Simulation Design: Trans-Paradigm Best-Practice from Software Engineering*

Stuart Rossiter (University of Southampton)

2nd February 2015

Abstract

There are growing initiatives to apply software engineering (SE) best-practice to computational science, which includes simulation. One area where the simulation literature appears to be particularly light is in the overall structural design of simulations, and what architectures and features are valuable for what reasons. (Part of the problem is that parts of this knowledge are abstracted away in simulation toolkits which are often not easily comparable, and have different conceptual aims.)

To address this, I outline three key software properties which embody SE best-practices, and then define an 'idealised' software architecture for simulation—what SE would call a reference architecture—which strongly exhibits them. I show that this is universal to all simulations (largely because modelling-paradigm-specific detail is encapsulated into a 'single black box' layer of functionality) but that simulation toolkits tend to differ in how they map to them; this relates to the aims of the toolkits, which I provide a useful categorisation of.

I show that, interestingly, there are several core features of this architecture that are not fully represented in *any* simulation toolkit that

*This is a post-print version of the forthcoming journal article with the following citation:

Rossiter, S. (Forthcoming). Simulation design: Trans-paradigm best-practice from software engineering. *Journal of Artificial Societies & Social Simulation (JASSS)* .

The definitive published version will be available open access online in its default HTML format (and this text will be updated with the authoritative location when available). This document represents a print-typeset version of the LaTeX original (before HTML conversion using the tools available on the JASSS site at `http://jasss.soc.surrey.ac.uk/admin/submit.html`). This may therefore be a more print-friendly version. (The JASSS site supports PDF downloads, but these are created directly from the HTML, and thus not typeset as this original.) *Because the published version uses a paragraph numbering scheme (with no page numbers), you should use that numbering scheme in any location specific citations.*

I am aware of. I present a library—JSIT—which provides some proof-of-concept implementations of them for Java-based toolkits. This library, and other ideas in the reference architecture, are put into practice on a published, multi-paradigm model of health and social care which uses the AnyLogic toolkit.

I conclude with some thoughts on *why* this area receives so little focus, how to take it forwards, and some of the related cultural issues.

Keywords: software-engineering, simulation-toolkits, reference-architecture, best-practice

1 Introduction

As much of science becomes more and more dependent on software, there has been increasing interest in promoting the development of scientific software which is reliable (well-tested), reusable, well-maintained (sustainable), and can be used in ways which provides open, reproducible research. Such initiatives have been championed by groups such as the UK's research-council-funded Software Sustainability Institute[1]; the Science Code Manifesto[2] and related authors (Stodden *et al.* 2010, 2013); and the Software Carpentry movement[3] (Wilson *et al.* 2014; Wilson 2014). By definition, these aims involve selectively applying software engineering (SE) best-practice ideas to the development of scientific software, including simulation.

If we restrict ourselves to *simulation* software, and to simulation *design* (ignoring aspects of development *process* and code access), there are three main strongly-related areas of best-practice which I would regard as universal to all simulation (precisely because they are universal to all software), and which are echoed in computational science best-practice papers (Sandve *et al.* 2013; Wilson *et al.* 2014), software engineering textbooks (Sommerville 2011), and practitioner best-practice handbooks (McConnell 2004); all backed by empirical research (Oram & Wilson 2010)[4]:

Automated Reproducibility. Being able to recreate any run of the software—for testing purposes and to check claims about its outputs—*in an automated way* (not just via manual recreation from documentation). This includes provenance (and perhaps automated recreation) of the

[1] See `http://www.software.ac.uk`.

[2] See `http://sciencecodemanifesto.org`.

[3] See `http://software-carpentry.org`.

[4] There are also a large number of references to empirical studies in the best-practice papers and books mentioned.

entire computational environment (since results can vary based on things like the versions of external libraries used).

Cohesive, Loosely-Coupled Design. A design separated into components with well-defined (cohesive) functions, and minimised dependencies on other components (loose-coupling). This massively aids the debugging, maintenance and reusability of the code. This often involves reusing recurring structural and behavioural forms that have been shown to help solve common design issues: SE calls such forms **design patterns** (Gamma *et al.* 1995; Buschmann 1996). Such forms help establish a shared software design vocabulary at a higher level of abstraction.

Testability. Being designed in a way that facilitates testing at different levels (e.g., single class, component or whole system) and, where possible, includes *automated* tests as part of the software deliverable. In particular, automated tests provide a bank of **regression tests** which can be continually re-run to check that changes have not caused bugs elsewhere (i.e., caused previously successful tests to fail). Such tests become the central driver of the development process in the increasingly-used Test-Driven Development (TDD) approach (Jeffries & Melnik 2007).

Software exhibiting these properties is highly **reusable** (given access to it) and its implementation will typically involve reuse of existing software where it exists. In the simulation domain, there are many toolkits which provide reusable, well-tested software for simulation development which include (a) templates for model elements relating to one or more modelling paradigms—such as agent-based modelling (ABM), discrete-event simulation (DES) or system dynamics (SD); and (b) supporting infrastructure code to create and run models, such as for visualisations, simulation control interfaces, and input/output handling.[5]

There has been some emerging work which tries to define new simulation frameworks and abstractions which better embody some of these principles; e.g., the modular architecture and best-practice of JAMES II (Himmelspach & Uhrmacher 2007; Uhrmacher 2012), or test and experiment specifications which are model-based (Djanatliev *et al.* 2011) or domain-language-based (Ewald & Uhrmacher 2014).[6]

[5] I use the term 'toolkit' generically to refer to all the existing platforms for simulation which, as well as 'platforms' and 'toolkits', can also be referred to as software libraries or frameworks. There are some technical distinctions between these terms, but they are not relevant for the purposes of this paper.

[6] The foundational theory of modelling and simulation in the DEVS literature (Zeigler

However, there appears to be virtually no discussion of these issues more generally for 'mainstream' simulation using widely-used toolkits such as, in the ABM case, NetLogo (Tisue & Wilensky 2004), Repast Simphony (North *et al.* 2013), MASON (Luke *et al.* 2005), or AnyLogic (Borshchev & Filippov 2004). In particular, there is nothing which allows simulation practitioners to understand how these ideas might be embodied in some best-practice simulation design, and to therefore have some frame to assess existing toolkits and make more informed decisions on their choice of simulation platform (and thus understand the strengths and weaknesses of their simulation software design with respect to this best-practice).[7]

If we restrict ourselves to the ABM domain for now, literature which does not really discuss this best-practice includes (a) textbooks (Gilbert & Troitzsch 2005; Grimm & Railsback 2005; Miller & Page 2007; Railsback & Grimm 2012); (b) toolkit comparisons (Railsback *et al.* 2006; Nikolai & Madey 2009; Allan 2010); (c) toolkit description papers (Borshchev & Filippov 2004; Tisue & Wilensky 2004; Luke *et al.* 2005; North *et al.* 2013); and (d) best-practice papers (Ropella *et al.* 2002; North & Macal 2014). Grimm & Railsback (2005, §8) and North & Macal (2014) get closest, mentioning such things as automated testing, design patterns and separation of model from visualisation. However, none of them really discuss how this relates to concrete design principles. Toolkit description papers discuss some features which embody some of these best-practices, but do not explicitly make the connections or discuss alternatives. There are also a few recent papers focusing on automated testing and TDD (Gürcan *et al.* 2013; Collier & Ozik 2013), but not on how to architect simulations for testability.[8]

Outside of ABM (but from a less rigorous exploration), it appears to be a similar situation. The practical operational research literature (which includes some of the ABM references above) is reasonably interested in simulation software development but, again, typically in *process*, and in paradigm-specific *conceptual* design.

---

et al. 2000)—which extends to ABM (Müller 2009)—sets up some very useful frames and terminology to talk about these issues more formally for simulation (which I unfortunately do not have space to touch on here), but does not directly address them.

[7] This would also help practitioners understand the contribution of the emerging ideas just mentioned.

[8] There is a reasonable amount of focus across these references on the development *process*, but that is not the same thing. There is also a growing literature on model comparison and reproduction (Rouchier *et al.* 2008), but the 'reproducibility' there (of model results from separate implementations of the same model or potentially equivalent models) is in a different sense to the reproducibility here (of runs for the same model implementation).

1.1 Aims

This paper aims to do several things: (1) draw attention to this gap and provide (indirectly) a rough guide to relevant SE and simulation literature; (2) provide a frame which captures some best-practice simulation architecture and can be used to generically understand how simulations are/can be constructed (section 2); (3) use this frame for a better understanding of simulation toolkits and, in particular, highlight some *consistent* omissions (section 3); (4) develop a software library which begins to address the omissions for a broad family of toolkits (section 4); and (5) provide a case study where the frame and library are used on a published, multi-paradigm simulation model using the AnyLogic toolkit (section 5), where AnyLogic was chosen for particular reasons.

To make these ideas relevant for all (dynamic) simulations, the abstraction of the real-world system is cleanly separated from the rest of the simulation (see section 2) and not 'opened up'; we shall see that there is still lots of detail to work with in the remainder.

In the final conclusions (section 6), I draw back to consider *why* this knowledge gap seems to exist, and some of the more general cultural barriers in applying this SE-oriented thinking. I also discuss where this work fits with respect to the emerging ideas mentioned earlier. Although I believe that the ideas here are largely universal, I focus mostly on ABM modelling in this paper, so the conclusions also reflect briefly on this broader applicability.

2 A Best-Practice Paradigm-Independent Frame

There are some recurring, generic SE design patterns which are known to strongly assist in creating software which exhibits the properties in section 1; these still need applying to a simulation domain using domain understanding, and many interpretations are possible. Equally, there are also some *simulation-specific* features which I believe are also important for simulations to exhibit the best-practice properties, and are less directly related to generic SE design patterns.

I develop these ideas to create a kind of 'idealised' software architecture for simulations. In SE terms, this is known as a **reference architecture**: "[they] capture important features of system architectures in a domain. Essentially, they include everything that might be in an application architecture [...] The main purpose [...] is to evaluate and compare design proposals, and to educate people about architectural characteristics in that domain" (Sommerville 2011, §6.4). 'Software architecture' is a slightly contentious term, but I like Fowler's take (Fowler *et al.* 2003, p.1):

> "'Architecture' is a term that lots of people try to define, with little agreement. There are two common elements: One is the highest-level breakdown of a system into its parts; the other, [design] decisions that are hard to change."

The second part of Fowler's quote is important here: modellers should think about these issues up-front, because it is difficult to retrospectively re-design a model to match this architecture and, perhaps more relevantly, the choice of simulation toolkit will affect how possible this is and how much work is involved. Toolkits typically also present themselves conceptually in a particular way, dependent on their design goals, which will cause differing degrees of 'dissonance' with the reference architecture even if, underneath, the toolkit code is directly mappable to it.[9]

The reference architecture is cumulatively built-up in the sub-sections which follow (and in figures 1–3). We will see concrete examples of the architecture in practice in sections 3 and 5.

2.1 Layered Functionality

A key SE best-practice is to separate **domain model** code—the code which represents the concepts of the domain which the software relates to (e.g., banking, aircraft-control)—from non-domain-model code as part of a **layered architecture** (Fowler *et al.* 2003; Evans 2004): functionality is split into layers, where each layer has a well-defined role and is dependent only on layers 'below' it.[10] This directly forms a cohesive, loosely-coupled design, but also aids testability (in more easily being able to compose parts of the application needed for different levels of testing) and automated reproducibility (in terms of isolating the core part—the domain model—whose behaviour needs to be reproduced).

In the case of simulation, the domain model is the abstraction of the relevant real-world system (simuland) as some set of behavioural entities acting and interacting over simulated time, each with their own state. This thus *includes* the overall handling of space and time. (Time, for example, is typically handled via fixed time steps or a discrete-event schedule.) *Modelling paradigms are encapsulated in this layer:* they define what these entities

[9] One of the main points of a reference architecture is normally that *any* actual implementation of the software system in question could be rewritten to conform to it (possibly with some pieces missing) whilst retaining the same functionality. I strongly believe this is true here, but I make no serious attempt to 'prove' it; I hope that the presentation intuitively makes the idea at least likely to be true, especially for readers with stronger programming backgrounds.

[10] In more strict layered architectures, a layer only uses the services of the layer *directly* below it. This is not true here.

can or should be, how they may act or interact, and how space and time are represented.

How this architecture works is best understood by describing the full set of layers from the bottom upwards, summarised in figure 1. (I do not always explicitly state it, but the descriptions should make clear how each layer will only need the services of layers beneath it.)

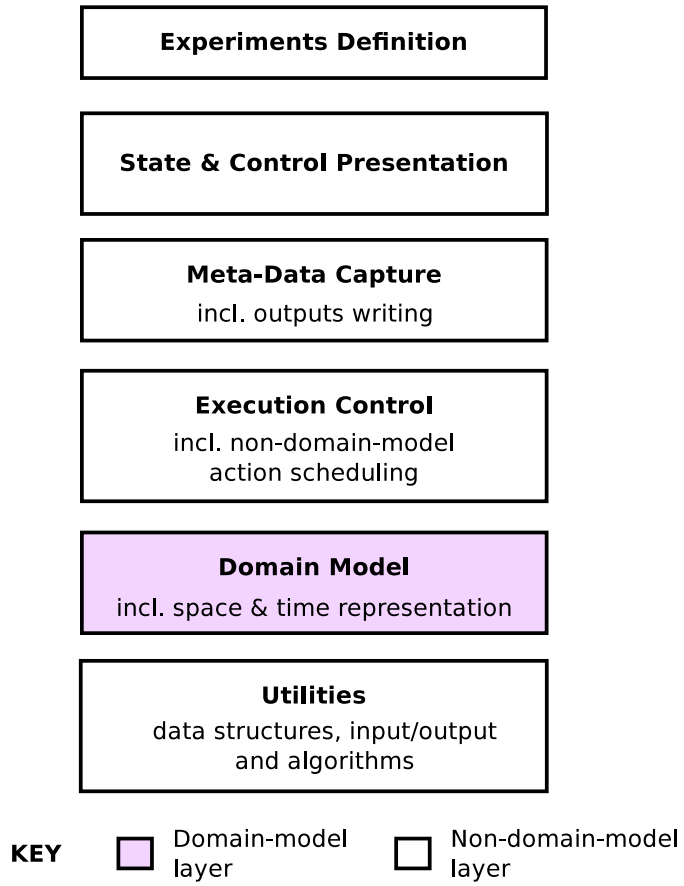

Figure 1: *Layered functionality for simulation code, which defines a generic layered architecture. This is extended in figures 2 and 3.*

Utilities. General utilities (not specific to the domain model or upper layers) for (a) data types (e.g., linked lists); (b) input/output capabilities, such as to/from different file formats; and (c) general algorithmic facilities such as random number generators, probability distributions or differential equation numerical solvers.[11] These can inter-

[11]Since particular instances of probability distributions represent part of the domain model (abstracting some aspect of the real-world), one could argue that they belong in the

act; e.g., probability distributions could be initialised from external files.

Domain Model. The code representing the abstraction of the real-world system, including the representation of space and time.

Execution Control. How the domain model is actually executed, which typically amounts to instantiating a 'root' object and stepping through a schedule of actions (provided by a domain model component) to 'unfold' time dynamically. Because non-domain-model objects also need to interleave their actions in simulated time, this layer includes that capability. This is a 'thin' layer, but nevertheless a well-defined one.

Meta-Data Capture. Code (scheduled in simulation time) to capture and calculate meta-data; i.e., derived model state (possibly held as a time series to capture changes over time) or atomic model state captured over time.[12]

This layer includes any writing of outputs to file (or database) because this can be tightly coupled with meta-data capture; in larger-scale simulations, time series data may be captured in a rolling window for storage reasons (perhaps with this window used for visualisation), with outputs written to file as they 'drop out of' the window (or via some other buffering strategy).

State & Control Presentation. The parts of the user interface which present model state and controls as part of a user interface. The presentation may be visual or textual. Where current model state is being presented, this directly uses the Domain Model layer. If meta-data is being presented, this uses the Meta-Data Capture layer.

In particular, note that a given domain model might have multiple presentations, with multiple alternative visualisations per component; such solutions *require* a layered domain model separation.

Experiments Definition. The parts of the user interface which support the definition of simulation runs (experiments), possibly including multi-run experiments. This mainly consists of how model inputs

domain model layer. However, they are such generic utilities (not usable just for domain-modelling) that I do not think it is controversial to place them here.

[12] In some models, state *that is part of the model* is derived from other state; e.g., the average income of all agents in a spatial neighbourhood might influence the behaviour of those agents. This is 'atomic' state from the perspective of the meta-data layer.

are defined and passed on to the model, and any automated manipulation of them across multiple runs for things like sensitivity analysis. Because this tends to be particularly generic to any simulation (and modellers using multiple toolkits may want a vendor-neutral solution), separate experimental platforms exist (Gulyás *et al.* 2011), and I am aware of simulation consultancies who develop their own in-house.

2.2 Relevant Cross-Cutting Concerns

There are some capabilities that require code across most or all layers; i.e., the functionality cannot be isolated into a single layer. In SE, these are called **cross-cutting concerns**. A good example outside of our reference architecture is the capability to 'freeze' simulation state to file, and 'reload' a simulation later to run from that point, which many simulation toolkits support. This typically relies on **serialisation** technologies in the underlying programming language, but requires that all the objects that are going to be serialised are coded in a particular way. In some cases, one might just want to freeze the state of the domain model but, in others, there might also be a need to capture the state of the model presentation (so higher layers also need to be coded to support serialisation).

In relation to our three best-practice properties, there are three specific cross-cutting concerns that need adding to our reference architecture (see figure 2):

Logging. Logging frameworks are a common tool in virtually all programming languages. They allow messages to be logged from applications, and support useful features such as multiple levels of logging detail which can be switched to as needed (including turning on more detailed logging only for certain areas of code); separation of messages per thread (for multi-threaded applications); and the automatic addition of fields (such as timestamp headers). Even where a debugger is available, the 'global' (full history of the run) and user-defined nature of log messages makes them complementary and perceived by many as preferable: "The most effective debugging tool is still careful thought, coupled with judiciously placed print statements" (Kernighan 1979). Such logs can also be used to automate model tests (see later).

In simulation terms, we typically want diagnostic logs of what the domain model is doing, with the ability to vary detail level per component at run-time. Such logs should be separated per simulation run, and with message headers useful for the simulation context

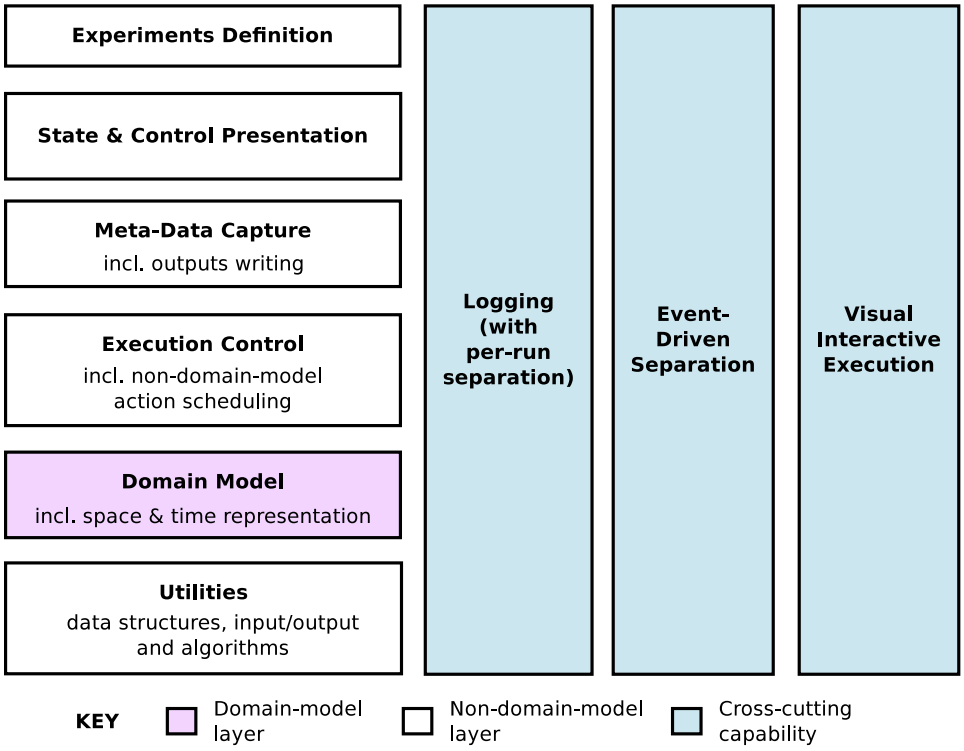

Figure 2: *Figure 1, with three important cross-cutting concerns added.*

(e.g., with the current simulated time in them). Even when simulation tools allow for visual, run-time navigation through the model and its state, this is still just an immediate snapshot view (compared to a full history in logs), has a fixed level of detail, and focuses on *state*, not the (algorithmic) details of entity behaviour.

Event-Driven Separation. In a layered architecture, upper layers often need to know when particular events (in the general sense) occur in lower layers; the classic example being when a visualisation of some data needs to know when the data has changed (to update itself).[13] But lower layers should be unaware of the existence of upper layers (i.e., be independent of them in their operation). More generally, this applies when any object want to know about changes in state of another object without the latter having to explicitly know about the former.

The normal design pattern to achieve this is to use a **publish-subscribe** design (Gamma *et al.* 1995, p.293), also known as an observer pattern. Objects publish events when they occur (to some events manager), and observers subscribe to events that they are interested in, being notified when they occur.[14]

In simulation terms, this is clearly useful in separating the State & Control Presentation and Meta-Data Capture layers from the Domain Model one in certain circumstances. (If some meta-data is captured daily, then the capture process can just run daily and derive the state directly; no publish-subscribe design is needed. Compare that with needing to display every time some specific random change happens to a model entity.)

However, this can also be very useful *within* the Domain Model layer. Most state changes that happen there have meaning in the domain model in terms of a causal event; e.g., 'Agent A sold 6 widgets to Agent B on market day 1'. (I call these **domain events** to distinguish them.) In this example, other agents might be interested in what transactions occurred so as to change their trading strategies for future days. We do not want each agent to explicitly call every other agent to communicate this; the publish-subscribe design is a much

[13] It could check (poll) every few seconds to see if changes have occurred, but this is incredibly wasteful of processing (especially if needed in many places) and imposes a minimum response time to changes.

[14] There are lots of variations, such as in when the subscriber gets the notification (synchronously or asynchronously), how events are formatted, and the granularity to which subscriptions can be specified.

cleaner solution. This is also advantageous in aligning the model implementation with its conceptual design: often we are modelling the *indirect* receipt of such information (possibly with imperfect transmission) in real-life, which is much more well-represented in the publish-subscribe design (given that we are abstracting this transmission process).

In fact, these domain events are *also* typically the events that the upper layers are interested in anyway: we want to present or output about the domain-contextual 'things that are happening'. Thus, one does not typically end up publishing two different 'styles' of event. Finally, the domain events (perhaps with some filtering) naturally provide a *narrative* for what is happening in the model which is very useful for communicating with a model audience, in testing, and in understanding how a model achieves the outputs it does (Millington *et al.* 2012). Outputting such a narrative naturally combines with the previous logging feature (as another type of log to diagnostic ones).

Visual Interactive Execution. Being able to observe the simulation unfolding in a suitable graphical user interface (GUI) is very useful for detecting both coding and conceptual errors (Grimm & Railsback 2005, §8.5.1), and thus enhancing testability. This does not have to be a cross-cutting concern if this view is just 'read-only'. However, what is additionally useful is to be able to *invasively interact* with the model at run-time, typically to change model parameters or state on-the-fly so as to be able to experiment more dynamically (often cued by information from the model visualisation). This is a form of **computational steering** for model exploration, though that term is normally reserved for large-scale mathematical computations (Mulder *et al.* 1999, §1). This does require support across the layers since, for example, domain model components should expose their state in a given way so that there exist 'built-in' visualisations that allow them to be changed; the components also need to code how they react to the change, depending on the function of the state or model parameter changed.

2.3 Some Layer-Specific Features

To complete our full reference architecture (figure 3), there are two simulation-specific features which sit in particular layers.

Run-Reproducibility Support. If we want to be able to reproduce simulation runs in an automated way, we need features which can record

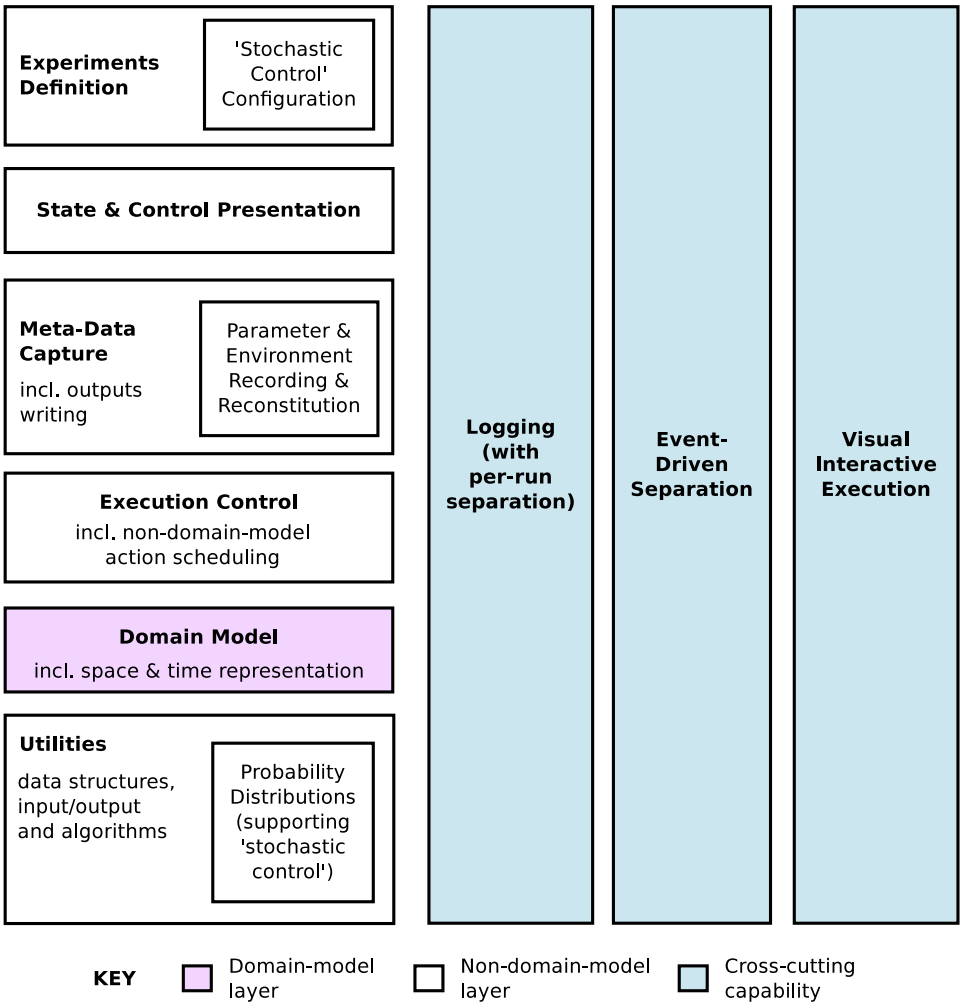

Figure 3: *Figure 2 extended to form the full reference architecture by adding some specific in-layer features.*

(a) details of the model code (e.g., its location in a version control system—see Wilson *et al.* (2014)); (b) details of all domain-model-specific parameters; and (c) details of the environment (e.g., name and version of all toolkits used, Java virtual machine details, operating system). Ideally we also want features which can *reconstitute* the model and its environment from this data, though that is significantly harder to automate.

This naturally falls into the Meta-Data Capture layer.

Test-Oriented Stochasticity Control. When testing or exploring a stochastic simulation, we often want to adjust the stochasticity so that we can 'better see what's going on'. For example, we might want to turn off randomness in some areas (typically by reverting to mean values) and/or accentuate it in others *in the same functional direction* so that we can see the effect of a particular sub-component (area of functionality) more clearly. In all cases, it is much more preferable if we can do this without having to manually change original model parameters or, worse, model code. (It is very common for particular stochastic elements to be hardcoded into the model, either for ease of coding or because the modeller does not expect users to need to change them.[15])

Thus, functionality which allows us to make these temporary adjustments *external to the model code and parameters* would be very useful. This requires support both at the Utilities layer (to have probability distributions coded in such a way that they support this dynamic stochasticity control) and at the Experiments Definition layer (to be able to specify the particular 'overrides' one wants for this run).

As mentioned at the start of this section, there are other possible reference architectures and, in particular, JAMES II (Himmelspach & Uhrmacher 2007) treads similar ground but for a different purpose. To set the reference architecture in better context, I make some brief comparisons in appendix A.

3 Understanding Simulation Toolkits

I want to show how the reference architecture can help provide insight into toolkit design by showing how some toolkits map to it. However, a

[15]Even if the *parameters* of the relevant probability distribution are exposed as model parameters, there are some distributions, such as the exponential one, which cannot even be *made* to revert to a mean value by changing their parameter(s).

particular toolkit classification is needed first to help set things in some useful context. (It is based on my own assessment of what makes most difference in the nature of the toolkit.) Figure 4 provides a visual summary for a number of popular toolkits: primarily ABM-oriented-ones, though Simul8[16] is a DES-specific toolkit and AnyLogic supports ABM, DES and SD.[17]

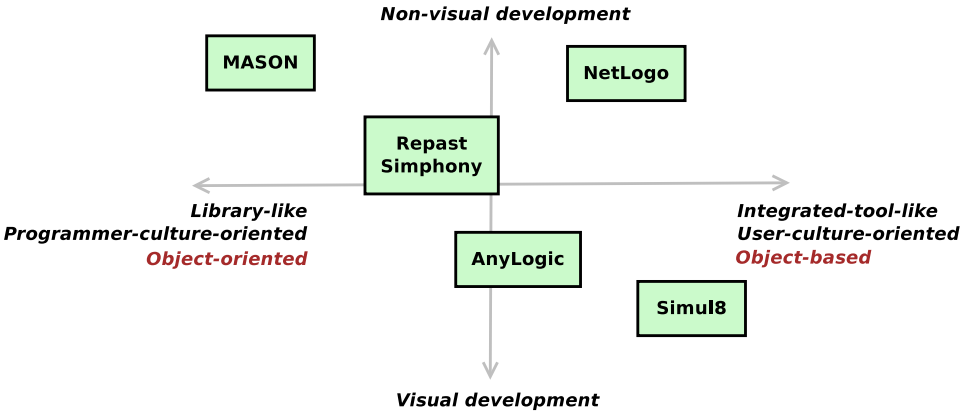

Figure 4: *A rough but useful categorisation of selected simulation toolkits by two axes.*

The axes represent two strong (and somewhat correlated) distinguishing 'positions':

Visual vs. Non-Visual Development. Some toolkits, to a greater or lesser degree, permit code to be designed visually, typically by dragging and dropping configurable 'widgets' and (where appropriate) linking them together. Some form of textual code is normally required in addition to provide logic which cannot easily be represented visually, but a visually-oriented toolkit will normally attempt to minimise this. There is still considerable debate in computer science over when such visual programming languages provide benefits (both in terms of the type of code they are best for and what types of users most benefit), and when there are trade-offs (e.g., in the expressivity of the language). Authors such as Green & Petre (1996) and Whitley & Blackwell (2001) give a flavour of this.

Visual coding has long been the norm in commercial operational research (OR) oriented offerings, such as Simul8 for DES, iThink for

[16] See `http://www.simul8.com`.

[17] Actually, Simul8 can add agent-like behaviour to DES entities, and has some SD constructs; Repast Simphony includes some SD constructs. I would argue more generally that the paradigm distinctions are somewhat artifical at the toolkit level, and becoming increasingly so, but that is a debate for another time!

SD[18], and the multi-paradigm tool AnyLogic. Elements of visual coding are becoming more prominent in ABM—e.g., Repast Simphony's visual statechart construction, and the ABM components of AnyLogic. When the code is designed visually, this also normally provides a *run-time visualisation* of the code as it executes, thus providing a particular fixed form of visualisation 'for free'.

Although NetLogo is coded non-visually, it takes an alternative route towards conceptual abstraction: providing a paradigm-specific simulation language (as most DES toolkits historically did).

Library– vs. Integrated–Tool–Like. Some toolkits exist 'just' as a set of core libraries which provide a framework for constructing simulations, such as MASON. Others integrate this framework together with related GUI tools to aid code construction and testing, such as those for debugging, version control, and code navigation. (The visual GUI nature of the IDE should not be confused with visual programming of the actual model components.) In general, such integrated tools are referred to as integrated development environments (IDEs) in SE.

When using a more library-like toolkit written in a general-purpose programming language, the modeller can use third-party general IDEs (such as Eclipse[19] for Java). However, a simulation-toolkit-specific IDE will also typically integrate 'helper' tools tailored to the simulation-specific reusable components provided by the framework, and the way that simulations are used via experiments. Visual coding will tend to lead to this form of IDE, since most elements of it are needed to do the visual coding.

This continuum normally goes hand-in-hand with another distinction: integrated tools tend to be **object-based** rather than **object-oriented** (Joines & Roberts 1999, §4). Whilst all modern toolkits are *written* in an object-oriented programming language, an object-based one presents an abstraction to the user of a fixed set of paradigm-specific components (or, in NetLogo's case, a fixed syntax), which can only be extended via *composition*. An object-oriented one exposes a set of classes that can be extended via standard object-oriented techniques, which is typically more flexible at the expense of complexity.

Less formally, I think that the two ends of the spectrum are closely linked to *cultures* of software use, and reuse the terms **user and pro-**

[18] See `http://www.iseesystems.com`.

[19] See `http://www.eclipse.org`.

grammer cultures used by Guo.[20] Library-like toolkits lean towards a programmer culture, which focuses on the qualities of good software espoused by SE. Thus, things like expressive power, flexibility and reusability are important. There is also an understanding that best-of-breed external libraries would be used as needed by the programmer, with the flexibility in this choice outweighing the need to pick one and strongly integrate it into the toolkit. Integrated-tool-like toolkits lean towards a user culture, where the focus is on providing integrated tools for the task in-hand. Such tools typically provide their own conceptual models which try to hide the underlying implementation complexity and provide convenient high-level abstractions (which also steer the user towards a particular way of thinking about the task). In simulation terms, this is also related to ideas that domain experts (scientists) should be able to create models without also being programmers (Borshchev & Filippov 2004, §6), and that SE skills are potentially too difficult, intimidating and/or time-consuming to learn (Brailsford 2014, §2).

Both types of toolkit may also support extensibility via other mechanisms: either 'hook points' where snippets in a textual programming language can be inserted (e.g., Simul8[21]) or being able to extend the syntax of the domain-specific language via lower-level code (e.g., NetLogo). AnyLogic is a good example of a 'halfway house' which provides an object-oriented platform (but with some restrictions) and lots of object-based components with configuration and extensibility via Java snippets.

We will now have a look at how MASON and AnyLogic map to the layered aspects of our reference architecture. (We will come on to the cross-cutting concerns for all toolkits afterwards.) This is a good pair to choose because they are relatively far apart in our classification, whilst both still being object-oriented (or largely so in AnyLogic's case) which tends to expose the internal code architecture more clearly. In both cases, I do not explain the exact function of all the components shown for space purposes; where they are not self-explanatory, the interested reader can look them up in the toolkits' documentation (which is publically available for both toolkits).

[20] In his blog post about teaching programming: `http://pgbovine.net/two-cultures-of-computing.htm`. Guo is an academic (University of Rochester), but I know of no literature explicitly discussing these cultures, though related ideas are implicit in human-computer interaction (HCI) research.

[21] Actually, Simul8 models can be extended both with a Simul8-specific simulation language—Visual Logic—or general purpose Visual Basic.

3.1 MASON Reference Architecture Mapping

The mapping is summarised in figure 5. There are three main things to notice.

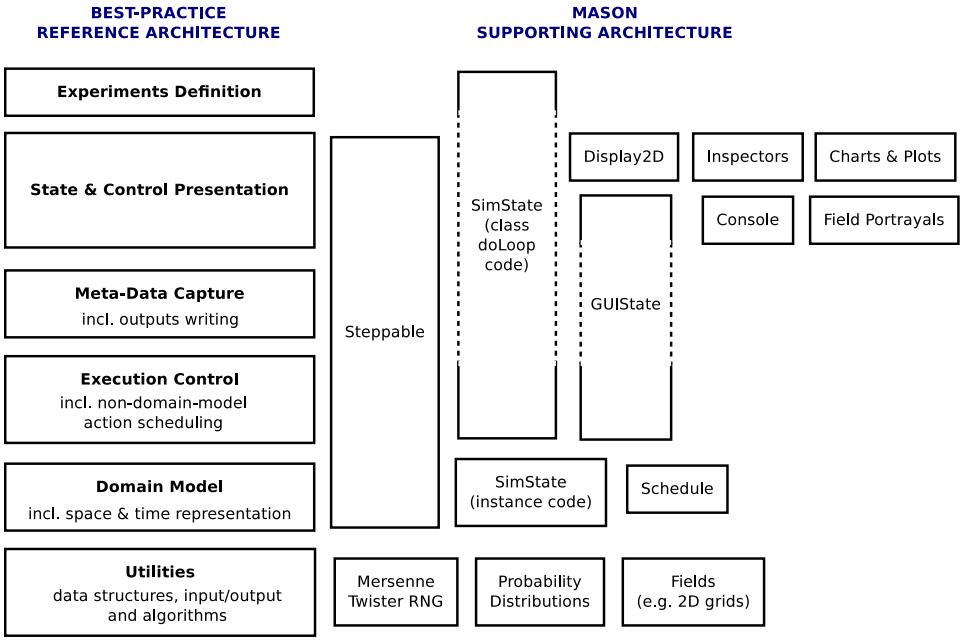

Figure 5: *How core components of the MASON toolkit map onto the reference architecture. The dotted border sections of some multi-layer components indicate layers that they do not cover. An RNG is a random number generator.*

Firstly, despite being very library-like and 'programmer friendly', MASON still diverges from the reference architecture in some areas (see the three layer-spanning components). Having said that, these divergences are largely pragmatic decisions: (i) There is a sharp layered distinction between model and presentation (SimState and GUIState)[22], but meta-data capture components can just be included in one or the other, depending on whether the modeller considers them part of the model 'core' or not. If one thinks of the model as an input-output converter, it makes sense to put any meta-data capture used for writing permanent outputs into the core model. (ii) Steppable is a generalised interface for something that performs actions in the simulation, and so applies to any model component doing things in simulated time. However, domain model components are still strongly partitioned by being part of the SimState. (iii) The class-level (Java static) SimState code which spans two layers (the doLoop method)

[22]It is still possible for pragmatic reasons to have an agent 'visualise itself'; i.e., embed the visualisation within the agent's code. However, this is a modeller choice, and is not emphasised (cf. AnyLogic in section 3.2).

is really just a helper function to instantiate and launch models, running them a given number of times for a given time window. It would be overkill to separate that code into layers. (iv) Similarly, GUIState primarily covers the 'thin' Execution Control functionality, but also serves as a container ('root' object) for all the presentation components, as well as the domain model which it 'wraps'.

Secondly, MASON gives the modeller transparent, fairly low-level access to the building blocks, particularly for model user interfaces. The main GUI Console is augmented with user-defined displays which contain inspectors (agent state presentation), portrayals (presentation of spatial or topological fields), or charts and plots. The underlying Java Swing GUI components are 'visible' and accessible in many places.

Thirdly, though it is not clear from the figure, MASON does have a very clean separation of the Execution Control layer, in that there is a separate schedule for non-domain-model objects, with the Execution Control logic stepping through the domain-model schedule and, after all processing for a given simulation time, processing any on the non-domain-model schedule.

3.2 AnyLogic Reference Architecture Mapping

The mapping is summarised in figure 6.

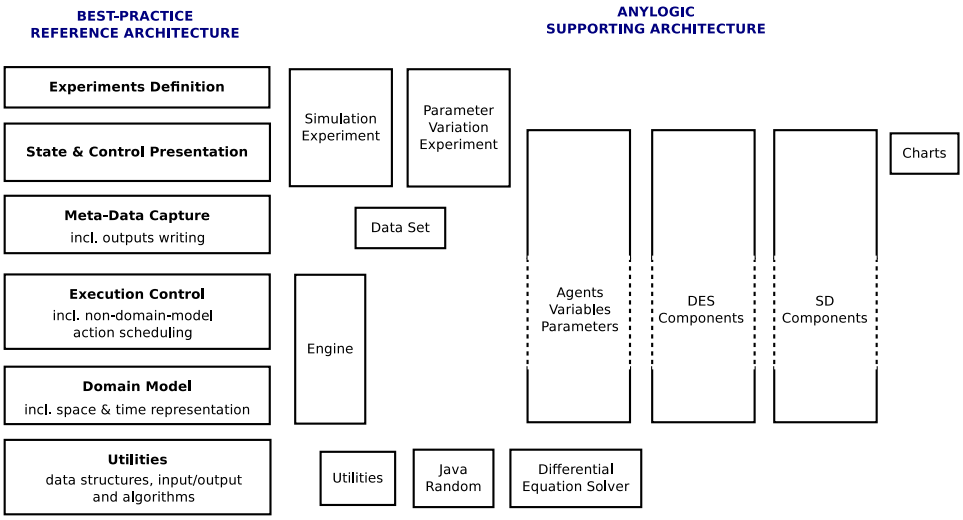

Figure 6: *How core components of the AnyLogic toolkit map onto the reference architecture. The dotted border sections of some multi-layer components indicate layers that they do not cover.*

Firstly notice how, compared to MASON, the components are more spread across the layers. This is primarily because all the domain model

components (agent and DES/SD components) have built-in run-time visualisations, and thus they have 'vertical' functionality. This is misleading in the sense that *user* presentation code can still make the layered separation if it wishes to (though AnyLogic does not tend to encourage this), which we will see in the case study.[23] Since AnyLogic focuses more on being an integrated tool, it wants a lot of its components to provide 'useful' default functionality where the user does not typically make distinctions between model and visualisation. (There are also commercial reasons to make model components not reusable outside of the AnyLogic ecosystem.)

Secondly, unlike MASON, there is an explicit Meta-Data Capture layer component (Data Set). This is primarily to capture some standard functionality in a visually-developed widget: maintain a time series of real numbers (floating-point values), with an event to periodically populate it (where the 'interleaving' of this event with others is hidden from the user). In fact, AnyLogic merges the Execution Control and Domain Model layers in having non-domain-model objects sharing a single master schedule with domain model ones. With the current implementation, this means that these meta-data capture events are *not robust*: they can end up occurring before domain model actions have finished at a given simulation time.[24] AnyLogic charts can also use this data type explicitly as the source for their visualisation.

MASON essentially allows the same thing, but the modeller creates the parts explicitly, using whatever data structure they want. (They may not want to capture a numeric value, for example.) An AnyLogic modeller can also take this approach if desired, since AnyLogic also exposes low-level Java features.

3.3 Consistent Omissions in all Toolkits

What is perhaps surprising is that I know of *no* simulation toolkit (including several outside of those in figure 4) which makes *any* of the features in sections 2.2 or 2.3 (except for visual interactive execution) available to

[23] Lower-level presentation and non-presentation elements are still distinguished in terms of different types of drag-and-droppable widget, where the user has control over what presentation is visible at run-time at what 'levels' of the model. However, the modeller is still encouraged to embed presentation elements within the thing that it visualises, and meta-data capture elements within the most relevant domain model component. Thus, although there is some *class hierarchy separation*, a *layered separation* is not encouraged, though possible, for user code. Note that it is the embedding of presentation elements within domain model elements which breaks the layering; the converse can still preserve it, as is done in MASON where the GUIState embeds the SimState.

[24] See `http://ofscienceandsoftware.blogspot.co.uk/2014/09/subtleties-of-anylogic-event-scheduling.html`.

the modeller in its full form. That is, all the following are either absent or partially present:

- logging;
- event-driven separation;
- run-reproducibility support;
- test-oriented stochasticity control.

For logging, Simul8 provides logging automatically, but only for its own components and with fixed content. Most tools provide the capability for user code to write messages to the console, but with no switchable detail level, no per-run separation for multi-run experiments, and no permanent file capture.[25]

For event-driven separation, features such as Repast Simphony "watchers" and AnyLogic message passing can be used to do similar things. However, neither centralises the idea of domain events (with a related narrative), with both focusing on specifying senders (or sources) and receivers without decoupling them.

For run-reproducibility, some toolkits (e.g., AnyLogic) allow experiment settings to be defined and retained, so experiments can be reproduced by re-running with those saved settings. However, this captures nothing about the environment (e.g., the version of the toolkit used) and relies on user discipline and effort to maintain a proper traceable link between run and experiment. (Ideally, one should create a new AnyLogic experiment for every run that needs to be reproducible, and there is nothing to stop the user inadvertantly changing such experiments.)

For test-oriented stochasticity control, Simul8 effectively has probability distributions defined as separate object instances (unlike many other toolkits, including MASON and AnyLogic). Thus, these objects can be changed to a simpler distribution as needed for testing purposes (e.g., an exponential distribution with mean of 2 can be replaced with a fixed distribution returning 2 so as to 'collapse the exponential to its mean'). However, this still requires the user to determine a relevant alternative distribution (i.e., the *operation* the user wants to perform—e.g., collapse to mean—is not explicit) and this still involves changing the 'real' parameters of the model (and remembering to change them back afterwards!), rather than applying per-run override settings.

Even where the capabilities partially exist, the lack of consistency across toolkits is problematic. In terms of *why* these are not included, I think this

[25] In AnyLogic, for example, the console also only retains a rolling window of messages.

is largely just a question of toolkit focus and style (see section 3). They are not included in user-culture-oriented toolkits because they tend to present a programmer-oriented way of thinking about how to develop and test simulation software which is not consistent with the user-oriented view; instead, limited user-centric versions of them are sometimes included, and users that might want them would be expected to develop those facilities themselves. I expected them to be more likely to exist in programmer-culture-oriented toolkits, and I can only think they are missing because (a) the toolkit focuses on the simulation core and not the overall modelling process (which these aspects strongly relate to); (b) they are areas where programmers might prefer the flexibility of defining their own approach—all the aspects have a few different ways to approach them (some more heavyweight than others), plus logging and event-driven separation are standard SE design patterns, with the former having widespread library implementations.

I do not think that any of these reasons are suitable justification not to look at developing standardised solutions for simulation, and there *is* considerable implementation complexity, even for less simulation-specific areas like logging where existing libraries provide much of the functionality.

4 The JSIT Library

To address the consistent omissions detailed in section 3.3, I have developed an open-source Java-based library—Java Simulation Infrastructure Toolkit (JSIT)—which works towards a solution for the partially-missing aspects. JSIT *only* provides the four section 3.3 capabilities; it is *not* an attempt to implement the entire reference architecture. It works *with* existing simulation toolkits which, as section 3 discussed, each implement their own partial mapping to the reference architecture. (When I use 'toolkit' henceforth in this section, I mean the toolkit JSIT is being used with, not JSIT itself.)

JSIT should be usable with *any* Java-based simulation (or one that can interoperate with Java) but, in terms of use with specific simulation frameworks, it has currently only been 'proven' for AnyLogic and MASON. Using it requires that the base simulation is coded in particular ways, and the degree of integration-specific 'glue code' required depends on the toolkit used. The idea is that JSIT **helper libraries** will exist for commonly-used toolkits that do most of this work for the modeller in a generic way, leveraging any useful features in the toolkit. Currently, only a helper library for AnyLogic has been written, but there is also a sample MASON-based

model that shows how to integrate with JSIT in a 'raw' way (i.e., without use of a helper library, which effectively means implementing a simplified version of the helper library functionality as part of the model; since this does not have to be generic, it can take various short-cuts to work just for the model in question).

AnyLogic was explicitly chosen as the initial helper library focus because, compared to MASON and Repast Simphony, I expected it to have the most implementation issues (and thus the most potential influence on how the core JSIT code would need to work) because it is the most oriented towards being an integrated, visual development tool (see figure 4). My hypothesis was that the user culture focus (and commercial interests) might compromise the technical architecture, in the sense that the user-oriented features and conceptual model 'leak into' the user-visible parts of the technical architecture (or result in less developer focus on it) in a way which might impede the implementation.[26] I believe this hypothesis to be correct, although I cannot be sure until the other implementations are fully complete. (I should also point out that AnyLogic-specific features also *aided* in some aspects of the implementation.) In any case, outlining the specifics in the AnyLogic case is interesting, and gives an insight into some more general ideas (see section 6).

Figure 7 shows how a model uses JSIT. The core user-written model is some 'root' class which aggregates a set of model components (which may themselves aggregate other components). The root class is coded to specify itself as a JSIT-specific 'main model', and then uses JSIT features via the required interfaces. The helper library (or equivalent user-written code) provides the underlying link to the simulation toolkit used, but is never called directly by the core user code (i.e., the interaction with JSIT is always via a toolkit-agnostic core API).

Much more detail, together with the case study model (section 5) and sample MASON-based model, are available with full source code at `https://github.com/sprossiter/JSIT`. A user guide is included.

I now explain briefly how each of the aspects in section 3.3 is implemented, with some notes on the AnyLogic helper library implementation.

4.1 Logging

This reuses a widely-used, open source Java logging framework: Logback.[27] Simulation-specific message header information (simulated date and time) are added in a generic way, and diagnostic log files are split per

[26] I had already had inklings of this from my previous work developing models with AnyLogic, MASON and Repast Simphony.

[27] See `http://logback.qos.ch`.

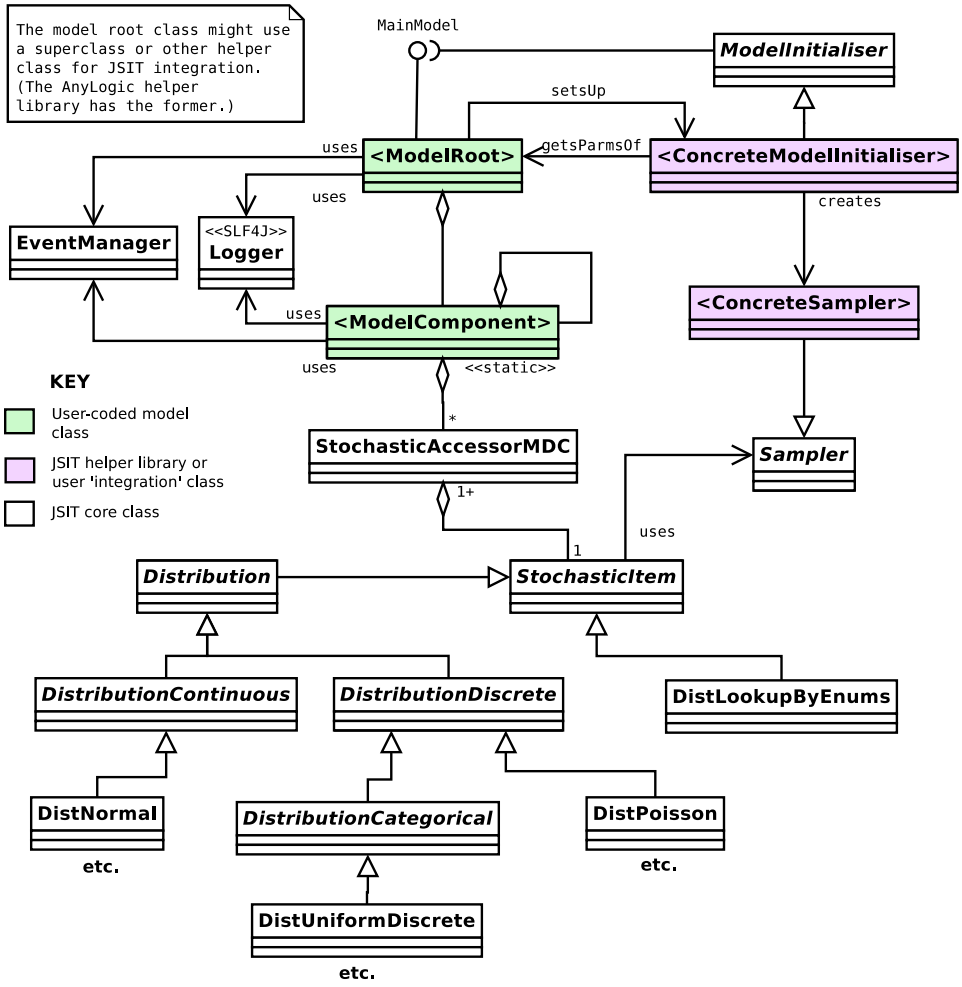

Figure 7: *A UML class diagram showing the main classes and relationships when a model uses the JSIT framework, and how this splits into core JSIT code, JSIT helper library code (which might be coded by the user if no helper library exists), and user model code. Extra Java interfaces related to the EventManager class are not shown. For a background to UML, I recommend Fowler (2004).*

simulation run in multi-run batch experiments. The advanced features of Logback (compared to other Java logging frameworks) were needed to implement the per-run separation. User code just uses standard Logback mechanisms to get an appropriate Logger and log messages at differing detail levels. A per-run configuration file is used to specify what diagnostic levels are required from what classes.

The AnyLogic integration was particularly problematic because AnyLogic has some strange threading strategies; for example, single-run experiments can have models initialised in a different thread than the one time is stepped within, and certain circumstances cause models to switch execution to totally different threads. This significantly complicates the logic needed to separate log files per run, and involves AnyLogic-specific alternatives to the normal way that JSIT logging works.

4.2 Event-Driven Separation

JSIT provides a simple EventManager class and a set of interfaces that event sources and receivers need to implement. Event sources declare whether an event is a domain event or not, so the framework can also be used for non-domain-events (see section 5 for an example). The EventManager writes all domain events to a special domain events log file, which provides the model's narrative (and reuses the logging solution). This proof-of-concept implementation is restricted to synchronous messaging; i.e., those object subscribing for certain domain events receive them immediately on creation. Objects can subscribe to events from all instances of a source class, or just to those for a specific instance.

There are no particular AnyLogic integration issues.

4.3 Run-Reproducibility Support

When a model is run, the helper library ModelInitialiser subclass (see figure 7) automatically records environmental information *and* all the model parameters, using Java objects for this information which are then serialised to an XML file (via the open source XStream library[28]) in a fairly human-readable form. Run-reproducibility information is also included at the start of the diagnostics log.

Reproducibility is significantly aided by storing model code in a version control system (VCS)—see Wilson *et al.* (2014). If so, environmental information is recorded on the location of this model version's code so that the exact version can be restored as needed.[29] The information also

[28] See `http://xstream.codehaus.org`.

[29] This is currently limited to a Subversion VCS. See `http://subversion.apache.org`.

specifies if the model code had been amended from the version retrieved from the version control system.

The automated reconstitution of a model run from the information in this file is much more complicated to implement than its creation (and has not yet been done), though the object-serialised nature of the file means that it is easy to recreate the objects that it was created from. (But that just provides the model parameters and environment details; that environment ideally should still be *constructed* in some automated way where needed.)

AnyLogic model parameters are held in a particular way in the Java code generated for the root class. This means that model parameters can be retrieved in a generic way without requiring the user to specify or define them in any particular way. However, one key model parameter is the random seed value (if stochastic), which AnyLogic does not make accessible (largely due to restrictions in Java's Random class); it also internally generates some extra RNG instances that affect the seed that the 'real' one gets (and may impact reproducibility if these internal details ever change). The code works round these issues by defining its own RNG (which the model must use). AnyLogic also does not currently allow the AnyLogic version used to be determined at run-time.

It is also not normal to separate an AnyLogic model from the experiments which run it, but this is possible by defining an 'experiments-only model' which has a dummy root Agent which wraps the real one. (If this separation is not done, JSIT has no way of distinguishing changes to the model code from changes to an experiment.)

4.4 Test-Oriented Stochasticity Control

A set of classes (with StochasticItem as the top-level class) provide a set of probability distributions which the model includes instances of. The *implementation* of these distributions (in terms of sampling them normally) is still provided by the toolkit, and the helper library Sampler sub-class (see figure 7) makes this link. However, JSIT provides the implementation for *overridden* sampling: currently, the only override operation supported is to collapse distributions to their mean, but a number of other useful ones are intended for the future. The user sets up an external configuration file to define any overrides required for the run (and for what distributions). There are also some other forms of convenient 'stochastic items' supported, such as lookup tables of distributions (useful where the distribution sampled from depends on the attributes of an individual entity, such as death rates by gender and age).

A separate advantage of this solution is to standardise the representation of probability distributions across toolkits, whilst still leveraging

the per-toolkit implementations. As I said earlier, most toolkits I know of do not represent them as object instances, and the sampling methods provided often have confusing differences between parameters.

In terms of AnyLogic integration, AnyLogic multi-run experiments can launch parallel runs which run *in the same Java virtual machine* (but in different threads). This means that there needs to be very careful concurrency-related design: distributions are typically *shared* by all instances of a given entity type, but there need to be separate instances *per run* which will be accessed from different threads (and we should assume this might be required for non-AnyLogic-models as well). In the generic case, the modeller uses StochasticAccessorMDC instances (see figure 7) to manage this but, because of the threading issues with AnyLogic discussed in section 4.1, there is an alternative solution for AnyLogic.

The 'distribution lookups' mentioned above also leverage the AnyLogic HyperArray element, which has a useful visual interface that can be used to set up lookups to Bernoulli distributions.

5 A Case Study on a Health & Social Care Model

To bring everything together, an existing multi-paradigm model of health and social care (Viana *et al.* 2012), using AnyLogic, was designed to use the JSIT library *and*, where possible within the constraints of AnyLogic, conform to the reference architecture. Since one of the goals of the reference architecture is testability (section 1), this also included setting up automated tests which compare outputs to the 'narrative' events log.

5.1 Architecture

The relevant architecture is shown in figure 8, which the reader should refer to in what follows.

The model looks at eye clinic patients suffering with age-related macular degeneration (AMD). These patients are treated at an eye clinic with eye injections which can slow the central vision loss associated with AMD, but they are contending for resources with other non-AMD eye clinic patients. The more general social care needs of the AMD sufferers is also modelled, because one broader aim is to explore how health and social care interact in an ageing population. AMD sufferers and the clinic exist in a 2-D space.

Thus, the Domain Model layer consists of AMD and non-AMD patients and an eye clinic, with AMD sufferers and the clinic composed by a root 'core model' object (but this is never run directly). The eye clinic holds and

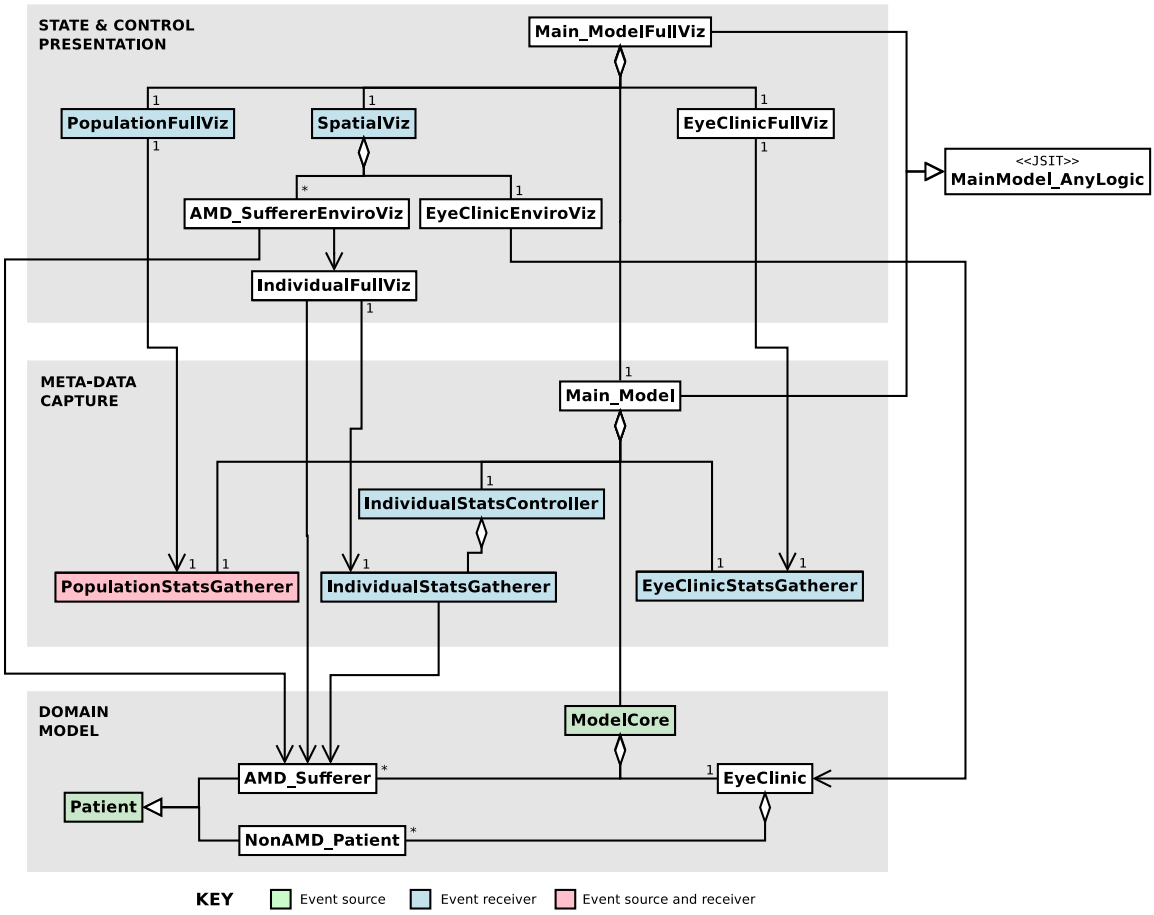

Figure 8: *A UML class diagram showing the architecture of the AMD model, emphasising the mapping to the reference architecture. The EyeClinic actually includes a separate component which schedules clinic appointments and handles the dynamic creation of non-AMD patients, but that separation is not important for the exposition here.*

generates non-AMD patients itself. The AMD sufferers and the root object produce domain events, such as starting and completing an appointment at the clinic. (The root object produces events for when new AMD sufferers are created, abstracting the AMD development and clinic referral process.)

The model produces detailed outputs over simulated time for the operation of the clinic, AMD sufferers' characteristics (e.g., sight level over time), and aggregate population statistics (e.g., numbers of AMD sufferers with different social care need levels). These are also the basis for some of the run-time visualisation. Thus, the Meta-Data Capture layer includes 'stats gatherers' for these aspects, and a controller for the multiple per-AMD-sufferer stats gatherers. Some of these statistics are just time series of agent characteristics at regular intervals, and so can just be sampled from the agent directly. However, most of the statistics relate to specific domain events occurring dynamically (e.g., appointments being completed) and here domain events are used: they receive notification of events they have subscribed to, and can then query the relevant component for the state they need to capture.[30] In particular, the controller is notified of new AMD sufferers, and can then dynamically create a stats gatherer for that agent. A root 'main model' object composes all these into a simulation which can be run as a visualisation-less version of the model.[31] (There could also be multiple variant root objects representing different combinations of meta-data capture and output.)

Modeller-coded visualisation consists of (i) a visualisation of the 2-D space with AMD sufferers and clinic represented; (ii) a visualisation of the clinic in operation, with patients and staff moving around a layout of the space; and (iii) graphs and charts corresponding to the clinic, individual AMD patient and AMD population areas captured in the lower layer. There is also navigation around the various visualisations, including being able to 'click-through' AMD sufferers in the spatial visualisation to go to their statistics presentation, and on from there to view the actual agent state (provided by the built-in AnyLogic visualisation). Figure 8 shows the State & Control Presentation layer objects used to achieve this (and their relationship to lower-layer classes). Both domain and non-domain events are used here to good effect. Domain events for AMD sufferer creation are used to add their visualisation to the spatial presentation, where these visualisations graphically show certain agent characteristics (such as their

[30] This is how I chose to design the JSIT library; another option is to include the relevant state information within the event itself. However, the JSIT solution results in simpler model code, and conceptually separates the event as a 'thing that happened' from the detailed state related to it.

[31] The built-in visualisations provided by AnyLogic are still created unless experiments are run in batch-mode.

stage of AMD in each eye); thus, the spatial visualisation also receives domain events relating to changes in these characteristics and updates the agent visualisation accordingly. A non-domain event is used for the population statistics visualisation to be aware when the related stats gatherer has stopped collecting (typically at the end of the model). This is needed so it can provide final updates to its charts. (They otherwise update at regular intervals, but will miss some data without this event-driven update.)

5.2 Testing

To do automated tests of the whole system, the model can be set up with simplified parameters to perform logic whose outputs can be worked out a priori 'on paper'. To test individual components, the same thing can be done but with only those components needed composed together into the test model, with fake objects used to replace domain model components not under test where needed. (A 'fake' is a specific SE testing term; see Gürcan *et al.* (2013).) Figure 9 shows the case when testing the eye clinic component (which is effectively a DES model). Note how only the components needed are composed via a single test-specific root object, and AMD sufferer objects are replaced by fakes which include only the minimal functionality needed to operate as part of the clinic appointment cycle. (Most of this logic is in the existing Patient class, so the fake logic is relatively simple.)

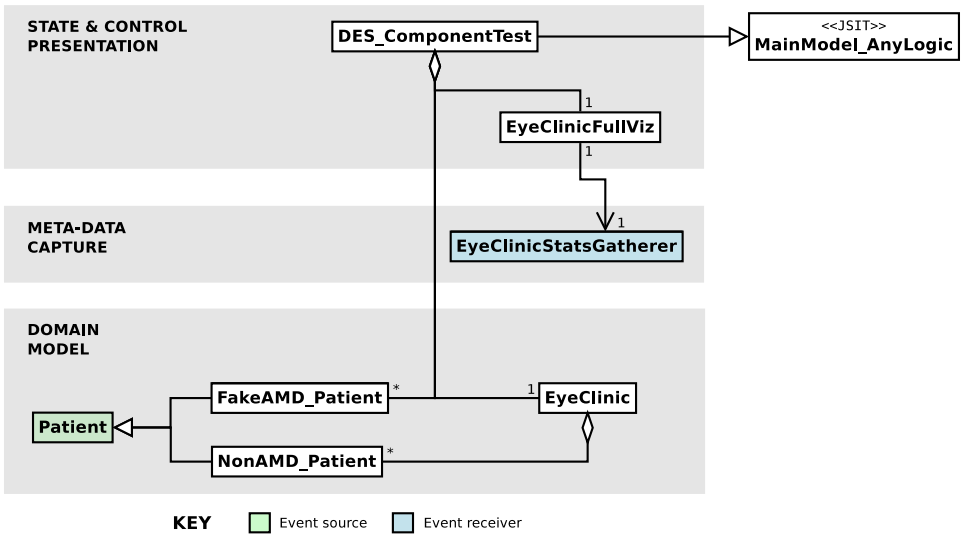

Figure 9: *A UML class diagram showing the architecture of the reduced model used to test the DES-based eye clinic component; compare this to figure 8.*

Full details can be seen in the online material.

5.3 Other JSIT-Enabled Functionality

By use of domain events, a narrative events log is automatically created. All classes use the logging feature, subdividing their messaging by diagnostic detail level. For example, the default INFO level is used for 'progress' messages that we would normally expect the user to want to see. The lower DEBUG and TRACE levels provide increasing detail about what the logic is doing and relevant state. The run-reproducibility features of JSIT also automatically create a settings file per run, with details of all model parameters and environmental characteristics.

Appendix B shows some example file extracts.

5.4 Reflections

The usefulness of the design really proved itself in practice, and the abstract best-practices it attempts to encourage reinforced each other. The layered, loosely-coupled design was invaluable in assisting the creation of tests at different levels, and the separation encourages the modeller to think much more clearly about the set of features that they need and which ones really need to be coupled together. Combining this with defining model parameters as an object hierarchy gives a really clean mapping of parameters to components, which also makes reusing components simpler. The logging significantly helped in debugging, and nicely complements the code visualisation AnyLogic provides; the former focuses on *behaviour* (with relevant state secondary), whilst the latter focuses on *current state* and exploratory navigation between components. (With diagnostic logs, one can do this exploration over several re-runs by changing the diagnostic configuration to focus in on different potential problem areas, or just log the full detail and navigate through that via knowledge of the logic and message content.)

The automatic settings file production, even without the ability to *automatically* reconstitute a run, has already helped in providing a 'permanent' record *within the run outputs* of the exact settings used, which has been useful for run provenance and debugging.

However, there are also some less obvious *conceptual* benefits: (i) Using domain events really encourages the modeller to think about the flow of actions in their model, what are the critical things that happen, and what agents (or model components) might care about them. This is a nice complement to thinking of the model as an 'algorithm'. (ii) Setting up the expected outputs for automated tests (and the inputs needed to achieve the required behaviour) is time-consuming, but forces the modeller to effectively re-review their entire *conceptual* design. I picked up on several

design flaws by doing this, as well as a number of subtle bugs. Such testing also tends to highlight unnecessary complexity in the design, because this often makes testing trickier and the modeller is encouraged to consider whether that part of the design is really necessary.

In terms of AnyLogic 'getting in the way', most of this complexity is abstracted away into the JSIT framework. AnyLogic is still an open platform with object-oriented extensibility, and thus the non-JSIT design was reasonably straightforward. The only difficulty is that all the AnyLogic help, example models, and textbooks do not encourage one to think in this layered way—no example models *ever* have visualisation-only Agents, for example—and the hierarchy of nested Agents is assumed to be the way the user would want to navigate the model. This meant adding some simple navigation facilities in user code that would handle the transitions the model intended. Testing is also made more complicated because, unless one uses the very expensive Professional Edition, AnyLogic does not provide external access to AnyLogic models (to be able to execute tests relating to them), so any testing needs to be integrated into AnyLogic experiments.

6 Conclusions

This paper presents a reference architecture which embodies key SE design patterns, and helps modellers understand how to create simulations which exhibit a core set of SE best-practice properties. I have tried to show how this (together with some classificatory background) helps critically understand and evaluate simulation toolkits, and that it is possible to practically apply these ideas on real models using mainstream toolkits, aided by the JSIT library (developed as part of this paper) to provide four specific generally-missing capabilities.

The reference architecture is clearly *one such* architecture (see appendix A), but I believe that it captures the most important decisions; i.e., the ones with most architectural impact (and thus, referring back to Fowler's quote, the hardest decisions to change later). I am keen to collaborate directly with toolkit developers (most of whom are also active researchers) and related initiatives such as JAMES II to see if useful consensus can be established. It would also be useful to use the ideas in this paper to better 'place' the emerging ideas discussed in section 1 within the landscape of toolkits and conceptual/software design abstractions. The JSIT AnyLogic integration also highlighted that there could be value in trying to define a form of **architectural contract** for simulation toolkits that would guarantee consistent architectural behaviour. As one example, there should be an

expectation that a non-parallel model (i.e., one that runs single-threaded) should exist in a single well-defined thread for its full lifecycle (or at least a set of sibling threads).

In terms of the case study, I am somewhat biased in that I developed these ideas and the JSIT framework as part of creating the AnyLogic-based AMD simulation model. There is a need for other modellers to work with the ideas, and the JSIT library, and feed back their own experiences. There is clearly significant overlap between this task and those in the previous paragraph.

I also focused primarily on ABM modelling (partly due to the readership of JASSS, and partly due to my own experience), though I claim the ideas are universal. There is a need to demonstrate that this is so via case studies and discussion in other areas (though note that the AMD case study here is in fact a combination of ABM, DES and SD, developed by complexity scientists and operational researchers). In particular, ideas such as logging and event-driven separation seem ill-suited to continuous-time models such as SD ones, but there is still *some* potential for use there. For example, an SD stock representing a market price may have domain-meaningful events when certain critical values are reached, or at intervals relating to financial reporting cycles. There is no reason why an SD model could not include 'ABM-like' ideas, such as having a rate equation change form when a critical event occurs elsewhere in the system. So, in fact, the adoption of the ideas here may even *encourage* more cross-paradigm conceptual thinking, as well as generic SE best-practice.

The JSIT library is still partly proof-of-concept and there is lots more I intend to do in terms of providing more helper libraries, 'hardening' the code (e.g., so that it scales better for large simulations) and improving documentation. The more it is used on different style models in different disciplines, the more its generality will be tested and the likelihood that useful design improvements emerge. For run-reproducibility support, the idea has recently been taken forwards for general scientific computing by the Sumatra toolkit (Davison *et al.* 2014), and integration with that looks promising (though it is Python-based).

Finally, there are two broader issues which are worth slightly more extended discussion, where the first informs the second.

6.1 Why is there so little Focus?

If these kinds of SE-driven ideas are important for producing bug-free, flexible and reusable simulation code which reuses design patterns tried-and-tested across years of SE research and practice, why is there so little focus on it in the simulation literature, and how even is this neglect across

disciplines?

I do not think that there is a particularly deep methodological answer, or one with historical specifics per discipline. There are the same set of reasons as discussed in the broader scientific computing best-practice literature and by related organisations (see section 1): (i) the focus has always been on the science, with a feeling that these SE ideas are too 'heavy-weight' or too 'commercially-focused', especially for single-scientist-developed simulations; (ii) there is little to no formal SE training for computational scientists, and there are always issues of what to remove from the curriculum if it was added; (iii) there is a perception that SE is 'too hard', compounded by toolkits which try to abstract away under-the-covers detail as much as possible (Brailsford (2014, §2), though see Segal (2008) for an alternative viewpoint); and (iv) the current academic system does not incentivise the production of quality software, or making it open (despite reproducibility being a cornerstone of science). All these aspects are beginning to change, and progress tends to be more advanced in areas where the research is more universally computationally intense (e.g., bioinformatics), and where lauded science has been shown to be flawed due to software errors (Wilson *et al.* 2014, p.1).

In relation to this, Grimm & Railsback (2005) state in their simulation textbook that "software tools and technologies are themselves complex and adaptive, and different technologies are best for different [models...] We cannot make this a software engineering book, and if we did it would likely be out of date by the time you read it." Whilst I have sympathy with this (and their general outline of important software engineering ideas is good), this paper contends that, whilst technologies may rapidly change, good *design patterns* tend not to and can be usefully taught.

Remember that we are not talking about the software development *process* here, where there are perhaps more legitimate arguments that some forms of research require a more lightweight, ad hoc process (Segal 2008), though I would argue that these are *also* variants of established SE best-practice in agile methodologies (i.e., there is nothing special about scientific research as a software development domain).

6.2 Cultural Issues in Adoption

Because of the above, there are significant cultural issues to adopt these ideas but I hope that, like much other SE best-practice, the benefits tend to be self-evident to most modellers when they try to put these ideas into practice, although the cultural context will certainly affect this and empirical evidence is hard to disentangle from it—Turhan et al. (Oram & Wilson 2010, Ch.12) give a nice treatment of this with respect to TDD. One

particular problem I think should be avoided is in introducing everything by analogy and simulation-specific terminology, 'hiding' the generic SE origins and thus obscuring the interdisciplinary links. (Analogy is fine to *complement* the type of SE-driven exposition in this paper.)

Gürcan *et al.* (2013) and North & Macal (2014) are both somewhat guilty of this. Gürcan *et al.* (2013) introduce a testing framework, but talk about micro-, meso- and macro-level testing (social theory ideas) instead of unit, component and system testing (SE terms) which is *exactly* what they are; there is no extension to, or change in, the ideas for the ABM context. North & Macal (2014) talk about product and process patterns for ABM, and *do* directly talk about SE concepts (especially design patterns). However, their patterns are couched in simulation-specific terms, with many not making clear how they are related to SE concepts: for example, their "step-by-step" is a direct application of SE incremental development, and their idea to treat model validation as a court case also fails to make clear how much of this is an analogy for existing SE best-practice, and how much is a simulation-specific innovation.

7 Acknowledgements

The author wishes to thank Jason Noble for useful discussion and support. Joe Viana was also very patient in allowing the JSIT extensions to be used in the case study model! This work was conducted under the EPSRC-funded Care Life Cycle (CLC) project, which is EPSRC grant EP/H021698/1.

Appendices

A Appendix A: Alternative Reference Architectures

Since any reference architecture is a subjective assessment of how to partition functionality for an application domain, it is worthwhile briefly comparing the reference architecture given here with some partial or full alternatives.

A.1 Model Instance as a Domain Model

Conceptually, it may feel that the domain model separation in figure 1 is not quite right, because the particular domain model for the experiment is also defined by its inputs (part of the Experiments Definition layer). In

ABM, there is also often a separation between the domain model components (agents) and their composition into a model; for example, Repast Simphony (North *et al.* 2013) uses "contexts" for this purpose. Thus, figure 10 might seem better in this regard.

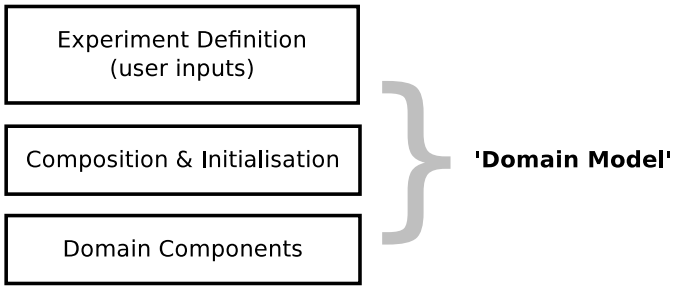

Figure 10: *An alternative way to view the constituents of a 'domain model'.*

However, figure 1 is preferable for two reasons.

Firstly, from a software perspective, the Experiment Definition layer has no additional domain knowledge. The Domain Model layer defines the set of model parameters; the Experiment Definition layer just gives them values for experiments. Setting these values requires domain knowledge, but that is something outside the context of the simulation as a piece of software. Conceptually it still makes sense: the Domain Model represents the domain model *type*, not the specific parametrised *instance* used for an experiment.

Secondly, separating out composition makes little sense for DES and SD models because the components are typically not really meaningful on their own: the particular composition *is* the model. (This is partially true in ABMs anyway, since agent types are often tightly coupled with each other, meaning that one type may not make sense without the other.) Equally relevantly, this composition really reflects what happens in all object-oriented code: there has to be some top-level object which composes a set of objects. Thus, one should really define such sub-layers for Meta-Data Capture and State & Control Presentation as well, but these separation do not really add anything useful for the purposes of best-practice design. (The 'separation' exists however one codes the simulation.)

A.2 JAMES II

JAMES II (Himmelspach & Uhrmacher 2007) is a simulation toolkit which tries to provide a generic, loosely-coupled framework for simulations of *any* paradigm by (a) abstracting common infrastructural features; and (b) separating models from simulators, where simulators can encapsulate the

building blocks of a paradigm, and models represent the choices and configuration of these blocks into a model. (A library of simulators exists, and users choose or create one to represent the 'behavioural toolkit' for a given model.) Thus, their JAMES II architecture (figure 11) has a similar role to my reference architecture.

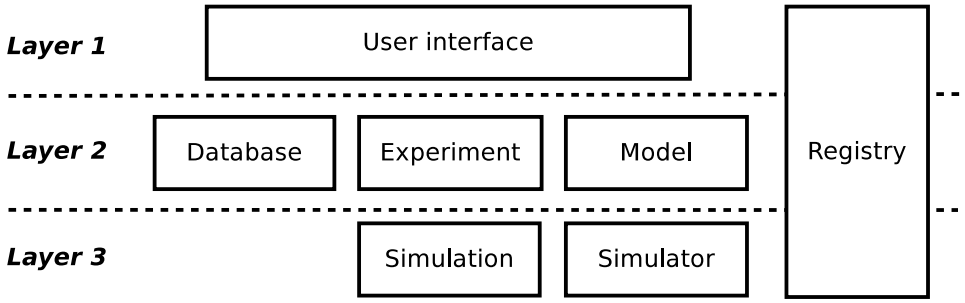

Figure 11: *The generic architecture of JAMES II, as presented in Himmelspach & Uhrmacher (2007).*

We can note some similarities, such as the separation of user interface from meta-data capture and experiment, though the meta-data capture—in the form of the Database component—applies only for the data itself, not for any capturing objects. However, they also separate simulation (a single model run) from experiment (a set of runs) and, aligning with DEVS theory (Zeigler *et al.* 2000, §2), model from simulator. Since the purpose is for a generic framework with 'pluggable' components (potentially replaceable at run-time) including simulators, there is also a cross-cutting registry component.

There is definitely some common ground worth synthesising, which I hope to do in future; I just note some relevant points for now. Firstly, toolkits such as AnyLogic can be viewed as simulators from JAMES II's perspective, and it seems would need to be encapsulated as such. However, the object-oriented extensibility of toolkits like AnyLogic sits a bit awkwardly here because models are not clean object-based-style compositions of pre-built elements (such as the microsimulation models on the MicMac project (Zinn *et al.* 2013) using a custom-designed JAMES II simulator MicCore). Secondly, other than the registry, they do not explicitly include any cross-cutting aspects in their architecture; since they are focused more on the pluggable framework, they seem less concerned with 'pragmatic' SE best-practice than I am. (Basically, the normative aspects of their architecture differ in focus and granularity to mine.)

B Appendix B: Example JSIT Outputs from the Case Study

To make things more concrete, below are some extracts from the case study events log, diagnostic log, and run-reproducibility settings file. Figure 12 shows the events log. Notice the simulated day and time in the header for each event, and how it reads as a 'narrative' of the events. The inclusion of details (such as the length of stay at the end of each appointment) allows the log to be used for automated testing. Figure 13 shows the diagnostics log. Note how this has a more 'programmer-friendly' message header (with thread ID, class producing the message and detail level). Typically, INFO is the default message level and is used for messages we expect the user to generally be interested in (such as tracking each day or week of processing). In this case, one particular component (the appointment scheduler) had been set to maximum TRACE detail level, so as to debug a potential problem there whilst suppressing detail elsewhere. There are also messages near the top relating to the run-reproducibility functionality. Figure 14 shows the settings file. Notice how it includes environmental information, model parameters, and the stochastic items (typically probability distributions) used.

```
DAY 1 05:10 AMD Sufferer #1 had an AMD_INITIAL appointment booked for day 2
DAY 1 05:10 Created referred AMD Sufferer #1: FEMALE aged 71
DAY 1 09:00 Non-AMD Patient #4 started appointment type OTHER without FFA or OCT
DAY 1 09:01 Non-AMD Patient #1 started appointment type OTHER without FFA or OCT
DAY 1 09:46 Non-AMD Patient #4 completed their OTHER appointment: length of stay 46, total wait time 0
DAY 1 09:57 Non-AMD Patient #5 started appointment type OTHER without FFA or OCT
DAY 2 09:00 AMD Sufferer #1 started appointment type AMD_INITIAL
DAY 2 09:15 Non-AMD Patient #11 started appointment type OTHER with FFA
DAY 3 09:42 AMD Sufferer #1 had AMD injection(s) in affected eye(s)
DAY 3 09:53 AMD Sufferer #1 completed their AMD_INJECTION appointment: length of stay 53, total wait time 0
DAY 3 10:20 Non-AMD Patient #13 started appointment type OTHER with FFA
DAY 3 10:26 Non-AMD Patient #16 completed their OTHER appointment: length of stay 71, total wait time 0
DAY 3 10:47 Non-AMD Patient #17 started appointment type OTHER with FFA and OCT
DAY 3 11:03 Non-AMD Patient #19 started appointment type OTHER without FFA or OCT
DAY 3 11:08 Non-AMD Patient #14 started appointment type OTHER with OCT
DAY 3 12:15 Non-AMD Patient #19 completed their OTHER appointment: length of stay 71, total wait time 0
DAY 35 00:58 AMD Sufferer #34 has died aged 96
DAY 72 05:27 AMD Sufferer #3 changed AMD level from EARLY to INTERMEDIATE
DAY 91 01:00 AMD Sufferer #2 changed care need from CRITICAL to MODERATE
DAY 91 01:00 AMD Sufferer #3 changed care need from NONE to CRITICAL
DAY 91 01:00 AMD Sufferer #4 changed care need from NONE to MODERATE
DAY 91 01:00 AMD Sufferer #4 changed care provision from NONE to INFORMAL_ONLY
```

Figure 12: *Sample events log output (extracts) from the case study model using the JSIT library.*

```
THREAD 19 MODEL INIT ModelInitialiser INFO ******** Per-Run Model Setup ***********
THREAD 19 MODEL INIT ModelInitialiser INFO Running model version 0.1r233 (with local modifications)
THREAD 19 MODEL INIT ModelInitialiser INFO Model repository source https://svn.soton.ac.uk/CLC_Project/
    Simulations/AMD/Trunk/Code
THREAD 19 MODEL INIT ModelInitialiser INFO Run ID 20140903163311-FullViz_IC_Synthesis-1 calculated by
    Main_ModelFullViz and used as outputs folder name
THREAD 19 MODEL INIT ModelInitialiser INFO Using normal stochasticity (no overrides)
THREAD 19 MODEL INIT ModelInitialiser INFO ModelCore.MaleAMD stochastic item set up for run ID 20140903163311-
    FullViz_IC_Synthesis-1
THREAD 19 MODEL INIT ModelInitialiser INFO ModelCore.AgeIn10YearRange stochastic item set up for run ID
    20140903163311-FullViz_IC_Synthesis-1
THREAD 19 MODEL INIT Main_Model INFO AMD model run launched from experiment FullViz_IC_Synthesis (running to
    population size >= 500 on a Monday)
THREAD 19 MODEL INIT Main_Model INFO Using random seed 2 (from base seed 1)
THREAD 19 MODEL INIT ModelInitialiser INFO Written model settings to settings.xml
THREAD 19 MODEL INIT ModelCore INFO Model core starting up...
THREAD 19 MODEL INIT EyeClinicWithNetwork INFO Fixed visualisation clinic DES starting up...
THREAD 19 MODEL INIT EyeClinicAppointmentScheduler INFO Clinic appointment scheduler starting-up...
THREAD 19 MODEL INIT EyeClinicAppointmentScheduler DEBUG Adding appointment slots to end of week from weekday 2
    (1=Sun)
THREAD 19 MODEL INIT EyeClinicAppointmentScheduler TRACE First non-AMD appt start point (raw delay
    1.2658953624627303) adjusted to 540.0 with 180.0 mins left of this first opening period
THREAD 19 MODEL INIT EyeClinicAppointmentScheduler TRACE Resultant next non-AMD appointment time
    541.2658953624627
THREAD 19 MODEL INIT SpatialViz INFO Spatial visualiser starting up...
THREAD 19 MODEL INIT EyeClinicFullViz INFO Eye clinic full stats visualiser starting up...
THREAD 19 MODEL INIT PopulationFullViz INFO Population-level stats visualiser starting up...
THREAD 25 SIM-TIME 0.0 DAY 1 00:00 ModelCore INFO ******** Model Execution ***********
THREAD 25 SIM-TIME 0.0 DAY 1 00:00 ModelCore INFO Start of week 1 processing
```

Figure 13: *Sample diagnostics log output (extracts) from the case study model using the JSIT library.*

```
<environmentSettings>
  <modelName>AMD Whole-System Health &amp;amp; Social Care</modelName>
  <modelVersion>0.2</modelVersion>
  <modelVCS>SVN</modelVCS>
  <modelVersionSource>https://svn.soton.ac.uk/CLC_Project/Simulations/AMD_HealthSocialCare/Trunk/AMD_Model/Code</
      modelVersionSource>
  <modelVCS__CommitID>r412</modelVCS__CommitID>
  <runtimeCodeHash>271f49ea1e112c272235b8a9c241e473</runtimeCodeHash>
  <modificationStatus>NO</modificationStatus>
  <javaVersion>1.8.0_05</javaVersion>
  <javaVM>Java HotSpot(TM) 64-Bit Server VM 25.5-b02 (Oracle Corporation)</javaVM>
  <librariesDetail>
    <libraryDetail>
      <jarName>com.anylogic.engine.jar</jarName>
      <isPartOfSimSource>false</isPartOfSimSource>
    </libraryDetail>
    <libraryDetail>
      <jarName>jsit-core-0.2-SNAPSHOT.jar</jarName>
      <isPartOfSimSource>true</isPartOfSimSource>
    </libraryDetail>
  <randomnessSettings>
    <seed>1415964022012</seed>
    <baseSeed>1415964022011</baseSeed>
  </randomnessSettings>
  <anyLogicVersion>UNKNOWN</anyLogicVersion>
</environmentSettings>
```


Figure 14: *Sample output (extracts) from the case study model settings file.*

```
<uk.ac.soton.clc.amd.testing.DES__ComponentTest>
  <parameters>
    <waitingCapacityOfDepartment>50</waitingCapacityOfDepartment>
    <receptionCapacity>10</receptionCapacity>
    <networkBasedDES>null</networkBasedDES>
    <appointmentScheduling>
      <nonAMD__AverageInterarrival>99999.0</nonAMD__AverageInterarrival>
      <nonAMD__ProbFFA>0.5</nonAMD__ProbFFA>
      <nonAMD__ProbOCT>0.5</nonAMD__ProbOCT>
    </appointmentScheduling>
    <maxWaitingTimeDist>
      <k>21</k>
      <range>
        <min>60</min>
        <max>80</max>
      </range>
    </maxWaitingTimeDist>
  </parameters>
</uk.ac.soton.clc.amd.testing.DES__ComponentTest>
<stochasticItems>
  <item id="DES_ComponentTest.MaxWaitingTime" sampleMode="COLLAPSE_MID">
    <distUniformDiscrete>
      <k>21</k>
      <range>
        <min>60</min>
        <max>80</max>
      </range>
    </distUniformDiscrete>
  </item>
</stochasticItems>
```

Figure 14: *Continued.*

References

Allan, R. J. (2010). Survey of agent-based modelling and simulation tools. Tech. Rep. DL-TR-2010-007, Science & Technologies Facilities Council (STFC), UK.

Borshchev, A. & Filippov, A. (2004). From system dynamics and discrete event to practical agent based modeling: Reasons, techniques, tools. In: *Proceedings of the 22nd International Conference of the System Dynamics Society*.

Brailsford, S. C. (2014). Discrete-event simulation is alive and kicking! *Journal of Simulation* **8**(1), 1–8.

Buschmann, F. (1996). *Pattern-Oriented Software Architecture: a system of patterns*. Wiley, volume 1 ed. URL `http://www.worldcat.org/isbn/0471958697`.

Collier, N. & Ozik, J. (2013). Test-driven agent-based simulation development. In: *Proceedings of the 2013 Winter Simulation Conference* (Pasupathy, R., Kim, S. H., Tolk, A., Hill, R. & Kuhl, M. E., eds.). IEEE.

Davison, A. P., Mattioni, M., Samarkanov, D. & Teleńczuk, B. (2014). Sumatra: A toolkit for reproducible research. In: *Implementing Reproducible Research* (Stodden, V., Leisch, F. & Peng, R. D., eds.), chap. 3. Chapman & Hall, pp. 57–78.

Djanatliev, A., Dulz, W., German, R. & Schneider, V. (2011). VERITAS—a versatile modeling environment for test-driven agile simulation. In: *Proceedings of Wintersim 2011* (Jain, S., Creasey, R. R., Himmelspach, J., White, K. P. & Fu, M., eds.). URL `http://www.informs-sim.org/wsc11papers/325.pdf`.

Evans, E. (2004). *Domain-Driven Design: tackling complexity in the heart of software*. Addison-Wesley.

Ewald, R. & Uhrmacher, A. M. (2014). SESSL: A domain-specific language for simulation experiments. *ACM Trans. Model. Comput. Simul.* **24**(2). URL `http://dx.doi.org/10.1145/2567895`.

Fowler, M. (2004). *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley, third ed. URL `http://www.worldcat.org/isbn/0321193687`.

Fowler, M., Rice, D., Foemmel, M., Hieatt, E., Mee, R. & Stafford, R. (2003). *Patterns of Enterprise Application Architecture*. Addison-Wesley.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Gilbert, N. & Troitzsch, K. G. (2005). *Simulation for the Social Scientist*. Open University Press, 2nd ed.

Green, T. R. G. & Petre, M. (1996). Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages & Computing* **7**(2), 131–174. URL http://dx.doi.org/10.1006/jvlc.1996.0009.

Grimm, V. & Railsback, S. F. (2005). *Individual-based modeling and ecology*. Princeton Series in Theoretical and Computational Biology. Princeton University Press. URL http://www.worldcat.org/isbn/0691096651.

Gulyás, L., Szabó, A., Legéndi, R., Máhr, T., Bocsi, R. & Kampis, G. (2011). Tools for large scale (distributed) agent-based computational experiments. In: *Proceedings of CSSSA 2011*. Computational Social Science Society of the Americas (CSSSA).

Gürcan, O., Dikenelli, O. & Bernon, C. (2013). A generic testing framework for agent-based simulation models. *Journal of Simulation* **7**(3), 183–201. URL http://dx.doi.org/10.1057/jos.2012.26.

Himmelspach, J. & Uhrmacher, A. M. (2007). Plug'n simulate. In: *40th Annual Simulation Symposium (ANSS'07)*. Washington, DC, USA: IEEE. URL http://dx.doi.org/10.1109/anss.2007.34.

Jeffries, R. & Melnik, G. (2007). TDD: The art of fearless programming. *IEEE Software* **24**(3), 24–30. URL http://dx.doi.org/10.1109/ms.2007.75.

Joines, J. A. & Roberts, S. D. (1999). Simulation in an object-oriented world. In: *Proceedings of the 1999 Winter Simulation Conference* (Farrington, P. A., Nembhard, H. B., Sturrock, D. T. & Evans, G. W., eds.). URL http://dx.doi.org/10.1145/324138.324178.

Kernighan, B. (1979). *UNIX for Beginners*. Bell Telephone Labs, 2nd ed.

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

McConnell, S. (2004). *Code Complete: A practical handbook of software construction*. Microsoft Press, second ed. URL http://www.worldcat.org/isbn/9780735619678.

Miller, J. H. & Page, S. E. (2007). *Complex Adaptive Systems: An Introduction to Computational Models of Social Life*. Princeton Studies in Complexity. Princeton Press.

Millington, J. D. A., O'Sullivan, D. & Perry, G. L. W. (2012). Model histories: Narrative explanation in generative simulation modelling. *Geoforum* **43**(6), 1025–1034. URL http://dx.doi.org/10.1016/j.geoforum.2012.06.017.

Mulder, J. D., van Wijk, J. J. & van Liere, R. (1999). A survey of computational steering environments. *Future Generation Computer Systems* **15**(1), 119–129. URL http://dx.doi.org/10.1016/s0167-739x(98)00047-8.

Müller, J. P. (2009). Towards a formal semantics of event-based multi-agent simulations. In: *Multi-agent Based Simulation IX*, no. 5269 in LNCS. Springer.

Nikolai, C. & Madey, G. (2009). Tools of the trade: A survey of various agent based modeling platforms. *Journal of Artificial Societies & Social Simulation (JASSS)* **12**(2), 2+. URL http://jasss.soc.surrey.ac.uk/12/2/2.html.

North, M. J., Collier, N. T., Ozik, J., Tatara, E. R., Macal, C. M., Bragen, M. & Sydelko, P. (2013). Complex adaptive systems modeling with Repast Simphony. *Complex Adaptive Systems Modeling* **1**(1), 3+. URL http://dx.doi.org/10.1186/2194-3206-1-3.

North, M. J. & Macal, C. M. (2014). Product and process patterns for agent-based modelling and simulation. *Journal of Simulation* URL http://dx.doi.org/10.1057/jos.2013.4.

Oram, A. & Wilson, G. (eds.) (2010). *Making software : what really works, and why we believe it*. O'Reilly. URL http://www.worldcat.org/isbn/9780596808327.

Railsback, S. F. & Grimm, V. (2012). *Agent-based and individual-based modeling : a practical introduction*. Princeton University Press. URL http://www.worldcat.org/isbn/9780691136745.

Railsback, S. F., Lytinen, S. L. & Jackson, S. K. (2006). Agent-based simulation platforms: review and development recommendations. *Simulation* **82**, 609–623. URL http://www.humboldt.edu/ecomodel/documents/ABMPlatformReview.pdf.

Ropella, G. E., Railsback, S. F. & Jackson, S. K. (2002). Software engineering considerations for individual-based models. *Natural Resource Modeling* **15**(1), 5–22.

Rossiter, S. (Forthcoming). Simulation design: Trans-paradigm best-practice from software engineering. *Journal of Artificial Societies & Social Simulation (JASSS)* .

Rouchier, J., Cioffi-Revilla, C., Polhill, J. G. & Takadama, K. (2008). Progress in model-to-model analysis. *Journal of Artificial Societies & Social Simulation* **11**(2), 8. URL `http://jasss.soc.surrey.ac.uk/11/2/8.html`.

Sandve, G. K., Nekrutenko, A., Taylor, J. & Hovig, E. (2013). Ten simple rules for reproducible computational research. *PLoS Comput Biol* **9**(10), e1003285+. URL `http://dx.doi.org/10.1371/journal.pcbi.1003285`.

Segal, J. (2008). Scientists and software engineers: A tale of two cultures. In: *Proceedings of the Psychology of Programming Interest Group PPIG 08*.

Sommerville, I. (2011). *Software engineering*. Pearson, 9th ed. URL `http://www.worldcat.org/isbn/9780137053469`.

Stodden, V., Donoho, D., Fomel, S., Freidlander, M. P., Gerstein, M., LeVeque, R., Mitchell, I., Larrimore Ouellette, L. & Wiggins, C. (2010). Reproducible research: Addressing the need for data and code sharing in computational science. *Computing in Science & Engineering* **12**(5), 8–13. URL `http://dx.doi.org/10.1109/mcse.2010.113`.

Stodden, V., Guo, P. & Ma, Z. (2013). Toward reproducible computational research: An empirical analysis of data and code policy adoption by journals. *PLoS ONE* **8**(6), e67111+. URL `http://dx.doi.org/10.1371/journal.pone.0067111`.

Tisue, S. & Wilensky, U. (2004). NetLogo: Design and implementation of a multi-agent modeling environment. In: *Proceedings of Agent 2004*.

Uhrmacher, A. M. (2012). Seven pitfalls in modeling and simulation research. In: *Proceedings of the 2012 Winter Simulation Conference* (Laroque, C., Himmelspach, J., Pasupathy, R., Rose, O. & Uhrmacher, A. M., eds.).

Viana, J., Rossiter, S., Channon, A. A., Brailsford, S. C. & Lotery, A. (2012). A multi-paradigm, whole system view of health and social care

for age-related macular degeneration. In: *Proceedings of the Winter Simulation Conference*, WSC '12. Winter Simulation Conference. URL http://dl.acm.org/citation.cfm?id=2429759.2429884.

Whitley & Blackwell, A. F. (2001). Visual programming in the wild: A survey of LabVIEW programmers. *Journal of Visual Languages & Computing* **12**(4), 435–472. URL http://dx.doi.org/10.1006/jvlc.2000.0198.

Wilson, G. (2014). Software carpentry: lessons learned. *F1000Research* URL http://dx.doi.org/10.12688/f1000research.3-62.v1.

Wilson, G., Aruliah, D. A., Brown, C. T., Chue Hong, N. P., Davis, M., Guy, R. T., Haddock, S. H. D., Huff, K. D., Mitchell, I. M., Plumbley, M. D., Waugh, B., White, E. P. & Wilson, P. (2014). Best practices for scientific computing. *PLoS Biol* **12**(1), e1001745+. URL http://dx.doi.org/10.1371/journal.pbio.1001745.

Zeigler, B. P., Gon Kim, T. & Praehofer, H. (2000). *Theory of modeling and simulation : integrating discrete event and continuous complex dynamic systems*. Academic Press, 2nd ed.

Zinn, S., Himmelspach, J., Uhrmacher, A. M. & Gampe, J. (2013). Building Mic-Core, a specialized M&S software to simulate multi-state demographic micro models, based on JAMES II, a general M&S framework. *Journal of Artificial Societies and Social Simulation* **16**(3), 5. URL http://jasss.soc.surrey.ac.uk/16/3/5.html.