Natural Computing manuscript No.
(will be inserted by the editor)

Environment Orientation

a structured simulation approach for agent-based complex systems

Tim Hoverd · Susan Stepney

Received: date / Accepted: date

Abstract Complex systems are collections of independent agents interacting with each other and with their environment to produce emergent behaviour. Agent-based computer simulation is one of the main ways of studying complex systems. A naïve approach to such simulation can fare poorly, due to large communication overhead, and due to the scope for deadlock between the interacting agents sharing a computational platform.

Agent interaction can instead be considered entirely from the point of view of the environment(s) within which the agents interact. Structuring a simulation using such Environment Orientation leads to a simulation that reduces communication overhead, that is effectively deadlock-free, and yet still behaves in the manner required. Additionally the Environment Orientation architecture eases the development of more sophisticated large-scale simulations, with multiple kinds of complex agents, situated in and interacting with multiple kinds of environments.

We describe the Environment Orientation simulation architecture. We report on a number of experiments that demonstrate the effectiveness of the Environment Orientation approach: a simple flocking system, a flocking system with multiple sensory environments, and a flocking system in an external environment.

1 Introduction

Complex systems are collections of independent agents interacting with each other and with their environment to produce emergent behaviour. When constructing agent-based computer simulations of such systems, the communication overhead of directly interacting agents becomes expensive, $O(N^2)$. Furthermore such a direct communication architecture makes it difficult to par-

S. Stepney
Department of Computer Science, University of York, UK, YO10 5DD

allelise and distribute such simulations without running in to deadlock issues, where parallel or pseudo-parallel agents are waiting for each other to progress before being able to progress themselves.

Here we demonstrate an alternative simulation architecture—Environment Orientation—designed to overcome these issues. The structure of the paper is:

- Section 2 overviews how agent-based complex system simulations are traditionally implemented, and the issues that arise.
- Section 3 discusses an alternative approach to agent interaction, which motivates the Environment Orientation architecture.
- Section 4 defines Environment Orientation.
- Section 5 discusses the requirements for an Environment Orientation simulation architecture.
- Section 6 presents experiments developed to demonstrate that Environment Orientation can be used to build complex systems simulations, and that its architecture results in a flexible simulation platform. The three experiments described are:
  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;
  3. to implement a simulation which includes an external environment representing the landscape within which the agents are moving.
- Section 7 discusses further issues needed to scale up, including to distributed platforms.

2 Complex systems simulation implementations

Agent-based modelling [27] is an approach commonly proposed for providing a complex systems simulation framework. The most obvious way of implementing such a model is the use of object-oriented programming [22]. This programming paradigm is now almost universal in industrial programming [3, 14] but, as a technique, it has its origins directly in the construction of explicit simulations [16].

A naïve implementation of an agent-based complex system simulation would represent each agent as an object; interaction between the simulated agents could then most obviously be achieved by sending messages. However, such a simple approach leads directly to significant difficulties. The issue that causes most implementation problems is 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.

Deadlock is a major concern in distributed systems in general, and has been much studied since problems such as the *Dining Philosophers* problem were described [5,15]. Current items of distributed enterprise software, such as Microsoft's SQL Server relational database management system [23], include deadlock detection and implement a brutal approach to resolving a deadlock: one of the deadlocked processes is summarily killed, allowing the rest of the implementation to proceed.

Deadlocks arise naturally in the context of complex systems simulations where agents in separate threads querying each others' states. A single-threaded implementation, typical of simple simulations with a small number of agents, would not suffer issues of deadlock. However, any larger scale simulation exploiting multiple cores or multiple processors would be multi-threaded. Predicting the likely patterns of interaction in such a situation is very difficult; hence the brutal solutions adopted by existing enterprise software.

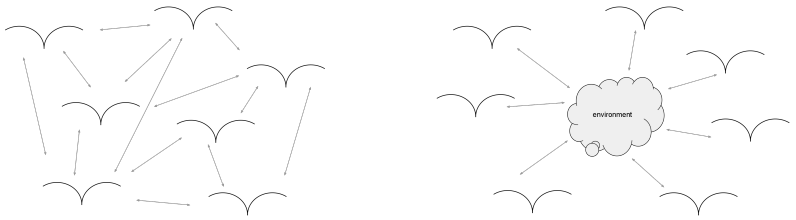
Some computational techniques, such as the "client server" pattern for concurrent systems [21,1] and barrier synchronisation [4], impose a processing pattern onto communications between the various components of the simulation that prevents deadlocks. All object oriented programming libraries include classes to serialise interactions and share physical threads.

These computational devices, though, are a consequence of the predominantly serial world of computation, and frequently have no simple analogue in the highly parallel world of independent agents interacting in complicated environments. These devices, typically expressed as patterns and class libraries, must be used explicitly by the programmer, and impose additional protocols onto the interaction of the simulation's agents, moving the simulation design away from the structure of the complex system being simulated.

For example, in the case of the client server pattern, processes are designated as either *clients* or *servers*. Clients request a service from a server and wait for a response. Servers wait for a request, which they honour, 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. In a complex systems simulation, such a simple configuration is not feasible for most situations, because all agents may be both clients and servers, at different moments, meaning that such an acyclic graph does not naturally appear.

Barrier synchronisation [4] provides a way for multiple agents to synchronise with each other. Enrolled agents individually synchronise on a barrier, and are blocked until all the enrolled agents have synchronised, at which point all the enrolled agents are rescheduled. This allows agents to 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 that is common in enterprise systems. They are a special case of layering in systems architecture, something that is now expected in all architectures [6, 7]. Although such patterns are of clear use in enterprise systems, they do not



(a) (b)

Fig. 1 (a) Diagrammatic representation of a collection of birds interacting while flocking with each other. Some interaction arrows have been removed for clarity. (b) Diagrammatic representation of a collection of birds interacting exclusively with their environment.

appear to exist in real world complex systems. Flocking birds 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 doing what it wants, when it wants, in whatever order it wants, in a deadlock-free world.

One objective of an explicit agent-based complex systems simulation is to reflect the complex system itself, without an excess of computational artefacts. Using patterns such as barrier synchronisation introduce such artefacts, and does not mirror the way the real world complex systems work.

3 Real world agents and their environment

3.1 Action at a distance *versus* mediating environment

The naïve model of a complex system, with agents directly interacting with each other, is essentially "action at a distance" (figure 1a). Each agent must explicitly know what other relevant agents exist, and must directly interact with them.

But is this how agents in a real world complex system actually interact? In reality the agents are not interacting with each other directly. Rather, a bird flying along reflects ambient light into the space around it; as it sings it causes sound waves in the air. Other birds sense this propagated light and sound. The birds are not directly communicating with each other; each is interacting only with its physical environment (figure 1b). The first bird places information into that environment; the environment propagates the information; the other birds sense changes in their own environment, some of which may be a consequence of the presence of the first bird. Other changes in their environment will be a consequence of the rest of the world, such as the level of ambient sunlight and the wind speed and direction.

This alternative model of interaction between the birds depends on the environment as the repository for information about the agents, which are embodied in the wider environment that can be observed by other agents. One bird can always look in the environment and see what another bird is

placing in the environment. It cannot, however, observe anything that is not in the environment.

The real world is such an environment. In this view the agents are *embodied* in the environment [31], and the environment provides services to those agents. Each agent does what it wants, without regard to direct interactions with other agents. 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.

If we take this model of the real world into simulation, we have a system where agent-agent interaction is expressed not directly, but rather as communication through the mediating environment. In this approach there is no direct interaction at all; the individual agents affect each other by virtue of existing within the same environment: all interaction is entirely stigmergic.

3.2 State

What information is placed in the environment?

In real world flocking, 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. But, from the point of view of flocking behaviour, other birds are interested only in its distance and relative velocity.

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 a subset of the agent's internal state, for example, in the case of the bird it could just be that part of its state that represents its position, velocity and current song.

However, complex system agents are essentially egocentric, even solipsistic. A flocking bird does not need to know where it is or what direction it is flying—it is always "here", flying "forwards"—it merely needs to know 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 overall context of execution of the complex system. However, that something does not need to be the agent. The environment can know where each agent is and can therefore also know which other agents are in the neighbourhood of each agent. That is, the environment knows things about the agents that are not part of the agent's internal state.

An agent can generate external state just by virtue of the physics of its environment. Photons bounce off a bird into the environment. Other birds detect some of those photons, enabling them to see the original bird, and to infer relative position and velocity information. This "involuntary" external state is contrasted with other state information "voluntarily" placed into the environment by an agent. Voluntary state could be singing a song.

3.3 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. The natural implementation is some form of loop where each agent asks the environment for information about other relevant agents' states (the agents it can see, or hear, for example), then it uses this state information to update its own internal state appropriately, then it tells the environment its updated external state.

The environment can be responsible for modulating one agent's state as peceived by another agent, depending on its distance, for example. In bird flocking, one of the items in a query result could represent a bird that is close to the querying agent. As such, the environment could describe the (relative) position of the nearby bird and its velocity quite precisely. If the bird being described is distant from the querying agent, however, then the environment might not report the position of the bird so precisely. For example, it might report in what direction the bird lies, but report its distance as "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 physics of the world.

Concentrating on the environment provides a way, in a simulation, 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 modelling 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 ant-trail formation, ants leave a trail of pheromones. The responsibility for simulating the decay of the pheromone may be allocated, depending on the requirements of the simulation, either to the environment itself, or to a special agent whose external state is the level of pheromone along the trail.

3.4 Multiple environments

Interaction between agents may simultaneously follow a number of different patterns. One extension of the simple spatial model is to note that a person is physically "near" a collection of other people, but may nonetheless communicate simply with spatially distant people via their telephones. (Milner's *bigraphical model* [24] is designed to model both a spatial and a connectivity configuration simultaneously.) That is, there are two kinds of "nearness", physical nearness and "communicable" nearness, forming two distinct neighbourhood environments.

So agents can be simultaneously embedded in multiple environments, each with its own particular properties. For example, birds reflect photons, allowing them to be detected visually. They also can sing, exploiting a separate audio environment, so that other birds can detect them even if they are not easily

visible. The characteristics of a sonic environment are rather different from a visual one: sound can go round corners, for example.

The approach discussed here allows these multiple environments to exist simultaneously. An agent may place relevant external state in each environment. An agent that receives multiple states for another agent, from multiple environments, can fuse the information to form a coherent picture of the other agent.

4 Environment orientation

The contrast between current simulation implementations (section 2) and real world interactions (section 3) motivates our introduction of Environment Orientation as a new architecture for constructing agent-based complex systems simulations. Adopting this approach removes all direct interactions between agents. Rather, all agent behaviour is mediated through environments, within which the agents are embodied.

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 modelling decision. Each simulation should be constructed with the explicit knowledge of which aspects are to be embodied in the environment.

The elements of Environment Orientation are:

1. Each agent has its own internal state, representing what the agent knows of itself.
2. Each agent publicises some aspects of its state, its "external state", to the environment(s) within which it is embodied. The agent may decide when to publicise its external state.
3. Simulation of an agent's behaviour is performed without reference to other agents. The behaviour depends only on the contents of the agent's internal state and environment(s).
4. An agent can retrieve environmental information about the external state of other agents.
5. An environment mediates external state information, determining which other agents can perceive it, and how it is propagated to them.
6. An environment itself is a first class computational component and can have state and behaviour of its own.

The rest of this paper investigates our claim that **Environment Orientation is an appropriate software paradigm for agent-based simulation of complex systems**. We test this claim through a number of experiments, building complex systems simulations of several forms.

5 Software architecture requirements and implementation

5.1 Client Server architecture

The Environment Orientation model is essentially a client-server one: 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;
3. presenting the state information to agents in a manner assimilable by the agents.

This 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 [11]. 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.

The "server" (environment) in a complex systems simulation must provide a number of services to a client (that is an agent of the simulation). We discuss these services and the manner in which they can be provided in more detail.

5.2 Environment service requirements

The services discussed in section 5.1 are requirements for an abstract architecture.

5.2.1 State retention

An agent (client) must be able to supply some state to the environment (server) that will be retained and supplied to agents at a later time.

Much of this state information will need to be interpreted only by the agents themselves; we assume that the agents share the metadata that describes the contents of the external state. However, the environment server 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. So the environment is also required to have access to the state metadata. For example, all parts of a simulation agree as to the representation of things like "position" and "velocity".

The environment must store a large collection of agent external state objects, elaborated by additional information added by the environment itself, in some sort of store.

5.2.2 State update

Agents update their own internal states, and supply updated external state information to the environment.

A simulation may have other environmental state information, for example, pheromone intensity on an ant trail. Pheromone decay could be implemented as a function of the environment, or it could be implemented by agents explicitly representing the pheromone physics. Whichever approach is taken, the simulated environment must provide for such state update.

5.2.3 State access

The environment must provide the agents with access to the external state of other agents. There are two general strategies.

Query oriented: an agent that wishes to see the external state of a set of other agents makes a *query* of the underlying environment server. The query provides the server with the information it needs, along with its knowledge of the agents, to select the relevant information and provide it to the querying agent. This is appropriate for systems, like bird flocking simulation, where an agent's environment changes rapidly and apparently continuously.

Subscription oriented: agents inform the server of the sort of information they are interested in, and have that information delivered as and when it is available. In the meantime the agent carries on with its normal behaviour. This is appropriate for systems where some information is available only occasionally and unpredictably, or where it is needed to "interrupt" an agent from its normal activities, in situations where the particular environment is changing intermittently.

Both these approaches provided *communication orthogonality* [10]. A receiver of a message (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. The sender of a message (the state placed in the repository) does not care which agent sees that information, as it is freely available in the environment.

5.3 Implementation issues

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 [21,32].

5.3.1 Transactions and locking

Environment Orientation can be considered as a *transactional* approach to agent-based complex systems simulation. Each agent 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 obtains from the environment, whether by querying or getting a subscription update, the external states of other agents and, in turn, *updates* the environment with its new external state.

The transactional approach to software design is common in commercial computing. In the context of Environment Orientation a simplification is possible. The external state stored for each agent is only written to by that agent, so there is no possibility of contention when updating the environment. As long as updates to the environment are atomic and invisible until committed, no further locking is necessary. These atomic updates of external state by individual agents mean that it is never necessary to roll back a transaction.

5.3.2 Time and fairness

When agents do not directly communicate with each other and do not need to synchronise, as when using Environment Orientation, there is no external shared sense of time: agents can perform transactions whenever they like, which makes it possible for agents to execute at different virtual rates. 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 limited only 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 would use the "virtual time" technique used in event-based simulation [19], in which a dimensionless *virtual time* is represented as a monotonically-increasing tag tracked by simulation components.

The notion of being able to effectively run a simulation without excessive concurrency control and allowing "sloppy" synchronisation is appealing.

5.3.3 Reproducibility

Complex systems simulations are generally seen as providing completely reproducible experiments. However, Environment Orientation inherently introduces non-determinism into simulations, in particular if the notion of virtual time (section 5.3.2) is adopted.

Simulation can be considered as a scientific instrument [2] that we use to understand the behaviour of a system. Like all instruments it has a degree of uncertainty in its results. The scientific method deals 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. 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. 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.

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 [26] 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.

5.3.4 Robustness

An ideal complex systems simulation would be as robust as the real world complex system itself. In Environment Orientation, each agent's behaviour is largely independent of the other agents in the simulation, and is not affected by precise time steps and explicit global synchronisation. This decoupling should tend to increase the robustness of the simulations.

The flexibility of Environment Orientation, from 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.

5.4 Implementation techniques

A possible implementation of environment orientation is as a multi-tier architecture, as in the commonest commercial software architecture. This abstract architecture has the following layers:

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

5.4.1 Repository

One motivation for Environment Orientation comes from the issues of deadlocks, arising from inter-thread, or process, resource contention. In Environ-

ment Orientation the only locus of contention between threads is in the repository. As long as the repository provides for atomicity and durability [12] of updates then the system is guaranteed to be deadlock free. 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;
3. atomicity of updates.

A number of implementation techniques could provide for these requirements:

1. tuple spaces, or "blackboards", a concurrent access memory implementation, as originally provided by Linda [10], and now available in JavaSpaces [8] (Java/Jini), TSpaces [20] (Java), Rinda [30] (Ruby);
2. a relational database management system (RDBMS), for a query oriented approach;
3. conventional object-oriented collections, with external state represented as an opaque object (the structure being defined by the agents), directly stored in the collection; such collections can be made to operate atomically; the requirement for the environment server to augment the states with additional information can be supported using the Decorator pattern [9].

5.4.2 Threading and locking

An implementation of a significant Environment Orientation simulation would likely execute on a multi-core processor, or a distributed platform, or both. Execution would take place simultaneously in a number of threads. 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.

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.

6 Validation

We have designed and implemented a flexible Environment Oriented simulation platform, and run a variety of different experimental simulations, in order

to validate the Environment Orientation approach, and the derived architecture requirements above (see [17, ch.5] for detailed design).

We have deliberately chosen a simple implementation approach, as our aim is to investigate the essential concepts, not to maximise performance, at this stage. We use Java [3] as the implementation language. Java includes a rich set of primitive implementation classes which ease the implementation. In particular, the threading classes and primitives [25] ease the use of parallelism which is a necessary characteristic of the required platform.

In this section we provide experimental results aimed at some basic claims relating to Environment Orientation:

1. Emergent properties expected from agent interactions in conventional agent-based simulations also appear in Environment Oriented simulation with no direct communication between the agents.
2. Simulations can be extended to multiple environments without reworking the simulation's architecture.
3. Simulations can be extended to include representations of the physical world without without reworking the simulation's architecture.

These experiments are performed by incrementally extending the implemented simulation platform with simulation-specific classes. In each case we describe the extensions made to the platform to provide the new facilities, and the results of running the simulations. The complex system chosen for the experimental evaluation is Reynold's boids [28].

6.1 Flocking in a single environment

Reynolds' algorithm describes how an agent representing a bird behaves according to a simple set of rules resulting in emergent flocking. Each boid:

- moves towards the centre of mass of the other boids in its neighbourhood;
- attempts to match its velocity with the overall velocity of the birds in its neighbourhood;
- moves away from nearby boids, so avoid collisions.

The simulation generates a movement vector for each boid, formed from the vector sum of the movements due to each rule. This vector is applied to the position of each boid at each simulation step.

6.1.1 Claim

The known emergent property, flocking, arises from an Environment Orientation simulation.

In order to investigate this claim it is necessary to define what is meant by a flock. For this experiment this was defined as follows. In order to be "flocked":

1. an agent must retain over 20 simulations steps a minimum of 10 agents in its vicinity
2. at least 80% of the set of neighbourhood agents must be the same agents across all those steps

6.1.2 Platform specialisation

The bare simulation platform was specialised by adding several concrete subclasses (including FlockingAgent, FlockingEnvironment) of abstract platform classes (Agent, Environment).

FlockingNeighbourhood, a subclass of Neighbourhood, knows the absolute position and velocity of each boid agent. It updates boid positions and velocities from the velocity state published by the boids. It implements the behaviour of calculating the three Reynolds vectors of the flocking rules: centre of mass, average velocity, and proximity, of the agents in the relevant location.

FlockingAgent, a subclass of Agent, adds a velocity state to the basic Agent. Each agent knows its own velocity; at each step it updates its velocity by an amount according to the three Reynolds vectors provided by the environment, and publishes its new velocity.

In order to determine if the notion of an agent being "flocked" was being achieved (section 6.1.1) an instrumentation class was added to produce information describing how many of the simulation's agents were flocked.

6.1.3 Simulation results

The simulation of the flocking complex system has several parameters, including the number of boids, and parameters in the three the Reynolds rules. As a test of the platform, we ran a number of tests of flocking, varying a single one of these parameters, the neighbourhood proximity, determining whether other agents are "in the neighbourhood" of a specific agent.

We used the instrumentation class to determine the proportion of the agents in a simulation that were "in a flock" after a two million simulation steps. We ran 20 simulations for each value of the proximity, to check how successful the agents were at ending up in a flock.

The chart shown in figure 2 summarises the results of these simulations. In each simulation 300 agents were introduced into the central 500 unit radius circle of an infinitely-sized flat world. The chart boxplots show the proportion of the agents that were in a flock at the end of each simulation run.

The agents are indeed forming into flocks. As expected, changing the proximity parameter improves the system's ability to incorporate more of the agents into the emergent flocks. At small proximities, where an agent's neighbourhood would be expected to include only a few other members of the simulation, flocking does not occur.

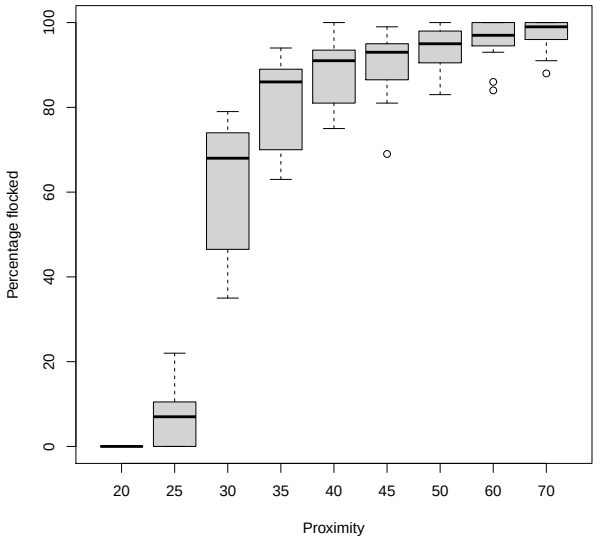


Fig. 2 Box plot showing the result of flocking with a single environment that builds a neighbourhood based the proximity of agents to the querying agent. Each box shows the lower quartile, median and upper quartile of the percentage of agents that ended in a flock over a set of 20 simulation runs for 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.

6.1.4 Discussion

This flocking example validates our core claim, that the Environment Oriented simulation displays the expected emergent property. The agents do indeed form flocks, although entirely without direct communication between the agents. The agents essentially know nothing of each other, other than that other agents might exist. All they do is interact with their environment.

In this spatial simulation the agents have no knowledge of their position. The environment is a real participant in the simulation, not merely a repository. It knows the absolute positions of the agents, and is able to populate each agent's neighbourhood with the states of neighbouring agents.

6.2 Flocking in multiple environments

The flocking simulation in section 6.1 has a single environment. However, it is possible for agent behaviour to depend on multiple sensory inputs, for example, visual and auditory.

The experiment described in this section extends the single environment experiment to one supporting multiple environments. Each agent is simultaneously in two environments. In addition to the previous "visual" neighbourhood, each agent shares a second environment with a small number of other agents. Agents in this second "auditory species" environment can perceive each other regardless of proximity.

6.2.1 Claims

The main claims under evaluation here are:

1. Complex systems emergent properties can appear in an environment oriented simulation that uses a number of environments.
2. An individual agent that can perceive more of the set of agents sharing the environments should flock more successfully. That is that a larger proportion of the agents in a simulation should end up flocked.
3. The Environment Oriented simulation platform is readily extensible to a simulation which uses multiple environments

6.2.2 Platform extension

The simulation platform architecture explicitly supports the notion of multiple environments, and the modifications required support specific multiple environments are straightforward. Each agent places its external state in one or more environments, and uses onput from all these environments to determine its future interactions.

For this experiment, the single environment flocking simulation is extended to support a simple version of the multiple environment concept. Each agent is in two environments, a single ProximityEnvironment (for visual state, as before) and a new single SpeciesEnvironment (for auditory state).

6.2.3 Simulation results

The boxplots in figure 3 summarise the results of performing simulations of flocking using the multiple environments. These data are presented in the same form as in figure 2; the earlier results are equivalent to a simulation where the number of agents in the species environment is one.

6.2.4 Discussion

In a single environment, 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 results are more complicated. The agents are continuing to flock, and the number of agents in the species environments is affecting the success of the flocking. At larger proximities, in particular $proximity > 35$, adding more agents to the species environment improves the

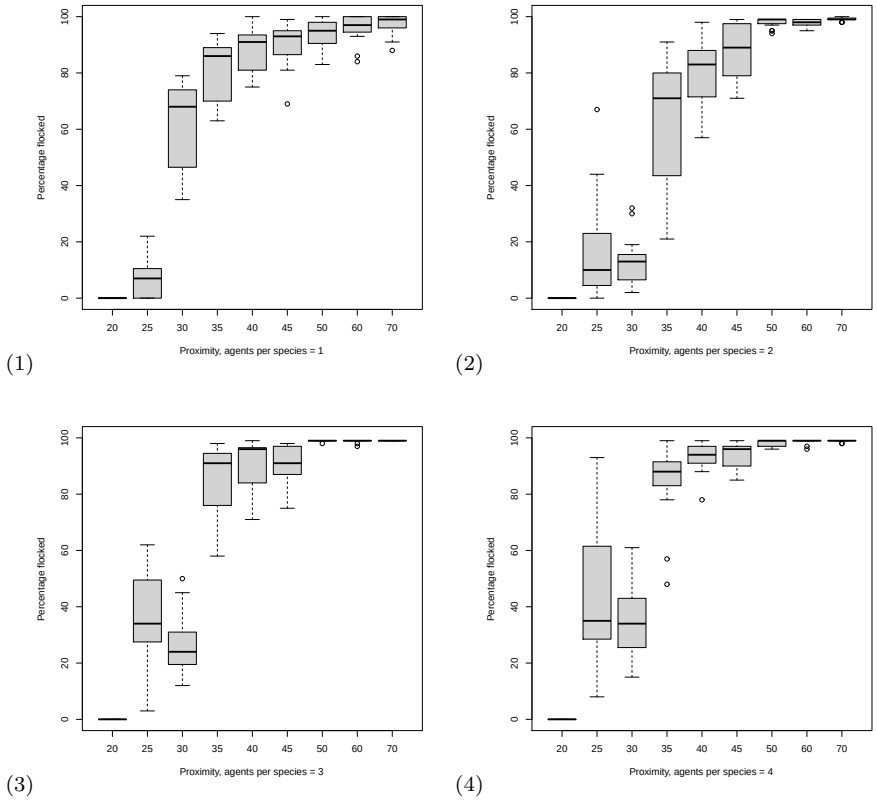


Fig. 3 Box plots showing the results of flocking in multiple environments: there is a single proximity environment; the labels indicate the number of agents in the species environments. Each box shows the percentage of agents that flocked over 2 million simulation steps, over 20 simulation runs. Each 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.

amount of flocking. The multiple environments allow an agent to "see" further than it could previously.

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 distribution is much broader. This is an effect of the random allocation of agents to the species environment. For example, if agents on opposite sides of the non-cyclic "world" are in the same species environment then it is slightly more likely that they will move towards each other, since the members of the species environments can see across the entire world. As these agents move towards each other, they are more likely to approach other agents in their 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 initially close, then even though they move together they never manage to attract any further agents; even four agents of the same species next to each other are not sufficient to be classed as a flock (section 6.1.3).

These experiments in multiple environments include rather more complex interactions between the agents. Yet there is no possibility of deadlock in the simulation, 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 that are serialised.

6.3 Hill climbing

The environments discussed in the previous experiments involve properties only of the agents themselves. Many complex systems include a significant external environment, for example the geography. A simulation with a more complicated external environment is investigated here. The agents' neighbourhoods are extended by an environment that describes the landscape in which the agents move. Each agent has an altitude which is the height of the landscape at the position where the agent finds itself. (The "birds" are now "walking" on a hilly landscape.)

Here the aim is for the agents to climb to the highest point in the landscape. However, the agents cannot directly observe the global landscape, they can observe it only through their immediate local state, and the external state of other agents. This case is designed to be directly analogous to metaheuristic optimisation algorithms where particles swarm over a fitness landscape, searching for the maximum by observing the fitness of other particles.

Here, each agent move towards the highest part of the landscape of which it are aware, given by the altitudes of the agents in its neighbourhood. The local repulsion rule prevents agents getting too close to each other.

6.3.1 Claim

In this experiment we investigate the claim that an environment oriented simulation can be readily extended to include external concepts, representing the physical world outside the agents. Here this concept is a simple "landscape".

6.3.2 Platform extension

The platform is specialised by using a different implementation of the Environment class, LandscapedEnvironment. This uses another class, Landscape, to find the altitude of the landscape at the agent's position. The agents (instances of an new class ClimbingAgent) implement the relevant behaviour given their neighbourhood state information. In these experiments the landscape is the classic "sombrero" function [33].

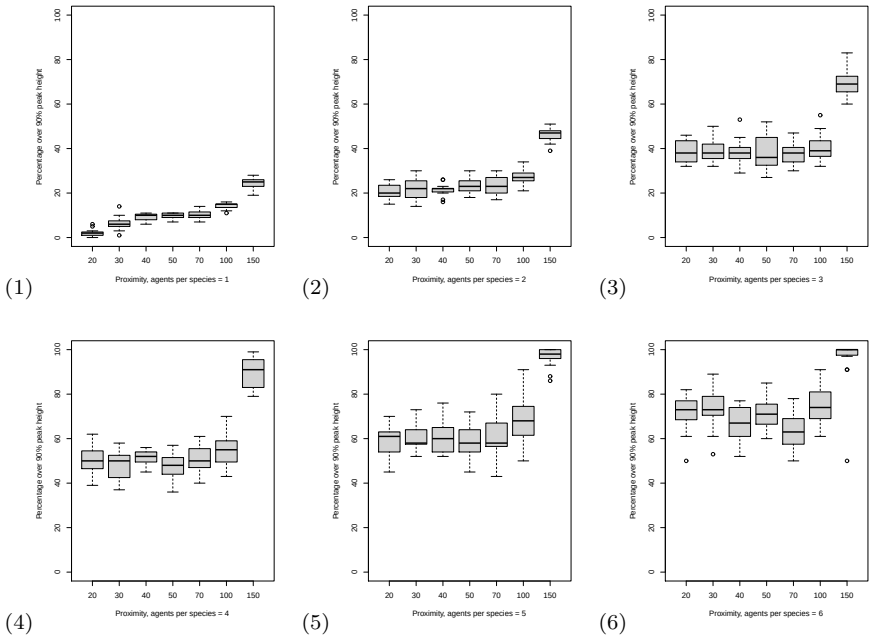


Fig. 4 Box plots showing the results of hill climbing in multiple environments. The labels indicate the number of agents in the species environment. Each box shows the percentage of agents that had reached a height of $> 90\%$ of the height of the central peak, with 20 simulation runs for the data in each box. Each horizontal axis shows the value of *proximity* in the ProximityEnvironment.

The implementation is enriched with a specialised monitoring class, which collects information about how many agents are at an altitude above 90% of the greatest height in a particular landscape.

6.3.3 Simulation results

The box plots in figure 4 summarise the results of the hill climbing simulations. The agents are relatively unsuccessful at finding the central peak when 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. With a larger value of *proximity* each agent is more likely to "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. An agent tends to find a local maximum, the top of one of the surrounding "foothills" of the sombrero landscape.

As the number of agents in the species environments increases then the agents become more successful at finding the central peak. Consider two distant agents that are "connected together" by being in the same species envi-

ronment. Then, as they move towards each other, one might find the central peak, or the start of it. If so, 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 claim 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 Validation Discussion

We have demonstrated some simple simulations built on the Environment Orientation platform. These simulations work with no further attention paid to deadlock. The behaviour of an 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. 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 here is responsible for imposing an overall "environment-centric" view on individual agents' states. For example, the environment is calculating and retaining 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 information in addition to the states of other 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, this need not always be the case (see section 7.1.2).

7 Scaling up

7.1 More complex environments

In addition to the three experiments reported above, we have used the Environment Orientation platform to perform complex system simulations with more complex environments. These further support our claims that the Environment Oriented architecture is flexible and extensible, and supports deadlock-free simulations. We briefly report on those here.

7.1.1 Fuzzy flocking

The experiments reported above take a simplistic approach to fusing the information received from multiple environments. The external states for accessible agents in all environments are placed *verbatim* in the querying agent's neighbourhood. So 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.

In a real world complex system, distance would degrade the resolution at which 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 [28]. However, if a distant agent were perceived in an auditory environment then the detecting agent would be able to determine roughly in which direction the distant agent lay, but its precise distance and velocity would be less easy to determine. The visual environment provides three dimensions of position information and three of velocity information, but the auditory environment provides just a direction.

To address this issue, we have further extended the simulation platform to implement a fuzzy flocking simulation, where the information available to an agent from different sensory modalities is combined, by the environment, using a fuzzy logic approach [17, ch.7].

The fuzzy process provides a mechanism for fusing together the combinations of multiple environments, although at the cost of a more complex implementation. The environment has been tasked with more responsibility, for the fuzzification process, determining what the value is for an observation's membership of a fuzzy set. Despite the more complex implementation, it is still consistent with the original Environment Orientation architecture; agents still construct their neighbourhoods and use a simple output from that neighbourhood to control their behaviour.

7.1.2 Energetic evolution

We have used the Environment Oriented platform to implement a simulation incorporating a more complex environment, and more complex agent behaviours. Agents interact, reproduce with variation (evolve), and die in an environment that includes energy (which agents need to live) and geography. The aim was to investigate the effect of an energy supply on the diversity of evolved creatures [17, ch.8],[18].

In this case, we augmented the simulator design with meta-models of the energy, the world and the agent organisms. Constructing explicit meta-models is useful, as it defines the sort of world that is being simulated. The relationships between these meta-models are explicit and implemented in the simulation model. The implementation follows the model so constructed, fitting into the Environment Oriented simulation platform architecture.

The Environment Orientation architecture provides a simple and elegant approach to implementing simulations of complex systems with a large number of complex agents interacting in a complicated world.

7.2 Concurrency

Deadlock is not an issue for small simulations that can run single-threaded. However, any large scale simulation will have to use a multi-threaded implementation to have any chance of providing the computational power required to support the necessary number of relatively complex agents. The number of threads will be much less than the number of agents in such a simulation. The requirement for concurrency comes from the large number of separate agents sharing the same computational platform.

Once an implementation is multi-threaded deadlock is an inescapable issue. The difficulties in using concurrent programming are well known and a large collection of mechanisms have been developed [13, 29] to manage them. Environment Orientation is another such technique, based on the observations of agent interaction in complex systems, where it can be argued (section 3) that individuals agents interact only with their environment, not 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.

Building Environment Oriented complex systems simulations is feasible, as we have demonstrated; such simulations exhibit the expected emergent properties and their implementation is simplified by the given architecture.

The example Environment Oriented simulations discussed here were all run on a single multi-core processor. So it was reasonable to delegate the concurrency control that is necessary, specifically that around updates of the state repository, to a simple mechanism: the Java `synchronized` keyword. A large scale Environment Oriented simulation would include separate processors. In such a context the underlying state repository would have to be implemented to be

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 [6].

7.3 Caching

Commercial systems address the performance requirements of a distributed system in several ways, some of which are 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 an agent updates its state at each simulation step, and that information is potentially read by many other agents before it is next updated.

As such, this information is ripe for local caching. If the environment server, implemented in each processor, can find the required states in the local cache, then they can be processed rapidly. 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 are needed once there is a high probability that they are going to be needed again.

This apparently simple process is complicated by several issues. See [17, §9.4.1] for a detailed discussion.

8 Conclusions and future work

Environment Orientation is an approach to the construction of computer based complex systems simulations where the agents do not directly communicate with each other, but rather communicate through the medium of some environment. In such a simulation the agents are regarded as placing their relevant state information into the environment, and observing other agents by querying the environment; the environment determines which agents are in the detection neighbourhood. The notion of an agent existing simultaneously in multiple environments with quite separate characteristics is realistic.

We have demonstrated that Environment Oriented simulations of complex systems are feasible, and that certain technical issues, such as the combination of environments, are eased. The software architecture is relatively straightforward, building on well-established notions of transactional software design. Simulations built on this architecture are free from deadlock by design. As such the simulator developer can concentrate on the complex system itself.

Inevitably, though, there are limitations common to highly distributed implementations to do with scaling up (section 7). The simulations performed

so far here have used only a few thousand simple agents, which can work effectively on single-processor multi-core hardware, and where these issues are easier to solve.

There are, though, engineering approaches (section 7) 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 next step is to build such a much larger simulation using Environment Oriented techniques, and using larger-scale multi-processor hardware.

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

References

1. 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 *Artificial Life XI*, pages 17–24. MIT Press, 2008.
2. 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.
3. Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Pearson, 3rd edition, 2005.
4. 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.
5. E. W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1(2):115–138, June 1971.
6. Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. *ACM Trans. Inter. Tech.*, 2(2):115–150, May 2002.
7. Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, Boston, MA, USA, 2002.
8. Eric Freeman, Susanne Hupfer, and Ken Arnold. *JavaSpaces Principles, Patterns and Practice*. Addison-Wesley, 1999.
9. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.
10. David Gelernter. Generative communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, January 1985.
11. J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
12. Jim Gray. The transaction concept: Virtues and limitations. In *Proc. 7th International Conference on Very Large Databases*, pages 144–154. IEEE, 1981.
13. Per Brinch Hansen. *The architecture of concurrent programs*. Prentice-Hall, 1977.
14. Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *The C# Programming Language*. Addison-Wesley, 2nd edition, 2006.
15. C. A. R. Hoare. *Communicating sequential processes*. Prentice Hall, 1985.
16. Jan Rune Holmevik. Compiling SIMULA: A historical study of technological genesis. *IEEE Annals of the History of Computing*, (4):25–37, 1994.
17. Tim Hoverd. *Environment Oriented Simulation*. PhD thesis, University of York, 2011.
18. Tim Hoverd and Susan Stepney. Energy as a driver of diversity in open-ended evolution. In *ECAL 2011*, pages 356–363. MIT Press, 2011.
19. David R. Jefferson. Virtual time. *ACM Trans. Program. Lang. Syst.*, 7(3):404–425, July 1985.

20. 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*, 35:457–472, 2001.
21. J. M. R. Martin and P. H. Welch. A design strategy for deadlock-free concurrent systems. *Transputer Communications*, 3(4):215–232, 1997.
22. Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 2000.
23. Microsoft. SQL Server technical bulletin — How to resolve a deadlock, 2007. http://support.microsoft.com/kb/832524, accessed 1 April 2013.
24. Robin Milner. *The Space and Motion of Communicating Agents*. CUP, 2009.
25. Scott Oaks and Henry Wong. *Java Threads*. O'Reilly, 3rd edition, 2004.
26. 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*, pages 177–192. ACM, 2009.
27. S.F. Railsback and V. Grimm. *Agent-Based and Individual-Based Modeling: A Practical Introduction*. Princeton University Press, 2011.
28. Craig W. Reynolds. Flocks, herds, and schools: A distributed behavioral model. *Computer Graphics*, 21(4):25–34, 1987.
29. Adam T. Sampson. *Process-Oriented Patterns for Concurrent Software Engineering*. PhD thesis, University of Kent, October 2010.
30. 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.
31. Susan Stepney. Embodiment. In Darren Flower and Jon Timmis, editors, *In Silico Immunology*, chapter 12, pages 265–288. Springer, 2007.
32. 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.
33. Wikipedia. Sombrero function. http://en.wikipedia.org/wiki/Sombrero function, accessed on 6 April 2013.