtime inspecting a model is instrumenting the model's behaviour code with desired output statements and using boolean flags for toggling this inspection. This produces simple runtime inspection which is fit for task and does not require any additional tooling or skillset for the programmer. Although this method seems to be immature software development, it is very appealing for short sprints towards finishing projects and getting the results required to perform further research. The disadvantage is that this method results in single-use code scattered and tangled throughout a project, meaning any changes to inspection or addition of new inspection means working through core concern code with a high likelihood of human error at some point. Furthermore, future projects cannot directly benefit from the apparatus designed for this project.

A more mature method is having runtime inspection code written into the ABM framework being used or an auxiliary library, so inspection features may be re-used across projects and human error is minimised by containing logic in one place. This minimises the boilerplate creation problem for each project yet still leaves calls to the inspection code scattered and tangled throughout the solution. While this is a cleaner solution than ad-hoc code scattered to inspect a model, the creation of a runtime inspection library or selection of a framework which provides appropriate inspection mechanisms requires foresight and overhead at the beginning of a project for a pay-off through the end of the project and future projects.

The reduction of scatter and tangling can be done by adopting Aspect-Oriented Programming (AOP), although as stated in Nusayr and Cook (2009) the implementation of runtime inspection in a domain may not directly map to implementations within programs written past simple logging of methods. These issues have been tackled by DSALs explicitly designed for runtime verification tasks. For example, Logical Automata for Runtime Verification and Analysis (LARVA) from Colombo et al. (2009) is a runtime verification architecture which uses a script as input which we consider a DSAL aimed towards the monitoring of real-time properties within Java programs. It compiles to runtime monitoring code with AspectJ aspects for instrumentation. This type of DSAL forms a wrapper over a General-Purpose Aspect Language (GPAL) to produce a suitable specification language for a broad domain. Domain-Specific Language for Instrumentation (DiSL) from Javed et al. (2016) is a DSAL which extends AspectJ with an extensible Join Point (JP) model and coverage of both Java and Android class libraries. This is an implementation-level DSAL which is intended to replace AspectJ as a base for runtime verification tools such as JavaMOP (Jin et al., 2012) and LARVA to provide functionality otherwise not available. An example of a tightly domain-specific DSAL

is Embedded Real-Time Systems Aspect Language (ERTSAL) from Sousan et al. (2007) designed for monitoring, evaluating and debugging of embedded real-time systems. Interestingly as embedded systems are predominantly written in C and C++; ERTSAL compiles to AspectC++ rather than the common AspectJ.

As such, the most mature method of expressing runtime inspection on models is using a DSAL designed for this task, either at the framework level or model level. A DSAL allows for a cross-cutting concern to be modularised from the core concerns of a program at the domain-level of abstraction improving processes of development (Soule, 2010). This combines the notational advantages of Domain-Specific Languages (DSLs) with the organisational advantages of AOP, both of which have been used throughout the ABM literature as covered in our Chapter 2.

To use a DSAL require pre-planning which is seldom given in short projects or article experiments where ad-hoc solutions may be added without planning. The main barriers to using this approach is a lack of knowledge about DSALs to be discussed as a program is starting and a lack of guidance in the literature for implementing DSALs in practical manors. The initial cost of implementing a DSAL can be paid back across many rounds of experiments and dissemination of results by reducing the complexity of writing experiments and improving software quality (Chibani et al., 2013). Furthermore, once the research ethos of a group has DSALs implanted within it, future projects are more likely to use the approach. We now look at the philosophical and technical challenges towards adoption of DSALs for ABM runtime inspection.

3.2 DSALs for ABM Runtime Inspection

Moving towards a solution for making DSALs available for ABM runtime we should consider the existing literature on DSALs for ABM towards runtime inspection of models. We then can move into DSL and DSAL techniques which are worth further examination. ABMs scientific nature means that having a language for runtime inspection during the experimentation stage is especially useful; this provides a division of labour between the implementation of experiments and conceptualisation of experiments as illustrated in Figure 3.1. A recent move towards having research software engineers to create languages for teams, working cooperatively with allowing more freedom for domain experts to write and share domain-level code could be used here. A similar approach has been seen with probabilistic programming languages where the different tasks of creating and using state of the art machine learning languages are separated into technical areas of domain experts, probabilistic programming to be dealt with by programming language and machine learning representation experts, machine learning to be dealt with by machine learning solver and compiler experts and inference engines to be dealt with by compiler experts (Defense Advanced Research Projects Agency, 2013).

The primary technical consideration for wide adoption of DSALs for ABM runtime inspection is the sheer variance of ABMs. Although abstract concepts of ABM are similar across the field, the implementation of these concepts and the styles of agents can vastly differ as seen from the discussion in Section 2.4.2.

The differences in implementation of a model's weave target can be avoided by controlling the weave target as in Gulyás et al. (1999). Multi-Agent modelling language provides a DSL environment where agents can be split into agents and observer aspects which are weaved together at DSL compile time into swarm

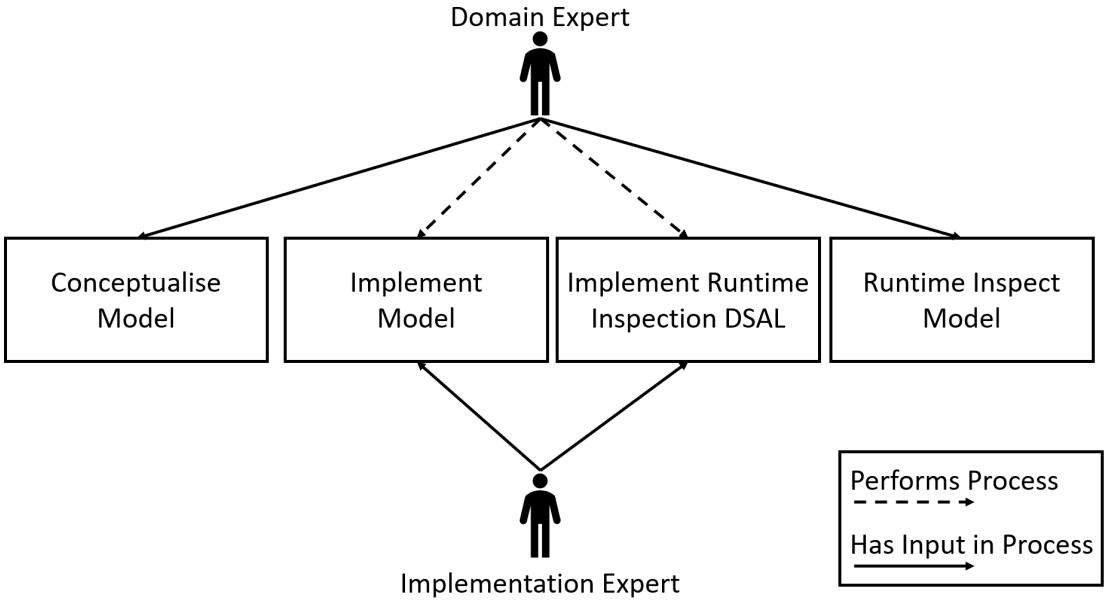

Figure 3.1: The division of labour between roles creating an ABM with accompanying runtime inspection DSAL.

observable agents using the xmc compiler and then can then be compiled to binary using an Objective-C compiler such as the GCC. As this language's compiler generates both the weave target and aspects during the same weave, it has generality across all code written using the DSL. An illustration of this process his shown in Figure 3.2. This DSAL is designed as a full approach to ABM development which contains runtime inspection aspects as a piece of the language. The implementation environment of the ABMs is delegated to the generated swarm, and the observable agent classes used are available without MAML. The benefit of this approach is forming an aspect-oriented abstraction level above the popular framework to allow for better modularity without sacrificing the use of a mature framework; the framework even moved towards creating a prototype GUI to populate the language for non-programming domain experts. This is especially useful for frameworks such as swarm which have a reputation for its steep learning curve (Allan, 2010).

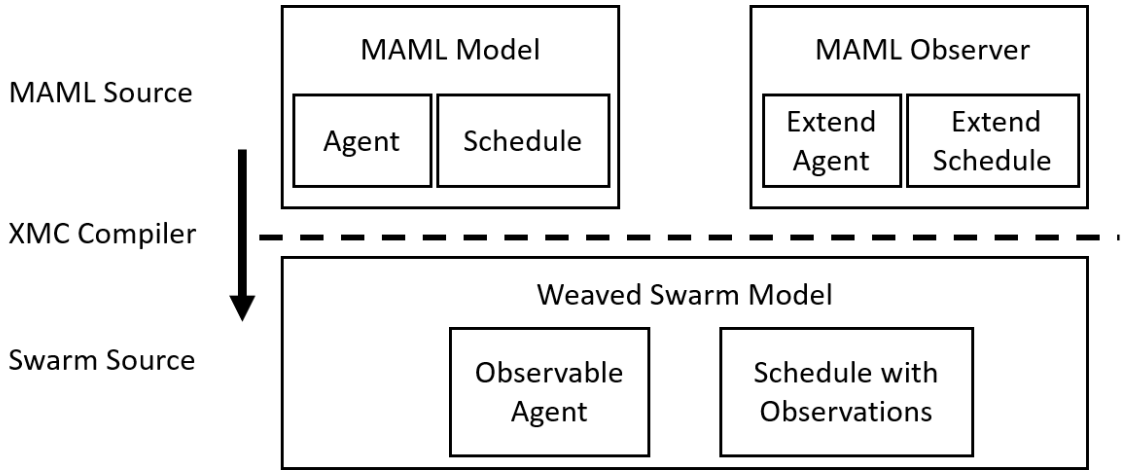

Figure 3.2: The MAML framework aspect-orientated splitting of observation and model code adapted from Gulyás et al. (1999).

Another implementation of domain-specific runtime inspection features can be seen in methods such as Virtual Overlay Multi-Agent System (VOMAS) from Niazi et al. (2009). The VOMAS technique is designed towards verification and validation of ABM by domain experts (subject matter experts in their terminology), allowing data and animation of errors at runtime. It comprises of creating a custom-built overlay agent system which can contain aspects validation from logs, animations or invariant checks, this is illustrated in Figure 3.3. This method relies on VO agents which are located throughout the model coupled with the implementation of simulation agents to relay information to the VO manager and watcher. As such, this is a novel agent-oriented technique which rather than dealing with runtime inspection as a cross-cutting concern, it translates it into a core concern of proving a virtual overlay agent system with the model. As this technique is based around the validation and verification of ABMs which require rigorous runtime inspection, they recommend domain experts should be involved in the development of the model and the custom-built VOMAS from the start of a project to ensure proper delivery to maintain in line with best practices for model creation from Law (2008).

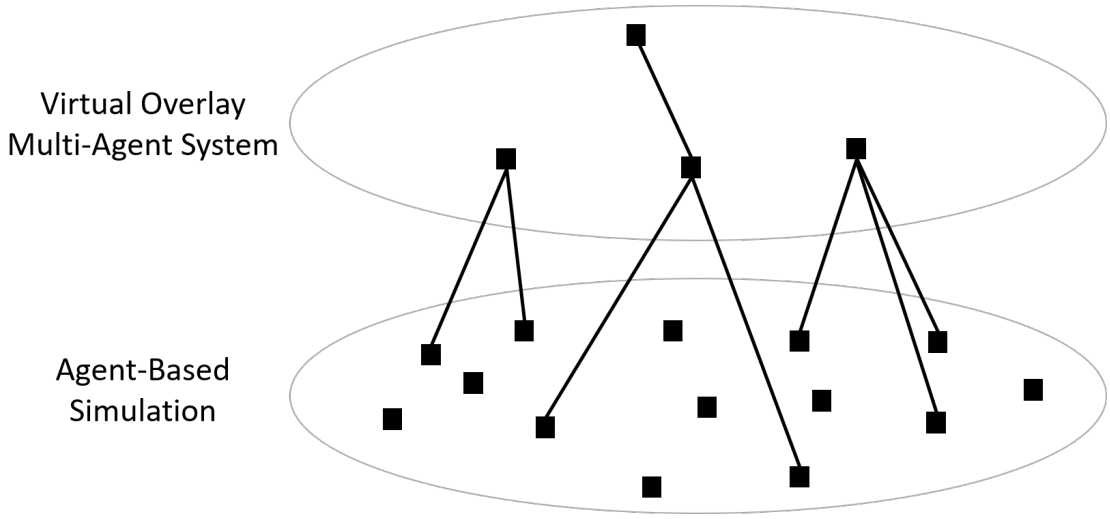

Figure 3.3: The VOMAS method of observing an agent-based simulation, adapted from Niazi et al. (2009).

While these approaches give novel modularity approaches to inspection code they are still coupled to and involved in the process of creating the model, meaning the aspect writer must be familiar with the base implementation. This stems from a use case where a model and its experiments are written simultaneously. We would like to separate the runtime inspection from the core model, so inspection code may be written as a model is developed or after a model is finished. Our intended runtime inspection language is detached from the implementation so that a domain expert who has a conceptual map of a model can perform experiments without implementation specifics of the model. Note that the domain expert may be the same person as the model developer although at a different time.

Because ABM is a vast and varied domain any attempt to cover the whole of the domain with a DSAL would be either large, restrictive or resemble a GPAL with some auxiliary functions added for ABMs. As we have covered in Sections 2.1.2 and 2.3.2 this could still be classed as domain specificity, for example having general-purpose representation and identification yet with domain-specific effect aiding runtime inspection. In the absence of a catch-all solution, developers may

want to create their own DSALs to work with their models or specialised generic base DSAL to a specific model they want to work on. This sort of work could be done during development time or when runtime inspection is required. If this is done for a model before runtime inspection is required only the domain expert role has to be played upon model completion instead of fishing through the model implementation. As research groups will already have implementations of their models within certain frameworks, the availability or creation of frameworks which works with their set of existing models. This will lower the barrier to entry for using a DSAL, which we defined as a primary reason for DSALs lacking adoption for projects previously. This brings us to an approach where the likely options are using a DSAL which has been targeted towards a popular framework or built into these frameworks as may happen because of the specification of this being an important way forward for ABM frameworks, creating a DSAL for in-group use which may be used across other projects by the group or creating a DSAL with intention to only use it once as in Hadas and Lorenz (2016). Furthermore, to properly use DSALs the people involved in the project must understand the philosophical frameworks of DSLs and AOP; and DSAL development requires a skill set which may not be available to the average agent-based modeller.

We now look at middle-out approaches from DSL research and how they can be applied to DSALs to allow the creation of languages for specific models or specific implementation concerns with specialisation towards models at a later stage.

3.3 Middle-Out DSALs

The middle-out process is a language-oriented technique which originated with Ward (1995) comparing it to top down, bottom up and outside-in development methods. The essence of this technique is creating a language which can effectively solve the problem, then implement and use the language separately. Directly mapping the concept of middle-out development for DSLs to a middle-out ABM runtime inspection DSAL means that you create a language which a person playing the role of domain expert would like to runtime inspect a program with, then a person acting as a language developer would implement the language and a person acting as a domain expert would inspect models using the language. This method fits perfectly with the workflow mentioned in the previous section, yet the previously mentioned extra considerations of DSALs and the field of ABM mean the middle-out process will need to be adapted to be usable in this scenario.

The middle-out approach is similar to other approaches based around language-oriented programming, and as such, there is significant overlap with approaches such as generative programming from Czarnecki and Eisenecker (2000) based around the idea of generation of software families, language-driven development from Clark et al. (2015) based around engineering languages using meta-modelling, software factories from Greenfield and Short (2003) aimed towards model-driven product lines and language-oriented programming from Dmitriev (2004) aimed towards projection language workbenches. Through this thesis, we focus on the middle-out approach because the process fits in well with the concept of a core model and cross-cutting runtime inspection.

The first mention in the literature of a DSAL being treated as a DSL frontend is from Bagge and Kalleberg (2006) where aspect-oriented languages are implemented using a DSL and a library written in a transformation language. The library must be written in a transformation language because the straightforward translation of DSL to library call is not possible for DSALs using simple macro-expansion because aspects may affect many parts throughout a program. They view aspects as meta-programs that transform the code of the base program, in this context aspect languages are domain-specific program transformation languages which hide the complexity of program transformation from programmers using notions such as JPs, pointcuts and advice. They illustrate their approach using a small error-handling DSAL extension to the functional language Tiny Imperative Language (TIL). To implement this approach DSALs are not directly weaved into the subject language, they are translated into transformation language calls to be performed on the base program at compile time. They claim provided the transformation language has a sufficiently powerful transformation library for the subject language this is an easy task, but implementing an arbitrary DSAL in this fashion may be too complicated for regular developers. This is especially true over large languages such as C or Java because the complexity generally comes from the subject language semantics rather than the DSAL. The weaving is performed at compile time by checking each function for alerts, performing logic to find active handlers on this alert, if multiple handlers are on a single alert sorting by precedence, and finally re-writing the function according to a template to weave. The compilation process is shown in Figure 3.4, the weaving process is done at compile time by executing the DSAL meta-program after initial syntax and type checking, once weaved the program is a pure TIL program which is compiled with an unmodified TIL compiler.

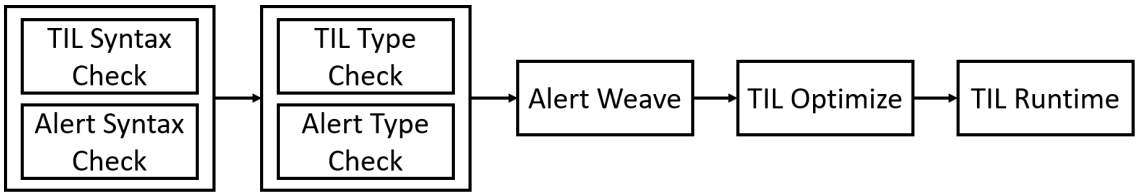

Figure 3.4: The compilation process of the TIL+Alert implementation adapted from Bagge and Kalleberg (2006).

This approach is suitable for people who require a simple DSAL and have a firm grounding in meta-programming or transformation languages. In programming language design, a first-class citizen is an entity which supports all the operations generally available to other entities in a given programming language. Second-class citizens are not fully integrated with a given programming language and its development environment, which may lead to difficulties during use. This method does not move towards the first-class adoption of DSALs yet provides a novel method of program transformation for implementing them without using a base GPAL. As this method transforms the code of the base program after compilation but before execution, all debugging must be performed on generated code yet is effectuated in pre-generated code. A slightly adapted version of this method could be used to do runtime weaving using meta-programming or create a simple to implement source text pre-processor using regular expressions for small prototype DSALs on consistent code bases.

We can now move onto the first explicit mention of language-oriented programming towards DSALs in the literature, with a position paper from Hadas and Lorenz (2015) who consider DSALs in their current state to be second-class citizens in the language space; second-class DSLs because they are harder to develop through lack of cohesive tooling, and second-class AOP languages because they are harder to use through incompatibility with general aspect tools. This results in the practical language-

oriented modularity project from Hadas and Lorenz (2017) which moves towards bringing language-oriented programming productivity to DSALs as first-class DSLs and AOP languages by providing a process and DSAL infrastructure to allow the creation of DSALs throughout the software modularisation process. This requirement for the first-class development of DSALs comes from the problem that off-the-shelf DSAL solutions will likely not be suitable for an application even in the same domain because of changes in weave target implementation and domain expert terminology. Thus, most projects will have to either create bespoke DSALs or modify existing ones to fit their needs. As we noted this is especially pronounced for ABM because of the wide variance of models available and schools of thought within the domain. Moving towards language-oriented programming for DSALs means that in the same way we can create a DSL as a productive measure to solve a particular type of problem we are having, a DSAL could be created with similar productivity because appropriate methodology and tooling is available. Their approach creates a DSAL using a language workbench which transforms the DSAL code to a kernel language that is based on a GPAL. This gives a first-class DSL creation process using a language workbench and first-class GPAL Integrated Development Environment (IDE) support for DSALs which are in some sense reducible to GPAL code. Towards this goal they state it is preferable to use mature, general software to maximise tool support and features, for example, language workbenches such as Spoofax, MPS or Xtext and aspect languages such as AspectJ or Spring. They then comment that the approach of directly generating GPAL code has two main pitfalls:

- The semantic gap between DSAL and GPAL often creates a non-linear mapping of semantics between the languages. This mapping must be protected when a program's JP signatures change or previously matching pointcuts may behave unexpectedly on new base code. Furthermore, the semantics of the DSALs

generated code may clash with existing GPAL or DSAL code if there is no composition measure in place. For example, the transformation done by the DSAL in TIL+Alert from Bagge and Kalleberg (2006) could invalidate aspects in another DSAL.

- The abstraction gap between DSAL and GPAL means that development tools at the generated GPAL source level will not represent the code in the DSAL at the appropriate abstraction level. For example, aspects which are written to detect changes in agents which requires a view of multiple agent states would appear at the implementation tooling level to merely be checking the status of arbitrary variables with no relation to the overarching goal of the original pointcut advice.

They address these pitfalls through the addition of metadata to the GPAL, in their implementation this is done by extending the AspectJ compiler with domain-specific annotations. These annotations bring the ability to suppress JP shadows, control the order of execution for conflicting advice, and a means to point to an external source of the DSAL code for use when navigating aspect code. As this approach uses a GPAL code transforming weaver, there is no inherent disadvantage in runtime performance compared to the direct use of a GPAL, and their compiler extensions will work with standard aspects programmed for that version of AspectJ meaning they could be used in a project which already uses AspectJ.

This method places high importance on retaining both DSL and AOP first-class status in a cohesive package for language development and use akin to what a language workbench provides for a DSL. Their general synopsis comes in the form that DSL workbenches do not provide back-end weaving capabilities, and AOP

composition frameworks do not provide domain-specific front-end user experience capabilities, and both are required in a cohesive process to produce a first-class DSAL.

3.4 Direction

The literature we have gone through focuses on methods which can be used generally across abstract sets of DSALs, through this thesis we focus on our target domain of ABM runtime inspection. While we only relate our work to this domain at the abstraction level of implementing DSALs, this domain is a representative domain of many others, especially those which comprise of autonomous components within some bounded environment such as multi-agent systems, robotics and distributed teller systems.

The context of our domain relying on short-term projects or implementations towards articles means we must find a method which minimises start-up cost to provide practical DSAL adaption while being capable of facilitating effective re-use or repurposing towards new projects and articles. We adapt middle-out practices from a pure DSL approach towards a DSAL as a DSL with a semantic model that provides transparent AOP. The aspect orientation is provided transparently by static GPAL JPs being passed through the semantic models Domain-Specific Join Point (DSJP) processor. This allows the use of existing mature tooling and existing skillsets to create and re-appropriate custom DSALs. This is possible because the abstraction target of our language is at the domain-level to be used by the domain expert rather than the implementation-level to be used by an aspect-oriented software engineer,

meaning the aspect orientation provided by our middle-out process can be transparent to a user. This is very important as it allows us to not focus on first-class AOP for the top layer aspects, but only for the bottom layer utilities.

We believe that for domain-level DSALs using a middle-out approach allows the front-end DSL development and back-end AOP implementation to be implemented separately without consideration of providing first-class aspect orientation. This is because the language user in a domain-level DSAL will not be looking through implementations for AJDT markers but can trust the DSAL will provide appropriate weaving for the written aspects transparently. The developer of the language is free to use a fully tooled GPAL or create their own weaver because from the middle layer's point of view it is merely important the necessary utilities be implemented. The implementation and use of the front-end DSAL will be at the same productivity level provided by using a language workbench for DSLs, albeit with a different process than the creation of a non-aspect-oriented DSL.

Problems of semantic and abstraction gaps are not relevant for middle-out development targeting a semantic model because the language is based around the semantics of the model being populated. This approach means that the semantic model must be checked for feasibility during its development but once instantiated the semantics of language calls will match the model's performance. The abstraction of the user's input to the model is at the domain-level rather than the aspect-oriented level. Any issues with the aspects found by the semantic model are issues for the back-end language developer to deal with at the correct level of abstraction for their static GPAL code or the semantic models processing engine.

Aspect composition problems stem from a choice to directly generate GPAL code which matches the semantics of the DSAL code; this problem can be avoided by using static AspectJ aspects which are used as input for a DSJP interpreter. This is a runtime performance hit which can be reduced using partial evaluation. Despite the runtime performance hit this removes the developer time hungry step of creating a working compiler from the DSAL to the GPAL code in place of the more straightforward task of creating and populating a semantic model for the DSAL. As the developer time to implement a DSAL has been considered a hindrance to DSAL adoption, this may be a worthwhile trade. We will investigate the performance of aspects as a primary concern through our experiments. This approach allows composability with other handwritten GPAL code, especially as our aspects are contained in their own file and any clashes will be apparent at compile time for the developer of new aspects written in the same GPAL. Further work on composability could be done by combining ideas from the literature, such as, using the AspectJ compiler extensions from Hadas and Lorenz (2017) which is compatible with the semantic model based middle-out approach as a bottom layer implementation method. The compiler changes could provide benefit over using a standard GPAL for writing the static aspects because composability with other DSALs generated aspects, or other's GPAL code could be dealt with explicitly.

3.5 Summary

This literature review has covered the initial problem of ABM runtime inspection methods, currently available implementations of runtime inspection for ABM with emphasis towards scientific projects, and finally framed our research direction of middle-out DSALs for runtime inspection of ABM.

We have discussed the benefit to giving DSALs to domain experts to inspect ABMs. This is through moving to understandable domain-level syntax and reduction of scatter and tangle within core concern code, allowing for experiments to be created, modified and toggled freely. DSALs could be implemented towards a specific model directly or implemented as a general core across an implementation middleware which can then be extended towards specific models. We explore these techniques in Chapters 4 and 5 respectively. To allow the adoption of DSALs for scientific projects a philosophy and implementation method must be readily available. Currently, there is no cohesive literature on how the middle-out process for DSLs can be adapted to a runtime inspection DSAL. This hampers the adaption of DSALs because implementation technique requires full knowledge of both DSL implementation and aspect orientation before it can be attempted. Through Chapters 4 and 5 we lay out a clear process with illustrative implementations that apply to ABM runtime inspection.

In Chapter 4 we begin our investigation into how a DSAL can be middle-out, moving into external DSAL model-specific implementations for Sugarscape and Kawasaki models within our in-house ABM framework Animaux. This is our first step towards realising DSALs for ABM projects and is used as a base for Chapter 5. In Chapter 5 we move towards a solution for middleware targeting DSAL implementations which can be used across many projects with model-specific internal DSALs using the middleware target of Foundation for Intelligent Physical Agents (FIPA) ACL communications in JADE. This is a solution where a mature core can be created once for dissemination between projects, and the relatively fast implementation of extension DSALs allows a model-specific environment for domain experts. This could even be used to form open source projects released as add-ons to

major ABM frameworks. Finally, in Chapter 7 we will conclude on our research and the potential for the use of middle-out techniques in the broader research community with reference to this chapter's direction.

Chapter 4

AnimauxRI - Runtime Inspection Targeting Specific Models

This chapter is the first of our two main contribution chapters, focusing on our primary investigation of the middle-out process for Domain-Specific Aspect Languages (DSALs) and implementation towards specific models. This approach is a high specificity, low generality approach, and as such, is a very domain-specific area to focus on. The target audience for a DSAL such as this is somebody using novel in-house technology without domain-oriented interfaces for a large number of different experiments, written at the domain expert level, on a small set of models. We first motivate and present our approach, followed by details of our implementations, closing with experiments, results and discussion. Our implementations are extensions to the Animaux Agent-Based Modelling (ABM) framework (Hawick, 2012a; Preez et al., 2012) written using a combination of Java, AspectJ, Xtext and Xtend.

The primary research question answered by this chapter is:

How can we implement a middle-out DSAL for specific models without domain-oriented interfaces?

4.1 Motivation

ABMs are used as apparatus for simulating data for scientific articles, a means of capturing this data dynamically is runtime inspection. Runtime inspection is an indispensable tool in an agent-based modellers kit when using software as apparatus for scientific articles. Each model will have specific characteristics which must be recorded, understood and then conveyed to a reader. Runtime inspection is useful for this in that it is the querying and control of execution of programs; this is both a high boilerplate and cross-cutting concern task. Having an infrastructure to base runtime inspection code from dramatically helps the development of such systems.

Through this chapter we focus on two reflex driven ABMs, we explain our case study models in Section 4.2. Common runtime inspection tasks for this type of model include:

- Logging, saving or graphing agent, environment and observer level statistics of a model for use in populating results for or presenting research
- Breaking execution on certain events much alike a breakpoint
- Augment visualisation of agents within a model, for example, making anomalies stand out
- Implementing domain laws such as events to never or always happen

Current state of the art object-oriented frameworks for ABM are excellent for expressing the core components of an ABM, but leave cross-cutting concerns such as runtime inspection scattered and tangled throughout models using implementation-specific syntax. This scattering and tangling is inevitable using object-oriented technology because a cross-cutting concern affects the performance or semantics of many components in systematic ways which cannot be cleanly separated into a different component. A further consideration is the mapping of implementation code to domain-level constructs, preserving the semantics of runtime inspection queries. In an implementation of a model, a variable within an agent may change multiple times through a step as it processes although in the semantics of the model only the final version is important. For example, the wealth of a Sugarscape agent as it moves and degrades, yet the inverse may also be true as in Kawasaki model particles which may legitimately exchange multiple times within a single step. As such, directly monitoring the implementation variable may or may not be sufficient to inspect the model's behaviour depending on the semantics of the model.

The above problems are exacerbated when a framework is in the early stages of development or has poor documentation because this encourages incorrect assumptions to be made. This is more of an issue where people who have not been involved in development or have forgotten specific implementation details want to run experiments on a model.

As such, we want to have a means of runtime inspection which allows the scientist to better observe, interact and understand the model. The objectives we have highlighted to providing an ideal solution for this are:

- Separating the runtime inspection code from the simulation code
- Controlling the coupling of the simulation code and observation code

- Having coherent defined semantics
- Expressing the display or exportation of data without re-writing boilerplate code
- Giving solution level Integrated Development Environment (IDE) support for the user

DSALs have been deemed an excellent match for the domain of runtime inspection through the literature (Mehner and Rashid, 2002; de Borger et al., 2012; Colombo et al., 2009; Javed et al., 2016). Specifically, in our case, a DSAL could be used to provide a small set of coherent model level abstractions to shield the user from any concern except for runtime inspection of our chosen model while writing experiments.

In work by Fabry et al. (2015) it has been found that most implementations of DSALs are ad-hoc extensions of languages for single purposes, potentially because of lack sufficient tooling or lack of understanding of DSALs. Using specific tooling requires a steep learning curve with a resulting high vendor lock-in and low transferable skills to other DSAL frameworks which results in low adoption.

We believe a structured approach using mature and reusable software could result in better production of DSALs, and adoption of DSALs for more projects once a team has used them. Language-oriented programming's middle-out architecture has been related to DSALs by the language-oriented modularity project from Hadas and Lorenz (2017) and the AWESOME project from Kojarski and Lorenz (2005, 2007) with an emphasis towards aspect composition. Although they believe the Domain-Specific Language (DSL) language workbench with GPAL backend is not a suitable technique because the semantics of the DSAL code may not map to the semantics of the GPAL code. In Section 4.3 we propose a language workbench with a semantic model implemented using a GPAL is a suitable approach for model-specific

DSALs and can be done using mature, stable software projects which have high transferable skills for developers. This is applying lessons learnt from middle-out DSL development to DSAL research. In Section 4.4 we produce an implementation of our approach using an Xtext DSL over a Domain-Specific Join Point (DSJP) interpreting framework utilising AspectJ General-Purpose Join Points (GPJPs) as input.

4.2 Case Study Models

Through this article, we focus on Kawasaki and Sugarscape models. These models have been chosen because through their simplicity they display the essence of two types of ABM applications. The Sugarscape model is an ant-based model where choices are dictated by environmental factors. Kawasaki based models are exchange or spin based where choices are dictated by interacting with neighbouring agents.

4.2.1 Sugarscape

A Sugarscape model is an agent-based social simulation based upon the rules presented in Epstein and Axtell's (1996) book growing artificial societies. It serves as a means to demonstrate the abilities of reflex ABMs which only interact with their environment.

The Sugarscape model involves a distinct set of agents that move around a spacial system, usually a regular mesh. There might be some regular field of material such as wealth or some other spatially-oriented property spread around the whole mesh, but not every spatial cell is occupied by an agent. This differentiates Sugarscape from cellular Kawasaki models where all cells do have a single agent. We illustrate the

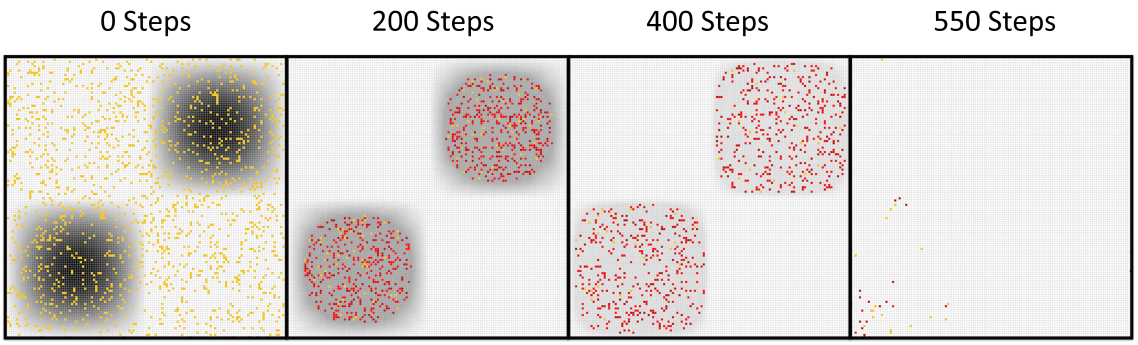

Figure 4.1: Example Sugarscape model visualisation at steps 0,200,400 and 550.

list of Sugarscape style agents as travelling around a regular mesh, but this model generalises to a real-space particle-based model where agents need not necessarily be restricted to discrete integer positions.

This model is illustrated in Figure 4.1 where 0 steps shows a random distribution of ants placed over an environment with 2 peaks of sugar. 200 steps shows the gathering of ants around these two peaks, with 400 steps showing a spreading apart as sugar supplies run low. 550 steps shows the population of ants coming to an end as there is not enough sugar to sustain activity.

The Sugarscape Algorithm used in this work is based on a sugar-field of N sites with N_A agents distributed across it spatially. We have simplified the model so that I is 2 units of sugar and the costs of moving C_M and surviving C_S are one unit of sugar each respectively. We restricted the models so that only one agent can occupy a site at a time, dead agents are culled from the system to stop blocking the movement of live agents.

Throughout the experiments the model is populated using the common twin peaks sugar pattern. It is useful as it clearly gives an unfair advantage to those agents who are near one of the two peaks and disadvantages agents located on the flat plains.

Algorithm 1 Sugarscape Agent Model.

```
choose lattice size, shape, eg square 256^2
choose neighbourhood 𝒩 eg Nearest, Moore, or Radial
for all runs do
  initialise N sugar-field sites by pattern eg Flat/Peaks
  initialise N_A unique agent locations randomly
  for all time steps, e.g. 500 do
    for all agent i ∈ N_A do
      identify unoccupied neighbouring site with most sugar
      pick one at random tie-break if more than one
      move to chosen new site
      metabolise, using up "cost C_m of moving" sugar units from wealth
      metabolise, using up "cost C_s of surviving" sugar units from wealth
      consume, accumulating income I units of sugar from new site
    end for
    remove dead agents (with negative wealth)
    record measurements, e.g. Gini coefficient
    exit when no live agents remain
  end for
end for
normalise averaged measurements
```

For our purposes we can break down a Sugarscape model into its constituent observable elements as shown in Table 4.1.

Agent Type	Events	Properties
Ant	Movement, Wealth Change	Wealth, Location, Neighbourhood
Environment	Occupation Change	Sugar, Occupation
Observer	Step	Step Number, Gini Coefficient, Ant List, Environment List

Table 4.1: Summary of Sugarscape constituent elements.

4.2.2 Kawasaki

A Kawasaki model focuses on the evolution of a model through 'spin-exchange' or 'spin-flip' particle dynamics. It serves as a means to demonstrate the abilities of reflex ABMs where every space of a lattice is an agent.

Models based on a Kawasaki exchange pattern use a spatial arrangement of agent state variables on a mesh or network. Each agent has one or more state variables such as atomic species, wealth or opinion. Kawasaki exchange dynamics involves selecting an agent; selecting one of its neighbours; considering the consequences of exchanging the two agents spatially; and either carrying out that change or not according to a computed probability based on those consequences. Repetition of this basic algorithm, often to an initially random mixture of agents, can lead to surprisingly complex spatial patterns of agents that can spontaneously segregate, or cluster, or form other patterns with properties that emerge from the system as a whole and which are not prescribed by the individually encoded agent behaviours.

This model is illustrated in Figure 4.2 where 0 steps shows an initial square distribution of A-species agents, upon a vacant background of B-species agents. Then 1000, 2000 and 4000 steps show the dispersion of the initial square as A-species and B-species agents exchange position with one another.

The Kawasaki Algorithm used in this work is based on a mesh occupied by A-species agents, represented as a 1 and B-species agents, represented as a 0. This can be simplified as A-species agents in a vacant background of B. Throughout the experiments the 'spin-exchange' Kawasaki dynamics method (Kawasaki, 1966) is used to simulate exchange probability where neighbouring site variables c_i and $c_{i\pm1}$ are exchanged. This interaction between species is performed by a function consisting of an assignment of bonds between neighbouring agents. This function has

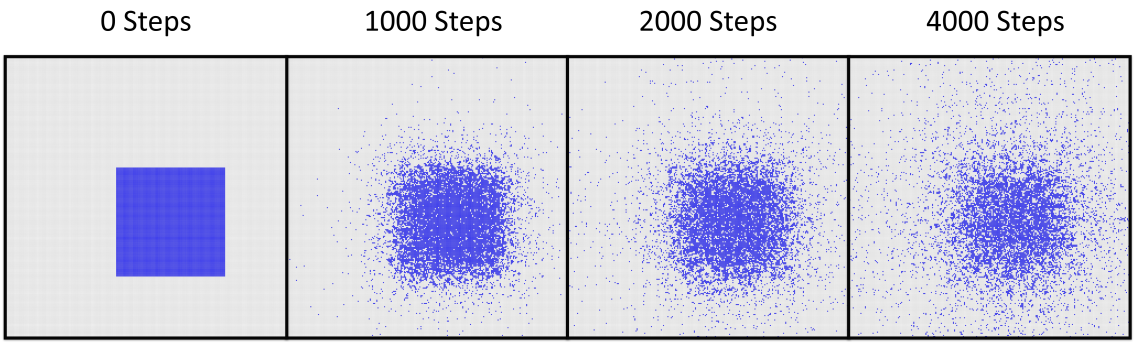

Figure 4.2: Example Kawasaki model visualisation at steps 0, 1000, 2000 and 4000.

no explicit dynamical scheme associated with it, and so one is imposed artificially using Monte Carlo techniques to make the model system evolve between different microstate configurations.

Algorithm 2 Monte Carlo Kawasaki Model Algorithm.

```
N = L^2 sites on square lattice
for all runs do
  initialise sites with p_v = 0.5 vacancies
  for all time-steps do
    for all sites i in random order do
      choose a random neighbour site j
      compute energy change if i, j exchanged
      if energy falls then
        accept change and do exchange
      else
        compute Metropolis probability p
        generate random probability r
        accept change conditionally on r < p
      end if
    end for
  end for
  average results over runs
end for
```

For our purposes we can break down a Kawasaki model into its constituent observable elements as shown in Table 4.2.

Agent Type	Events	Properties
Particle	Selected Site, Selected Neighbour Exchange	Type, Bind, Metropolis Probability
Observer	Step	Particle List, Step, Exchange Numbers

Table 4.2: Summary of Kawasaki constituent elements.

4.3 Analysis, Design and Approach

Through this section, we explain the process of choosing our approach and how we can conceptually use it for ABM. As has been covered in Section 2.1, a majority of the productivity benefits for this case can be provided by the movement from ad-hoc code on a per model basis towards a general-purpose framework designed to allow re-use of boilerplate code in the area. The next step can be adding domain-specific support for models which raises the software maturity by reduction of semantic gap and hiding boilerplate away from the user. Although to this point the scattering and tangling of the cross-cutting concern is merely moved away from the user, to alleviate the cross-cutting problem we must use aspect-oriented techniques. To do this, we move from our object-oriented environment to an aspect-oriented environment for a marked improvement in scatter and tangle, then allowing the addition of domain-specific features to aid the framework further forming a DSAL. This process is illustrated in Figure 4.3, although using a GPAL and DSL are visualised as the same maturity in this illustration this is dependent on circumstance. GPALs give a better reduction of scatter and tangling, DSLs reduce the requirement to directly write scattered and tangled code.

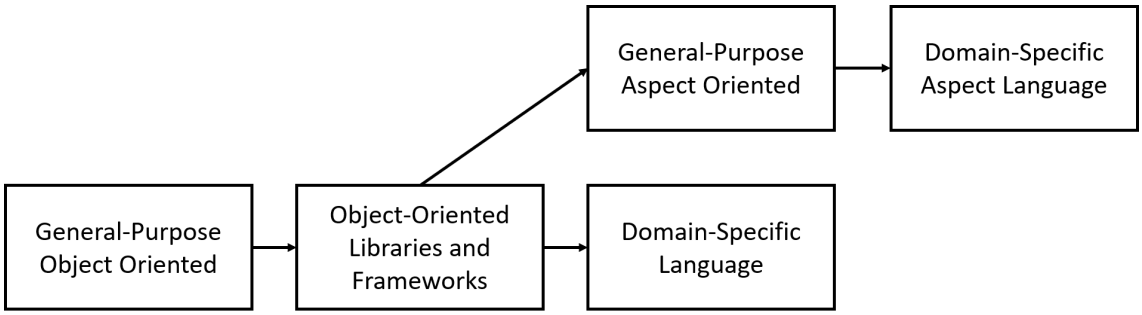

Figure 4.3: Maturity of runtime inspection through object-oriented GPL to DSAL.

We now consider how is best to implement a DSAL framework over an object-oriented framework without domain-oriented interfaces through the lens of language-oriented programming.

4.3.1 Middle-Out Language-Oriented Programming

Language-oriented programming is one of the many umbrella terms for a type of development which has a significant focus on the use of DSLs towards some end. Through this thesis we focus on Ward's (1995) definition of language-oriented programming based upon middle-out centric development which specifies the focus and order of the development process. Middle-out development's thesis is that a DSL can be used to package domain knowledge, which in turn can be used to dramatically reduce development effort while increasing maintainability and enabling reuse. As such, the first stage of development should be a definition or selection of a language which is suitable for the task. Only then moving onto system implementation using the language and implementation of the language, probably using some existing language forming a cascade of abstraction levels through projects. The order of implementation may change, and in many cases will be an iterative process with many layers of languages but the core idea of using a DSL as the structure for a task is required.

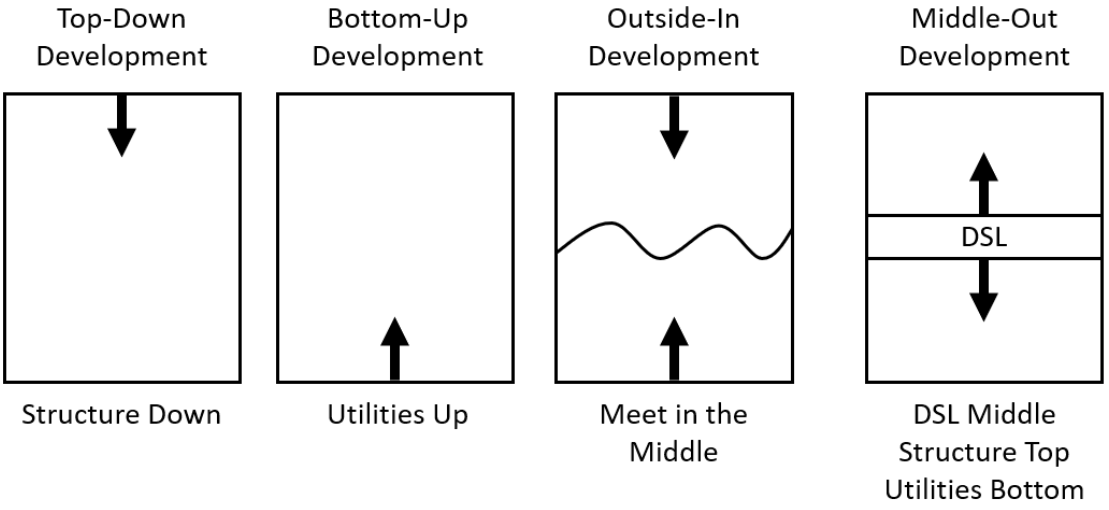

Figure 4.4: Software development processes adapted from Ward (1995).

Middle-out development is compared to top-down, bottom-up and outside-in development methodologies by Ward (1995). Middle-out provides benefit by not requiring a complete concept of the system to begin, giving domain context to bottom-up utility elements and formalising the boundaries of the system in the DSLs semantic model. This middle layer forms an abstract machine that the top-level problems can be written using and low-level utilities can aim to implement. This is much like earlier ideas for software productivity such as virtual machines for program families by Parnas (1976, 1979). The four development processes are illustrated in Figure 4.4, where arrows denote development activity.

As software development is an ever-changing environment, recent agile methods should also be applied to this process because changes in the needs of a middle layer throughout development are inevitable. Rapid prototyping of the DSAL may be the appropriate solution in some cases where the DSLs implementation can be focused on to achieve complete knowledge of possible system boundaries, or DSLs use to attain if it is complete for a task.

To allow iterations of the middle layer we prefer using a semantic model to separate the implementation and parsing of the language. Using a semantic model makes language definition and parsing easier by giving an abstract data type which matches the kind of data captured in the DSL, which can be updated throughout development as new requirements arise.

The use of a semantic model allows the parsing of the language and the testing of the domain model to be done separately which is especially useful when they are implemented in different frameworks. For example, in our illustrative implementation where a DSL is implemented in Xtext, separately to a DSJP interpreter using Java and AspectJ Join Points (JPs).

Currently, we have focused only on language-oriented programming for DSLs, which begs the question of what differences there are for DSALs. Bagge and Kalleberg (2006) has noted that simple macro-expansion as found in DSLs is not sufficient for DSALs because of their cross-cutting nature. Language-oriented modularity by Hadas and Lorenz (2017) also comments on the preservation of semantics by current methods of implementing DSALs. We now explore the extra considerations which aspect orientation brings compared to the standard DSL middle-out scenario.

4.3.2 How a DSAL can be Middle-Out

A middle layer is based upon a high abstraction level concept in both DSLs and DSALs, but DSALs have the extra consideration of a weaving target. This dependency means that although a DSALs middle layer is based upon a conceptual model, the bottom layer must be coupled with the implementation of a specific ABM as shown in Figure 4.5.

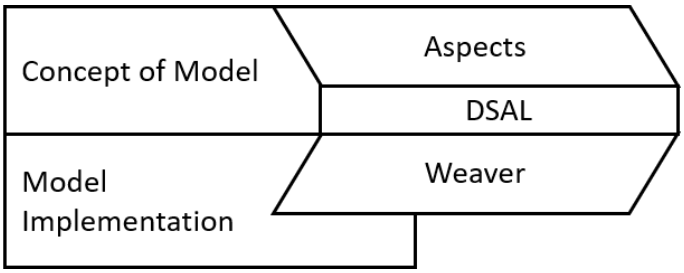

Figure 4.5: The coupling of a DSAL to the concept of a model, with the weaver coupled to the implementation of a model.

This creates limitations which need to be taken into account at the initial creation stage of the middle layer and future iterations. As we are using a semantic model for this, the semantic model essentially contains the JP representation of the system. Knowledge of the underlying weave target is required to know what identification and effect are feasible. This is less of a problem for a General-Purpose Aspect Language (GPAL) because their target is the static grammar of a language, thus supported and unsupported JPs are the same throughout any program. The issue for a DSAL, if implemented using a GPAL is GPJPs may not correspond to domain events. Examples of limitations in AspectJ are no JP for loop iterations and a lack of access to auxiliary information at a JP such as no array index support within the set JP.

Because of the difficulty in guaranteeing DSJPs being accessible in the target program, the process of creating the middle and bottom layers must have higher priority than creating top layer programs for a DSAL. This means the formal specification of the middle layer should begin with a specification of a semantic model which can be populated by available DSJP, then once feasibility has been established, middle layer DSAL development and top layer programs may be worked on.

We define the JP model of our middle layer using the specialise, aggregate, create model from Fabry et al. (2015). This method of creating DSJP uses GPJPs as a base JP which are then specialised, combined or added into program flow. This is achieved through a three-step process.

- Step 1: define a DSJP which is required.
- Step 2: find the plausible GPJPs which may be specialised, aggregated or created to provide this DSJP.
- Step 3: combine these into a specification for that DSJP.

This method of DSJP creation means that JPs may be conditional non-atomic events in the running of the host program. This makes before and around advice problematic to implement as it requires some lookahead or backtracking mechanism. As such we have taken the choice to allow advice to only be fired after a DSJP is triggered, removing options for before and around as is provided in AspectJ. For future work, it is plausible to imagine useful applications for pre-emption of events in deterministic simulations, although this would be with a considerable performance decrement.

A representative amount of runtime inspection can be performed using only after advice where aspects are used primarily as a logging tool for offline processing. The important consideration is consistent semantics, such as all timings of events are after an event. Interactive runtime inspection methods may be detrimentally affected through being limited to after behaviour as JPs would ideally be placed before an event occurs giving time for a user to assess situations. Examples of interactive runtime inspection behaviour include: augmenting the visualisation of simulations, corrective warnings or pausing simulations.

As runtime inspection of models should not affect the underlying model, we must also consider how we can control the effect advice has on the resulting model's core semantics. Using an external DSL for advice which is controlled by our environment, this can be solved by containing the potential action by the user. If using an internal DSL or direct general-purpose advice, this requires more consideration, as runtime inspection advice could change the results of the inspected code. It should be noted that for some use cases such as augmenting visualisation, constrained effect on underlying framework code is required.

In summary, a middle-out process for DSALs must place focus onto the representation, identification and effect of JPs within the middle layer. This can be achieved through decoupling the bottom and top layers by creating a semantic model during middle layer development. This semantic model should be populated through the top layer aspects and allow the population of weaving engines in the bottom layer.

4.3.3 Application to ABM

ABM is an interesting target for a runtime inspection language because it allows the language to not only be an executable runtime inspection language saving time for the developer but also a means of communicating what runtime inspection is happening to a model during hypothesising, dissemination and repetition of research. The use of a DSAL means that this process can be done through iterations of the modelling process with great reuse of code, retaining the semantics of experiments throughout model development. The behaviour level description of a model will remain the same regardless of implementation details. This means a description of agent events can be written without knowing the full details of the implementation, assuming the DSAL

is maintained along with changes of the model when required. The ABM process suits the required heavier emphasis on the bottom layer of middle-out development because a majority of models will be conceptually defined with information on what must be inspected. This gives a solid base for the initial creation of DSJP at an early stage, allowing a dialogue between domain expert and language developer which may improve the implementation of a model.

The selection of DSJP for ABMs is simple because the behaviours of simple agents form direct cause for inspecting the environment, things such as wealth movements in Sugarscape ants or exchanges for Kawasaki particles can be highlighted as important and added to a DSAL's JP representation. Macro behaviours which are the result of emergence are by definition far harder to assign a JP, and interesting work around them would be what states at JPs may lead to behaviours occurring.

To retain the integrity of models inspected by our code we limit the runtime advice possible to external logging, changing visualisation artefacts and pausing of stepping in the simulation. This means that runtime inspection code written using the DSAL will not affect the results of experiments, and as such, can be used as a safe method of extracting results from a model.

4.4 Implementation

Through this section, we will discuss our implementation of a middle-out DSAL over the Animaux ABM framework. We discuss our implementation through the lens of middle-out's three layers. First, we discuss the language definition and considerations taken throughout its definition; secondly, we discuss the implementation of the language using a semantic model technique; finally, we discuss the intended use of

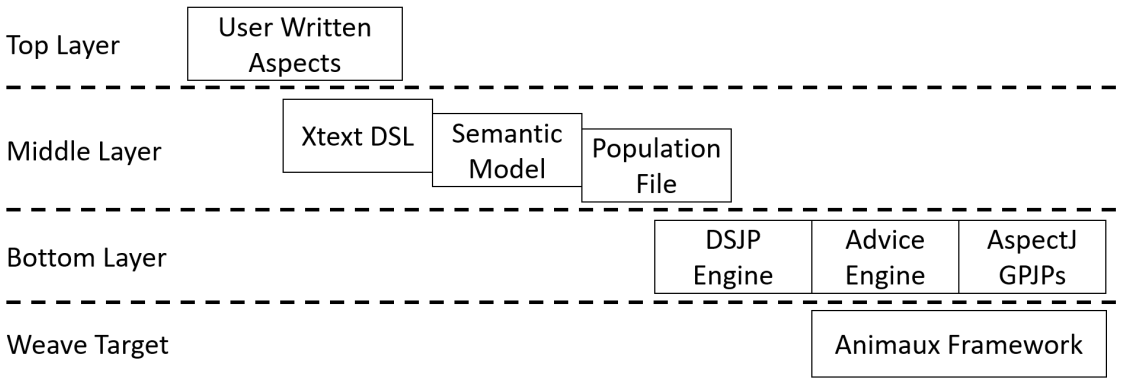

Figure 4.6: The top, middle and bottom implementation layers of AnimauxRI in relation to a target.

the language with regards to the next sections experiments. It should be noted that although we discuss the bottom layer before the top layer, these two layers can be independently developed in any order.

The architecture of our implementation is displayed in Figure 4.6. Our middle layer is an Xtext DSL which populates a semantic model, this semantic model then generates code for our bottom layer. Our bottom layer is comprised of AspectJ GPJPs and a DSJP runtime which interprets GPJP into DSJP and controls advice. The top layer of our DSAL is intended for users and they need not be aware of the implementation of the lower two layers. Using this approach allows us to use the mature Xtext system for creating our DSL, the mature AspectJ system for weaving GPJPs and domain-specific engines to manage application-specific DSJPs.

4.4.1 Middle Layer: The Language

In line with the DSL philosophy of building abstraction levels, the tooling we have used for our middle layer design and creation is Xtext. A language workbench for making DSLs which have good default support for generators, validators and IDE plug-ins. This is done by utilising ANother Tool for Language Recognition (ANTLR)

for parsing and Eclipse Modelling Framework (EMF) models for tree storage, making for a simple experience to make simple DSLs with defaults and opportunities to work with changing the back-end structure of things if required by advanced users. The DSL produced by this suite has the semantics of a DSAL provided by the DSJP interpreting runtime explained in the bottom layer section.

Language Definition

The language definition forms the basis of the top and bottom layers, which may be implemented independently of each other. As such, a definition should be formally specified to allow both a specific and immutable target. In large projects, such before-development waterfall type definitions are bound to require changes, especially during the early stages of development. An extra consideration for an aspect-oriented middle layer is the implementations of base programs will differ, so the domain-specific representation must be generic across all back-end systems. The benefit of a language being very domain-specific is the required vocabulary should be stable, so definitions can be made from the domain jargon which will be consistent across different implementations (Mernik et al., 2005).

In a DSAL environment to ensure plausibility of the middle layer, it is required to focus on the bottom layers abilities as well as the middle layer far more than is required in a standard Turing complete environment. The middle layer needs to be defined with consideration not only to the domain but also the intended weaving toolings potential given a set of target programs. An example of this is if using a pure AspectJ weaving back end, arbitrary positions within for loops cannot be captured directly because AspectJ's GPJP representation does not include mechanisms for capturing loop executions. This can be remedied by using a tool such as the aspect

bench compiler to include for loops (Allan et al., 2005; Harbulot and Gurd, 2006), create an additional static analysis solution using a parsing tool such as ANTLR to locate for loops and add them to the JP stream (Parr, 2013) or using a bytecode manipulation framework such as ASM from Bruneton et al. (2002). If the designer does not want to deal with these workarounds or the tooling used for the language backend is set in stone they will need to define the language with semantics which can be captured by that back-end tool, either by limiting the language features, enforcing rules upon targets or limiting the set of programs the language is intended for use with.

We begin by forming an informal specification of general-purpose Aspect-Oriented Programming (AOP) tasks which will be required to form the DSJP to runtime inspect the chosen models. This step is an exploration of potential problems with implementation to decide if anything needs to be done or scope needs to be limited before moving forward with DSAL creation.

The requirements we have found for this project are:

- Inter-type declarations. To augment storage of state within agents such as previous positions.
- Beginning and end of steps. Specialisation for step counting and use as a scaffold for DSJP which require checking of state on a step-by-step basis. This would require extra work in AspectJ like languages if stepping was implemented through a loop rather than method calls.
- Around visualisation of agent colour. To augment the display of agents without affecting their internal state.
- Kawasaki chosen neighbour and site. Specialisation from inside steps.

- Tracking Kawasaki exchanges. Requires DSJP creation combining chosen site and neighbour because of use of non-AspectJ supported array indexing.

We can then begin creating DSJP which may populate our semantic model. The chosen DSJP for our two selected models are shown in Tables 4.3 and 4.4.

Observer	"StepStart " | " StepEnd"
Sugarscape	"(No)ChangeWealth" | "(No)Movement" | "(No)SugarChange" | "(No)UnoccupiedChange" | "SugarTotal" | "AntNumber"
Kawasaki	"Exchange" | "ChosenSite" | "ChosenNeighbor"

Table 4.3: DSJP available for AnimauxRI.

Sugarscape	"Step" | "Name" | "Location" | "MaxSugar" | "Wealth" | "Neighbourhood" | "Sugar" | "Unoccupied" | "SugarRemaining" | "AntRemaining"
Kawasaki	"TypeSite" | "TypeNeighbour" | "ExchangeSite" | "ExchangeNeighbor" | "ExchangeNumber" | "StepsExchangeNumber"

Table 4.4: Pointcut comparators available for AnimauxRI.

Xtext DSL creation

Now we have a list of JPs to work from we can begin creating our language with Xtext. We have chosen to use an AspectJ like style for our aspects.

Listing 4.1: Xtext grammar excluding keyword definitions.

```
Model:
   toggle=("ASPECTS_ON"|"ASPECTS_OFF")?
      pointcutadvices+=Pointcutadvice*;

Pointcutadvice:
   pointcut=Pointcut '{' advices+=Advice+ '}';

Pointcut:
   parts+=PointcutPart (operators+=POINTCUT_OPERATOR
      parts+=PointcutPart)*;

PointcutPart:
   part=POINTCUT_PART_TERMINALS ('(' sfield=COMPARABLE_FIELD
      soperator=COMPARABLE_OPERATOR snum=INT ')')?;

Advice:
   typedadvice=PrintAdvice | typedadvice=ControlAdvice
      | typedadvice=ColourAdvice;

PrintAdvice:
   type=("PRINT"|"GRAPH") PrintPart=ADVICE_PRINTABLE;

ControlAdvice:
   control="PAUSE";

ColourAdvice:
   "COLOUR" all="ALL"? colour=COLOUR;
```

We populate our semantic model using the EMF model generated by Xtext, this transforms the implementation-specific parsed model into an in-memory model of our languages behaviour. Our in-memory model is a list of PointcutAdviceSM which contains lists of included PointcutPartSM and AdviceSM. These can then be viewed at runtime, acted on programmatically or operated to generate the matching Java code. From this point the Java generation is merely mechanical firing of the semantic models as shown in Listing 4.2 as all parsing from DSL has been finished by this stage. The translation from EMF model to semantic model transforms the data

gathered from parsing the DSL into problem level information which we can use to populate our DSJP runtime. An example of the changes in making the semantic model is moving the list of pointcut operators from the pointcut level where they are in the Xtext grammar to the pointcut part level in the semantic model as this is how they are stored in our DSJP runtime. A further advantage of this is any changes to the DSLs specification will only require changes in the population of the semantic model once, rather than changes throughout the generation of code. A class model of our semantic model can be found in Figure 4.7.

Listing 4.2: Object-oriented semantic model Java generator.

```
def generateJavaCode() {
   return '''
package dsjpruntime.gen;
import dsjpruntime.*;

public class Population {
   public static void Populate(DSJPRuntime runtime) {
   AdviceEngine adviceEngine = runtime.getAdviceEngine();
   «FOR pa : pointcutAdvices»
      «pa.compile»
   «ENDFOR»
   }
}
'''
}
```

Xtext quick fixes allow for IDE support of the DSL, giving suggestions for user code errors at the domain-specific level. Example code for doing this can be found in Listing 4.3, which is illustrated in Figure 4.13.

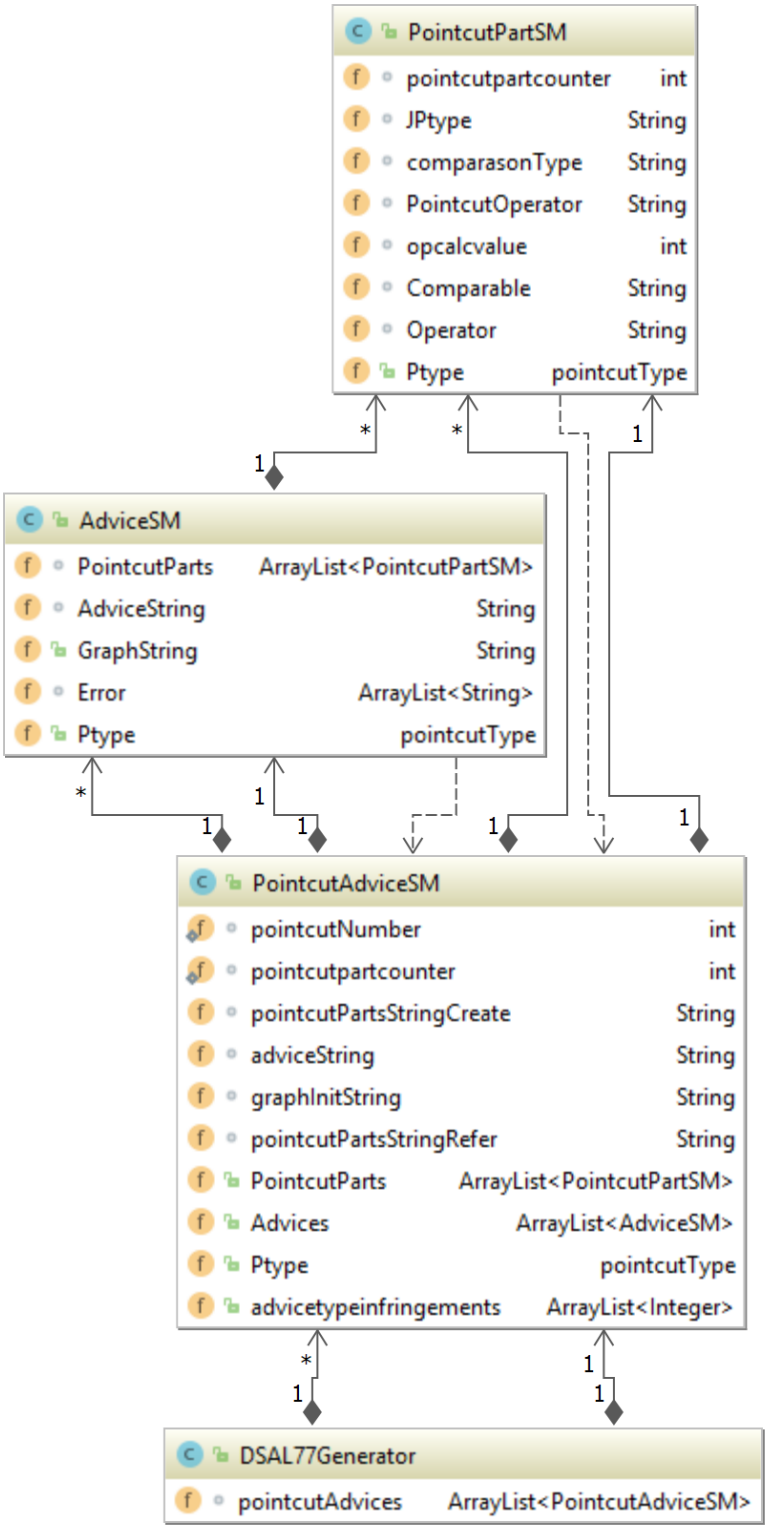

Figure 4.7: AnimauxRI semantic model class dependency diagram.

Listing 4.3: Example snippet of Xtext quick fix implementation.

```
@Check
def checkPointcutAdvice(Pointcutadvice PA) {
   var checker = new PointcutAdviceSM(PA)
   for (var counter = 0; counter < checker.Advices.size ;
      counter++) {
      if (checker.Advices.get(counter).GetError
         .contains(NO_COLOUR_INDEX)) {
         warning("Add all to give colour agent target",
            DSAL77Package.Literals.POINTCUTADVICE__ADVICES,
            counter, NO_COLOUR_INDEX, {counter.toString})
      }
   }
}

@Fix(DSAL77Validator.NO_COLOUR_INDEX)
def noColourTarget(Issue issue, IssueResolutionAcceptor acceptor) {

   acceptor.accept(issue, "No Target Agent, Change to ALL" ,
      "Give Colour to All Agent", "", new ISemanticModification() {

      override apply(EObject element, IModificationContext context)
         throws Exception {
         var PA = element as Pointcutadvice
         var index = Integer::parseInt(issue.data.get(0))
         var colouradvice = PA.advices.get(index).
            typedadvice as ColourAdvice
         colouradvice.all = true
      }
   })
}
```

4.4.2 Bottom Layer: AspectJ and the DSJP runtime

Our bottom layer is a Java framework for interpreting GPJP into DSJP populated through an API and operated by AspectJ GPJPs.

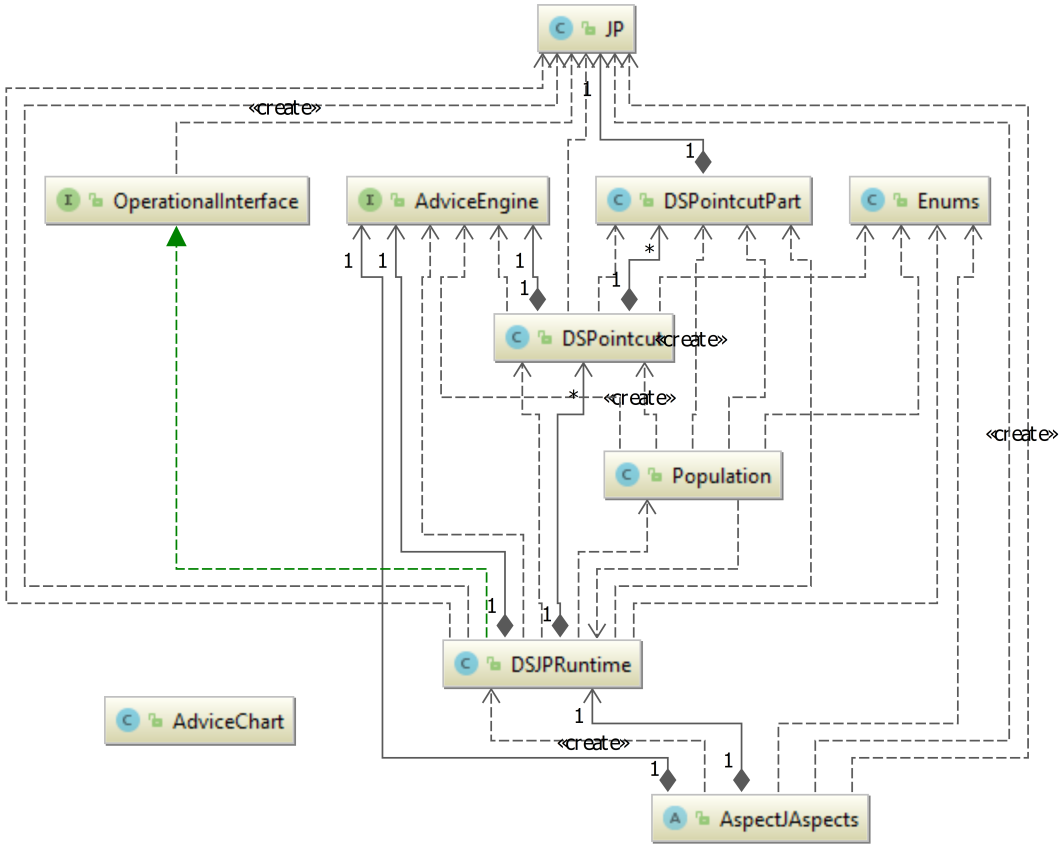

Figure 4.8: AnimauxRI class dependency diagram.

The use of a DSJP interpreter added to the target's project which handles aspects within it is an alternative to transforming the program's source code to incorporate the aspects directly. This approach follows on from the semantic model idea of separating the creation of GPJP (parsing) and forming domain-specific abstractions at runtime. As such, the APIs to populate and operate our DSJP runtime act alike the matching interfaces for a semantic model. Use of this pattern allows us to separate the weaving of GPJPs to AspectJ's mature general weaver and the processing of GPJPs into DSJPs in inherently domain-specific code explicitly tailored for the target models. A class dependency diagram of our implementation of this approach can be found in Figure 4.8 and system flow in Figure 4.9.

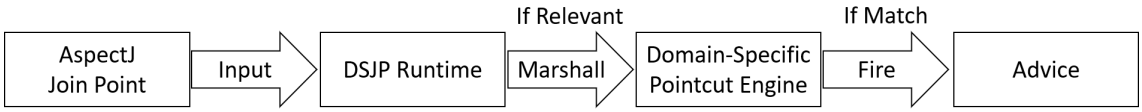

Figure 4.9: The join point flow through AnimauxRI's bottom layer.

AspectJ GPJP

The models dealt with in this project's runtime inspection can be covered by a small set of powerful GPJP locations, which are augmented to match each pointcut and advice's requirements. The AspectJ file is responsible for creating an instance of the DSJP runtime to populate at program start-up and operating this DSJP runtime by inputting GPJPs into it. The set of AspectJ pointcuts used in this project is shown in Listing 4.4. We will discuss how this set of GPJP combined with the context provided through the advice engine can provide our full set of GPJP.

Listing 4.4: AspectJ pointcuts sorted by advice operator.

```
//before and after:
pointcut Step(): execution(
   void Framework.AnimatModel.Evolver.evolve());

//around:
pointcut antColourGet(): execution(
   Color Framework.Ant.getColor());

//after:
pointcut KawasakiSite() : execution(
   void Framework.AnimatModel.KawasakiSite());

//after returning:
pointcut getk1(): execution(
   int Framework.AnimatModel.Neighbourhood.randomSite());

pointcut getk2(): execution(
   int Framework.AnimatModel.Neighbourhood.randomNeighbour(int));
```

DSJP runtime

The DSJP runtime contains the pointcut matching and advice engine for our language. Pointcuts are matched by taking the stream of GPJP and marshalling them to domain-specific pointcut matchers which may be interested in their occurrence. The partial evaluation of JPs by the DSJP runtime is vital to ensuring good performance from an interpreting JP approach. As seen in Listing 4.5, only pointcuts which can match a JP are passed a JP. Furthermore, pointcut advice are toggled at the AspectJ level depending on the models used within pointcuts. For example, if there are no Sugarscape pointcuts, then Sugarscape JPs will be ignored without the excess computational overhead.

Listing 4.5: DSJP runtime marshalling of GPJP.

```
@Override
public void InputGPJP(JP jp) {
   //Direct JP marshalling
   for (dsjpruntime.DSPointcut DSPointcut :
      gpjptypesforuse[jp.getType().index]) {
      DSPointcut.inputGPJP(jp);
   }

   //Step JP also used to create auxiliary DSJPs
   if (jp.getType() == Enums.JPtype.O_StepStart
      || jp.getType() == Enums.JPtype.O_StepEnd) {
      CreateStepBasedDSJP(jp);
   }
}
```

The advice engine works through an interface which has an implementation within the framework of Animaux. Although this could be done through AOP, we have added it as a component of the framework because the behaviours are so tightly coupled to the implementation of the framework and are generally core concerns rather than cross-cutting concerns. Examples of behaviours in the advice engine

are getting properties from the agents, the environment of the model and overall observations upon the model. The advice engine provides means to create many of the DSJP which require further context and provide our advice methods in a single class, rather than scattering and tangling highly coupled calls throughout the project.

We now give examples of how we form our DSJP using the specialise, aggregate and create model.

- **Specialise**

  The specialisation of JPs is the simplest form of creating a DSJP; it is especially useful where a single DSJP may have multiple implementations. For example, step start and step end DSJP are created directly from the accompanying execution of the evolve method. This specialisation moves the implementation of a models step method to a higher level of abstraction for use within pointcuts.

- **Aggregate**

  Aggregation of JPs is useful for tracking common contextual events. We do not provide any direct aggregation DSJP because of our deliberately small selection of base GPJP. Our DSAL allows DSJP to be aggregated by the user using the then keyword.

- **Create**

  Creation of DSJP which do not exist in the GPJP flow of the target program is the most complex form of defining DSJP, and can facilitate some of the hardest to manually code runtime inspection. For example, Kawasaki exchange DSJP are formed by collecting data as the random site and random neighbour GPJP which is then processed to check if an exchange has taken place. This must be done because the exchanges are done through indexed array assignments

which although are direct events within the general-purpose program are not accessible with sufficient context by AspectJ. Specifically, AspectJ cannot give the index of set pointcuts performed on arrays.

Simpler creation DSJP such as ant number, total sugar and space occupied DSJP are created from the base of the step GPJP and context from the advice framework.

4.4.3 Top Layer: Use of the Language

The approach gives the user a very high-level Xtext DSL as a front-end which generates the population of our back-end framework. This is a complete separation of user-written top layer programs and language developer-written bottom layer code. The user can view generated code if they wish by looking at the population.java file found in the gen directory as shown in Figure 4.10.

The proposed use of this DSL is as an axillary measure to ABMs in development in the Animaux framework, specifically the Sugarscape and Kawasaki models. We have chosen to use the Xtext provided Eclipse editor for our aspects, which generates population files directly into our Animaux project's gen folder.

As this environment will require the toggling and modification of aspects throughout development of the project, the DSL allows for toggling of aspects at the start of each file using the optional 'ASPECTS_ON' and 'ASPECTS_OFF' commands which default to aspects on. Java-style comments are supported through the DSL allowing quick modifications and notes about aspects to be stored, this is shown in Figure 4.11.

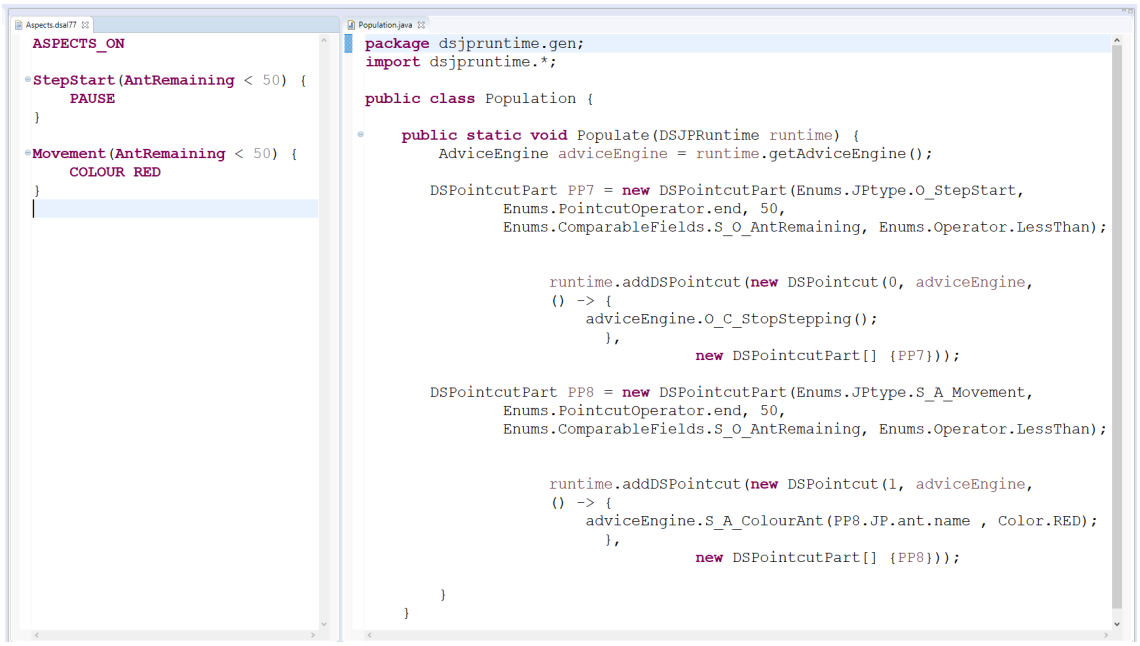

Figure 4.10: Example Sugarscape DSL code in Eclipse editor (left), with generated code (right).

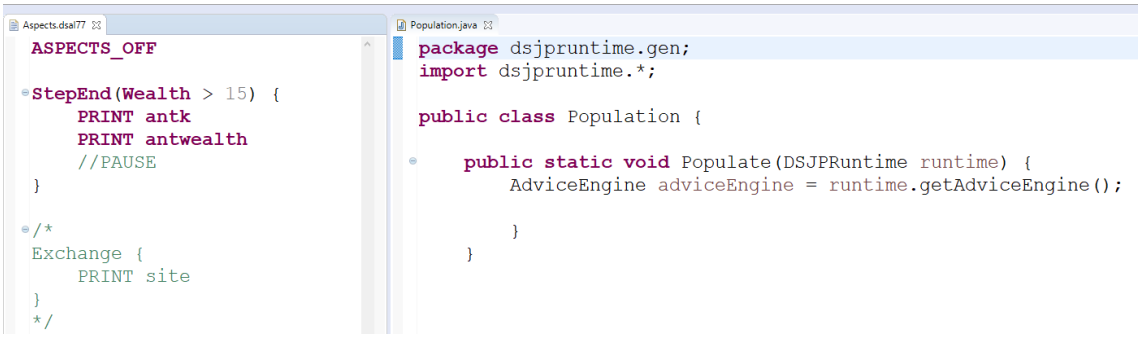

Figure 4.11: Example of toggling and commenting of aspects.

Although the use of a DSAL is at the solution level, language implementation semantics must still be taken into account by the user. For example, pointcut advice are applied in the order in which they are written, meaning the code in Listing 4.6 would not colour any ant orange as the magenta colouring advice will overwrite it.

Listing 4.6: Example of pointcut advice priority.

```
//Applied first
Movement (Wealth > 10) {
   COLOUR ORANGE
}

//Applied second, overwriting the first advice
Movement {
   COLOUR MAGENTA
}
```

Using an Xtext DSL as a middle layer allows the organised application of DSL validation with support for IDE assistance in Eclipse, IntelliJ and a web editor. This means domain-specific checks which are not present in the back-end model can be added to the DSLs validation with feedback and suggestions pushable to a user at the IDE level. An example of this in our application is not being able to pair a Sugarscape and Kawasaki JP together in a pointcut as is shown in Figure 4.12. An example of Xtext's quick fix mechanism providing auto-complete proposals is giving a target to colour advice where the pointcut provides none as shown in Figure 4.13.

This moves supported compile time errors and debugging potential to the user level, although errors which have not been given custom validation rules will arise at the grammar's level of abstraction rather than the target. Problems which arise during runtime will be revealed at the implementation-level of abstraction; These will be handled and logged by the DSJP runtime for inspection by the language developer.

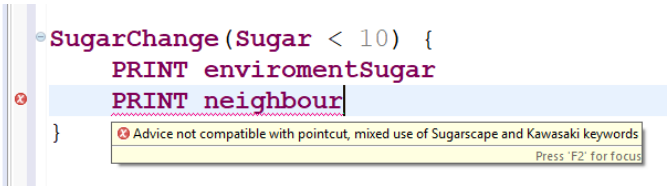

Figure 4.12: Example of Xtext validation of pointcut advice type matching within Eclipse.

Figure 4.13: Example of Xtext quick fix provision within Eclipse giving domain-specific corrections.

4.5 Results

We now will perform experiments with performance and code metrics. We use code metrics on intended use cases, showing the core usefulness of this approach with respect to the language developer and end user. We perform performance metrics to ascertain the practical viability of this method, with discussion of potential alike methods which could give different results.

Experiments are performed on a Windows 10 machine with an i7-4790k @ 4.4ghz with minimum processor frequency at 100% to reduce variability and 16GB of 2133mhz cl9 ram. No performance throttling issues were encountered through testing. These experiments are done through the Animaux GUI, timings are taken directly around the stepping loop, recording directly before the first step has begun to directly after the last step has finished. Timings for these experiments are hard-coded into the Animaux framework using system nano time to avoid any overheads of an aspect

approach misrepresenting data. Each experiment has 20 data points, represented as box plots in our figures. The models used for these experiments are a mountains pattern in Sugarscape and a 50% chance random distribution of particles in Kawasaki.

A large portion of the benefit provided by the DSAL is provided by the framework which the DSAL is based upon rather than the DSAL being a one-step process. The underlying framework can be used without the DSAL giving the same amount of semantic power albeit at a lower level of abstraction. As such, the DSAL can be considered a population layer of abstraction over the framework which implements the semantic model. Through these experiments, we will use the comparison between inline code, DSAL code and generated framework code as a metric.

We use lines of code as a metric to illustrate the implementation size of our framework. Although this metric does not give a full representation of difficulty to write or length of the code in said lines, we believe it is a reliable indication of progress given software practices are consistent and the code is written in good faith. The implementation of the DSAL consists of roughly 1000 lines of code because of the use of object-oriented techniques to keep DSJP runtime code to a minimum. The interfaces and Xtext grammar form roughly 150 lines of code, the DSJP runtime forms roughly 550 lines and the Xtext generator forms roughly 300. Miscellaneous additional code such as validation rules in Xtext are not included in these figures.

Lines of code does not translate well across programming languages, for example, most statements in our generated framework can be considered as one line of code but are significantly more complicated than their DSL counterparts. In our comparisons between DSL and generated code, we use total characters excluding white space as a base mark and logical entities as a metric. A logical entity is a variable use, method call (chains taken as single) or a stand-alone keyword. These metrics are

limited as they merely record length of code rather than the complexity although for our purposes in small snippets of code this is suitable. Character readings are taken ignoring white space and entities ignore punctuation. Statistics on generated code exclude class definition and DSJP runtime access set-up as this is generated regardless of pointcuts.

4.5.1 Aspect Performance Base Benchmark

We first benchmark the performance of the Animaux framework with no active advice using the Java compiler, the AspectJ compiler without AnimauxRI imported and the AspectJ compiler with AnimauxRI imported. This experiment performs 500 steps of each model from a fresh initialisation, results are shown in Figure 4.14 and 4.15.

These benchmarks show low performance overheads of roughly 3% for Sugarscape and around 6% for Kawasaki with the DSJP runtime running with no aspects loaded. The Kawasaki model's higher difference is because there are more AspectJ pointcuts on each step, even though the Kawasaki has a higher base time meaning each JP's overhead is less significant. Because of the higher overhead of the Kawasaki tests, we have included an inline DSJP injection case within further experiments. This inline injection approach allows us to compare what overhead is from the DSJP interpretation and what is from the DSJP creation.

We test this injection case in a worse case performance scenario by activating pointcuts for chosen site, chosen neighbour and exchange. The advice for these steps is incrementing an integer to avoid optimisations removing blank aspects. The experiment results are found in Figure 4.16.

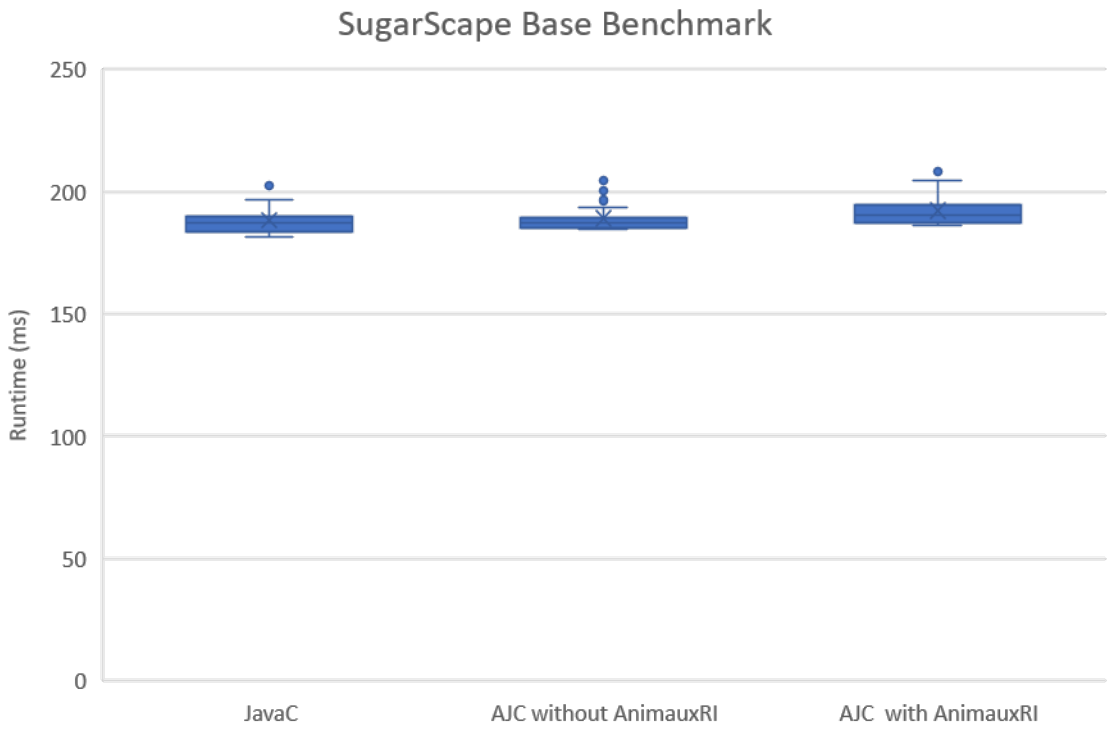

Figure 4.14: Sugarscape base benchmark results.

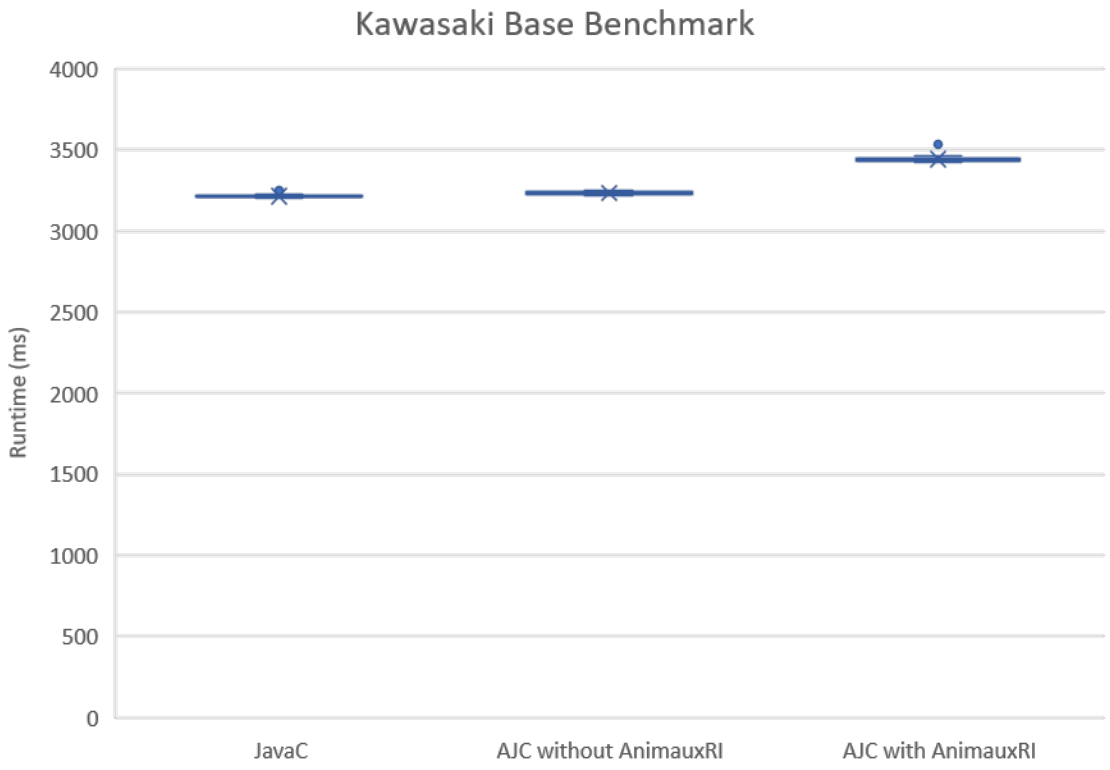

Figure 4.15: Kawasaki base benchmark results.

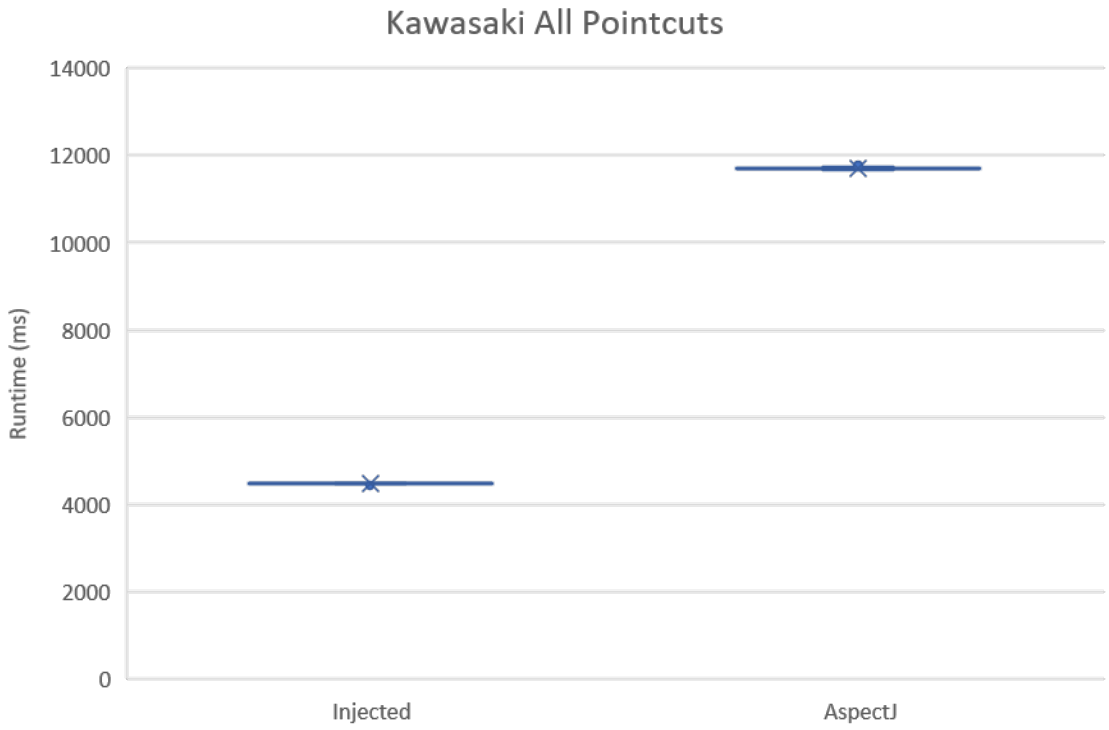

Figure 4.16: Kawasaki all pointcuts benchmark results for injected and AspectJ implementations.

There is a considerable difference between the performance of the injected and AspectJ methods. This is because the AspectJ method has overhead from creating the GPJP and then has to check if it has to create DSJP on these GPJP. The injected approach uses the logic already existing in the model's code to check if a DSJP is needed which requires significantly less overheads. The injected approach does not reduce scatter, although it does mean that tangle is still retained within the DSJP engine except for the single statement of creating the DSJP. This sort of approach may be useful when performance must be improved or for a selection of high impact JPs. This approach could also be applied to the Sugarscape model although most Sugarscape DSJP are specialisations of the AspectJ step JP which has roughly the same performance as injected JPs.

4.5.2 Printing Steps

We now benchmark the performance of common intended use cases of the DSAL. Our first experiment is the simple task of printing the step number upon each step. This is a specialisation pointcut with direct advice, and as such, does not have much overhead. The overhead from the aspect, in this case, is because of matching the step pointcut and accessing the step variable through the advice engine.

The generated code is shown in Listing 4.7 and the inline implementation of the advice is shown in Listing 4.8, performance results are shown in Figures 4.17 and 4.18, and code metrics can be found in Table 4.5.

	Characters	Logical Entities	Scatter	Tangle
Inline	25	2	1 block	Inside model class
DSAL	20	3	1 pointcut advice	Independent aspect
Generated	219	12	1 pointcut advice	Independent aspect

Table 4.5: Code metrics for printing step number on each step.

Listing 4.7: Printing step aspect framework generated code. Located in aspect file.

```
DSPointcutPart PP1 = new DSPointcutPart(Enums.JPtype.O_StepStart,
    Enums.PointcutOperator.end);
runtime.addDSPointcut(new DSPointcut(1, adviceEngine,
   () -> {
      System.out.println(adviceEngine.O_getStep());
   }, new DSPointcutPart[] {PP1}));
```

Listing 4.8: Printing step inline code. Located inside stepping loop.

```
System.out.println(step);
```

This is a simple demonstration of how aspect orientation can remove a task from the model's inline code and into an aspect file with little impact on performance. Kawasaki has more overhead compared to the Sugarscape model because of the higher number of JPs meaning more overhead. This code only removes a single piece of inline code into an aspect, although seemingly small this allows the writing of aspects without knowledge of the implementation of the stepping or logging system

Figure 4.17: Sugarscape print step experiment results.

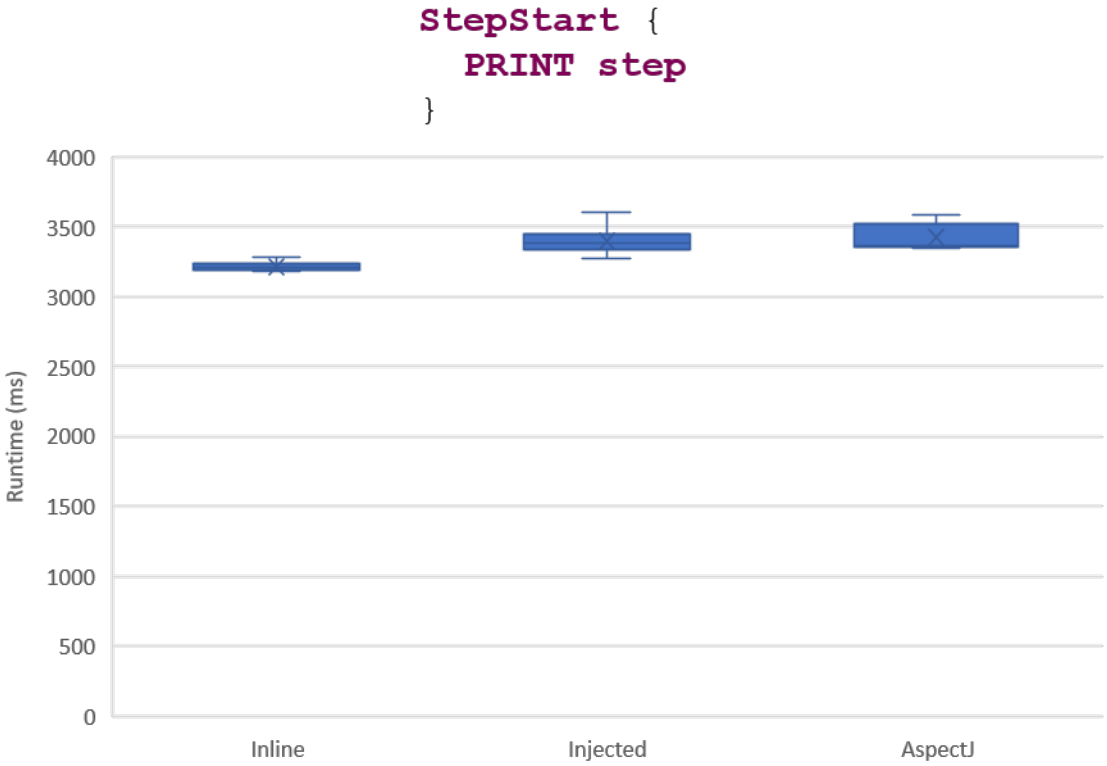

Figure 4.18: Kawasaki print step experiment results.

used by the model. Furthermore because of the declarative potential of DSLs similar simple aspects can be used to perform complex domain-specific tasks with similar code.

4.5.3 Printing Gini Coefficient at Intervals

Following on from the previous experiment, a simple, practical use case of a language like this is the printing of some important model-specific metric at intervals throughout runtime. For this, we have chosen the relatively expensive to calculate Gini coefficient to be printed every 50 steps through execution. The pointcut logic required for this is a simple specialisation of the step JPs with a comparison on the step number, the overhead over normal execution is mainly from the calculation of the Gini coefficient. This experiment's aspect has very little overhead compared to the inline implementation. The inline and aspect implementations can be found in 4.9 and 4.10, Performance metrics are found in Figure 4.19 and code metrics are found in Table 4.6.

This experiment is similar to the previous experiment but has a domain-specific variable available to the user. The context of the DSAL allows for checking at compile time so a Gini coefficient check can only be performed on appropriate models, and the user does not have to think about calculating the Gini coefficient themselves.

	Characters	Logical Entities	Scatter	Tangle
Inline	52	6	1 block	Inside model class
DSAL	38	5	1 pointcut advice	Independent aspect
Generated	285	16	1 pointcut advice	Independent aspect

Table 4.6: Code metrics for printing Gini coefficient at intervals of 50 steps.

Listing 4.9: Printing Gini coefficient at intervals aspect framework generated code. Located in aspect file.

```
DSPointcutPart PP1 = new DSPointcutPart(
   Enums.JPtype.O_StepEnd, Enums.PointcutOperator.end, 50,
      Enums.ComparableFields.O_step, Enums.Operator.Modulo);

runtime.addDSPointcut(new DSPointcut(1, adviceEngine,
   () -> {
      System.out.println(adviceEngine.S_O_getGiniCoefficient());
   }, new DSPointcutPart[] {PP1}));
```

Listing 4.10: Printing Gini coefficient at intervals inline code. Located in stepping loop.

```
if(step % 50 == 0) {
   System.out.println(CalculateGini());
}
```

```
StepEnd (Step % 50) {
  PRINT ginicoefficient
}
```

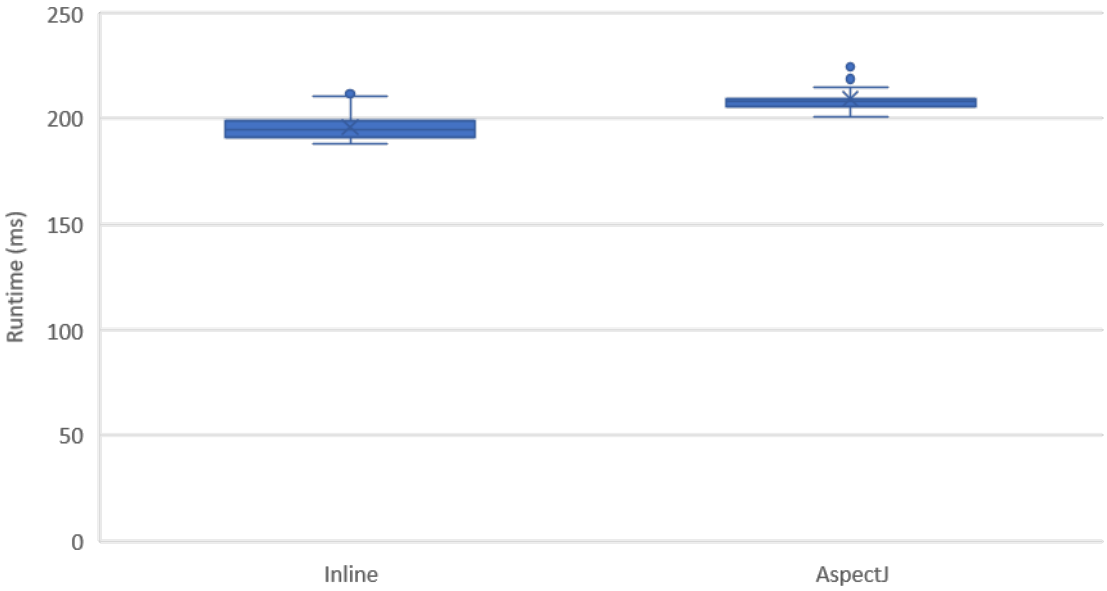

Figure 4.19: Sugarscape printing Gini coefficient experiment results.

4.5.4 Colouring a Selection of Moving Ants

We now move onto a relatively expensive pointcut and advice of colouring of ants conditionally as they move within Sugarscape, the results of this aspect are shown in Figure 4.20. This aspect colours ants which have a wealth of over 10 which is high for this scenario. This allows a more detailed look at wealth distribution than the default red and yellow dichotomy. This is an interesting case because while there is a significant performance hit, the inline implementation requires changes in multiple parts of the Animaux framework due to the design of the ant colouring. This means there is an apparent trade-off between faster scattered and tangled implementation specific code and slower domain-level aspect-oriented code. This trade-off is especially interesting for this type of advice because colouring ants at runtime implies a user watching and possibly feeding back to the system, which will be the performance bottleneck rather than the aspect code. This experiment also requires an additional pointcut advice which resets the colour of ants at the end of each step.

The inline and aspect implementations can be found in Listings 4.11 and 4.12, performance results are shown in Figure 4.21 and code metrics can be found in Table 4.7.

In the base model ant colour is only calculated when they need to be shown to the GUI, rather than being set after every wealth change as if not drawn to the GUI the colour is not used. As such, externally changing the colour value of an ant is not sufficient as it will be overwritten prior to display. The aspect method of dealing with this is using AspectJ around advice on the colour calculation method and changing the return value to an externally set value if a pointcut advice has matched.

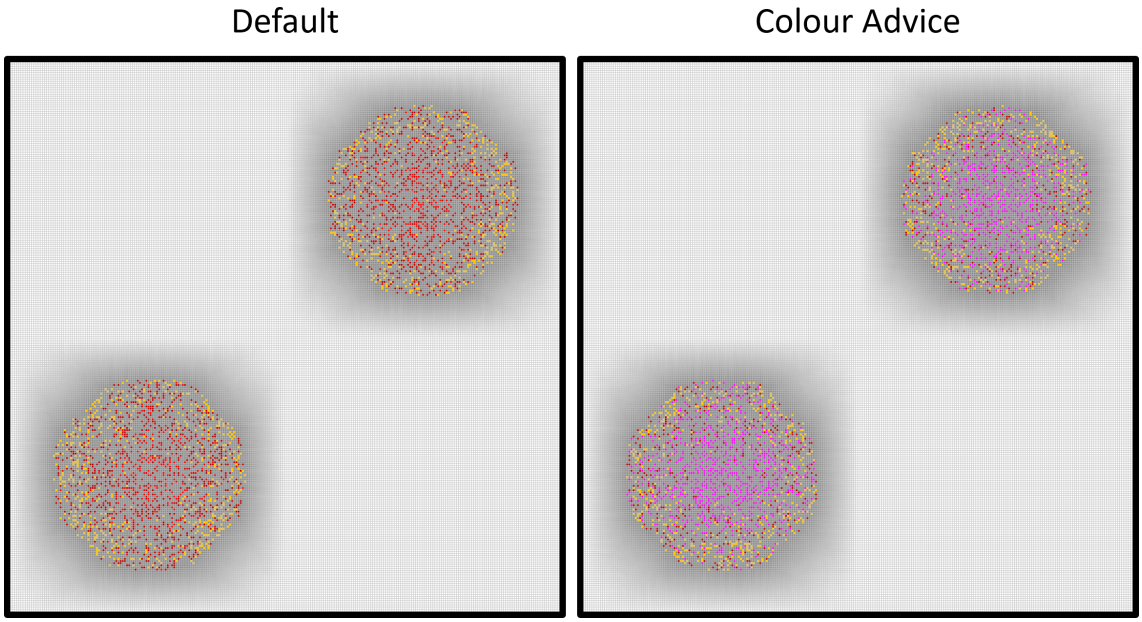

Figure 4.20: Sugarscape colouring wealthy ants aspect side to side with default visualisation at 150 steps into a model.

This method does not alter the runtime of the framework when the pointcut is not matched and changes behaviour by the interception of normal operation rather than the changing of model logic.

Even though the advice for this method is only required when the GUI draws the ants, in the current version of our DSAL there is no static check or partial evaluation aimed to only setting ant colour when it will be visualised; the aspect will change each ant's colour every time it moves rather than only during the 500th step. Partially evaluating this JP's execution by only checking when the ants will be visualised would give great performance bonus, this could be done by only firing colour setting advice upon steps which will be visualised. A more drastic measure of increasing performance would be using a dependency injection approach as we have tried with Kawasaki or a code transformation weaver rather than our runtime interpretation weaver.

The relative performance cost is exacerbated by the short runtime of the Sugarscape models, models with longer base runtime may not consider a few seconds of runtime overhead to be a problem and many projects would happily trade developer time for small increases in compute time. Despite the computational inefficiency of this advice, it provides great separation of the user from the non-trivial implementation semantics of the framework and the domain concept which they want to realise. From a user perspective, the performance hit may be worth it because of the coding and thought time saved each time an aspect like this is written. A similar aspect to this is shown in Listing 4.13, where once the aspect begins firing performance will not matter because any human inspection to test, verify or validate the model will be by the bottle-neck for speed.

	Characters	Logical Entities	Scatter	Tangle
Inline	215	23	4 blocks over 2 classes	Inside ant and model classes
DSAL	57	9	2 continuous pointcut advice	Independent aspect
Generated	505	27	2 continuous pointcut advice	Independent aspect

Table 4.7: Code metrics for colouring a selection of moving ants.

Listing 4.11: Colouring a selection of moving agents aspect framework generated code. Located in aspect file.

```
DSPointcutPart PP1 = new DSPointcutPart(Enums.JPtype.S_A_Movement,
   Enums.PointcutOperator.end, 10, Enums.ComparableFields.
       S_A_wealth, Enums.Operator.GreaterThan);

runtime.addDSPointcut(new DSPointcut(1, adviceEngine, () -> {
      adviceEngine.S_A_ColourAnt(PP1.JP.ant.name , Color.MAGENTA);
   }, new DSPointcutPart[] {PP1}));


DSPointcutPart PP2 = new DSPointcutPart(Enums.JPtype.O_StepStart,
   Enums.PointcutOperator.end);

runtime.addDSPointcut(new DSPointcut(2, adviceEngine, () -> {
      adviceEngine.S_O_ResetAntColour();
   }, new DSPointcutPart[] {PP2}));
```

Listing 4.12: Colouring a selection of moving agents inline code. Located inside the Ant class and Sugarscape logic code.

```
//Inside Ant class
public Color changedColor = null;

Color getColor() { //only called when shown to screen
   if (changedColor == null) {
      checkColor();
   }
   else {
      color = changedColor;
   }
   return color;
}

//Inside Sugarscape logic code
int oldk = k;
...
if (oldk != k) {
   if(a.wealth > 10) {
      a.changedColor = Color.cyan;
   }
   else {
      a.changedColor = null;
   }
}
```

Listing 4.13: Pausing simulation on low ant count and colouring stationary ants DSAL code. Located in aspect file.

```
StepStart(AntRemaining < 50) {
   PAUSE
   COLOUR ALL OFF
}

NoMovement(AntRemaining < 50) {
   COLOUR RED
}
```

```
Movement(Wealth > 10) {
  COLOUR MAGENTA
}

StepStart {
  COLOUR ALL OFF
}
```

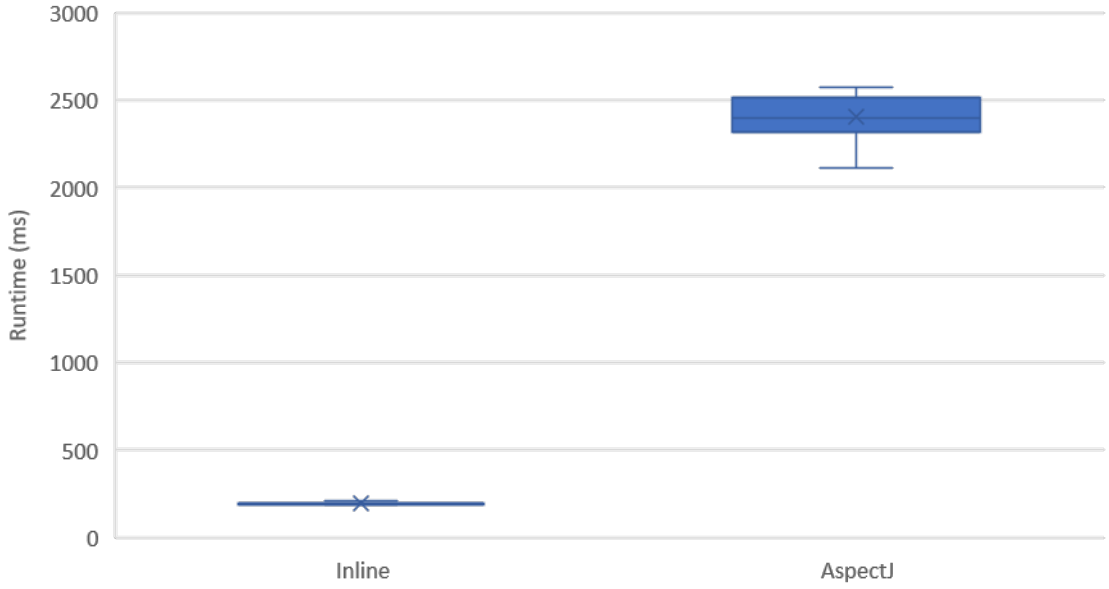

Figure 4.21: Sugarscape colouring wealthy ants experiment results.

4.5.5 Printing Kawasaki Exchanges

Moving onto a Kawasaki specific test of tracking exchanges we can consider the creation of DSJP which do not exist as AspectJ GPJP. This experiment prints the site and neighbour of each exchange which occurs on the first 2 rows of a 256x256 model. Animaux's exchange mechanic relies on assigning into arrays of integers, yet AspectJ's set pointcut does not expose the index of array assignments. Without the use of third-party tools such as ABC from Allan et al. (2005), extra steps must be taken in the capture of GPJP to create this exchange DSJP using AspectJ. As such, our implementation is an expensive operation which is set up using the chosen site and chosen neighbour JP specialisations to create shadow values of a site's spins which are then compared after the site has finished executing. The inline and aspect implementations can be found in 4.14 and 4.15, performance metrics of this can be found in Figure 4.22 and code metrics in Table 4.8.

This implementation is similar to our colouring a selection of moving ants Sugarscape experiment, although we have our injection-based approach to compare times with. The scatter and tangle improvement from this experiment is low because the exchange is located in a single place. Although as we stated in the printing steps and Gini coefficient examples, hiding the implementation from the user using transparent aspect orientation gives benefit for showing the code to external domain experts and maintaining code without full implementation semantic understanding.

The injection approach gives a significant performance increase over the AspectJ approach, again because the creation of GPJP and the matching to potential DSJP is not required in the injected approach. Much like the last model, the project's requirements for performance will define if the extra scattering through the project will define if this is worthwhile.

The performance results of this experiment are roughly the same as the all pointcuts experiment from Figure 4.16 because the exchange JP requires the site and neighbour JPs to be created.

	Characters	Logical Entities	Scatter	Tangle
Inline	79	9	1 block	Inside model class
DSAL	43	7	1 pointcut advice	Independent aspect
Generated	323	19	1 pointcut advice	Independent aspect

Table 4.8: Code metrics for printing Kawasaki exchanges.

Listing 4.14: Printing Kawasaki exchanges aspect framework generated code. Located in aspect file.

```
DSPointcutPart PP1 = new DSPointcutPart(Enums.JPtype.K_O_Exchange,
   Enums.PointcutOperator.end, 512, Enums.ComparableFields.
   K_O_ExchangeSite, Enums.Operator.LessThan);

runtime.addDSPointcut(new DSPointcut(1, adviceEngine,
   () -> {
      System.out.println("Site : " + PP1.JP.k);
      System.out.println("Neighbour : " + PP1.JP.k2);
   }, new DSPointcutPart[] {PP1}));
```

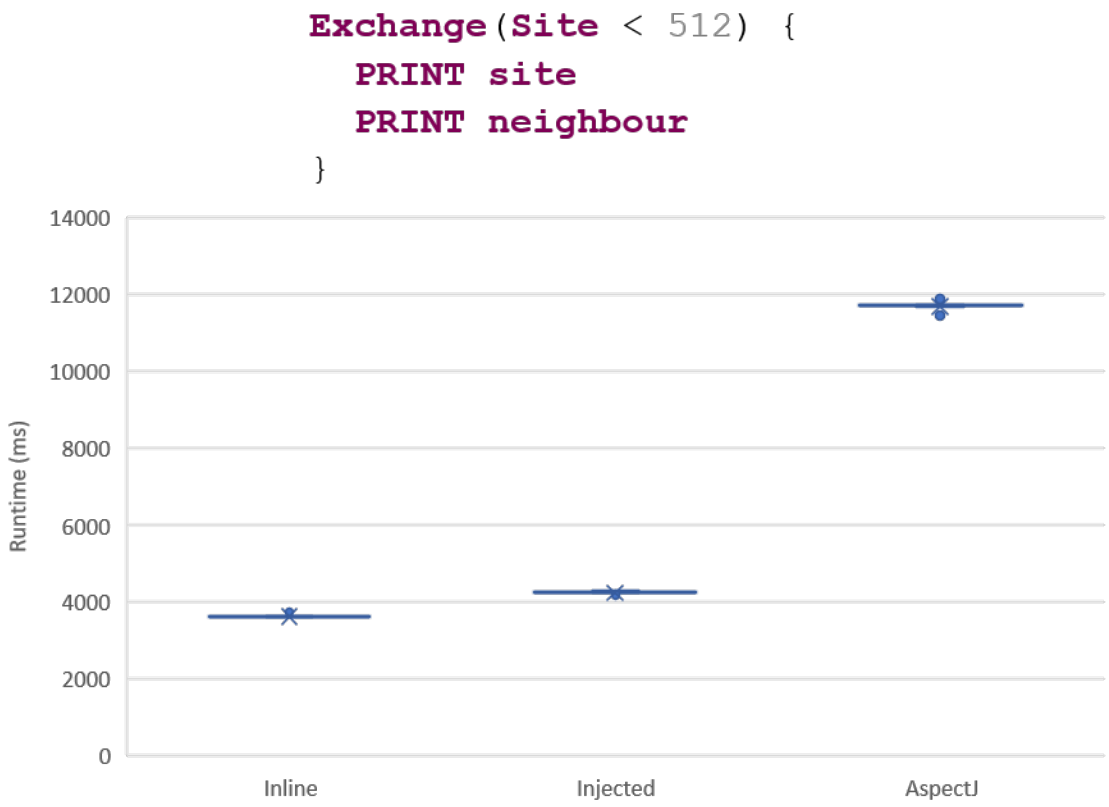

Figure 4.22: Kawasaki printing a selection of exchanges experiment results.

Listing 4.15: Printing Kawasaki exchanges inline code. Located inside Kawasaki logic code.

```
if(k1 < 512) {
   System.out.println("Site : " + k1);
   System.out.println("Neighbour : " + k2);
}
```

4.6 Discussion

Through this chapter, we have motivated, analysed, designed, implemented and examined an approach to providing separated domain-level runtime inspection on a set of models running on the Animaux framework. This work will be forwarded in Chapter 5 with work on the JADE framework using an internal DSAL. We now discuss the process, results and implications covered in this chapter.

4.6.1 on Aspect-Oriented Runtime Inspection

The language we have designed using our approach is essentially similar to the domain-specific pattern-action languages used for text processing in Unix systems. Extending the Unix philosophy of writing programs which do one thing well and can be chained with other programs to perform large tasks (Kernighan and Pike, 1983). The language captures the essence of commonly asked questions about a model as it runs and allows the user in simple terms to state when this happens, do this. The advent of AOP had made this type of language more powerful because it is used to augment patterns in computational systems rather than being a stand-alone program for offline textual data. The selective runtime inspection of models allows for models which would take massive amounts of storage space to be observed as a whole to be observed in specific parts which the scientist is interested in.

The code which is written and modified the most times gives the potential for the most significant savings of development time. Runtime inspection is one of these tasks where similar queries will be written for many similar models across many similar observation experiments. AOP allows the separation of runtime inspection

code from the core concern of simulating a model, allowing this code to be written, modified and applied as a stand-alone entity rather than changing multiple pieces of code throughout a solution to achieve the same effect.

The use of aspect-oriented techniques does create an overhead in terms of an extra build step and another level of system geometry for the programmer to keep track of. The extra build step of compiling DSAL code and AspectJ aspects is hidden from the user by modern IDEs, and as such, is not a pressing concern. However, the introduction of a further level of system geometry in the form of aspects does influence the number of things a programmer must hold in their heads when considering changes to program code. This introduces a trade-off of if the added complexity of code being weaved into the system is worthwhile because of the reduction of complexity in expressing the concerns in an aspect-oriented manner. For the domain of runtime inspection where simple runtime inspection may be adequately performable using simple logging with boolean toggles scattered and tangled throughout the code, although this becomes more difficult as more models are introduced to a framework which the code may interfere with or more complex inspection tasks are required. We believe as the domain of runtime inspection is inherently separated from the core concerns of the model execution and is so commonly used, modified and toggled that the use of aspect-oriented techniques is warranted for all but the most straightforward cases.

The language implementation used in this chapter generates code from a static set of aspects at compile time. There is significant scope for live programming while a model is running, with feedback for which aspects have fired and what effect they have caused. For example, having multiple colour aspects on a single model it would be useful to know which aspect has coloured a specific ant and why. This live approach is suited towards our dynamic interpretation framework far more than code

generation approaches which are inherently static. If the decision to use a runtime interpreter of the DSJP remains, then it may be profitable to consider the use of a live editor during runtime rather than our single populate at start-up design decision.

4.6.2 on External Middle-Out DSALs

A separate DSL to the underlying framework means the user-facing code can be organised in a way that the user would like rather than how the implementation is done. In this DSAL we have chosen to use the AspectJ style of pointcut advice, although we could have also investigated declarative styles following how a domain expert may request an behaviour. For example, saying colour the moving ants rather than saying when an ant moves colour it. This change could be done to the Xtext DSL without affecting the underlying framework; opposite approaches could even be used in the same project with two different DSLs for the same semantic model.

The middle-out used in this approach may best be described as a mix between bottom-up and middle-out development because of a DSALs heavy reliance upon plausibly available GPJP. When developing a middle-out DSL previous experience of what bottom-up utilities can be provided using a base language is consistent and probably re-usable across DSLs. When developing a middle-out DSAL, the target code dictates the plausible JPs to a great extent, which only then can be used to create an implementable semantic model for the middle layer. This means that to some extent the possible utilities providing the features for the DSAL are set by the target framework or programs rather than the language designer. We retain use of the language-oriented middle-out approach rather than coining a new term because, despite the increased reliance on the plausibility of the bottom layers DSJP, the focus of working from a DSL first is still the primary concern.

If a framework target has not yet been developed and the framework can be designed to suit the required JPs for a language, this allows for increased autonomy during middle layer scoping. This approach to framework creation also allows planned use of injection-based implementations for problematic JPs rather than traditional aspect-oriented implementations. Using dependency injection creates a higher scattering of code in the framework implementation, but at the top layer the scattering and tangling will remain the same as if using traditional AOP. This is only possible because the scattering and tangling of DSJP instrumentation within an application will be orders of magnitude less than the scattering and tangling of GPJP instrumentation.

This approach's layered structure provides excellent separation of concerns of the language use, development and aspect implementation which is generally desired when solutions work. This separation is provided by the distribution of specialised work throughout the layers which may be performed by one or more people in different roles. The disadvantage of this approach is dependencies between layers may cause work on one layer to be halted while a solution is found. For example, if the DSL is generating code which does not match the expected semantics, the user must first notice the problem is the generated code rather than the DSL code and then modify the middle layer's generator to fix this. This is especially infuriating if the problem in the generated code is a simple issue such as a missing parenthesis, yet it cannot be fixed manually because it is overwritten with every change to DSAL code.

The use of an external DSL comes with the disadvantage of having to provide the axillary support now expected by programmers. This task is significantly reduced through the use of language workbenches, which make external DSLs a more economical and mature option. Without the use of language workbenches making

an external DSL is either a direct textual parsing task which does not provide IDE support or a combined task of creating a language and toolchain. These tools have steep learning curves although a majority of the initial curve overlaps with the use of BNF based parser generators such as Flex, Bison and ANTLR. Once this learning curve has been passed full featured DSLs can be designed, implemented and deployed quickly making language-oriented ideas more feasible in everyday projects.

4.6.3 on Domain-Specific Aspect-Oriented Development Tools

This chapter's contribution is predominantly a display of DSL techniques adapted for DSALs allowing language creators to use the mature language creation and AOP weaving tools to their advantage while retaining semantics throughout. Although we have used the industry standard frameworks AspectJ and Xtext for our Animaux implementation, the approach is not tied to these pieces of software.

A problem with the use of GPJP as a base for a DSAL is the reliance on the underlying general-purpose aspect frameworks JP model which will generally be incomplete. Examples of this problem emerging through this project is the lack of support for providing array index on set JPs and lack of support for loop invocations as JPs. Within this project, as we have control of the underlying framework, and have only a limited domain to deal with these did not cause critical issues. In larger DSALs especially those targeted towards proprietary frameworks or hardware such as general-purpose graphics processing unit programming, the underlying GPJP model will almost certainly constrain a projects potential.

The use of an interpreting runtime rather than code manipulation was taken for this project because it considerably reduces the development time of a DSAL. An interpreting framework which accepts GPJP and marshals them to appropriate DSJP matchers is based on a homogeneous client-server idea where all JPs are treated the same. Further work could include the addition of code transformation for JPs which are inefficient to match using the interpreter model. This could be because they are especially simple specialisations which are not worth the overhead of the full runtime or are specialised towards small sections of runtime such as only specific agents may match within a specific position and may be partially evaluated at compile time.

Chapter 5

JADERI - Runtime Inspection Targeting Middleware

This chapter follows on from our work in Chapter 4, moving towards a middleware targetting core Domain-Specific Aspect Language (DSAL) implementation, and accompanying extension DSALs focused targetting specific models without needing to re-implement weaving logic. This is a generic approach which requires more initial investment for the core, although has more potential for reuse across many projects with a small investment for extension towards a specific model. The target audience for a DSAL such as this is researchers who use existing open technology to produce prototypes, proof of concepts and working systems to be experimented on for short-term projects. We first motivate and present our approach with the associated Foundation for Intelligent Physical Agents (FIPA) background, followed by details of our implementations, closing with experiments, results and discussion. Our implementations are extensions of the JADE framework (Bellifemine et al., 2003) written using a combination of Java and AspectJ.

The primary research question answered by this chapter is:

How would we implement a middle-out DSAL for a generic cross-cutting concern within a framework using domain-oriented interfaces, which could then be specialised to specific models without re-implementing weave logic?

5.1 Motivation

The motivation for this work is allowing researchers who use the same frameworks to make many Agent-Based Models (ABMs) throughout different projects to have tooling and processes which allow them to create and use DSALs for the runtime inspection of agent communication models. This kind of tool allows the implementation of runtime inspection features at the level of abstraction of the agent communication rather than the implementation of this communication. Examples of runtime inspection tasks could be gathering statistics throughout runtime, creating domain laws providing an exception-like tool for monitoring behaviour which should never or always should happen at runtime which may indicate problems in the conceptual model rather than the implementation of this conceptual model. Our technical solution is towards creating an aspect-oriented core of Join Points (JPs) based upon filters on middleware operations which happen across many models and using internal Domain-Specific Languages (DSLs) as a means of packaging domain knowledge into more specific DSALs based upon this core. These choices are in contrast to what was done in Chapter 4, where JPs are targeted towards individual models and an external DSL is used to capture domain knowledge. Although we focus specifically on internal DSLs for this chapter; external DSLs are also applicable for this approach as we discuss later.

The layered approach to dealing with abstraction follows the notion that separating application specific and domain-specific code allows for better reuse as the domain-specific code may be a base for many application-specific languages without modification (Poulin, 1995). A concrete example of this is our internal Python DSL for family tree data (Maddra and Hawick, 2016) which used the DOT graph DSL to produce output, removing the need to re-create a graph drawing library as discussed in Chapter 2.

The idea of a core DSAL is illustrated in Figure 5.2. Rather than creating a new DSAL for each model, the core may be used directly, wrapped with utility classes for a certain field of problems or packaged into a very specific internal language for a specific model. This means the original boilerplate work of weaving into a framework is done once, and further domain-specific specialisation can be done if and when required.

The middle-out uses of this core can be seen in Figure 5.1. This is a stronger form of middle-out development than is seen in Chapter 4 because the middle layer is explicitly formed as a core DSAL for language developers, which can then be used to form new languages for subdomains as required. This is possible when targeting middleware because the weave target is standardised ahead of time. In Chapter 4 the focus on many specific models which can be implemented in many ways meant we had to focus more on the bottom layer as we defined a middle layer because of the inherent dependency on it.

The middleware supported cross-cutting concern we have aimed our language at is agent communication. The communication between agents is a vital part of agent-based systems as it allows the cooperation, collaboration and negotiation which can result in emergent behaviours. We use the JADE framework as the middleware

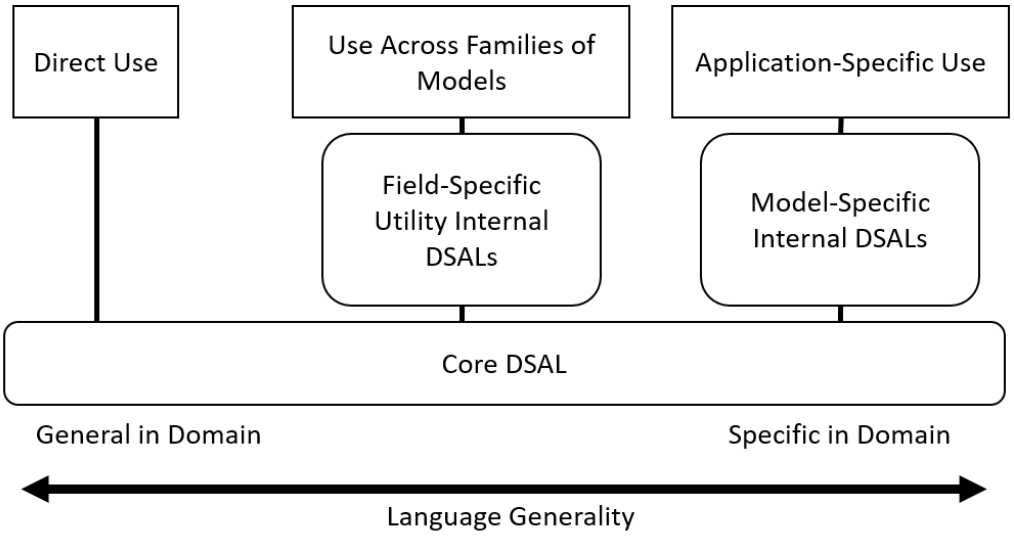

Figure 5.1: Concept of an extensible core DSAL with internal DSALs for specific problems.

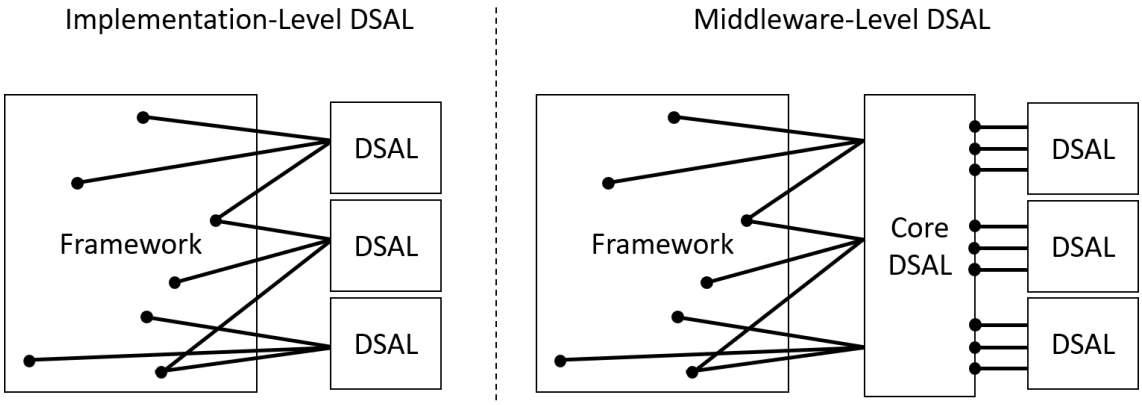

Figure 5.2: Comparison of responsibility when weaving with and without the use of a core DSAL at the middleware level.

base. JADE is a software framework to develop distributed agent-based applications in compliance with the FIPA specifications for interoperable intelligent multi-agent systems (Greenwood, 2004; Bellifemine et al., 2001). Our work adds to the existing support tools provided with JADE such as the remote management agent, dummy agent, introspector agent, log manager agent, directory facilitator agent and sniffer agent (Bellifemine, 2004; Bellifemine et al., 2007).

We have chosen to use JADE because of the framework's affiliation with the FIPA standard giving a strong base to build our JP model from. The specifications released by FIPA are basic building block technologies which can be integrated into many fields to produce complex systems with a high degree of interoperability. This makes them an excellent fit for targeting as a core implementation of a JP model. This standardised base also makes it far more likely that our DSAL will be compatible with JADE models in many versions of JADE without any changes or preparation by the model programmer during development. JADE is targeted towards multi-agent systems rather than ABM but is suitable for both, generally speaking, multi-agent systems focus on smaller numbers of more intelligent agents than ABM. At the inter-agent communication level, a multi-agent system and interaction-oriented ABM are similar enough for our method to be used for either approach. The specialisation towards multi-agent systems provides needed tools for ABM and allows for better use in realising simulation models in hybrid-physical demonstrations.

The concept of using JADE-like multi-agent system frameworks for agent-based simulation rather than multi-agent systems has been a topic of debate within the software engineering community because ABM frameworks generally do not follow the agent-oriented middleware approach as many multi-agent systems frameworks

do. Cardoso (2015) is moving towards using JADE design principles to design ABM which can be generated into Repast models, with future work of allowing JADE generation as well.

Our target audience and JADE goes hand in hand with our choice to focus on internal DSALs through this chapter. JADE is an advanced agent framework which allows for large complex system by skilled programmers. Using an internal DSAL gives the user higher flexibility and responsibility in being able to use Java features within pointcuts and advice. An external approach relies more heavily on the developer of the external DSAL to fit everything needed in the DSAL in each release as the user is locked into using the DSL for all features.

There have been extensions to the JADE framework in the literature. JADEL from Bergenti et al. (2017) is an Xtext based DSL for simplifying the creation of JADE agents, behaviours and ontologies using an agent-oriented approach. JADEx from Braubach et al. (2005) is a BDI-extension of JADE which supports the construction of agents using XML-based agent descriptions and procedural plans in Java.

5.2 The Foundation for Intelligent Physical Agents Standards

FIPA is an IEEE Computer Society standards organisation with the goal of promoting the success of emerging agent-based applications, services and equipment by providing internationally agreed specifications for interoperability between solutions. Through this chapter, we focus on the FIPA implementation within the JADE framework. There are other many systems which use FIPA standards such as: Java-based Intelligent Agent Componentware (JIAC) (Hirsch et al., 2009), Zeus Agent Toolkit

(Nwana et al., 1999) and FIPA-OS Agent Platform (Poslad et al., 2000). There are also agent frameworks such as JACK intelligent agents who are not tied to any specific agent communication languages although reference FIPA standards as a possible choice in their publications (Howden et al., 2001). As the adherence to FIPA standards should allow interoperation between different frameworks, there has been work done on this between JIAC and JADE by Soklabi et al. (2013) although through this chapter we focus solely on JADE. We have chosen JADE because of its maturity and the aforementioned wide use within research, especially in the DSL area. Thus far FIPA has no official procedure to test compliance to standards. JADE gains it claim to compliance from successfully participating in both the Seoul 1999 and London 2001 FIPA interoperability tests and through an active participation of the JADE team as a member on the FIPA architecture board (Greenwood, 2004; Bellifemine et al., 2003).

The FIPA agent platform is illustrated in Figure 5.3. The main components are a Directory Facilitator (DF), Agent Management Service (AMS) and Message Transport service (MTS). The DF is an optional component responsible for providing yellow pages services to other agents. It maintains a list of agents which is available to all agents with authorisation to view it. Many DFs may be active on a single agent platform and may register with each other to form federations.

The AMS is responsible for monitoring and controlling the agent platform and all of its remote containers, this includes the remote management of agent life-cycles and sending of custom messages from external sources. The JADE white pages service is hosted by the AMS, and all agents must register with the AMS to get a valid Agent IDentifier (AID). The AMS also launches the graphical interfaces of auxiliary agents such as the dummy agent or sniffer agent.

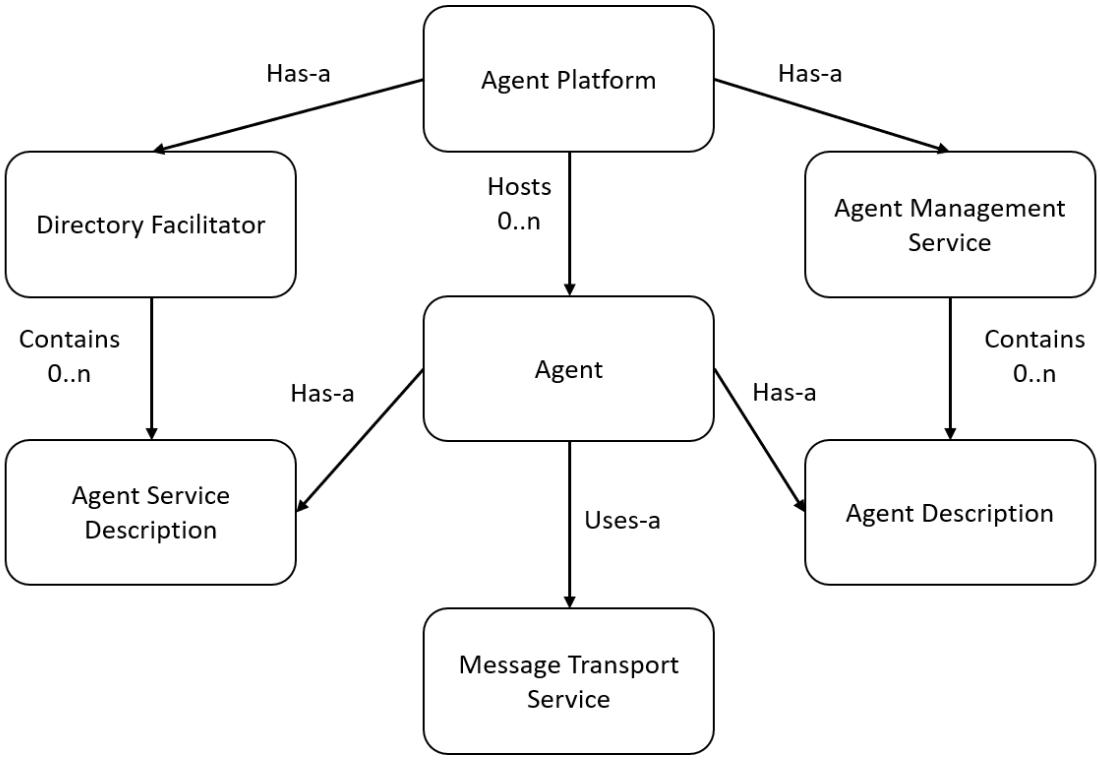

Figure 5.3: Depiction of the agent management ontology adapted from Greenwood (2004).

The MTS is a service responsible for the delivering of FIPA Agent Communication Language (ACL) messages across agents on the same or different agent platforms using an appropriate method of transportation such as HTTP, RMI-IIOP or SMTP.

Through this chapter, we focus on JADE ACL messages. ACL messages are modelled after speech acts within speech act theory. A speaker utters speech acts, which are known as performatives or communication acts. These speech acts may be used to refer to the agent's intention in the world. This display of agent intention is a large part of why FIPA ACL discussions are worthy of runtime inspection. The set of FIPA communicative acts are shown in Table 5.1 and the set of FIPA protocols are shown in Table 5.2. These messages also contain content which may be in any language, this could be a FIPA SL content language; a standard programming language such as PROLOG, HTML, SQL or a bespoke language created for a project.

Communivative Act (CA)	Base CA	Assertive	Commisive	Directive	Mediate	Phatic	Query
accept-proposal	inform			x			
agree	inform		x				
cancel	disconfirm					x	
cfp	query-ref			x			
confirm	confirm	x					
disconfirm	disconfirm	x					
failure	inform					x	
inform	inform	x					
inform-if	inform	x					
inform-ref	inform	x					
not-understood	inform					x	
propagate	inform				x		
propose	inform			x			
proxy	inform				x		
query-if	request						x
query-ref	request						x
refuse	disconfirm; inform					x	
reject-proposal	inform			x			
request	request			x			
request-when	inform			x			
request-whenever	inform			x			
subscribe	Request - whenever						x

Table 5.1: Types of FIPA Communicative Acts (CAs) from Poslad (2007).

Interaction Protocol (IP)	Task Info-sharing	Push / Pull	1-1 / 1-m Receivers	Other Features
Request	Task	Pull	1-1	Cancelable (by initiator)
Request-when(ever)	Task	Push	1-1	Cancelable
Query	Info	Pull	1-1	Cancelable
Contract-Net (CN) / Iterated CN	Task	Push	1-m	Cancelable, iterated version is a multi-round IP
English / Dutch Auction	Info	Pull	1-m	Cancelable
Broker	Info	Pull	1-m	Cancelable
Recruit	Task	Pull	1-1	Cancelable
Subscribe	Info	Push	1-1	Not cancelable
Propose	Task	Pull	1-1	Not cancelable

Table 5.2: FIPA interaction protocols from Poslad (2007).

5.3 Analysis, Design and Approach

This section builds upon the analysis found in Chapter 4, specifically in regard to the issues of creating model-specific DSALs. The main points of discussion are how can we benefit from creating a middle layer based upon a strong middleware standard, what approach should be used to realise implementations of this, and how will this present itself in real-world projects.

5.3.1 Middle-Out DSALs for Middleware

Given the difficulty of implementing DSALs compared to DSLs (especially internal DSLs), it would be favourable to move the implementation of DSALs into a process closer to that of creating a DSL. When implementing a DSAL at model

implementation-level, the base code will be different and require different JPs to express each model. This problem of runtime inspecting on a per model basis was seen in Chapter 4 where Sugarscape and Kawasaki models use different data structures for model operation, resulting in requiring different General-Purpose Join Points (GPJPs) to create Domain-Specific Join Points (DSJPs). Although the resulting DSAL allows for programming at the model abstraction level, the creation of the DSAL needs to use GPJPs highly coupled to either a single model or a program family of models. An example of a program family models is the Animaux framework's Kawasaki models with differing evolution dynamics of gravity, erosion and algae. A further disadvantage to implementing based upon non-standardised code is the use of difficult to quantify dynamics within models, namely use of arrays for storing values and looping around calls as AspectJ does not support these directly. This means that a model may require the inefficient creation of JPs or require refactoring of the model to allow feasible aspect-oriented runtime inspection.

we can use a middle-out approach with middleware to implement a single implementation-level core DSAL which can then be extended using DSL techniques at the middleware level of abstraction to create model-specific DSALs. This allows for many models to use the same set of JPs and use appropriate arguments upon these JPs to form pointcuts upon this set of JPs. Instead of weaving into a model's implementation, we weave onto a middleware which many models will use. This is suited to fields of runtime inspection where middleware is commonly used although is less suited to things which are ad-hoc or internal to an object as these will not have standardised domain-oriented interfaces.

The middle-out implementation of such a core is similar to the middle-out of Chapter 4 although with differences for the bottom and top layers. The middle and bottom layer are mediated by the static middleware standard meaning different

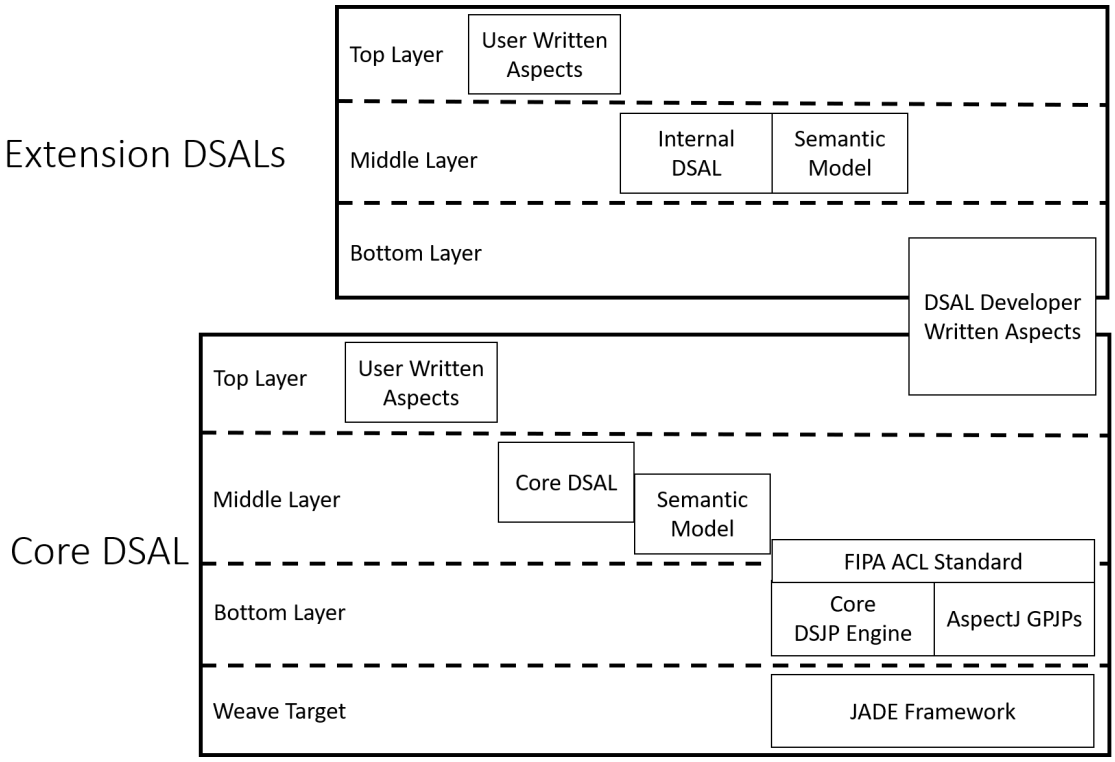

Figure 5.4: Logical layering for extendible middle-out development targeting middleware.

bottom layer targets will be applicable. The top layer may be used as the bottom layer implementation of new internal DSALs created for more specific domains. This is shown in Figure 5.4

5.3.2 Middle-Out DSALs for Agent Communication

One of the key concepts of agent-based systems is communication between agents. It may be the basis which agents cooperate, collaborate and negotiate with one another, and can give direct signs of the agent's beliefs, desires and intentions within a system. This is especially true of standards such as the FIPA ACL where messages are given performatives, protocols and languages which directly denote the behaviour of an

agent. Although the agent's behaviour is out of the scope of FIPA specifications, we can use the communication of agents to track behaviours and trends within the system.

Given the FIPA specification, we can create pointcuts on certain behaviours and weave advice for inspection, analytics or logging at these points. This has the advantage over directly weaving into the behaviours because the communications specification will be followed for all FIPA compliant agents, meaning the core DSAL will not need to be re-compiled or modified to add these extra DSJP.

This approach could also be implemented to work across a set of frameworks which implement FIPA ACL messages. The mapping between different frameworks could be dealt with internally, making for a framework agnostic core and extension DSALs. To implement this in our implementation this would require a wrapper class over the input parameter for advice for JADE ACL messages and the equivalent from the other frameworks. This does not account for framework-specific code written within custom filters and advice added through Java code. Using Java as an advice language couples the advice to the elements it calls and parameters passed to the advice, this makes cross-framework use problematic. To solve this issue a framework agnostic ACL message class could be used as a parameter for advice rather than directly using the JADE ACL message class, this will incur a small performance hit in deep copying the results of each message at pointcuts. The weaving into other models would merely require the addition of AspectJ pointcuts for that model with the appropriate GPJP marshalling. Through this chapter, we focus on a JADE only implementation.

5.3.3 Filter Based Core Implementation

The core of the approach should be based directly upon the FIPA standard using interfaces to allow for strong type checking through the core implementation. This is important to formalise the capabilities of the core, although this is not an external language it is as important to specify properly.

We have chosen a subset of the FIPA ACL which we deem to be useful for our implementation although this may be extended using custom filter and advice methods within the implementation. We have maintained these choices to maintain the simplicity of illustrating the approach. The core implementation has three primary considerations: completeness, direct usability and extendibility.

Completeness of domain coverage is a consideration for all framework and language development. A language which is not complete for a domain either does not have a range of tools wide enough to deal with the intended tasks or misses tools which are required in certain situations. For internal DSLs this is less of an issue because of the access to host language features around the DSL, for example, looping of DSL constructs will be provided by the host. In an external DSL, a domain-incomplete language could stop useful tasks from being possible. Reducing the scope of a language can also be an intended feature, for example, preventing effect on underlying agents. The choice to use an internal DSL also allows for access to the core directly through Java code without modification which may not be possible if the core is built into the back-end of an external DSL.

The direct use of the core DSAL is useful for low use cases which do not warrant a pre-emptive language to call them and ad hoc work adding to already created extension DSALs in fringe cases. The direct use of the core should be the same as writing the equivalent calls in an internal DSL in implementation, as both ways are

populating the same semantic model. The usability of the core is important because it will be the base which internal DSALs can be based off; a poor core design will lead to difficulty in creating extension DSALs.

The extensibility of the core is the fundamental part of this approach working in a middle-out fashion. It relies on the previous two considerations along with the ethos of the team working on the projects.

5.3.4 Model-Specific Internal DSLs

The core language gives appropriate expressive power for the target cross-cutting concern although does not capture the boilerplate of writing runtime inspection code for specific models. We can give this expressive power by creating DSAL layers over our initial core; these could be internal or external DSLs. We focus on internal DSLs because we believe it is more appropriate for the model's programmers remain in the Java projects environment, removing the need for incorporating new tool kits into development. If we were to use external DSLs then extra tooling and skill sets for language development would be required.

The internal DSALs can be implemented as another layer on of the specialise, aggregate and create model we have used throughout this thesis, this is illustrated in Figure 5.5. The core's JPs can be further specialised, aggregated or created upon to create a language which is especially expressive using a simple publish-subscribe like model on existing DSJP without requiring any further aspect-oriented tooling. Furthermore, code which has been used to create an internal DSL in one domain may be reusable across many causing an upward spiral of productivity within projects.

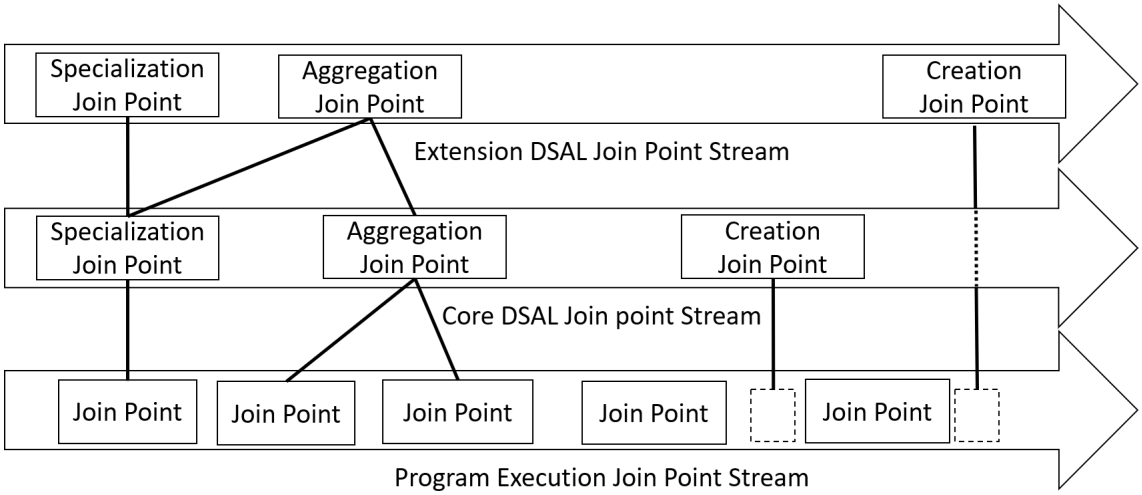

Figure 5.5: The two-step specialisation, aggregation and creation process of creating extension DSJP.

5.3.5 Summary of Approach

In summary, the motivation for our approach is towards making communication runtime inspection DSALs available and viable for using across many ABM in short-term projects. This problem consists of making the approach known, the barrier for entry low and potential to create complex solutions that fit real-world projects.

This is achieved through the use of a middle-out language development process where a core language's middle layer is based on middleware which is consistent across many models. Once this language has been implemented the resulting language can be used directly or as a base to create new DSLs without having to re-implement the weaving.

5.4 Core Implementation

We now discuss how we have implemented a core middle-out DSAL for runtime inspecting JADE ACL messages. We produce a core DSAL as a Java framework using AspectJ GPJPs to weave into the JADE middleware. This framework can be directly used as a fluent interface DSL at the core message filter level of abstraction or can be used to create higher level internal or external DSLs.

We have decided to use a similar DSJP engine approach to Chapter 4, modified to a filter approach for pointcuts and Java lambdas as advice rather than creating an advice engine or event-based parser. The filter model suits message passing domains well, and allowing Java advice means that an end-user can create their own advice libraries, reducing the scope of the core to functional necessities.

The architecture of our core is shown in the dependency diagram in Figure 5.6. There is one instance of the method pointcut class for each pointcut, and this contains each of our method filters which hold the contents of our DSJP model. Each pointcut holds one advice which is executed if the pointcut is met on a JP. Everything flows into the JP class which is a wrapper over GPJPs for sending and receiving an ACL message, in our implementation, these are AspectJ JPs.

5.4.1 Join Point Model

The first step is the creation of the semantic model which will be used to create our DSJP model using a set of base filters. These JPs are created from two AspectJ GPJP shown in Listing 5.1, namely handle send and handle receive in the JADE framework with copies of parameters for filtering. These two GPJP allow for recording all ACL messages sent throughout a JADE model.

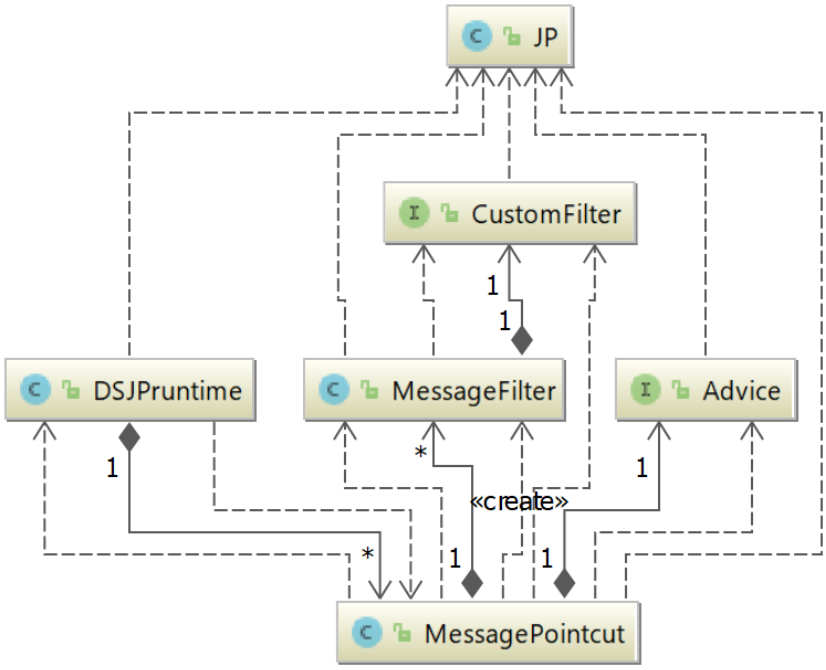

Figure 5.6: JADERI core class dependency diagram.

The base of our semantic model is a subset of JADE's ACL message properties and the filter operations we can perform on them; we outline these parts first as interfaces and enums within the message filter class as seen in Figure 5.7. The protocols and communicative acts used are interfaces as they may be non-FIPA compliant values to check for. The type of JP is an enum which can be checked statically for compatibility by the provided apply statement. There are still runtime errors which may be caused by using an operator which is not allowed for the type being checked, for example, we do not allow contains operator on integers. This could be statically checked by having different operator types for string and integer although causes repetition of code; this is a downside of using an internal approach because extra compiler semantic checks are not possible.

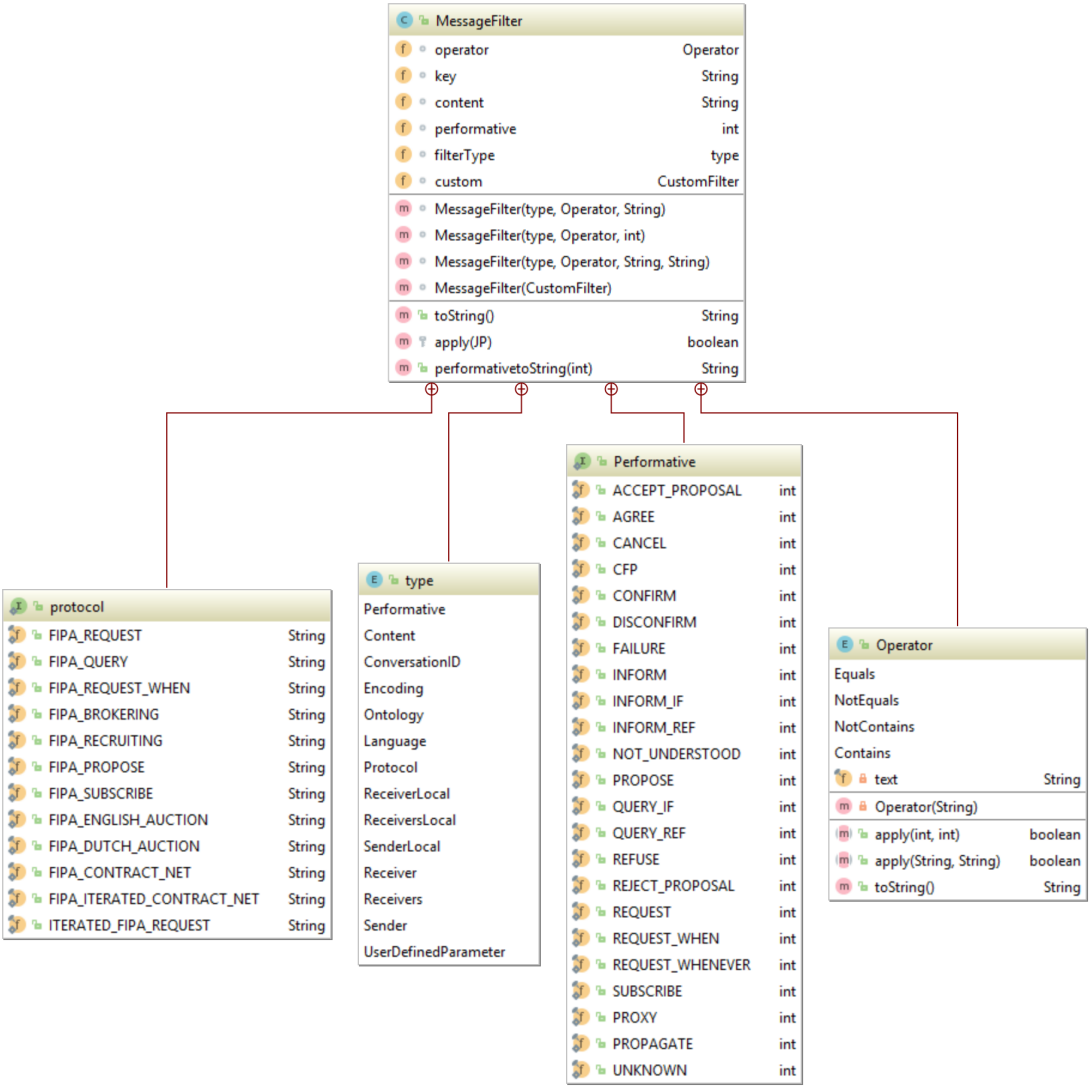

Figure 5.7: Message filter class implementation.

Additional filters can be added using the custom filter lambda which performs arbitrary code onto a JP, returning a boolean result.

Listing 5.1: AspectJ implementation of send and receive handling aspect.

```
pointcut Send(ACLMessage msg, AID aid, boolean clone) : execution(
   void jade.core.AgentToolkit.handleSend(ACLMessage, AID, boolean)
       )
      && args(msg, aid, clone);

before(jade.lang.acl.ACLMessage msg, AID aid, boolean clone) :
   Send(msg, aid, clone) {
      if(DSJPruntime.runtime.active) {
         JP point = new JP(JP.JPtype.send, aid, msg);
         DSJPruntime.runtime.InputGPJP(point);
      }
}


pointcut Receive(AID aid, ACLMessage msg) : execution(
   void jade.core.AgentToolkit.handleReceived(AID, ACLMessage))
      && args(aid, msg);

before(AID aid, ACLMessage msg) :
   Receive(aid, msg) {
      if(DSJPruntime.runtime.active) {
         JP point = new JP(JP.JPtype.receive, aid, msg);
         DSJPruntime.runtime.InputGPJP(point);
      }
}
```

5.4.2 Identification

The identification for our model is done through adding filters to a pointcut to match a set of JPs. This can be done through a fluent interface or separate method calls.

A cascading system handles the identification of JPs, this object-oriented style of creating the filters suits maintains encapsulation well and is highly extensible. JPs are first input into the DSJP runtime which marshals them to each message pointcut instance which marshals it to each of its message filter instances.

A message pointcut is of type send or receive and defaults as firing on every message of that type, for a pointcut with filters to fire all filters must return true. Message filters process JPs through the operator enum's abstract apply method which has an instantiation for each constant. This is a compromise for using an internal DSL because semantic feedback cannot be provided for non-supported operator calls without the creation of a different filter enum for each type of comparison, requiring a repetition of code. We throw an arithmetic exception if a filter is a non-supported operation such as contains on an integer.

Custom filters can be added for situations which cannot be easily done through the default filters. These filters are Java lambdas with the parameters of the JP and pointcut which return a boolean for matching a pointcut. They are treated the same as default message filters above their level of abstraction.

5.4.3 Effect

A Java lambda provides the advice within our core with the JP and pointcut as parameters. This allows for unfiltered access to runtime inspect code at the JP in Java. Auxiliary features such as graphing libraries and analytic toolkits are not within the scope of the core framework, and our internal DSL approach supports their integration on a pay as you go basis.

Our proposed method of extending advice is by wrapping the JP or pointcut into objects which supply domain-specific and declarative features for use by end users.

An external DSL or GUI wrapper could be used as the advice mechanism with a compilation step into a lambda. This could be useful for live programming or involving novice programmers with a system such as this, although requires significant work as seen in Chapter 4.

5.4.4 Runtime Toggling

Toggling of individual pointcut advice is supported in the core framework. A GUI is not within the scope of the core as we believe it cross-cuts with the target's GUI. The addition of a GUI to the core language is a good use for Aspect-Oriented Programming (AOP) or could be implemented by extending the core library.

We add a simple GUI for the toggling of pointcut advice using a JavaFX list view updated using an AspectJ aspect as seen in Listing 5.2. It is a simple use of AOP which requires no extra tooling as AspectJ is already used in our projects to Marshall GPJP into the core. The aspect ties into the filter adding method of every message pointcut and calls to update the list ensuring an updated list at all times.

Listing 5.2: AspectJ implementation of updating the list of aspects for the toggling interface.

```
pointcut ModifiedFilterPointcut() : execution(
 com.JADERI.MessagePointcut com.JADERI.MessagePointcut.Filter(..));


after() : ModifiedFilterPointcut() {
   Main.updateToggleList();
}
```

Live coding or changing of aspects is supported by changing the filter objects at runtime although we have not implemented this in our experiments. This is more suited towards an external DSL approach because to allow full support for advice a Java Integrated Development Environment (IDE) would be required rather than a lightweight external DSL IDE such as a web interface generated by Xtext.

5.5 Use of Core and Internal DSAL Extensions

Through this section, we will show the core DSAL implementations for a set of example models and any internal DSAL extensions we have created for these models. We have used example models shipped with the JADE framework as they suit our needs well in showing off the runtime inspection capabilities of the DSALs. As we focus on the middleware implementation for our filtering, we could substitute these models for many others.

The generality of models the core can support further backs up our choice to remain with internal DSALs for this project. The use of internal DSALs allows for time to be spent only implementing the important domain-specific features and leaving edge case completeness for general-purpose code as needed. The access to underlying Java, especially in internal DSAL backends means that edge cases can be dealt with in a pay as you go fashion rather than requiring pre-emption from a language designer. We can then move into very domain-specific internal DSALs for domain experts who may not be competent programmers.

5.5.1 Thanks Agent

Our first example is the thanks agent example packaged with JADE. This model proposes an agent creating two agents and having a simple conversation with them. We have chosen to use this model as it is a clean showcase of ACL message passing in JADE and the stock example already has runtime logging included which we can use as a baseline for our results.

The included runtime logging regarding ACL message passing is done through print line statements scattered and tangled throughout the implementation as is shown in Figure 5.8. This can be used as a benchmark for our solution's scattering and tangling. Note how this is the level of scatter and tangle in a simple model with a single agent class, it would be less manageable as the project scales.

We have re-produced this logging using our core framework and then created a further internal DSAL for runtime inspecting at an application-specific abstraction level. We can see an example of our core DSAL implementation in Listing 5.3, this uses the fluent interface of the core DSL to chain the instantiation, filtering and advice of two of the print statements from the example. Note how the core DSAL works at the implementation-level of abstraction for a generic ACL message, and as such, the filter setup requires full context and the advice uses the base JADE ACLMessage class as a parameter.

Although the amount of code required to express the concern is more than the in-line implementation, there is a clear improvement in scattering and tangling because all runtime inspection is moved into a separate file and may be toggled on or off at will.

```
public class ThanksAgent extends Agent {
   private static boolean IAmTheCreator = true;
   private int answersCnt = 0;
   public final static String GREETINGS = "GREETINGS";
   public final static String ANSWER = "ANSWER";
   public final static String THANKS = "THANKS";
   private AgentContainer ac = null;
   private AgentController t1 = null;
   private AID initiator = null;
   protected void setup() {
      System.out.println(getLocalName()+" STARTED");
      Object[] args = getArguments();
      if (args != null && args.length > 0) {
         initiator = new AID((String) args[0], AID.ISLOCALNAME);
      }
      try {
         DFAgentDescription dfd = new DFAgentDescription();
         dfd.setName(getAID());
         DFService.register(this, dfd);
         System.out.println(getLocalName()+" REGISTERED WITH THE DF");
      } catch (FIPAException e) {
         e.printStackTrace();
      }
      if (IAmTheCreator) {
         IAmTheCreator = false;
         String t1AgentName = getLocalName()+"t1";
         String t2AgentName = getLocalName()+"t2";
         try {
            AgentContainer container = (AgentContainer)getContainerController();
            t1 = container.createNewAgent(t1AgentName, ThanksAgent.class.getName(), null);
            t1.start();
            System.out.println(getLocalName()+" CREATED AND STARTED NEW THANKSAGENT:"+t1AgentName + " ON CONTAINER "+container.getContainerName());
         } catch (Exception any) {
            any.printStackTrace();
         }
         Runtime rt = Runtime.instance();
         ProfileImpl p = new ProfileImpl(false);
         try {
            ac = rt.createAgentContainer(p);
            AgentController t2 = ac.createNewAgent(t2AgentName,getClass().getName(),new Object[0]);
            t2.start();
            System.out.println(getLocalName()+" CREATED AND STARTED NEW THANKSAGENT:"+t2AgentName + " ON CONTAINER "+ac.getContainerName());
         } catch (Exception e2) {
            e2.printStackTrace();
         }
         ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
         msg.setContent(GREETINGS);
         msg.addReceiver(new AID(t1AgentName, AID.ISLOCALNAME));
         msg.addReceiver(new AID(t2AgentName, AID.ISLOCALNAME));
         send(msg);
         System.out.println(getLocalName()+" SENT GREETINGS MESSAGE  TO "+t1AgentName+" AND "+t2AgentName);
      }
      addBehaviour(new CyclicBehaviour(this) {
         public void action() {
            ACLMessage msg = receive(MessageTemplate.MatchPerformative(ACLMessage.INFORM));
            if (msg != null) {
               if (GREETINGS.equalsIgnoreCase(msg.getContent())) {
                  System.out.println(myAgent.getLocalName()+" RECEIVED GREETINGS MESSAGE FROM "+msg.getSender().getLocalName());
                  ACLMessage reply = msg.createReply();
                  reply.setContent(ANSWER);
                  myAgent.send(reply);
                  System.out.println(myAgent.getLocalName()+" SENT ANSWER MESSAGE");
               }
               else if (ANSWER.equalsIgnoreCase(msg.getContent())) {
                  System.out.println(myAgent.getLocalName()+" RECEIVED ANSWER MESSAGE FROM "+msg.getSender().getLocalName());
                  ACLMessage replyT = msg.createReply();
                  replyT.setContent(THANKS);
                  myAgent.send(replyT);
                  System.out.println(myAgent.getLocalName()+" SENT THANKS MESSAGE");
                  answersCnt++;
                  if (answersCnt == 2) {
                     try {
                        Thread.sleep(10000);
                     }
                     catch (InterruptedException ie) {}
                     try {
                        ac.kill();
                        t1.kill();
                        IAmTheCreator = true;
                        if (initiator != null) {
                           ACLMessage notification = new ACLMessage(ACLMessage.INFORM);
                           notification.addReceiver(initiator);
                           send(notification);
                        }
                     }
                     catch (StaleProxyException any) {
                        any.printStackTrace();
                     }
                  }
               }
               else if (THANKS.equalsIgnoreCase(msg.getContent())) {
                  System.out.println(myAgent.getLocalName()+" RECEIVED THANKS MESSAGE FROM "+msg.getSender().getLocalName());
               }
               else {
                  System.out.println(myAgent.getLocalName()+" Unexpected message received from "+msg.getSender().getLocalName());
               }
            }
            else {
               block();
            }
         }
      });
   }
   protected void takeDown() {
      try {
         DFService.deregister(this);
         System.out.println(getLocalName()+" DEREGISTERED WITH THE DF");
      } catch (FIPAException e) {
         e.printStackTrace();
      }
   }
}
```

Figure 5.8: Scatter and tangle of message specific logging in default thanks agent implementation.

Listing 5.3: A snippet of code from the runtime inspection of the thanks agent example using the core DSAL.

```
new MessagePointcut(send)
   .Filter(MessageFilter.type.Content, MessageFilter.Operator.
      Equals,"GREETINGS")
   .advice((JP, PC)-> {
     System.out.println(JP.getSource().getLocalName() +
        " SENT GREETINGS MESSAGE TO " +
        AIDListtoString(JP.getMsg().getAllReceiver()));
   });
new MessagePointcut(receive)
   .Filter(MessageFilter.type.Content, MessageFilter.Operator.
      Equals,"GREETINGS")
   .advice((JP, PC)-> {
     System.out.println(JP.getSource().getLocalName() +
        " RECEIVED GREETINGS MESSAGE FROM " +
        JP.getMsg().getSender().getLocalName());
   });
```

We created an internal DSAL on top of the core for this model which replaces implementation-specific boilerplate with model-specific enumerated types with associated methods to fill the boilerplate. The class diagram for this can be seen in Figure 5.9. This still leaves the core DSAL objects available for use by the internal DSAL, meaning backwards compatibility is maintained. The advice of this DSL is made more domain-specific by providing a wrapper class around the JP parameter of the advice lambda. This wrapper method moves advice method calls towards the model's intention level by providing the boilerplate around the implementation objects.

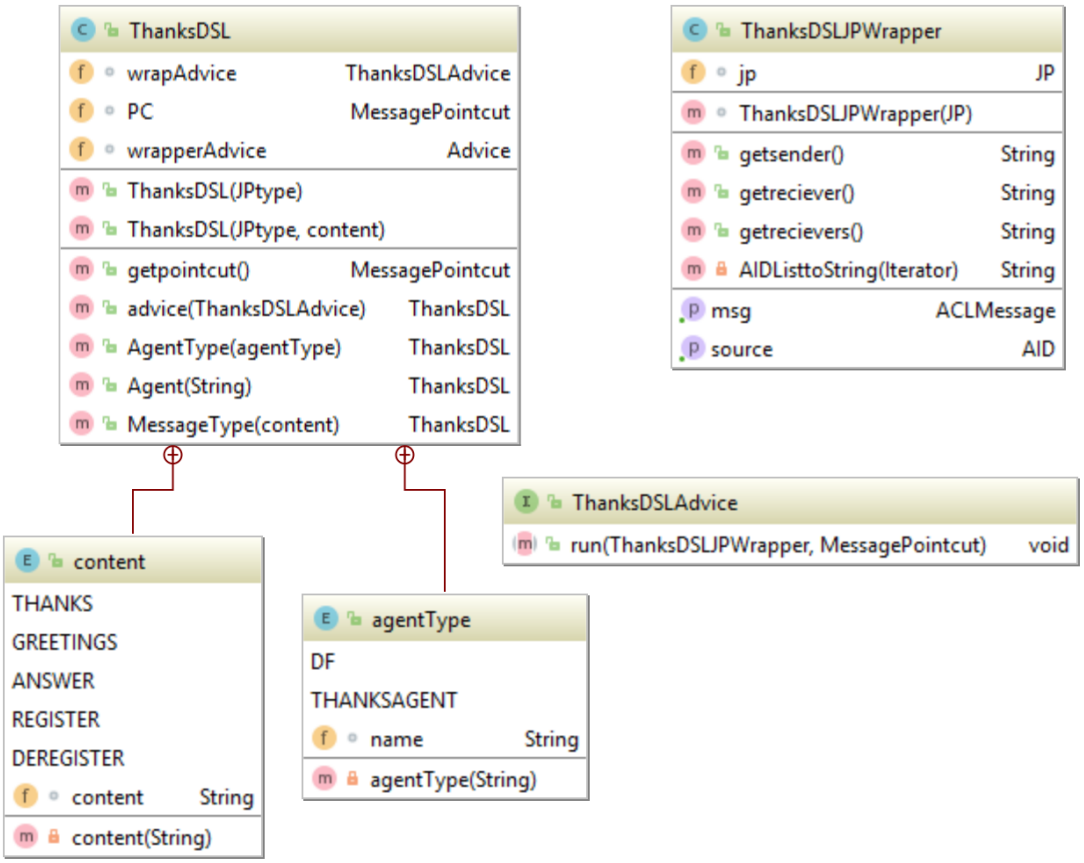

Figure 5.9: Thanks agent internal DSAL class diagram.

We can see an example of the code created by this DSAL in Listing 5.4. The difference between this and the core implementation is in the movement of abstraction level into domain-centric language rather than the implementation specifics of the ACL message passing. This means that the code is slightly shorter because of removing boilerplate and the intention of the filters is clear in regard to the model.

Listing 5.4: A snippet of code from the runtime inspection of the thanks agent example using an internal DSAL extension.

```
new ThanksDSL(send)
   .MessageType(GREETINGS)
   .advice((jp, PC) -> {
      System.out.println(jp.getsender() +
         " SENT GREETINGS MESSAGE TO " +
         jp.getrecievers());
   });

new ThanksDSL(receive)
   .MessageType(GREETINGS)
   .advice((jp, PC) -> {
      System.out.println(jp.getreciever() +
         " RECEIVED GREETINGS MESSAGE FROM " +
         jp.getsender());
   });
```

We also provided further application specificity by allowing pre-made filters to be packaged into the DSAL through constructor arguments. A full implementation of the runtime inspection provided in Figure 5.8 is shown in Listing 5.5. This approach means that commonly used pointcuts can be added declaratively into a project without consideration for implementation or added to save time on small modifications. This approach gives a large decrease in program size and makes pointcuts essentially declarative, but requires heavy amounts of boilerplate writing in the implementation of the DSAL and, as such, is only suitable for commonly used pointcuts.

Listing 5.5: The full code from the runtime inspection of the thanks agent example using pre-built filters in an internal DSAL extension.

```
new ThanksDSL(send,GREETINGS);
new ThanksDSL(receive,GREETINGS);
new ThanksDSL(send,THANKS);
new ThanksDSL(receive,THANKS);
new ThanksDSL(send,ANSWER);
new ThanksDSL(receive,ANSWER);
new ThanksDSL(receive,REGISTER);
new ThanksDSL(receive,DEREGISTER);
```

5.5.2 Book Trading

The book trading example presents a simple multi-agent system for trading books between buyers and sellers; it is featured throughout the JADE programming documentation (Bellifemine et al., 2007). We have modified the book trading example to showcase other features of our approach, especially the use of Java around our aspects and the custom filters. We have made booksellers stock an unavailable book when requested to allow for performance timed runs of book sales.

A common inspection task on such a model would be tracking who buys which books and from who, in the default model during the sale acceptance message the book purchased is not mentioned. As the runtime inspection framework's scope only covers the ACL messages and not internal agent state, this means a workaround must be created to store which book an agent is trying to buy and cross-reference this with successful sales. The core DSAL code for this is shown in Listing 5.6. This shows the use of a hash map to maintain state between messages and using this to display information through print statements. The potential to do this is a benefit of the internal DSAL approach because these Java types are provided without

implementation as would be required in an external DSAL. This operation could be moved into the back-end of an extensions DSAL if it is commonly required through the model.

To test the implementation of our hash map aspect, we modified the book trading model to send an object of type book containing the book's title when a sale is successful. We use a custom filter for this because the filtering of content objects is not within the scope of the default filters. The code for this can be found in Listing 5.7. In this example we combine a custom filter with other filters to ensure the custom filter is only used when necessary, this could also be done within the custom filter as if statements.

Listing 5.6: An example of using the Java advice to do non-straightforward message inspection tasks.

```
HashMap bookTarget = new HashMap<String,String>();

new MessagePointcut(send)
   .Filter(MessageFilter.type.SenderLocal, MessageFilter.Operator.
      Contains, "Buyer")
   .Filter(MessageFilter.type.Performative, MessageFilter.Operator.
      Equals, MessageFilter.Performative.CFP)
   .advice((JP, PC) -> {
     bookTarget.put(
       JP.getSource().getLocalName(), JP.getMsg().getContent());
   });

new MessagePointcut(send)
   .Filter(MessageFilter.type.SenderLocal, MessageFilter.Operator.
      Contains, "Seller")
   .Filter(MessageFilter.type.Performative, MessageFilter.Operator.
      Equals, MessageFilter.Performative.FAILURE)
   .advice((JP, PC)-> {
     String reciever = (
       (AID)JP.getMsg().getAllReceiver().next()).getLocalName();
     System.out.println(reciever + " FAILED PURCHASE OF " +
       bookTarget.get(reciever) + " FROM " +
       JP.getMsg().getSender().getLocalName());
   });
```

Listing 5.7: An example of using a custom filter instead of the provided filters to check content objects in messages.

```
new MessagePointcut(receive)
   .Filter(MessageFilter.type.ReceiverLocal, MessageFilter.Operator
      .Contains, "Buyer")
   .Filter(MessageFilter.type.SenderLocal, MessageFilter.Operator.
      Contains, "Seller")
   .Filter(MessageFilter.type.Performative,
     MessageFilter.Operator.Equals,
     MessageFilter.Performative.INFORM)
   .Filter((JP) -> {
     boolean result = true;
     try {
        String msgbooktitle = ((Book)JP.getMsg()
           .getContentObject()).getTitle();
        String aspectbooktitle = (String)bookTarget.get(
           JP.getSource().getLocalName());
        if (msgbooktitle.equals(aspectbooktitle)) {
           result = false;
        }
     }
     catch(Exception e) {
        System.out.println(e.toString());
     }

     return result;
   })
   .advice((jp, PC) -> {
     System.out.println(jp.getMsg().getSender().getLocalName()
        + " Aspect Hashmap Records Inconsistent");
   });
```

5.6 Results

We will now display the code and performance metrics from our implementations of this approach. We first discuss a benchmarking model we have made to test the scalability of this approach then move to the models discussed in the previous chapter.

Experiments are performed on a Windows 10 machine with an i7-4790k @ 4.4ghz with minimum processor frequency at 100% to reduce variability and 16GB of 2133mhz cl9 ram. No performance throttling issues were encountered through testing. These experiments were run from batch scripts, timings are taken using Java's system nano time method. Code metrics are consistent with those taken in Chapter 4; characters are counted excluding white space and a logical entity is a variable use, method call (method chains taken as a single call) or a stand-alone keyword. Each experiment has 20 data points, represented as box plots in our figures. The models we have used for this testing are a custom message passing benchmarking model, the JADE examples thanks agent model and the JADE examples book trading model.

5.6.1 Benchmark Model

For performance benchmarking, we have created a simple model of a sending agent and a receiving agent who exchange a set number of messages and then shut down. We use this model to reliably test the DSALs against the performance of JADE using a synthetic load.

We performed tests with 1000 messages and 1000000 messages on three instances of the benchmark model. A base model without aspects, a single matching filter aspect shown in Listing 5.8 and nine matching filter aspect shown in Listing 5.9.

Listing 5.8: Core DSAL aspect code for benchmarking with 1 string comparison filter.

```
new MessagePointcut(receive)
   .Filter(MessageFilter.type.SenderLocal, MessageFilter.Operator.
      Contains, "B")
   .advice((jp, pc) -> {
     i++;
   });
```

Listing 5.9: Core DSAL aspect code for benchmarking with 9 string comparison filters.

```
new MessagePointcut(receive)
   .Filter(MessageFilter.type.SenderLocal, MessageFilter.Operator.
      Contains, "B")
   .Filter(MessageFilter.type.SenderLocal, MessageFilter.Operator.
      Contains, "E")
   .Filter(MessageFilter.type.SenderLocal, MessageFilter.Operator.
      Contains, "N")
   .Filter(MessageFilter.type.SenderLocal, MessageFilter.Operator.
      Contains, "C")
   .Filter(MessageFilter.type.SenderLocal, MessageFilter.Operator.
      Contains, "H")
   .Filter(MessageFilter.type.SenderLocal, MessageFilter.Operator.
      Contains, "M")
   .Filter(MessageFilter.type.SenderLocal, MessageFilter.Operator.
      Contains, "A")
   .Filter(MessageFilter.type.SenderLocal, MessageFilter.Operator.
      Contains, "R")
   .Filter(MessageFilter.type.SenderLocal, MessageFilter.Operator.
      Contains, "K")
   .advice((jp, pc) -> {
     i++;
   });
```

The results for 1000 messages are shown in Figure 5.10 and 1000000 messages in 5.11. The experiment is split into time to send and time to receive all messages, as can be seen, these are closely correlated inferring the sending of the messages is

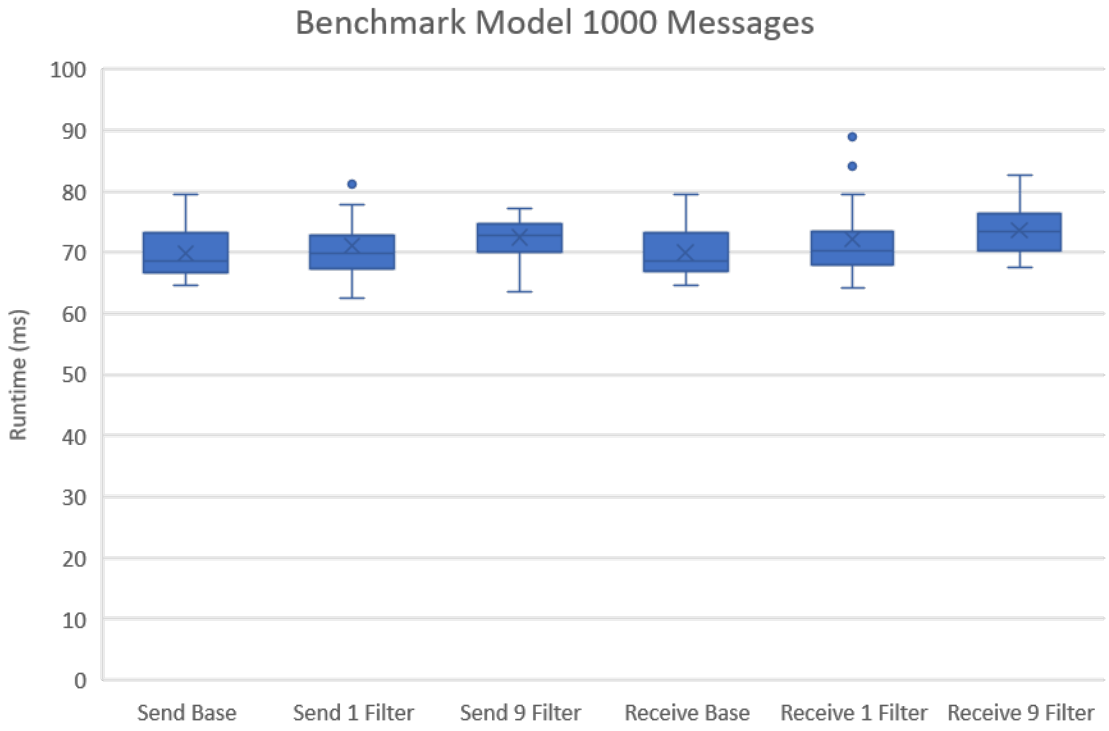

Figure 5.10: Benchmark model 1000 message experiment results.

the bottleneck due to queue size limits. The addition of our sample aspects causes a low impact on this model, in real-world tests the relative impact would be even less because agent behaviours would be a larger percentage of runtime.

Early work on the scalability of JADE by Cortese et al. (2003) found JADE scales linearly when adding thousands of agents, DFs, containers etc. Our experiments show moving from 1000 to 1000000 messages only caused roughly a 40x increase in performance time. This synthetic load test for a high number of messages with good scaling further compounds that JADE is a suitable framework for large complex interactive agent-based systems.

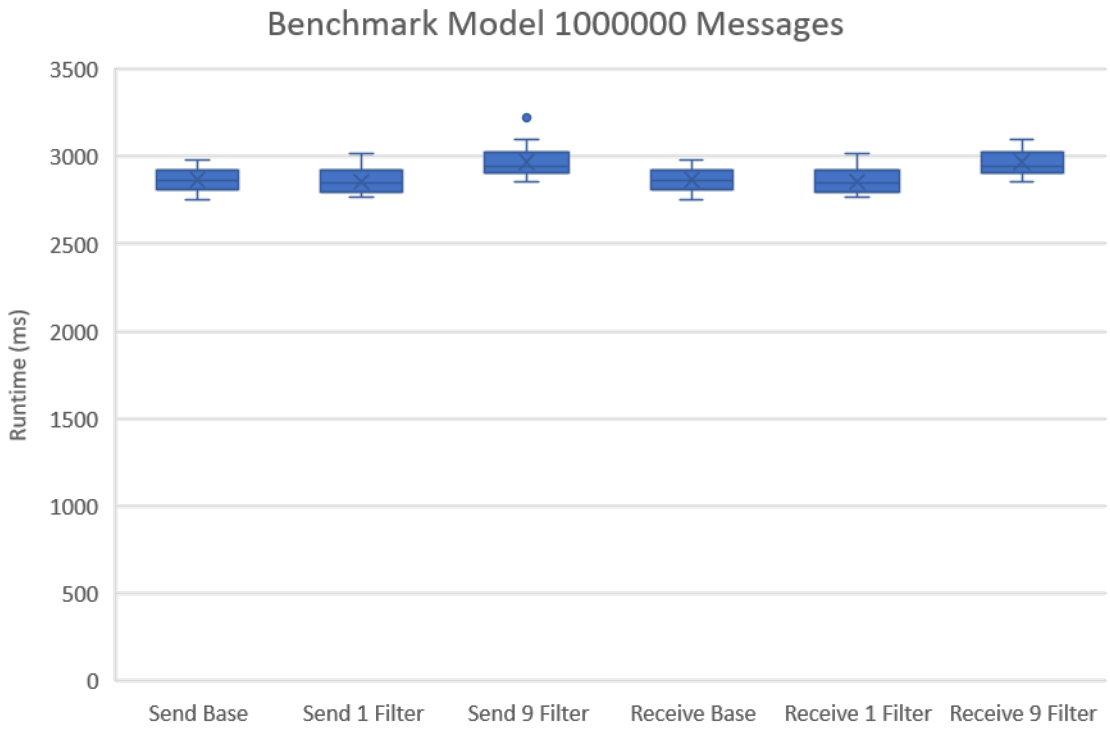

Figure 5.11: Benchmark model 1000000 message experiment results.

5.6.2 Thanks Agent

To test the performance of aspects on the thanks agent example, we have recreated the message logging performed by inline system print line statements throughout the base model using our aspects and commented these out for the tests with our aspects. We test both the core DSAL implementation and internal DSAL versions, code snippets can be found in Listing 5.4 and Listing 5.3.

Performance results can be found in Figure 5.12. There is a very low overhead to this approach because the filters are simple single checks for message type to log, the overhead of the advice is included in the inline approach, and most messages require advice thus do not waste time checking messages not related to the concern.

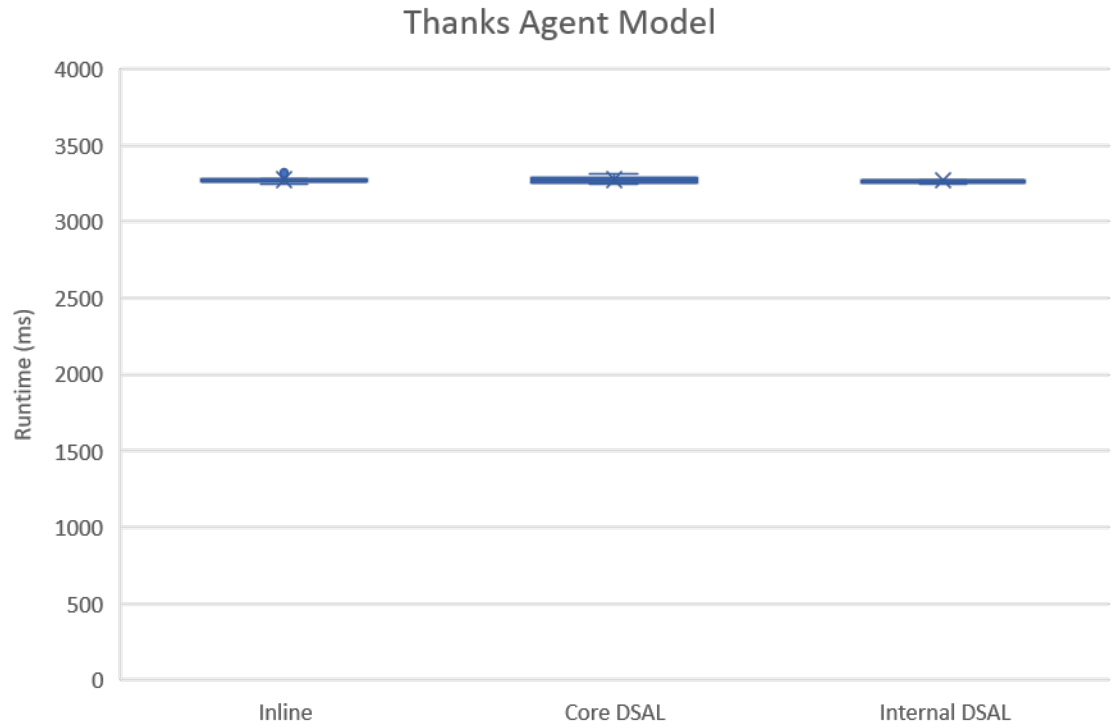

Figure 5.12: Thanks agent inline, core DSAL and internal DSAL experiment results.

There is no statistical performance difference between the core DSAL and extension DSAL because they are essentially equal in implementation, the only performance overhead of the internal DSALs are the construction of filters at a message pointcut's population.

Code metrics for the original print statements, core DSAL and two versions of the DSAL extension are found in Table 5.3. The inline print statements are included as a base context for the aspects although it should be noted the aspects have the additional concern of identifying the JP which is not included in the size of the print statements. As such, the core and specific DSAL approaches include the same print statement and the constructor DSAL has it implemented as a pre-made option.

	Characters	Logical Entities	Scatter	Tangle
Inline Printing	641	30	8 blocks over 112 lines	Inside agent class
Core DSAL	1920	109	8 continuous pointcut advice	Independent aspect
Internal DSAL	1132	88	8 continuous pointcut advice	Independent aspect
Constructor DSAL	235	24	8 continuous pointcut advice	Independent aspect

Table 5.3: Code metrics for recreation of thanks agent message logging.

Furthermore, every character or logical entity is not equally expressive or difficult to write. These subjective measures of code are difficult to bring out in quantitative analysis. We give a visual comparison in Listing 5.10 for the difference in abstraction level between the DSALs. The move from imperative implementation-level code, through domain-specific population code and finally to declarative application-specific code can be seen.

Listing 5.10: Code samples of each type of implementation.

```
// Inline Print
System.out.println(getLocalName() + " SENT GREETINGS MESSAGE TO " +
     t1AgentName + " AND " + t2AgentName);

// Core DSAL
new MessagePointcut(send)
   .Filter(MessageFilter.type.Content, MessageFilter.Operator.
       Equals,"GREETINGS")
   .advice((JP, PC)-> {
      System.out.println(JP.getSource().getLocalName() +
         " SENT GREETINGS MESSAGE TO " +
         AIDListtoString(JP.getMsg().getAllReceiver()));
   });

// Specific DSAL
new ThanksDSL(send)
   .MessageType(GREETINGS)
   .advice((jp, PC) -> {
      System.out.println(jp.getsender() +
         " SENT GREETINGS MESSAGE TO " +
         jp.getrecievers());
   });

// Constructor DSAL
new ThanksDSL(send,GREETINGS);
```

5.6.3 Book Trading

To test the performance of aspects on the book trading example, we use the runtime inspection aspects from Listings 5.6 and 5.7, located in Section 5.5.2.

The performance results for trading 10000 books can be seen in Figure 5.13. The runtime is subject to availability of books because of agent's random choices which account for some spread across runs. There is an increase in runtime between the base and aspect-enhanced programs because of the addition of new features although this increase in runtime is within expected bounds. This is because as shown in the

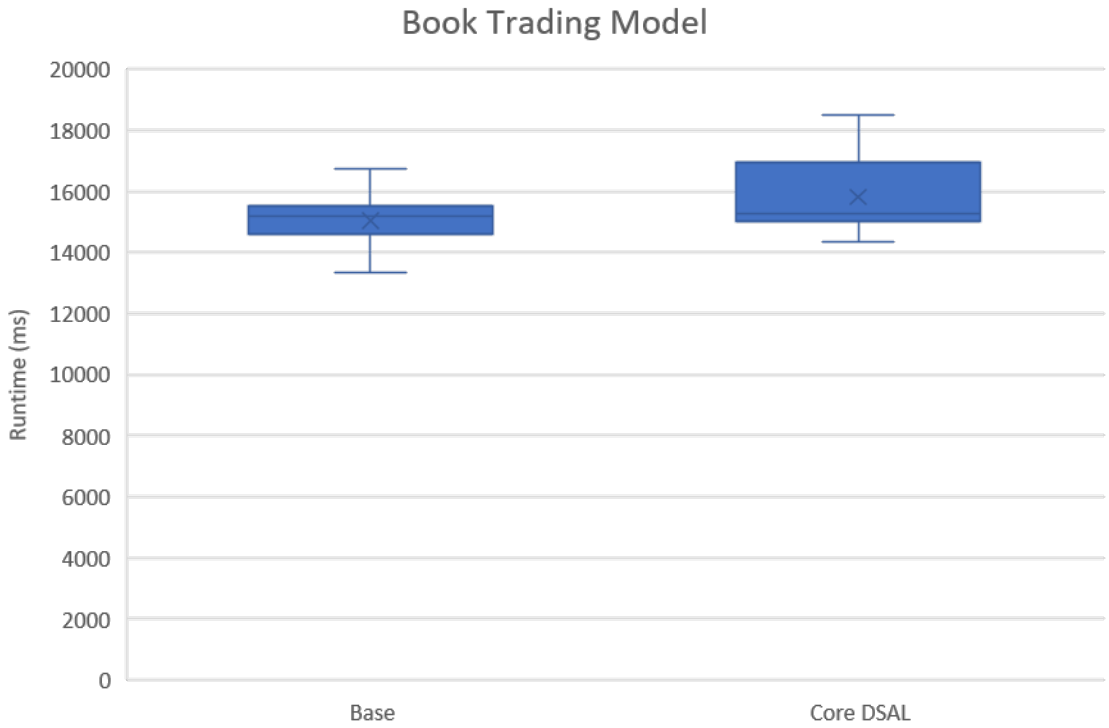

Figure 5.13: Book trading 10000 books experiment results.

benchmark model the filters have little effect on runtime performance and as shown in the talk agent example the advice's runtime increase is similar to that of an inline approach.

	Characters	Logical Entities	Scatter	Tangle
Core DSAL	1055	57	2 continuous pointcut advice	Independent aspect

Table 5.4: Code metrics for our book purchase tracking example.

Code metrics can be found in Table 5.4. Interestingly this single concern implementation required dependency between two pointcut advice. This scattering is unavoidable because the collection of this context cannot be done from the

context of the seller response pointcut advice. The important thing is maintaining understandability of code; it is more beneficial to have slight scattering in aspect code when two concerns meet rather than scattering throughout the implementation. This scattering can be removed at the user level by moving required context collection into the back-end of an extension DSAL or method to package needed filters.

5.7 Discussion

In closing, we will discuss this approach, our implementation and results.

5.7.1 on Limitations due to Concern Specificity

This approach focuses on the middleware-specific implementation of a cross-cutting concern in ABM by providing a core DSAL which can be extended with model-specific DSALs. This allows for coverage of a concern in ABM which is not tied to any specific model, which is a benefit over targeting a specific model. The downside of this is other concerns such as internal agent behaviour, environment context and model statistics are not within the scope of the DSALs. Using model-specific techniques from Chapter 4 a DSAL could be made to support all runtime inspectorate behaviour within a model, although without consideration for re-use across models written in the same framework. Through this chapter, the core DSAL will work for any FIPA ACL communication without requiring re-implementation of weaving logic, but extending support to other concerns may break cross-model compatibility.

This means the inspection of non-communication will not be possible using the DSAL, making projects need to either use an implementation-specific DSAL or scatter and tangle code. Through future work multiple concerns could be provided with

a group of middleware-oriented DSALs targeted towards the frameworks different concerns, leaving only edge case which require re-implementing weavers or scattering and tangling code.

5.7.2 on Internal vs External DSALs

One of the core threads through this chapter has been the comparison between this internal approach and the external approach in Chapter 4. The approach in Chapter 4 is a very high-level external DSAL aimed towards a specific model which is not extensible by the user. This very high-level language allows for domain-level description of what to be inspected and what to print although only allows for simple behaviours. This is in line with the external DSL for a domain expert or common problems scenario, we have traded generality for ease in a specific domain. The internal approach in this chapter, on the other hand, gives power to the person using the DSAL to create and extend languages using full Java code around the DSAL's constructs. This is more in line with a programmer who is working through a project and wants to improve software economics by shielding themselves from the boilerplate of weaving into agent communications and modularising this code appropriately for toggling and maintenance. This internal approach also allows for specific internal DSALs to be created for use by a domain expert as was shown in the thanks agent example from Listing 5.5 where the internal DSAL is used as a declarative language.

From a language developer perspective, the use of an internal DSL as the core form of access to our semantic model eased the development of the project because testing cases and developing extension DSLs felt like a fluid process rather than API checkbox filling. This means that programmers who have a grounding in the

filter-based aspect-oriented approach will be able to use the core system and create extension DSALs without needing to learn a new API style. An internal DSAL as a core entry method to the implementation's semantic model could also be useful for developing external DSAL extensions because through Chapter 4's code generation it would have been useful to be able to write and test generated code samples in a more human-accessible form. We believe this is in line with the concept of a DSL being a step past an API in software maturity because of the readability and maintainability they bring (Mernik et al., 2005).

From a language use perspective, the internal approach foregoes domain-specific IDE support for the familiar IDE environment of the Java program around it. Much of the domain-specific support provided in Chapter 4's external DSL cannot be provided in an internal DSL at compile time, a distinct disadvantage for internal DSLs. Yet internal DSLs can provide runtime feedback through logging and exception handling. There is scope for further enhancement to the internal DSL could be done by using a language such as Scala or Groovy which give more flexibility for language enhancement while remaining comparable with the JVM and Java IDEs.

5.7.3 on Cross-Framework DSAL use

This approach allows the cross-framework use of pointcuts, and through wrapping ACL messages into a platform agnostic wrapper, limited cross-framework use of advice. The implementation we have produced is aimed only at use within JADE because it is the dominant FIPA compliant agent framework. Cross-platform aspects would be especially useful for the verification and validation of models through docking; docking is the process of creating a model in two different frameworks to compare and contrast the outputs of both models (Appleget et al., 2014). Comparing

different implementations allows for identification of mistakes in models although without a cross-framework runtime inspection method the inspection must be written separately for each model.

To allow for transparent cross-framework development, aspects could contain implementation-specific details of different frameworks and select the appropriate implementation depending on runtime context. An example of this would be running the same aspect on two models which have different naming conventions for ping messages; where one model may send 'pings' between agents, another may send 'dings', yet the semantics of agent behaviour remains the same. This allows a single aspect to be used across implementation styles, with the implementation transparently changing for the appropriate naming convention. As JADERI only focuses towards a JADE implementation, this cross-implementation development step is not included in our implementation.

5.7.4 on Potential Adoption in Future Research Projects

The target audience of this approach is researchers who create many models and run many experiments which are implemented using a common middleware. Compelling use-cases for this are team projects where a domain expert is involved or projects which require many different experiments with changing points of interest and parameters. As shown through our results there is a negligible difference between the implementation performance of DSAL aspects compared to inline implementations, yet a considerable difference in the code abstraction level and modularity from the model.

Projects that require runtime inspection of agent behaviour which is not available through the ACL communication of agents are not in the scope of this chapter. We consider the processes and development of other middleware-specific DSALs to be exciting future research following the steps taken through this chapter. An example of further runtime inspection possible in JADE would be weaving into the agent base class to inspect agent lifecycle state with agent-specific extensions for internal states.

Chapter 6

Synthesis of AnimauxRI and JADERI

This section discusses the synthesis of the techniques in our two main contribution chapters. The approaches mentioned in Chapter 4 and Chapter 5 are different variations along the same approach, with significant commonalities and differences to discuss. There is no reason why Chapter 4's approach is limited towards external DSLs and Chapter 5's approach is limited towards internal. The approaches may be used interchangeably or in combination as is suitable for purpose. Chronologically our work began with Chapter 4 and matured into the approach in Chapter 5. The initial work was performed with an external DSL because using a concrete grammar for the middle layers semantic model to be based upon aided the middle-out process. We then moved to implement an internal DSL approach, extending the initial work from Chapter 4. The model-specific work in Chapter 4 is based around the Animaux framework where models are implemented within the framework's step logic without domain-oriented interfaces. The middleware-specific work in Chapter 5 is based

around the JADE framework with set interfaces separately from the underlying framework code. We now discuss the abstract approach which both these chapters utilise, followed by discussion of the considerations for different approaches within the two chapters.

6.1 The Abstract Approach

The middle-out DSAL approach is a software development process which is agnostic towards the type of DSAL used for the middle layer and domain size targetted. This is because the approach first focuses on the creation of a middle layer, centred around a semantic model. A semantic model can then be used to separate a top layer program's population of the model from a bottom layer utility's operation of the model. This technique is an extension of the language-oriented programming paradigm from Ward (1995), paired with the semantic model concept discussed in Fowler (2010). Different approaches towards implementing this abstract process have advantages and disadvantages. We will discuss considerations for an internal or external DSAL and for a model-specific or middleware-specific focus.

6.1.1 DSL Implementation Techniques

The type of DSL chosen effects the language development, use and maintenance. Internal DSLs are generally faster to implement, maintain and document than an external DSL. Internal DSALs are based within a host language and benefit from the features provided by the host language. This is especially useful for advice languages because operations such as looping and standard output will be included in a DSAL as standard. This benefit comes with the disadvantage of being limited

by the extension potential of the host language and not being able to constrain actions within the language using domain-specific context at program write time. For an end-user an internal DSL does not require any new items in the toolchain of a project and it should be relatively simple to include it in projects. Generally internal DSLs are more suited towards software developers than pure domain experts because it provides a familiar development environment. For effective inclusion of domain experts, internal DSAL input can be moved into a separate class file allowing a pseudo-external environment for domain experts. The separation of an internal DSL from base general-purpose code suits aspect-oriented languages.

External DSLs are generally more time consuming to develop and maintain than an internal DSL. In addition, external DSL development techniques generally require knowledge of language development toolkits where internal DSLs can be developed using design patterns from the host language. The formal nature of creating an external DSL may be an aid to a project because it enforces a serious look at the middle layer before moving onto implementation of the bottom layer utilities. External DSLs give more control of language features and development environment possibilities compared to an internal DSL. This is especially useful for pointcut languages because they differ from what is expected in a standard object-oriented host language. For an end-user an external DSL may be presented without a development environment, with a bespoke environment such as an Xtext generated IDE or as a plug-in to an existing IDE such as Eclipse or Intellij. As an external DSL cannot benefit from the host language's infrastructure as an internal DSL can, the provision of suitable development tools and masking of implementation concerns is an important consideration.

There may also be situations where non-conventional methods such as visual or projectional languages would be suitable for a domain. This approach will add language development and use considerations not applicable to the ubiquitous textual programming languages used commonly. This approach will exacerbate the disadvantages of developing an external DSL because of the widened difference from a conventional programming language. This approach may be suitable when dealing with a domain which has clear graphical representations or targeting domain experts who are not textual programmers.

6.1.2 Weaving Target Specificity

The scope of the cross-cutting concerns captured by a DSAL determine the applicability of a language across projects and application-specificity of its concepts. The architecture of the underlying agent simulations have an impact on the feasibility of a model-specific or middleware-specific approach. ABMs which are written without interfaces governing the implementation semantics of a model provide a difficult target to create re-usable weaving rules. Frameworks which use standardised middleware perform a majority of the aspect-mining for the language developer. For example, the FIPA communication framework gives communicative acts and interaction protocols which can be used to describe agent actions.

Creating a model-specific language allows for specificity toward specific applications, at the cost of generality across other frameworks in a model. This approach can allow for a direct domain-specific dialect towards a single set of problems, without considering implementation patterns across models. The implementation of a specific model determines the DSJP representation potential within the model, for example an agent-oriented interface may aid aspect-mining. Aiming at a specific

model removes the potential for re-use en masse across a framework of models but internally some re-use may still occur. The weaver underpinning a model-specific DSAL could contain weaving logic for many DSALs as in Chapter 4, where Sugarscape and Kawasaki use the same weaver. This means that model agnostic join points such as step logic can be reused across models.

Creating a middleware-specific language allows for greater reuse across sets of models which are similar in implementation but can be different in conceptual semantics. This approach first targets the dialect of the implementation of a set of models, such as a domain-oriented protocol. The middleware's cross-cutting concern is then either directly monitored i.e. counting message sends as in Chapter 5 or converted into a model-specific concern by the DSAL's end user i.e. tracking book transactions as in Chapter 5. Aiming at a middleware target means that the models in question must be implemented using a standardised architecture, or re-use across many models may run into compatibility issues.

Both model-specific and middleware-specific methods can be combined within the same set of DSALs. An example use case for this could be within the JADE framework where communications can effectively be captured at the middleware level, but internal agent behaviours may require model-specific join points for effective capture.

6.2 AnimauxRI Discussion

Chapter 4 is the first development of the middle-out DSAL technique which investigates the creation of external DSALs for 2 specific models in the Animaux framework, Sugarscape and Kawasaki. The case study DSLs written during this

chapter are very domain-specific external DSLs, serving as an illustration of the middle-out process developed through Chapter 4. The purpose of this chapter's research was to excavate the potential for this method using two example models with differing characteristics. Model-specific was chosen because Animaux is written using direct modification within a step's evolve mechanic without an interface for implementation of behaviours, this is in contrast to an agent-oriented framework with well-defined interfaces. Implementing a middleware-oriented DSAL where a framework does not have well-defined components interfaced to perform concerns adds a layer of difficulty. In many situations, informal changes in implementation between models will stop such a DSAL from being applicable across models. This means that frameworks without declared interface may suit the model-specific approach more than a middleware-oriented approach. External DSLs were investigated through this chapter because this was our first work into creating a middle-out DSAL and the use of a concrete grammar helps formalise the process. Model-specific languages can benefit from the added control of an external DSL, and potential for standalone use may favour domain expert involvement.

6.2.1 Alternative Approaches with a non-agent-oriented framework

An internal DSL approach could have been applied to Animaux either as a direct replacement of the external DSL, or an internal DSL layer underpinning an external DSL. Directly replacing the external DSL with an internal DSL would reduce the burden of implementation associated with an external DSL, yet also lose the benefits of using a mature language workbench such as Xtext. This design decision would be mainly concerned with the implementation burden of the language against the use

cases of the intended end-user. Having an internal DSL alike the one from Chapter 5 underpinning an external DSAL would allow layered use of the same DSAL features, with little extra implementation cost. For example, a program developer suited internal DSAL to be used during development and a domain expert suited external DSAL for use during operation of models. The API used to populate the pointcut advice engines for AnimauxRI was designed specifically for direct population by the Xtext DSL, without consideration of human readability. This limitation of Chapter 4 brought on the idea for producing an internal DSAL core in Chapter 5. A middleware-specific approach would be difficult to apply to the Animaux architecture. As Animaux agents are arbitrarily altered through evolutions of a storage array, individual agent actions are not explicitly written in a generic way. Cross-cutting concerns within program families which share implementation techniques could be implemented into a middleware-specific DSAL. For example, a model with multiple growth mechanics or an economic model with differing taxation methods could be captured using a single language.

6.3 JADERI Discussion

Chapter 5 focused on a middleware specific approach, implemented using internal DSLs. The case studies through this chapter are used to display differences between a generic core language implementation and domain-specific extension languages as wrappers over this. The middleware focus of this chapter is chosen because concerns which are dealt with across many models can have model-independent runtime inspection implementations. This suits protocol-oriented frameworks such as JADE, especially for agent communications where it implements FIPA standards. This chapter also introduced the concept of chaining together DSALs to reduce

implementation burden for additional targets. This is achieved through a middle layer core language which can be extended to create the bottom layer of higher level extension DSALs for specific models or application areas. Internal DSLs were used through this chapter for the core language to improve on Chapter 4's API for populating the semantic model to be more human readable. Furthermore, creating extension DSALs using an internal DSL does not require additional code generation or tools to be added into the toolchain.

6.3.1 Alternative Approaches with an Agent-Oriented Framework

An external DSL approach could have been used for this chapter for the core language or extension languages upon the core language. As a full middleware level cross-cutting concern is a larger domain than the AnimauxRI example in Chapter 4, using an external DSAL for the core language of this chapter would require a comparatively large amount of work. External DSL creation costs do not scale linearly, and larger domains mean much larger development, documentation and maintenance costs. Extension DSALs for specific purposes being written in an external DSAL would require less development effort than a core external DSAL, and as such may be a more achievable approach. Combining internal and external approaches would be especially interesting because different models sharing this middleware have different purposes and end-users. An experimental model which is being directly inspected by a developer could use either the existing core DSAL, or a low development effort internal DSAL packing commonly used features for this type of problem. An established model which will be experimented on by many external domain experts may benefit from an external DSAL approach which provides them with

the application-specific abstractions they need. This use case showcases the direct interplay between the domain expert level language external DSLs can provide and the higher development, documentation and maintenance burden they require.

A model-specific implementation of agent-communications within the JADE framework is also possible, and given understanding of the JADE architecture may be more straightforward than the implementation in Chapter 4 because of direct agent-oriented interfaces to hook the language from. Directly targeting a subset of the middleware cross-cutting concern to provide support for a discrete set of join points required to examine specific models may require less development effort than implementing support for generic agent interaction. Having said this, the model-specific weaver could not be reused across generic models so this loses re-usability. A main advantage of the middleware targeting approach is that the middleware weaver only needs to be implemented once, then model-specific languages can be created with less development effort. Concerns such as internal-agent behaviour which are not fully standardised by the JADE framework may benefit from a model-specific approach because of the lack of standardised middleware.

6.4 Summary

In summary the middle-out DSAL approach is viable for use with internal, external or mixed types of DSL. This thesis has focused on an external approach for specific models within an in-house framework which does not have domain-oriented interfaces and an internal approach for a middleware concern within a framework using domain-oriented interfaces. We believe the model-specific approach is equally suited to malleable and interfaced frameworks. It is suited towards malleable code bases because difficult DSJP's underlying GPJPs can be written as specific implementations

inside a framework. Writing generic wrappers of GPJP for difficult DSJP may be difficult with an established interface. The model-specific approach is also suitable for interfaced projects where subsets of multiple interfaces can be joined together without having to implement all interface GPJP. On the other hand, we believe the middleware approach is more suited towards stable interfaced frameworks because they give a stable base to target, improving both maintainability and ease of domain capture. In malleable solutions targeting an adhoc middleware may be difficult to scope and test, furthermore reusability across models will be subject to adherence to malleable processes.

The selection of using an internal or external language should generally be based on the end users of the program and the anticipated effort to creating such a language, similar to the process undertaken when developing a DSL. For example, generally an internal language will take less development effort than an external one, yet the external DSL may give a more enticing domain-specific jargon for an end user. The consideration of language development time and effort should be a significant one, especially if the language developer will not be available for the duration of the project.

In future work, we are especially excited for mixed methods which use an internal DSAL as a core language which can be extended by external, internal or visual languages depending on the project needs. We especially think this is valuable for targeting the middleware of large frameworks, where a mature core DSAL can have significant development and maintenance time because it has a wide potential for use.

Chapter 7

Conclusion

We now conclude our research in relation to the hypotheses set out in Chapter 1, the contributions within this thesis and future work.

7.1 Relation to Initial Hypothesis

The hypothesis we laid out at the start of this thesis are:

- Applying a middle-out approach utilizing a semantic model to Domain-Specific Aspect Language (DSAL) development and use allows separation between domain-application and weaving implementation. This separation allows multiple sets of aspects, with differing weaving implementations to be written using the same middle layer across multiple instances of one or more models.
- Using custom-built middle-out DSALs allows sets of runtime inspection experiments to be run on Agent-Based Models (ABMs) with reduced scatter, tangle and boilerplate code compared to inline object-oriented methods.

- A direct General-Purpose Aspect Language (GPAL) implementation of Domain-Specific Join Points (DSJPs) will provide similar to inline performance when there is a direct mapping to DSJP available, yet poor performance when not. Dependency injection of General-Purpose Join Points (GPJPs) into target code may be used to allow better performance, with the disadvantage of more scattering throughout the base code.

Through this thesis, we have implemented and tested our approach with performance and code metrics which show it is feasible to use in short projects. The reduction in scatter, tangle and semantic gap improves software economics, especially where the small performance hits are cheap because computational time is cheaper than programmer time. The performance drop was manageable through our experiments with the interpreter approach although for DSJP with many parameters or many potential instantiations, the injection approach offers superior performance. The performance becomes more important as model runs become longer or computation time costs more.

We designed and implemented semantic-model based middle-out processes for both model-specific DSALs and middleware cross-cutting concern specific DSALs which can be extended for specific models. We have created internal and external DSALs with transparent aspect orientation. Transparent aspect-orientation does not harm the use of the DSAL because domain-level Integrated Development Environment (IDE) support gives feedback which is at a more appropriate level than implementation-specific first-class GPAL support.

The main discussion point we found in regards to performance is how 'good' is good enough for a particular project, creating a simple DSAL which covers common tasks takes little time compared to creating a fully-fledged DSAL with IDE support and high performance.

We rank the approaches for creating DSJP as follows: AspectJ Join Points (JPs) are fast to create and maintainable in an aspect-oriented fashion although have poor performance where there is not a direct GPJP to DSJP mapping; DSJP call injection is fast to create with near inline performance although causes high scatter; creating a code transformer is the most difficult approach and causes high coupling especially if the target language is a full GPL although this offers inline performance without code scattering.

If we forego obliviousness by including the model in the bottom layer of our DSAL, we can improve the ease of aspect mining and performance. A problem for this is despite how beneficial a DSAL can be for a project, it hampers model development by adding in the consideration of a Domain-Specific Language (DSL) which exists separately from the core concerns of the system. This can be considered a more middle-out approach because the runtime inspection language is considered as the 'bottom layer' ABM is implemented. While this may be extreme for specific models, considering the DSAL before implementing middleware may be a useful option if a high-performance DSAL add-on to a framework is intended to be provided. In the case of the cross-cutting concern oriented DSALs more time spent on the core DSAL may be worthwhile because of the considerable re-use potential.

We believe the most influential idea for use within academia and industry is Chapter 5's DSAL aimed at middleware because it allows a high-quality core DSAL to be created which can then be shared among projects. This type of DSAL creation

could aid large frameworks such as Repast Simphony which wants to find ways to provide facilities for verification and validation of models (North et al., 2013). The large base of models available for use with the DSAL and ready availability for those already familiar with a framework means that a middle-out approach beginning with scoping of what sort of DSALs would be ideal for this middleware, then moving towards its implementation and use may be worthwhile. The ability to further specialise the core DSALs is important as well, because it brings the pay as you go philosophy into development so a core DSAL for a concern can be minimalistic with extra features only developed by those who require them. Without this the core DSALs would have to be massive languages with extensive features for the domains. This is similar to the approach taken by the MASON framework providing a minimal sandbox for ABM which openly supports DSL extensions and re-use of its components (Luke et al., 2005).

7.2 Our Contributions

We now discuss the three main contribution areas of this thesis in regard to providing DSALs for scientific projects.

Middle-Out DSALs

We began this thesis by defining the concept and use of DSALs with an emphasis towards DSLs. We present this definition chapter because we believe having a strongly seated philosophy aids the adoption and use of emerging techniques. We bring concepts from other fields such as prototype theory in this chapter because they are rarely considered through computer science research although play a vital

role in how the field and research around the field develops. The languages which are considered DSLs by an average software engineer will be different to those of a project scientist primed with the discussion in our Chapter 2, and these changes will affect the outcomes of projects.

This is especially important for the field of DSALs because it is a niche of domain-specific within a niche of aspect-oriented. Without informed practitioners there will be a lack of suitable application areas for research and the techniques developed through research will not be brought to fruition. We would like to aim our research towards creating confluences between prominent application areas and interesting domain-specific and aspect-oriented research areas as is discussed by Walker (2016).

The combination of work from DSLs into DSALs has been covered several times through the literature. This thesis differs from the other work by taking a strong stance on making DSALs which have transparent aspect orientation rather than trying to provide first-class aspect orientation. The processes described in this thesis only require project-specific code, main-stream aspect libraries or DSL language workbenches. Despite not offering new tooling within our contributions, we believe that the improvement of tooling is an important issue and new tooling can be integrated into approaches such as the ones presented in this thesis. This is similar to how language workbenches have improved large external DSL development productivity while maintaining similar developer workflows to old tools such as Flex and Bison.

We believe the work done in this thesis is becoming especially important as programming skills become readily available within research groups. Whether this is the set of scientists which have coding skills as part of their primary skill set or with software engineers within inter-disciplinary research groups to help produce research production quality code.

Model-Specific DSALs

Our first approach towards a middle-out DSAL considered very domain-specific DSLs for use in a framework without domain-oriented interfaces. The approach we outlined allows for great separation of the front-end DSL creation and back-end weaver creation. These DSALs were interesting because they provided very strong members of the DSL category and allowed the use of standard DSL user experience features such as IDE quick fixes and compiler error reporting. While they do not offer any first-class aspect-oriented features the domain knowledge tied into the DSL syntax allows for effective first-class DSL support at a level GPAL based first-class support cannot provide. Transparent aspect-orientation is admissible because the aspects are written at the domain-level, and aspect-oriented rules are implicit within the DSL code. For example, Sugarscape and Kawasaki advice cannot be used in the same pointcut because it is semantically invalid, stopping the need for checking of GPAL implementation.

The DSALs produced with this method have a high initial cost and a small target market. As such we believe their use lies in projects where external, programming novices would like to inspect models. An example of this would be the operational analysis of supply chain simulations during logistics research projects.

An interesting change to this approach would be making the population of the DSJP runtime use an internal DSL as is done in Chapter 5. The population API of this DSL is designed to match a direct mapping to the semantic model of the DSL rather than a fluid experience for a user to write or understand pointcut advice. While this reduced the complexity of creating a code generator, it means that the direct use of the framework and debugging of aspects is difficult.

Middleware Specific DSALs

Our second approach towards a middle-out DSAL consists of creating a core DSAL which can be used directly or as an API for creating more specialised DSALs for specific problem sets.

These internal DSLs are a weak member of the category of DSL, as some don't even consider fluent interfaces to be DSLs. Yet their use dramatically improves the experience of populating aspects compared to the semantic-model structured API from Chapter 4. This type of DSL does not take long to make and offers substantial benefit in fluidity.

The core DSAL takes substantial time to create and should be complete enough to create all base pointcut and advice required by the potential extension DSALs. Subsequent extension DSALs, on the other hand, are fast to create and can package domain-knowledge in either internal or external DSLs. The advice of the aspects in our implementation is plain Java with the potential to provide domain-specific context using wrapper classes around the JP passed to advice.

This approach is especially useful for projects where end-users are competent programmers, but the implementation of aspect-oriented code is still desired. This could be because one developer is more familiar with the implementation of a model or models will be re-visited after implementation details are forgotten. Furthermore, this approach is useful across-projects because a core designed for an extensive framework such as JADE can be used across many models and many projects.

The significant downside of this approach is being constrained to the context exposed by the middleware, for example in this case only data exposed by the ACL message is available within advice. To expand past this DSAL composition would need to be considered, which has not been considered throughout this thesis. The semantic model approach would be suitable for this for small-scale composition of DSALs although for larger sets of DSALs a composition framework may become the appropriate choice.

It would also be interesting to try external DSL extensions to the core, the generator for these would potentially be more difficult to write because the population API is suited to fluent input rather than a direct translation of the semantic model as in Chapter 4.

As this approach has high start-up costs, but then the core may be used across many projects. There is good potential for moving towards the dependency injection and code translation approaches where it would improve performance for many users.

7.3 Future Research

To further our research, we would like to forward our research into DSALs where it can be directly applied in relevant application areas, which will then bring new opportunities for practically important research through the confluence of ideas.

Although we do not touch on them through this thesis, we are interested in exploring creating higher-level internal DSALs using languages such as Groovy and Scala. These languages provide far more options for dynamic and domain-specific internal DSL creation than in Java yet maintain compatibility with the JVM. An approach using this type of language as a DSL base could allow what are classically external DSL features to be brought into Java projects.

Projectional editing is also interesting for the aspect-oriented space because we could change what the programmer views based upon their intention as a model implementation developer, DSAL developer or DSAL user. This could be used to provide first-class support for developing a DSAL by separating DSAL implementation abstract syntax tree components and core model components, allowing different visualisations depending on the developer. Using current approaches this would require a high development investment to see a proper implementation for using DSALs alongside ABMs.

The next step for the aspect-oriented implementation of our DSALs is exploring the effects of foregoing obliviousness in implementation to allow the use of dependency injection, annotations or projectional editing for providing aspects. Hadas and Lorenz (2017) did this with their addition of annotations for creating first-class aspect-oriented support for DSALs although we are more interested in using this for increasing the experience of building the DSJP representation and performance at

runtime. The desire to design code with aspects in mind comes from as far back as Murphy et al. (2001) where it is noted exposing JPs is hard because of the lack of planning when writing the initial program.

This moves away from implementing our bottom layer with a pure aspect-oriented solution, and does require the scatter and tangle of JP locations. Yet this allows fast implementation of DSJP which will be high-performance compared to GPAL interpreted actions. Most of the scatter and tangle will still be modularised into aspects, so this approach will still fulfil its role as a DSAL. We have decided not to follow the code transforming method of implementation because the process is highly coupled to the base code and requires significant work to implement over arbitrary sets of large General-Purpose Language (GPL) based projects. This approach moves from aspect-oriented implementation of the DSAL to physical instrumentation of the base programs which could be done during or after model development to a middle layer specification. This would be infeasible for GPAL aspect-oriented programming although is a suitable fit for domain-specific actions as they will be fewer DSJP in a program compared to GPJP by orders of magnitude. Instrumentation further reduces the amount of interpretation required to convert GPJP into DSJP with checks by piggybacking off the base codes program flow. This means that a bottom layer approach would provide an instrumented scaffolding suitable to implement a core DSAL and any extension DSALs from this. The scaffolding could be either togglable through use of variables or in the case of middleware targeted DSALs placed into swappable versions of the middleware libraries.

For middleware targeting DSALs using dependency injection within the middleware code could be especially useful because the end-users will not have to deal with the scatter and tangle, and performance bonuses would benefit many users across many re-uses of the core and extension DSALs. This could be used

to solve the problem of providing suitable facilities for inspecting, verifying and validating models which has been flagged by North et al. (2013). For model-specific DSALs, injection approaches allow problematic DSJP to be implemented with high performance, but the low re-use potential and volatile target makes the extra development effort less worthwhile. This approach is not suitable during especially volatile development stages as arbitrary changes to the model may inadvertently change DSAL semantics.

In closing, we believe that AOP, DSLs and more specifically DSALs will thrive as multi-paradigm languages and development environments progress. As we reduce our dependence on scattered code within object-oriented projects and improve our knowledge of appropriate development techniques, DSALs have the potential to become an invaluable domain-specific programming practice.

Bibliography

Allan, C., P. Avgustinov, A. S. Christensen, B. Dufour, C. Goard, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, J. Tibble, and C. Verbrugge (2005). Abc the aspectbench compiler for aspectj a workbench for aspect-oriented programming language and compilers research. In *Companion to the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, New York, NY, USA, pp. 88–89. ACM.

Allan, R. J. (2010). Survey of agent based modelling and simulation tools. Technical report, Science & Technology Facilities Council.

Amiguet, M., A. Nagy, and J.-A. Baez-Barranco (2004). Towards an aspect-oriented approach of multi-agent programming. In *MOCA'04: 3rd Workshop on Modelling of Objects Components and Agents*, pp. 131–148.

Appleget, J., R. Burks, and M. Jaye (2014). A demonstration of abm validation techniques by applying docking to the epstein civil violence model. *The Journal of Defense Modeling and Simulation 11*(4), 403–411.

Bagge, A. H. and K. T. Kalleberg (2006). Dsal = library+notation: program transformation for domain-specific aspect languages. In *Proceedings of the Domain-Specific Aspect Languages Workshop*, pp. 1–8.

Balaghan, P., K. A. Hawick, D. Chalupa, and C. Maddra (2017). Dominating set algorithm implementation in graph databases. Technical Report CSI-0014, Computer Science, University of Hull, Cottingham Road, Hull, HU6 7RX.

Bandini, S., S. Manzoni, and G. Vizzari (2009). Agent based modeling and simulation: an informatics perspective. *Journal of Artificial Societies and Social Simulation 12*(4), 4.

Bellifemine, F. (2004). Using jade. In *2004 IEEE International Conference on Systems, Man and Cybernetics Tutorial on the Java Agent Development Framework (JADE)*, Volume 7.

Bellifemine, F., G. Caire, and A. P. Rimassa (2003). Jade - a white paper. *exp in search of innovation (Telecom Italia Lab) 3*(3), 6–19.

Bellifemine, F., A. Poggi, and G. Rimassa (2001). Jade: a fipa2000 compliant agent development environment. In *Proceedings of the Fifth International Conference on Autonomous Agents*, AGENTS '01, New York, NY, USA, pp. 216–217. ACM.

Bellifemine, F. L., G. Caire, and D. Greenwood (2007). *Developing multi-agent systems with JADE*, Volume 7. John Wiley & Sons.

Bentley, J. (1986). Programming pearls: little languages. *Commun. ACM 29*(8), 711–721.

Bentley, J. L. and B. W. Kernighan (1986). Grap - a language for typesetting graphs. *Commun. ACM 29*(8), 782–792.

Bergenti, F., E. Iotti, S. Monica, and A. Poggi (2017). Agent-oriented model-driven development for jade with the jadel programming language. *Computer Languages, Systems & Structures 50*(Supplement C), 142 – 158.

Bergmans, L. (2004). Principles and design rationale of composition filters. In *Aspect-oriented Software Development* (1st ed.)., Chapter 5, pp. 63–96. Addison Wesley.

Bolte, J. P., D. W. Hulse, S. V. Gregory, and C. Smith (2007). Modeling biocomplexity–actors, landscapes and alternative futures. *Environmental Modelling & Software 22*(5), 570–579.

Bonabeau, E. (2002). Agent-based modeling: methods and techniques for simulating human systems. *Proceedings of the National Academy of Sciences 99*(suppl 3), 7280–7287.

Braubach, L., A. Pokahr, and W. Lamersdorf (2005). Jadex: a bdi-agent system combining middleware and reasoning. In *Software Agent-Based Applications, Platforms and Development Kits*, Basel, pp. 143–168. Birkhäuser Basel.

Bruneton, E., R. Lenglet, and T. Coupaye (2002). Asm: a code manipulation tool to implement adaptable systems. In *Adaptable and extensible component systems*.

Campagne, F. and F. Campagne (2014). *The MPS Language Workbench, Vol. 1* (1st ed.). USA: CreateSpace Independent Publishing Platform.

Cardoso, H. L. (2015). Sajas: enabling jade-based simulations. In *Transactions on Computational Collective Intelligence XX*, pp. 158–178. Springer International Publishing.

Cegielski, W. H. and J. D. Rogers (2016). Rethinking the role of agent-based modeling in archaeology. *Journal of Anthropological Archaeology 41*, 283 – 298.

Chibani, M., B. Belattar, and A. Bourouis (2013). Toward an aspect-oriented simulation. *International Journal of New Computer Architectures and Their Applications 3*(1), 1–11.

Cicirelli, F., A. Furfaro, A. Giordano, and L. Nigro (2009). Distributed simulation of repast models over hla/actors. In *2009 13th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications*, pp. 184–191.

Clark, T., P. Sammut, and J. S. Willans (2015). Applied metamodelling: a foundation for language driven development (third edition). *CoRR abs/1505.00149*.

Coakley, S., M. Gheorghe, M. Holcombe, S. Chin, D. Worth, and C. Greenough (2012). Exploitation of high performance computing in the flame agent-based simulation framework. In *2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems*, pp. 538–545. IEEE.

Cohen, T. and J. Y. Gil (2004). Aspectj2ee = aop + j2ee. In M. Odersky (Ed.), *Proceedings of ECOOP 2004 – Object-Oriented Programming*, Berlin, Heidelberg, pp. 221–245. Springer Berlin Heidelberg.

Coleman, L. and P. Kay (1981). Prototype semantics: the english word lie. *Language 57*(1), 26–44.

Collier, N. and M. North (2013). Parallel agent-based simulation with repast for high performance computing. *SIMULATION 89*(10), 1215–1235.

Colombo, C., G. J. Pace, and G. Schneider (2009). Larva — safer monitoring of real-time java programs (tool paper). In *Proceedings of the 2009 Seventh IEEE International Conference on Software Engineering and Formal Methods*, SEFM '09, Washington, DC, USA, pp. 33–37. IEEE Computer Society.

Constantinides, C., T. Skotiniotis, and M. Stoerzer (2004). Aop considered harmful. In *Proceedings of European Interactive Workshop on Aspects in Software (EIWAS)*.

Cordasco, G., R. De Chiara, A. Mancuso, D. Mazzeo, V. Scarano, and C. Spagnuolo (2012). A framework for distributing agent-based simulations. In *Euro-Par 2011: Parallel Processing Workshops*, Berlin, Heidelberg, pp. 460–470. Springer Berlin Heidelberg.

Cortese, E., F. Quarta, G. Vitaglione, and P. Vrba (2003). Scalability and performance of the jade message transport system. analysis of suitability for holonic manufacturing systems. *exp in search of innovation (Telecom Italia Lab) 3*(3), 52–64.

Cosgrove, J., J. Butler, K. Alden, M. Read, V. Kumar, L. Cucurull-Sanchez, J. Timmis, and M. Coles (2015). Agent-based modeling in systems pharmacology. *CPT: Pharmacometrics & Systems Pharmacology 4*(11), 615–629.

Czarnecki, K. and U. W. Eisenecker (2000). *Generative Programming: Methods, Tools, and Applications*. New York, NY, USA: ACM Press/Addison-Wesley Publishing.

Czarnecki, K., U. W. Eisenecker, R. Glück, D. Vandevoorde, and T. L. Veldhuizen (2000). Generative programming and active libraries. In *Selected Papers from the International Seminar on Generic Programming*, London, UK, UK, pp. 25–39. Springer-Verlag.

de Borger, W., B. Lagaisse, and W. Joosen (2012). A domain specific aspect language for run-time inspection. In *Proceedings of the Seventh Workshop on Domain-Specific Aspect Languages*, DSAL '12, New York, NY, USA, pp. 33–36. ACM.

de S. Braga, D., F. O. M. Alves, F. B. de L. Neto, and L. C. de S. Menezes (2012). An aspect-oriented domain-specific language for modeling multi-agent systems in social simulations. In *Intelligent Data Engineering and Automated Learning - IDEAL 2012*, Berlin, Heidelberg, pp. 578–585. Springer Berlin Heidelberg.

Defense Advanced Research Projects Agency (2013). Probabilistic programming for advancing machine learning (ppaml) darpa-baa-13-31. *Combined Synopsis/Solicitation Notice*.

Dethlefs, N. and K. Hawick (2017). Define: a fluent interface dsl for deep learning applications. In *Proceedings of the 2nd International Workshop on Real World Domain Specific Languages*, RWDSL17, New York, NY, USA, pp. 3:1–3:10. ACM.

Dijkstra, E. W. (1968). Letters to the editor: go to statement considered harmful. *Commun. ACM 11*(3), 147–148.

Dinkelaker, T., M. Monperrus, and M. Mezini (2009). Untangling crosscutting concerns in domain-specific languages with domain-specific join points. In *Proceedings of the 4th workshop on Domain-specific aspect languages*, pp. 1–6. ACM.

Dmitriev, S. (2004, November). Language oriented programming: the next programming paradigm. *OnBoard Electronic Monthly Magazine*.

Elrad, T., M. Aksit, G. Kiczales, K. Lieberherr, and H. Ossher (2001, Nov). Discussing aspects of aspect-oriented programming. *Commun. ACM 44*(10), 33–38.

Epstein, J. M. and R. Axtell (1996). *Growing artificial societies: social science from the bottom up*. Brookings Institution Press.

Fabry, J., T. Dinkelaker, J. Noyé, and E. Tanter (2015). A taxonomy of domain-specific aspect languages. *ACM Comput. Surv. 47*(3), 40:1–40:44.

Ferber, J. and O. Gutknecht (1998). A meta-model for the analysis and design of organizations in multi-agent systems. In *Proceedings International Conference on Multi Agent Systems*, pp. 128–135.

Filman, R. E. and D. P. Friedman (2000). Aspect-oriented programming is quantification and obliviousness. Technical report, Research Institute for Advanced Computer Science.

Fowler, M. (2010). *Domain Specific Languages* (1st ed.). Addison-Wesley Professional.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides (1995). *Design Patterns: elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Geeraerts, D. (2016). Prospects and problems of prototype theory. *Diacronia 2016*(4), 1–16.

Greenfield, J. and K. Short (2003). Software factories: assembling applications with patterns, models, frameworks and tools. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '03, New York, NY, USA, pp. 16–27. ACM.

Greenwood, D. (2004). Fipa specifications. In *2004 IEEE International Conference on Systems, Man and Cybernetics Tutorial on the Java Agent Development Framework (JADE)*, Volume 7.

Gulyás, L., T. Kozsik, and J. Corliss (1999). The multi-agent modelling language and the model design interface. *The Journal of Artificial Societies and Social Simulation (JASSS) 2*(3).

Gutknecht, O. and J. Ferber (2000). The madkit agent platform architecture. In *Workshop on Infrastructure for Scalable Multi-Agent Systems at the International Conference on Autonomous Agents*, pp. 48–55. Springer.

Hadas, A. and D. H. Lorenz (2015). Demanding first-class equality for domain specific aspect languages. In *Companion Proceedings of the 14th International Conference on Modularity*, MODULARITY Companion 2015, New York, NY, USA, pp. 35–38. ACM.

Hadas, A. and D. H. Lorenz (2016). Toward disposable domain-specific aspect languages. In *Companion Proceedings of the 15th International Conference on Modularity*, MODULARITY Companion 2016, New York, NY, USA, pp. 83–85. ACM.

Hadas, A. and D. H. Lorenz (2017). Language oriented modularity: from theory to practice. *The Art, Science, and Engineering of Programming 1*(2).

Harbulot, B. and J. R. Gurd (2006). A join point for loops in aspectj. In *Proceedings of the 5th international conference on Aspect-oriented software development*, pp. 63–74. ACM.

Hawick, K. A. (2011). Engineering domain-specific languages for computational simulations of complex systems. In *Proceedings of the Int. Conf. on Software Engineering and Applications (SEA2011)*, Dallas, USA, pp. 222–229. IASTED.

Hawick, K. A. (2012a). Engineering internal domain-specific language software for lattice-based simulations. In *Proceedings of the Int. Conf. on Software Engineering and Applications*, Las Vegas, USA, pp. 314–321. IASTED.

Hawick, K. A. (2012b). Transients in a forest-fire simulation model with varying combustion neighbourhoods and watercourse firebreaks. In *Proceedings of the Int. Conf. on Engineering and Applied Science*, Colombo, Sri Lanka, pp. 202–208. IASTED.

Hawick, K. A. (2013). Fluent interfaces and domain-specific languages for graph generation and network analysis calculations. In *Proceedings of the Int. Conf. on Software Engineering (SE'13)*, Innsbruck, Austria, pp. 752–759. IASTED.

Heath, B., R. Hill, and F. Ciarallo (2009). A survey of agent-based modeling practices (january 1998 to july 2008). *Journal of Artificial Societies and Social Simulation 12*(4).

Hirsch, B., T. Konnerth, and A. Heßler (2009). Merging agents and services — the jiac agent platform. In A. El Fallah Seghrouchni, J. Dix, M. Dastani, and R. H. Bordini (Eds.), *Multi-Agent Programming: Languages, Tools and Applications* (1st ed.)., Chapter 5, pp. 159–185. Boston, MA: Springer US.

Hofstadter, D. and E. Sander (2013). *Surfaces and Essence: Analogy as the Fuel and Fire of Thinking*. Basic Books.

Holcombe, M., S. Coakley, and R. Smallwood (2006). A general framework for agent-based modelling of complex systems. In *Proceedings of the 2006 European Conference on Complex Systems*.

Holland, J. H. (2014). *Complexity: A Very Short Introduction*. Oxford University Press.

Howden, N., R. Ronnquist, A. Hodgson, and A. Lucas (2001). Jack intelligent agents - summary of an agent infrastructure. In *Proceedings of the 5th ACM International Conference on Autonomous Agents*.

Husselmann, A. V. (2014). *Data-parallel structural optimisation in agent-based modelling*. Ph. D. thesis, Massey University.

Janssen, M. A. (2012). *Introduction to Agent-Based Modeling*. www.openabm.org.

Javed, O., Y. Zheng, A. Rosà, H. Sun, and W. Binder (2016). Extended code coverage for aspectj-based runtime verification tools. In *Runtime Verification*, Cham, pp. 219–234. Springer International Publishing.

Jin, D., P. O. Meredith, C. Lee, and G. Roşu (2012). Javamop: efficient parametric runtime monitoring framework. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, Piscataway, NJ, USA, pp. 1427–1430. IEEE Press.

Kawasaki, K. (1966). Diffusion constants near the critical point for time dependent ising models. I. *Phys. Rev. 145*(1), 224–230.

Kernighan, B. W. and R. Pike (1983). *The UNIX Programming Environment*. Prentice Hall Professional Technical Reference.

Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin (1997). Aspect-oriented programming. In *ECOOP'97 — Object-Oriented Programming*, Berlin, Heidelberg, pp. 220–242. Springer Berlin Heidelberg.

Klügl, F., R. Herrler, and C. Oechslein (2003). From simulated to real environments: how to use sesam for software development. In *Multiagent System Technologies*, Berlin, Heidelberg, pp. 13–24. Springer Berlin Heidelberg.

Knuth, D. E. (1984). Literate programming. *The Computer Journal 27*(2), 97–111.

Kojarski, S. and D. H. Lorenz (2005). Pluggable aop: designing aspect mechanisms for third-party composition. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, New York, NY, USA, pp. 247–263. ACM.

Kojarski, S. and D. H. Lorenz (2007). Awesome: an aspect co-weaving system for composing multiple aspect-oriented extensions. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, New York, NY, USA, pp. 515–534. ACM.

Kumar, A., A. Kumar, and M. Iyyappan (2016). Applying separation of concern for developing softwares using aspect oriented programming concepts. *Procedia Computer Science 85*, 906 – 914. International Conference on Computational Modelling and Security (CMS 2016).

Labov, W. (1973). The boundaries of words and their meanings. In Bailey and R. W. Shuy (Eds.), *New Ways of analyzing variation in English*, pp. 340–373. Washington, D.C.: Georgetown University Press.

Lakoff, G. (1987). *Women, fire, and dangerous things: what categories reveal about the mind*. University of Chicago Press.

Law, A. M. (2008). How to build valid and credible simulation models. In *Proceedings of the 40th Conference on Winter Simulation*, WSC '08, pp. 39–47. Winter Simulation Conference.

Lippert, M. and C. V. Lopes (2000). A study on exception detection and handling using aspect-oriented programming. In *Proceedings of the 22nd international conference on Software engineering*, pp. 418–427. ACM.

Lorenz, D. H. and B. Rosenan (2011). Cedalion: a language for language oriented programming. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, New York, NY, USA, pp. 733–752. ACM.

Luke, S., C. Cioffi-Revilla, L. Panait, K. Sullivan, and G. Balan (2005). Mason: a multiagent simulation environment. *Simulation 81*(7), 517–527.

Macal, C. M. (2016). Everything you need to know about agent-based modelling and simulation. *Journal of Simulation 10*(2), 144–156.

Macal, C. M. and M. J. North (2006). Tutorial on agent-based modeling and simulation part 2: how to model with agents. In *Proceedings of the 2006 Winter Simulation Conference*, pp. 73–83.

Maddra, C. A. and K. A. Hawick (2016). Domain modelling and language issues for family history and near-tree graph data applications. In *Proceedings of the 14th Int. Conf. Software Engineering Research and Practice*, Las Vegas, USA, pp. 10–16. WorldComp: CSREA Press.

Marshall, B. D. L. and S. Galea (2015). Formalizing the role of agent-based modeling in causal inference and epidemiology. *American Journal of Epidemiology 181*(2), 92.

Martin, J. (1985). *Fourth Generation Languages: Principles* (1st ed.). Prentice Hall.

Mattiussi, C., P. Dürr, D. Marbach, and D. Floreano (2011). Beyond graphs: a new synthesis. *Journal of Computational Science 2*(2), 165–177.

McLane, A. J., C. Semeniuk, G. J. McDermid, and D. J. Marceau (2011). The role of agent-based models in wildlife ecology and management. *Ecological Modelling 222*(8), 1544 – 1556.

Mehner, K. and A. Rashid (2002). Towards a standard interface for runtime inspection in aop environments. In *Workshop on Tools for Aspect-Oriented Software Development at OOPSLA*.

Mernik, M., J. Heering, and A. M. Sloane (2005). When and how to develop domain-specific languages. *ACM Comput. Surv. 37*(4), 316–344.

Michaelson, G. (2016). Are there domain specific languages? In *Proceedings of the 1st International Workshop on Real World Domain Specific Languages*, pp. 1. ACM.

Minar, N., R. Burkhart, C. Langton, and M. Askenazi (1996). The swarm simulation system: a toolkit for building multi-agent simulations. Technical report, Santa Fe Institute.

Mitre (2014). *MITRE Systems Engineering Guide*, Chapter Verification and Validation of Simulation Models, pp. 461–469. Mitre.

Morgan, M. S. and M. Morrison (1999). *Models as mediators: Perspectives on natural and social science*, Volume 52. Cambridge University Press.

Murphy, G. C., R. J. Walker, E. L. Baniassad, M. P. Robillard, A. Lai, and M. A. Kersten (2001). Does aspect-oriented programming work? *Commun. ACM 44*(10), 75–77.

Niazi, M. A., A. Hussain, and M. Kolberg (2009). Verification & validation of agent based simulations using the vomas (virtual overlay multi-agent system) approach. In *Proceedings of the Second Multi-Agent Logics, Languages, and Organisations Federated Workshops, Turin, Italy, September 7-10, 2009*.

North, M. J., N. T. Collier, J. Ozik, E. R. Tatara, C. M. Macal, M. Bragen, and P. Sydelko (2013). Complex adaptive systems modeling with repast simphony. *Complex Adaptive Systems Modeling 1*(1), 3.

Nusayr, A. and J. Cook (2009). Aop for the domain of runtime monitoring: breaking out of the code-based model. In *Proceedings of the 4th workshop on Domain-specific aspect languages*, pp. 7–10. ACM.

Nwana, H. S., D. T. Ndumu, L. C. Lee, and J. C. Collis (1999). Zeus: a toolkit for building distributed multiagent systems. *Applied Artificial Intelligence 13*(1-2), 129–185.

Parnas, D. L. (1976). On the design and development of program families. *IEEE Transactions on Software Engineering 2*(1), 1–9.

Parnas, D. L. (1979). Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering 5*(2), 128–138.

Parr, T. (2013). *The Definitive ANTLR 4 Reference*. Oreilly and Associate Series. Pragmatic Bookshelf.

Poslad, S. (2007). Specifying protocols for multi-agent systems interaction. *ACM Transactions on Autonomous and Adaptive Systems (TAAS) 2*(4), 15.

Poslad, S., P. Buckle, and R. Hadingham (2000). The fipa-os agent platform: open source for open standards. In *Proceedings of the 5th international conference and exhibition on the practical application of intelligent agents and multi-agents.*

Poulin, J. S. (1995). Populating software repositories: incentives and domain-specific software. *Journal of Systems and Software 30*(3), 187 – 199.

Preez, V. D., B. Pearce, K. A. Hawick, and T. H. McMullen (2012). Software engineering a family of complex systems simulation model apps on android tablets. In *Proceedings of the Int. Conf. on Software Engineering Research and Practice (SERP'12)*, Las Vegas, USA, pp. 215–221. CSREA.

Richmond, P., D. Walker, S. Coakley, and D. Romano (2010). High performance cellular level agent-based simulation with flame for the gpu. *Briefings in bioinformatics 11*(3), 334–347.

Rosch, E. (1978). Principles of categorization. In E. Rosch and B. Lloyd (Eds.), *Cognition and Categorization*. Lawrence Elbaum Associates.

Rosenan, B. (2010). Designing language-oriented programming languages. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '10, New York, NY, USA, pp. 207–208. ACM.

Russell, S. J. and P. Norvig (2003). *Artificial Intelligence: A Modern Approach* (2nd ed.). Pearson Education.

Sammet, J. E. (1969). *Programming Languages: History and Fundamentals*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc.

Schlesinger, S., R. E. Crosbie, R. E. Gagné, G. S. Innis, C. lalwani, J. Loch, R. J. Sylvester, R. D. Wright, N. Kheir, and D. Bartos (1979). Terminology for model credibility. *SIMULATION 32*(3), 103–104.

Schützelhofer, W. (2016). Jcypher, issue closed: database-design & mapping. In *GraphConnect Europe*, London, England. Neo4j.

Shende, S., A. D. Malony, J. Cuny, P. Beckman, S. Karmesin, and K. Lindlan (1998). Portable profiling and tracing for parallel, scientific applications using c++. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, SPDT '98, New York, NY, USA, pp. 134–145. ACM.

Simon, H. A. (1996). *The sciences of the artificial*. MIT Press.

Simonyi, C., M. Christerson, and S. Clifford (2006). Intentional software. *SIGPLAN Not. 41*(10), 451–464.

Soklabi, A., M. Bahaj, and I. Cherti (2013). Jiac systems and jade agents communication. *the International Journal of Engineering and Technology (IJET) 5*(2), 1976–1984.

Soule, P. (2010). *Autonomics Development: A Domain-Specific Aspect Language Approach*. Springer.

Sousan, W., V. Winter, M. Zand, and H. Siy (2007). Ertsal: a prototype of a domain-specific aspect language for analysis of embedded real-time systems. In *Proceedings of the 2nd Workshop on Domain Specific Aspect Languages*, DSAL '07, New York, NY, USA. ACM.

Spies, T. A., E. White, A. Ager, J. D. Kline, J. P. Bolte, E. K. Platt, K. A. Olsen, R. J. Pabst, A. M. Barros, J. D. Bailey, S. Charnley, A. T. Morzillo, J. Koch, M. M. Steen-Adams, P. H. Singleton, J. Sulzman, C. Schwartz, and B. Csuti (2017). Using an agent-based model to examine forest management outcomes in a fire-prone landscape in oregon, usa. *Ecology and Society 22*(1).

Sprinkle, J., D. Spinellis, J. Tolvanen, and M. Mernik (2009). Guest editors' introduction: what kinds of nails need a domain-specific hammer? *IEEE Software 26*, 15–18.

Stanislaw, H. (1986). Tests of computer simulation validity. *Simulation & Games 17*(2), 173–191.

Thibault, S. A., R. Marlet, and C. Consel (1999). Domain-specific languages: from design to implementation application to video device drivers generation. *IEEE Transactions on Software Engineering 25*(3), 363–377.

Tolvanen, J.-P. (2016). Metaedit+ for collaborative language engineering and language use (tool demo). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2016, New York, NY, USA, pp. 41–45. ACM.

Tracy, M., M. Cerdá, and K. M. Keyes (2018). Agent-based modeling in public health: current applications and future directions. *Annual Review of Public Health 39*(1), 77–94.

Ubayashi, N., H. Masuhara, and T. Tamai (2004). An aop implementation framework for extending join point models. In *Proceedings of ECOOP'2004 Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'04)*, pp. 71–81.

Ubayashi, N. and T. Tamai (2001). Separation of concerns in mobile agent applications. In *Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, REFLECTION '01, London, UK, UK, pp. 89–109. Springer-Verlag.

van Deursen, A., P. Klint, and J. Visser (2000). Domain-specific languages: an annotated bibliography. *SIGPLAN Not. 35*(6), 26–36.

Veldhuizen, T. and D. Gannon (1998). Active libraries: rethinking the roles of compilers and libraries. In *Proceedings of the 1998 SIAM Workshop: Object Oriented Methods for Interoperable Scientific and Engineering Computing*.

Völter, M. (2009). Best practices for dsls and model-driven development. *Journal of Object Technology 8*(6), 79–102.

Völter, M. (2010). Architecture as language. *IEEE Software 27*(2), 56–64.

Völter, M., D. Ratiu, B. Schaetz, and B. Kolb (2012). Mbeddr: an extensible c-based programming language and ide for embedded systems. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, New York, NY, USA, pp. 121–140. ACM.

Völter, M., J. Siegmund, T. Berger, and B. Kolb (2014). Towards user-friendly projectional editors. In *Software Language Engineering*, Cham, pp. 41–61. Springer International Publishing.

Walker, D. (2016). Confluences in programming languages research (keynote). *SIGPLAN Not. 51*(1), 4–4.

Ward, M. P. (1995). Language oriented programming. *Software - Concepts and Tools 15*, 147–161.

Wierzbicka, A. (1985). *Lexicography and Conceptual Analysis*. Karoma Pub.

Wile, D. (2001). Supporting the dsl spectrum. *Journal of Computing and Information Technology 9*(4), 263–287.

Wilensky, U. (1999). Netlogo. http://ccl.northwestern.edu/netlogo/, Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL.

Winsberg, E. (1999). Sanctioning models: the epistemology of simulation. *Science in Context 12*(2), 275–292.

Wittgenstein, L. (1953). *Philosophical Investigations = Philosophische Untersuchungen*. Macmillan.