ENVIRONMENT ORIENTED SIMULATION

Tim Hoverd

Submitted for the degree of PhD
University of York
Department of Computer Science
September, 2011

Abstract

Complex systems are collections of independent agents interacting to as to produce emergent, often unexpected, behaviour. Computer based simulation is one of the main ways of studying complex systems and a naïve approach to such simulation is fraught with difficulty due to the scope for deadlock in various patterns of interaction between the agents which are of necessity sharing aspects of the computational platform.

Agent behaviour, though, can be entirely looked at from the point of view of the environments within which the agents interact. Structuring a simulation purely in this manner leads to a simulation that has essentially no tendency to deadlock and still behaves in the manner required.

A number of experiments are conducted to demonstrate the feasibility of this approach. These start with a simple flocking system and continue through an investigation of the ways in which multiple environments can best be combined. Finally, a larger scale experiment investigating the evolution of variety in a rich environment shows that interesting results can be obtained of a simulation constructed in this manner.

Contents

List of Figures

Acknowledgments

This research is supported by Microsoft Research.

My thanks to my supervisor Susan Stepney of the University of York for her extensive guidance and a great deal of thought-provoking discussion.

My thanks to the various members of the CoSMoS[1] project, and in particular Paul Andrews and Adam Sampson, for their continued good humour and many stimulating discussions.

Thanks to Chris Alexander for the discussions that led to the contents of chapter 2.

Most importantly, my thanks to Anthea for pretty much everything else...

The work described here is part of the CoSMoS project, funded by EPSRC grant EP/E053505/1 and a Microsoft Research Europe PhD studentship.

[1]http://www.cosmos-research.org.

Declaration

I confirm that the contents of this thesis are the results of my own work, except where otherwise stated. Much of the contents of this thesis is based on papers published during the prior period of research.

The material in chapter 2 is expanded from that in a paper *"Formalising harmony seeking rules of morphogenesis"* published at ALife 12: The Twelfth International Conference on the Simulation and Synthesis of Living Systems [HS10b].

The material in chapter 3 follows material published at the 2009 CoSMoS workshop [HS09] as *"Environment orientation: an architecture for simulating complex systems"*.

The contents of chapter 4 includes much that is derived from that previously published as *"A transactional architecture for simulation"* in the proceedings of the Fifteenth IEEE International Conference on Engineering of Complex Computer Systems: [HS10a].

The contents of chapter 8 are derived from that published as *"Energy as a driver of diversity in open-ended evolution"* at The European Conference on Artificial Life (ECAL) 2011 [HS11].

Chapter 1

Complex Systems

This thesis discusses a particular approach to simulation of complex systems, one that is oriented around the environments within which the complex system's agents exist rather than the communication amongst the agents.

A simple definition of "complex system" is hard to find. Nonetheless, we are surrounded by complex systems: biological structures rely on such systems for their behaviour; the built environment influences the behaviour of a collection of humans and complex engineered systems, such as the electrical power generation and distribution system, often behave in a manner unexpected by their designers [Wike, YP09] as a consequence of their underlying complexity and the context in which they perform.

Complex systems of the sort described here, and in the rest of this thesis, are in essence a large collection of independent agents with no overall control mechanism. Each agent interacts with other agents, within a multi-faceted environment. Each agent follows a relatively simple process that is described purely in the context of the agent itself. A cell that is part of the human immune system interacts purely within its immediate biochemical context with no overall controlling context; a human in a modern city moves around and interacts with the other entities in the city without a separate controlling mechanism. In a modern power delivery system, power generation facilities—be they fossil fueled, wind turbines or photo-voltaic arrays—interact with the main electricity grid, ignorant of the fact that that grid is itself a product of many power generation facilities.

In all these cases the overall emergent properties of the collection of agents—be it immunity from disease, a stock market crash or the maintenance of 230V/50Hz domestic power—are those that characterise the success or failure of the system. These gross properties arise from the interactions between all the agents in a system yet may be difficult to predict from a detailed knowledge of the system structure. This applies even if all interactions are the consequence of an engineering process rather than being the results of the twists and turns of natural selection.

Design or analysis of such a complex system can be difficult which leads directly to the notion of explicit simulation of such systems. This thesis is in essence concerned with software-based explicit simulation of complex systems where those simulations follow the approach described here as environment orientation.

A complex system is characterised by a potentially huge number of interacting agents, each of which takes decisions and performs actions independently. Each agent may well observe other agents and respond to them but this is done so independently, without any overall control. An immune system may contain many millions of active cells and even engineered systems, such as a power generation system, could contain millions of photo-voltaic generators, load-shedding units, power transmission lines and wind power generators.

The actions that take place inside a complex system are, though, restrained by the physics of the real world; a flocking bird cannot occupy the same location in space as another even if it decides it wants to. A software simulation of a complex system would ideally have the same vast number of agents making analogous decisions to those in the real world and limited by the same real world physics. That is, it would execute in a hugely parallel computational context within an environment that replicated the physics of the real world.

However, any realistic simulation must execute on real-world computer hardware and such a complete simulation is not currently feasible. Computer hardware is continually improving in performance but does not approach that necessary to provide a separate computational core for each agent in a complex systems simulation. In any currently foreseeable computer system, the hardware supports a limited number of parallel execution threads which is in itself a significantly smaller number than the number of independent agents in a complex system. Even if the approach to implementation of the complex system provides an *apparently* unbounded number of execution threads, such as when using a multi-threaded or process-oriented programming language, the real-world effects of resource contention between those processes and threads looms large. Any attempt in this sort of computational framework to implement complex system agents in a naïve, apparently realistic, manner quickly leads to gross effects such as deadlock and livelock that are not seen in the real world. As a consequence programming of such systems must necessarily involve sophisticated strategies, such as using the patterns defined in [Sam10].

1.1 Complex systems constituents

Complex systems are routinely studied, and modeled, for a number of reasons and [Eps08] identifies a number of them including prediction of their behaviour, demonstrating trade offs and revealing the apparently simple to be complex. In this thesis, though, I take it as read that the study of complex systems is useful and limit myself to the computer based simulation of complex systems as being one of the vehicles that supports their study. In particular I concentrate on the underlying software architecture of such a simulation.

Real complex systems contain a huge number of agents; bird flocks may contain millions of agents [Ogb61], the human nervous system alone contains in excess of 10^{11} agents [ACG+09].

The agents in a complex systems simulation are, though, only part of the story. The other part is the environment within which those simulated components exist. That is, that part of the simulation that is analogous to, for example, the biochemical environment occupied by an immune system cell or the city occupied by people. It is easy to ignore this environment, seduced by the attraction of the obviously interesting agents. But, some consideration shows that this is rather inadequate. It is the environment, or at least its physics, that prevents two flocking birds occupying the same space. The environment, or again its physics, governs the rate of decay of an ant-laid pheromone trail; the levels of concentration of biochemicals in the environment around immune system cells strongly influences their behaviour.

It is these environments that are the principal focus of the rest of this thesis. In particular, it proposes an approach to simulation here called “environment orientation”. This approach makes the environments the central feature of complex systems simulation rather than the interaction between the simulation agents. Such environments both mediate inter-agent communication and represent the physics of the real world that implements that communication. They form the central simulation paradigm not just the backdrop to the world of interacting agents.

Such an approach makes no claims as to any innate correctness with respect to the complex systems themselves. It is rather essentially an engineering approach to the difficulties of building a simulation using a small number, in comparison to the number of complex system agents, of computing cores.

1.2 Software engineering of simulations

Simulation of such systems poses huge computational challenges. Those challenges are essentially those of *engineering* a simulation that will be able to provide useful results in a sufficiently timely manner. Writing a complex systems simulation of a system with a few tens or hundreds of homogeneous agents is likely to be quite simple. Doing the same thing for millions or billions of potentially heterogeneous agents will likely not submit to the same implementation strategy; in particular because the "natural" way to write such a simulation by concentrating on the communication between the agents—which in itself immediately makes the problem of size $O(N^2)$—almost inevitably leads to an implementation that focuses hugely on avoiding deadlock as all those agents struggle to execute on a relatively small number of processor cores. In the complex system itself each agent has its own identity and functions independently of all other agents. Such an approach cannot be adopted for a simulation as no computing system can hope, at any foreseeable level of technology, to provide a completely independent computing platform for each individual agent. Rather, those agents must share computational resources and the approach to how that sharing is done is essentially one of how the simulation is engineered.

This issue of the engineering of a useful simulation of a potentially large complex system including all its agents and the environments within which they interact is the central topic of the rest of this thesis.

1.3 Thesis structure

The rest of this thesis details the arguments for the notion of environment orientation as a simulation framework for complex systems and discusses the means of implementing it. In particular, the thesis investigates a single hypothesis, that **the environment oriented approach is an appropriate software paradigm for the simulation of complex systems**.

This investigation ultimately takes the form of a number of experiments, building complex systems simulations of several forms, in order to test this hypothesis.

The structure of the thesis follows the path of the research whose origins were rather different from the eventual result.

Chapter 2 examines some aspects of the development of the built environment, as described in particular in [Ale04]. This work, and the publications on which it was based, was the starting point for the rest of the research presented here. The chapter, in particular, examines a computer simulation of some aspects of the processes described in [Ale04]. This simulation led directly to the notions used in the rest of this thesis.

Having developed the notion of the relevance of environment, chapter 3 discusses the role of environments in complex systems and simulations of complex systems leading to the concept of using the environment as the primary structuring mechanism for complex systems simulations; the concept referred to here as environment orientation.

If a complex systems simulation is to be structured around the environment then there must be a software architecture to support such a simulation. Building on a review of the possible software architectures for implementing an environment oriented complex systems simulation chapter 4 discusses the software architectures appropriate to an environment oriented complex systems simulation.

Once the concept of environment orientation had been developed an implementation was necessary for performing experiments; that is, for implementing a number of complex systems simulations. Chapter 5 describes the design of such an experimental platform, following on from the potential architectures discussed in chapter 4.

Initial experiments with the platform are described in chapter 6. This chapter provides initial validation of the platform, through experimental implementations of simple complex systems including the classic “flocking boids”. This chapter represents the initial validation of the experimental hypothesis by demonstrating that it can be used:

1. to implement a well-known example of complex systems behaviour as an environment oriented simulation,
2. to implement a similar simulation, albeit one that includes the notion of multiple fields in an environment — something that may be explicitly represented in an environment oriented simulation — and
3. to implement a simulation which includes an external environment representing the landscape within which the agents are moving.

Combining the effects of multiple fields in an environment, in essence multiple environments, is difficult, specifically because the fields are potentially orthogonal to one another. Chapter 7 describes one approach, based on the use of fuzzy logic, to the combination of the effects from multiple environments in a environmentally oriented simulation.

Chapter 8 takes the various notions developed in the earlier parts of the thesis and applies them to a larger simulation; in this case the simulation of open-ended evolution in an environment which provides the energy flux used by the agents in that simulation. This demonstrates the applicability of environment orientation to implement and experiment on a non-trivial complex system.

Finally, chapter 9 summarises the core issues and discusses some possible extensions, specifically ones to support larger scale simulations, to the work described in the earlier parts of this thesis.

1.4 Thesis contributions

This thesis contributes to the study of complex systems by simulation:

- the notion of environment orientation as a generalisation of inter-agent communication,

- a demonstration of the feasibility of implementating environment orientation using existing enterprise computing techniques and
- an environment oriented model of open-ended evolution in an energy-rich environment.

1.5 CoSMoS project

The work described here has been undertaken in the context of the CoSMoS project which was established to investigate general issues in the modeling and simulation of complex systems. The work has been informed by the rest of the project and in particular by the case studies carried out on particular complex system simulations, such as [FTA+09] and [RATK09].

The work described here is also being included as one of the implementation patterns in the CoSMoS handbook [SAA+13].

Chapter 2

The built environment

Complex systems appear all around us and one example is the buildings and streets that we construct. Architects design individual buildings, planners lay out urban environments and the end result over centuries of such activity—in particular in cities—is the built environment that surrounds us. We are all familiar with these environments and their history: cities sometimes still have walls that have long since ceased to be of use, streets are known to follow ancient ditches or now subterranean rivers. The physical shape and consequences of this environment strongly influence the complex system that manifests itself as the human use of the city: small businesses thrive underneath railway arches, newspapers are still represented as being published in Fleet Street even though the river itself has long since disappeared from view. Animal life also thrives in, and because of, this environment: rats inhabit the sewer systems and rock doves—adapted to living on coastal cliffs—now infest the steep sides of city centre buildings.

These city environments *grow* and *develop*. Typically their origin is at a place that is important for commerce, such as the furthest upstream part of a river that is navigable. Once established there they develop in a manner strongly influenced by the physical environment, for example that of rivers and hills, and the use the city is put to by humans, such as a physical place where a market or fair is established.

This growth process has been studied by architects and planners. For example, and of particular interest here, Alexander's *Generative Patterns* [Ale04] are a vision of the way that successful architectural forms can be seen as the product of the generative application of a small number of properties that are seen in those forms.

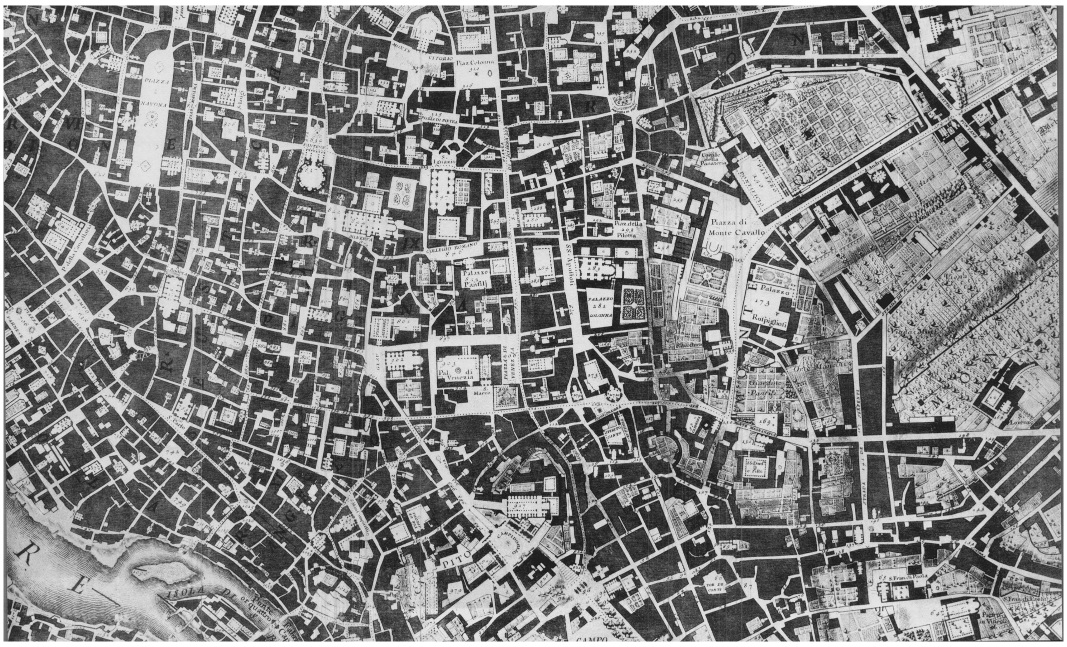


Figure 2.1: Part of the Nolli map [TS05] of 18th century Rome.

Alexander describes the built environment as a consequence of a long running developmental process. His patterns describe the way that an architectural whole, be it a house or a city, evolves as a consequence of its environment and use. For example, [Ale04] shows a diagram of 18th century Rome, shown here as figure 2.1, and discusses how that particular configuration emerged from the human use and development of the city.

This chapter describes these *Generative Patterns* and presents the results of some experiments into using the patterns to generate shapes reminiscent of architectural structures. Although these experiments remain to be completed the results, such as they are, provided the impetus for much of the work in the rest of this thesis.

2.1 The Nature of Order

The four volumes of *The Nature of Order* [Ale04] explore the notion of *wholeness* in relation to architectural structures. *Wholeness* is Alexander's enigmatic term for the "quality without a name" that he identified earlier in [Ale79]. Alexander sees "wholeness" as meaning that in some difficult to define manner, some architectural structure is perceived as being a single structure with a single overall purpose; this applies even if the structure is an entire housing development or a city. In *The Nature of Order*, he identifies 15 *generative properties* as the root characteristics of those architectural structures that form, for him, a satisfactory *whole*.

It would be reasonable to claim that this notion of "wholeness" is purely subjective and just Alexander's feelings about the world, albeit written down. However, he goes to some lengths to demonstrate that this perception of coherence is universal, and he presents some evidence to support this claim, notably the "bead game" [Ale04, vol.1, pp449–452].

Based on this experimental work, Alexander describes structures in terms of *centres*, each of which he describes as "a zone of coherence in space". A centre is a region that is in some way *coherent* in the way it represents the space and its use. By *"coherence"* Alexander means that a centre is distinct from those around it and within it, but that in some way it contributes to the coherence of those other centres. Alexander refers to these as "centres" because they are "centres of influence, centres of action, centres of other centres" [Ale04, vol.1, p108]. One particular reason for using the word "centre", although he is not using the normal notion of a single point, is that he is trying to describe things that may have no specific boundary. A pond, for example, might include the pipes bringing in water, the rocks on its edge [Ale04, vol.1, p84]. A centre is something noticeable about a structure; something that draws attention from neighbouring structures. Further examples might be [App97] a row of tiles on a ceiling or floor, a hallway, a pond in the countryside, and—in the context of software development—what are now known as "patterns" [GHJV95]. It is no coincidence that the patterns described in [GHJV95], probably the most influential published work on software design to appear in the last 25 years, are described there in a manner specifically referenced to Alexander's Pattern Language [AIS77].

The generative properties are used to describe a structure as a system of centres, and to show the ways that that structure can be further elaborated and extended, or *generated*, as a region is architecturally developed. Alexander sees this as a generative, developmental process, where the system of centres is progressively developed using the same set of generative processes with each application of these processes being dependent on the current structure. That is, an existing set of centres is the *environment* within which further centres develop and are elaborated. For example, Alexander [Ale04, vol.2, pp252–255] describes how the structure of St Mark's Square in Venice can be described as the current end product of an evolutionary process. At each step of this process, Alexander identifies *latent centres* and shows how, in his view, new building supported and strengthened these centres. That is, the latent centres are reified by subsequent development into real centres.

Inherent within Alexander's descriptions and in other communications [Ale08a] is that he regards his properties in two ways. On the one hand, they are a way of analysing an existing structure in order to determine its coherence. Secondly, he sees them as *generative* in that they direct the development of a structure over time whether that be in something like the St Mark's Square example or in [Ale08b] where he uses the properties to direct the design of a single window, applying the properties as generative transformations to a structure so that it develops in the context of its environment.

2.1.1 Generative Properties

The 15 properties are described in [Ale04, vol.1] as:

Levels of Scale *"how a centre is made stronger (more coherent) by the smaller strong centres within it and the larger strong centres that surround it."*

Positive Space *"the way that a given centre must draw its strength, in part, from the strength of other centres immediately adjacent to it in space."*

Roughness *"the way that the field effect of a given centre draws its strength, necessarily, from irregularities in the sizes, shapes and arrangements of other nearby centres"*

Alternating Repetition *"the way in which centres are strengthened when they repeat, by the insertion of other centres between the repeating ones"*

Thick Boundary *"the way in which the field-like effect of a centre is strengthened by the creation of a ring-like centre, made of smaller centres which surround and intensify the first. [It] also unites the centre with the centres beyond it, thus strengthening it further"*

Good shape *"the way that the strength of a given centre depends on its actual shape and the way this effect requires that even the shape, its boundary, and the space around it are made up of strong centres."*

Local Symmetry *"the way that the intensity of a given centre is increased by the extent to which other smaller centres that it contains are themselves arranged in locally symmetrical groups"*

Contrast *"the way that a centre is strengthened by the sharpness of the distinction between its character and the character of surrounding centres"*

Gradient *"the way in which a centre is strengthened by a global series of different-sized centres which then* ***point*** *to the new centre and intensify its field effect"*

Deep Interlock and Ambiguity *"the way in which the intensity of a given centre can be increased when it is attached to nearby strong centres, through a third set of strong centres that ambiguously belong to both"*

Echoes *"the way that the strength of a given centre depends on similarities of angle and orientation and systems of centres forming characteristic angles thus forming larger centres, among the centres it contains"*

Simplicity and Inner Calm *"the way the strength of a centre depends on its simplicity - on the process of reducing the* ***number*** *of different centres which exist in it, while increasing the* ***strength*** *of these centres to make them weigh more"*

The Void *"the way that the intensity of every centre depends on the existence of a still place - an empty centre - somewhere in its field"*

Not Separateness *"the way the life and strength of a centre depends on the extent to which that centre is merged smoothly - sometimes even indistinguishably - with the centres that form its surroundings"*

Strong Centre *"defines the way that a strong centre requires a special field-like effect, created by other centres, as the primary source of its strength"*

These 15 separate properties address the same thing: the manner in which centres interact to increase the overall coherence of the space. That is, they describe how the component centres in a structure interact with the centres that are in the surrounding environment in order to provide the overall "coherence" property. As has been described, the properties are seen in two ways: as a way of describing the relationship between a centre and its environment and describing how the structure should be developed to enhance the overall coherence.

My intention for originally examining these properties was to investigate their application, if any, to the evolutionary development of software architectures, particularly that of complex systems simulations. As such, I start by examining two of the properties in more detail, in order that some experimental evaluation of these properties might be performed: *Positive Space* and *Levels of Scale*.

2.1.2 Positive Space

"Negative space" is a term used to describe the space around a artistic representation; that is, the *ground* around the *figure*. Typically, an artist "relies on the space that surrounds the subject to provide shape and meaning" [Bar09]. Betty Edwards gives a vibrant illustration of the role of negative space in art [Edw99]:

> ...imagine Bugs Bunny speeding along and running through a door. What you'll see in the cartoon is a door with a bunny-shaped hole in it. What's left of the door is the negative space, that is the space around the object, in this case Bugs Bunny.

In contrast, "Positive Space" is conventionally used to describe *"space that is occupied by a filled shape or a positive form"* [Won93]. The positive space is the figure at the centre of attention; it is the part of the figure that the eye sees. In this sense positive space is in contrast with the negative space that surrounds the positive; it is the "figure" not the "ground".

For Alexander, *Positive Space* is that space which, although the space between other parts of a structure, itself contributes towards the "wholeness". That is, if the structure represents a coherent whole, then the space between the built artefacts is itself (also) positive in that it contributes to the overall coherence rather than just being the (negative) space between those artefacts. So the figure *and* the ground are both positive, in a coherent whole. In the context of the development of the built

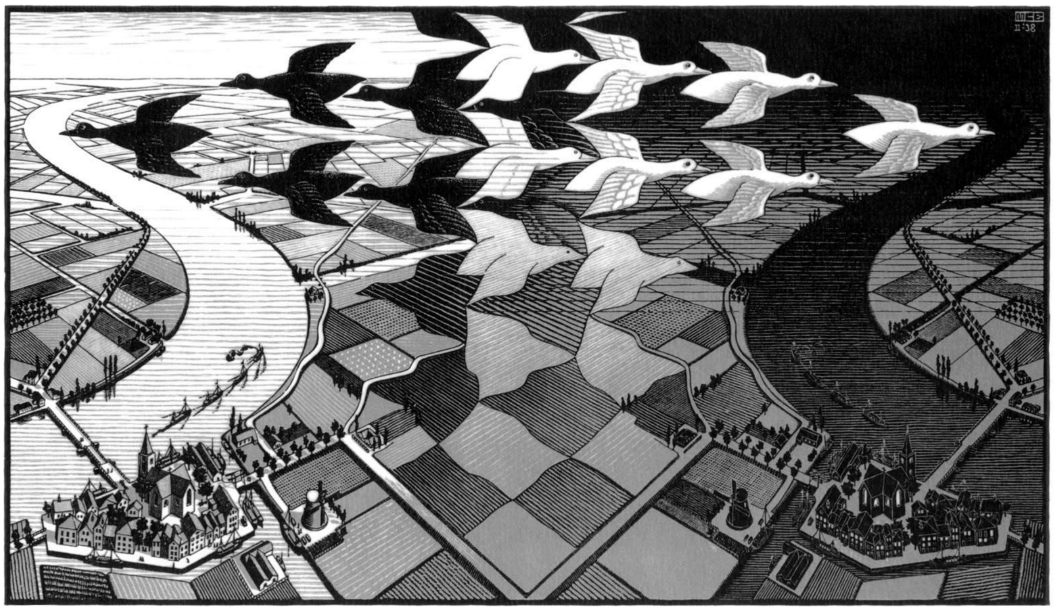


Figure 2.2: M. C. Escher: *Day and Night*.

environment this space could be, for example, the areas between the buildings that were the main concentration of individual developments. So, a market square or piazza could be the space between buildings that is itself *positive*.

An extreme example of the potential equivalence of figure and ground is the Escher wood-cut "Day and Night" [Esc38] shown here as figure 2.2; the space between flying geese is yet more geese, albeit heading in the opposite direction. That is, the "space" has its own positive structure. The same relationship appears in non-spatial examples, too. For example, Tsur shows how the same concepts occur in areas such as music and poetry [Tsu00].

2.1.3 Levels of Scale

Centres, the structural components of the architectural space, are made more "coherent" by the presence of both larger and smaller centres in the overall structure. A particular architectural space is overall more coherent if the various structures, and indeed the non-structures that are the *Positive Space* display a degree of gradation in their sizes. For example, a large structure placed next to a collection of smaller structures might represent an overall structure that was more "whole".

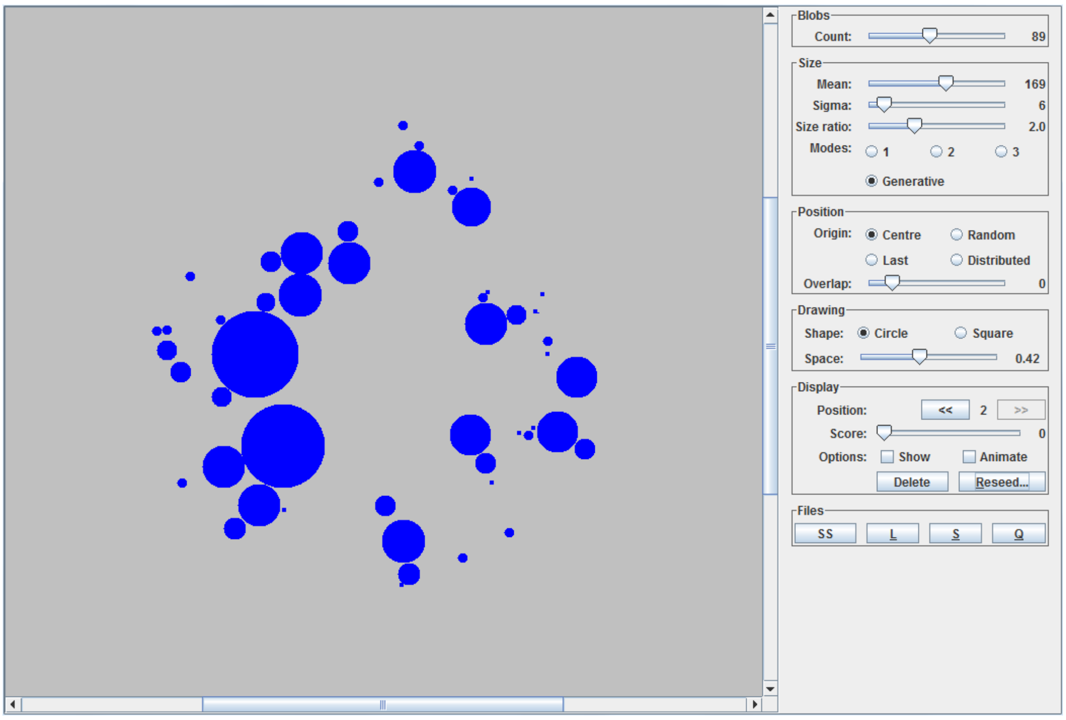
If the changes in scale are too extreme the centres would not be seen as increasing each other's coherence. Alexander shows how coherent structures often contain a number of levels of scale in the ratio of about 3:1 [Ale04, vol.1]. The same ratio appears elsewhere; Salingaros shows levels of scale in the centres of a carpet design which appear in the ratio 3:1 over eight levels of scale [Sal95].

2.2 BlobWorld: Exploring two properties

This all leads to a simple hypothesis that is amenable to experiment: it should be possible to construct diagrams that are subjectively coherent by the repeated application of simple rules that are an implementation of the notions of *Positive Space* and *Levels of Scale*. Although, it must in advance be admitted that the end notion of coherence is subjective, so any experiment must necessarily involve a subjective scoring process.

To this end, the *Blobworld* software was written. The intent of this was to explore the effects of algorithms whose intention is to generate spatial arrangements that might, or might not, exhibit *Positive Space* and *Levels of Scale*. Blobworld is very simple, it just places blobs—simple round or square shapes—at various sizes and positions in accordance with one or more placement and sizing algorithms. Figure 2.3 shows a view of the basic user interface presented by Blobworld containing one specific Blobworld diagram.

Blobworld places blobs in the environment, that is into the diagram, following the rules described by several algorithms whose various parameters are controlled by the user interface shown. The intent of these algorithms, whose definition was assisted by Chris Alexander, is that a set of simple generative rules would yield overall patterns that exhibited, as a contribution to a structure's overall wholeness, aspects of *Positive Space* and *Levels of Scale*. That is, any such *Positive Space*, for example, will appear as a consequence of the particular algorithm being invoked in the particular environment that has been generated within the space controlled by Blobworld.



Blobs
Count: 89

Size
Mean: 169
Sigma: 6
Size ratio: 2.0
Modes: 1 2 3 [✓ Generative]

Position
Origin: [✓ Centre] Random Last Distributed
Overlap: 0

Drawing
Shape: [✓ Circle] Square
Space: 0.42

Display
Position: 2
Score: 0
Options: Show Animate

Figure 2.3: Overview of *blobworld* user interface.

2.2.1 Blobworld implementation

Blobworld is implemented using the Java programming language and using the JetBrains IntelliJ IDEA programming environment. The user interface is constructed using the user interface designer aspect of that tool which is implemented using the Swing [ELW02] user interface classes as well as the various adapter classes that interface between native Swing and the IDEA classes.

Like all Swing interfaces the user interface itself uses a single thread of execution with the programmer being required to manage things so that this thread does not become blocked which would lead to a non-responsive user interface. The Blobworld code is fairly simple and as such it all executes within the Swing thread itself. That is, whenever a button is pressed or some other control is used its effect is directly passed to the underlying blob code and the diagram implied by the user's change is directly drawn. Of necessity with this design, the blob are placed on the diagram serially. The algorithms in subsequent sections are specified in this context.

The core design of Blobworld is shown in the UML [FS03] class diagram in-

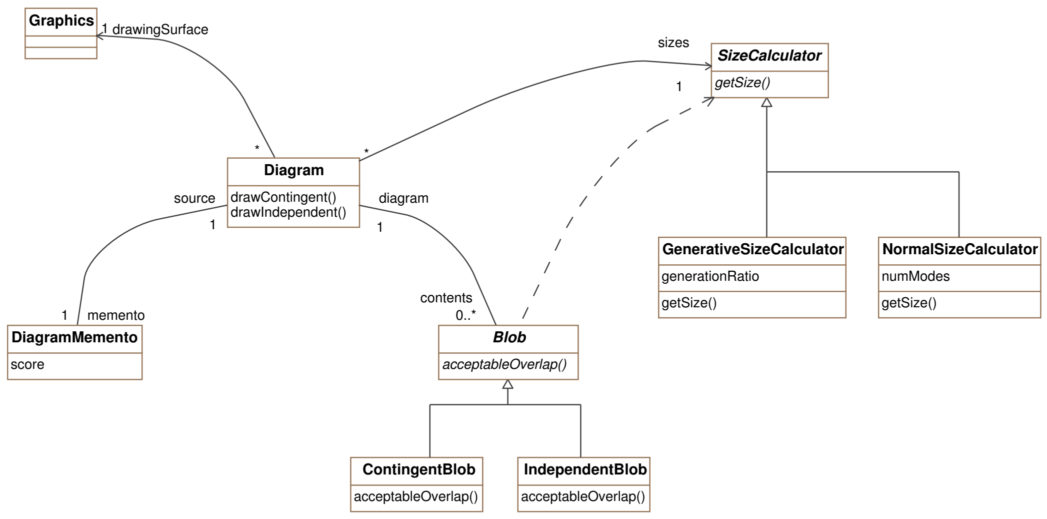


Figure 2.4: UML class diagram showing overview of *blobworld* implementation.

cluded as figure 2.4. The core of the design is the Diagram class which represents a collection of Blob objects. These objects are drawn on a Graphics object (a standard Java 2D class) for display on the screen. Diagrams are also saved as DiagramMemento objects (an implementation of the Memento pattern [GHJV95]) for later recall and redisplay.

The diagram class implements two operations for drawing different sort of diagrams, corresponding to two different layout algorithms which will be discussed in the sections 2.2.2 and 2.2.3. Each algorithm uses one of the concrete subclasses of the abstract Blob class for the actual blobs in the diagrams.

The sizes of blobs are calculated in two different manners, in ways that will be discussed in sections 2.2.4 and 2.2.5. Two implementations of a SizeCalculator class are provided to allow for these two algorithms.

2.2.2 Contingent Placement Algorithm

The first algorithm, *contingent placement*, attempts to produce emergent *Positive Space* by making the placement of blobs a reflection of the positions of other blobs in the environment; that is, so that blobs become other blobs' *Positive Space*. This algorithm was developed to explore a specific hypothesis, suggested by Chris Alexander, which is that the level of *Positive Space* in a diagram would increase when the positioning of the blobs was controlled by the positions of other blobs, some of which were in fact invisible. That is, the space was not just negative but, positively, contained blobs.

The intent of this notion is that the spaces in the diagram are themselves actually structures constructed using the same rules that apply to other structures; it's just that they aren't visible in the end result.

The algorithm works by placing blobs sequentially onto the display. Initially a position is chosen for a blob which can be one of: the centre of the space, the position of the last plotted blob or a random position within the space. However, blobs are only allowed to be plotted in places that satisfy a rule about allowable overlap. This rule says either that a blob cannot overlap another in any way, or that it is allowed to overlap by a specific amount, be it positive or negative. (A negative overlap implies a minimum gap between blobs.)

If a blob can be plotted in the position chosen, that is if it satisfies the overlap rule, then the software draws the blob and starts on the next one. If, as is likely, the blob cannot be accommodated in the initial location then a random direction is chosen within the 2D space that is the drawing surface. The blob is moved in that direction until a location is found where it is allowable, according to the overlap rules, to plot the blob.

The size of the blobs is determined by parameters that describe a size distribution. This process will be described later in this chapter.

This, though, is just an algorithm that packs blobs in a more or less efficient manner. The aspect of the algorithm that is hypothesised to produce the effects of positive space is that a blob may actually be invisible, although its position still affects the position of other blobs. That is, an invisible blob is still part of the environment of new blobs and as such affects the placement of those new blobs; there really is something in the space, something positive.

As the placement of blobs is dependent on the position of pre-existing blobs I refer to this as the *contingent placement algorithm*. Pseudo code for the algorithm is given in figure 2.5, in which:

blobShape is "round" or "square".

sizePDF is the probability distribution function (pdf) used to generate blob sizes

```
blob[0] := new Blob(blobShape)
blob[0].setSize(sizePDF)
blob[0].setVis(boolean according to visProb)
blob[0].setPosition(origin)
blob[0].draw()
for i = 1..blobCount-1 do
    blob[i] := new Blob(blobShape)
    blob[i].setSize(sizePDF)
    blob[i].setVis(boolean according to visProb)
    blob[i].setPosition{blobs[0].getPosition()
            | blobs[i-1].getPosition()
            | blobs[random(0..i-1)].getPosition()}
    blob[i].setDirection(rand in 0 ... 360°)
    while not blob[i].isOverlapAcceptable(
            allowedOverlap) do
        blob[i].movePositionAlongDirection()
    end while
    blob[i].draw()
end for
```

Figure 2.5: Pseudo-code for the *contingent placement* algorithm.

(see later).

visProb is the probability of a blob being visible.

blobCount is the total number of blobs (both visible and invisible).

allowedOverlap determines how much a blob is allowed to overlap other blobs: when positive, blobs may overlap by an amount determined by this parameter; when zero blobs just touch; when negative, blobs have a small amount, determined by this parameter, of clear space around them.

origin takes one of three arguments: the centre of the initial blob, or the most recently placed blob, or a random blob, to start off the current blob.

Experimentation with the algorithm shows that each run creates a unique pattern of blobs which is highly dependent on the various parameters. Examples of generated patterns are shown in figure 2.6.

All of the diagrams shown here, and many other produced using the software, were scored by Chris Alexander[1] for the presence of positive space. (Inspection of the user interface shown in Figure 2.3 shows the interface used for adding subjective scores to specific diagrams.)

Informally, the results of this scoring were such that, with appropriate parameter choices it is capable of generating patterns that display a significant degree of *Positive Space*. It is, though, a subjective view although nonetheless an appealing one; many viewers of the diagrams report the same effect.

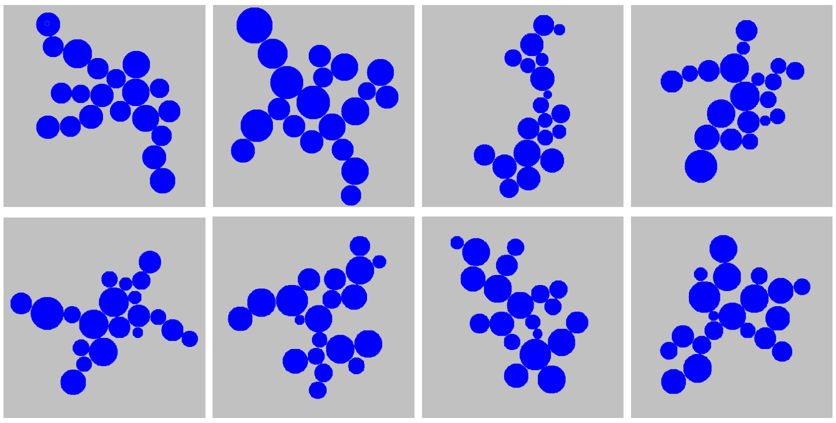
In most cases where **vis** = 1, (that is, where all blobs are always visible) the generated patterns show no significant degree of *Positive Space* (for example, figure 2.6a where the space is nothing more than a lack of blobs; it is ordinary "negative space").

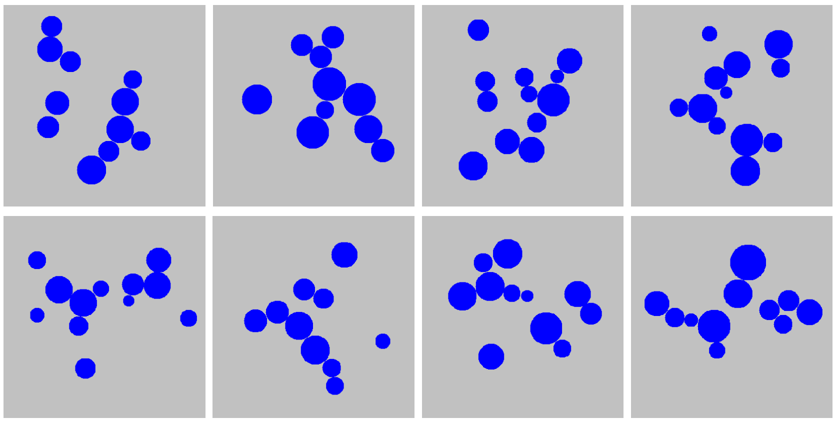
The algorithm is more successful, as hypothesised, at generating *Positive Space* when some blobs are invisible (for example, figure 2.6b). The invisible blobs generate additional space, which enables the appearance of *Positive Space*; they are still in each new blob's environment though and affect the placement of the new blobs. Figure 2.6b shows the effect of the *Positive Space* : in the leftmost of these pictures, the observer gets a powerful impression of the space itself constraining, for example, the curve of blobs at the lower right corner. In many of the diagrams generated in this manner, the *Positive Space* does not exactly align with the invisible blobs. That is, although the invisible blobs are in some way enabling the emergence of *Positive Space*, they are not themselves exactly that space (figure 2.6c where the "invisible" blobs are made visible).

This successful generation of positive space is not dependent on using round blobs. The same effects are generated with square blobs (figure 2.7). Again, without the invisible blobs there is little sign of *Positive Space* (figure 2.7a), but when invisible blobs are introduced they create *Positive Space* (figure 2.7b).

With the square blobs, a further effect is visible. Here there is a negative **allowedOverlap**, to separate the blobs from each other along their straight boundaries.

[1]At this point I should comment that the original intention was for the results of Blobworld diagrams to be subjected to a comprehensive subjective scoring process by Chris Alexander. However, it was not possible to perform this and hence the scientific conclusions of this experimentation are scant. However, the software is described here because the results, and indeed the implementation, formed the impetus for much of what follows.


(a)


(b)

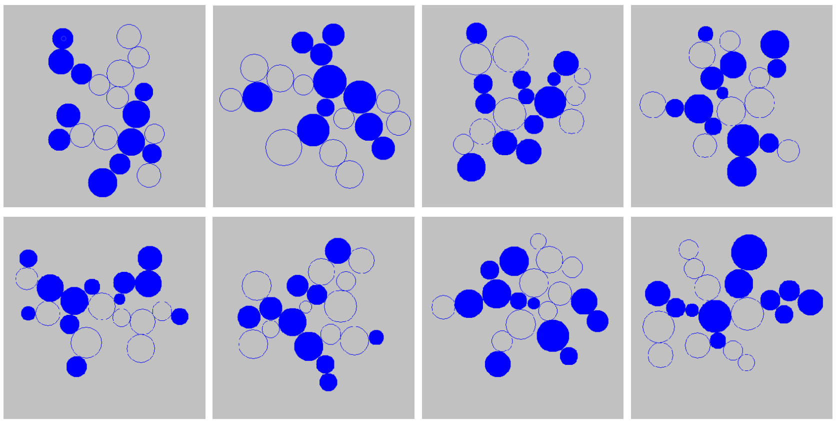
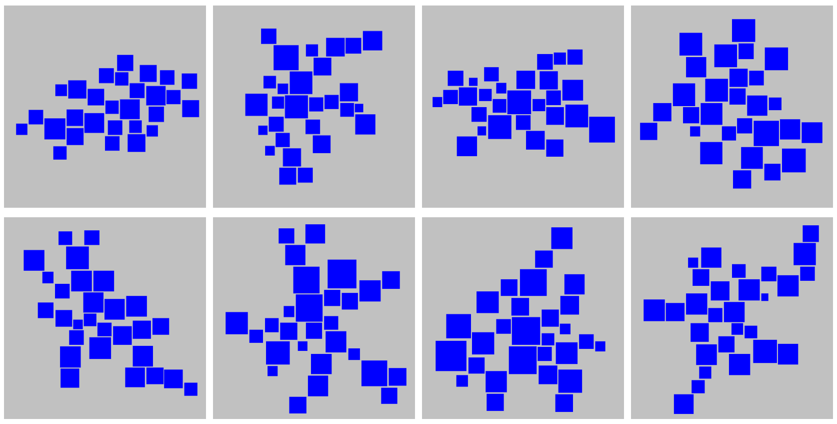
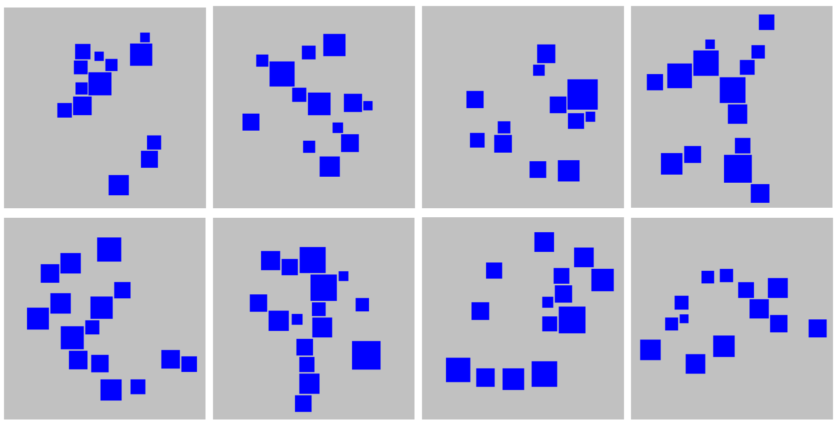

(c)

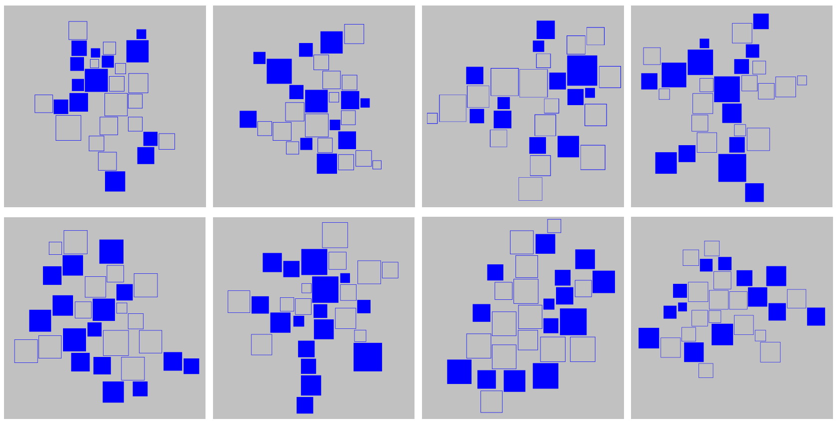
Figure 2.6: Results of the *contingent placement* algorithm with **blobCount** = 20, **sizePDF** = gaussian, **blobShape** = round, **allowedOverlap** = 0 : (a) **vis** = 1 ; (b) **vis** = 0.5 ; (c) as b, but with the position of the “invisible” blobs shown.



(a)



(b)



(c)

Figure 2.7: Results of the *contingent placement* algorithm with **blobCount** = 28, **sizePDF** = gaussian, **blobShape** = square, **allowedOverlap** < 0 : (a) **vis** = 1 ; (b) **vis** = 0.5 ; (c) as b, but with the “invisible” blobs shown.

```
for i = 0..blobCount-1 do
    blob[i] := new Blob(blobShape)
    blob[i].setGrowthRate(growthPDF)
    blob[i].setSize(1)
    blob[i].setVis(boolean according to visProb)
    blob[i].setPosition(positionPDF)
    blob[i].unfreeze()
end for
while exists an unfrozen blob do
    for i = 0..blobCount-1 do
        if blob[i] is unfrozen then
            blob[i].setSize(
                    blobs[i].getSize * blobs[i].getGrowthRate)
            blob[i].draw()
        end if
        if blob[i].overlapsOtherBlob(allowedOverlap) then
            blob[i].freeze()
        end if
    end for
end while
```

Figure 2.8: Pseudo-code for the *independent placement* algorithm.

Although the blobs are all perfectly aligned squares, an optical illusion makes some edges look slightly tilted or slightly bowed; this adds a degree of *Roughness* (another of Alexander's generative properties) to the picture.

2.2.3 Independent Placement Algorithm

In order to test whether *Positive Space* is manifested in any diagram that merely contains "invisible" blobs a second algorithm is also implemented by BlobWorld. This *independent placement* algorithm positions blobs not as a consequence of the positions of other blobs but as an initial step of the algorithm. Pseudo code for this algorithm is shown in figure 2.8. In essence, the *contingent placement* algorithm positions blobs of a pre-determined size in a field of other blobs as the diagram evolves from a single blob. In contrast, the *independent placement* places blobs entirely independently of each other but then manipulates the *size* of all of the blobs until the diagram, as a whole, achieves the stated requirements for blob overlap.

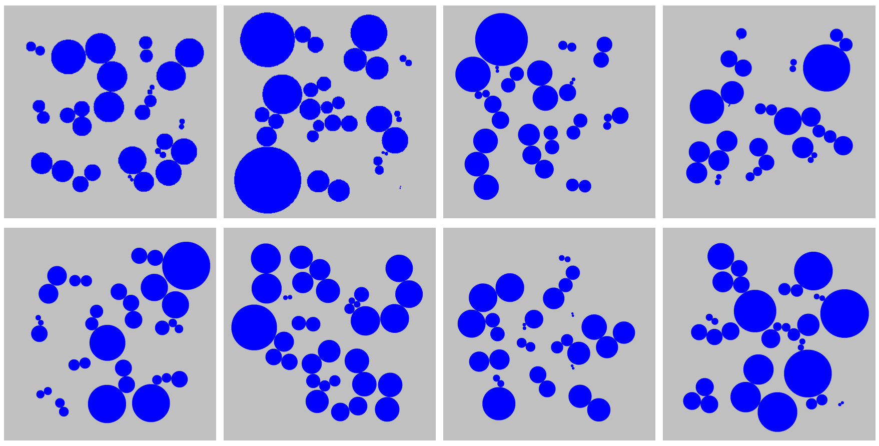


Figure 2.9: Typical results of the *independent placement* algorithm with **blobCount** = 34, **growthPDF** = gaussian, **blobShape** = round, **allowedOverlap** = 0 ; **vis** = 1.

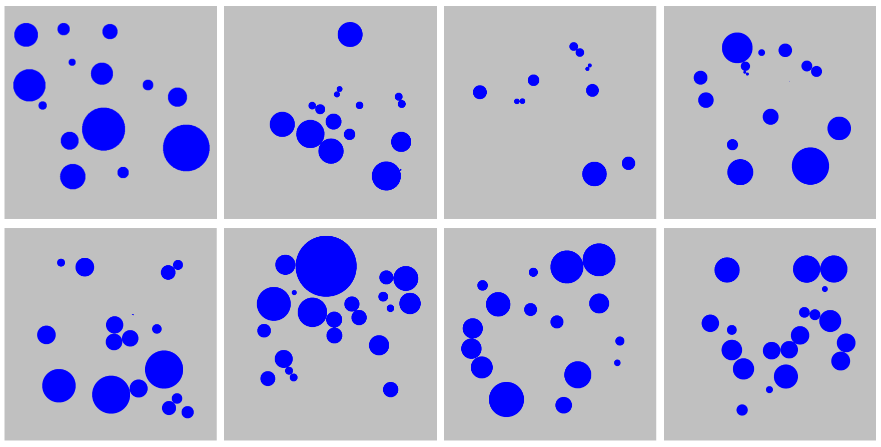
The *independent placement* algorithm is described by the pseudo-code in figure 2.8 in which:

growthPDF is the pdf used to generate the growth rate of each blob (see later).

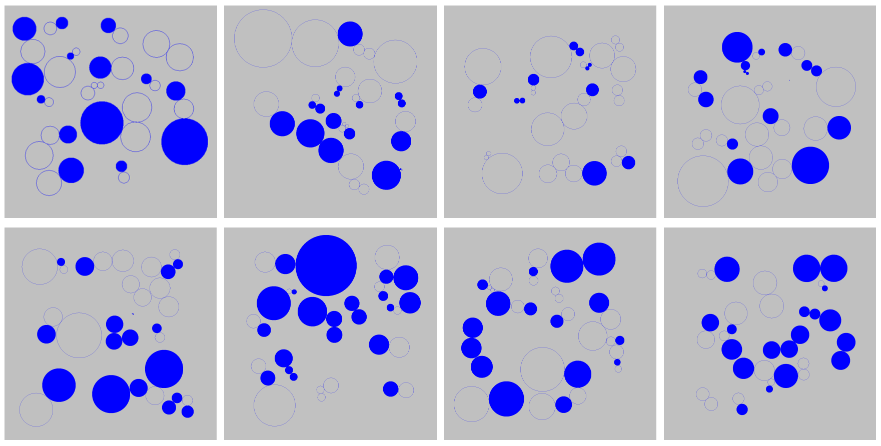
positionPDF is the pdf used to generate the initial position of each blob. (Here it is a uniform distribution across the drawing space.)

Examples of the *independent placement* algorithm are shown in figure 2.9. One of the effects of the algorithm is that pairs of same-sized blobs occur often: if two nearby blobs have similar growth rates, they both grow until they come into contact and become frozen. Although the diagrams generated with this algorithm do contain space which is a direct consequence of the blob growth being limited by contacting other blobs, it is not *Positive Space*. That is, space that is there does not contribute to the overall coherence of the pattern; essentially, it is merely a collection of blobs of different sizes.

Positive Space appears in the results of the contingent placement algorithm only when the invisible blobs are allowed. However, invisible blobs do not result in *Positive Space* in the independent placement algorithm (figure 2.10). It is clear that the space does not have the same coherent influence as that seen in the results of the contingent placement algorithm.



(a)



(b)

Figure 2.10: Typical results of the *independent placement* algorithm with **blob-Count** = 34, **growthPDF** = gaussian, **blobShape** = round, **allowedOverlap** = 0 ; **vis** = 0.5 (a) invisible blobs not shown; (b) as a, but with the “invisible” blobs shown.

The essential difference between the two algorithms is that the *contingent placement* algorithm places blobs in positions determined, to some extent, by the blobs that already exist. That is, it is essentially generative in nature with the sizes and positions of the blobs being strongly influenced by the environment within which the blobs "grow up".

In contrast, the *independent placement* algorithm pre-determines the placement of the blobs. It naturally results in space within the pattern: the blobs cannot enlarge to fill the entire space given their fixed starting positions. But it does not generate *Positive Space*.

2.2.4 Levels of Scale Algorithm

BlobWorld was also used to explore some aspects of the *Levels of Scale* property. As seen in the pseudo code for the *placement* algorithms shown in figures 2.5 and 2.8 the blob sizes are chosen according to a probability distribution function. In those algorithms a gaussian (normal) distribution is used with a user defined mean and standard deviation which are entered using the software's user interface. Although, of course, when using the *independent placement* algorithm this in itself overrides many aspects of the chosen sizes.

This approach generates a range of sizes as seen in figures 2.6 and 2.7 which does result in some *Roughness* but does not exhibit the 3:1 *Levels of Scale* property.

In order to investigate *Levels of Scale* in some more detail Blobworld encodes bi-modal and tri-modal distributions for size into the software. In these cases the mean (size) and occurrence likelihood (number) of blobs in the different modes have a fixed ratio of 3:1. Some sample distributions are shown in figure 2.11 which was generated using sizes generated by the same code used inside BlobWorld itself.

These size distributions were used to generate the diagrams shown as figure 2.12. This shows blob figures generated using the bimodal size distribution. The sizes follow the 3:1 distribution, but because that size has no effect on blob placement—other than influencing whether a blob will fit or not—there is little evidence of any *coherence* in the size distributions spatially.

However, on occasion, some degree of *Levels of Scale is* visible in BlobWorld patterns. For example in figure 2.13 where, coincidentally, the large blobs appear

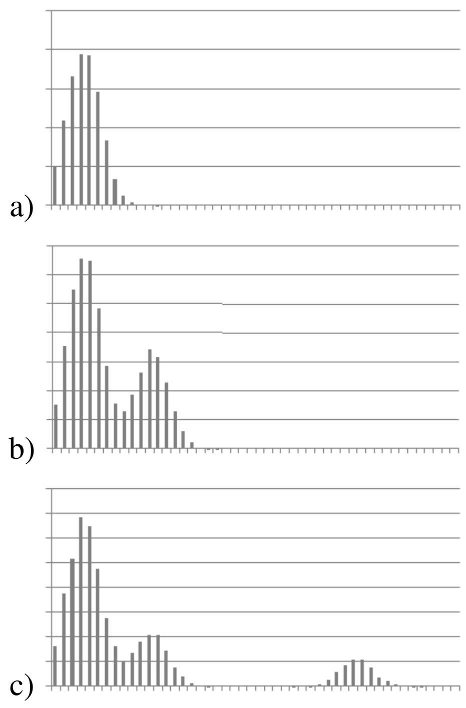


Figure 2.11: pdfs for investigating *Levels of Scale*. The x-axis is the blob size; the y-axis is the probability of that size: (a) single mode, gaussian distribution; (b) bi-modal, generating (approximately) three blobs of size 1 for every blob of size 3; (c) tri-modal generating (approximately) nine blobs of size 1 and three of size 3 for every blob of size 9.

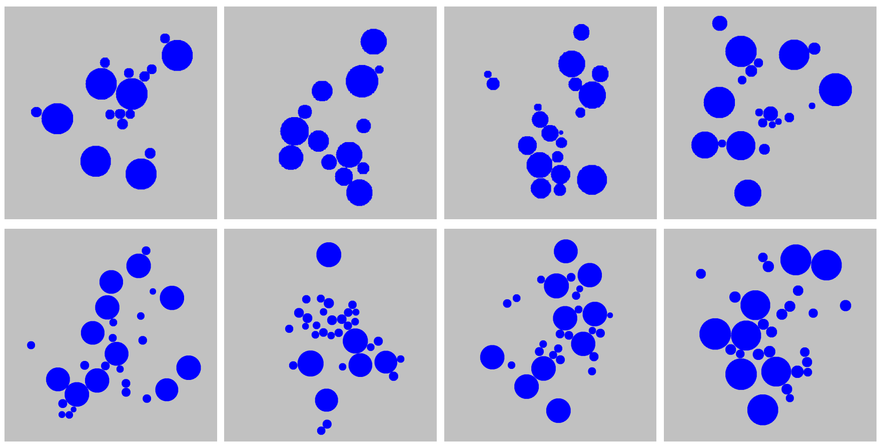


Figure 2.12: Attempts to generate *Levels of Scale*: contingent placement algorithm, bi-modal size distribution with a small standard deviation, **vis** = 0.5.

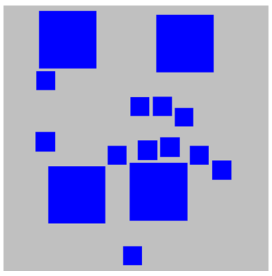


Figure 2.13: Attempts to generate *Levels of Scale* occasionally work: contingent placement algorithm, bi-modal size distribution, **vis** = 0.5.

to be constraining the small ones. This suggests that a small modification to the algorithm might well be capable of generating a suitable degree of *Levels of Scale*.

Following further experiment, and discussion with Chris Alexander, I hypothesised that in order to achieve the *Levels of Scale* property the various blob sizes would need to be arranged in such a way that changes in size are also, to some extent, reflected in their positions. Such an arrangement seldom appears in the context of either the contingent placement or the independent placement BlobWorld algorithms. In these cases either the blob sizes are pre-determined—in the case of the contingent placement algorithm—or they are a direct consequence of the position of only the nearest other blob—in the case of the independent placement algorithm.

These, and other similar, observations led to the *generative size* algorithm.

2.2.5 Generative size algorithm

Experience with the *independent placement* and *contingent placement* algorithms shows that when blobs are positioned *generatively* then a diagram that demonstrates Alexander's *Positive Space* property appears. That is, when the diagram evolves from a small core in accordance with its developing environment then the result approximates a beneficial property that is observed in the end result of human-developed architecture.

However, the initial *contingent placement* algorithm is generative only with respect to the position of the blobs; their size is determined independently according to the distributions discussed above.

Hence, I developed—following discussions with Chris Alexander—a further al-

```
blob[0] := new Blob(blobShape)
blob[0].setSize(sizePDF)
blob[0].setVis(boolean according to visProb)
blob[0].setPosition(origin)
blob[0].draw()
for i = 1..blobCount-1 do
    blob[i] := new Blob(blobShape)
    blob[i].setSize(sizePDF)
    blob[i].setVis(boolean according to visProb)
    blob[i].setPosition{blob[0].getPosition()
            | blob[i-1].getPosition()
            | blob[random(0..i-1)]. getPosition()}
    blob[i].setDirection(rand in $0 \ldots 360^\circ$)
    while not blob[i].isOverlapAcceptable(
            allowedOverlap) do
        blob[i].movePositionAlongDirection()
        blob[i].reduceSize(sizeRatio)
    end while
    blob[i].draw()
end for
```

Figure 2.14: Pseudo-code for the *generative size* algorithm.

gorithm which produces blob sizes that are also generative; that is they respect the process of development of the diagram and also of the environment within which the blobs are positioned. The algorithm produced is, in essence, a modification of the *contingent placement* algorithm. The modified algorithm generates the size of the blobs as a consequence of process of moving the blob until a suitable location is found for it. As the algorithm moves the blob along the randomly chosen vector looking for a site to place the blob then it also reduces the size of the blob in a fixed ratio for each "generation" of the process of moving the tentative blob site. Hence, blobs that have to be positioned a long way from their original location are reduced in size over several "generations". The pseudo-code for the *generative size* algorithm appears in figure 2.14. As compared with the earlier *contingent placement* algorithm, this algorithm has a further parameter:

sizeRatio is the ratio in size between different "generations" of blob.

Examples of a simple application of this algorithm is shown in figure 2.15. Here the blobs are always placed starting from the centre as the origin. As the algorithm searches for a position for the blob it progressively reduces the size of the blob in accordance with the size ratio; here set to 2 for clarity in the diagram. Subsequent "generations" of blob therefore reduce in size progressively, leading to the characteristic patterns seen in these examples.

Some further results, ones more representative of the intent of the algorithms, of executing this *generative size* algorithm are shown in figure 2.16. These diagrams are initially strongly reminiscent of the diagrams Alexander shows as representative of the layout of cities and structures which are the result of long-term human development [Ale04] as seen in figure 2.1: the blobs are positioned and sized in a generative manner that is a consequence of the positioning and sizing of pre-existing blobs as the diagram evolves. As can be seen from the diagrams in the figure the blobs are now showing evidence of the *Levels of Scale* property in that the blobs appear in a wide range of sizes but there are frequent clumps of similarly sized blobs.

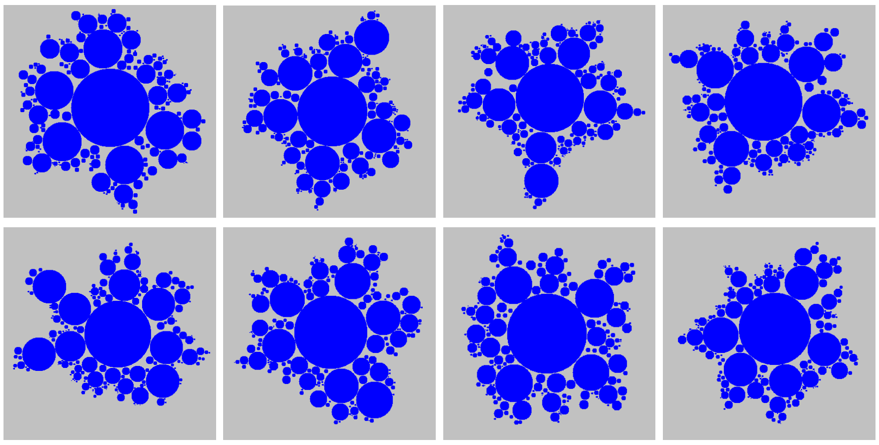


Figure 2.15: Example applications of the *generative size* algorithm with **blobCount** = 181, **sizePDF** = generative, **blobShape** = round, **allowedOverlap** = 0, **vis** = 1.0, **sizeRatio** = 2.0, **origin** = centre.

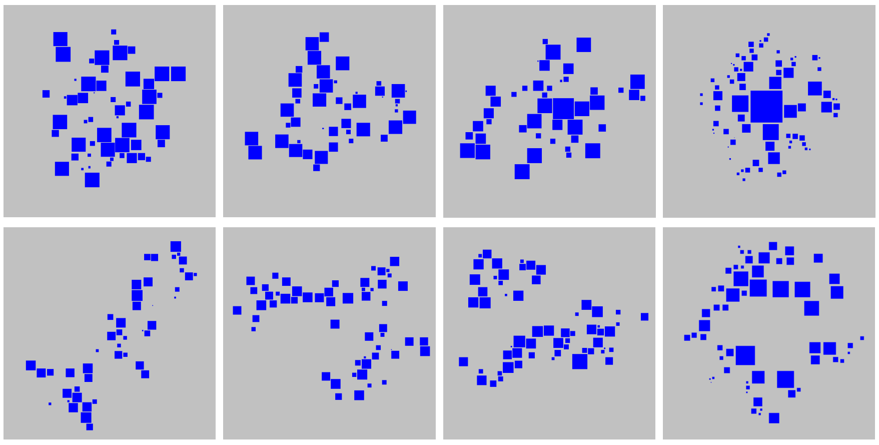


Figure 2.16: Typical results of the *generative size* algorithm with **blobCount** = 106, **sizePDF** = gaussian, **blobShape** = square, **allowedOverlap** = -3, **vis** = 0.5, **sizeRatio** = 1.4.

2.3 Blobs and environments

Blobworld shows how complicated results can be achieved from simple algorithms when applied across a large number of separate agents. Here the agents are simple blobs on a diagram. However, the arrangements of these blobs show patterns that reflect some of Alexander's properties (in this case *Positive Space* and *Levels of Scale*) in a subjectively pleasing manner. Alexander does insist though that this property is objective [Ale04, vol.1, pp316–317] [Ale04, vol.1, pp449–452].

At the core of the algorithms simple decisions are taken repeatedly about the size and position of blobs. The results reflect both the algorithms and the various parameters chosen but they are also a consequence of the environment that surrounds the blobs; specifically the positions and sizes of pre-existing blobs.

The algorithms that yield the best results are those that are more dependent on the blobs' environment. For example, the *generative size* algorithm yields blob sizes that are more reflective of the *Levels of Scale* property specifically because it responds to the diagram environment of each new blob.

Conversely attempts at generating *Levels of Scale* by merely setting the blob sizes according to bi- or tri-modal distributions were unsuccessful; apparently because although the blob sizes did follow the distributions they did not do so in a manner reflective of the process of generating a diagram within an environment.

The *contingent placement* algorithm is capable of generating diagrams that exhibit the *Positive Space* property. That the alternative *independent placement* algorithm does not have this capability indicates that the effects observed are more than mere chance.

It is likely that this capability of the *contingent placement* algorithm is due to the combination of two aspects. Firstly, the invisible blobs generate spaces that do indeed have a positive aspect, in that they contain blobs; the space is more than mere empty space, there is actually something there: (invisible!) blobs.

Secondly, the algorithm is to some degree generative, in that blobs are placed in positions that are strongly conditioned by the position of existing blobs; that is, by the environment that each blob is fitted into. The pattern does in fact *grow*, within its environment, towards its final configuration.

The *independent placement* algorithm also naturally generates space. However, those spaces do not jostle directly against the blobs; the blobs jostle against each other. That is, the space is not *positive*, it is merely empty (negative) space.

Attempts at generating the *Levels of Scale* property are also successful. The initial, somewhat explicit, attempt does not succeed in generating this property. However, the less explicit *generative size* algorithm shows that when blob size is made a direct consequence of the underlying generative process (that is when the size is a consequence of the evolution of the diagram) then the *Levels of Scale* property appears naturally in the resulting diagrams.

All of the scoring of the Blobworld diagrams is necessarily subjective, although as has been described, the original intention was to formalise that using, initially, Chris Alexander as the test subject. Although it was not possible to carry out this intention, there remains a strong subjective appeal in some of the diagrams, many of which are included here.

The next step for Blobworld would be the completion of a range of subjective scoring tests, with the objective of establishing a sound relationship between the apparent appeal of some of the diagrams with the algorithmic approaches used to construct those diagrams. However, that step is beyond the scope of this thesis which concentrates on the environment as part of the implementation of complex systems simulations.

This issue of environment is the key issue in the various blob sizing and positioning algorithms, Subjectively, when it is the prime mechanism for arranging and sizing the blobs then some of Alexander's properties appear. It is environments in simulations in general, then, that are the topic of the rest of this thesis.

Chapter 3

Complex systems environments

The work on Alexander's patterns established an initial interest in the notion of a collection of agents in a shared environment which, as I have discussed, became the background for the rest of the work described in this thesis. In this chapter I explore in more detail this background and discuss how it coalesces into the motivation for the rest of this document.

Complex systems, as I have discussed, consist of a number of agents interacting within some environment. Many complex systems consist of a very large number of agents, for example the number of cells that interact to provide the functions of the human adaptive immune system is vast. These agents interact in such a manner that interesting, or indeed useful, behaviour emerges from the specifics of that interaction.

Complex system behaviour can been seen to result from the interaction between the system's agents. For example, in the common example of bird flocking [Rey87] each bird (the agents in this particular system) observe where other birds are and their velocity and use that information to modify their own velocity. From this simple behaviour emerges the fascinatingly complex moves made by a flock as a whole (figure 3.1).

Complex systems are common in nature, where such distributed systems dominate. Computing systems, of all forms, are necessarily different in form. It is usually not possible to construct computational systems, such as complex systems simulations, with anything like the same number of agents as appear in complex systems. Although, there are some counter examples amongst which the Internet it-

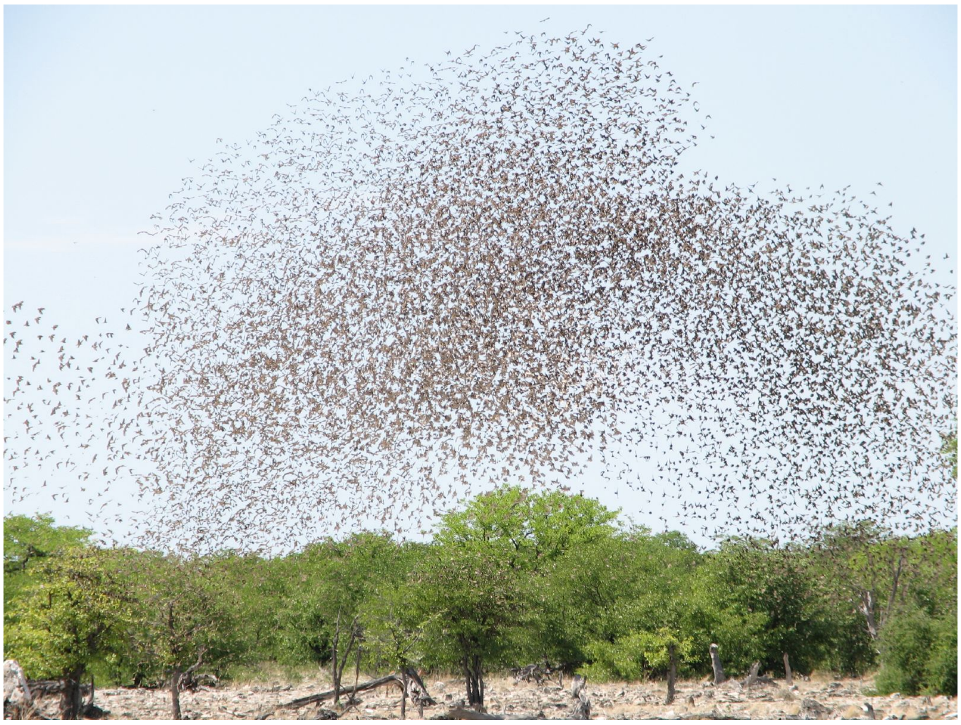


Figure 3.1: Red-billed quelea flocking at a waterhole.

self is the most obvious, being constructed using a large, and ever growing, number of individual agents. Even in this case, though, the communications between these agents are necessarily limited. Each individual agent can only communicate with a very small number of other agents; a typical desktop PC will only communicate with the local router even though those communications are specifically targeted at other agents elsewhere in the Internet.

In order to study complex systems we need to adopt an explicit simulation approach due to the difficulty, or infeasibility, of analytical solutions. By "explicit" here I mean some simulation approach that includes direct computational analogues of the agents in the complex system itself. Each such analogue would have to simulate the individual behaviour of the complex system agents themselves. The essence of the complex system is that the complexity arises from the agent interaction and consequently the computational analogues of those agents would also need to communicate with each other in order to simulate the overall behaviour of the complex system. The user of the simulation would need the implementation of the interaction between the agents to be understandable, and demonstrably able to operate in the same manner as the communication done between the complex system agents themselves.

Buried in this apparently simple design decision—for that is what it is in essence—is a problem. The complex system itself, is populated with agents which are genuinely independent of each other, subject only to the physics of the real world. That is, a real world bird can choose to fly in whatever direction it likes, regardless of what other birds in the vicinity are doing, as long as it does not try to violate real world physics, for example by trying to occupy the same space as another bird. Apart from such concerns each bird is completely independent of all others.

That is the background to this work. The motivation for what follows, though. is that this complete separation does not apply to the simulated agents in, for example, a flocking simulation. These agents are not completely independent because they will frequently be simulated by exactly the same computational elements as are used for the simulation of other birds. That they must share processors and threads, for example, means they are subtly inter-dependent.

3.1 Simulation background

Many implementation approaches have been proposed for providing a suitable simulation framework. The most common of these is the use of some sort of agent-based modeling approach [RG11] which focuses on the agents rather than the environment.

The most obvious way of implementing such a model is the use of object-oriented programming [Mey00]. The use of object orientation ([AGH05], [HWG06]) is now almost universal in industrial programming but, as a technique, it has its origins directly in the construction of explicit simulations [Hol94].

An alternative would be to use a process oriented approach such as that typified by the use of occam-π [WB04].

A naïve implementation of a complex system simulation would represent each agent as, say, an object in an object oriented programming language such as Smalltalk [GR83] or a process in a process oriented language such as occam-π [WB04]. Interaction between the simulated agents could then most obviously be achieved by sending messages (in Smalltalk) or communicating using channels (in occam-π).

However, such a simple approach leads directly to significant difficulties. Some of these, such as the sender and receiver of a message requiring access to the same metadata, have well-known solutions. (In this case the various interface description languages, for example Java IDL [LBS98] and CORBA [OHE97]) were developed for just this rôle.)

The issue that causes most implementation problems is that directly caused by processor resource contention: deadlocking. Deadlocks occur as a direct result of activities proceeding in parallel (in the simulated parallelism available on a single processor core). Whenever a number of resources are shared between activities proceeding in pseudo-parallel then there is the possibility of contention for those resources. In the case where shared resources are the agents in a complex systems simulation then, as all agents are potentially interested in all other agents the potential for contention is significant.

As soon as we get this sort of contention for shared resources then deadlocks are likely, in the same—albeit simplistic—manner that they appear in the classic *Dining Philosophers* problem [Dij71] [Hoa85].

Deadlocking is a major concern in distributed systems in general and has been much studied since things like the *Dining Philosophers* problem were described. Current items of distributed enterprise software, such as Microsoft's SQL Server relational database management systion [Mic07], include deadlock detection and implement a brutal approach to resolving a deadlock where one of the deadlocked processes is summarily killed which at least allows the rest of the implementation to proceed.

The motivation for much of the work in this thesis came from personal experience in implementing simple simulations using a process-oriented approach; specifically using JCSP [Wel02] which is an implementation of Hoare's *Communicating Sequential Processes* [Hoa85] for the Java programming language. This experience was frustrating as these early simulations were naïve, with no attention paid to the potential for deadlocking and, consequently, the simplest simulation led easily to deadlocks which needed very careful analysis of the system to resolve. That analysis was not into the world of the complex system, but into the simulation itself. That is, rather than concentrating harder on the complex system I was studying I was forced to delve deeper into the computational aspects of the simulation itself.

Deadlocks occur because two, or more, processes each require the same resource in order to proceed. In a simple implementation each process just waits, in a similar manner to one of the Dining Phiosophers waiting for a fork with which to eat their meal, until the resource is available. If all the processes that are dependent on the resource are also waiting in the same manner then the system deadlocks.

Such situations arise naturally in the context of complex systems simulations. For example a simulation might have just two agents with each implemented using a single thread using blocking interactions with other agents (the simplest "procedure call" implementation). If one agent poses a question of another agent where the answer to that question is dependent on querying other agents then deadlocks are likely. This is because if agent A asks agent B a question for which agent B might need to ask agent A for something then the simulation cannot proceed as agent A's thread of control will be blocked waiting for a reply to the original question and cannot reply to the query from agent B. It is a characteristic of a simple approach to complex systems implementation that the agents are essentially acting as both the competing processes (the philosophers in the problem) and the resources that are being competed for (the forks).

This is a particularly simple situation although it is, crucially, multi-threaded. A single-threaded implementation, typical of simple simulations with a small number of agents, would not suffer any issues of deadlock because in no situation could one process seize a resource and make it unavailable to another process.

Any realistic large scale simulation could involve chains of interactions with a large number of agents and must be multi-threaded in order to access the computing power requried. Predicting the likely patterns of interaction in such a situation is very difficult; hence the brutal solusions adopted by existing enterprise software as has been discussed.

Even once such deadlocks are located—by this study of the simulation—they must be resolved by the introduction of some computational technique, such as the "client server" pattern for concurrent systems [MW97, ASB$^+$08a] and barrier synchronisation [BWS05a], which impose a processing pattern onto communications between the various components of the simulation. These patterns seek to prevent the appearance of deadlocks and, indeed, they can be successful in doing that. Similarly, all object oriented programming libraries include copious numbers of classes for implementations to use to serialise interactions and share physical threads. These are computational devices, though, a consequence of the predominantly serial world of computation and frequently with no simple analogue in the highly parallel world of independent agents in complicated environments. These devices, typically expressed as patterns and class libraries must be used explicitly by the programmer, leading away from the complex system that is being simulation. For example [OW04] examines in excruciating detail the interaction of apparently parallel threads and the locking constraints that arise in a specific programming language.

The effect of the use of these patterns and classes is to impose additional protocols onto the interaction of the simulation's agents. For example, in the case of the client server pattern, processes are designated as either *clients* or *servers*. The former agents all operate by requesting a service from a server and waiting for a response. All servers just sit waiting for a request which they honour and then wait again. Servers can also be clients of other servers. Such a system is deadlock free if the graph of client/server relationships is acyclic. Such a structure is reminiscent of "client-server" enterprise systems [Wika, Ber92] or, more generally, multi-tier architectures [Wikd, Fow02]. In such systems the clients and the servers are layers in an architecture where the servers provide a pre-defined set of services to the clients. Each client is able to operate in a manner largely independent of others because the implementation of the system constrains the overall patterns of behaviour, for example by transactional access to an underlying repository [Wikh, GR93], in such a manner as to guarantee various overall system properties.

In a complex systems simulation, though, such a simple configuration is not feasible for most situations because all of the agents may be both clients and servers, at different moments, meaning that such an acyclic graph does not naturally appear.

Using barrier synchronisation gives a way out as it provides a way for all agents to synchronise with each other; essentially the agents can agree that at some point in time they will behave in a particular manner and at other points in a different manner.

These approaches are essentially engineered approaches to a technical problem, a technical problem that is common in enterprise systems. They are a special case of layering in systems architecture, something that is now expected in all architectures as in [FT02] and [Fow02]. Although such patterns are of clear use in such systems they do not appear to exist in real world complex systems. That is, it does not appear to be the case that all of the birds in a flock are synchronising their behaviour, in the manner that is implied by a barrier based solution; the birds that flock above a city-centre park are not working to some standard global pattern lest they deadlock and fall out of the sky. Rather, each bird is observing other birds and then doing what it wants, when it wants, and in whatever order it wants in the sure and certain knowledge that its world really is deadlock free.

One of objectives of an explicit complex systems simulation must be to reflect the complex system itself, without an excess of computational artefacts that could be argued to invalidate any results of the simulations. That is, using things like barrier synchronisation seems to essentially miss the point about building a complex systems simulation. Just because it does the same sort of thing as the complex system itself does not necessarily mean that it is doing it the same way.

3.2 Real world agents and their environment

In the previous section I discussed the background to the work described in this thesis. Here I will look at the concepts that became the motivation for building the simulations and experiments described later in this thesis.

I will now examine the specific issues that were the motivation for the approach, and subsequent experimentation, followed in the rest of the work.

3.2.1 Action at a distance *versus* mediating fields

Eschewing the notions of things like barrier synchronisation, how do agents in a complex system actually interact? In the complex system itself it is convenient, for discussion, to describe things like flocking birds as directly interacting with each other as is shown in figure 3.2. But, some thought shows that this is a simplification, albeit a useful one, of what is really going on.

Some thought about the physics of the real world situation shows that the agents, such as the birds in figure 3.2 are not really directly interacting with each other at all.

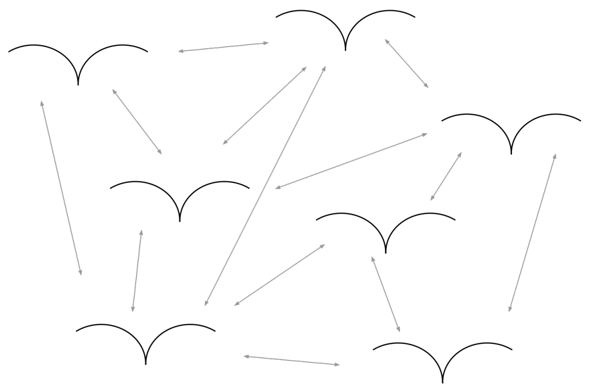


Figure 3.2: Diagrammatic representation of a collection of birds interacting while flocking with each other. Some interaction arrows have been removed for clarity.

Rather, a bird flying along reflects ambient light into the space around it; as it sings it pressurises and rarefies the air about it. Another bird, assuming that it is awake, is sensitive to light and air pressure and it processes the light and air pressure that it receives in order to form a mental model [JL86] of the first bird. That is, these two birds are not directly communicating with each other. Each is interacting with the physical environment only as is shown diagrammatically in figure 3.3. The first bird is placing information into that environment and the second is inspecting the various changes in its environment, some of which may well be a consequence of various physical laws and the effects of the first bird. Other changes in its environment will merely be a consequence of the outside world, such as the level of ambient sunlight and the wind speed and direction. If, and only if, the second bird is interested in the behaviour of other birds will it react to indications in its environment of the first bird's behaviour.

This approach, an alternative model of interaction between the birds, depends on the environment as the repository for information about the agents, which are embedded in the wider environment, which can be observed by other agents. One bird can always come along and look in the environment and see what another bird is placing in the environment. It cannot, though, observe anything that is not in the environment. A different bird might seldom update the environment, if it is just sitting quietly on some perch.

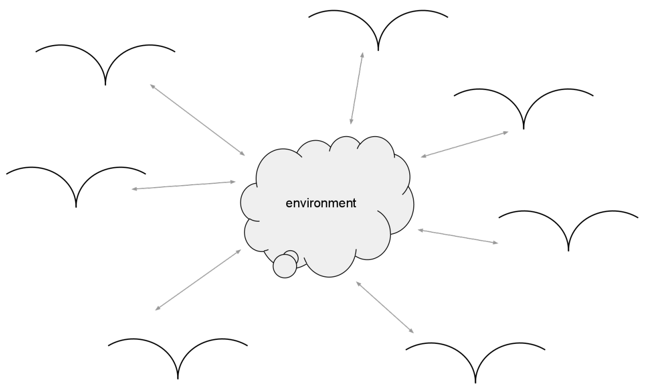


Figure 3.3: Diagrammatic representation of a collection of birds interacting exclusively with their environment.

The real world is such an environment; one where fields interact, photons pass each other and the rest of physics is implemented with ease. In this view the agents are *embodied* in the environment [Ste07], and it provides services to those agents. Each agent just does what it wants without regard to direct interactions with other agents. That is, even in the real world, the agents in a complex system are interacting in a manner reminiscent of a client server architecture. The environment provides services to the agents, in a manner analogous to a *server*. The agents are *clients* of those services.

The naïve model of a complex system, with agents directly interacting with each other, is essentially "action at a distance". One agent must know directly what other agents exist that are interested in it and must directly interact with those agents. As this is happening those distant agents are also potentially interacting in the reverse direction.

The subject of this thesis is that of taking the opposite view, one where the direct interaction between agents is not simulated but all interaction is expressed as communication through the mediating fields that exist in their environment. In this approach there is no direct interaction at all; the lives of individual agents just affect each other by existing within the same set of fields; within the same physics.

As a different example of this consider an adaptive immune system. Here the agents are the various molecules and cells that form the active components. The molecules are not directly signaling to each other about the feasibility of particular interactions. Rather, the simulation would express the concentrations within the environment of the various molecules. These concentrations which would affect the probability of interactions occurring as a consequence of the stochastic processes mediated by the environment. At any stage in a simulation of such a complex system, the computational objects simulating the agents would inspect their environment, yielding such information as the concentrations of the surrounding biochemistry, and make some decision as to what behaviour to exhibit at that moment.

3.2.2 State

When simulating some complex system in this manner it is all very well saying "agents place information into the environment" but some further thought is needed about what is placed in the environment.

In something like a collection of birds flocking in the real world each bird has a large and complex internal state: it knows whether it is flying or not, how hungry it is, whether it needs to drink or defecate. But, from the point of view of flocking, other birds are interested primarily in the distances between the birds and what the perceived relative velocities of the other birds are.

That is, each agent has an "internal state" that represents everything it needs to know to behave in its innate manner. Further, each agent exposes an "external state" to the environment, which is available to other agents in the same environment. This external state could be simply a subset of the agent's internal state. For example, in the case of the bird it could just be that part of the internal state that represents the position and velocity of the bird. However, there are cases where the agent could deliberately mislead other agents with its external state. For example, when one insect species mimics another it is deliberately creating an external state to mislead observers about its internal state (figure 3.4).

Complex system agents are essentially egocentric, even solipsistic. Emergent behaviour appears as a consequence of each agent just doing what it wants to do in its own environment. A flocking bird does not know precisely where it is, just merely where other birds are relative to it. In a complex system simulation, something does need to know where the agents are, because those positions are the over-

Figure 3.4: *Chrysotoxum verralli*, a hoverfly species that mimics the appearance of a wasp; its internal state is that of a hoverfly, externally it is a wasp.

all context of execution of the complex system. This context is the environment which must itself know where each agent is and, therefore, will also know what other agents are in the vicinity of each agent. That is, the environment knows things about the agents that are not actually part of the agent's internal state. For example, a bird just thinks that it is flying in the direction of an interesting looking food source, but the environment knows that it is actually flying north-by-northwest.

In retrospect, the notion of considering the environment is something that I, and my colleagues at the time, considered some years ago when looking at the use of object-oriented design and programming. At the time, in the early 1980s we were some of the earliest adopters of the notions of object oriented design and programming, having got access to an implementation of Smalltalk-80 [GR83] and the, at the time, advanced hardware needed to run it [Kra83].

As discussed, much of the object-oriented literature of the time addressed simulation as the notion of using objects to directly model components of a simulated world was a clear benefit of the approach. We fell to discussing how an object-oriented simulation of a snooker game would work. It was clear that we would have a class for Ball, instances of which would support messages to send ball instances

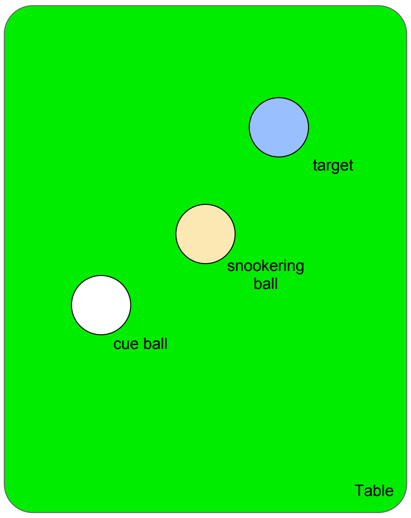


Figure 3.5: The cue ball, in a game of snooker, is "snookered" in that it cannot "see" all of the target ball.

on their way and to tell them that another ball had hit them in a particular direction. However, how would we implement the notion of the cue ball being able to "see" the target ball; that is, how would the cue ball know that it was "snookered" (figure 3.5)? As this information represented something about the combination of the cue ball, the target ball and all the other balls in the simulation it did not seem as though that information should be in any of those objects as it was not uniquely the concern of any single particular object. Sharing the information between various ball instances would reduce the cohesion [Mye78] of the overall design.

Our conclusion, after discussion, was that it was the *table* that knew whether two balls could "see" each other. That is, the environment that supports the balls, the table, is the thing that understands the relationships between the objects in this particular world. And, as a corollary, the table must then become a "first class" member of the simulation. This insight generalises to other environments such as the air around the balls, other interactions and ultimately other simulation contexts.

An agent can generate some external state just by virtue of the physics of its environment. Photons just bounce off a bird into the environment. Other birds

```
while true do
    Read external state of other agents from environment
    Update internal state to match perceptions of environment
    Write external state of this agent to the environment
end while
```

Figure 3.6: Pseudo-code for an agent operating within an environment.

detect some of those photons and are then able to see the original bird and are also able to infer position and velocity information. This “involuntary” external state is contrasted with other states placed into the environment by an agent in a “voluntary” manner. Voluntary state could be, in the example of birds, a song that is sung in response to hearing the song of another bird of the same species (which it hears through the mediating environment), sung maybe for territorial enforcement or for the purpose of finding a mate.

3.2.3 Accessing the environment

In a simulation constructed in this manner, each agent interacts with the environment to access information about the other agents' (external) states. In implementing this simulation we face a natural consequence of computation which is that the behaviour of an agent is expressed in an essentially serial manner. That is, the natural implementation is some form of loop such as is shown in the pseudo code in figure 3.6 where each agent first “asks” the environment for information about other relevant agents' state (the agents it can see, or hear, for example) and then this state information is used by the agent to update its internal state appropriately.

That is, the agent queries its environment, yielding a set of values in some topology [GM02], which not only represents the set of all possible values but also describes how the values might change.

For example, in a bird flocking example, one of the items in a query result could represent a bird that is close to the querying agent. As such, the environment can accurately describe the (relative) position of the nearby bird and its velocity in terms of a three-dimensional Cartesian space. Furthermore, the topology of the particular space used might show that the nearby bird could move freely in the two horizontal dimensions but it was constrained to move only upwards in the vertical dimension because it is, at the moment, standing on the ground. That is, the reply to a query about the position of the bird gives a precise position in a space, but that space is further described by its extent and its shape.

If the bird being described is distant from the querying agent then the position of the bird may not be accurately defined. For example, it might be clear in what direction the bird lies but its distance from the querier could be only poorly known when beyond a particular range. Similarly, the velocity of the bird might be only poorly known, if at all, as the velocity of a distant agent which appears merely as a distant speck might be very hard to determine. In this case the reply is again a position in a space, but, that space is poorly described in some dimensions. It is still a three dimensional space but the resolution, in this case in the range to the distant agent, does not support an accurate definition of the position of the agent; it could just be "nearby" or "far away".

Here the simulated environment is acting as the implementation of sophisticated functions performed in the real world by both the agent itself and the environment. The agent itself detects the photons impinging on its retinas from a distant bird and attempts to calculate size, distance and velocity of the bird from those photons and, probably, experience in these sorts of situations. The real world, that is the environment, affects many aspects of the passage of those photons; it understands the albedo of the distant bird and can calculate how photons from the Sun are reflected by the bird, and how effectively those photons are transferred to the observing bird.

Concentrating on the environments provides a way, in a simulations, to separate concerns between the agents and their environment. In a particular simulation, the choice of what computation is performed by the environment, and what by agents, is a modeling decision. Certain functions may be embodied in the environment itself, and those calculations performed by the environment. For example, in the context of the simulation of the complex system of ant-trail formation and decay ants leave behind them a trail formed with by a pheromone which may be followed, and reinforced, by other ants. In this way a trail of ants will follow the same route to a food source. When the food source is exhausted and ants cease following the same trail then the pheromone level decays with the effect that the trail does not confuse other ants. In a simulation of this system where the environments are first class residents the responsibility for simulating the decay of the pheromone may be allocated to either the environment itself or to a special agent whose external state is the level of pheromone along the trail. This agent, of course, does not exist in the real world where the pheromone decay is performed by physics. But, that is not necessarily the right solution for a software simulation as is common in software design, and in particular in object oriented design.

3.2.4 Multiple environments

The notion of the results of the query being embedded in a topology brings with it the concept that interaction between agents may simultaneously follow a number of different patterns. The example given above is a purely spatial one, the notion of space clearly being of significance in complex systems implementations [ASB$^+$08a]. However, the exact same query/response model could be used for any interaction between agents in a complex system. One extension of the simple spatial model is to note that a human agent is physically "near" to a collection of other human agents but may nonetheless communicate simply with other human agents whose telephone numbers are in the first agent's address book. That is, there are two sorts of "nearness" here: one is physical nearness, the other is "communicable" nearness. For some aspects of complex systems behaviour only the first sort of nearness would be relevant, for others both sets of "near" agents might be important. (This example is inspired by Milner's *bigraphical model* designed to model both a spatial and a connectivity configuration simultaneously [Mil09].)

This is analogous to agents being simultaneously embedded in multiple environments, each of which has its own particular properties. For example, birds reflect photons allowing them to be detected visually. They also can sing, compressing and rarefying the air in their environment so that other birds can detect them even if they are not easily visible. Birds use song to communicate for many different reasons. They may produce alarm calls to warn of the presence of predators, or to reinforce a territorial claim or to find a mate. However, the sonic characteristics of the real world are rather different from the visual ones. Sound is to a large extent an omnidirectional medium but it is capable of being transmitted over significant distances; certainly a receiving bird could hear song without being able to visually detect the singer.

The approach discussed here allows these two environments to exist simultaneously. That is, an agent may place external state in each environment although the meaning of that state might be different in each environment. An agent that receives multiple states for another agent, from multiple environments, must fuse the information provided in those states so as to form a coherent picture of the other agent.

3.3 Environment orientation

Given the simulation background described in section 3.1 and the motivation for further work described in section 3.2 the subject of the rest of this thesis is the adoption of the approach described here, now formalised as "environment orientation", as the fundamental mechanism for constructing complex systems simulations. Adopting this approach eschews all representations of direct interactions between agents. Rather, all agent behaviour is seen as mediated through the environments within which all the agents are embedded.

Although the notion of the role of the environment as the key part of a simulation is based on observations of the physical world, the particular agents and behaviours that exist in a simulation is a modeling decision. Each simulation should be constructed with the explicit knowledge of which aspects are to be embodied in the environment.

3.3.1 Definition

The elements of environment orientation are those that have been described in this chapter. A specific definition of the concepts Specifically:

1. Regardless of its particular role, each agent has an internal state, representing what the agent knows of itself.

2. Each agent publicises some aspects of its state, its "external state" to the environments within which it is embedded.

3. Simulation of an agent is performed without reference to other agents. The

simulation just reacts to the contents of the agent's environment. The agent may decide when to publicise its external state.

4. Agent behaviour is provided for by allowing the agent to retrieve, from its environment, information about the external state of the agents with which it is interacting.
5. The environment itself is a first class computation component and must be aware of the agents with which each other agent can interact.

3.3.2 Research hypothesis

The rest of this thesis investigates a single hypothesis, that **the environment oriented approach is an appropriate software paradigm for simulation of complex systems**. This investigation takes the form of a number of experiments, building complex systems simulations of several forms, in order to test this hypothesis.

The structure of the rest of this thesis is, then, as follows:

Chapter 4 contains a review of the possible software architectures for implementing an environment oriented complex systems simulation.

Chapter 5 presents the design and implementation details of a platform upon which the experiments in the rest of the thesis are constructed.

Chapter 6 presents the initial validation of the experimental hypothesis by demonstrating that it can be used:

1. to implement a well-known example of complex systems behaviour as an environment oriented simulation,
2. to implement a similar simulation, albeit one that admits to a number of environments within which the agents are embedded and
3. to implement a simulation which includes an external environment representing the landscape within which the agents are moving.

Chapter 7 examines the issues related to multiple environments more and shows experimental results where multiple environments are combined by the use of fuzzy logic.

Chapter 8 provides a more substantial evaluation by implementing, in an environment oriented manner a simulation of complex organisms evolving in a complex environment.

Chapter 4

Software architectures

In order to use the environment orientation approach to complex systems simulation then it must be readily implementable in some form. That is, an abstract architecture must be specified that supports this sort of systems implementation. Driven by the list of features in section 3.3, what are essentially architectural requirements, this chapter examines the software engineering background to architectures of this type, and the possible implementation strategies. The results of the discussion in this chapter are used, in turn, as the requirements for the experimentation platform described in chapter 5.

As I have described it, the model is essentially a client server one in that the agents function as clients of the environment server. The environment server must provide services for at least:

1. retaining the external state of the simulation's agents,
2. facilitating the updating of that information in a manner reflective of real world physics and
3. presenting the state information to agents in a manner assimilable by the agents.

As has been described, complex systems can be regarded as inherently "client server" in that the agents function essentially as clients of the environment. Some sort of client server approach is therefore appealing for complex systems simulations. As I have discussed, this sort of approach can be inherently deadlock free as there is a single locus of concurrency where it is straightforward to serialise updates using the standard techniques of transactional control [GR93]. Such an approach essentially forms the basis of most high performance commercial computing: applications that demand very high performance in the context of a world that is rapidly changing; something that seems quite similar to that of a complex system.

Just saying "client server", though hides a world of complexity. As I have already discussed, the "server" (that is the environment) in a complex systems simulation must provide a number of services to a client (that is an agent of the simulation). I will discuss these services in more detail and the manner in which they can, in general, be provided.

4.1 Environment service requirements

The services discussed in section 3.3 are, then, requirements for an abstract architecture that can be implemented by the techniques to be discussed later. In this section I will examine each of these requirements and discuss the extant techniques for their implementation in section 4.2.

4.1.1 State retention

The first server service is that of state retention. That is, an agent (the client) must be able to supply some state to the environment (the server) which will be retained and supplied to agents at a later time.

The form that agent state takes initially seems simple in that it is just some opaque information provided by an agent and entrusted to the environment for safe-keeping. However, the environment must retain more information that just an opaque BLOB[1] representing the external state information for each agent.

Much of this this state information will only need to be interpreted by the agents themselves, which will therefore be capable of understanding the information; I am here making the reasonable assumption that the agents share the metadata that describes the contents of the external state. However, the environment server itself

[1]A *Binary Large Object*, merely a collection of binary data that is not interpreted other than by the originator.

also needs to understand some of the agents' information, in particular any that relates to the aspects of physics that it is required to implement. For example, if a simulation requires that the information that is delivered to an agent is sensitive to spatial position then the environment must know the position of each agent. That is, the implementation of the environment is also required to have access to the state metadata. As a concept this is reasonable as it essentially means that all parts of a simulation agree as to the representation of things like "position" and "velocity". Furthermore, it is possible that that information is not known to the agent as I have already discussed in section 3.2.2. That is, it must be possible for the environment to elaborate the information provided by an agent.

The environment must therefore store a large collection of agent external state objects[2], elaborated by additional information added by the environment itself in some sort of store, many implementations of which are feasible; some of which will be discussed later in this document.

4.1.2 State update

External state objects supplied by an agent would be expected to be updated by an agent. For example, the velocity of a flocking bird would be updated by the bird agent itself.

However, some information provided by an agent is not of this form. For example, in a simulation of ant-trails, as discussed in section 3.2.3 the ant agents would emit information describing the depositing of a level of a pheromone as they move through the environment[DG89]. The levels of pheromone in ant trails decays with time so that trails that are not reinforced disappear but those that are used frequently, for example because they lead to that apple core you left on the floor, are augmented by frequent use.

The information that describes these pheromones would not be expected to be updated by the original agent. It would be possible for the computational agent to do that, but it would not be representative of the real world behaviour and hence not a suitable approach to take if simulation fidelity is a target. Rather, some other

[2]I use the language of object orientation in this thesis, and in particular the terminology of Smalltalk [GR83] and the Unified Modeling Language [FS03] with no further explanation when discussing the abstract and concrete implementation of environment orientation.

simulation artefact must implement that aspect of real-world physics that causes the level of the pheromone to drop.

There are two obvious choices for this artefact. Firstly, it could be argued that pheromone decay is a function of the environment, and in particular the physics of the environment, and as such it should be provided innately by the simulated environment itself. Alternatively, the decay of pheromones could be seen to be the effect of other agents, ones that explicitly represent physics; in this case the physics of biochemical compounds and their concentrations.

A concern here is the complexity of the physics and the purely software engineering concerns of whether embedding that complexity into the environment itself represents a coherent software module. For example, if simulating a complex system representing beaver dam building then a beaver-built structure, the dam, must be checked to see if it will support the weight of the beaver as it continues to extend the structure. A dam that would initially support the beaver might subsequently, after further extension, cease to do that. The physics behind a calculation of this would likely not fit well within the environment itself but could easily be seen to be an additional agent; a reification of the physics of weight-supporting structures.

Whichever approach is taken, the simulated environment must provide for such state update. However, from a purely engineering perspective I would expect that the latter approach would be the one to take. This would be for two reasons: firstly, it would be sensible from the point of view of data independence if the environment provided no domain specific services so as to enhance its scope of applicability. Secondly, the environment must already provide the facility to deposit and update state information, so why not use that for such purposes? This implies that the set of agents in a complex system simulation could include some which implement those aspects of real world physics that it is required the environment embody.

This is analogous to a common concept in object-oriented design. There has been much discussion about the best way to determine what specific classes should be designed in order to satisfy the requirements of a particular system. A particularly naïve approach is to find the real world classes, for example some authors recommend using the nouns that appear in a requirements document, and making those the classes. In general, this turns out to be a rather poor approach [Mey00] as many good object oriented designs include classes that would never appear in

```
while (true)
{
  Neighbourhood n = env.query(queryText,
                      <parameters drawn from internal state>)
  internalState.update(n)
  env.update(generateExternalState(internalState))
}
```

Figure 4.1: Pseudo-code for agents using a *query oriented* approach.

such a document and which are consequence of the chosen design rather than the requirements. For example, this applies to the Command and Factory classes that are implicit in some of the patterns in [GHJV95].

Similarly, many of the agents in a complex systems simulation will be present purely to represent the computational analogues of real world agents. This does not preclude others that represent artefacts of the chosen design of the simulation.

4.1.3 State access

Finally, the environment must provide the agents with facilities to allow access to the external state of other agents.

There are two general strategies here, relating to a possible inversion of control.

One approach would be for an agent that wishes to see the external state of a set of other agents, to make a *query* of the underlying environment orientation server. The query would provide the server with all the information it needed, along with its knowledge of the agents, to select the information required and provide it to the agents. This strategy, referred to here as *query oriented*, is summarised by the pseudo code in figure 4.1.

A complementary approach would be for agents to inform the server of the sort of information they were interested in, and to have that information delivered as and when it was available. In the meantime the agent would carry on with its normal behaviour. This strategy, referred to here as *subscription oriented*, is summarised by the pseudo code in figure 4.2.

These two approaches have different characteristics. The query oriented is appropriate for systems, perhaps like bird flocking simulation, where an individual agent can always be sure that its environment will change rapidly and appar-

```
...
env.registerInterest(topic, callback)
...

void callback(Neighbourhood n)
{
  internalState.update(n)
  env.update(generateExternalState(internalState))
}
```

Figure 4.2: Pseudo-code for agents using a *subscription oriented* approach.

ently continuously. The subscription oriented approach would be useful for systems where some information was available only occasionally and unpredictably, or where it was needed to "interrupt" an agent from its normal activities. That is, in situations where the particular environment was not changing apparently continuously.

In this abstract architecture, the server is the entire locus of inter-agent concurrency. That is, the agents execute without consideration for each other, simply relying on the server to provide pertinent information. This is the approach used in the world's largest commercial systems.

One observation of both of these approaches, query oriented and subscription oriented is that they are both providing *communication orthogonality* as described in [Gel85]. That is, a receiver of a message (in this case an agent getting details of another agent's external state) does not care which particular agent provided that state as long as it satisfies the criteria for being provided. Furthermore, the sender of a message (again, the state placed in the repository) does not care which agent sees that information as it is freely available in the environment.

4.2 Implementation background

Given what are essentially requirements for the abstract architecture that appear in section 4.1 in this section I will discuss the existing background material that relates to the possible implementation strategies for these requirements.

4.2.1 Implementation issues

Firstly, I discuss the major implementation issues that would be expected to influence the specific implementation choices to be made.

Transactions and locking

The regularity of the interactions between agents and their environment permits a wide variety of implementations; for example, massively-concurrent simulations can be constructed using the client-server pattern for process-oriented implementations [MW97, WJW93] as I have discussed.

Within the CoSMoS project some practical use has been made of the environment orientation approach to simplify some aspects of simulations [ASB$^+$08b, APS$^+$08]. In order to see a little more deeply into the structure of a simulation constructed this way there are a number of other issues we need to consider, which I will discuss in the following sections.

Environment orientation can be considered as a *transactional* approach to complex systems simulation. Each agent in a simulation is responsible for maintaining the information published about itself. It does this by performing a sequence of update transactions against the environment, which is a shared repository of external states: during each cycle, an agent will obtain, whether by querying or getting a subscription update, from the environment the external states of the other agents it is interested in and, in turn, *update* the environment with its new external state.

The transactional approach to software design is very common, almost ubiquitous, in commercial computing. The abstract architecture shown as figure 4.3 summarises this approach. Typically a very large number of computational nodes will act as clients to a smaller number of application servers which will, in turn, act as clients to a central (albeit perhaps distributed) resource manager, typically implemented using a database management system. In such a system all interactions are atomic and are mediated by a transaction coordinator which controls the transactional behaviour of the various computational nodes. Most importantly, the possibility of contention is avoided by making all transactions capable of being "rolled back": undone so that the state of the system is as if the transaction had never happened. So, for example, if two users attempt to book a place for a large

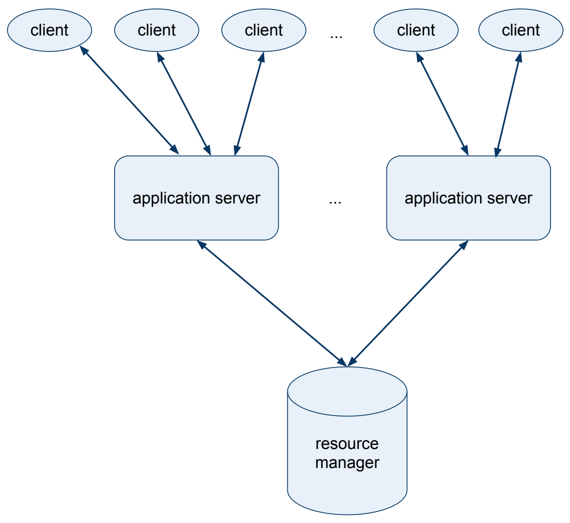


Figure 4.3: Typical abstract architecture of modern commercial software system.

cargo shipment on the same aircraft then only one such transaction will be committed, the other will be rolled back leaving the user, typically, to find another aircraft capable of carrying their cargo. Implementation of transactional management is done by various locking mechanisms, typically a low level pessimistic database lock and higher level optimistic locks [BHG87, Wikb] in order to allow for high user performance.

In the context of environment orientation a refinement is possible. As the external state stored for each agent is only written to by that agent, there is no possibility of contention when updating the environment. That is, as long as agents reading the contents of a particular agent's external state are prevented from viewing inconsistent state information while an update is in progress—that is, updates to the environment are atomic and invisible until committed—then no further locking mechanism is necessary. Several approaches exist to ensure this sort of consistency, the simplest is to ensure that individual updates are atomic using a low-level pessimistic lock and ensure that read-committed isolation[3], at least, is in place.

In addition, these atomic updates of external state by individual agents mean that it is never necessary to roll back a transaction. That is perhaps unsurprising, as

[3]One of the standard mechanisms for defining isolation between transactions [Ree00, Wikc], in this case ensuring that a transaction can only see data that has been committed by another transaction.

the rationale for adopting environment orientation is that the simulation operates as the real world does and it seems as though the real world never rolls back.

With such an overall design, there are many possible implementations of the state repository used in the complex systems simulation which will be discussed later in this chapter. Another possibility is software transactional memory [ST95], an approach to concurrent programming which makes database-like transactional operations available on shared memory. While the guarantees of full atomicity for multi-step operations that software transactional memory provides are not required for environment-oriented simulation, the approaches to lock-free atomic memory updates to shared memory on modern multicore systems are directly applicable to fine-grained simulation, without the considerable overhead of rollback.

Embodiment

In an environment-oriented complex systems simulation, the environment is responsible for managing the state database and providing those facilities that are *embodied* [Ste07] by the environment. Typically, the environment embodies aspects of real-world physics. For example, [HS09] describes how these embodied services may result in information being presented to agents, the clients of the environment, using various topological representations of the information.

As already discussed in section 4.1.2 the choice of properties to embody within the environment, though, is not always clear. The environment server could implement some properties directly or they could be implemented by specialised agents which are not directly associated with agents in the complex system itself.

There are trade-offs to be made here in terms of simplicity and generality, and in terms of efficiency. If all physical properties are implemented using specialised agents, the external state database is effectively just a tuple store (albeit one that will be accessed by, perhaps, many simulated agents executing across a distributed system); this simplifies its implementation and makes it directly applicable to all types of environment. However, these agents increase system load, and may require different patterns of interaction with the environment, complicating the communication patterns of the simulation—for example, you may need to ensure that all environmental properties have been updated before agents can observe the environment.

```
while (true)
{
  barrier.wait();
  Neighbourhood n = env.query(queryText,
                      <parameters drawn from internal state>)
  internalState.update(n)
  env.update(generateExternalState(internalState))
}
```

Figure 4.4: Pseudo-code for agents using a *query oriented* approach and synchronising at each step.

On the other hand, if properties are implemented by the environment itself, this does not complicate the patterns of interaction between agents and the environment—but it requires the environment to be aware of the details of these properties, reducing its generality. It may prove convenient to strike a balance between the two in a practical simulation framework; implement a few common physical properties in the environment itself, but make it possible to extend the simulation with additional properties by writing specialised agents.

Time and fairness

If a simulation's progression were mediated by synchronisation on a global barrier, for example as shown in the modified pseudo code in figure 4.4 then that synchronisation of the communications would give the simulation as a whole a shared sense of granular time: no agent could proceed to the next time step until it has communicated with all its neighbours. This is indeed the approach used in many CoSMoS simulations [BWS05b]. When agents do not directly communicate with each other and do not need to synchronise, as when using environment orientation, this shared sense of time is lost: agents can perform transactions whenever they like, which makes it possible for agents to execute at different virtual rates.

This is, of course, what happens to real-world complex systems agents as is discussed in [HS09]. However, in the real world, no agent can "run ahead" of the others; agents execute in a perfectly fair parallel manner, with the rate of their behaviour only limited by the inherent physical properties of the world.

In a simulation, freewheeling—allowing one agent to rush ahead of others—is not acceptable: a sense of time must be introduced in order to ensure that agents' access to the shared computational resources is scheduled fairly, with no agent able to starve another of execution time.

One approach to implementing this would use the "virtual time" technique used in event-based simulation [Jef85], in which a dimension-less *virtual time* is represented simply as a monotonically-increasing tag tracked by simulation components. A likely implementation would be to tag external states with the virtual time of the agent that generated them. The environment server would then have to be aware of the virtual time of an agent querying the server and to only provide states that "matched" the virtual time of the querying agent.

The definition of "matching" is not provided here and I have not investigated it in detail although I return to it in chapter 9. It is, however, appealing to consider that it could encompass both an exact match—where an agent at virtual time $t = n$ could only receive states at virtual time $t = n - 1$—and a "sloppy" match—where an agent could receive states at a range of times $n - k <= t <= n + k$. In the latter case an agent perceives states from times slightly before and after its own time thereby allowing for some agents to be more alert, and some to be a bit slow on the update; rather as happens in the real world. The phase-regulating "clock" primitives provided by the X10 programming language [CGS$^+$05] may provide some implementation support for this approach.

However, it should be recognised the problem with all these approaches is that they offer an inherently discretised representation of time; regardless of how it is implemented all of the mechanisms conceive an agent as something proceeding in a number of discrete steps, even if those steps are very fine-grained. Nonetheless, the notion of being able to effectively run a simulation without excessive concurrency control and allowing "sloppy" synchronisation is appealing.

Reproducibility

Complex systems simulations are generally seen as providing the ability to perform completely reproducible experiments—providing a clear advantage over experimentation on the complex systems themselves, where reproducibility is generally not feasible, and allowing published results to be directly reproduced by other researchers. However, a core consequence of the use of environment orientation is that it inherently introduces non-determinism into simulations, in particular if the notion of virtual time as discussed in section 4.2.1 is adopted.

I believe, though, that a degree of nondeterminism will be acceptable in many circumstances. Simulation can be considered as a scientific instrument [APS+10a] that we use to understand the behaviour of a system, and like all instruments it has a degree of uncertainty in its results that can be established by calibration. The scientific method is very good at dealing with real-world experiments with nondeterministic behaviour; the same techniques—error bounds, sensitivity analysis, and other statistical techniques—can be applied to interpret the results of nondeterministic simulations.

It is rare that the results of a single run of a complex system simulation are directly useful, just as the result of a single real-world experiment is rarely considered sufficient. Normally a simulation would be run many times with the same parameters, and the results aggregated to give a better understanding of the typical behaviour of the system; this will have the effect of "averaging out" the effects of nondeterminism on the individual results. In addition, permitting a greater degree of nondeterminism will generally speed up the simulation, making it practical to run it more times—and, unlike in the real world, we can ensure that the initial conditions for a set of experiments are always exactly the same.

Most importantly, the real world is not absolutely reproducible. A collection of starlings over Milan does not flock in exactly the same manner from one day to the next. This is inherent in the complexity of the these sorts of systems. And, as the objective of environment orientation is to make the simulations operate in a manner directly attributable to the real world system it is not surprising that the simulations are not completely reproducible.

All the same, when debugging a simulation implementation, being able to reproduce a single run exactly is sometimes useful. This could be supported by giving the programmer a degree of control that allows them to trade determinism against performance—for example, by reducing the degree of concurrency and enabling additional explicit synchronisations when greater determinism is required. Furthermore, techniques exist for debugging nondeterministic concurrent systems [PZX+09] where software is instrumented so that a rough trace of its execution path is retained. Subsequent debugging runs can then be automatically steered down the same execution path, with the trace being iteratively refined with feedback from the programmer until the desired behaviour is reproduced. This approach could be applied to a nondeterministic simulation.

Robustness

The real world—the most complex of all complex systems—is inherently extremely robust. Systems composed of a huge number of independent agents from which useful behaviour emerges will usually continue to function in a wide range of different circumstances: agents are born and die, information is delayed, lost or corrupted, and interactions are complex and unpredictable. While undesirable behaviours exist, such as the auto-immune diseases that appear in organisms equipped with complex immune systems, they are rare; complex systems usually return themselves to some kind of stable state.

Engineered systems fare poorly by comparison, usually demonstrating extreme sensitivity to both initial and changing conditions. Systems built by humans tend to be fragile, with a wide variety of spectacular failure modes [Ris]. This applies even to existing simulations of complex systems, which often display high sensitivity to parameter values and to implementation details such as the scheduling order of concurrent processes.

An ideal complex systems simulation would be as robust as the complex system itself. In the transactional approach implied by the use of environment orientation, each agent's behaviour is largely independent of the other agents in the simulation, and is not affected by the heavy hand of precise time steps and explicit global synchronisation. This decoupling should tend to increase the robustness of the simulations. The flexibility of environment orientation, because of the lack of explicit synchronisation, should allow simulation robustness to be explicitly tested, for example by testing the behaviour of the simulation in the presence of artificially-induced "faults" such as blocked access to parts of the simulation, changes to the order in which state updates are recorded, and forcible introduction or termination of agents.

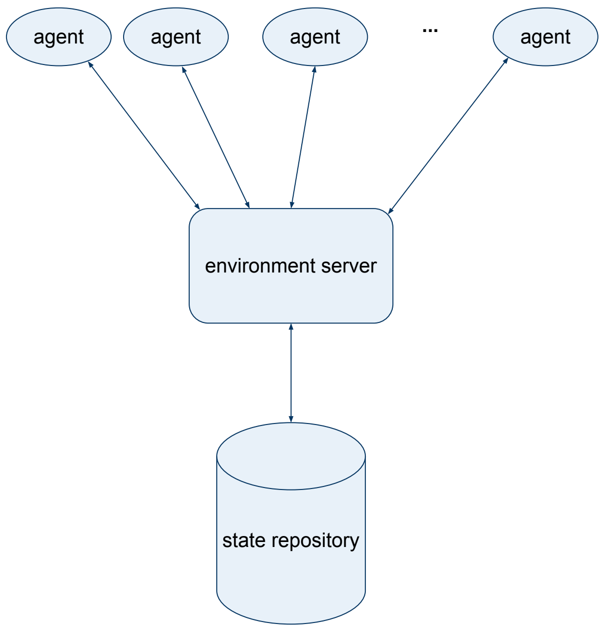


Figure 4.5: Overview of environment orientation implementation.

4.2.2 Implementation techniques

In previous sections I have described what amounts to a transactional approach to complex systems simulation, where the repository at the base of the implementation represents the environment within which the various complex systems agents are interacting. The most likely implementation of this notion is as a multi-tier architecture, again representing the commonest commercial software architecture as has been discussed. Applying those ideas to an environment oriented simulation leads to a layered abstract architecture as is shown in figure 4.5. The layers here are:

- at the bottom, the repository for agent external states,
- the environment server which understands the environmental aspects of the agents such as their spatial position and
- the agents themselves.

That is, though, just a conceptual diagram with little information about the way such a thing could actually be implemented. In the following sections I will discuss the requirements for the components of the architecture and the implementation choices that present themselves from the huge collection of implementation techniques that are available.

Repository

One thing that cannot be seen on a diagram such as figure 4.5 is the thread and distribution structure of the implementation. The original motivation for environment orientation came from the difficulty in avoiding deadlocks, arising from inter-thread, or process, resource contention, in implementations. In the context of environment orientation the only locus of contention between threads is in the repository. As long as the repository provides for atomicity and durability [Gra81] of updates then the system is guaranteed to be deadlock free, meaning that the agents can operate in one or any number of threads with no effect on the system behaviour.

Hence, the repository must provide at least:

1. long term storage (that is, for the period of the simulation runs) of agents' external states,
2. storage of a form that allows the environment server to inspect, elaborate and update specific parts of those states and
3. atomicity of updates, for the reasons described.

A number of implementation techniques could provide for these requirements.

One possibility is that of tuplespaces where a repository for tuples, of any form, is provided along with some simple mechanisms for access and coordination; it would be straightforward to encode an agent's external state into a tuple. Tuplespaces arise from the Linda programming language which was first proposed in the mid 1980s [Gel85] as a new way of handling concurrency and coordination. In Linda all communication and coordination is provided by a "tuple space" which is populated, and examined, by a potentially large collection of concurrently executing agents. Linda provides primitives allowing the connected agents both to query the tuplespace for tuples that match some expression and to block waiting for an appropriate tuple to appear. Specifically Linda's primitives operate atomically and include:

out() which adds a tuple to the tuplespace,

in() which takes parameters describing a set of tuples in which the caller is interested and which blocks until some such tuples are available at which point they are returned and

read() which operates as in() except that the returned tuples are removed from the tuplespace.

As such, the basic Linda model supports the required functionality through the out() and in() primitives. Such implementations would be of the query oriented variety, although other Linda-like systems support more access mechanisms.

The Linda concepts have been implemented in a number of modern programming languages. For example, JavaSpaces [FHA99] provides Linda-like facilities in the Java programming language as part of the Jini infrastructure. Rinda [Sek09] provides tuplespaces for Ruby. TSpaces [LCX$^+$01] is a simple implementation of the Linda ideas within Java from IBM.

The use of a relational database management system (RDBMS) is a further possible implementation mechanism. Relational databases are essentially large containers for tuples. Each table in the RDBMS is a set of tuples with the same layout. Furthermore RDBMSs provide a highly expressive declarative query language (SQL [ISO99a, ISO99b]). As such they provide an attractive mechanism for the query oriented approach to the abstract architecture.

It is less clear how an RDBMS could be used for the subscription oriented architectural pattern. RDBMSs do support mechanisms that are capable of use in this manner (typically, triggers). However, they are clumsy in use and probably not suitable for the very flexible scenarios of complex systems simulations.

The generality of RDBMSs often, though, imposes a particularly exacting set of constraints on users due to the need for precise schema definition and management. The various NoSQL databases (such as Cassandra [Fou]) could well offer the right solution for large scale simulations. However, I have not examined these techniques.

A rather more straightforward implementation, and that adopted for the validation of environment orientation that will be described in later chapters, is to use conventional object-oriented collections. In this case each external state is represented as an opaque object (the structure being defined by the agents) which would be directly stored in such a collection. Such collections can be made to operate atomically. For example, an implementation of a repository that used a Java HashMap would merely have to declare its accessing methods as *synchronized* to achieve the desired level of atomicity.

The requirement for the environment server to augment the states with additional information can be supported using the Decorator pattern [GHJV95] as will be described later.

Publish/Subscribe systems

The publish/subscribe pattern [BJ87, Wikf] is frequently supported by enterprise middleware, in particular by message oriented middleware such as the Java Message Service [RMHC09] which provides publish/subscribe facilities for users of the Java 2 Enterprise Edition. The publish/subscribe pattern provides for a server to distribute information on a number of *topics* to a number of connected clients. The pattern is often used, for example, in financial trading systems where some clients might require to be informed of changes in the prices of particular instruments when they occur. This is a very similar situation to that described here as subscription oriented. A topic here could be, for example in the context of a bird flocking system, “the state of agents in the vicinity”. Whenever one of those agents does indeed move the agent that registered the topic could be informed of a set of new tuples of information.

Publish/subscribe systems are used commercially in situations where there is a very high data rate, such as the instrument/price situation described above. As such they are also suitable for distributing information in a complex system simulation.

Process oriented programming languages

Process oriented programming is at the heart of much complex systems simulation and is a core part of the CoSMoS project. The environment oriented architecture could be implemented using a process oriented language such as occam-π [WB04]. This is the language used for the models of space described in [ASB$^+$08a]. Using occam-π to implement simulations with the environment oriented architecture would ideally require the definition of a set of standard libraries that would hide many of the internal details, and allow the programmer to operate at a higher level of abstraction, purely in terms of things like tuples and queries.

Threading and locking

An implementation of environment orientation would likely execute on a computational platform consisting of more than one processor core, whether that be as a consequence of the use of multi-core processors or by execution on a distributed platform. Or, more likely, both.

As such execution would be required to take place simultaneously in a number of threads[4]. As the locus of locking in an environment orientation implementation is entirely in the state repository there should be no particular dependence on the threading structure, as long as locks do not persist in the repository and the computational performance is fully used.

The most likely structure is to create a fixed number of threads, of an amount reflective of the physical characteristics of the computational environment, and to locate each agent within a particular thread. As an agent is the prime mover of simulation behaviour that will mean that each agent and the interactions it has with the environment server and, indirectly, the state repository execute within a single thread.

This does mean that each thread will have to schedule the activities of the agents it supports, but that can easily be done either serially by simple code or by some OS provided facilities.

4.3 Environment Orientation

The notion of using environment orientation to build simulations of complex systems brings with it some requirements for the software architecture that implements the concepts. This architecture can be constructed using many different techniques already used and documented in the software engineering literature and often widely

[4]I am avoiding any processor or operating system specific interpretation of "thread". The word merely implies a body of sequential code that operates at the same actual time as another such body.

used in practice. These techniques are, to a large extent, those of sharing collections of objects around a networks of processes using transactional control to guarantee characteristics such as atomicity and durability.

As such, there are some appropriate implementation routes. In the next chapter I distill the various approaches and the concepts discussed in chapter 3 into specific requirements for an implementation. I describe a specific implementation of those requirements, using the architectural components discussed in this chapter. This platform is used for the experiments described in chapters 6, 7 and 8.

Chapter 5

Validation platform

Having described the rationale and general approach to implementing a complex systems simulation entirely based on environment orientation it is necessary to actually build some simulations. In order to do this I constructed a flexible implementation platform that allowed me to run a variety of different experimental simulations. In this chapter I describe the implementation of the experimentation platform. Subsequent chapters address the results of experiments performed using this platform.

5.1 Requirements

The platform should satisfy the following requirements. These requirements arise from a number of distinct sources, specifically:

1. the required services discussed in chapter 3,
2. the possible implementations discussed in chapter 4,
3. the concepts of sound software engineering, in particular the ability to be able to extend the platform for a number of experimental scenarios and
4. the notion that a number of repetitive experiments would have to be executed and subsequently analysed.

From these considerations arise a number of discrete requirements:

Agent and types The platform should support the (pseudo) parallel existence of a number of agents, each of which is of a specific type although not all agents should be required to be of the same type. Any specific agent should be able to have its own internal state, invisible to rest of the platform.

External states Agents executing in the platform should be able to define their own external state much of which may have a detailed structure opaque to any other part of the platform other than the defining agent.

Environments Each agent should be able to exist in one or more environments, where an environment is something that accepts an agent's external states for storage and processing. Environments should be able, in order to represent some aspects of the environment's physics, to determine basic aspects of agents' external state. For example, in a flocking simulation position and velocity should be so accessible. Similarly, environments should be able to extend specific external states with additional information known to the environment but not known to the state's originating agent. For example, an agent might not know its absolute position in space but the environment should know this and should be able to retain that information along with the agent-provided state. Information from an environment should be provided to the interacting agents by providing them with a summary of the relevant parts of the environment: each agent's neighbourhood. This emphasises the role of the agent as a simple query/react/update unit and limits the scope for building "magic sensors" into the agent which could encode functionality beyond the capabilities of the complex systems agent. For example, if an agent could interact with the entire environment it might be able to "see" much further than would properly be allowed.

Repository Whether implemented as a separate component or not, the environments should store the external states in a manner that is thread-safe and extensible to very large sets of external state objects.

Instrumentation Ideally, to support experimentation it should be possible to extend any platform instance with additional facilities allowing bulk inspection of a simulation's state. For example, this should support the ability to visualise the position of all agents even if the agents themselves are not aware of the overall state of the simulation.

Simulation context In order to support repeatable experimentation, it is desirable that a simulation should support the notion of being able to be controlled. This would allow a simulation to be initialised according to some defined parameters and to be started and stopped as required by a user.

User interface It is desirable that the platform should support a user interface allowing a user to control a simulation, providing the various initialisation parameters and observing the subsequent simulation execution.

5.2 Design and implementation

In this section I present the overall logical design of the platform that satisfies the requirements details in section 5.1. The design is represented as a straightforward object-oriented design using the UML to represent the specification model [CD94] for that design.

As is described in chapter 4, there are a range of possible implementation choices. I deliberately chose something simple and relatively easy to implement because my intention was to investigate the essential idea, not to search for maximum possible performance. Hence, I chose to implement the platform using Java [AGH05] as the implementation language. Java includes a rich set of primitive implementation classes which ease the implementation. In particular, the threading classes and primitives [OW04] ease the use of parallelism which is a necessary characteristic of the required platform.

5.2.1 Overall structure

Following the requirements described in section 5.1 the overall view of the design is shown here as figure 5.1.

The classes in this diagram are as follows:

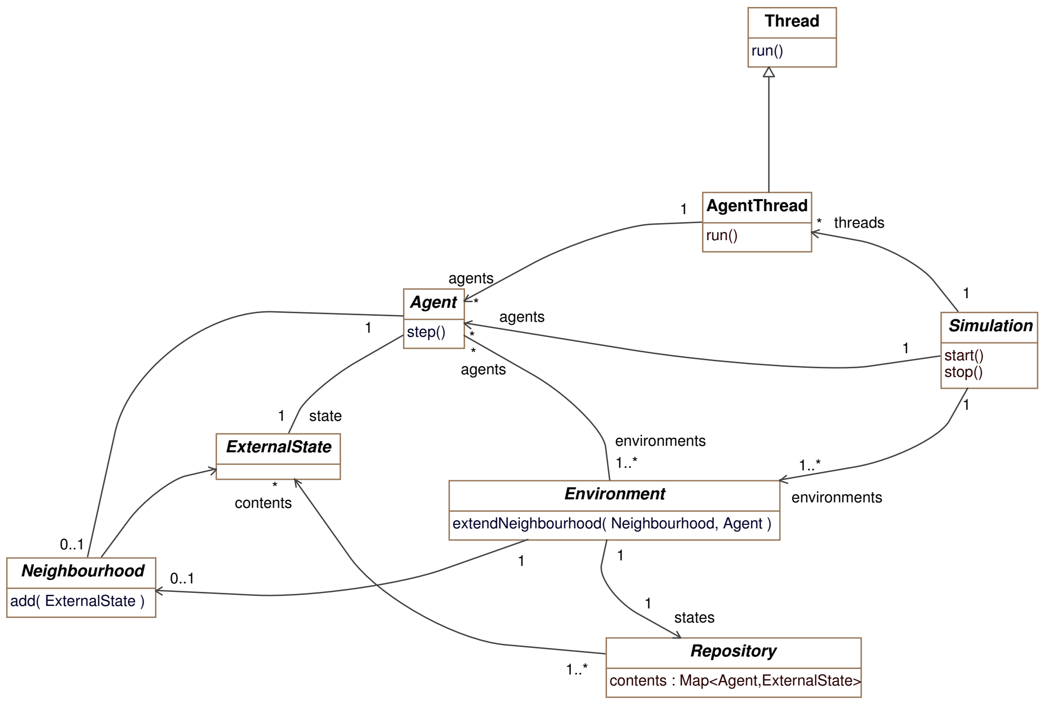


Figure 5.1: UML class diagram showing overview of environment orientation experimentation platform. Although not strictly correct UML this diagram, and other similar ones in this thesis, uses curved lines for associations to emphasise their distinction from generalisation and to avoid confusion with the straight edges of class boxes.

Simulation represents the overall simulation and provides operations for initialisation and control of a simulation.

AgentThread a Java thread class (that is, a specialisation of Thread). Each instance of a Simulation coordinates a number of AgentThread classes. The specific agents that are coordinated by particular agentThreads is determined by the simulation itself. The intent here is to be able to change the relationship between the number of agents being serialised by each thread.

Agent which is the core of a complex systems simulation, each instance representing a single agent in the complex system. The platform is intended to support a query oriented approach with each cycle of query/update being performed by the Agent's implementation of the step() operation.

Environment which represents a single environment within which Agent instances indirectly interact. The platform permits, as has been discussed, a number of Environment instances each representing a different means of agent interaction.

ExternalState representing the external state of a specific Agent instance.

Repository which is the central state repository containing the external state instances for each environment.

Neighbourhood which represents the neighbourhood of an individual agent when it performs a simulation step. It is this class that implements the notion of limiting the scope of access of each agent, reducing the scope for inclusion of "magic sensors".

The classes shown here are just the overall structure. Many other details will be described in the following paragraphs. The basic operation of the platform is described by the sequence diagram in figure 5.2. This shows that when a Simulation object is created it then creates, using various user-defined parameters such as one defining the number of agents in the simulation, the agents, threads and environments required by the simulation. Subsequently, the simulation is run which

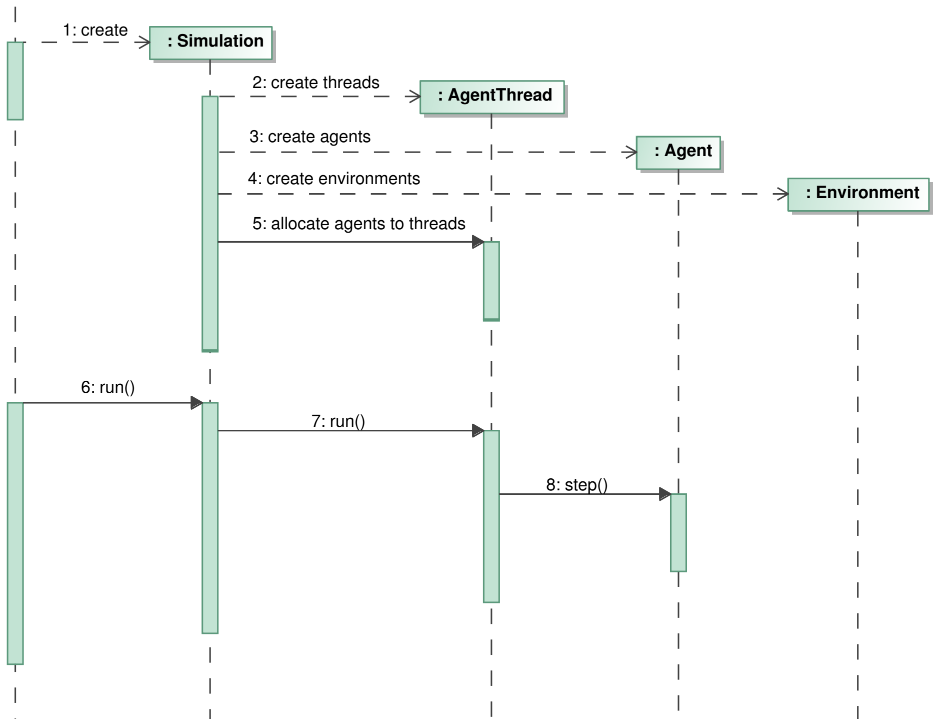


Figure 5.2: UML sequence diagram showing initialisation of a simulation using the environment orientation experimentation platform. This diagram is not strictly correct UML as it shows instances of abstract classes. Nonetheless, it does should the pattern of interaction that is intended to occur.

invokes the run() operation of each of the threads. Each thread then runs continuously invoking the step() operation on each of its agents until eventually the user stops the simulation.

The experimental platform uses the query oriented approach to simulation so each agent has an implementation of the step() operation. This method performs one cycle of query oriented behaviour: the agent queries its environment, based on that and its own state, makes a decision how to update its internal state and then its external state is updated in the environment.

Agents exist in a number of Environments and the contents of these Environments is used, at each simulation step, to construct a Neighbourhood which determines the Agent's subsequent actions. Each Neighbourhood contains a collection of ExternalState objects which an Agent's Environments deem relevant for the Agent

0

```
1: Neighbourhood n = new Neighbourhood(this, proximity);
2: for Environment e in environments do
3:     e.extendNeighbourhood(n, this)
4: end for
5: this.updateInternalState(n)
6: for Environment e in environments do
7:     e.updateState(this)
8: end for
```

Figure 5.3: Pseudo-code for the method that implements an Agent's step() operation.

and is constructed with the parameters that are necessary for determining the set of agents that are in the neighbourhood of the subject agent. For example, an Environment that represents simple three dimensional Cartesian space would just return the external states for other Agents that were in range of the requesting agent.

Pseudo code for the body of the method that implements an Agent's step() operation is shown in figure 5.3. As can be seen, the core parts of the behaviour are delegated to the environment, as would be expected of an environment oriented implementation. Specifically, the environment is tasked with updating an agent's neighbourhood; that is, the environment makes the decisions about which other agents are in the neighbourhood of an agent. When a neighbourhood is constructed, information is provided to determine which other agents are in the neighbourhood, in the example shown the proximity—within which it is necessary to be to be deemed to be "in the neighbourhood"—is provided.

State decoration

Rather than an agent just supplying its updated external state to environment, at the end of the step, the agent asks the environment to update its information about the agent. This means that the environment can determine what actually needs to be done with respect to any additional information that needs to be retained in addition to that available from the agent itself. For example, in a flocking simulation the environment needs to retain the absolute position of each agent with respect to its coordinate system. However, the agent itself does not need to know the absolute position (section 3.2.2). The environment therefore needs some means of enhanc-

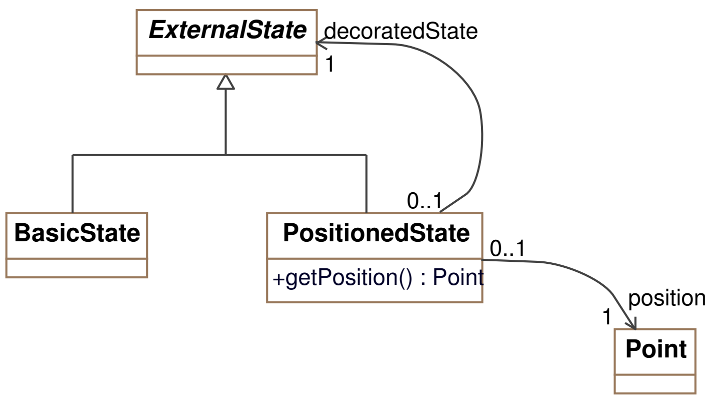


Figure 5.4: UML class diagram showing overview of state decoration.

ing an agent's external state with its absolute position, which will be needed when agents invoke the extendNeighhourhood() operation.

This requirement is implemented using the Decorator pattern [GHJV95], the essential structure of which is shown in the UML class diagram in figure 5.4.

This shows that in the platform the ExternalState class is abstract with a default implementation provided by the concrete subclass BasicState. In addition a number of *decorators* are provided; one of these, PositionedState, is shown on the diagram. This is a further concrete class, which also implements the contract implied by ExternalState but adds an additional property, in this case one for the absolute position of an agent described by an instance of the Point class.

This would be used, inside the implementation of the updateState() operation in the manner outlined in the pseudo-code shown in figure 5.5. The environment calls back to the agent to retrieve its external state and then decorates that state, by constructing a new instance of the class PositionedState that encapsulates the agent's provided state, with the existing position of the agent translated by whatever movement the agent had just performed.

This gives an outline of how the decoration of external states works for the simple example of retaining the absolute position of agents. In reality, the experimentation platform provides for a collection of possible decorations including the position of agents and their speed and their altitude in a hilly landscape. The Decorator pattern allows decorated states to be subsequently further decorated meaning

```
ExternalState state = agent.getState()
Point position = this.findState(agent).getPosition()
position.translate(state.movement());
ExternalState decoratedState = new PositionedState(state, position);
this.saveState(agent, decoratedState);
```

Figure 5.5: Pseudo-code for an environment's updateState() implementation.

that an arbitrary collection of decorations can be added.

For example, in the case where a simulation takes place in a hilly landscape then as well as there being a (plan) position for an agent there is also an altitude provided by an implementation of the Landscape interface. The environment uses this implementation to determine the altitude of an agent from its position when its state is updated. This altitude can then be saved away inside a specially decorated state.

A further positioning aspect of states is that an offset state decoration is provided. This is used by environments when populating a state's neighbourhoods. In this case the states placed in the neighbourhood must contain the offset from the position of the agent whose neighbourhood is being populated. That is, the environment calculates the offset of each agent in a neighbourhood with the position of the requesting agent and makes the offset determinable by the agent. In this manner the agent may completely ignore its absolute position as the agents in its environment are all expressed in a manner relative to its position and, if necessary, velocity.

State repository

The experimental platform stores the external states of agents, possibly decorated in the manner described, in instances of classes that implement the Java Map interface. In essence this means that given an agent (or in practice the identifier of an agent which for simplicity are sequentially allocated integers) the state for that agent can be located.

The maps used are embedded within a class which implements the contract described as a Repository and acts as a facade [GHJV95] to the map. Repository is a simple interface allowing clients, such as an environment, to add, remove and update states stored in the repository, and hence in the embedded map.

I implemented two different realisations of Repository, a simple straightforward one (ObjectsRepository) and a more complex one (PartitionedRepository) that reflects the fact that many simulations take place in a landscape that may be partitioned in a two-dimensional manner. A partitioned repository is actually a two-dimensional collection of simple repositories along with some code that calculates which simple repository will contain a particular state based on the position of the agent for that state.

As discussed in section 4.2.1 implementations of the Repository interface are the only place in the platform where decisions are made concerning synchronisation between different threads. The specific decisions made are very simple, limiting a single thread to being active in a repository at any time. As all threads operate completely independently of each other, apart from a shared dependence on the repository, this means that simulations are entirely free of deadlocks.

Current versions of the experimental platform provide facilities to access the state repository by position. For example, the environment can ask for either all the agent states in a repository or those that are physically proximate to a given position. Other implementations could provide further facilities, perhaps based on a query language or Linda-like tuple matching. I have not investigated such approaches, as I wanted to restrict myself to elaborating and testing the basic environment orientation concept and its implementation.

5.2.2 Instrumentation

The preceeding discussion addresses the construction of the platform to support an environment oriented complex systems simulation. However, in addition to the simulation itself, the platform needs to support some sort of instrumentation. There are two obvious requirements for instrumentation: some form of visualisation is necessary and it is likely that a specific simulation would have specific metrics (for example, the size of a flock in a bird flocking simulation) which must be calculated.

These requirements are essentially for different views of the contents of the repository. For visualisation, some code needs to access all the states in the repository—including the decorations that show absolute positions—and use that to populate some screen display. Metrics calculation is essentially the same thing; some code

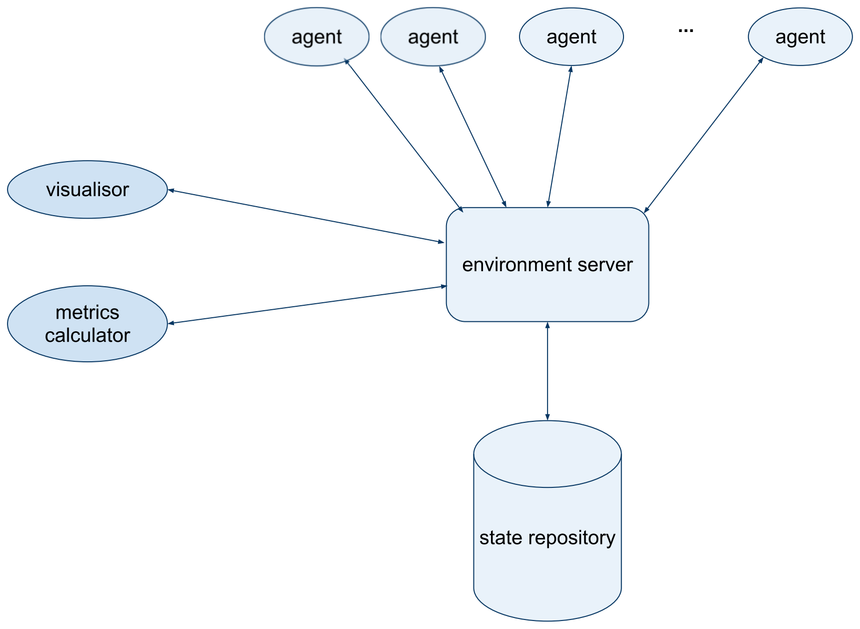


Figure 5.6: Overview of experimental platform with instrumentation.

needs to access all the states in the repository in order to calculate the value of the metric.

That is, both of these two requirements are satisfied by two further clients of the environment, in the manner summarised in figure 5.6.

Similarly, these new clients fit into a similar design structure, as shown in figure 5.7. In this case I chose to implement the instrumentation agents directly as Java threads as there is no requirement for multiplicity as exists for the Agent objects. Again, there is no issue with concurrency because the control of this still resides within the environments themselves and their state repositories. That is, the same structure is used for the instrumentation as is used for the agents themselves; all of these activities are accessing the same shared environments.

5.2.3 User interface

The preceding discussion applies purely to the internals of the platform. In addition to the structures I have described the platform provides a user interface that allows control of the many parameters appropriate to any particular simulation including, for example, the number of agents in a simulation, the behaviour of the agents and

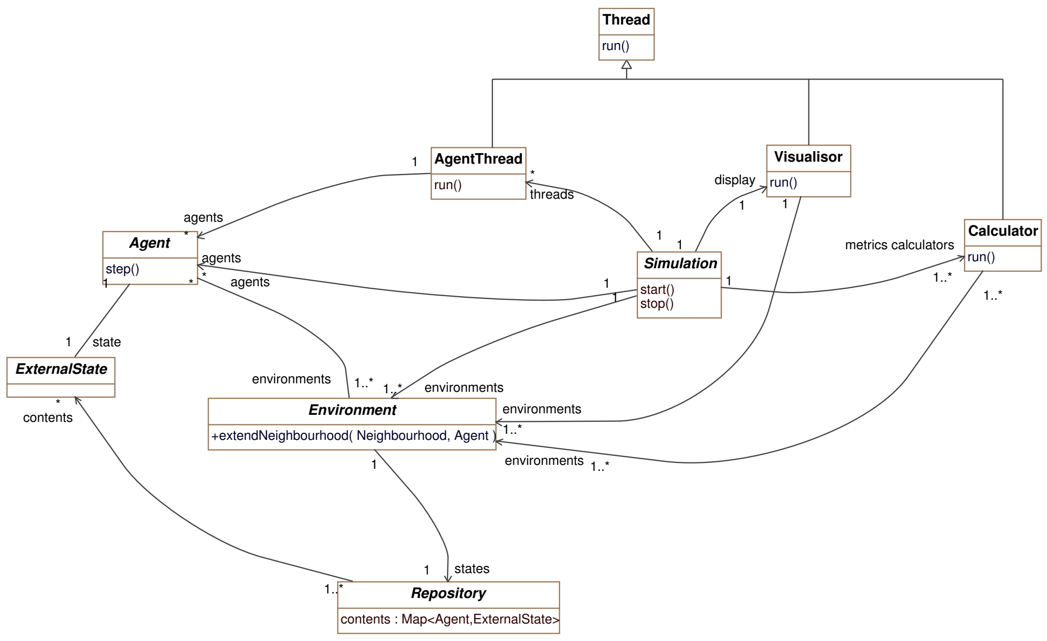


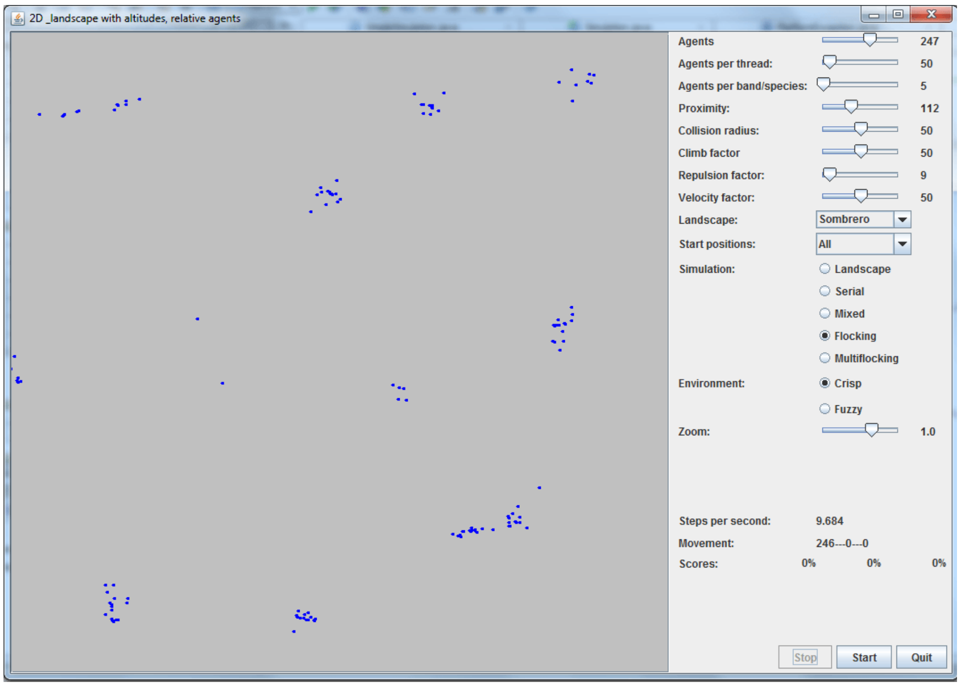
Figure 5.7: Overview of platform with instrumentation classes.

the landscape within which they are operating.

One such user interface is a GUI constructed using the Java Swing classes and a sample view of the interface is shown in figure 5.8. This particular figure shows a simulation of bird flocking. The dots in the display to the left each represent an instance of a BirdAgent class, a subclass of the Agent class discussed earlier. An intuitive observation of these agents is that they are indeed clustering into flocks.

The controls such as the sliders at the top right of this interface define the various simulation parameters. The graphical display to the left and the statistical information at bottom right (for example showing the number of updates per second) is displayed using the instrumentation facilities described earlier in this chapter.

In addition to the graphical user interface I built a command line interface, which interacts with exactly the same simulation classes, in order to allow simulations to be run without user interaction.



Agents: 247
Agents per thread: 50
Agents per band/species: 5
Proximity: 112
Collision radius: 50
Climb factor 50
Repulsion factor: 9
Velocity factor: 50
Landscape: Sombrero
Start positions: All
Simulation: Landscape [✓ Flocking] Serial Mixed Multiflocking
Environment: [✓ Crisp] Fuzzy
Zoom: 1.0
Steps per second: 9.684
Movement: 246---0---0
Scores: 0% 0% 0%

Figure 5.8: Graphical user interface for platform.

5.3 Satisfaction of requirements

Having outlined the design of the validation platform here I return to the requirements outlined in section 5.1 and describe in outline how the platform satisfies each requirement.

Agents and types The Agent class described is a contract [Mey00] for further types of agents that can be implemented by subclassing. As part of the contract implementations of Agents must integrate into the structure defined for agents and threads, ensuring that new agents will execute in a simulation.

External states Similarly, the ExternalState class describes give a contract for defining specific classes of state for the use of specific classes of agent. This contract allows new external states to exist within the simulation framework.

Environments The platform defines the contract for an implementation of an Environment to satisfy in order to fit into a simulation. The used of the Decorator pattern allows classes that satisfy the requirements of the ExternalClass contract to be retained in an environment and contribute to the manner in which that environment's contents are delivered to agents.

Repository The Repository platform class provides a contract which can be satisfied by many implementations each of which is the locus of concern with respect to the concurrency that exists in the simulation.

Instrumentation, User Interface These requirements are both satisfied by the concept of instrumentation and visualisation classes.

Simulation context The Simulation class again provides a contract for controlling the context of a simulation. Specific implementations may implement this contract in order to provide facilities such as initialising, starting and stopping a particular simulation.

Chapter 6

Experimental validation

In this chapter I provide experimental results aimed at some basic hypotheses relating to environment orientation:

1. A simulation constructed in an environment oriented manner within a single environment functions in that emergent properties expected from earlier simulations also appear in the environment oriented simulation with no direct communication between the agents.

2. Simulations can be extended without reworking the simulation's conceptual model to work with multiple environments. The expected emergent properties still appear.

3. Simulations can be extended to include representations of the physical world without major modifications.

These experiments are performed by extending the simulation platform described in chapter 5. In each case I describe the extensions made to the platform to provide the new facilities.

6.1 Flocking in a single environment

The complex system chosen for the initial experimental evaluation was an implementation of Reynold's boids [Rey87] algorithm.

Reynolds' algorithm describes how an agent representing a bird can behave according to a simple set of rules so as to reveal flocking behaviour within a collection of such agents. In outline, these rules are:

- Each bird should move towards the centre of mass of the other birds it can see in its vicinity.
- Each bird should attempt to make its velocity the same as the overall velocity of the birds it can see in its vicinity.
- Any bird should ensure that it flies away from another that is near to it so as to not collide with that bird.

The essence of these rules is that at all times the simulation generates a movement vector for bird, formed from the vector sum of the movements due to each rule. This vector is applied to the position of each agent at each simulation step.

6.1.1 Experimental hypothesis

For this first experiment the experimental hypothesis is that **the known emergent property, flocking, would arise from a simulation performed in an environment oriented manner**. In order to investigate this it is necessary to define what is meant by a flock. For this experiment this was defined as follows. In order to be "flocked":

a. an agent must retain over 20 simulations steps a minimum of 10 agents in its vicinity and

b. at least 80% of the set of neighbourhood agents must be the same agents across all those steps.

6.1.2 Platform extension

In order to perform this simulation, the platform was specialised to experiment with flocking by writing several concrete subclasses of existing platform classes, satisfying the various contracts discussed in section 5.3; the end result is summarised in the class diagram in figure 6.1. The new concrete subclasses included:

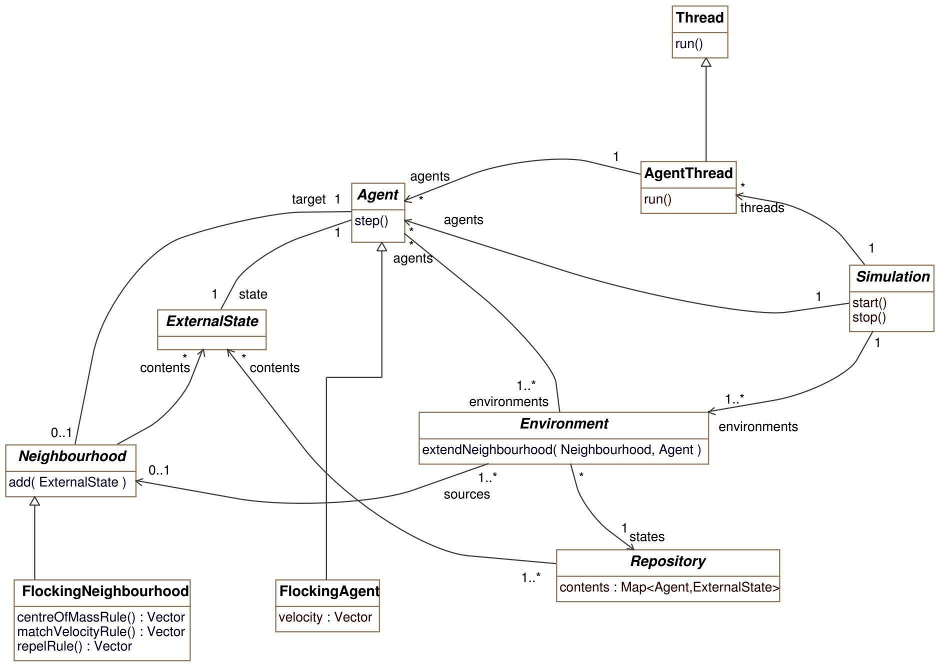


Figure 6.1: UML class diagram showing overview of flocking variant of experimentation platform.

FlockingAgent A subclass of Agent which adds the notion of a velocity to the basic Agent. That is, each agent knows its velocity and at each step it updates the velocity by some amount according to the other agents in its neighbourhood

FlockingNeighbourhood A subclass of Neighbourhood that adds the notion of being able to calculate vectors that correspond to the results of the three flocking rules as described above. The agent sums these vectors so as to update its velocity.

The FlockingAgent method that implements the step() operation is summarised in the pseudo code shown in figure 6.2. As can be seen, this delegates to the FlockingNeighbourhood the responsibility for calculating the results of the flocking rules. The velocity vector in the agent's internal state is updated, at each step, with the perturbation that arises from the agent's neighbourhood.

In order to determine if the notion of an agent being "flocked" was being achieved, as described in section 6.1.1 an instrumentation class, as discussed in section 5.2.2,

```
Vector perturbation;
FlockingNeighbourhood n = new FlockingNeighbourhood(this, proximity);
for Environment e in environments do
    e.extendNeighbourhood(n, this)
end for
perturbation = n.centreOfMassRule()
        + n.matchVelocityRule()
        + n.repelRule();
velocity = velocity + perturbation;
for Environment e in environments do
    e.updateState(this)
end for
```

Figure 6.2: Pseudo-code for the flocking agent's updateInternalState.

was added to the platform that produced information describing how many of the simulation's agents were flocked.

6.1.3 Simulation results

The simulation of the flocking complex system has a potentially large number of parameters, as can be seen on the controls on the user interface shown in figure 5.8. As a test of the platform, I ran a number of tests of flocking, varying a single one of these parameters. This is the parameter for neighbourhood proximity, determining whether other agents are "in the neighbourhood" of a specific agent as is shown in figure 6.3.

I used the instrumentation class described in section 6.1.2 to determine the proportion of the agents in a simulation that were "in a flock" after a large number of simulation steps (two million in this example). I then ran a number, 20 for each value of the proximity, of simulations to check how successful the agents were at ending up in a flock.

The chart shown in figure 6.4 summarises the results of these simulations where the only parameter being varied is that for neighbourhood proximity. In each simulation a set of 300 agents were introduced into the central 500 unit radius circle of an infinitely-sized flat world. The chart shows boxplots showing the proportion of the agents that were in a flock at the end of each simulation run. In this simulation,

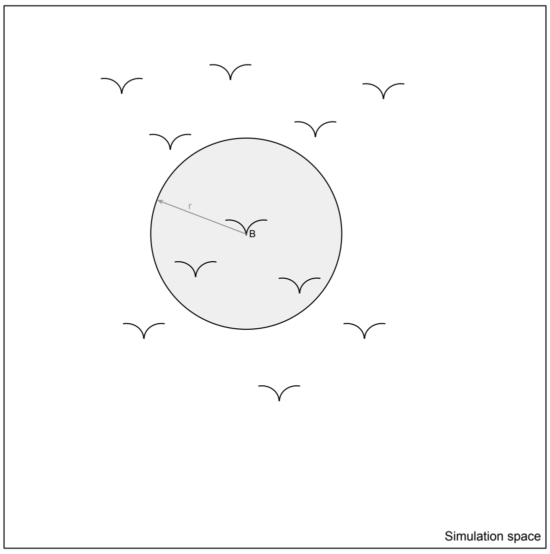


Figure 6.3: In a flocking simulation the circle shown is the radius of proximity, r, of the bird B. There are two other birds within this neighbourhood.

the agents were allocated, at simulation start-up time, to one of a fixed number of Agent threads. The number of threads was chosen to fit the machine architecture in use. However, experimentation shows, as expected, that this parameter has little effect on the result of simulations.

As can be seen, the agents are indeed forming into flocks, supporting an intuitive reading of figure 5.8. Changing the proximity parameter improves, as would be expected, the ability of the flocks to incorporate more of the agents into the emergent, flocking, property. At small proximities, that is where an agent's neighbourhood would be expected to include only a few states due to other members of the simulation, flocking does not occur. That is, the agents just wander off in the simulation world and do not find any other agents with which to flock.

6.1.4 Discussion

This flocking example validates the core hypothesis; that the environment oriented simulation displays the same emergent property as was expected from other simula-

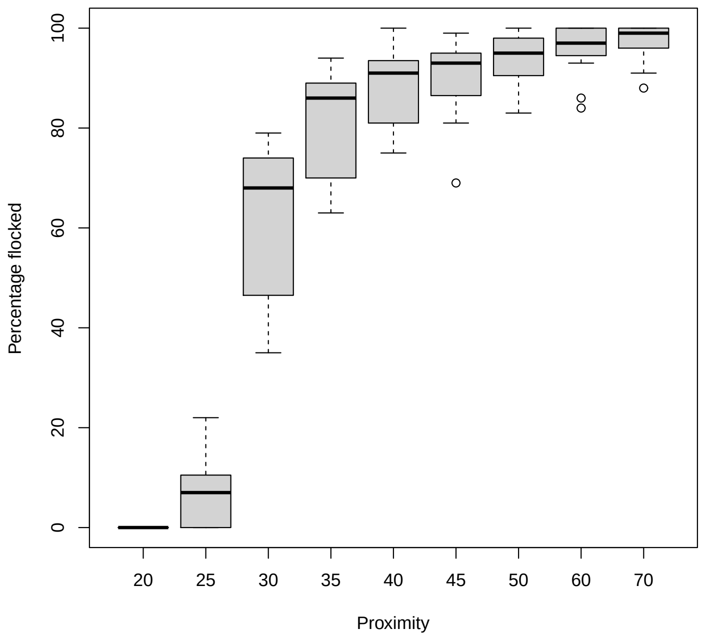


Figure 6.4: Box plot showing the result of flocking with a single environment that builds a neighbourhood based simply on the proximity of agents from the querying agent. Each box shows the lower quartile, median and upper quartile of the percentage of agents that ended in a flock over a set of 20 simulation runs for each value of $proximity$, the values of this variable being shown in the horizontal axis. Each simulation ran for 2 million simulation steps and started with an initial population of 300 agents, randomly distributed in a 500-unit radius circle around the origin.

tions of the same complex system. That is, the agents do indeed form into flocks in the manner expected of the Boids algorithm, although in this case entirely without direct communication between the agents. That is, in the simulation I have shown, the agents essentially know nothing of each other, other than the notion that other agents might exist. All they do is interact with the environment.

Further, in this spatial simulation the simulation agents have no knowledge of their position, operating in a purely solipsistic manner, restricting their interactions with the environment alone. In contrast, the environment is a real participant in the simulation, not merely a repository. It understands the actual positions of the agents and is able to efficiently populate the agents' neighbourhoods with the states of local agents.

6.2 Flocking in multiple environments

As has been discussed in section 3.2.4 complex systems may well include a collection of heterogeneous agents moving within a world where their inter relationships may be expressed as appearing in multiple environments.

The flocking model discussed so far is simple in that it has but a single environment. However, in the context of flocking birds it is conceivable that the flocking behaviour of birds depends on both visual perception of other birds and also auditory perception.

In this case these two environments, visual and auditory, have different spatial characteristics. In the visual environment one agent can accurately perceive the three dimensional position of another assuming that the distance between the two agents is sufficiently small. The auditory environment allows an agent to know about the existence of other agents (for example of the same species) in the system but is less precise at describing exactly where the agents are; one can tell that a blackbird is part of the dawn chorus by its distinctive song without actually perceiving exactly where it is.

The experiment described in this section extends the prior single-environment experiment to one supporting multiple environments. Any agent in the simulation discussed here is simultaneously in two environments. In addition to an environment of the form used in the prior experiment, that is one in which an agent can only perceive other agents in its immediate neighbourhood, it also shares a second environment with a relatively small number of other agents. Agents in this second environment can perceive each other regardless of proximity.

The conceit[1] here is that the former environment represents vision whereas the latter represents auditory information with the set of agents sharing the environment representing a separate “species”.

In this experiment In this initial experiment the “seeing” agent perceives the exact position of all of the agents in the species environment, even though they are potentially distant. In a further experiment, described in chapter 7, the ability to describe positions less accurately is investigated.

6.2.1 Experimental hypotheses

There are two main experimental hypotheses under evaluation here:

1. Complex systems emergent properties can appear in an environment oriented simulation that uses a number of environments.

2. As a consequence of an individual agent being able to “see” more of the set of agents sharing the environments it should be the case that the flocking should be more successful. That is that a larger proportion of the agents in a simulation should end up flocked.

Further, this experiment should show that the simulation platform is easily extensible to a simulation which uses multiple environments.

6.2.2 Platform extension

The simulation platform described here and summarised in figure 5.1 explicitly supports the notion of multiple environments. That is, as can be seen from the UML diagram, each agent is placing its external state in one or more environments and using a neighbourhood derived from all those environments to determine its future interactions.

[1]Although multi-species flocks are well known [KT04] the notion of separate species used here is not intended to be realistic with respect to bird behaviour. Rather, it is used as a convenience for exposition.

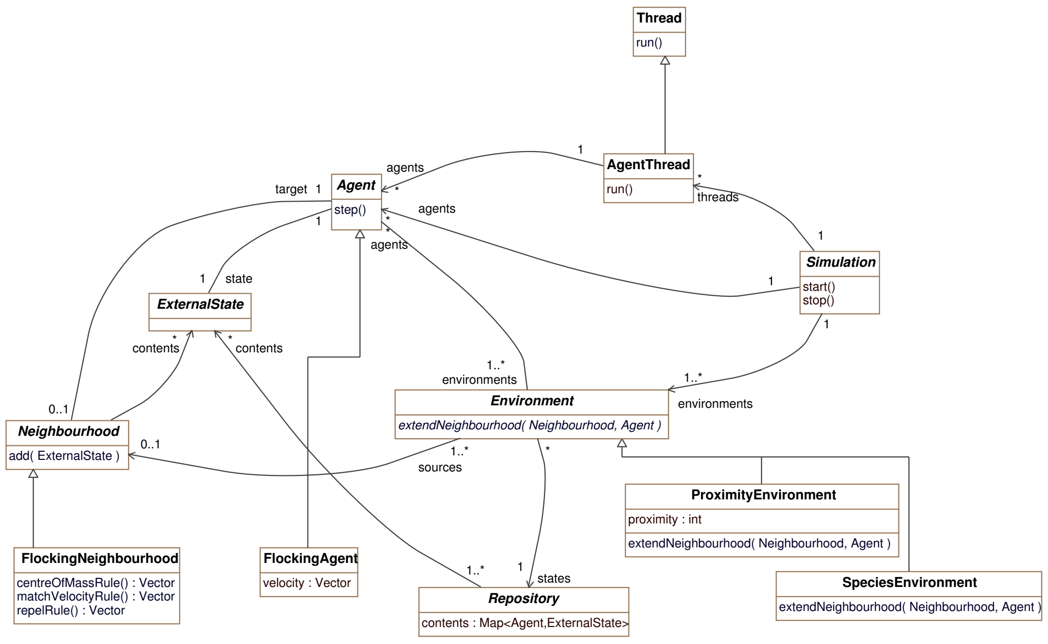


Figure 6.5: Overview of the platform configured for flocking in multiple environments. Each agent exists in a single proximity environment and, optionally, a species environment.

Here, the single environment flocking simulation is extended to support a simple version of the multiple environment concept. Again, further classes are added to the implementation implementing the contracts discussed in section 5.3.

Specifically, as in figure 6.5 there are two new implementations of the Environment contract. The Simulation class used this time knows about the possibility of the multiple environments and is responsible for setting up, and executing, each simulation with each agent in two environments, a single ProximityEnvironment and a single SpeciesEnvironment.

6.2.3 Simulation results

The boxplots in figure 6.6 summarise the results of performing simulations of flocking using the simple notion of multiple environments as described. These data are presented in the same general form as those in figure 6.4 of which they are an extension as the earlier results are essentially for a simulation where the number of agents in the species environment is one. These plots therefore repeat the earlier plot in a matching form.

As before it is clear that many of the agents in the simulation are, indeed flocking with each other. The data are, though, more confused likely as a consequence of the more complex experimental situation.

6.2.4 Discussion

As discussed in section 6.1.3, in a single environment there is a simple rule, that the further an agent can see the more likely it is to end in a flock. That is, the larger the $proximity$ attribute of the ProximityEnvironment the more agents will end up in a flock. There is, though, a considerable degree of uncertainty due to the random initialisation of the space. In particular the distribution of the results at small proximities is large. This effect appears because if an agent moves away from others at the beginning of the simulation then it might well move to a position where it never again interacts with the other agents; it just moves away across the world never seeing all the other agents. But, when the $proximity$ is larger, that is when the agents can "see" further, they are more likely to end up in a flock.

When other environments are added the data are, at first, rather more confusing. What is clear is that the agents are continuing to flock; further, the number of agents in the species environments is having an effect on the success of the flocking.

Looking harder at the data, it can be seen that at higher proximities, in particular above $proximity = 35$, then adding more agents to the species environment improves the amount of flocking. In essence this is not surprising as the multiple environments may well allow an agent to "see" further than it could previously.

However, with a small value for proximity, in particular at $proximity = 25$ and $proximity = 30$ there is much more variation in the data. Although the median value for the flocking percentage increases as the number of agents in the species environment increases the shape of the distribution is much flatter.

Although not easy to see in the box plots, observation of the simulation graphical display showed that this was an effect of the random allocation of agents to the species environment. For example, if it happens to be the case that agents on opposite sides of the non-cyclic "world" are in the same species environment then it

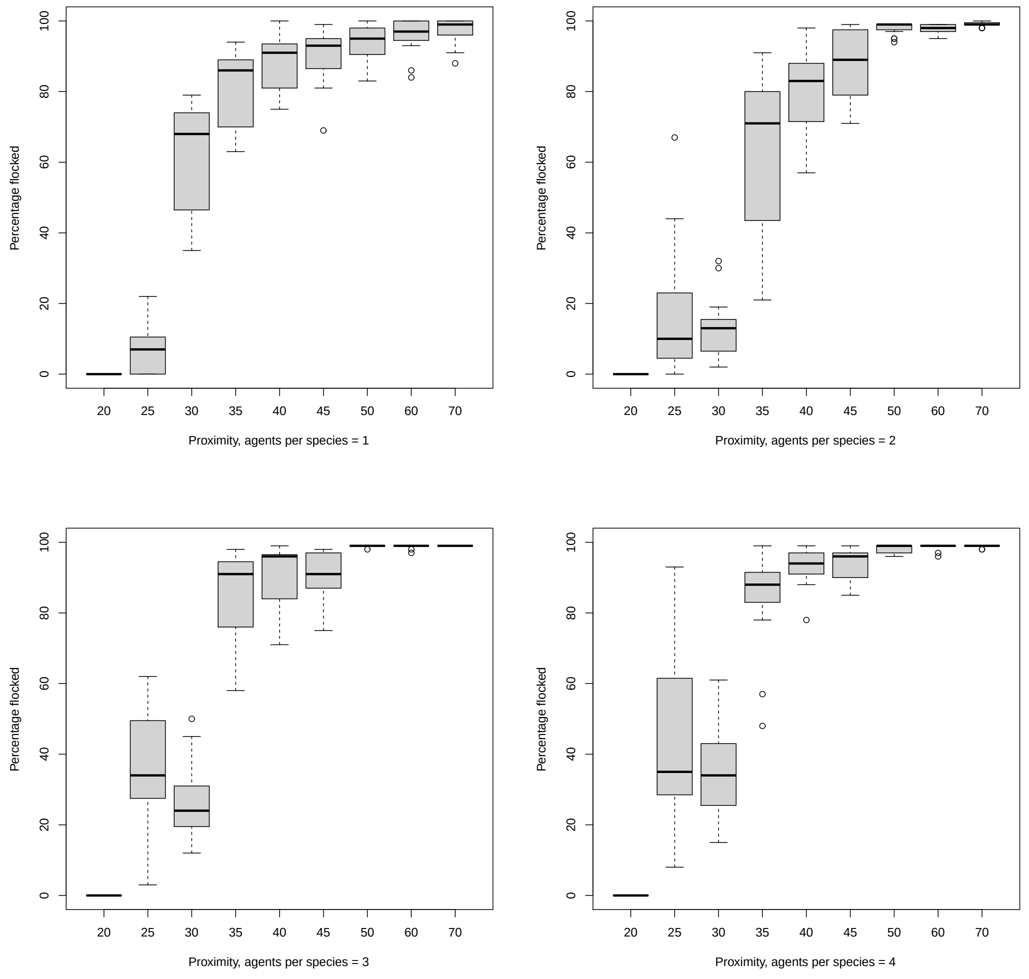


Figure 6.6: Box plots showing the results of flocking in multiple environments. The plot at top-left is for an agent in a single proximity environment, and essentially the only agent in its species environment. The top-right plot is the same with 2 agents in the species environment, 3 at bottom-left and 4 at bottom-right. Each box shows the percentage of agents that flocked over 2 million simulation steps with 20 simulations being run for each box. Each run started with 300 agents, randomly distributed in a 500-unit radius circle around the origin. Each horizontal axis shows the value of $proximity$ in the ProximityEnvironment.

is slightly more likely that they will move towards each other as the members of the species environments can see across the entire world. As such agents move towards each other, they are more likely to approach other agents in the proximity environment and so aggregate a collection of agents that the instrumentation deems to be a flock. But, if the agents of the same species are not so advantageously positioned then even though they try to move together they never manage to attract any other agents. Even if four agents of the same species end up next to each other the rules for being a flock, as listed in section 6.1.3, do not allow that to be described as a flock.

Notwithstanding this variation at low $proximity$ values the experimental hypotheses are upheld. Specifically, agents simulated by a multi-environment environment oriented simulation do continue to flock and, subject to some observations relating to the initial configuration, the multiple environments do increase the amount of flocking observed.

In support of the central notion of environment orientation these experiments in flocking in multiple environments include rather more complex (indirect) interaction between the agents. Even so, there is no possibility of deadlock as the only shared resource on which multiple threads could deadlock is the underlying repository and all contending access to that repository is localised to two specific methods of that class which are serialised by being marked using the Java `synchronized` attribute. So, if one thread blocks when accessing the repository it is inevitable that at some point it will become unblocked: the currently blocking thread must leave the `synchronized` region at some point as it is proceeding atomically.

Finally, as also mentioned in section 6.2.1 the modifications required to the platform to support multiple environments are straightforward.

However, the environments discussed in this experiment are essentially internal. That is, they are mostly a consequence of the (simulated) perceptual capabilities of the agents. Many complex systems include a significant external environment, for example the geography and commercial world that is the backdrop to the situations discussed in chapter 2. A simulation with a more complicated external environment is therefore investigated next.

6.3 Hill climbing

In the flocking experiments described so far, the environment is just mediating agent state and presenting it to requesting agents. That is, as discussed, it is to a large extent an artefact of the agents themselves. In the experiment described in this section the agents' neighbourhoods are extended by an environment that describes the landscape in which the agents move. This landscape provides information about the height of an agent, each of which is required to sit on the surface of the landscape. That is, every agent has an altitude which is the height of the landscape at the position where the agent finds itself.

In this experiment the simulation code is attempting to get agents to climb to the highest point in that environment. However, the agents cannot observe the landscape directly, only through the external state of neighbouring agents. In this environment each agent is trying to climb to the highest point that it can see by observing the altitudes of neighbouring agents.

Given the information available, agents in the hill-climbing simulation move towards the highest part of the landscape of which they are aware; this being limited to the altitudes of each of the agents in a querying agent's neighbourhood. Additional rules prevent agents being positioned directly on top of each other, in a manner analogous to the repulsion rule in the flocking simulations already discussed.

6.3.1 Experimental hypotheses

In this experiment I investigate the hypothesis that an environment oriented simulations can be readily extended to include external concepts, representing the physical world outside the agents. Here, as discussed, this concept is a simple "landscape".

6.3.2 Platform extension

The implementation of the hill climbing simulation is very similar to that for the flocking simulation. A UML diagram giving an overview of the entire platform for this context is shown in figure 6.7. It is specialised by using a different implementation of the Environment class, LandscapedEnvironment. This uses an implementation of another abstract class, Landscape, in order to find the altitude of the

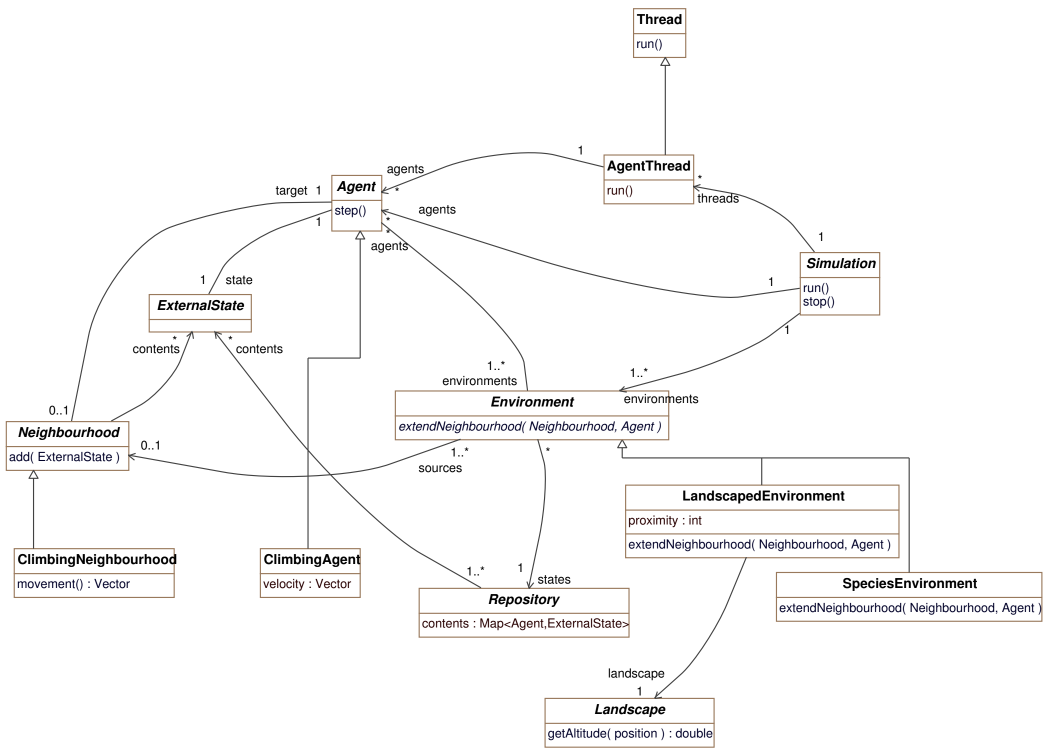


Figure 6.7: Overview of the platform configured for hill climbing in multiple environments. Each agent exists in a single proximity landscaped environment and a species environment.

particular bit of landscape at the agent's position. In a similar manner to the absolute position of an agent, its altitude is recorded by decorating the agent's external state with its altitude.

As would be expected, the agents (in this case instances of ClimbingAgent) implement a different process when they are run by their controlling thread. Pseudo code for the climbing agents' step implementation is shown in figure 6.8.

In this case the climbing neighbourhood is used which can determine which of a set of states represents the one with the greatest altitude and provides the agent with the movement needed to move towards that position.

Again, the implementation is enriched with a specialised monitoring class. An implementation of this class collects information as to how many agents have found themselves at an altitude about 90% of the greatest height in a particular landscape. Using this it is possible, again, to instrument the running simulation and hence to

```
Vector movement;
int targetAltitude = environment.getAltitude(this)
ClimbingNeighbourhood n = new ClimbingNeighbourhood(this, proximity, targetAltitude)
for Environment e in environments do
    e.extendNeighbourhood(n, this)
end for
movement = n.movement()
for Environment e in environments do
    e.updateState(this)
end for
```

Figure 6.8: Pseudo-code for the climbing agent's updateInternalState().

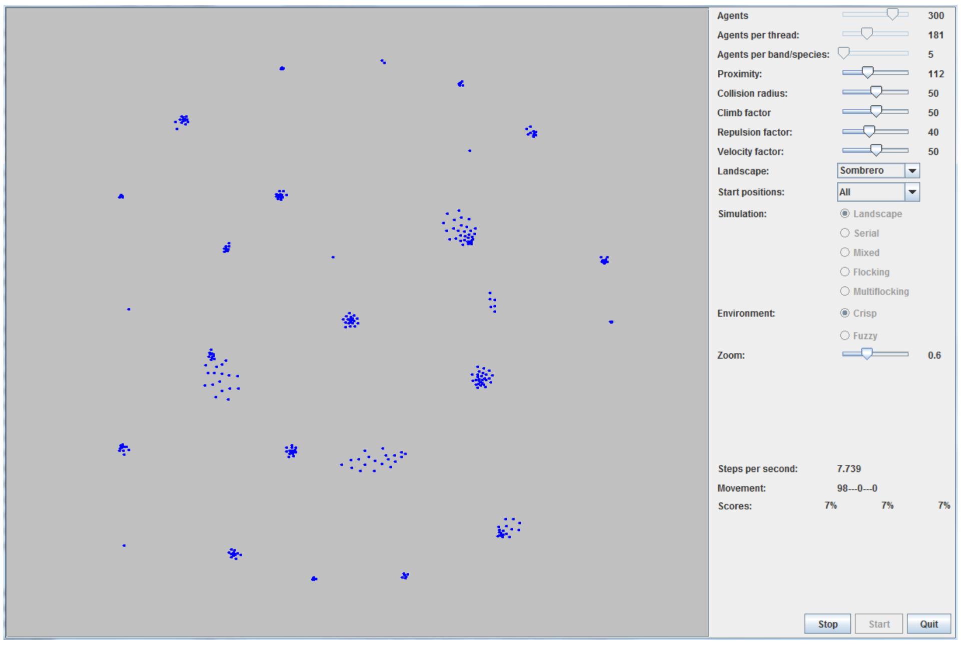
produce some graphical information.

An example of the user interface for the climbing variant of this simulation is shown in figure 6.9.

In this particular case the simulation is being done with a landscape that implements the classic "sombrero" function [Wikg] (figure 6.10). For simplicity I did not implement a visualisation of the landscape but some aspects of it can be inferred from the positions of the agents in the graphical display. In this particular case, as shown in the display metrics, 7% of the agents have reached a height greater than 90% of the maximum height which in this case is the height of the central peak. As can be seen, the agents are clustered on the central peak and along the ridges of the surrounding "hills".

6.3.3 Simulation results

In the same manner as in the earlier discussions of flocking the box plots shown as figure 6.11 summarise the results of a large number of hill climbing simulations. As can be seen the agents are relatively unsuccessful at finding the central peak in the situation where each agent is in its own species environment in addition to the proximity environment. (That is, where there is essentially only a single proximity environment.) However, even in this situation, the agents are more successful at finding the central peak as the value of $proximity$ increases. Of course, this is hardly surprising as with a larger value of $proximiity$ each agent is more likely to



Agents 300
Agents per thread: 181
Agents per band/species: 5
Proximity: 112
Collision radius: 50
Climb factor 50
Repulsion factor: 40
Velocity factor: 50
Landscape: Sombrero
Start positions: All
Simulation: [✓ Landscape] Serial Mixed Flocking Multiflocking
Environment: [✓ Crisp] Fuzzy
Zoom: 0.6

Steps per second: 7.739
Movement: 98---0---0
Scores: 7% 7% 7%

Figure 6.9: Graphical user interface for hill climbing simulation using the platform.

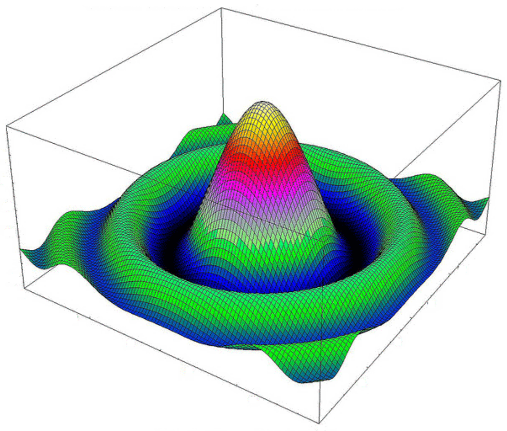


Figure 6.10: A sombrero function, a 3D analogue of the $sinc(x) = sin(x)/x$ function.

"see" another agent which is further up the landscape. All the same, even at the largest value there is not a great deal of success. The reason for this is that an agent tends to find a local maximum which actually turns out to be the top of one of the surrounding "foothills" of the landscape (as in figure 6.10). This effect can also be seen in the view of the user interface shown in figure 6.9 where clusters of agents can be seen around the central peak.

As the number of agents in the species environments increases then the agents become more successful at finding the central peak. The reason being that if two distant agents are "connected together" by being in the same species environment then, as they move towards each other if one or other of them finds the central peak, or the start of it, then it will tend to stay there and drag the other one (along with all the agents in its nearby neighbourhood) towards it. The apparently much larger success at $proximity = 150$ is a consequence firstly of the non-linear horizontal axis in the figures and the increased chance of success due to 150 being a large proportion of the distance between the central peak and the surrounding hills.

6.3.4 Hill-climbing discussion

The hypothesis that environment oriented simulations can be extended to include aspects of the physical world surrounding the agents is supported. In this case the LandscapedEnvironment is adding into the simulation information about the world in which the simulation is occurring, in this case the particular instance of Landscape that is being used. Even though the implementation is still using the basic notion of environment orientation, and the implementation of that embodied in the simulation platform, the simulation works effectively. In particular management of concurrency has become no more complex as each agent is still merely using its environments in a transactional manner.

6.4 Discussion

In this chapter I have discussed some simple simulations built using the environment orientation experimentation platform. The initial item of note is that they do, indeed, work with minimal attention paid to deadlock. That is, the behaviour of a complex

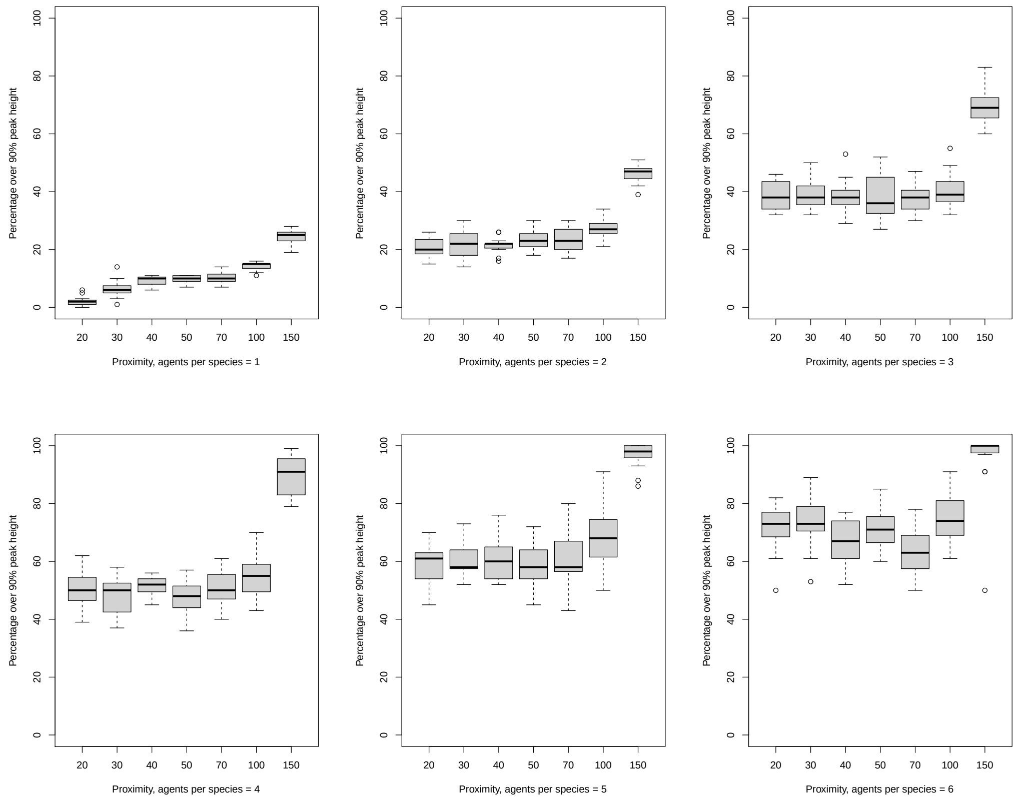


Figure 6.11: Box plots showing the results of hill climbing in multiple environments. The plot at top-left is for an agent in a single proximity environment, and essentially the only agent in its species environment. The other plots are for 2, 3, 4, 5 and 6 agents in the species environment. Each box shows the percentage of agents that had reached a height of $>= 90\%$ of the height of the central peak with 20 simulation runs for the data in each box. Each horizontal axis shows the value of $proximity$ in the ProximityEnvironment.

system agent can be expressed, and simulated, merely by discussing the relationship of that agent to its environment.

The environment, though, it not merely some passive container. Even in the examples already discussed, it is providing a number of discrete services:

State retention The environment is retaining the external states of agents for later supply to other agents.

State decoration The environment, in my implementation at least, is responsible for imposing an overall "environment-centric" view on individual agents' states. For example, the environment is calculating and retaining (by the use of the Decorator pattern) the actual positions of each agent even though the agent itself is operating in a solipsistic manner at the centre of its own universe.

Landscape information The environment provides other information to agents, not merely the states of other agents. This information represents aspects of the environment itself that are needed by the agents. In the hill climbing example this includes the height of a particular piece of the landscape. In this particular case the height, at a particular point, is constant. However, there is no reason to suppose that that would always be the case and I describe a more complex example of this in chapter 8.

However, the examples discussed so far take a particularly simplistic approach to fusing the information received from multiple environments. It is reasonable for complex systems agents to perceive other agents in a number of different environments, such as the visual and auditory environments discussed in section 6.2. In the examples presented so far these environments are combined in perhaps the simplest possible manner in that the external states for accessible agents in all environments are added *verbatim* to the querying agent's neighbourhood. That is, in this example the querying agent can perceive as much information about a physically distant agent in a species environment as it can about an adjacent agent in the proximity environment. This is not what I discussed earlier in section 4.3 and, more importantly, this is not representative of what would happen in the complex system itself.

In the complex system, the effects of distance would be, at least, to degrade the resolution at which distant agents' states could be observed. In a visual environment a nearby agent would be perceived precisely in three dimensions from the point of view of the querying agent. Furthermore, its motion in that space would also be perceptible. That is a flocking bird would be able to tell where another bird was and what its current velocity was; indeed this is necessary for the flocking algorithms mentioned here and taken originally from [Rey87]. However, if an agent were perceived in an auditory environment then the detecting agent would be able to determine roughly in which direction another agent lay but it is likely that its range and velocity would be less easy to determine. That is, the visual environment provides three dimensions of position information and three of velocity information but the auditory environment provides just a single dimension of direction.

In the next chapter I examine one approach to merging information from different environments when each such environment contributes such varying information.

Chapter 7

Synthesis of multiple environments

As I discuss in section 3.2.4 the notion of multiple environments in a complex systems simulation is obvious. In some simulations an agent might respond independently to the information derived from two different environments. That is, specific parts of the behaviour of the agent would be derived from observations in separate environments with no consequential requirement to merge observations from multiple environments.

Here, though, I am concerned with a situation where the behaviour of an agent is derived from observations in multiple environments. That is, there is a requirement to combine states from multiple environments in such a manner as to allow a querying agent to make a single decision as to its immediate behaviour.

As such, an environment oriented simulation must provide the capability for combining observations drawn from multiple environments where those environments provide information that is potentially embedded in a different space and provided at a different resolution. That is, the neighbourhood must be able to combine observations like "there is an agent in direction $45°$ at a range of 100 units" with others like "there is a nearish agent that it quite likely in front although it could be over to the right but it is unlikely to be behind". This combination of observations is properly the responsibility of the neighbourhood, which is a construction of the environment, rather than the agent itself as the intention of the environment oriented approach is to provide an agent with a view of all of the surrounding environments with which the agent is able to interact in a simple manner. Further, making this combination the responsibility of environment enforces the role of the agent as a relatively simplistic one. The ability to build in "magic sensors" which could do processing properly beyond the scope of the complex system's agents as the agents do not have access to the extensive information required.

Both of these observations are potentially relevant to the querying agent but must be combined somehow to generate the information needed for the next action of the agent. That action, of course, must be specific; the agent must be able to make a decision as to how precisely to modify its behaviour. Such combination cannot be made in the manner I have outlined previously where I use simple vector addition.

However, this approach does not naturally extend to contexts where the behaviour of an agent is derived from observations in different environments, ones that represent different vector fields. Summing vectors from these different fields would not necessarily make sense, in the same manner that summing velocity and temperature would not yield a meaningful value. In this chapter I discuss this issue and show experimental support for the hypothesis that an environment oriented simulation can be produced where an agent observes information in multiple environments and where that simulation can be produced in a manner that is consistent with the simulation architecture already developed.

7.1 Control systems

The concept of determining behaviour in the context of observations in multiple fields is not one unique to complex systems. It is commonplace in control systems, something I studied as an undergraduate [Rav68]. In general, a control system must make multiple observations, frequently including the system output itself, and apply some control to the system so as to constrain the system output within acceptable limits. That is, usually such systems are *Feedback Control Systems* [PH95]. Such engineered systems are everywhere: in aircraft, automobiles and all manner of industrial and domestic machinery.

One particular approach to the construction of control systems is particularly appropriate to the issue here. This the is use of "fuzzy control" [Cos92]. This approach is specifically intended for situations where observations, frequently ones that are not completely specific, are made of multiple inputs and combined into an overall control strategy. As such, this approach adapts to the complex systems

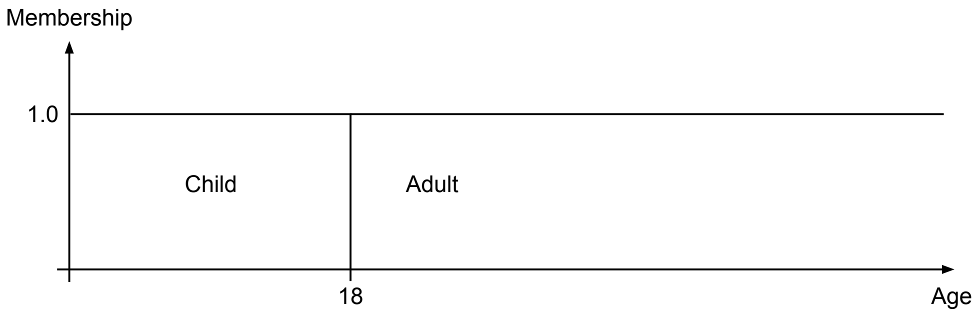


Figure 7.1: Simple summary of a crisp separation as appears in the law. In this case all people are deemed to be a member of the set “adult” when they reach the age of 18.

simulation under discussion.

7.1.1 Fuzzy logic

Fuzzy control is based on the use of fuzzy logic [YRP94]. Fuzzy logic is based on the notion that observations, rather than being a member of a “crisp” set based on absolute rules, may belong partially in one or more “fuzzy” sets.

One example often cited in fuzzy logic texts is that of determining, based on a person’s age, whether they are an adult or a child. Such a question has a specific answer in the context of the law, for example with relation to criminal responsibility or being able to purchase and consume alcoholic drinks. Such a “crisp” definition for the concept of “adulthood” is shown in figure 7.1; according to this separation a person is deemed to be adult when they reach the age of 18, regardless of any other consideration. In fact, transition from being a child to being an adult happens at a specific point in time, at midnight on their birthday. That is, from a legal point of view, the observation of a person’s age is a “crisp” observation: it either says “child” or “adult”, admitting no concept of indeterminacy.

But, this specific separation is something that does not match our perceptions of different people’s growing maturity. As such, fuzzy logic is based on the notion that in the real world, and the argument used is often strongly based on real world observations, everything is to some extent a “matter of degree” [Kos94]. That is, the

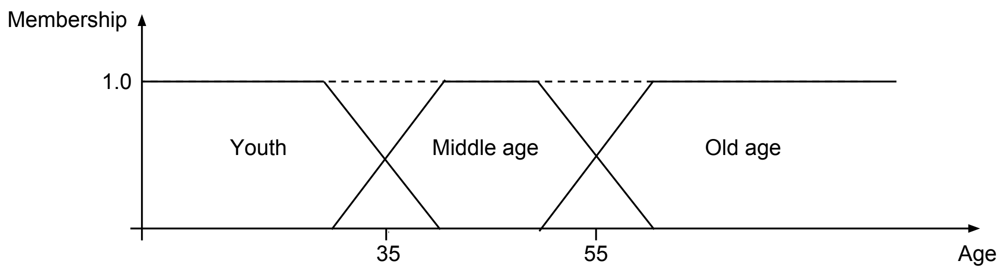


Figure 7.2: Simple summary of a fuzzy concept relating to whether someone is perceived to be young, middle-aged or old.

notion of "adulthood" is not as crisp as the law declares. A fuzzy separation of age into three separate fuzzy sets is shown in figure 7.2 which shows the membership of three fuzzy sets corresponding to youth, middle age and old age.

In this diagram people are deemed to be youthful, middle-aged or old, each of which is a fuzzy set. The separation between these sets is not specific, according to this diagram a 55 year old can be equally perceived as being middle-aged or old, but would never be taken for young. A 55 year old, is somehow half a member of the set "middle aged" and half a member of the set "old aged". Similarly, a 56 year old would perhaps be 0.4 a member of the first set and 0.6 of a member of the second.

This approach is relevant to the notion of combining observations in multiple environments because it allows the combination of exact membership of a set (someone of age 5 is definitely young) with less precise membership (someone of age 35 can be regarded as "half young"). That is, the membership of a collection of observations of these fuzzy sets corresponds to a combination of some things that are known precisely with some things known with less certainty. This is analogous to the notions that arise from multiple environments, as discussed in section 6.4.

There is some dispute about fuzzy logic, with many authors remarking that the fuzziness can be simulated by crisp systems. More so, there is almost a religious aspect to some objections. Quoted in [Kos94, p3] and attributed to Professor William Kahan of the University of California, Berkeley is:

> Fuzzy theory is wrong, wrong and pernicious. What we need is more logical thinking, not less. The danger of fuzzy logic is that it will encourage the sort of imprecise thinking that has brought us so much trou-

ble. Fuzzy logic is the cocaine of science.

Notwithstanding that, fuzzy logic seems to me to be useful, merely because it specifically allows an indeterminacy of observation, something that matches the notions discussed here of observations within environments, especially when those observations are not, and could not be, precise.

Fuzzy logic itself, though, is is not imprecise, the values for the membership of the fuzzy sets is precise, it is just not zero or one. The notion of someone being "might be old" simultaneously with "probably middle aged" fits our real world perceptions where we don't use such specific and sharp boundaries.

In support of the use of fuzzy logic, it is in fact frequently found in control systems; industrial fuzzy logic control systems are commonplace and available as packaged products. Even household items such as washing machines are marketed as using fuzzy logic control.

Such a control system must take various control inputs and assess them for memberships of fuzzy sets (in a similar manner to a 55 year old being half a member of the set "middle aged" and half a member of the set "old aged") representing the control inputs and produce the appropriate values for the control outputs.

For example, if such system were indeed controlling a domestic washing machine[1] then there might be sensors detecting the level of dirt in the washing water, and the size of load in the machine. These sensors will deliver precise observations—perhaps 5,000 ppm of dirt in the water and a load of 3.4kg—but from these observations will be calculated the membership of various fuzzy sets such as "very dirty", "slightly dirty", "full load" and "medium load".

That is, there is a precise value for "washing machine load", in this case 3.4kg. However, the control calculation is expressed fuzzily, perhaps in this case the load can be regarded as 0.4 "half load" and 0.6 "full load". The control calculation is done this way because that rapidly yields a suitable result and also because the control logic can typically be expressed as something like IF-THEN rules. For example, we might have:

IF the water is very dirty and there is a full load
THEN set the agitation level to high

[1]This example is adapted from that in [Kos94, p39 pp180–182].

In this case the control system is applying a control input—changing the agitation level—as a result of multiple, fuzzy, observations of a range of parameters. The benefit of using fuzzy logic is that some other control rule might refer to the load as being medium. That rule could simultaneously apply as the observation of the load is not a crisp one, but a fuzzy one.

A further example, from [YRP94] in the context of controlling a car's braking system, is:

> IF the distance between two cars is short and the speed of the car is high
> THEN brake hard for speed reduction

Again, in this case, there is a precise value for the distance between the two cars, perhaps it's 8.6m. The control system, though, is only told that distance as a membership of some fuzzy sets such as "short", "medium" and "distant". The control system is applying a control input, in this case braking, based on multiple observations in multiple parameters. As before the fact that the underlying observations are fuzzy means that the control logic can be succinctly expressed as multiple rules.

Applying this notion in the context of observations in multiple environment is appealing because, again, there are precise values for things like agent position and velocity. However, an observing agent likely is not able to perceive variables with that precision and, it needs to be able to express its behaviour, in a manner analogous to the control logic expressed, in a simple manner.

7.1.2 Fuzzy flocking

In the case of something like simulated bird flocking the observing agent can be seen as another control system, taking in control inputs representing observations of other birds and generating an acceleration vector to add to its current velocity.

In order to test the notion of combining environments using fuzzy logic I modified the implementation of the bird-flocking experiment discussed in chapter 6. In this new implementation each agent is described by two variables[2]: *distance* and *direction*. Each of these is assessed for membership of one or more fuzzy sets, the process known as "fuzzification" in the literature.

[2]Often called "linguistic variables" in fuzzy logic texts.

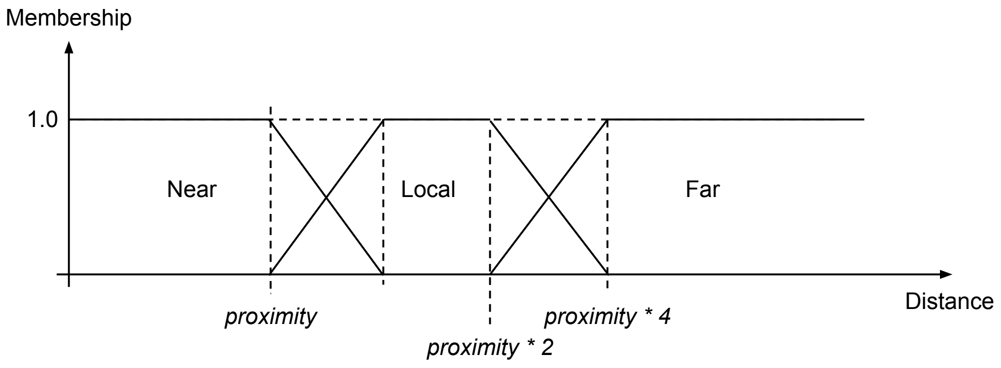


Figure 7.3: The distance to an agent is “fuzzified” into membership of three fuzzy sets: *near*, *local* and *far*.

In the case of distance the simulation allows for fuzzy sets *near*, *local* and *distant* as is summarised in figure 7.3. Agents that are nearer to a querying agent than the proximity value are definitely near, if nearer than $proximity * 2$ they are definitely local and if further away than $proximity * 4$ then they are definitely distant. At other distances there is a linear relationship defining the membership of the fuzzy sets as is shown diagrammatically in figure 7.3.

It is, though, a requirement for the issue of combining environments in the case of simulated bird flocking that in some cases the distance to the remote agent is just unknown. For example, in the case of the species environment the conceit is that these agents communicate by sound in this environment and that therefore very little is known of distance. In the case of these agents, then they are ascribed an equal membership of each of the fuzzy sets. That is, the likelihood of an unknown agent being “near”, or “local” or “far” is always $1/3$.

A similar approach is taken for the observations of direction. In this case, observations of direction to remote agents are described in terms of the likelihood of an agent being a member of the fuzzy sets *north*, *south*, *east* and *west* in the manner described in general in figure 7.4.

7.1.3 Fuzzy flocking implementation

In order to support the notion of fuzzification of observations in the experimental platform it was modified as shown in the UML diagram in figure 7.5.

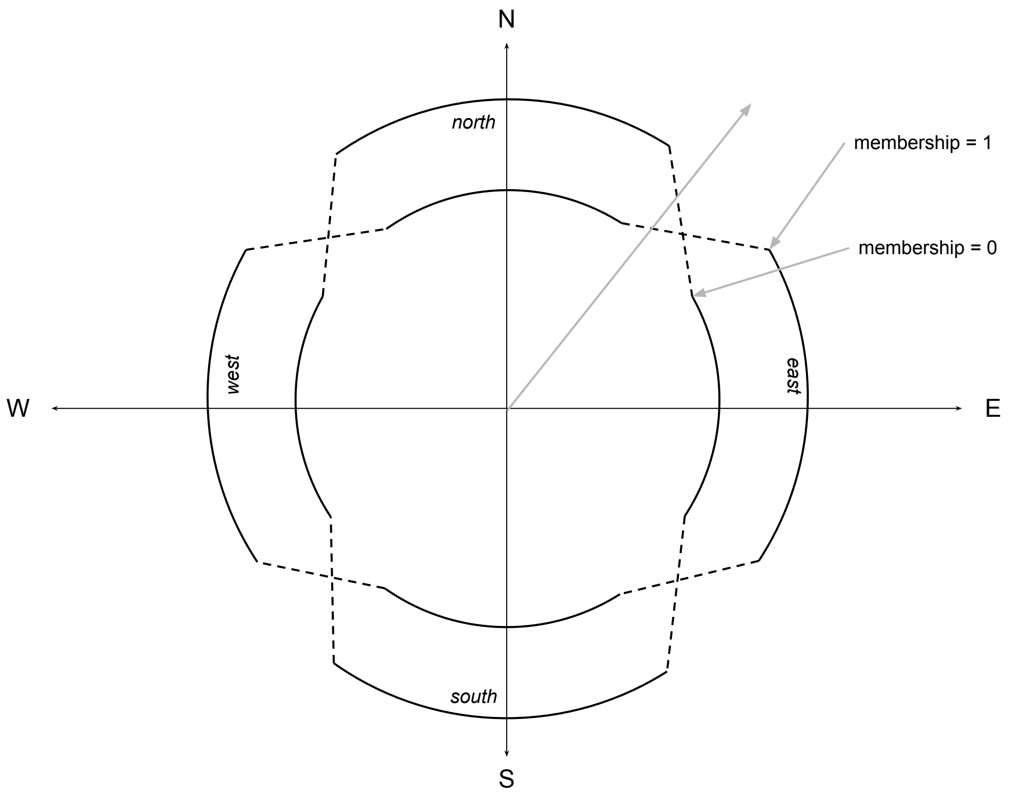


Figure 7.4: Diagrammatic representation of the fuzzification of directions into membership of the fuzzy sets *north*, *south*, *east* and *west*. The direction shown by the grey line at approximately 35°, for example, has membership of *north* at about 0.7 and of *east* at about 0.3.

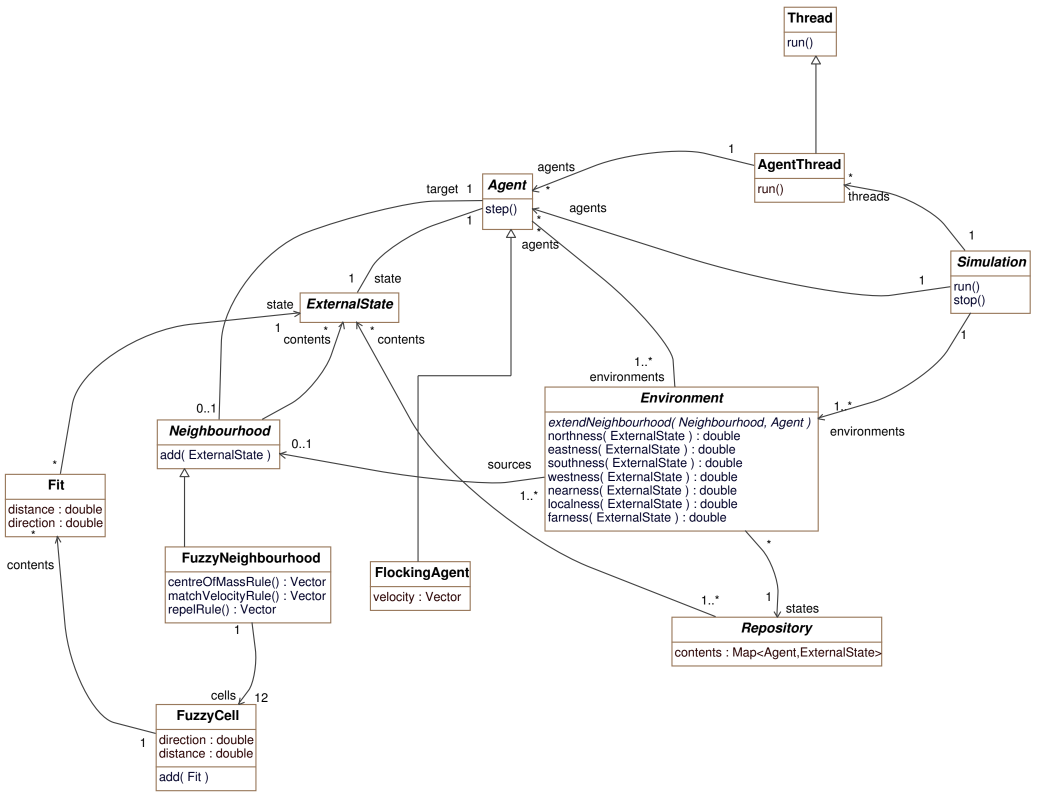

Thread
run()
AgentThread
run()
Agent
step()
Simulation
run()
stop()
ExternalState
Environment
extendNeighbourhood(Neighbourhood, Agent)
northness(ExternalState) : double
eastness(ExternalState) : double
southness(ExternalState) : double
westness(ExternalState) : double
nearness(ExternalState) : double
localness(ExternalState) : double
farness(ExternalState) : double
Neighbourhood
add(ExternalState)
Fit
distance : double
direction : double
FuzzyNeighbourhood
centreOfMassRule() : Vector
matchVelocityRule() : Vector
repelRule() : Vector
FlockingAgent
velocity : Vector
Repository
contents : Map<Agent,ExternalState>
FuzzyCell
direction : double
distance : double
add(Fit)
agents
threads
target
state
contents
environments
sources
states
cells

Figure 7.5: Overview of implementation of fuzzy neighbourhoods and support from the environments.

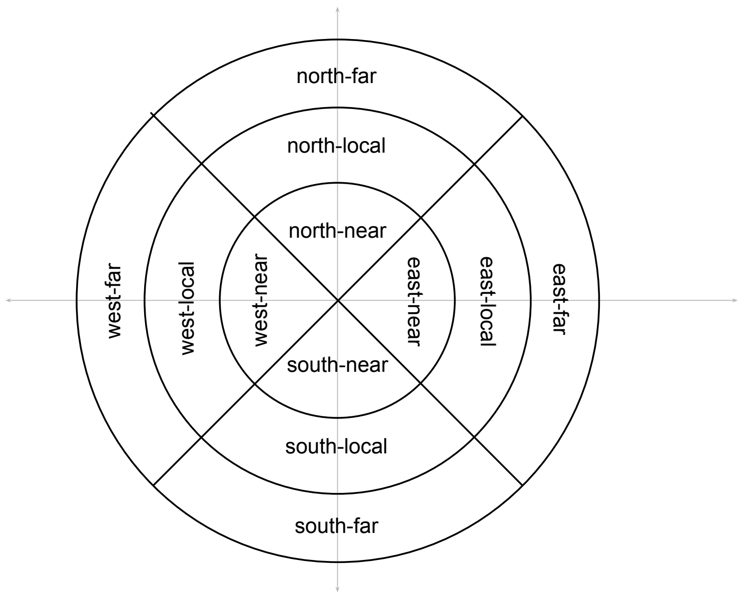


Figure 7.6: Each fuzzy neighbourhood has 12 instances of the FuzzyCell class, shown diagrammatically here with the pair of fuzzy sets each represents.

As before this adds new facilities to provide support for the fuzzy behaviour. The new neighbourhood class FuzzyNeighbourhood functions as the earlier FlockingNeighbourhood but the membership of the neighbourhood is described using the classes Fit and FuzzyCell. The additional complexity here is due to the fact that a single observation can appear multiple times in the neighbourhood.

In order to allow for the various fuzzy sets, each fuzzy neighbourhood has 12 instances of the FuzzyCell class, conceptually structured as shown in figure 7.6, and representing those states that have a non-zero membership of a particular pair of fuzzy sets.

For example, an observation of an agent in a species environment could generate, if it represented an agent at about 45°, an item in each of the cells representing the pairs *north-near*, *north-local*, *north-far*, *east-near*, *east-local* and *east-far*. In another case, for example a nearby agent in a proximity environment which was definitely south of the querier would only be represented as an entry in the *south-near* cell.

These instances of FuzzyCell are analogous to the possible configurations of the underlying IF-THEN rules in a control system's control logic. That is, the right-most segment in figure 7.6 corresponds to a rule like:

IF the observed agent is to the east and is far away
THEN move carefully towards the east

Each entry in a FuzzyCell object is provided by an instance of the Fit class which contains the actual value of the membership of the sets in question.

In order to construct this representation it is necessary to calculate the values for the fuzzy set membership, which will appear as attributes of the Fit objects. Some object must therefore have the responsibility of saying that a particular external state object represents an agent whose membership of the sets near, north and east is 1.0, 0.5 and 0.5 respectively. This object must be the environment. As before it is the environment that knows where things actually are and presents to the agents, via their neighbourhoods, a summary of the world. Hence, the platform implementation of the Environment class implements operations to define how "north", and so on, a particular external state is, these operations are included in figure 7.5.

An example of how this is implemented is shown in the UML sequence diagram in figure 7.7. The environment is asked to extend a supplied neighbourhood and it duly adds the appropriate states to that object. As the neighbourhood is fuzzy it calls back to the environment requesting values for the set memberships and for the non-zero ones it constructs the appropriate Fit object and adds that to the appropriate FuzzyCell. These call-backs are to the operations like northness and nearness. That is the methods implementing these operations are responsible for fuzzifying a crisp observation into membership of the fuzzy concepts in use, in this case the "north" and "near" fuzzy sets.

The FuzzyNeighbourhood class provides fuzzy method implementations of the normal rules for bird flocking. Rather than using IF-THEN rules the structured space provided by the various FuzzyCell objects means that the calculation of, for example, the vector to fly to perform the centre of mass rule is done by vector-summing a number of unit vectors, one for each FuzzyCell, in proportion to the weight of the Fit objects within that cell. In this way all of the Fits, one for every possible membership of the fuzzy sets representing the control inputs, contribute towards the eventual result. The resulting vector is then treated in exactly the same

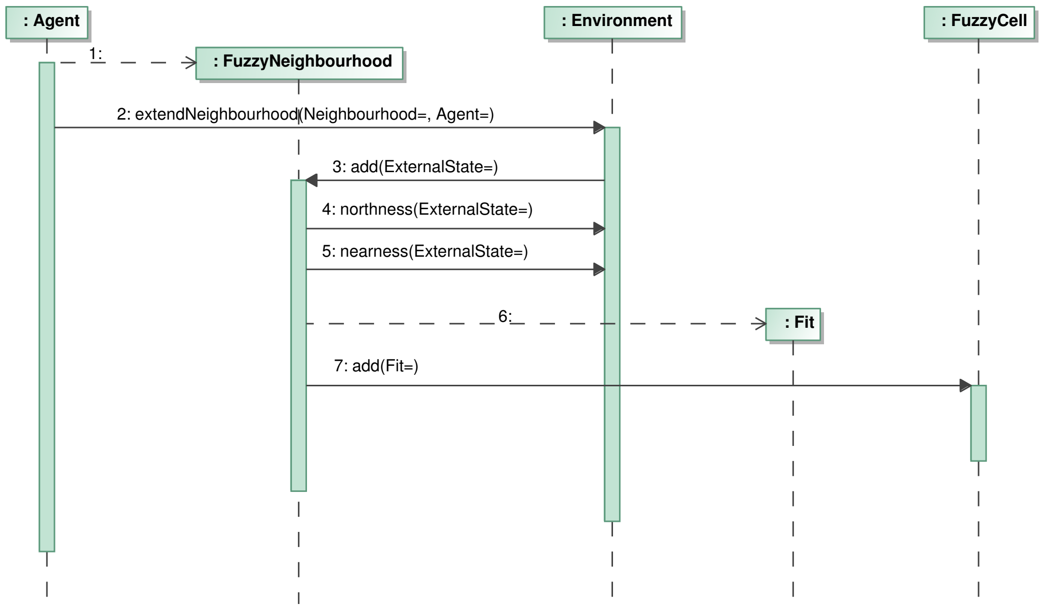

: Agent
: Environment
: FuzzyCell
1:
: FuzzyNeighbourhood
2: extendNeighbourhood(Neighbourhood=, Agent=)
3: add(ExternalState=)
4: northness(ExternalState=)
5: nearness(ExternalState=)
6:
: Fit
7: add(Fit=)

Figure 7.7: A simple example of constructing a FuzzyNeighbourhood.

manner as applied in the crisp multi-environment flocking described in section 6.2.

7.2 Experimenting with fuzzy flocking

The structure already presented in this chapter shows how fuzzy sets can be used for combining observations in multiple environments and also shows how these ideas were implemented in the experimental platform. This whole structure rests on the hypothesis that flocking will still be achieved in multiple environments even when the information available about the contents of the species environment is rather less informative than in the experiments described in section 6.2. That is, the contents of the species environment are now only contributing imprecise information about direction and nothing about distance. Furthermore, the chunking of the world into just 12 possible locations, albeit with fuzzy membership providing a bit more information, would be expected to strongly affect the efficacy of the overall flocking.

Nonetheless, I hypothesise that flocking will continue to appear in this structure. Furthermore, the addition of contributions from the species environments should improve flocking, for the reasons already discussed, although the contribution from the species environments should appear in some way different, as rather less information is now being provided by them.

7.3 Results

A similar series of experiments was conducted using the fuzzy flocking platform. Box plots summarising these results are shown in figure 7.8 which are presented in the same manner as those in figure 6.6.

The immediate observation from these results is that, indeed, the birds are continuing to flock and the number of agents in the species environment is having a marked effect on the results. There are many subtleties to notice though. One is that with only a single agent in the species environment, which is analogous to only using a single proximity environment then although there is some flocking going on, especially with larger values for $proximity$, then it is not as successful as in the crisp flocking. This was borne out by watching the display while conducting some of these experiments where the flocked birds were falling out of contact with each other after a while. I believe that this is because the fuzzy approach taken is actually rather coarse. In particular there are only 4 possible directions. On some occasions these directions could be seen on the screen as the agents forming into square patterns. I hypothesize, but have not been able to confirm, that making the fuzzy sets a little less coarse (for example, *north*, *north-east*, *east*, *south-east*, etc.) would have a beneficial effect.

With more than one agent in the species environments a dramatic change appears; even two agents is much more successful than one with smaller improvements for larger numbers of agents. Again, I suspect that this is due to the effects of distant agents, even when working through the fuzzy process, causing agents to be dragged into closer proximity until the proximity environments have an effect.

Nonetheless, the agents are flocking even in the face of adversity. It is clear that the fuzzy process does provide a mechanism for fusing observations in multiple environments. However, the code for performing the fuzzy flocking is rather more complicated than for the crisp flocking as has been seen in this chapter. In particular, the requirement for fuzzification of crisp observations creates significant complication and, indeed, processing time.

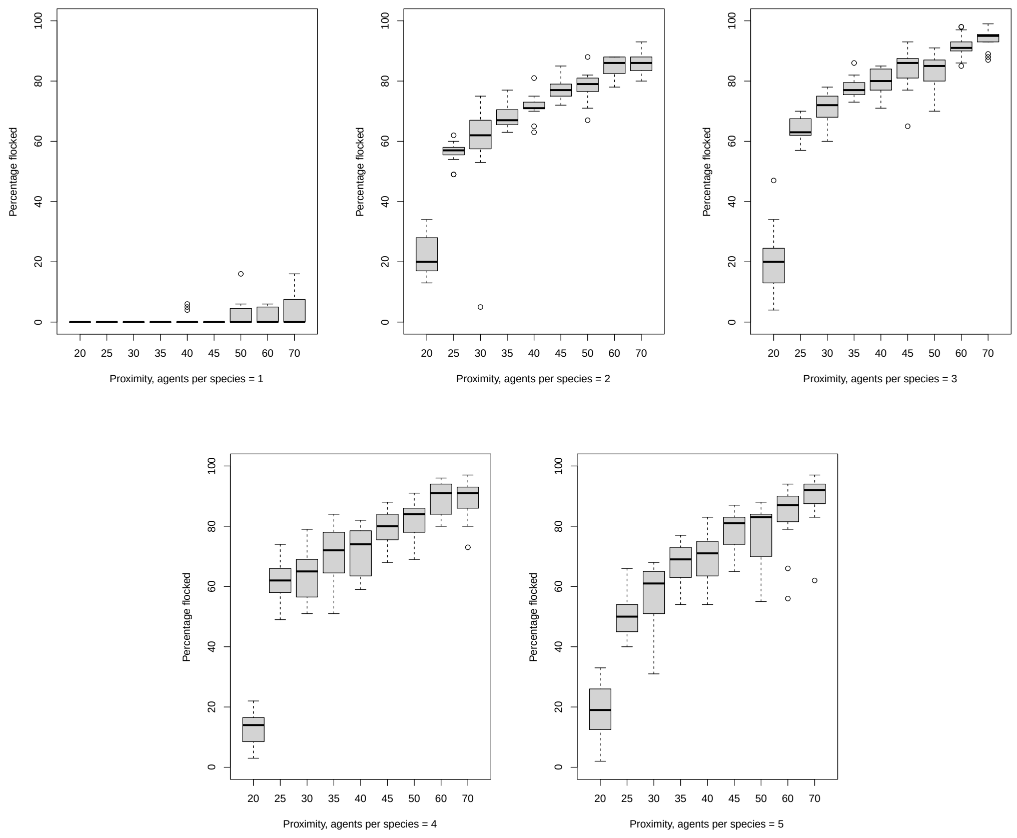


Figure 7.8: Box plots showing the results of flocking in multiple environments when using fuzzy matching. The plot at top-left is for an agent in a single proximity environment, and essentially the only agent in its species environment. The other plots are for 2, 3, 4 and 5 agents in the species environment. Each box shows the percentage of agents that flocked after 2 million simulation steps with 20 simulations being run at for each box. Each run started with 300 agents, randomly distributed in a 500-unit radius circle around the origin. Each horizontal axis shows the value of $proximity$ in the ProximityEnvironment.

7.4 Discussion

As has been discussed, the fuzzy process described has provided a mechanism for fusing together the combinations of multiple environments, although at the cost of a more complex implementation. Nonetheless, the implementation is still completely consistent with the original architecture; agents still construct their neighbourhoods and use a simple output from that neighbourhood to control their behaviour.

That is, the hypothesis that the basic simulation architecture is extensible to one where an agent observes information in multiple environments and where that simulation can be produced in a manner that is consistent with the simulation architecture already developed is supported.

Furthermore, the basic flocking emergent property persists in this situation and, still, no deadlocks did or could appear even though the only attention such issue remains the use of the `synchronized` keyword on a couple of methods in the Repository class. If the environment oriented approach had not been used and a similar multi-threaded implementation constructed then it would doubtless have suffered from resource contention leading to deadlocks due to the inevitable cycles of interaction amongst the agents. The alternative would have been to use some programming artefact like barrier synchronisation which has no clear analogue in the complex system itself.

In this fuzzy implementation the environment has again been tasked with more responsibility. It is now responsible for the fuzzification process; determining what the value is for an observation's membership of a fuzzy set.

Chapter 8

Evolution within an environment

I have described the concept of environment orientation in chapter 3 and investigated the hypotheses that such an approach does actually demonstrate emergent behaviour (chapter 6) and that it is extensible to environments comprising multiple vector fields (chapter 7). In this chapter I apply the concept of environment orientation to a more complex simulation. I take an existing simulation and extend it by the introduction of a non-trivial external environment. I use this simulation to investigate evolution within this environment. This is the sort of investigation that is generally required for the study of complex systems, thus demonstrating the applicability of the approach to scientific study of complex systems.

The motivation for this chapter is thus two-fold: firstly to apply environment-orientation within a more complicated application and, secondly, to use that application to investigate the effects of the environment (specifically the environmental energy flux) on an evolutionary model. This latter investigation is targeted at experimentally testing specific hypotheses about the application.

Specifically, I discuss simulation of a complex system that represents open-ended evolution and in particular evolution within systems that are open to a simulated energy flux, open to changes in the simulated environment, and open to the representation of evolutionary mechanisms. I focus on the energy flux that pervades the natural world, mostly deriving from the Sun, as a key component of the environment in these situations.

Representing the energy flux permits a simulation of many aspects of real world systems, such as the availability of food supplies. Further, it allows the investigation

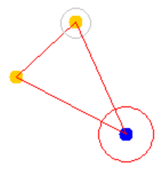


Figure 8.1: A simple example sticky-feet creature with three feet, the dots, connected by three springs, the straight lines. One of the feet is the heart, surrounded with a red circle at the bottom right, and another the mouth, in the grey circle at the top centre.

of different means of making a living within an environment, be they predatory or sessile.

That is, in this chapter the role of the environment has been further enhanced, as well as it being the location, in a simulation, of the agents' external states it now represents a more fundamental part of the domain that is being simulated; it allows the simulated agents to continue to exist.

8.1 Sticky feet

In order to investigate these issues I chose to extend Turk's Sticky Feet [Tur10] model. This gives a simple mechanism for implementing mobility and experimenting with open-ended evolution.

A Sticky Feet simulation is a collection of simulated creatures moving in a 2D domain. Each such creature is a graph of *springs* connecting together *feet*. Motion is achieved as a consequence of simple harmonic oscillation of the springs, which pushes the feet around within the simulation space. The coefficient of friction experienced by the feet is modulated—at times slippy, at times sticky—which results overall in motion through the space if the modulation is properly controlled. A typical simple creature is shown in figure 8.1.

Each creature has a heart and a mouth, each of which is a distinguished type of foot. The heart represents the creature's "essence" and is the component that is targeted by others wishing to consume creatures. Such consumption occurs when one creature's mouth is coincident with the heart of another whether that be accidental or because the prey was actively targeted. This causes the consumed creature to be removed from the simulation. The likelihood of a creature happening upon another is facilitated to some extent by the springs being equipped with sensors, which may modulate the oscillation of the spring when in the presence of another creature's heart. This allows a creature to turn towards another, with the chance that it might then be able to consume the target. In Turk's original implementation when a creature is consumed the eater produces a single offspring, which may be a mutation of the parent. Mutations that include additional feet, springs and sensors allow the creatures to evolve in a manner that eventually produces offspring that are better adapted to hunting for and eating other creatures.

8.1.1 Evolution in Sticky Feet

A Sticky Feet world is one in which creatures evolve to improve their performance at consuming other creatures, and therefore being able to pass on their genome. As such, it provides some aspect of a model of open ended evolution. I use this term here in the sense of an evolutionary system where components continue to evolve new forms continuously, rather than halting when some "optimal" or stable position is reached [Tay99].

Sticky Feet [Tur10] works in this manner, as there is no explicit fitness function and all creature behaviour is expressed in a single large environment rather than relying on artificial two-creature tournaments. As such it is representative of many aspects of real-world evolution.

There is, though, no mechanism for sticky feet creatures to pass on their genomes other than by consuming other creatures. The model of reproduction used in Turk's Sticky Feet is particularly simple in that a creature generates a single progeny whenever it consumes another creature. That is, the simulation is closed to the development of non-predatory behaviour because the only reproduction that happens is directly a consequence of predation. This is useful from the point of view of maintaining a constant sized simulation, but is not representative of real world evolution where population sizes can change dramatically.

8.2 The Sticky Feet Domain

Creatures in natural environments must be able to extract some sort of living from that environment, supported either by consuming other creatures, or by turning some flux in the world, for example sunlight or the chemical nutrients consumed by extremophiles, into food.

This argument is essentially that famously made by Malthus in 1798 [MMG08], which led Darwin towards the principle of natural selection [Dar59]. Although Malthus discussed the availability of food I have generalised this to the availability of *energy*. Although most real world environments are continually bathed in an energy flux it is a limited resource. The flux may be used, and stored, by components of the environment, but if it is ignored it disappears and is no longer of use.

Natural systems are open: they are in receipt of many sorts of resource flux such as sunlight. I am modeling all of these as "energy". There are many complex systems that can be simulated in such a world and as such some notions of the set of possible worlds that can be modeled is useful. In order to describe the set of possible simulations in the context of an environment bathed in an energy flux I have chosen to construct a meta-model, implicitly describing many possible simulations of this same general form.

As a part of the CoSMoS project I use the CoSMoS approach [APS$^+$10b] when approaching simulations. This approach is essentially to recognise that a simulation is inevitably a simulation of a particular aspect of some domain. As such, it is necessary to specifically describe—to model—the aspects of the domain that are being simulated, this model being described as a *domain model*. The actual simulation is described using the *platform model*, which executes on the *simulation platform*, producing results that can be analysed with respect to the *results model* [ASH$^+$11].

In the situation where the simulation is being used as a means of exploring the real world this is extended by means of the results so obtained being compared with the domain model allowing comparisons to be made between the model of the domain and the simulation of that model.

Crucially, we must be careful to describe our simulations as simulation of a model of a domain, and not of the real world domain itself with all its subtleties and foibles; inevitably in the construction of the domain model many aspects of

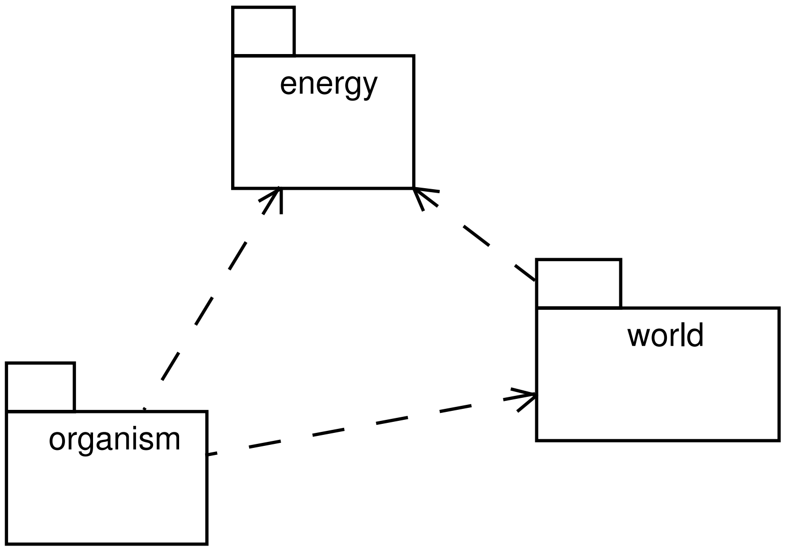


Figure 8.2: Domain metamodel. There are three components, represented as UML packages, with their inter-dependencies.

the spectacularly complicated real world will have been declared to be "outside the domain".

In this chapter I am essentially describing a *class* of models, ones that permit a particular sort of open ended evolution of sticky feet like creatures in a world, a domain, which is bathed in an energy flux. That is, I must define a *domain metamodel* to which domain models must conform. The domain metamodel, describes all possible domain models that are to be explored without limiting the particular domain. The approach I have taken to the domain metamodel is shown in figure 8.2 and shows the dependencies of the three main three parts, as UML packages, in the model. These three components relate to metamodels constraining three aspects of the domain metamodel:

energy which describes the rules that apply to energy models,

world which describes the general structure of the world within which evolution is to occur and

organism which describes the abstract structure of evolving organisms; at least as

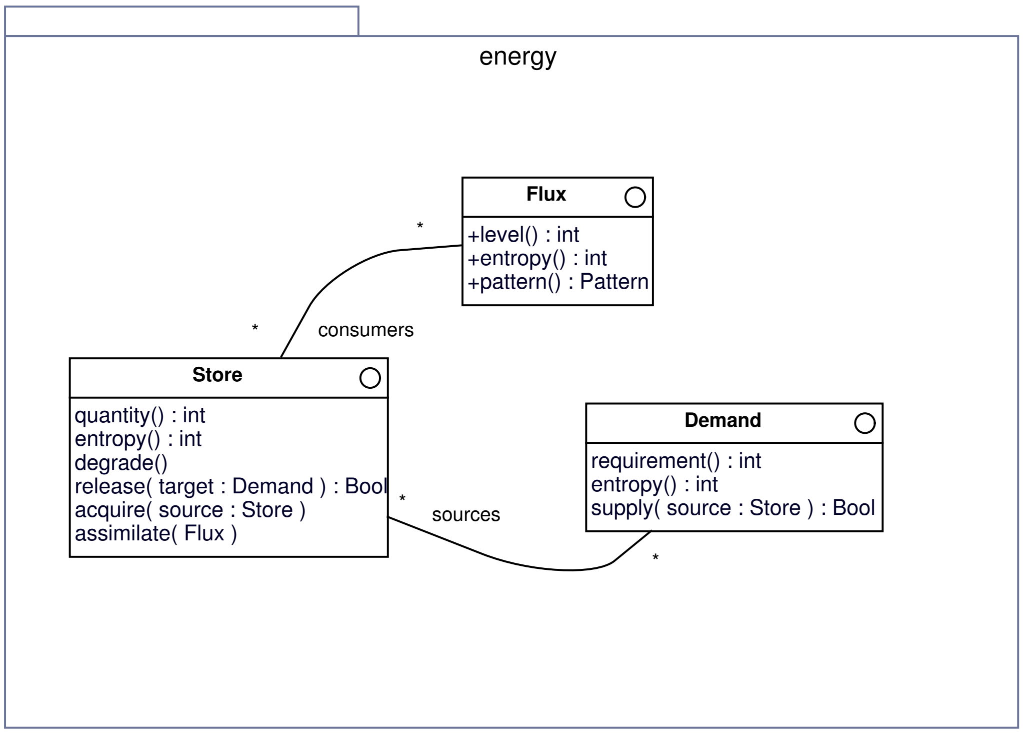


Figure 8.3: Energy metamodel.

far as this metamodel goes, there are clearly many, many, more details to be defined for a biological organism.

I address each of these aspects of the metamodel in the following sections. I describe each part of the metamodel as a *conceptual model* [FS03, CD94], without reference to a subsequent implementation.

8.2.1 Energy metamodel

The energy metamodel is elaborated in figure 8.3.

Energy is modeled as a scalar quantity in arbitrary units. However, as well as the *amount* of energy it is necessary to describe the *entropy* of the energy. This might be thought of as the temperature of the energy which allows the description of essential aspects of the energy economy.

For example, in the natural world a continuous low flux of low entropy energy is available in the form of sunlight. Plants sequester this energy in a form that allows other organisms, such as animals to consume them and acquire the stored energy. Those animals subsequently excrete waste products which still represent energy, albeit in a higher entropy form, but which may still be metabolized by organisms such as dung beetles. The concept of entropy described here is a part of the energy metamodel. However it is not used in the simulations that will be described later in this chapter. It is, though, an important concept to retain in the metamodel even if subsequent models do not instantiate it in order to use the concept in any future work.

The description of the energy metamodel is done using three separate concepts:

Flux The most basic part of the energy model is that which represents a flow of energy from outside the modelled system. This flux represents energy with a defined entropy and with a particular temporal pattern; for example at a high level during daytime but a much lower level during night time.

Store One action of all members of a simulated world is to store energy. An organism might maintain its existence by consuming other stores, in the manner of herbivores eating plants, or by assimilating the flux itself as the plant itself does.

Demand Many components of a simulated world make energy demands. Such components could be the physical structure of an organism, which requires energy to build and maintain, or an activity that an organism undertakes, such as hunting for other organisms to consume.

8.2.2 World metamodel

The world metamodel is elaborated in figure 8.4. As can be seen the concepts in the world meta-model are elaborated in the context of the energy meta-model already described. In essence this is emphasising the pre-eminence of energy in all of the domains that are described by this meta-model.

A world represented as a collection of *Regions*, each of which is the recipient of a particular *Flux*. Regions are connected together by routes each of which allows organisms to move from one region to another, albeit at a certain energy cost. Located within regions are, optionally, the products of various organisms. These are

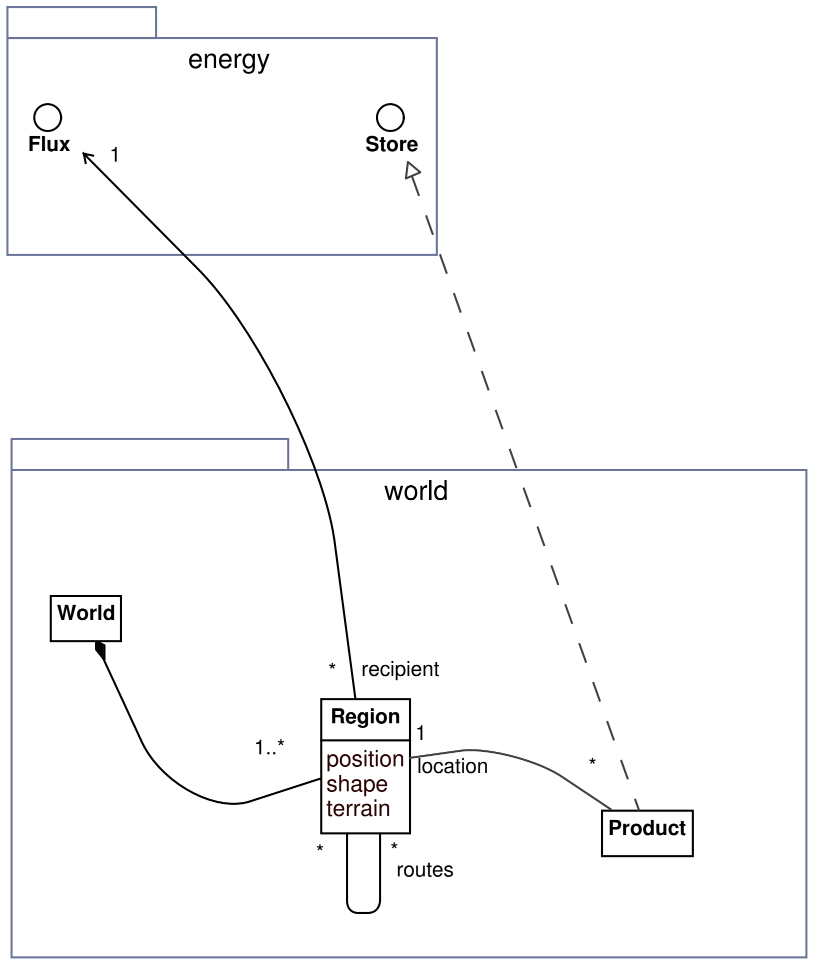


Figure 8.4: World metamodel.

energy stores left in the environment by organisms which might well form some of the energy supply for other organisms.

8.2.3 Organism metamodel

The organism metamodel is elaborated in figure 8.5. It has two components: the phenotype and the genotype, which are interdependent.

Genotype

The Genotype metamodel requires that an organism model (that is, something conforming to the organism metamodel) includes a specific genome, an instance of the Genome concept. This is the repository of the information needed to grow a new organism and should describe, for example, the specific behaviours that a particular organism can express. The genome of a phenotype is the result of a replication process that also creates a new Phenotype.

The whole point of this metamodel is to describe aspects of evolution within an environment bathed in an energy flux. For some even vaguely realistic form of evolution to exist there must be a definition of an organism's genome along with a mechanism for producing an offspring's genome, which must necessarily—in order for evolution to exist—be based on its parents' genomes but not necessarily identical.

The organism's genome is the result of a replication process which is an expressed behaviour of the parent[s] of a new organism and it is this behaviour that performs any mutation and variation. Subsequently, the new organism's morphogenesis behaviour acts so as as to grow the organism's phenotype, as a consequence of its genome.

Phenotype

The Phenotype metamodel expresses that an organism's phenotype, its structure, consists of a number of *body parts* and a number of *behaviours*.

Body parts are those components of the phenotype that store energy: this is expressed in the metamodel by the body parts realising the Store component of the

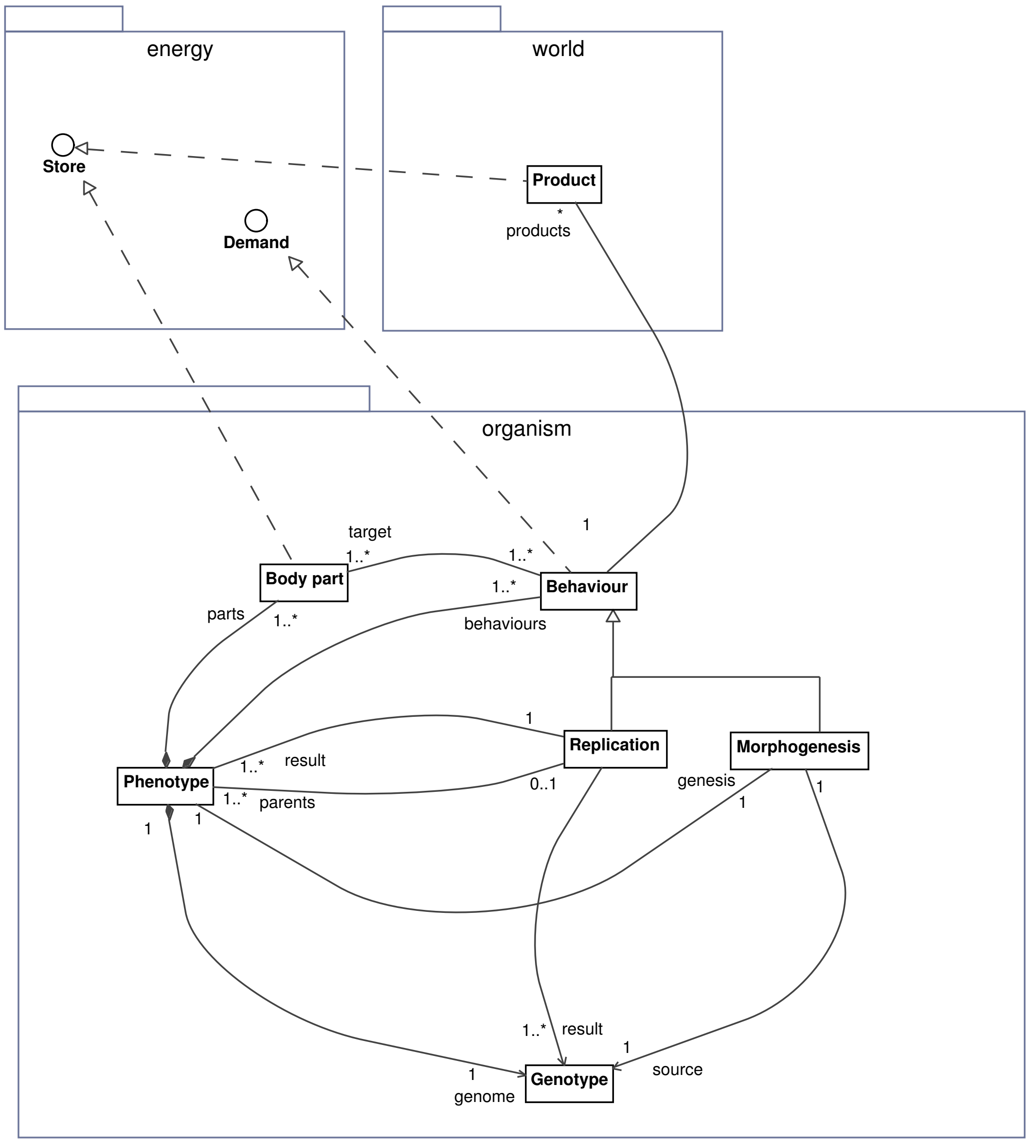


Figure 8.5: Organism Metamodel.

energy model. The body parts are also the *target* of the organism's behaviours. For example, a bird's wings might be the target of its "flying" behaviour. Each behaviour affects at least one part of an organism's body, and all parts must be a target of at least one behaviour. That is, the entire phenotype must be manipulated by the overall behaviour of the organism.

Behaviours are the component of an organism's phenotype that consume energy; they realise the Demand component of the energy model. The metamodel requires that all energy consumption of an organism is a consequence of the set of behaviours that that organism expresses. So, for example, a purely sessile organism must still include a behaviour that it continually expresses, which demands the energy needed to maintain its metabolism.

All behaviours, then, consume energy and that energy must be supplied by the energy stores represented by the set of body parts of which the organism is constituted. If an expressed behaviour requires more energy than is available from the organism's stores then then the organism dies.

Some of an organism's behaviours produce waste products, included as the *Product* component of the metamodel. Such waste products are in themselves further energy stores, although they are not part of the organism's phenotype. The entropy of the energy content of such waste products would usually be higher than that of the original energy source, but that does not preclude some organisms being able to scrape out an existence using such low grade sources of energy. If dung beetles, and similar organisms, did not exist there would be far too much dung in the world. A further waste product is the phenotype of a dead creature. Again this represents a low-grade source of energy, providing carrion-eating as a possible way of making a living in a world that conforms to this metamodel.

The model requires that all organisms possess a particular behaviour, the *Morphogenesis* behaviour that was the genesis of the organism's phenotype. Each organism must have its own genome. It may be that a number of individual organisms share the same genome but for a particular organism this is of no concern; it has its own genome and that is sufficient. The morphogenesis behaviour is responsible for building a new organism following the "instructions" implied by the organism's genome. The morphogenesis behaviour itself requires energy and consequently a new organism cannot come into existence without the energy to actually construct its phenotype.

The morphogenesis behaviour is required for an organism to construct its phenotype following its genome. Similarly, organisms must express a Replication behaviour which is responsible for creating the genome of an offspring organism. That is, it is this behaviour that must express the mutation, or similar functionality, that actually creates an offspring organism's genome as in some way different from that of its parents with the possibility that the new organism will compete in a different manner in the simulated world.

8.2.4 Discussion

The metamodel expresses essential requirements for evolution in an energetic context. It is not, of course, the only such metamodel as many others could exist, but this particular metamodel shaped the domain within which I experimented. It does not, though, define the specific models that might be implementable; that is, many models can conform to this metamodel. That is, a number of models could be produced, each of which conformed to this metamodel in the sense that the model's components were instances or realisations of components in the metamodel. Each such model would describe the domain model for a particular set of simulations in a particular domain that conformed to a particular metamodel.

Some other artificial life simulations incorporate a very basic notion of a constrained resource. Tierra [Ray92] uses CPU time-slices as an analogue of energy, with the size of the time slice being a tunable function of the entity's size. However, there is no analogue of an energy store that would enable entities to 'time-shift' their use of the resource, or hand on a surplus to their progeny; Tierra is a 'use it or lose it' model. (Ray [Ray92] mentions a possible extension allowing capture of CPU slices.) Stringmol [HCS$^+$10] is an artificial chemistry with an explicit, but very simple, energy model: a fixed number of energy units are added to the container at each timestep, and molecules need to use an amount to execute each instruction. However, the energy is a global resource (energy is not stored in individual entities, but in the system and accessible to all).

The rich energy metamodel described here provides a number of features that simulated organisms should be able to exploit to enable a range of different ways of making a living.

8.3 Energetic sticky feet

A wide range of simulation models are feasible that conform to the metamodel described here. I specifically chose to implement what is essentially a version of Turk's Sticky Feet [Tur10]. An outline of the model of this simulation (that is, something that has the role of a CoSMoS domain model) is shown in figure 8.6. This model conforms to the energy metamodel already discussed. This is an 'energetic' variant of Turk's Sticky Feet [Tur10]. It discusses the same sort of concepts that Turk uses, albeit in the context of energy, world and organism as prescribed by the metamodel.

I constructed the energetic sticky feet model, and the related simulation itself, with the intention of performing a range of experiments. The energetic sticky feet model is a simple model that conforms to the metamodel. It does not, though, include all of the concepts in the metamodel. In particular, I have not implemented the notions of entropy associated with energy, leaving that for future work. The regions in the world ("Zones" in the model), as well as suffering the energy flux required by the metamodel, also have a particular coefficient of friction, mu, which determines how much the creatures' feet stick to the floor in a particular part of the world.

My hypothesis, that these experiments should be capable of refuting, is that the presence of the energy model will influence the evolution of the simulated creatures in such a manner that a more diverse world will result than the sort of world that appears in the absence of an energy model. The rationale for this hypothesis is that even with the simplifications noted there are a number of possible ways for sticky feet creatures to make a living in the world bathed in energy. One of these is that which Turk's original implementation investigated, that where the creatures evolved to be better hunters of other creatures. In this implementation successful hunters were rewarded with the opportunity to breed, replacing a consumed creature with one that was (possibly) a mutation of the successful hunter. In my implementation successful hunters are rewarded by acquiring the energy of the consumed creatures which may then be used for survival and breeding, both of which are behaviours that consume energy. In general successful sticky feet hunters are relatively small creatures.

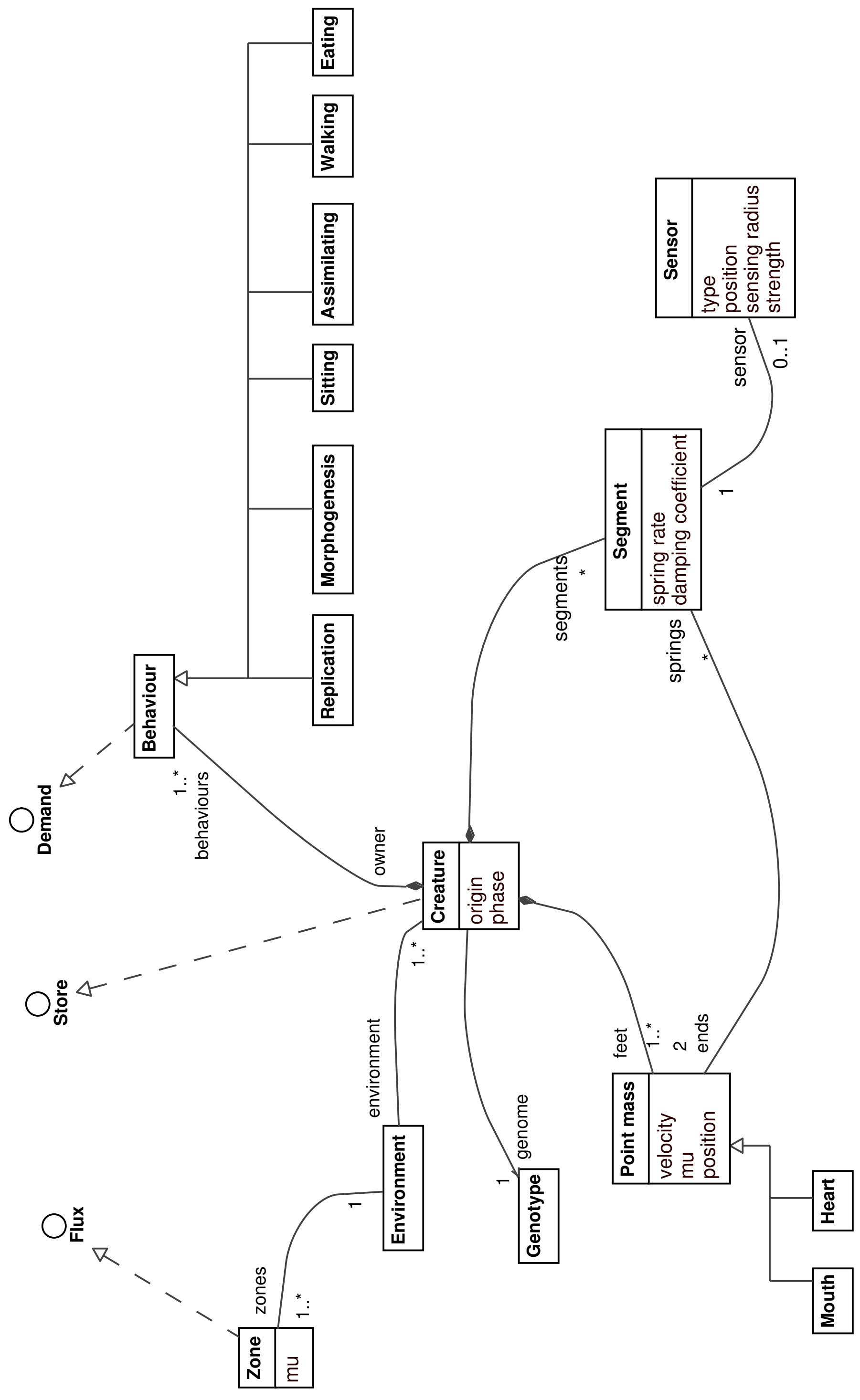


Figure 8.6: Energetic sticky feet overview.

Regardless of the size of the creature it must manouevre a mouth onto the heart of the hunted creature and this takes much more energy for a larger creature, simply because dragging around the potentially heavy feet of a larger creatures takes more energy.

A second way of making a living is to evolve a phenotype that is large and hence soaks up a large amount of energy from the world as it is exposed to more of the world's energy flux. If this mode of life is used it makes little sense to waste energy scuttling around and a successful creature could adopt a purely sessile way of life.

I tested the hypothesis by running energetic sticky feet simulations for a range of flux levels, measuring a key characteristic of the evolved creatures as the simulation progressed. In order to provide some sort of experimental control I also developed a variant of the simulation which did not use the energy model. Rather, this works in the same manner as the original implementation with a single reproduction being possible whenever a creature consumed another. That is, in this version of the simulation a constant population size is maintained in the same manner as in Turk's original implementation.

8.3.1 Energetic sticky feet implementation

Here I describe the important aspects of the energetic sticky feet implementation. The implementation follows the general guidelines established by the metamodel but is necessarily specific in several ways. Further, it follows the general pattern of the experimental platform described in chapter 5 although the actual code was largely re-implemented due to the rather different characteristics of the evolutionary context.

Body parts

The creatures in the model follow the metamodel in that each creature has a number of parts and a number of behaviours. The specific body parts are *feet* and *segments*. Following Turk [Tur10]: a foot is a point mass with a particular, and modulatable, coefficient of friction.

The equations that govern the forces that the springs exert on the feet are those of a damped spring. In order to provide control of the sticky feet creature the rest length of the spring is modulated with respect to the phase of the creature. That is, each segment changes its rest length sinusoidally with the phase of the creature. Each spring has a strength, which affects how hard it "pushes" on the feet—and how much it will compress when pushed on by other parts of the creature—and its motion is damped by the effects of a damping coefficient.

The feet themselves appear in three varieties. The basic ones are augmented with special variants, representing a heart, and a mouth. The heart represents the "essence" of a creature. When one creature's mouth gets close to the heart of another creature then the former may 'eat' the latter (assuming that the former creature is expressing the 'eating' behaviour).

Each foot has a coefficient of friction which determines how it slips in the current world, this takes into account the specific coefficient of friction of the region of the world.

The movement of the creatures in the world depends on the creature's feet slipping in the world in a manner that is modulated by the coefficient of friction of the feet. In order to coordinate this, each creature has at any point in time a *phase*, varying between 0 and 2π, describing where it is with respect to the overall phase of the simulation. The coefficient of friction of each foot is varied by a *Phaser* object that is attached to each foot. This object modulates the coefficient of friction of the foot in a manner that varies in synchronisation with the creature phase, although with a predefined lead or lag. So, for example, one foot might have its coefficient of friction varied in phase with the creature phase, another's might vary $\pi/2$ radians behind the creature's phase.

Because the different feet of a creature vary their coefficient of friction differently it is possible for a creature to move around the world; at one point the feet at the "front" of the creature might slide across the floor with little resistance, later in the cycle of movement the front feet might stick to the floor and the "rear" feet are easily dragged across the surface closer to the front feet.

As a consequence, the creatures appear to "walk" across the simulated world.

Each segment may optionally have an attached sensor as shown in figure 8.7, which senses the position of other creatures. A sensor may sense either the heart or the mouth of another creature and, when it does because it comes in range of that part, may perturb the oscillation of its attached segment by systematically modify-

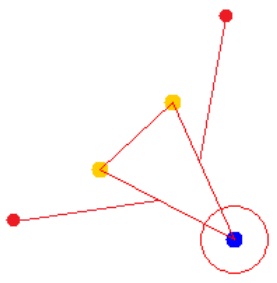


Figure 8.7: Sticky feet creature with sensors. The sensors are the red dots, attached to the indicated segments.

ing the rest length of that segment. In this manner the sensors may allow a creature to turn towards prey, or away from a predator.

Behaviours

The overall behaviour of each creature is represented by attaching a collection of individual behaviours to the creature. Each of these acts in a manner reminiscent of the *Command* pattern [GHJV95], and applies itself if it determines that the time is appropriate. Every behaviour demands energy, which must be provided by the owner of the behaviour. If the owner cannot supply the energy then the creature dies: it has exhausted its energy supplies.

The energetic sticky feet implementation does not implement the waste product component of the metamodel. Consequently, when a creature dies, it just disappears from the simulation, taking with it any residual energy.

The behaviours available to a sticky creature are as follows:

Sitting The 'null' behaviour that all creatures must express. This behaviour forces a creature to continually consume energy. The amount of energy consumed is a function of the complexity of the creature's phenotype; a larger, more complex, creature requires more energy just to sit in one place compared to a small, simple, creature.

Walking The behaviour that expresses the mode of walking explored by Turk [Tur10], by oscillation of the creature's segments. The size of the energy requirement is proportional to the friction against which work is done by the springs.

Eating The behaviour that allows a creature to look to see if any other creature's heart is in the vicinity of one of its mouths. If so, the former creature may 'eat' the latter. This adds to the eating creature's energy stores all of the energy of the eaten creature. The eaten creature is removed from the simulation.

Reproducing The behaviour that allows a creature to create offspring, with a genome that is a mutation of the single parent's genome. At each simulation step there is a probability, encoded in the genome, that a creature may express this behaviour. Energy costs are allocated to all the components of the phenotype, and a check is made that the parent has sufficient energy to construct the child organism. If so, and the child organism is deemed to be viable, then it is created and the energy store of the parent is shared equally between the parent and the child.

The new creature has just the one parent. That is, there is no opportunity for mixing the genomes of two parents as can occur in the biological world.

Morphogenesis The behaviour that is followed to construct the phenotype of a new organism from the genome generated by, optionally, mutating the genome of the organism's parent. This differs from the Reproducing behaviour in that it is responsible for building the phenotype of the organism from its genome whereas the reproducing behaviour creates the new organism's genome.

Assimilating The behaviour that allows an organism to gather energy directly from the flux in the current world. The amount of energy available is determined by the flux applied to the region of the world that the creature is inhabiting, and by the physical size of the creature. A larger creature, in the same manner as a large tree, can extract more energy from the flux, but needs correspondingly more energy to construct and maintain the larger phenotype.

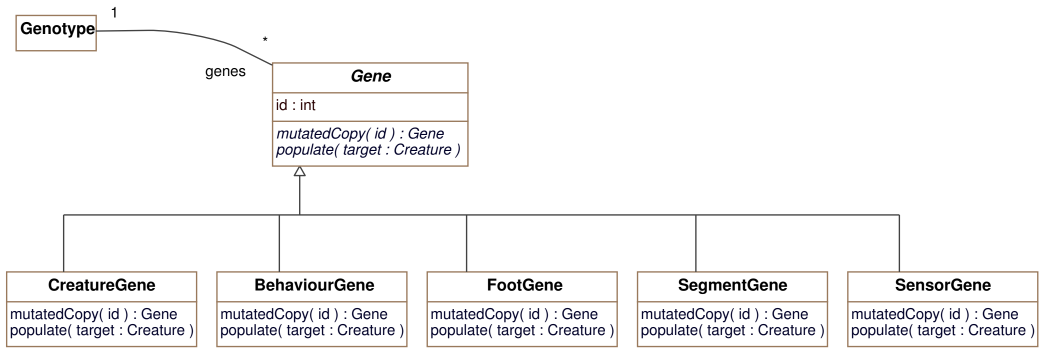


Figure 8.8: Energetic sticky feet genome structure.

Mutation and morphogenesis

In order to achieve the goal of evolving suitable behaviour for the available energy flux it is necessary to implement an approach to generating an offspring's genome as, possibly, some sort of mutation of the parents' genomes. Here I describe the approach taken in my implementation.

The implementation of the creatures' genomes follows the diagram in figure 8.8.

Each genome is, a sequence of genes each of which has a simple relationship with a part of the phenotype, either a physical part of the creature or a component of its behaviour.

Unlike Turk [Tur10] the model code does not express the notion of a 'species' in any way. Rather, each organism just has its own genome; even though it is likely that many other creatures have the exact same genome no use of this is made in any part of the simulation code.

Following Turk's lead I implemented two general forms of mutation. One of these is the modification of the various parameters that apply to each component. For example this allows the position of the creature's feet, the stiffness of the springs in the segments, and the probability that a creature will attempt to express the reproducing behaviour at any particular point in time to be varied.

The second form of mutation represents structural modifications of the phenotype. Specifically, these modifications may be performed:

- adding feet or segments,

- removing feet or segments,
- adding a sensor to a segment and
- modifying a segment so as to connect to a different foot.

A possible result of one or more of these mutations is that the eventual creature does not form a viable phenotype. For example, it is possible to generate a genome that implies a phenotype where the feet and segments are not connected as a single structure, or where a creature does not have a heart. I chose to declare these mutations non-viable, and to terminate the current cycle of reproduction when they occur.

Even if a mutation represents a viable creature, it is possible that the resulting creature cannot be incorporated into the current simulation world. Although in general the simulation allows creatures to crawl over the top of each other, unless one creature's mouth happens upon another's heart when the latter is consumed, I chose to disallow creatures to overlap upon initial creation so as to prevent the world becoming overcrowded. This, again, follows Turk's original implementation and, in the same manner as Turk my simulation attempts to find a place where the new creature will fit when it is created. In a manner different from Turk, who creates new creatures at a random location in the simulation space, my simulation attempts to place the new creature at a location that is in the region of its parent. This change is because the meta-model of the World allows the simulation space to include regions of different energy flux and regions with restricted connectivity. In such a situation creating progeny in a different region would not allow an adapted population to emerge in an isolated region. If no suitable location can be found then the current cycle of reproduction is, again, terminated.

This means, of course, that the simulation is essentially two dimensional in nature.

Viable creatures are created at a point in the simulation space that is local to their parent and also within the same Region as the parent. The simulations whose results are described below do not include worlds with multiple regions so this restriction does not, as yet, have any effect.

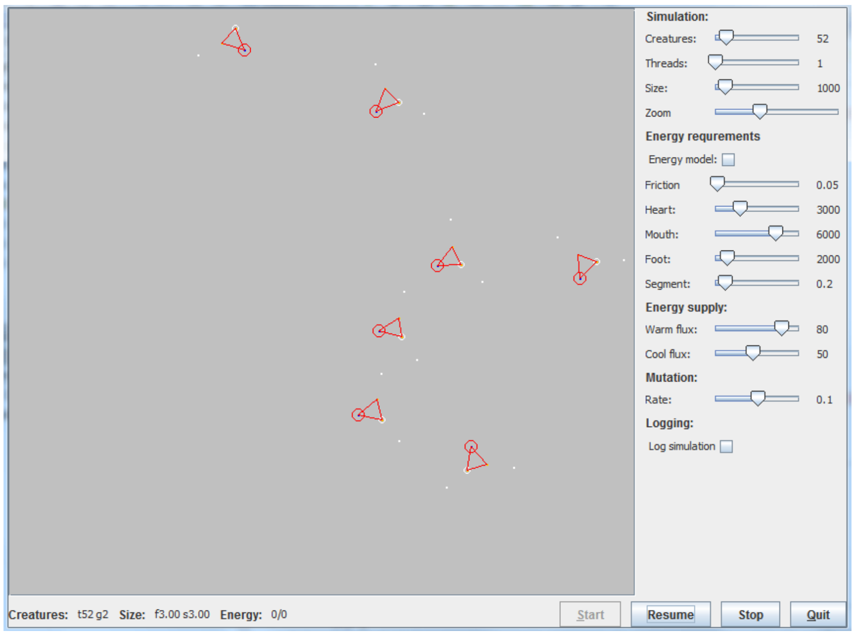
8.4 Implementation

The energetic sticky feet implementation follows closely the model shown in figure 8.6. The implementation is written in pure Java and uses the same environment oriented approach to represent the interaction of many creatures in a multi-threaded implementation. The implementation follows the same route as that discussed in chapter 6. In particular:

- Each sticky-feet creature exists within environments with which it exclusively communicates.
- One environment is analogous to the ProximityEnvironment seen earlier and provides the creature with information that its sensors and its mouths are able to interpret.
- A further environment represents the energy flux in the region in which the creature finds itself.
- The creatures behave by placing states in the environment and, as before, are provided with a Neighbourhood of information about their locality.

In this case the world is two dimensional with cyclic boundary conditions. As might be expected, though, the creatures know nothing of the shape of the world, that is represented in the environments. Each creature exists at a particular position in the world which is the position of its heart. The other parts of the creature are positioned relatively with respect to the origin, the heart, of the creature.

I produced two versions of the simulation. One has a graphical user interface and a view of this is shown in figure 8.9. The other implementation has a command line interface and runs without user interaction; this version being constructed so that a number of experiments could be run without the overhead of a user interface. Both versions have a built-in logging mechanism, following the structure discussed in section 5.2.2. This mechanism creates a (very large) log file describing various characteristics of the creatures and their lives within the simulation. This log file includes information about birth and death of creatures, where the creatures move to in each simulation step, the area of the creatures and their stored energy levels. The statistical data that is shown later in this chapter is derived by post-processing this



Simulation:
Creatures: 52
Threads: 1
Size: 1000
Zoom

Energy requrements
Energy model: ☐
Friction 0.05
Heart: 3000
Mouth: 6000
Foot: 2000
Segment: 0.2

Energy supply:
Warm flux: 80
Cool flux: 50

Mutation:
Rate: 0.1

Logging:
Log simulation ☐

Creatures: t52 g2 Size: f3.00 s3.00 Energy: 0/0

Figure 8.9: User interface of graphical version of energetic sticky feet implementation.

log file so as to get summary information about particular characteristics of each creature in a simulation.

Of particular importance here is the area of the creatures as it is this attribute that determines how much energy a creature receives from the available energy flux. The area is calculated by examining the positions of the creature's feet. Firstly, the feet that have only a single connected segment are ignored as they represent "whiskers" which just project from the creature. This process is repeated so that chains of single feet are removed. The remaining feet positions are used as the vertices of an irregular polygon whose area is then calculated (following [Res]). This process is outlined in figure 8.10. A creature that was two feet connected by a single segment (a frequently occurring shape) would have $area = 0$ and would not receive any energy from the world's flux. Although, of course, it might be able to survive by consuming other creatures. In situations where creatures overlap, and in particular where the energy absorbing polygons overlap, the flux is shared between those creatures.

Each run of the simulation takes a large number of parameters, some of which are directed by controls on the graphical user interface in figure 8.9. These param-

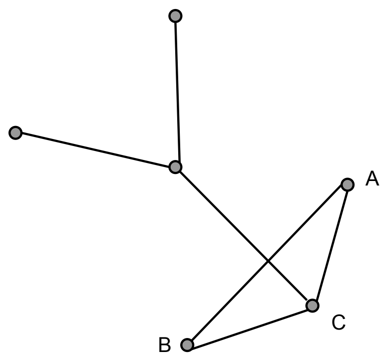


Figure 8.10: A sticky feet creature whose area is calculated. The algorithm used calculates just the area of the triangle ABC; feet with only a single connected segment are elided.

eters, a complete list of which appears in section 8.6, include:

- the level of the energy flux in the world,
- the number of creatures with which the simulation is seeded,
- the various energy levels that characterise the construction energy for the parts of the creatures' bodies and
- the mutation rate.

8.4.1 Creature seeding

I chose to initialise the simulation with a defined number of creatures, each of which carries a randomly (Gaussian) distributed amount of energy and which shares a common genome with all other initial creatures. The number of initial creatures is important because each creature is an energy store that is available for consumption (assuming that the creature hasn't wasted all its energy by scuttling around the world) by other creatures. That is, with a single initial creature a very high level of energy flux would be needed to ensure that that creature stayed alive, and reproduced sufficiently, in order to generate a longer-lived population of other creatures. This very high flux would essentially make life too easy in the simulation and the

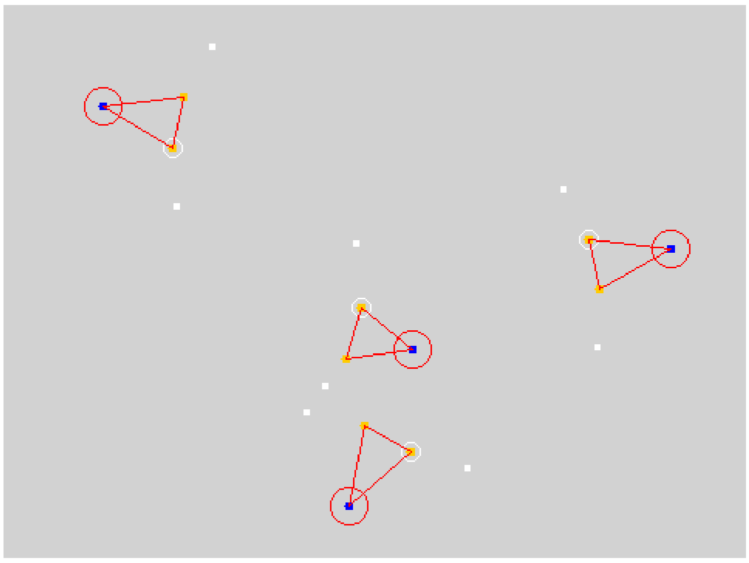


Figure 8.11: Initial seed creatures. The sensors are shown here as disconnected white dots.

population would, and indeed does in experiments, grow until the world is crammed with creatures and little new reproduction is possible.

A number of the initial creatures, then, are essentially there to act as food for other creatures. Unlike Turk's implementation, though, where many initial creatures are frozen in position to act as tethered targets for hunting, all the initial creatures are identical and move around the world in exactly the same general manner as all other creatures. In Turk's implementation the tethered creatures are provided in order to allow mutation and evolution to commence (in his case reproduction only occurs when another creature is consumed).

The specific creatures that are seeded all have a copy of the same genome, which is that of the triangular creatures seen on the display in figure 8.9. A detailed view of some of the initial creatures can be seen in figure 8.11.

8.4.2 Evolved creatures

As a simulation proceeds the creatures evolve in form and size. A typical collection of creatures after a period of evolution is shown in figure 8.12. These particular creatures appear very frequently in simulations, so much so that I gave them names.

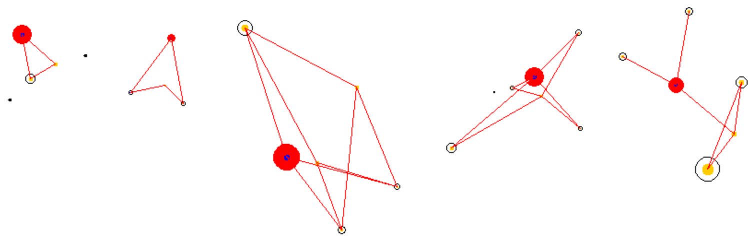


Figure 8.12: Some example evolved creatures; the filled circle is the heart, the open circles are mouths. The size of the blob drawn for the feet is derived from the mass of the foot. From left to right these are: a) the initial 'seed' creature; b) the 'manta ray', only a few mutations away from the seed; this creature has two mouths; c) the 'killer', large and fast; d) the 'multimouth', with lots of mouths that stab outwards; e) the 'spiky', with lots of mouths but little area.

There are a large number of parameters to the sticky feet simulations.

Initial experiments with the implementation show that careful setting of these parameters is necessary in order to allow the creatures to survive. In particular, it is very easy to set the parameters so that there is insufficient energy in the world for a population of creatures to survive; even though they can mutate to take advantage of their world they run out of time in which to do so. This is in some ways perhaps a consequence of the approach of seeding the world with a collection of fully formed creatures with significant energy demands.

8.5 Experimentation

In order to compare the simulations with something more representative of Turk's implementation [Tur10], that is an "energy-less" implementation, I needed a way of 'turning off' the energy model. That is, I needed to be able to run simulations in a manner that is not constrained by the availability of energy. In Turk's implementation the simulation has a fixed size population. This is a consequence of each creature reproducing once only when it consumes another. In this manner, the creatures can evolve without over-running the simulation world into complete chaos.

In a similar manner, I coded the energetic simulation to include an "unconstrained energy" option where the creatures function exactly as they do in the energetic world except that the *demand* of all behaviours is set to zero, so no energy is ever consumed, and the *reproducing* behaviour is only available, and indeed is forced, in the situation where the *eating* behaviour has been invoked. This has the effect of creating a fixed-population simulation[1] of a form similar to Turk's.

The differences between the implementation of the 'energetic' and 'unconstrained' variants of the simulation are minor. Hence, I can be sure that measured differences in the results of the simulations are a consequence of the inclusion, or exclusion, of the energy model.

In order to track the development of creatures as they evolve I use a notion of *mutation distance* in the experiments. As discussed I have no specific notion of 'species' in the implementation. Rather, each creature has its own genome, which has a mutation distance from the original genome of the seed creatures. All of these creatures have a copy of the same genome, which has *mutation distance* $= 0$. Whenever a creature reproduces it may also mutate the genome which is passed on to the child creature. The likelihood of allowing such a mutation is one of the simulation's parameters. After this mutation, following the process described earlier, the implementation compares the resulting genome with the initial genome. If they are different (they might not be because of the random nature of choosing whether to adopt specific mutations) and the genome represents a viable creature, then that new genome's mutation distance is incremented. If the mutation process does not yield a different genome then the child creature carries a copy of the parent's genome with the same mutation distance. It would be possible for a mutated genome to yield exactly the same genome as one that had previously existed, essentially reversing a set of mutations. The implementation makes no attempt at tracking such things, whose probability of appearance is very small.

In this manner every creature has a mutation distance, and this is part of the experimental results. There is not a simple relationship between *time* and mutation distance; it is possible, although unlikely, for example, for a creature with mutation distance 150 to co-exist in a simulation with another of mutation distance 0. The latter creature could have survived from the outset—the creatures do not die of old age—or it could be the end result of a series of reproductions that involved no mutations.

[1]Albeit on occasion a new creature cannot be 'fitted in' to the existing simulation, in which case reproduction is delayed until space is available.

8.5.1 Hypotheses

The direction of my experimentation was towards investigating two hypotheses. Firstly, the creatures evolving in the context of an energy model should do so in a manner that is measurably different from that which applies in a 'unconstrained energy' world. That is, the energy model, which is a richer model for the complex system, should be having some effect.

Secondly, the presence of the energy model creates a wider range of ways of the sticky feet creatures "making a living". For example, a creature could survive by eating other creatures, or it could survive by growing large enough to acquire sufficient energy from the regional flux. Such a mode of life could be further enhanced by abandoning movement as that could be seen as wasting precious energy. Hence, I hypothesise that when evolving in the presence of an energy model the sticky feet creatures will appear in a wider range of sizes during their evolution than happens in an 'unconstrained energy' world.

Similar hypotheses could be expressed about other physical aspects of the creatures such as their speed of movement. In the discussion that follows, though, I just look at the size of the creatures.

8.6 Simulation parameters

It is in the nature of complex systems simulations that there are a large number of parameters, which frequently have a profound effect on the results of the simulations. The main parameters affecting the results of the simulations performed here were as follows. In many cases following initial experimentation the values of these parameters were fixed for subsequent experiments and, in particular, the experiments that led to the results to be described in section 8.7.

Energetic a boolean parameter determining whether the simulation used the energy model or was the control no-energy version.

NumCreatures the number of creatures with which the simulation was seeded. All initial creatures matched the simple triangular creatures seen in figure 8.9. After initial experimentation this parameter was always set to 200.

Flux the level of the energy flux in the world. This was in many ways the key variable parameter.

MouthEnergy the amount of energy needed to construct a mouth part of a creature. After experimentation this was set to 6000 units, quite a large number because there is an advantage to a creature having many mouths.

HeartEnergy the amount of energy needed to construct the creature's heart. Set to 3000 units in all experiments described here.

FootEnergy the amount of energy needed to construct a foot that is not a mouth or heart. Set to 2000 units.

SegmentEnergy this parameter allows the energy needed to construct a segment to be calculated. The actual energy needed for a segment is calculated as $FootEnergy * length_{segment}^2$, the rationale being that the area of the creature will be proportional to the square of the lengths of the segments. Hence, if there are two creatures of similar shape but with one having longer segments, then that creature will consume more energy in construction.

SensorEnergy the energy needed to construct a sensor, set to 2000 in all simulations.

BehaviourEnergy the energy needed to construct a behaviour, set to 2000 in all simulations.

WorldSize the length of one rotation around the world, which has periodic boundary conditions.

FrictionEnergy a parameter that is proportional to the energy needed to overcome the friction of the feet sliding across the world; a critical parameter as small changes have a large effect on the amount of energy consumed by just moving around. After experimentation this was set to 0.13.

MutationRate the probability that a reproduction behaviour will attempt to mutate the parent organism's genome. Set to 0.10 in all experiments.

NormalMu the "normal" value for the coefficient of friction in the world.

ViscousDamping the viscous damping parameter k_d.

MouthRange the distance a mouth has to be from a heart in order to be allowed to eat the latter creature.

SensorModulation the amount that a sensor modulates the length of its attached segment.

FootMutationProbability if a reproduction behaviour is mutating the genome, this is the probability that a foot will be mutated.

BehaviourMutationProbability if a reproduction behaviour is mutating the genome, this is the probability that a behaviour will be mutated.

SegmentMutationProbability if a reproduction behaviour is mutating the genome, this is the probability that a segment will be mutated.

PhaserMutationProbability if a reproduction behaviour is mutating the genome, this is the probability that a phaser will be mutated.

FootMutationProbability if a reproduction behaviour is mutating the genome, this is the probability that a foot will be mutated.

FluxTakeup the parameter that allows the flux takeup of a particular area of creature to be calculated.

8.7 Results

8.7.1 Initial observations

The results summarised here are collected from a number of simulation runs where the runs were logged in the manner already described. Each such simulation is potentially very lengthy. In some circumstances, as will be seen typically when the energy flux is too low to sustain existence, all the creatures in the simulation die out relatively quickly. In other cases the simulation carries on effectively for ever. Creatures are born and die, both by running out of energy and by being consumed, but the simulation can still persist.

In total, I ran 500 separate simulations, varying the energy level. In addition I ran 50 simulations of the "no energy" variant, which essentially provide the experimental control. In all cases the experiments were terminated after 200 mutation steps; my experience was that after this point, assuming that the creatures hadn't died out, that the simulation had settled down into a typical mode of existence and no significant change occurred subsequently.

In the process of running these simulations, I captured over 100Gb of logging data.

Each simulation, of course, encompasses the lives of a potentially large number of creatures. Over all simulations the experimental results describe the lives of just over 1 million creatures.

Even early on in the experimental process it was clear that the energy flux had a significant effect on the simulations. At low energy levels the creatures rapidly died out. Only 44,000 creatures ever existed in simulations at an energy flux of 30 units, the figure at a flux of 80 units is 350,000 in comparison. At high energy levels there were also relatively few creatures in the simulations. At 150 units only 3,500 creatures ever existed. In this case the reason is because they could just live for ever; they tended to rapidly evolve sessile behaviour at which point they just sat in one place and tried to reproduce as much as possible.

8.7.2 Diversity measure

The hypothesis I was looking to demonstrate relates to the diversity of the population. There are a number of ways of measuring this and I chose a relatively simple one, nonetheless one that from experience is closely related to the success of the energetic sticky-feet organisms. The measure was the area of the creatures, as calculated to determine the flux takeup of the creatures and which was logged in the simulation files. I processed the simulation file to extract a simple summary of the entire life of each creature. As part of this summary I calculated the mean area (over time) of the creature. The rationale for using the mean is that the area of each creature varies in a cyclic pattern as it moves around the world. Furthermore, the area at its birth is often different from that which predominates over its life as each creature often "unfolds" to some extent after its birth as the strength of the springs in the segments overrides that of the initial positions of the feet.

These mean areas were then used for subsequent calculations. In particular I calculated the way that the mean areas changed for creatures as their mutation distance from the initial creatures increased at various different energy levels.

8.7.3 Results

A single summary of all this data is shown in figure 8.13. This figure shows a plot of the inter-quartile range of the sizes, that is the areas, of the population of creatures as it changes with the genome mutation distance. This figure includes data for three different configurations: the unconstrained ‘control’ situation, one with a single energy flux of 80 (arbitrary) energy units, and one with an energy flux of 100 units. Data for this plot are taken from a total of over 80 separate simulation runs and summarise the simulated lives of over 500,000 energetic sticky feet creatures.

I have chosen these particular energy levels based on experience running the simulations. Below an energy flux of 80 units it is invariably the case that the population of creatures dies out. For example, in all the experimental data no creature has existed in a simulation with a flux of 70 with a higher mutation distance than 94. At a flux of 50, I have seen nothing beyond mutation distance 73. That is, at this low energy flux then eventually the creatures die out. That they live as long as they do is because there are effectively two energy sources in the short term: the world’s energy flux and the energy stores that are represented by the seed creatures. Once the seed creature energy has been used up, the remaining creatures cannot survive with the world’s energy flux; if there is not enough of this, then the population disappears.

Looking to the figure, and concentrating at first on the no energy model, it can be seen that the creatures in these simulations initially explore a range of sizes from roughly 200 to 1000 units. However, as the simulations progress and the creatures continue to evolve, the range of sizes reduces until eventually there is little diversity in the population with all the creatures sitting around a size of 200 units.

In contrast, both the plots using energy show a much larger range of sizes throughout their history. At the end of the simulations, the range of sizes (from first quartile to third quartile) of the creatures in the $flux = 100$ simulations is around 1300 units. For the $flux = 80$ population the size of the IQR at a mutation distance of 200 rises to around 1900 units.

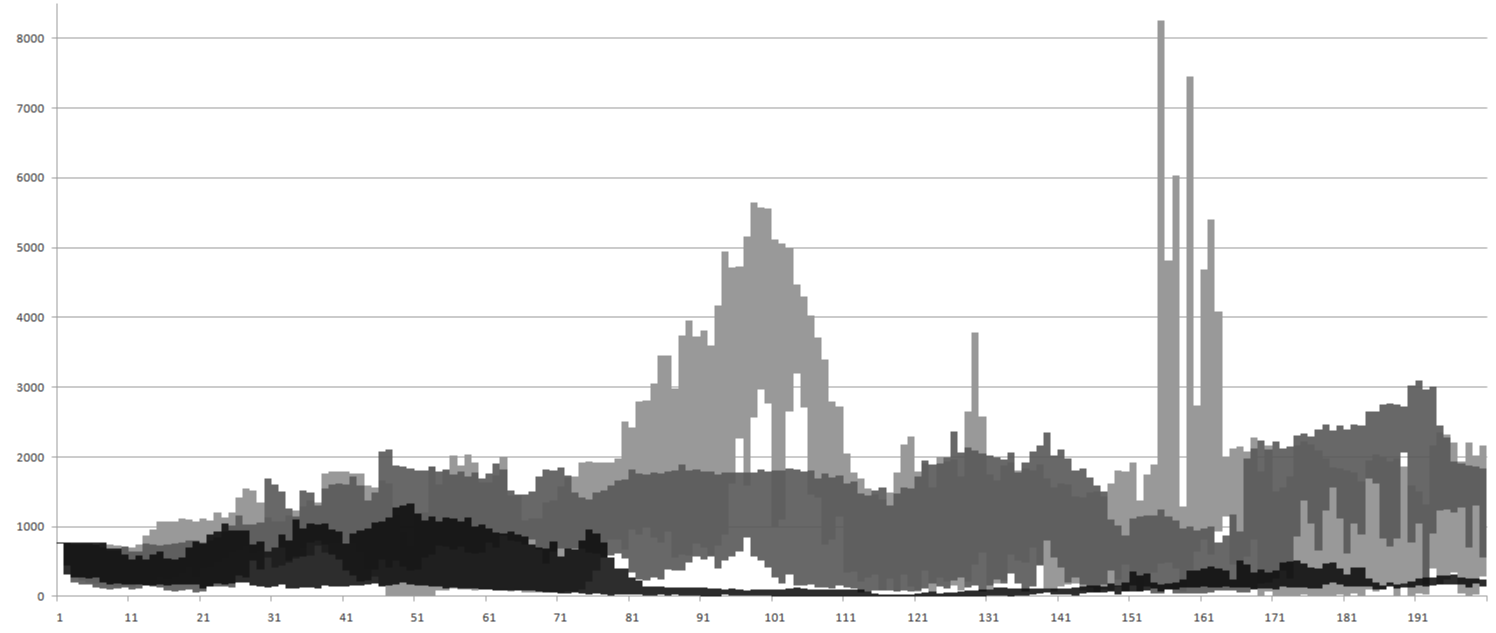


Figure 8.13: Summary of results of execution of energetic sticky feet simulation. Each vertical bar on the graph shows the inter-quartile range of the sizes of creatures at a particular mutation distance from the original seed population. The foreground, darkest, distribution shows these sizes using the control implementation that did not use an energy model. The mid-grey plot is the same information using the energy model at a flux level of 100 units. The pale grey plot shows the results using the energy model at a flux of 80 units.

So, from this summary of that data it is clear that there is indeed a distinct difference in the “with energy” and the “no energy” simulations. Secondly, that difference is indeed manifested in a greater diversity in the population. At the $flux = 80$ level, the inter-quartile range of the creature sizes at the start of the simulation is from 750 to 760 units. This population is entirely the initial seed creatures; the only reason for any variation at all is the way the sizes of the creatures changes as they move around. At mutation distance 200, the equivalent range is 282–2165 units, a roughly 7-fold size range; the first quartile is actually smaller than the initial population whereas the third quartile is about 3 times the size of of the initial creatures.

8.7.4 A sweet spot

Experience with the experiments, along with observation of the results shown here, leads to a further hypothesis. This is driven by the observation, seen in figure 8.13, that at $\mathit{energy} = 100$ there is a lower population diversity than at $\mathit{energy} = 80$. As I know that at lower energy levels the populations of sticky feet creatures usually die out I hypothesise that there is a critical energy flux density, in a set of simulations with otherwise consistent parameters, that generates creature populations of the widest diversity. This hypothesis essentially states that at low energy levels there is insufficient energy for populations to survive and hence they die out before generating significant diversity; at higher energy levels it becomes easier and easier to make a living, all the way up to the unconstrained world.

In order to summarise this I chose a single statistic to represent diversity of simulations with a particular energy flux, and looked to see if it varies in the hypothesised manner. The statistic I used is the distribution of the interquartile range of the creatures sizes across all energy levels. Figure 8.14 is therefore a box plot of the interquartile range (IQR) of sizes of creatures over all mutation distances. Each box therefore summarises the distribution seen in the interquartile range of the sizes of the population at a particular energy level. In figure 8.14, the median represents the median IQR of sizes over mutation distance at a particular energy flux (the median size of the bars in figure 8.13): the larger the median, the larger the range of sizes, hence the greater the diversity. In figure 8.14, the IQR represents the variation in the IQR of sizes over mutation distance at a particular energy flux (the

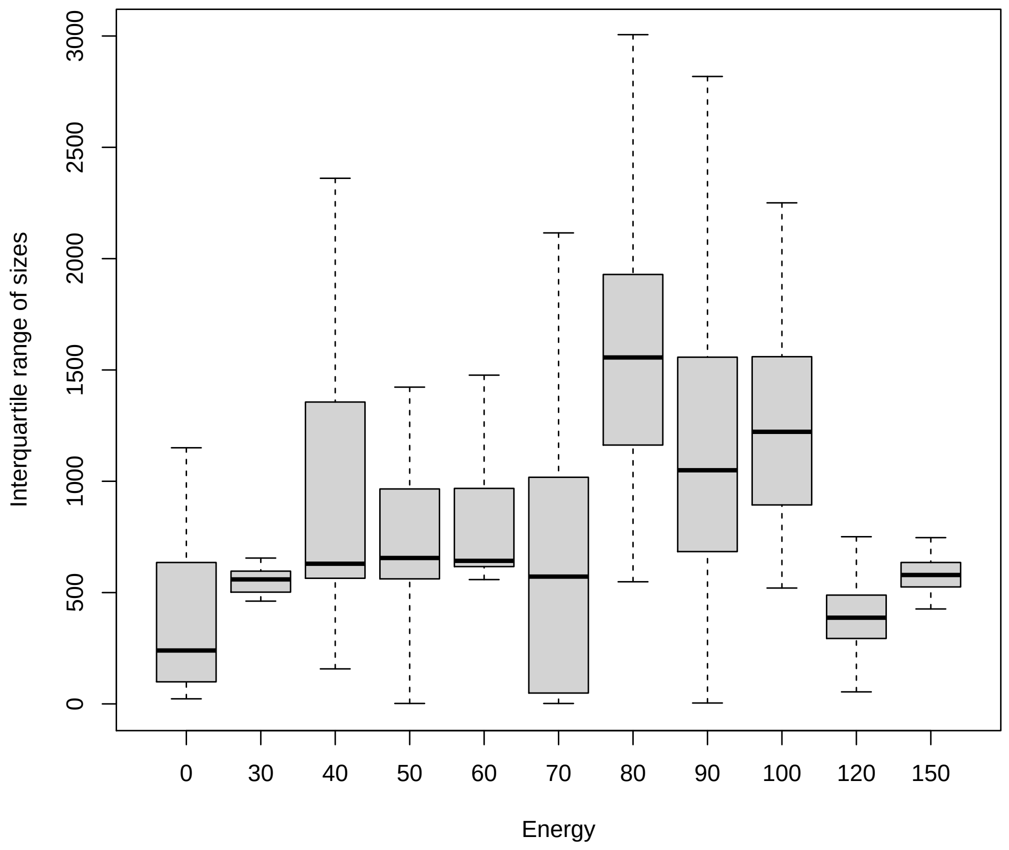


Figure 8.14: Box plot of inter-quartile ranges of creature sizes at various energy levels. Each box shows the distribution of inter-quartile ranges of creature size at a particular energy level. Energy 0 is the unconstrained case.

range of sizes of the bars in figure 8.13): the larger the IQR in figure 8.14, the larger the range of range of sizes, hence the greater the range of diversity. Observation of figure 8.14 does indeed show the hypothesised characteristic of a critical energy flux with maximum diversity which appears to be at 80 units.

8.8 Discussion

The role of this chapter, as I discussed at its start, is essentially two fold. Firstly to extend environment orientation to a non-trivial simulation context and, secondly, to experimentally investigate specific hypotheses in that context. I discuss these separately.

8.8.1 Open-ended evolution

The hypotheses that I have discussed are supported by the experimental results I have included. Specifically, the results seen when running the 'energetic' simulations show a more diverse range of creatures being produced than in similar "unconstrained energy" situations. Furthermore, there is a critical energy level that supports the widest diversity. At lower energies we see less diverse populations that soon die out as life is too hard in a cold world; at higher energies, and the unconstrained case, we see less diverse populations that nonetheless persist because life has become easy.

The critical energy level is the point between a low energy world where eating other creatures is a necessity of life, but nevertheless there is not enough influx of energy to survive, and a high energy world where there is little evolutionary pressure, and sessile behaviour is common.

At the start of this chapter I described the notion of open-ended evolution and how the simulations described here were experimenting with this notion in the context of an environment that itself had a strong effect on the simulation as well as being the mechanism for inter-agent action. However, visual inspection of the simulations make it painfully clear that although the simulations generate creatures with a wide range of sizes and structures they are still recognisably the same sort of thing: variations on a theme of feet and springs (figure 8.12). That is, it is not really doing that which was discussed in section 8.1.1: continually generating new forms. The end result is interesting but does not compare with biological evolution and the vast range of forms and structures that we see there.

Nor could it do so.

These simulations could never generate such a range of structures because the creatures' representation, morphogenesis and mutation operators are fixed, even though the various probabilities of their application and effect may change. That is, although I have a general notion of the sorts of energy and its place in the world, and this is encoded in the metamodel, there is not a similar notion of a range of organisms. Therefore, the evolution being explored here is not fully open-ended. Although there is indeed no fitness function, the creatures just exist and compete to survive in a world, they can never generate any genuinely different creatures. In order to do that a more abstract description of evolution would be required.

Nonetheless, the metamodel summarises the essential components of an energy-rich world which is a basic feature of real world evolution, and also of artificial life. With the guidance of the metamodel, whose elucidation was a key aspect to seeing the way into the structure of the simulations themselves, environment oriented simulations can indeed generate something that approximates some aspects of life, albeit artificially.

Creatures in this world exist in an environment which is their way of interacting with the world and other creatures as in other environment oriented simulations. Furthermore, that environment itself provides important aspects for the creatures' continuing existence: the energy to survive. Application of this metamodel in even a simple manner yields more complex, more interesting, results than is the case in an unconstrained world.

8.8.2 Environment oriented simulation

As before, the simulation in this chapter is constructed in an environment oriented manner. The specific implementation here, though is rather more complex due to way it is built up from a separate models of parts of the simulation domain; specifically the energy, world and organism meta-models. The relationships between these meta-models are explicit and implemented in the model built of the simulation implementation.

The meta-model/model implementation relationship is, here, elaborated by hand. Even so, constructing the explicit meta-models is useful, and was the first part of this chapter's work to be completed, as it defines the sort of world that is being simulated. Here, that definition is useful because the original motivation was, indeed, to scope the sorts of issues that would be relevant in the simulation of an "energy-rich" world.

The ultimate implementation follows the model so constructed, using much of the simulation platform described in chapter 5 and extended in chapters 6 and 7.

The underlying implementation, though, remains the same and the consequences are the same. In this case a large number of creatures evolve within a complicated environment, interacting with each other, without any need to address the issues of deadlock other than to to serialise updates to the underlying repository. As before, this interaction is not done directly. Rather, each creature interacts purely with the environments within which it exists. In those environments, in the same manner as I discussed originally in section 3.2 as the motivation for this work, the sticky feet creatures find information about the other creatures that share the environment and also the energy that feeds their continued existence. Again, because the interactions are all simply between the creatures and their environments deadlocks do not occur as long as transactions are serialised at the one point where this is the possibility of contention, at the underlying repository itself.

In essence, the architecture used and discussed in the rest of this thesis is a simple and elegant one allowing simulations of complex systems with a very large number of agents in a complicated world to be effectively completed. The experiments in this chapter, and previous ones, demonstrate that that is the case.

Chapter 9

Summary and conclusions

In this final chapter I revisit the contents of earlier chapters and discuss the avenues for future work.

9.1 Origins

The genesis of the work in this thesis was the work on "BlobWorld" described in chapter 2. Alexander's work discussed here is not specifically framed in the context of complex systems but it is clear from his published work, in particular [Ale04], that his notions of development of the built environment are very much those of independent agents coordinating, and competing, within physical, social and commercial environments.

The BlobWorld software is intended to support investigation into algorithms that produce structures that display some of Alexander's properties. The original intention here was to perform a large scale experiment looking to see if experimental subjects shared a subjective interpretation of various patterns that matched Alexander's concepts and which he explored in his various architectural projects. This did not turn out to be possible for various personal reasons. Nonetheless it is still an interesting concept; whenever I return to the diagrams in chapter 2 I am struck by the coherence of some of them.

Even without the experience of carrying out this experimentation the BlobWorld software, in particular for an interactive user, vividly creates a notion of independent entities juxtaposed, competing and collaborating, in a space. These concepts, and discussions with co-workers, led directly to the concept of environment orientation discussed in chapter 3.

In addition to the BlobWorld experiment discussed above it would be intriguing to return to the design of the software itself and to re-address it from an explicitly environment oriented point of view. It would be quite feasible to do this with the environment orientation experimental platform first described in chapter 5. The most appealing such implementation would lose the purely serialised approach of the original design and replace it with a concurrent one where the blobs exist within the fields due to other blobs, and other parts of the environment such as the physical geography. In such an implementation the blobs would try to optimise their individual positions; the interesting aspect is whether the overall view of all those positions would amount to a structure that displayed the concepts Alexander discusses and labels, rather enigmatically, as “wholeness”. That does, though, lead back to the necessity of a subjective view.

9.2 Environment Orientation

Environment Orientation as has been discussed, is an approach to the construction of computer based complex systems simulations based on the argument that agents in a complex system are not directly communicating with each other but through the medium of some environment and specifically the fields within that environment be they electromagnetic, aural, chemical or something else. In such a simulation the agents are regarded as placing information into the environment as a description of themselves for other agents to observe, and observing other agents by inspecting what the environment itself regards as their neighbourhood.

Agents in complex systems can always be regarded as interacting entirely with their environments even though that is just way of describing inter-agent communication. Similarly a complex systems simulation can be approached by constructing the sort of simulation described in this thesis.

Furthermore, the notion of an agent existing simultaneously in multiple environments, and consequently multiple fields with quite separate characteristics, is realistic. Implementing this aspect is harder, specifically because the fields are potentially orthogonal, but submits to the sort of approach used in control systems implementations, as discussed in chapter 7. Again, not surprising because the observations used in a control system are frequently made of variable values from different fields.

9.3 Concurrency

Building an explicitly environment oriented simulation is immediately appealing because it offers the hope of being able to build such simulations with only slight consideration paid to concurrency control. Approaching such a simulation from the point of view of the agents, and the communication between the agents, inevitably leads to concurrency management being the major issue in the implementation. As the number of agents in such a simulation grows the management of that concurrency becomes a huge problem.

This would not be the case if a simulation could run single-threaded. However, any large scale simulation will have to use a multi-threaded implementation to have any chance of provding the amount of computational power required for a simulation that would ideally support a very large number of agents. However, the number of threads (or indeed processes) will not approach the number of agents in a realistic simulation. Even things like the massive SpiNNaker project [PFT$^+$07], which is aiming to simulate complex neural networks, is aiming to support many thousands or even millions of agents (in this case neurons) with a single processor core.

The requirement for concurrency is ultimately because a large number of separate agents are sharing the same computational platform and this will continue to be the case for the foreseeable future.

As soon as an implementation is multi-threaded there is the possibility of deadlock as those threads must necessarily interact with each other, especially in the simulation of a complex system which is inherently about collaboration between the many agents in the system. The difficulties in using concurrent programming are well known and a large collection of mechanisms have been developed [Han77, Sam10] with which to manage them. Environment orientation is essentially another such technique, albeit one that is based on the observations of agent interaction in complex systems where it can be argued, as discussed in chapter 3, that any individual agent is only interacting with its environment rather than directly with other agents. Environment orientation uses one of the standard techniques for control of concurrency to serialise interactions with the environment itself thereby guaranteeing freedom from deadlock. Usefully, the particular such technique used in the simulation platform is that supported directly by the programming language in use.

Building environment oriented complex systems simulations is feasible, as I have shown in this thesis; such simulations continue to expose the expected emergent properties and their implementation is simplified by computational techniques developed in commercial computing in contexts that are in many ways similar to complex systems. For example, financial instrument trading systems include large numbers of agents spread around the globe who are interacting with the system, placing orders and interrogating the system to determine the current state of the market which is in itself influenced by those same orders. The sorts of computational structures developed for these systems bear some similarities to those I have proposed here although, of course, the number of agents in a complex systems simulation potentially outstrips anything found in this sort of context.

However, the example environment oriented simulations discussed in this thesis, and in particular the subject of the experiments described in chapters 6, 7 and 8 share a common over-simplification in that they have all worked with a single processor, albeit one with between 2 and 8 cores depending on the particular machine in use at the time. In such a situation it was reasonable to delegate the concurrency control that is necessary, specifically that around updates of the state repository, to a simple mechanism such as the Java `synchronized` keyword. A large scale environment oriented simulation would necessarily include many processor cores in many separate processors. Only some of those processors would be sharing memory and able to communicate quickly via that memory.

In such a context the underlying state repository would have to be implemented in some way that it is shared between the various processor address spaces. In commercial situations this is commonly resolved by the use of an underlying database management system; even though computations are carried out in the application level processors their results are persisted to the underlying database for access by any other part of the system. This is the basic approach of modern stateless computing and the common patterns such as REST [FT02].

9.4 Performance

The inevitable concern, though, is performance which rears its ugly head in such a context: certainly in commercial computing and inevitably in a complex systems simulation. There are, though, some immediately apparent approaches to attacking this issue. I describe these here, although at this point I am discussing what might be, rather than what I have actually done in the context of complex systems simulations.

9.4.1 Repository caching

Commercial systems address the performance requirements of such a distributed system in several ways, some of which seem at first sight to be relevant to complex systems simulations. One approach is to observe that any particular piece of information, characterised in the complex systems simulation context as an agent's external state, is read much more often that it is written. In the context of a query oriented bird flocking simulation a bird agent will update its state at each simulation step but that information will potentially be read by many other agents before it is next updated. All of the agents in the neighbourhood of the original will be interested in this state and the environment server would have to inspect the state. As such, this information is ripe for local caching, in a manner illustrated diagrammatically in figure 9.1.

In this situation when one agent requires to see the states for agents in its locality (which of course may not simply be a spatial locality) it retrieves these via the environment. If the environment server, which must be implemented in each processor, can find the required states in the local cache then they can be processed rapidly; typically a local cache would be an in-memory implementation. If some states are not found in the cache then they must be retrieved from the central repository, persisting them in the local cache on the way past; if they're needed once there is a high probability that they're going to be needed again. This process is illustrated by the sequence diagram in figure 9.2.

This apparently simple process is, inevitably, complicated by issues such as those discussed here:

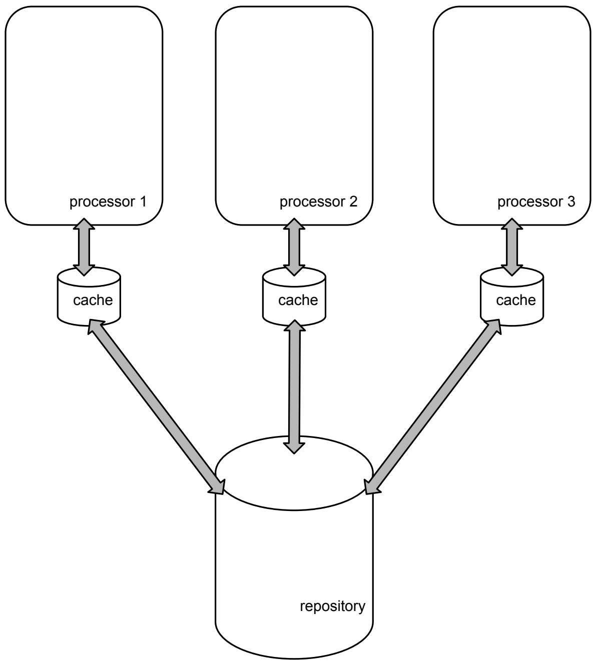


Figure 9.1: Per-processor state caches in a multi-processor environment oriented simulation

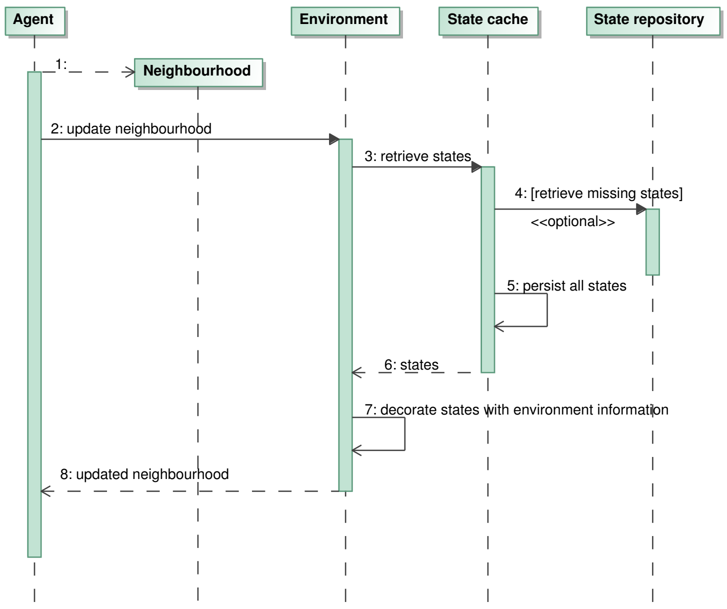


Figure 9.2: Sequence diagram outlining process of persisting information in a local cache.

State retrieval

Which particular states in a repository are those which are part of the querying agent's neighbourhood may not be easy to determine, without inspecting each one individually. In many commercial systems it is easy to retrieve items because an identifier of some form is specified. However, for a complex systems simulation this is not likely to be possible. The best way of approaching this issue is probably to adopt something that is analogous to the use of topics in subscription oriented systems; that of pre-marking states with something that will be of help in deciding what to retrieve. For example, in a system using spatial neighbourhoods the space could be partitioned in regular chunks and the chunk which contained a state could be decorated into the state. This approach, of partitioning the repository with respect to space, is followed in the implementation used for the experiments described in chapters 6, 7 and 8. In this the partition used is based on two dimensional space but other such partitions are equally possible.

Cache eviction

Once a state is in a cache then there would have to be a mechanism for removing it, if for no other reason than the original producing agent updating the state. In such a situation the normal approach is to broadcast something like a "cache eviction" message (often using support for such broadcasts such as multicast IP). Such a message, which would be received by all caches, would remove from the cache the identified state. (In this case, because the message is coming from the agent that "owns" the state and can uniquely identify it there is no difficulty in finding the specific state.)

Locality of reference

The structure as discussed here would only work with some sort of element of locality. In particular, if all the agents that typically needed to use each other's states resided in the same processor then that would reduce the likelihood of traversing to the repository all the time. An ideal distributed complex system implementation would migrate the agents between the processors to optimise this locality of reference. This is a problem similar to one that is being actively studied at the moment

[RSB09], that of processes in a multi-core system migrating between threads so that they can co-exist on the same thread, and more importantly core, if they are likely to interact. As such, it seems likely that migrating[1] agents that need to see each other's states can be performed.

9.4.2 Synchronisation

As soon as the agents are distributed across more than one processor (and, in reality, when they execute on separate cores of the same processor) the issues discussed in section 4.2.1 become important. That is, it becomes more important to ensure that some parts of the simulation, likely supporting a specific set of agents, is not allowed to "run ahead" of other parts of the simulation. Once the agents are allocated to different cores this becomes more problematic.

An appealing solution here is to adopt the virtual time approach discussed in section 4.2.1. This would be implemented by ensuring that all agents in the system have their own notion of virtual time. On every occasion when they chose to update their environments with some new state the environment would supply the current system-wide virtual time. The environments would decorate the states with the initial virtual time of the supplying agent, using the state decoration mechanism used in the experimental platform and discussed in chapter 5.

With the approach the environments, when building an agent's neighbourhood, would only use states that had a time appropriate to the agent.

The use of the virtual time technique might also be considered as related to the issue of cache eviction as discussed earlier. For example, as discussed in section 4.2.1 in a complex systems simulation then a small amount of "stale" data in not just acceptable, it is representative of the real world. As such, processing cache eviction messages in the context of what current virtual time of the agent that indirectly generated the eviction message, and the virtual time of the states becoming liable for eviction may well provide an opportunity of improvement of performance and, in particular, scalability.

[1]The agents themselves would not move, because all processors would have available the code for all agent. Rather, the responsibility of executing the n^{th} agent would move, in a similar manner as is common in stateless computer system architectures.

9.5 Conclusions and future work

This thesis, as discussed, demonstrates that environment oriented simulations of complex systems are feasible and certain technical issues, such as the combination of orthogonal environments can be solved. Furthermore, the software architecture needed for such simulation is actually relatively straightforward, building as it does on well-established notions of transactional software design that are still being evolved [ST95]. As such the simulator can concentrate on the complex system itself as is seen in the discussion of meta-models in chapter 8.

Inevitably, though, there are limitations to what has been described here, some of which are the basis for the discussions earlier in this chapter. These limitations are essentially those common to highly distributed implementations and include performance, scalability and synchronisation; although these are by no means independent. The experiments described here have all used numbers of agents which have at most been several thousand, something that has worked effectively on single-processor multicore hardware and where these issues are easier to solve.

There are, though, engineering approaches—such as those discussed in section 9.4—that offer a way of addressing these limitations to the point where a real complex systems, with potentially a great many more agents, could be simulated. The most important thing to do in the future with these ideas would be to build such a much larger simulation using the techniques discussed in this chapter and using larger-scale multi-processor hardware.

Bibliography

[ACG$^+$09] Frederico A.C. Azevedo, Ludmila R.B. Carvalho, Lea T. Grinberg, Jos Marcelo Farfel, Renata E.L. Ferretti, Renata E.P. Leite, Wilson Jacob Filho, Roberto Lent, and Suzana Herculano-Houzel. Equal numbers of neuronal and nonneuronal cells make the human brain an isometrically scaled-up primate brain. *The Journal of Comparative Neurology*, 513(5):532–541, 2009.

[AGH05] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Pearson, 3rd edition, 2005.

[AIS77] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, New York, 1977.

[Ale79] Christopher Alexander. *The Timeless Way of Building*. Oxford University Press, 1979.

[Ale04] Christopher Alexander. *The Nature of Order, Volumes 1–4*. Center for Environmental Structure, 2004.

[Ale08a] Christopher Alexander. Coherence operators. Personal communication, a chapter of a future book: Sustainability and Morphogenesis, 2008.

[Ale08b] Christopher Alexander. Morphogenesis of a window in a texas house. Personal communication, a chapter of a future book: Sustainability and Morphogenesis, 2008.

[App97] Brad Appleton. On the nature of the nature of order, 1997. http://www.bradapp.com/docs/NoNoO.html, last accessed 6 April 2013.

[APS+08] Paul S. Andrews, Fiona A. C. Polack, Adam T. Sampson, Jon Timmis, Lisa Scott, and Mark Coles. Simulating biology: Towards understanding what the simulation shows. In Susan Stepney, Fiona Polack, and Peter Welch, editors, *Proceedings of the 2008 Workshop on Complex Systems Modelling and Simulation, York, UK, September 2008*, pages 93–123. Luniver Press, 2008.

[APS+10a] Paul S. Andrews, Fiona A. C. Polack, Adam T. Sampson, Susan Stepney, and Jon Timmis. The CoSMoS process version 0.1: A process for the modelling and simulation of complex systems. Technical Report YCS-2010-453, Department of Computer Science, University of York, March 2010.

[APS+10b] Paul S. Andrews, Fiona A. C. Polack, Adam T. Sampson, Susan Stepney, and Jon Timmis. The CoSMoS process, version 0.1: A process for the modelling and simulation of complex systems. Technical Report YCS-2010-453, Department of Computer Science, University of York, March 2010.

[ASB+08a] P. Andrews, A. Sampson, J. Bjørndalen, S. Stepney, J. Timmis, D. Warren, and P. Welch. Investigating patterns for the process-oriented modelling and simulation of space in complex systems. In S. Bullock, J. Noble, R. Watson, and M. A. Bedau, editors, *Artificial Life XI: Proceedings of the Eleventh International Conference on the Simulation and Synthesis of Living Systems*, pages 17–24. MIT Press, 2008.

[ASB+08b] P. S. Andrews, A. T. Sampson, J. M. Bjørndalen, S. Stepney, J. Timmis, D. N. Warren, and P. H. Welch. Investigating patterns for the process-oriented modelling and simulation of space in complex systems. In S. Bullock, J. Noble, R. Watson, and M. A. Bedau, editors, *Artificial Life XI: Proceedings of the Eleventh International Conference on the Simulation and Synthesis of Living Systems*, pages 17–24. MIT Press, Cambridge, MA, 2008.

[ASH$^+$11] Paul S. Andrews, Susan Stepney, Tim Hoverd, Fiona A. C. Polack, Adam T. Sampson, and Jon Timmis. Cosmos process, models and metamodels. In *Proceedings of the 2011 Workshop on Complex Systems Modelling and Simulation*, pages 1 – 14. Luniver Press, August 2011.

[Bar09] Noma Bar. *Negative Space*. Mark Batty Publisher, 2009.

[Ber92] A. Berson. *Client-server architecture*. J. Ranade series on computer communications. McGraw-Hill, 1992.

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[BJ87] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. *SIGOPS Oper. Syst. Rev.*, 21(5):123–138, November 1987.

[BWS05a] Fred R. M. Barnes, Peter H. Welch, and Adam T. Sampson. Barrier synchronisation for occam-pi. In Hamid R. Arabnia, editor, *PDPTA*, pages 173–179. CSREA Press, 2005.

[BWS05b] Frederick R. M. Barnes, Peter H. Welch, and Adam T. Sampson. Barrier synchronisation for occam-pi. In *2005 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, pages 173–179. CSREA Press, 2005.

[CD94] Steve Cook and John Daniels. *Designing Object Systems: Object-Oriented Modelling with Syntropy*. Prentice Hall, 1994.

[CGS$^+$05] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM.

[Cos92] E Cos. Fuzzy fundamentals. *Spectrum, IEEE*, pages 58–61, Oct 1992.

[Dar59] Charles Darwin. *On the Origin of Species*. John Murray, 1859.

[DG89] J. L. Deneubourg and S. Goss. Collective patterns and decision-making. *Ethology, Ecology & Evolution*, 1:295–311, 1989.

[Dij71] E. W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1(2):115–138, June 1971.

[Edw99] B. Edwards. *The new drawing on the right side of the brain*. Jeremy P. Tarcher/Putnam, 1999.

[ELW02] Robert Eckstein, Marc Loy, and Dave Wood. *Java Swing*. O'Reilly, Beijing, 2nd edition, 2002.

[Eps08] Joshua Epstein. Why model? *Journal of Artificial Societies and Social Simulation*, Oct 2008.

[Esc38] Maurits Cornelis Escher. Day and Night, 1938. http://www.worldofescher.com/gallery/A11.html, last accessed 6 April 2013.

[FHA99] Eric Freeman, Susanne Hupfer, and Ken Arnold. *JavaSpaces Principles, Patterns and Practice*. Addison-Wesley, 1999.

[Fou] The Apache Software Foundation. Cassandra. http://cassandra.apache.org/, accessed on 6 April 2013.

[Fow02] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[FS03] Martin Fowler and Kendall Scott. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley, 2003.

[FT02] Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. *ACM Trans. Inter. Tech.*, 2(2):115–150, May 2002.

[FTA$^+$09] Anton Jakob Flügge, Jon Timmis, Paul Andrews, John Moore, and Paul Kaye. Modelling and Simulation of Granuloma Formation in Visceral Leishmaniasis. In *2009 IEEE Congress on Evolutionary Computation (CEC 2009)*, pages 3052–3059. IEEE Press, 2009.

[Gel85] David Gelernter. Generative communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, January 1985.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[GM02] Jean-Louis Giavitto and Olivier Michel. Data structure as topological spaces. In *Proceedings of the 3rd International Conference on Unconventional Models of Computation*, pages 137–150, 2002.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. The Morgan Kaufmann Series in Data Management Systems Series. Morgan Kaufíann Publishers, 1993.

[Gra81] Jim Gray. The transaction concept: Virtues and limitations. In *Proceedings of the Seventh International Conference on Very Large Databases*. Tandem Computers Incorporated, September 1981.

[Han77] Per Brinch Hansen. *The architecture of concurrent programs*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1977.

[HCS$^+$10] Simon Hickinbotham, Edward Clark, Susan Stepney, Tim Clarke, Adam Nellis, Mungo Pay, and Peter Young. Diversity from a monoculture: effects of mutation-on-copy in a string-based artificial chemistry. In *ALife XII*, pages 24–31. MIT Press, 2010.

[Hoa85] C. A. R. Hoare. *Communicating sequential processes*. Prentice Hall International, 1985.

[Hol94] Jan Rune Holmevik. Compiling simula: A historical study of technological genesis. *IEEE Annals of the History of Computing*, (4), 1994.

[HS09] Tim Hoverd and Susan Stepney. Environment orientation: an architecture for simulating complex systems. In Susan Stepney, Peter H. Welch, Paul S. Andrews, and Jon Timmis, editors, *Proceedings of the 2009 Workshop on Complex Systems Modelling and Simulation, York, UK, August 2009*, pages 67–82. Luniver Press, 2009.

[HS10a] Tim Hoverd and Adam T. Sampson. A transactional architecture for simulation. In *ICECCS 2010: Fifteenth IEEE International Conference on Engineering of Complex Computer Systems*, pages 286–290. IEEE Press, March 2010.

[HS10b] Tim Hoverd and Susan Stepney. Formalising harmony seeking rules of morphogenesis. In Harold Fellermann, Mark Dörr, Martin M. Hanczyc, Lone Ladegaard Laursen, Sarah Maurer, Daniel Merkle, Pierre-Alain Monnard, Kasper Stoy, and Steen Rasmussen, editors, *Artificial Life XII: Proceedings of the Twelfth International Conference on the Simulation and Synthesis of Living Systems*, pages 386–393. MIT Press, Cambridge, MA, August 2010.

[HS11] Tim Hoverd and Susan Stepney. Energy as a driver of diversity in open-ended evolution. In Tom Lenaerts, Mario Giacobini, Hugues Bersini, Paul Bourgine, Marco Dorigo, and René Doursat, editors, *ECAL 2011, Paris, France, August 2011*, pages 356–363. MIT Press, 2011.

[HWG06] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *The C# Programming Language*. Addison-Wesley Professional, 2 edition, 2006.

[ISO99a] ISO. *ISO/IEC 9075-1:1999: Information technology — Database languages — SQL — Part 1: Framework (SQL/Framework)*. International Organization for Standardization, Geneva, Switzerland, 1999.

[ISO99b] ISO. *ISO/IEC 9075-2:1999: Information technology — Database languages — SQL — Part 2: Foundation (SQL/Foundation)*. International Organization for Standardization, Geneva, Switzerland, 1999.

[Jef85] David R. Jefferson. Virtual time. *ACM Trans. Program. Lang. Syst.*, 7(3):404–425, July 1985.

[JL86] Philip Johnson-Laird. *Mental Models (Cognitive Science)*. Harvard University Press, June 1986.

[Kos94] Bart Kosko. *Fuzzy Thinking*. HarperCollins, 1994.

[Kra83] Glenn Krasner, editor. *Smalltalk-80: Bits of History, Words of Advice*. Addison-Wesley, 1983.

[KT04] Matthew T Kemp and J Khai Tran. Competition in tropical bird flocks: Army ant followers versus mixed species foraging groups. *Dartmouth Studies in Tropical Ecology*, pages 90–94, 2004.

[LBS98] Geoffrey Lewis, Steven Barber, and Ellen Siegel. *Programming with Java IDL: Developing Web Applications with Java and CORBA*. Wiley, New York, 1998.

[LCX$^+$01] Tobin J. Lehman, Alex Cozzi, Yuhong Xiong, Jonathan Gottschalk, Venu Vasudevan, Sean Landis, Pace Davis andBruce Khavar, and Paul Bowman. Hitting the distributed computing sweet spot with TSpaces. *Computer Networks*, pages 457–472, 2001.

[Mey00] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall PTR, 2nd edition, 2000.

[Mic07] Microsoft. SQL Server technical bulletin — How to resolve a deadlock, 2007. http://support.microsoft.com/kb/832524, last accessed 1 April 2013.

[Mil09] Robin Milner. *The Space and Motion of Communicating Agents*. CUP, 2009.

[MMG08] Thomas Robert Malthus, Robert Malthus, and Geoffrey Gilbert. *An Essay on the Principle of Population*. Oxford Paperbacks, 2008.

[MW97] J. M. R. Martin and P. H. Welch. A design strategy for deadlock-free concurrent systems. *Transputer Communications*, 3(4), 1997.

[Mye78] Glenford J. Myers. *Composite Structured Design*. John Wiley & Sons, Inc., New York, NY, USA, 1978.

[Ogb61] J. Ogburn. *The Passing of the Passenger Pigeon*. American Heritage, June 1961. American Heritage Publishing Co., Inc., 1961.

[OHE97] Robert Orfali, Dan Harkey, and Jeri Edwards. *Instant CORBA*. Wiley, 1997.

[OW04] Scott Oaks and Henry Wong. *Java Threads*. O'Reilly Media, 3 edition, 2004.

[PFT$^+$07] Luis A. Plana, Steve B. Furber, Steve Temple, Mukaram Khan, Yebin Shi, Jian Wu, and Shufan Yang. A GALS Infrastructure for a Massively Parallel Multiprocessor. *IEEE Design and Test of Computers*, pages 454–463, Sept-Oct 2007.

[PH95] Charles L. Phillips and Royce D. Habor. *Feedback Control Systems*. Simon & Schuster, 3rd edition, 1995.

[PZX$^+$09] Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, and Shan Lu. Pres: probabilistic replay with execution sketching on multiprocessors. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 177–192, New York, NY, USA, 2009. ACM.

[RATK09] Mark Read, Paul S. Andrews, Jon Timmis, and Vipin Kumar. A domain model of experimental autoimmune encephalomyelitis. In *Proceedings of the 2009 Workshop on Complex Systems Modelling and Simulation*, pages 9–44. Luniver Press, 2009.

[Rav68] F.H. Raven. *Automatic control engineering*. McGraw-Hill, 2nd edition, 1968.

[Ray92] Thomas S. Ray. An Approach to the Synthesis of Life. In *Artificial Life II*, pages 371–408. Addison-Wesley, 1992.

[Ree00] George Reese. *Database Programming with JDBC & Java*. O'Reilly Media, 2 edition, 2000.

[Res] Wolfram Research. Polygon area. http://mathworld.wolfram.com/PolygonArea.html, accessed on 6 April 2013.

[Rey87] Craig W. Reynolds. Flocks, herds, and schools: A distributed behavioral model. *Computer Graphics*, 21(4):25–34, 1987.

[RG11] S.F. Railsback and V. Grimm. *Agent-Based and Individual-Based Modeling: A Practical Introduction*. Princeton University Press, 2011.

[Ris] Risks. Forum On Risks To The Public In Computers And Related Systems. http://catless.ncl.ac.uk/Risks, accessed on 6 April 2013.

[RMHC09] Mark Richards, Richard Monson-Haefel, and David A Chappell. *Java Message Service*. O'Reilly Media, 2009.

[RSB09] Carl G. Ritson, Adam T. Sampson, and Frederick R. M. Barnes. Multicore Scheduling for Lightweight Communicating Processes. In John Field and Vasco Thudichum Vasconcelos, editors, *Coordination Models and Languages, 11th International Conference, COORDINATION 2009, Lisboa, Portugal, June 9-12, 2009. Proceedings*, volume 5521 of *Lecture Notes in Computer Science*, pages 163–183. Springer, June 2009.

[SAA$^+$13] Susan Stepney, Kieran Alden, Paul S. Andrews, James L. Bown, Alastair Droop, Teodor Ghetiu, Tim Hoverd, Fiona A. C. Polack, Mark Read, Carl G. Ritson, Adam T. Sampson, Jon Timmis, Peter H. Welch, and Alan F. T. Winfield. *Engineering Simulations as Scientific Instruments*. Springer, 2013. in preparation.

[Sal95] Nikos Salingaros. In defense of Alexander. *HALI: The International Magazine of Antique Carpet and Textile Art*, 78:67–69, 1995.

[Sam10] Adam T. Sampson. *Process-Oriented Patterns for Concurrent Software Engineering*. PhD thesis, University of Kent, October 2010.

[Sek09] Masatoshi Seki. dRuby and Rinda: Implementation and Application of Distributed Ruby and its Parallel Coordination Mechanism. *International Journal of Parallel Programming*, 37(1):37–57, 2009.

[ST95] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, New York, NY, USA, 1995. ACM.

[Ste07] Susan Stepney. Embodiment. In Darren Flower and Jon Timmis, editors, *In Silico Immunology*, chapter 12, pages 265–288. Springer, 2007.

[Tay99] Timothy John Taylor. From artificial evolution to artificial life. Technical report, University of Edinburgh, 1999.

[TS05] Jim Tice and Erik Steiner. The interactive nolli map of rome website, 2005. http://nolli.uoregon.edu, accessed on 6 April 2013.

[Tsu00] Reuven Tsur. Metaphor and figure-ground relationship: Comparisons from poetry, music, and the visual arts. *PSYART: A Hyperlink Journal for the Psychological Study of the Arts*, (000201), 2000. http://www.tau.ac.il/ tsurxx/Figure-ground+sound.html, last accessed 6 April 2013.

[Tur10] Greg Turk. Sticky feet: Evolution in a multi-creature physical simulation. In Harold Fellermann, Mark Dörr, Martin M Hanczy, Lone Ladegaard Laursen, Sarah Maurer, Daniel Merkle, Pierre-Alain Monnard, Kasper Støy, and Steen Rasmussen, editors, *ALife XII*, pages 496–503. MIT Press, 2010.

[WB04] Peter H. Welch and Fred R. M. Barnes. Communicating mobile processes. In Ali E. Abdallah, Cliff B. Jones, and Jeff W. Sanders, editors, *25 Years Communicating Sequential Processes*, volume 3525 of *LNCS*, pages 175–210. Springer, 2004.

[Wel02] P. H. Welch. Process Oriented Design for Java: Concurrency for All. In P. M. A.Sloot, C. J. K.Tan, J. J. Dongarra, and A. G. Hoekstra, editors, *Computational Science - ICCS 2002*, volume 2330 of *Lecture Notes in Computer Science*, pages 687–687. Springer-Verlag, April 2002.

[Wika] Wikipedia. Client server model. http://en.wikipedia.org/wiki/Client-server, accessed on 6 April 2013.

[Wikb] Wikipedia. Concurrency control. http://en.wikipedia.org/wiki/Concurrency control, accessed on 6 April 2013.

[Wikc] Wikipedia. Isolation (database systems). http://en.wikipedia.org/wiki/Isolation (database systems)), accessed on 6 April 2013.

[Wikd] Wikipedia. Multitier architecture. http://en.wikipedia.org/wiki/Multitier architecture, accessed on 6 April 2013.

[Wike] Wikipedia. Northeast blackout of 2003. http://en.wikipedia.org/wiki/Northeast Blackout of 2003, accessed on 6 April 2013.

[Wikf] Wikipedia. Publish-subscribe pattern. http://en.wikipedia.org/wiki/Publish subscribe, accessed on 6 April 2013.

[Wikg] Wikipedia. Sombrero function. http://en.wikipedia.org/wiki/Sombrero function, accessed on 6 April 2013.

[Wikh] Wikipedia. Transaction processing. http://en.wikipedia.org/wiki/Transaction processing, accessed on 6 April 2013.

[WJW93] P. H. Welch, G. R. R. Justo, and C. J. Willcock. Higher-Level Paradigms for Deadlock-Free High-Performance Systems. In *Transputer Applications and Systems '93*, pages 981–1004. IOS Press, 1993.

[Won93] Wucius Wong. *Principles of Form and Design.* John Wiley and Sons, 1993.

[YP09] W. Yu and M. G. Pollitt. Does liberalisation cause more electricity blackouts? evidence from a global study of newspaper reports. Cambridge Working Papers in Economics 0911, Faculty of Economics, University of Cambridge, March 2009.

[YRP94] Jun Yan, Michael Ryan, and James Power. *Using Fuzzy Logic.* Prentice Hall International, 1994.