Natural Computing, Springer, April 2012
(DOI: 10.1007/s11047-012-9304-2, ISSN: 1567-7818, © Springer 2012)

To Boldly Go:
an occam-π mission to engineer emergence.

Peter H. Welch · Kurt Wallnau · Adam T. Sampson · Mark Klein

Abstract Future systems will be too complex to design and implement *explicitly*. Instead, we will have to learn to engineer complex behaviours *indirectly*: through the discovery and application of local rules of behaviour, applied to simple process components, from which desired behaviours predictably emerge through dynamic interactions between massive numbers of instances. This paper describes a process-oriented architecture for fine-grained concurrent systems that enables experiments with such indirect engineering. Examples are presented showing the differing complex behaviours that can arise from minor (non-linear) adjustments to low-level parameters, the difficulties in suppressing the emergence of unwanted (bad) behaviour, the unexpected relationships between apparently unrelated physical phenomena (shown up by their separate emergence from the same primordial process swamp) and the ability to explore and engineer completely new physics (such as force fields) by their emergence from low-level process interactions whose mechanisms can only be imagined, but not built, at the current time.

Keywords complex systems · emergent behaviour · emergent relations · emergent discovery · process orientation · mobile processes · occam-pi

1 Introduction

The thesis underlying this paper is that there is a growing need for systems with *emergent* behaviour and that this will make rather different demands on our skills as software engineers. The aims for this paper are to make the case for this thesis and to explore ways to meet those demands.

Some new approaches are needed since behaviours required for future systems will be so complex that they will be beyond our capabilities for *direct* engineering. One reason for this is that the only way to program them will be *not* to program them, but to let them emerge in a way that has been *researched* (through simulations in a designed infrastructure that *can* be directly engineered), from which a *theory* has been discovered (quantified and supported by experiment), and through which they can be *safely controlled*. The emerging phenomena will have no representation in any directly engineered software entity – hence, the need for investigation and for *scientific method*.

The paper develops simple and familiar case studies (*boids* and *nanobots*) to explore these ideas. These studies do not reflect, as yet, real applications but they allow focus on essential difficulties, dangers and opportunities. The observed system behaviours are certainly emergent, no aspect of them being explicitly programmed. For example, there are no programming entities corresponding to flocks, squabbles, streams, shock waves, feeding frenzies, feeding caution, turbulence, maze solving, free traffic

This work is part sponsored by the CoSMoS project (EPSRC grants EP/E053505/1 and EP/E049419/1) http://www.cosmos-research.org/. The Software Engineering Institute, Carnegie Mellon University, also hosted and sponsored a sabbatical visit from one of the authors (PHW) at the start of this work.

P.H. Welch
School of Computing, University of Kent, Canterbury, UK
E-mail: p.h.welch@kent.ac.uk

K. Wallnau and M. Klein
Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA., USA
E-mail: kcw@sei.cmu.edu and mk@sei.cmu.edu

A.T. Sampson
Institute of Arts, Media and Computer Games, University of Abertay Dundee, UK
E-mail: ats@offog.org

flow, or force fields – yet they are present in the resulting systems. The paper speculates on possible applications for some of these behaviours that may now be considered *far fetched* – but only time will tell.

The approach presented in this paper relies on robust *designed* engineering of environmental frameworks that can be specialised with particular (designed) properties, and stirring into such environments a mass of specialised agents that can move through them under their own (still designed) low-level rules of behaviour. The resulting massively concurrent and non-deterministic mixture gives rise to behaviours that are hard to predict (i.e. are *not* designed), but whose properties can be researched, tuned and harnessed (using scientific method) – ultimately for real application. The approach seems to reflect natural processes whereby complex systems emerge in the (massively parallel) real world. Nature sets a standard that engineering would be wise to follow.

1.1 Rational design

Computer systems have traditionally been put together through a process of *rational design*: requirements are gathered, specifications for behaviour made, and designs for implementation developed, implemented and tested (though not always in that order and, usually, with many iterations). The process is meant to be *explainable*: for the system as a whole and for each component in the system, it should be possible to deduce, at an acceptable level of formality, how each implementation meets its specification. Even as a long-lived system *"evolves"* through maintenance (as requirements and technology change), each version should remain analysable at all levels with all behaviours understandable and explained. This evolution is not *natural* though – it is *planned*.

1.2 Emergent design

We are interested in *emergent design*. The problem with rational design is that we are simply not clever enough to build systems with the level of complexity (and, thus, application) our ambitions have begun to chase. With emergent design, system behaviours appear that have no correspondence with any programmed entity in their implementation – and these behaviours can be complex, interesting and very useful (see section 2). Making rational explanation of such effects is difficult – they have no obvious causes. Yet, to use them, we must find ways to predict and control them. It's programming, Jim, but not as we know it.

Emergent design builds upon rational design. Emergent behaviour arises from subtle and imprecisely planned interactions between low-level components whose designs and implementations *are* rationally planned and fully understood. We may need to impose boundaries on the areas and scales of interaction and we need rational design and analysis to be confident that such impositions will be effective. Interactions will take place within some modelled environment and we need rational design to ensure that the basic operations are safe (e.g. they do not cause deadlock or livelock), fair (e.g. all components progress together) and unsurprising (e.g. free from race hazards).

Only with such assurance, can we boldly go and explore the strange new worlds that emergence lets us construct – to *experiment* with adding (or subtracting) behaviours in the low-level components and with changing their rules for interaction (between themselves and their environment). Very likely, we will find *non-linearities*, whereby small changes in low-level component behaviour lead to big changes in high-level emergent behaviour, and we will not understand why. Even so, such experiments may let us develop theories for these worlds that will be accurate enough to predict their modes (and extremes) of operation and, so, control safe application. This is scientific method. This is emergent engineering.

1.3 Concurrency

Our approach further relies on *concurrent* engineering (section 3.1). The lack of central control implied by the descriptions above means that, *for simplicity*, concurrency will be a fundamental part of the solution. Our computational architecture, patterns and systems are written in occam-π [52,7,58,53,54,31,59,32,40,8], whose processes are sufficiently lightweight to enable a sufficiently large number (e.g. millions) to be run interactively for real-time experiments on emergent behaviour. Its semantics are based on process algebra (CSP [22,23,33,44,34] and the π-calculus [28]) and they are compositional (*what-you-see-is-what-you-get*), so that we can apply simple reasoning to verify the correctness of our low-level components (explicitly built) and their safe patterns of interaction [37] (explicitly programmed).

1.4 Challenges

Different varieties of behaviour may emerge within a single application, with interactions between them provoking ever-richer patterns – almost social systems. Sections 3.3 and 4.1 illustrate with a study based on Reynolds' boids [30]: emergent behaviours include flocking (of course), directional migration (with waves), fear and panic (of hawks), orbiting (points of interest), feeding frenzy (when in a large enough flock), turbulent flow and maze solving.

With this kind of engineering, a new problem shows up: the suppression of the emergence of undesired behaviours (section 4.2.2). The panic reaction within a flock to the sudden appearance of a hawk is a case in point. With our present rules, the flock loses cohesion and scatters too quickly, making individuals more vulnerable. What are the rules that will make the flock turn almost-as-one and maintain most of its cohesion? There are only the boids to which these rules may apply (there being, of course, no design or programming entity corresponding to a flock). More importantly, how do we set about finding such rules in the first place?

Something else that can emerge from emergent engineering are unexpected relationships between different domains of study (e.g. swarms, streams, plasma dynamics and maze solving). These relationships may not have been noticed before, as the low-level agents generating the emergent behaviours have quite different physics and scales. There is also the potential to discover completely new domains of interest from such studies. These issues are considered in sections 4.2.3 and 4.2.5.

1.5 Structure of this paper

This section has summarised the aims of the paper and rationale behind the reported work, including background ideas and motivation (rational versus emergent design), background technology (concurrency) and open challenges (for controlling emergence).

Section 2 gives motivating examples under current study elsewhere: social and biological systems, autonomous systems and the need for foundation technologies and theories that give *Verification and Validation* a chance.

Section 3 presents the enabling architecture and patterns used in the paper (process-orientation, occam-π, space and time simulation models, mobile agents) and introduces the running case study (based on Reynolds' *boids*).

Section 4 explores the many forms of emergent behaviour arising in the case study, the dangers and opportunities afforded by discontinuities between low-level designed behaviour and high-level emergent ones, the relationship of emergent engineering with scientific method, open questions on the control of emergent properties (e.g. flock cohesion in attack), unexpected relations between apparently different phenomena (through common generators of their emergent properties), a note on obstacle avoidance (simplicity, behaviour and computational lightness) and new physics (based on *nanobots* and speculative).

Section 5 summarises and draws some conclusions.

2 Examples and Assurance of Emergence

Emergence is not a new idea. From more than two centuries ago, Adam Smith's *Invisible Hand* [46] is sometimes cited as a paragon of emergence, whereby an economic agent:

> *"neither intends to promote the public interest, nor knows how much he is promoting it ... he intends only his own gain, and he is in this, as in many other cases, led by an invisible hand to promote an end which was no part of his intention."*

2.1 Socially inspired emergence

Contemporary economists have developed game-theory explanations of Smith's invisible hand and a range of techniques, referred to as *Mechanism Design*, that define economic institutions via games (auctions, markets, elections, etc.) from which desired, optimal and predictable effects *emerge at equilibrium* as a consequence of agents exhibiting individually rational (i.e. selfish) behaviour. *Computational* Mechanism Design (CMD) uses computers to compute agent strategies or mechanism outcome. The Google keyword auction [19] is one well-known computational mechanism. CMD remains a significant area of active research – and not only for economics and e-commerce [35,36]. In our own work, we have been exploring CMD as a way of allocating scarce computational resources, in particular scarce network bandwidth in mobile, ad hoc networks [25].

It is worth noting that not all theories that purport to account for the emergent social consequences of individual human behaviour are exclusively based in game theory and the narrow view of rational self–interest. The role of cultural norms, kinship and the social networks they induce also has determinate influence on emergent consequences of the behaviour of many individuals. Indeed, socio-analytic foundations usable by systems designers are emerging [18].

2.2 Biologically inspired emergence

Natural and biological systems are also frequently cited as paragons of emergence, often with particular emphasis on *self-organisation* wherein large-scale patterns of apparently coordinated behaviour (synchronisation of firefly signals, schooling, flocking, swarming), structures (termite mounds, bird nests), and optimisation (ants finding shortest paths to food) emerge from uncoordinated interactions of large numbers of agents having simple behaviour. Bio-inspired approaches to search and optimisation, often referred to as *meta-heuristics*, have already become mainstream: for example ant colony optimisation [17] and genetic search algorithms [9].

Bio-inspired approaches are increasingly commonplace in autonomous robot systems, but cannot be said to have obtained mainstream status. A representative example (among many research prototypes) was developed by Nagpal et al. [61] to assemble non-trivial structures from available environmental material, using a "local-to-global" programming technique. The technique combines simple *local* behaviour rules, executed by a collection of simple mobile robots with a form of digital *stigmergy*, to construct alphabet-shaped structures from building blocks randomly placed in the environment. Techniques to achieve coordinated *swarming* or *flocking* behaviour of Unmanned Air Vehicles (UAVs) and mobile robots [63] have also been reported [42].

2.3 Assurance of engineered emergence

The work reported here is inspired, at least in part, by the potential for combining socio- and bio-inspired emergence to achieve scale and robustness in mobile systems that combine human and autonomous robot behaviour. To illustrate, UAV swarms might be used to deploy self-managing, self-repairing airborne wireless networks. Using appropriate local-to-global swarm management techniques, the location and shape of the airborne swarm network can be controlled to produce effective qualities of network service (average latency, maximum latency, etc.) that are based on changing user needs, which are themselves reported and managed using economic mechanisms (such as online auctions).

A natural challenge for emergent design will be establishing an objective basis for justifiable confidence that the system-level behaviours we obtain *indirectly* (i.e. as emergent effects) will be the ones we want. The US Air Force Chief Scientist reports [15]:

> *Use of autonomous systems will require developing new methods to establish "certifiable trust in autonomy" through Verification and Validation (V&V) of the near-infinite state systems that result from high levels of adaptability; the lack of suitable V&V methods today prevents all but relatively low levels of autonomy from being certified for use.*

Recent work by Stonedahl and Wilensky [48] shows how the effectiveness of different forms of flocking behaviour can be expressed as *fitness functions* at design time, and how search-based optimisation, including genetic search, can be used to maximise fit at design time. However, only a relatively few features of fit are defined. It remains to be determined what kinds of fitness qualities can be specified; how these qualities can be measured in operational systems; and, how to determine a quantitative measure of confidence that can serve as an indicator that such qualities will be obtained and sustained in the variety of systems in which such swarms will be deployed.

3 Enabling Architecture and Patterns for Emergence

This section is about the rational design (section 1.1) of low-level components and the infrastructure that holds them together. Concurrency is used for simplicity, verification and multicore performance.

A model of space (*servers*) and time (*barriers*) is described through which agents (*mobile processes*) – in large numbers – can roam, modify and interact. An architecture for agents, with generic components for navigating their world and specialised components to govern individual behaviour, is given. Agents have a sense of time, that keeps them in step with each other, and locality, that means they can only interact when close enough to see each other.

The behaviour of individual agent processes, consistency of the space-time model and global safety of all interactions (e.g. freedom from deadlock, livelock, starvation and race hazard) can be *planned* and *verified* with rational argument against a semantic theory that is well founded and understood.

This lays the foundation for the engineering of *emergent* behaviours that are governed by theories that we do not understand and will need to research – the subject of section 4.

3.1 Process orientation and mass parallelism

We work with a natural model of concurrency, formalised by the process algebras of CSP and the π-calculus and built into the occam-π programming language. No knowledge of occam-π syntax and semantics is assumed for the reader of this paper. The semantic ideas underlying occam-π concurrency are presented in section 3.1.1, including the diagram language (figure 1) for expressing *process-oriented* design (used in figures 2, 3, 6 and 7). Only one 6-line fragment of occam-π syntax and structure is shown in this paper (section 3.2.2, to support the description of the *phased barrier* pattern used by mobile agents). Full open source codes for the case studies presented in this paper (*boids*, section 3.3, and *nanobots*, section 4.2.5) are downloadable from [60].

occam-π ideas (of simplicity, expressiveness, formalised reasoning and parallel safety) together with its performance (on multicore and instruction-reordering microprocessors – sections 3.1.3 and 3.3.5) underlie the massively parallel simulation architecture described in this section and facilitate the higher-level experiments on the engineering of emergent behaviour systems (section 4).

For further information, a condensed summary of the core occam-π syntax and semantics is given in sections 2 and 3 of [58]; much greater detail can be found in [52,53,51]. A different case-study (modelling the formation of blood clots) is reported in [32] and contains considerable lengths of occam-π code (and pseudo-code); however, its architecture is an earlier and more memory-intensive version of the one reported here.

The design and implementation of the reported case studies could, of course, be re-worked in a mainstream language with a library providing the occam-π/CSP/π-calculus model of concurrency (such as JCSP for Java [57,56,55], C++CSP for C++ [14,12] or CHP for Haskell [11,13]). However, extra care would be needed to avoid common concurrency errors – against which occam-π protects – and a loss in performance, or scale, accepted.

The open source KRoC compiler, libraries and run-time system for occam-π can be downloaded from [39]. The KRoC home page is at [8].

3.1.1 Processes and networks

A *process* encapsulates data and logic (like an *object*); but its logic is implemented by an internal thread (or threads) of control. A process is *alive* and solely responsible for its internal data. A process cannot see or change the data in another process. Thus, process-orientation has no need for *locking* mechanisms to protect shared data and there can be no *data race hazards.* A process variable changes its value if and only if its process makes that change – nothing changes unexpectedly, which gives a simple and intuitive semantics.

Processes are connected into *networks* by *channels* and/or *barriers.* These are the only means by which processes interact. A network is itself a process – so structured concurrent design (*networks within networks*) is naturally afforded. Channels and barriers are *synchronisation* mechanisms in which all relevant processes must engage. A process may *refuse* to synchronise (e.g. receive a message) simply by not being programmed to synchronise. This gives easy and powerful control over how a process manages its interactions (e.g. if it is not in a state to service some request, it simply refuses it – there is no need for complex logic such as *condition variables* [21] to resolve this).

Figure 1 shows the basic diagram language for process-oriented design. It's not large and is sufficient for all the case studies in this paper (and most elsewhere).

Figure 1(a) shows part of a network, with three processes connected by *point-to-point* channels to each other and to the rest of the network. A channel communication is between one sender and one receiver and is synchronised: the sender will block until the receiver is ready and vice-versa. When we

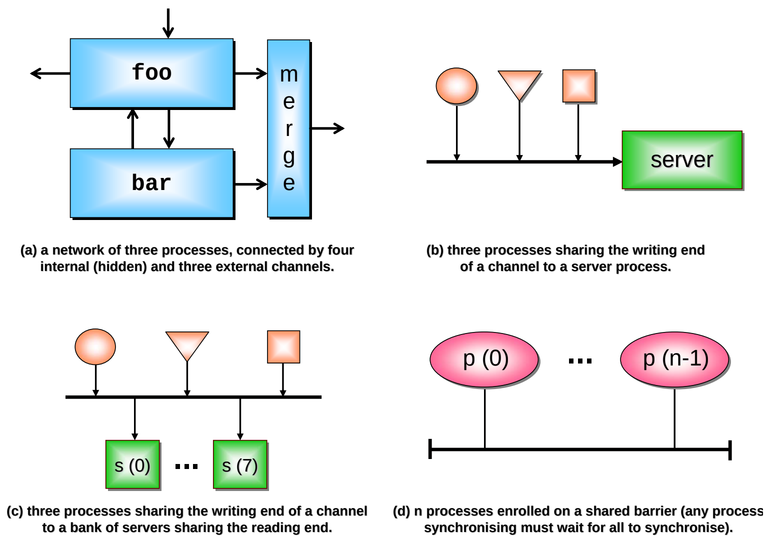

foo
bar
merge
server
s (0)
s (7)
p (0)
p (n-1)
(a) a network of three processes, connected by four internal (hidden) and three external channels.
(b) three processes sharing the writing end of a channel to a server process.
(c) three processes sharing the writing end of a channel to a bank of servers sharing the reading end.
(d) n processes enrolled on a shared barrier (any process synchronising must wait for all to synchronise).

Fig. 1 Process oriented design: components and connectors (*from [58]*). There are no rules for the shapes of component icons (*processes*) – whatever is meaningful to the designer is allowed.

need, buffered channels can be efficiently built just by inserting buffer processes – and, of course, we have full control over the specification of such buffering (e.g. its size, whether it blocks when full or grows or throws messages away). If helpful for this and later designs, the figure 1(a) subsystem can be abstracted into a single process, with three external channels (one input and two output) and implemented by the shown sub-network.

Figure 1(b) shows a simple *server* network. The three client processes *share* the writing end of the channel to the server, but *interleave* in its use (they form a queue to gain exclusive access). Not shown here is that channels (e.g. *request* and *answer* that operate in opposite directions) can be grouped together into *bundles* and that the ends of such bundles may also be shared. A client process gaining access to a bundle-end retains access for as long as it likes, so that a secure two-way exchange can be made. Of course, such exclusive access should be as brief as possible!

The reading end of a channel may also be shared. Figure 1(c) shows a more dynamic network, where (needy) clients queue up to seek service and (idle) servers queue up to provide. For this example, a client must not care which server it gets and a server must not care which client. The channel may also be a bundle of channels, so that client-server business can be two-way and they can identify themselves if they so choose.

Figure 1(d) shows an array of processes (which need not all be of the same type as shown here) synchronising on a barrier. This is a multi-process event: if one process offers to synchronise (i.e. tries to get over the barrier), it must wait for *all* processes connected to the barrier to do the same. The last process to synchronise releases all the others.

3.1.2 Hardware orientation

The iconography in figure 1 reflects hardware design: *chips* and *wires*. The physical constraints implied by hardware mechanics reflect needed semantic rules underlying process autonomy (layered construction, localisation and inviolability of state, interaction only if explicitly connected). Software has no need for such constraints: we could let threads of logic quietly interfere with and depend upon each other's state – just because we can! Even though we may try to retrofit controls (e.g. *locks*) to avoid the worst consequences, the freedom afforded means mistakes will be made and we exploit it at our peril.

The hardware discipline that process-oriented design imposes on software has important benefits for simplicity and performance. Engineers – whatever their field – have long experience of network (or *circuit*) diagrams, with intuition for their semantics long established. The underlying formal semantics of CSP and π-calculus align with that intuition so that we get no surprises, at least at low-level, from our concurrency. An example of this is *determinism*, which is a default property of the CSP parallel operator. *Non*-determinism has to be introduced *explicitly* as and when needed. This gives stability to concurrent design that helps keep things simple. Performance derives from this simplicity: most synchronisations can be managed by simple handshakes or shared counts (either by hardware or software) and can be made very fast.

3.1.3 Multicore scheduling and performance

The occam-π run-time [31] needs only eight machine words (maximum) to manage process state and performs most synchronisations in the order of tens of nano-seconds. It uses *lock-free* and *wait-free* logic, with minimum use of *atomics*, to schedule its processes across as many cores as it can find (batching them according to their dynamic behaviour to get best use from memory cache and *work-stealing* batches as necessary). These overheads are sufficiently low that very large numbers of processes can be efficiently managed on conventional platforms (e.g. PCs and clusters), which is important for the studies in this paper. The large numbers also mean that we do not worry unduly about processes becoming blocked (e.g. because the process to which they are sending are not ready) – there will always be other runnable processes to which their host cores can (very quickly) switch. However, we do have to guard against *deadlock*, *livelock* and *starvation* issues – but with the rational use of design patterns (for which we have theorems proving the absence of such dangers), these are usually not hard to solve [37].

3.1.4 Process mobility and network dynamics

Finally, we note the π-calculus extensions to this concurrency model. Processes, channels and barriers may be constructed dynamically, with channel-ends and barriers that can be *moved* through other channels. This lets us change network topologies at run-time, with individual processes gaining and releasing connections. For the emergent behaviours we wish to engineer, the *process mobility* enabled by this is crucially important.

3.2 CoSMoS space and time

This is one of the patterns developed for modelling complex systems as part of the (UK EPSRC funded) CoSMoS project [47,41,3,38,29,32]. It addresses the modelling of space and time for an environment in which mobile agents can roam and interact, and from which interesting behaviours emerge.

3.2.1 Space

Space is modelled as a network of server processes, each one responsible for a specific area and with the union of these areas making up the whole space. Local connections between servers define overall geometry. The geometry may be multi-dimensional (at least two), Euclidean (at least locally) and static. However, a dynamic space model – for example, one with the transient growth and decay of *worm-holes* connecting distant regions – can easily be built.

A server holds information about the area for which it is responsible: for example the concentration of certain chemicals (relevant to the model), wind/fluid vectors (that impact on visiting agent processes), the presence of visiting agents (their types, positions, vectors, sizes, levels of alarm etc.), and ways to contact those agents (channel bundle ends). Each server waits on the *request* channel within a single service bundle and responds on its *reply* channel. Between a request and its reply, only local and finite computations are performed – i.e. no channel or barrier synchronisations (that might provoke deadlock). At all other times, the servers are idle (unless worm-holes are forming or space topology is changing in some other way).

Each server also holds the client-ends of the service bundles for its neighbouring servers: this is what defines local and, ultimately, global geometry. However, servers do *not* make service calls on each other – that would lead to deadlock (as soon as two servers became committed to call each other). The client-ends of service bundles are *shared* for use only by visiting agents. Servers only hold them to pass on to their visitors (which is how those agents *move*).

Figure 2 shows a two-dimensional world implemented by a *Matrix* of servers. The thick arrows pointing down at the servers are the service bundles: one for each server. The client-ends ends of these bundles (i.e. arrow tails) are shared by visiting agents, represented by the spheres hovering over the servers and attached to the bundles (only five are shown). In this geometry, servers have four neighbours (left, right, up and down in the plane). To reduce clutter in the diagram, the holding of neighbouring bundles by the servers (which defines the geometry) is also shown only for five of them.

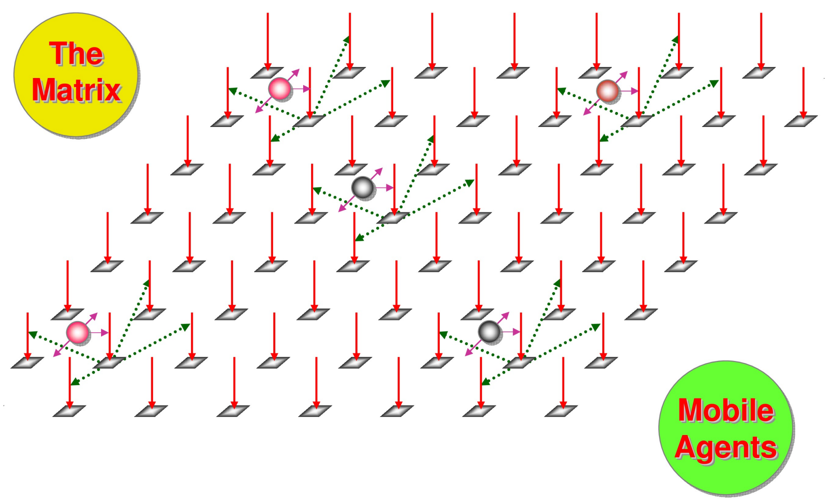


Fig. 2 The matrix and some agents

Depending on the nature of the region of space being modelled, servers may allow zero, one or many visiting agents at the same time. Figure 3 shows *one* of the Matrix servers holding the client-ends of the channel-bundles to its four neighbours (and *their* holding of the client-end of *its* service bundle). Several agents are shown on each server. These agents have (currently) *free* channel connections that they may choose to deposit in their current servers (for other agents to find and, so, make contact).

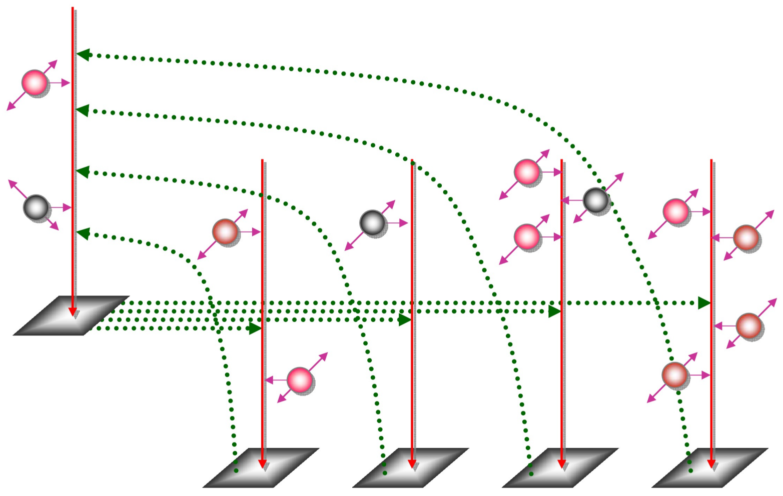


Fig. 3 Neighbourhood topology and social networking

A visiting agent registers its presence at a server and observes its neighbourhood by making requests through the service bundle. A bundle can be declared to allow many varieties of request and reply – for example:

- *I'm new here; here's my id, type, position, vector, size and state of panic. Please send me the client-ends of the service bundles for your neighbouring servers.*
- *I just moved within your space; here's my id, new position and new vector.*
- *I just got spooked; here's my id and new panic level.*
- *I just spilt 10 million barrels of oil at this location in your space – sorry.*
- *I'm happy to be contacted by any agent with size bigger than 42; here's a channel-bundle-end you can lend to such an agent (I'm monitoring the other end).*
- *I just left your space; here's my id.*

Note that, as an agent arrives at a server, it asks for connections to that server's neighbours. This allows it to inspect the environment in its neighbouring, as well as local, areas. To move to a neighbouring area, all an agent has to do is tell its current server it is leaving, replace its holding of the current server bundle with the one for its target area, and report to the new server (obtaining connections to its new neighbourhood).

The system so far described is a pure *client-server* system. The agents are pure clients making calls on the space servers. There is only one level of client-server relations and, therefore, no cycles of those relationships. Therefore, we can deduce that the system is free from deadlock, livelock and process starvation (see [27]).

3.2.2 Time

There can be many varieties of agent and any number of them, subject to memory constraints. Those numbers can vary at run-time. Agents must be controlled so that their activities play out at the same speed. Some agents must not be able to get ahead of the others – they must always stay in the same time frame. This is easy to achieve by making all agents synchronise on a barrier *at the start* of each cycle of their lives. Now, they will all grow old together.

In fact, we have another reason for controlling their execution. We need to control when change happens in the environment. If agents are free to change things at any time in their life cycles, what they observe and feel from their environment (and on which decisions about change are made) will be scheduling dependent. This introduces *server race hazards* that invalidate the accuracy of the system (even though low-level *data races* are prevented by the occam-π concurrency model used).

Fortunately, this is easy to eliminate. We divide agent execution cycles into two *phases* controlled by barriers. Here is generic occam-π *pseudo-code*:

```
WHILE alive
  SEQ
    SYNC tick
    ...  observe local neighbourhood
    SYNC tock
    ...  change local neighbourhood
```

At the start of each cycle, each agent waits for all of them to reach that point (by synchronising on the `tick` barrier). Then, the agents observe their neighbourhood (by interrogating their local and neighbouring servers, whose service bundle client-ends they hold). During this *observation phase*, all see a consistent view. They decide what effect this will have on them and what effect they will make on their neighbourhood, including how they will move.

The agents now wait for all to finish their observations (by synchronising on the `tock` barrier). Then, they enter their *change phase*, where they update their local servers and change their position (which may necessitate moving to a neighbouring server). If there is any contention between agents (e.g. to collect limited resources or move to a restricted area), this can be resolved *locally.* Additional barrier-controlled phases may be needed.

The barriers introduce synchronisations to the system that are distinct from the client-server synchronisations we had earlier – we need to reconsider the safety argument made at the end of (the previous) section 3.2.1.

Within each phase, agents and servers interact as a pure client-server system (with no cycles). Therefore, they will not deadlock or livelock during a phase and all will make progress. We just need to check that each phase will terminate (ignoring the client-server synchronisations, since we know they will complete). There will only be a finite number of things each agent has to do, so this is usually trivial.

We can now view each phase as pure computation that always terminates and ignore the servers and the agent interactions with them. We are left with just the agent processes that cycle through the same sequence of synchronisation on the same set of barriers. Clearly, there is no deadlock, livelock or starvation potential here. Therefore, we may conclude the same for the system as a whole – we have a *safe* combination of the *phased-barrier* and *client-server* concurrency patterns. **Q.E.D.**

The barriers, of course, also maintain simulated time within the system. In fact, only *one* barrier is really needed, so long as we make sure that all agents execute their phases in the same order. There are cache benefits in hitting the same barrier, so only one is used in the case study presented in section 3.3.3. Note that the servers have no need to engage with the barrier (or barriers) and must not be enrolled.

One further note: useful agents to include in the system are those that *visualise* what is happening. The servers may be partitioned into groups and a *visualiser* started for each group, connected permanently to each server in its group. During observation phases (not necessarily every time), it can gather visualisation data from each of its servers and render an image for each associated region. These can be combined and presented through some image technology (or saved for later analysis) during the non-observation phases.

3.3 Reynolds' boids (and some hoiks)

Our case study extends Reynolds' system of *boids* [30], a classical example of emergent behaviour (*flocking*). Individual boids follow three simple rules:

– *try to avoid collisions;*
– *try to match your velocity vector with those of any other boids you can see;*
– *try to move towards the centre of mass of any other boids you can see.*

Note that these rules depend only on information that is *local* to each boid.

3.3.1 Implementation: relative space coordinates

The space-time model from section 3.2 is used. We stay with two dimensions, but the space servers are connected to all eight immediate neighbours (since the boids may be looking towards the corners of their current region of space). For the convenience of having a closed system from which the boids cannot escape, space is wrapped around, top and bottom, to form a toroidal world.

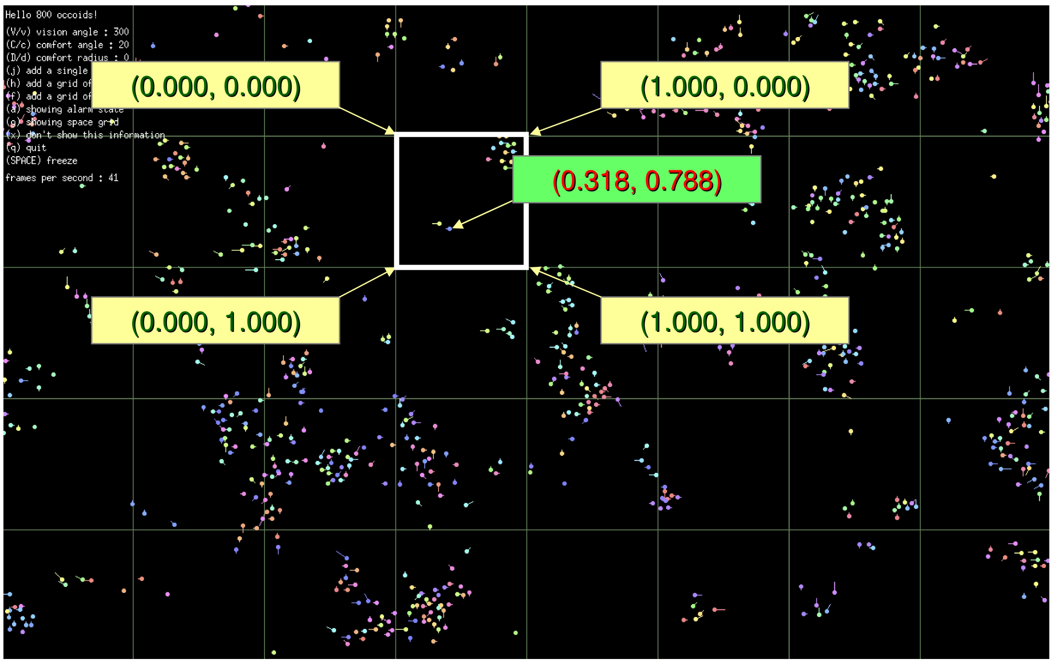

```
Hello 800 occoids!
(V/v) vision angle : 300
(C/c) comfort angle : 20
(D/d) comfort radius : 0
(j) add a single
(h) add a grid of
(f) add a grid of
(a) showing alarm state
(g) showing space grid
(x) don't show this information
(q) quit
(SPACE) freeze
frames per second : 41
```

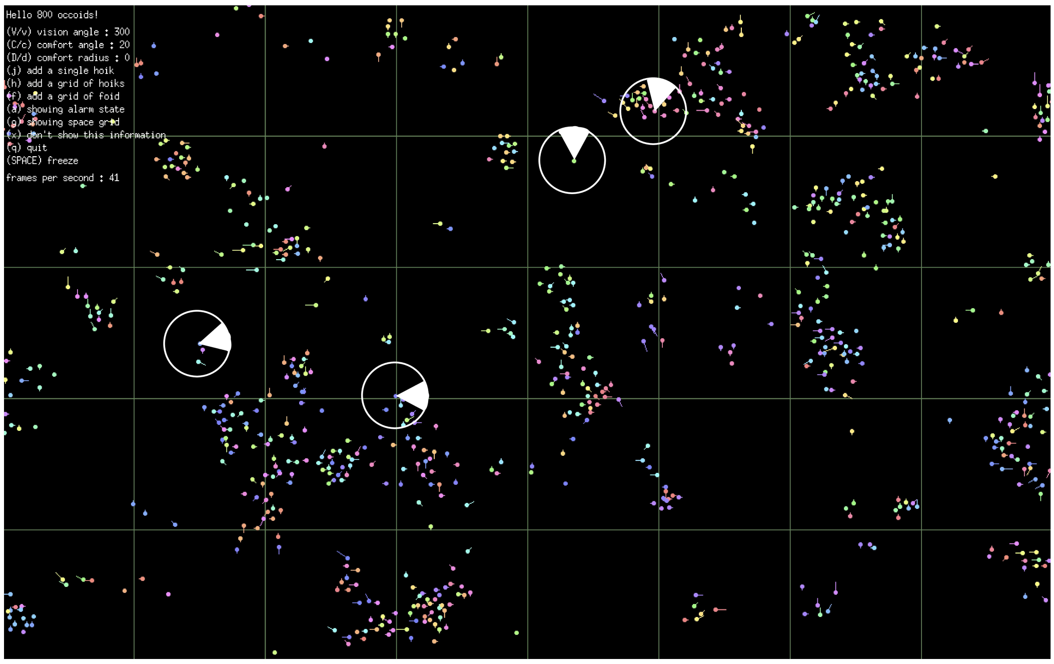
Fig. 4 Space server coordinates.

Figure 4 shows a toroidal world, divided into 40 square regions (an 8-by-5 grid, one square of which is outlined). Boids are visualised by small coloured discs with *tails* indicating their motion vectors (tail orientation is the boid vector angle and tail length is proportional to vector size). Region coordinates are floating-point (for accuracy) and range between 0.0 and 1.0. Like real birds, boids have no knowledge of their *absolute* location – they know only their *relative* position with respect to the region of space managed by their local server (which also does not know where it is). The figure shows the coordinates of one of the boids in the outlined region.

Boid state consists only of its position (relative to its current region), vector, colour and (section 4.2.2) *state of alarm*. It registers this information with the managing space server when it enters a new region and maintains this for its duration in that region.

3.3.2 Implementation: boid vision

Boids can see around them to a maximum range that must be less than 1.0. This limit is imposed so that boids only need to look over to *immediate* neighbour servers, if they are close to boundaries of their current region. We also limit their vision to a *vision angle*, which implies a blind area behind them.



```
Hello 800 occoids!
(V/v) vision angle : 300
(C/c) comfort angle : 20
(D/d) comfort radius : 0
(j) add a single hoik
(h) add a grid of hoiks
(f) add a grid of foid
(a) showing alarm state
(g) showing space grid
(x) don't show this information
(q) quit
(SPACE) freeze
frames per second : 41
```

Fig. 5 Boid vision

Figure 5 shows the areas of vision for four of the boids in the image. The white circles represent their maximum vision range and the white cones their blind area. In this snapshot, the boids have a vision angle of 300 degrees, leaving a 60 degree blind area.

3.3.3 Implementation: generic mobile agent architecture

Concurrent engineering, of course, is a natural way to design and implement the low-level rules governing the behaviour of the mobile agents. Our `occam-`π *boid* processes are christened `occoid`s.

The logic for managing movement across the space servers does not depend on individual behaviour and we can make this generic. It is abstracted into the `pilot` component of the `occoid` process (figure 6).

The `pilot` holds the bundle-ends to the agent's current server, all its neighbouring servers and the barrier for synchronising with all the other agents. The channel-bundles are shown by the 9 (1 local + 8 neighbour) double-headed arrows extending outside the `occoid`. The `pilot` also operates the barrier, which is the bar-shaped icon on top of the `occoid`.

The `pilot` extracts a copy of the database from its current space server and passes this back to its `filter.vision` process. Noting its position and orientation vector, it extracts database copies from neighbouring servers within its range of vision, adjusts agent coordinates within each database by adding or subtracting 1.0 (to align with the local coordinate system), and streams them to its `filter.vision`. The ordering in this stream does not matter.

What eventually feeds back to the `pilot` is a computed force vector representing the integration of all forces on the *boid* determined by its low-level rules of behaviour (section 3.3). Applying this force to its current position and vector, the new position and vector of the `occoid` is computed. If the `occoid` is still in the region of its current server, the server is updated with the new data. If its position has a coordinate outside the bounds [0.0, 1.0), the `occoid` has moved into a neighbouring region; the `pilot` informs its current server that it has left and registers with the appropriate neighbouring server, adjusting its position coordinates to be relative to the new region.

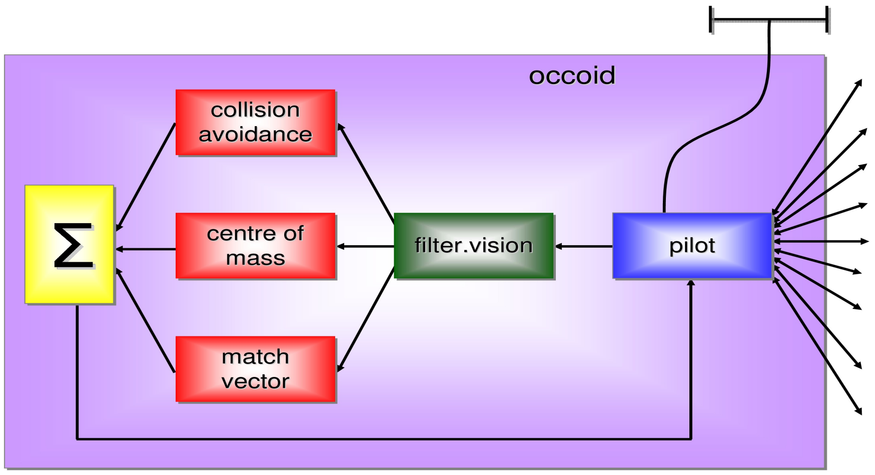


Fig. 6 Natural occoid architecture

The `filter.vision` process just marks each agent record in the passing databases to indicate whether it is within the vision field of this `occoid`. In figure 6, `filter.vision` broadcasts these databases to processes that compute forces according to the boid low-level rules (*avoid collisions*, *edge towards centre of mass*, *match vectors*) – one process for each rule. The rules are independent, so it is natural to process them concurrently (which a real bird brain may well do). Those forces are forwarded to an integrator process that computes a weighted sum, passing the result back to the `pilot`.

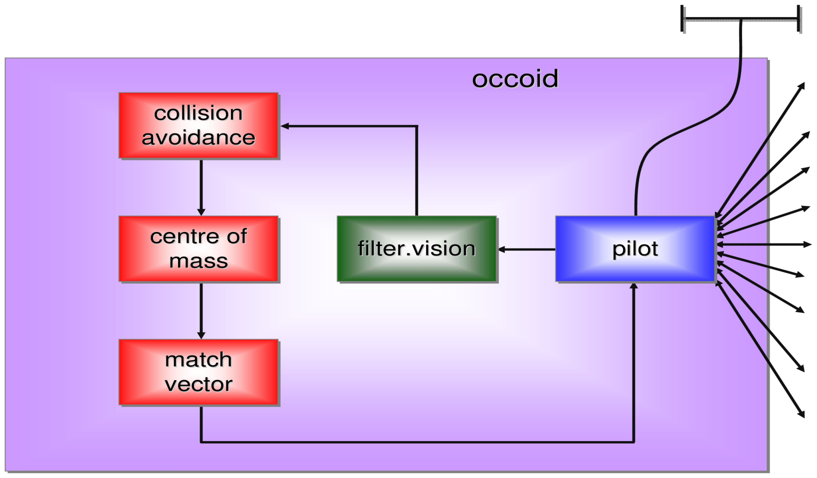


Fig. 7 Optimised occoid architecture

Figure 7 optimises this architecture using a *pipeline pattern* [37] for the same rule processes, extended slightly to add their contributions to the weighted sum (which is flushed through the pipeline following the complete set of server databases for the current observation phase). The order of processes within this pipeline does not matter, since adding commutes. It optimises because a pipeline has lower overheads than a *broadcast-integrate* network and, so long as the data stream keeps flowing, provides the same level of concurrency. Raised levels of concurrency can be exploited by multicore processors (section 3.3.5).

3.3.4 Hoiks

We introduce a new type of agent into the system: *hoiks*. Hoiks are predators, feeding on boids. If a boid sees a hoik, it shrieks an *alarm* and heads off – as best it can – in the opposite direction. If a boid sees an alarmed boid, it also shrieks and modifies its behaviour to concentrate on what already alarmed boids are doing. The aim is to disseminate information about the predator across a *flock* of boids almost instantly and to generate a suitable (survival maximising) response.

For the moment, we are interested in the behaviour of boids in response to a hoik. Hoik behaviour is kept very simple – they just stay still, lurking wherever they are first placed. It would, of course, be easy to make them fly but we would need to determine an appropriate set of low-level rules. Modifying boid behaviour requires adding a new rule process to implement its reaction to seeing a hoik (or hoiks) and, possibly, tagging the database stream with a *hoik-close* field that modifies the behaviour of the other rule processes. In this latter case, the hoik rule process needs to be at the start of the pipeline. This will be discussed further in section 4.1.4.

3.3.5 Multicore and multiprocessor performance

The system shown in figures 4, 5 and 8 has 800 occoids. Each *occoid* has 6 sub-processes (including a *hoik* rule), making 4800 so far. There are 40 space servers (one process each), a visualiser agent, several hoiks and a handful of others to manage user interaction (to modify crucial agent parameters on-the-fly and, so, conduct experiments). This totals less than 5000 processes – a trivial number for the occam-π run-time system.

Processes are dynamically scheduled on all cores in each processor platform (section 3.1), making best efforts to minimise cache misses by keeping directly communicating processes on the same core – even as that togetherness changes with movement. With 5000 processes and relatively few cores, there is more than enough *parallel slackness* [50] to keep all cores busy on application work almost all the time. Details of this scheduler (CCSP) are reported in [31].

This scheduling is completely free – the occam-π programmer does not have to make any hints; just program lots of concurrency and interaction.

Less dynamic concurrency has problems with highly dynamic systems of the kind reported here. If we had explicitly to schedule processes across cores, we might try dividing the cores equally amongst the space servers – but this becomes unbalanced (and, therefore, inefficient) when some servers contain more agents than others. Alternatively, we might try dividing the cores equally between the agents – but the amount of processing required by an agent depends on how many agents it can see and this will not be the same for all agents and can again become unbalanced.

When the occam-π simulation is running interactively, the time taken to render the display is the limiting factor on simulation performance. There is an option in the simulation to update the display only every 256 (or 512, etc.) cycles, removing this bottleneck. With this option enabled, the cycle time scales in inverse proportion to the number of CPU cores used up to 8 cores; this demonstrates CCSP's (and occam-π's) excellent scalability on multicore processors.

The same model can be used in a distributed simulation with message-passing over a network. However, scheduling across *processors* with separate memory (e.g. in a cluster) is *not* completely free for the occam-π programmer (as yet). Fortunately, the process-oriented programming model makes it straightforward to *refactor* a program into a distributed version without changing its behaviour. The occam-π standard libraries provide facilities to *extend* communication channels and barriers between cluster hosts [45, 10]. A trivial refactoring replaces local channels with network channels, which have the same semantics but much higher communication latency. Further refactorings mitigate the effects of increased latency on simulation performance [38]. Proxy processes are used to provide local caches of remote information, avoiding redundant cross-network communications. Mobile process techniques allow processes to be migrated dynamically between hosts to improve locality of communication.

While processes may be distributed simply to balance load between hosts, it is sometimes useful to provide an explicit mapping between physical or state space within a simulation and the structure of hosts in a cluster. Running *occoids* on the Display Wall at the University of Tromsø, a 22-megapixel display generated by 28 cluster hosts driving separate projectors, requires both distributed simulation and distributed visualisation [38]. In this case, it is convenient to map simulated space directly to the cluster structure, with boids migrating between hosts as they cross the boundaries between projected displays. The resulting simulation scales linearly, with speed of execution remaining constant as the area being simulated and the number of agents (boids, hoiks etc.) are expanded.

Finally, we note that zero-copy communication between occam-π processes is achieved through the use of *mobile data*, allowing data ownership to be tracked precisely [6]. The present occam-π language has restrictions on the structures that can be dynamically allocated – just processes, channels, barriers and simple data structures that do not permit cycles of reference. The data tracking and alias analysis performed by the compiler then allows passive garbage to be collected through simple reference counting. Processes reallocate their used space automatically upon termination. For the future, the MirrorGC project [24] at Kent is developing techniques that make use of tracking and alias information to support efficient garbage collection in concurrent programs, with no restrictions on data structure. Meanwhile, the current restrictions cause no problems for the systems described in this paper.

4 Engineering Emergence

Section 3 concerns *rational* design. It gives us an infrastructure whose mechanics are simple, verified to be safe and efficiently executable on multicore and distributed platforms. The infrastructure enables the simulation of natural (or invented) worlds populated with autonomous mobile agents interacting with each other and their environment, all with a consistent view of space and time. What happens when we release them, though, is not entirely predictable or under our control.

This section considers the unprogrammed behaviours that *emerge* from mass interactions between the mobile agents and the (virtual) worlds constructed by rational design. These behaviours have potential value (section 2) and we have growing ambition for them – but *directly* planned rational design and implementation may be beyond our engineering abilities, at least with our present construction tools and the theories underlying them. Learning to work with, provoke and ultimately control and exploit such emergent behaviours requires research into the phenomena we are being shown, with an approach based upon *scientific method*.

4.1 Observations and reflections on the boids and hoiks

The simulated system (discussed below) has 800 occoids, constructed as described in sections 3.2 and 3.3. The occoid *vision angle* (section 3.3.2) is initially set to 300 degrees (i.e. with a 60 degree blind region behind). Figure 8 shows a sequence of images from one run (with the occoids starting from randomised positions and vectors, and the vision angle set to 360 degrees). There is one *hoik* (hawk) in the system – stationary near the middle of each simulated image and rendered as a small red square. This sequence is discussed in Section 4.1.1. Figure 9 shows *phase changes* in emergent behaviour as the vision angle is reduced from 360 to 90 degrees (and is discussed in Sections 4.1.2 and 4.1.3).

All the behaviours described in this and following sections may be viewed in the demonstration videos at [41]. The execution platform was a Macbook Professional with a 2.4 GHz Intel Core 2 Duo processor and 2 Gbytes SDRAM.

4.1.1 Flocking

We have become accustomed to the basic emergent property arising from Reynolds' boids model (section 3.3) – *flocking*. Nevertheless, it still causes surprise, interest and some pleasure to observe ... and raises new questions.

When started with all occoids in random positions and with random vectors (and with all cycles being rendered and displayed), it runs at around 35 frames per second (with both cores fully utilised). Figure 8(a) shows the scene less than one second into a run. Small flocks form within ten seconds (350 cycles), moving with different trajectories. Larger flocks form as they collide – the ones shown in figure 8(b) are about one minute into the run. Frame rate has reduced to 25 per second (i.e. 1.4x slower) because larger flocks require more computation to process the greater number of birds each one now sees – not because of any imbalance in the occam-π multicore scheduling, which continues to work all the cores fully (with linear scaling in performance as described in section 3.3.5).

Figures 8(b) and 8(c) show strange shapes of flocks[1] that have arisen following earlier collisions, mergers and encounters with the hoik. Figures 8(d) through 8(f) are discussed in sections 4.1.4 and 4.2.2. We neither programmed for, nor anticipated, such shapes. Understanding and controlling the mechanisms involved will take some research.

If left to themselves (i.e. if the hoik were not present), the flocks become larger and more cohesive until only one *superflock* is left drifting slowly around its universe – in apparent violation of the second law of thermodynamics.

[1] Photographs showing remarkably similar shapes in flocks of starlings can easily be found by searching the *Internet*. They cannot be included in this paper for copyright reasons, nor referenced because of the volatility of their URLs. However, links to images matching those in figure 8 can be found in the supporting materials for this paper [60].

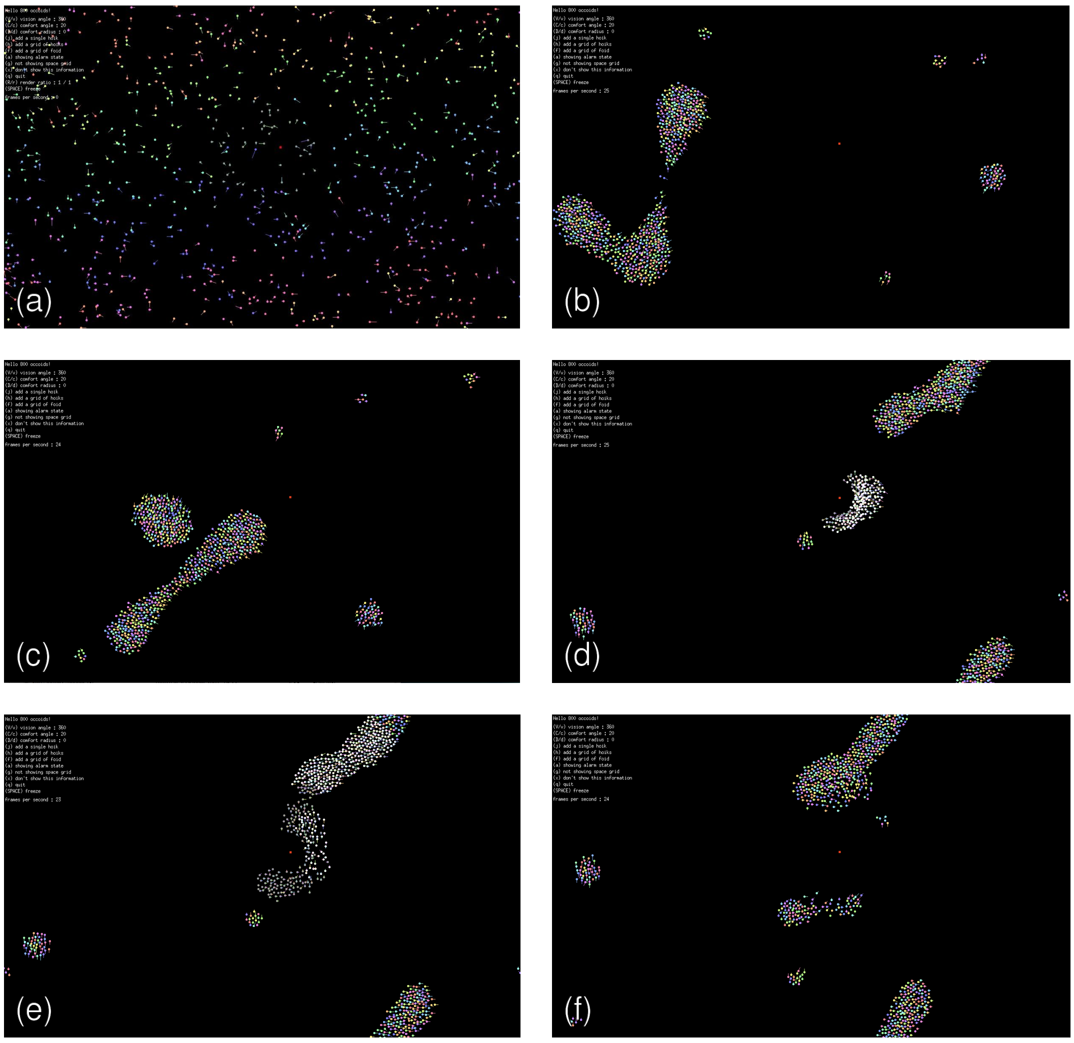


Fig. 8 Swarms: boids encountering a hoik (*hawk* or *falcon*).

4.1.2 Squabbling

With vision angles less than around 300 degrees (figure 9), the flocks (especially the smaller ones) show another unexpected phenomenon: *squabbling* between the birds at the rear. The leading front of the flock remains fairly disciplined and smooth. The trailing edge is less dense but very active, with apparent *fights* between the birds continually evicting *losers* that then hunt around looking for a way back. They make repeated darting attempts to rejoin. Usually, they do eventually succeed – but sometimes they are thrown out too far and lose contact, or some other passing birds attract them away.

The above "explanation" cannot, of course, be valid. We are ascribing motivation, planning and, even, consciousness to agents that we have built and which we know have no such capability – they can only follow Reynolds' low-level rules (section 3.3) that we programmed into them!

We can *experiment* some more to investigate this. Our occoids world allows an observer to change a number of crucial parameters during runs so that experimentation on a live system, giving real-time feedback, is easy to perform. We find that changing the vision angle of the birds makes a difference: reducing the angle increases the squabbling, increasing the angle reduces the squabbling. At 360 degrees (perfect all-round vision), the squabbling vanishes and the flocks take up an almost perfect circular shape, with smooth edges all around – see the top-left image of figure 9. As the vision angle is turned down to 270 degrees, the flocks take longer to form, are less stable (i.e. break up more easily) and the flock in-fighting extends from the back much further towards the middle.

Something strange and interesting is happening. There is a system parameter (*vision angle*), whose purpose and implementation was planned and rationally implemented, that is controlling a behaviour

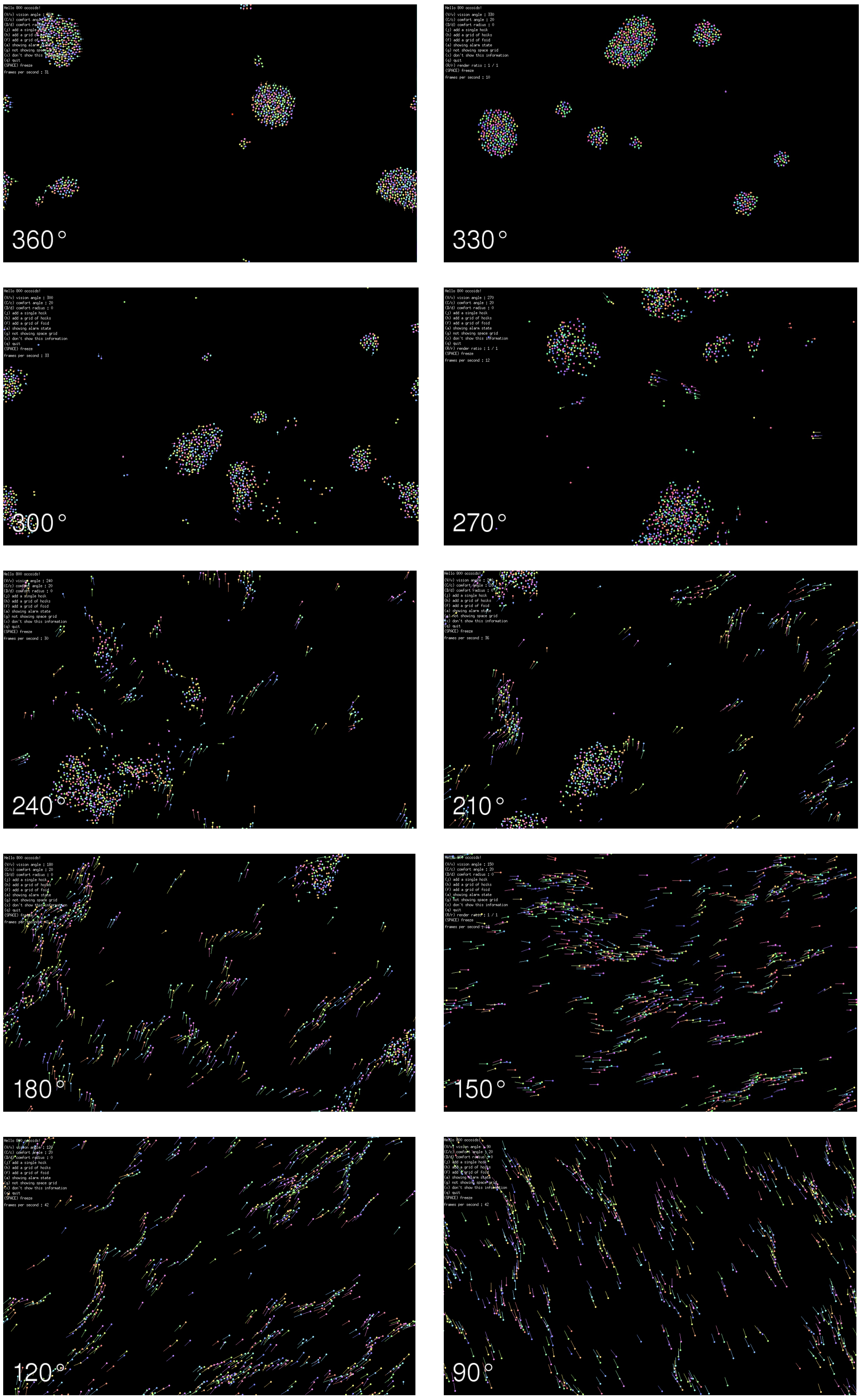


Fig. 9 Phase change in emergent boid behaviour from reducing vision angles.

that was not planned. That behaviour (*squabbling*) relates to a concept (*flock*) that was also unplanned and for which there corresponds no programmed entity! If such a discovered behaviour turns out to be useful, how are we going to engineer its safe harnessing and be confident in that engineering? We postpone considering this to section 4.2.2.

4.1.3 Streaming

If the vision angle is reduced still further, occoid behaviour changes suddenly and completely. At around 180 degrees (figure 9), the flocks break down and the birds scatter in all directions. They do notice each other still and gradually come together again – but this time in thin fast-moving *streams* that *undulate* across the screen. To begin with, there are several competing directions for these streams. Slowly though, one direction comes to dominate and the system settles to several parallel *undulating* streams.

This change is akin to phase changes in physical systems – e.g. the shift from water to ice. We can find a point for the vision angle – around 210 degrees – when the system exhibits elements from both behaviour patterns. There are one or two small flocks, trying to maintain cohesion, but continually losing birds to the passing streams and gaining some that plough straight into them. Whilst these exchanges remain in balance, the flock survives. As the vision angle narrows, the balance changes and the flocks disappear – leaving just the streams.

So, the same system parameter (*vision angle*) that controls the squabbling phenomenon (section 4.1.2) also controls this phase change to a completely different behaviour pattern. Now we are dealing with three interacting phenomena (*flocking*, *squabbling* and *streaming*) whose mechanisms we have not programmed, do not understand but may find useful to control. They need to be – and can be – researched just like any *natural* phenomena.

4.1.4 Fright and flight

Section 3.3.4 introduced *hoik* predators into the *boid*s' world. A simple fourth rule is added to those (Section 3.3) governing boid behaviour:

– *really try to avoid hoiks.*

This is implemented by an extra process (*fright*) added to the pipeline of the boid architecture shown in figure 7. The basic logic is similar to that for collision avoidance between boids – the difference being that the force felt by a boid spotting a hoik is much larger than that caused by a close-approaching boid and starts taking effect from a much greater range.

This determines the behaviour of an *individual* boid when it sees a hoik: the reaction force turns it quickly away from the danger. We are interested in controlling the behaviour of a *flock* to a hoik: but we have no programmed entity in the system corresponding to a flock and, therefore, nothing that we can directly engineer!

Letting the system run with *fright* in the boid brain, flock behaviour seems reasonably well engineered. Boids on the edge of a flock see the hoik first and turn away, heading always *into* the flock at some speed. Collision avoidance processes kick in, causing boids in their path to scatter and themselves to slow down. If the flock is large enough, the other rules (match-vectors, head-for-flock-centre) restore some cohesion and the flock is not completely disrupted. If the flock still drifts closer to the hoik, a wave of pressure seems to bow the flock edge closest to the hoik *away* from the hoik, with the rest of the flock flowing around the produced bay. We surmise that this pressure is the result of continued observations of the hoik by boids trapped between the hoik and the rest of its flock.

An example of this is shown in the sequence of images in figure 8. In (d), a small flock has encountered a hoik (the small red square at the centre) and a bay has formed. This flock was heading north and is now turning around the hoik and being pushed towards the north-east (and another flock). Image (e) is a little later in the run: the lower part of the flock has been stalled by the presence of the hoik, while the upper part has got past the hoik and is stretching away. In (f), the flock has been split with the upper part colliding and joining the other flock.

We would like to engineer flock behaviour so that it is more resistant to being broken up by approaching predators – although the hoik is probably quite pleased! Real starling flocks do seem more resilient [60]. We will return to this in section 4.2.2.

4.1.5 Hunger, caution and frenzy

A new agent type is added to the system: *foid* (or food). Like the hoiks (for now), foids do not move around the world. They are opposite to hoiks in that they *attract* boids, which now have a fifth rule:

– *try to catch foid.*

This is implemented in the obvious way, with another process in the pipeline inside the boid's brain that contributes a (modest) attractive force to any observed foid.

There was no special behaviour we wanted to engineer for this extended system – we were just curious to experiment.

Two behaviours emerge. If only a few (no more than four or five) birds pass close enough (and slowly enough) by a foid agent, they start circling closely and calmly – as if with great *caution.* Although they occasionally get very close, they never completely reach the foid. Eventually, they are attracted away by other passing boids.

On the other hand, if a larger flock approaches, its boids seem to give each other confidence to get to the foid and a *feeding frenzy* takes place, with quite excited zooming back and forth by *all* of them.

The frenzied behaviour over foid never happens with a boid on its own or in a small group – it needs a flock. The orbiting around foid never happens with a flock – it needs individuals. Neither of these behaviours was planned; but both are interesting and we may want to learn how to engineer them and manage them safely.

4.2 Complex systems, simulation, science and engineering

4.2.1 Research and scientific method

Simulation of complex worlds is interesting for its own sake, but has two important areas for application: one is to assist in the scientific understanding of natural phenomena and the other is the engineering of useful artifacts that might otherwise prove impossible. In science, we research the causes and mechanisms of phenomena and try to build theories that explain, quantify and predict. In engineering, we use those theories to control and use the phenomena.

Simulation also lets us test our theories when physical experiments may be too hard, expensive or dangerous. It is particularly useful when our theories are weak – for example, when we have only partial understanding of the agents involved (and, maybe, are not aware of all of them) and no understanding of how the studied phenomena arise (i.e. *emerge*) from them.

We compare simulated phenomena against real ones. When there are discrepancies, we try *tuning* parameters in the simulation (system wide or agent specific) to reduce them. If that doesn't work, we question the relevance or behaviour of particular agents and whether there are others that should be found. In this way, simulation becomes a valuable *instrument* for scientific research (like a microscope) [2].

Such iteration (observe real-world → theorise → build simulation from the theory → observe simulation → observe real-world) continues until we get it right enough, or give up. As understanding develops, the theories need to *quantify* relationships (i.e. write *equations*) between properties of the agents directly simulated and properties of the emerging phenomena. Those equations must stand testing in both the simulated and real world.

We may never understand precisely *how* emergent behaviours arise, but if we have the equations we can start to do some real engineering. This maturity has to happen before emergent properties can be used for safety-critical applications in the real world (see section 2.3). Many near future safety-critical applications will have such complex requirements that emergence may be the only way to approach them.

4.2.2 Research: flocks, cohesion and predator attack

Flocking models have relevance in zoology, biology, robotics, economics, social sciences, Internet commerce and other fields. An aspect of flocks of (potentially) crucial importance for such subjects is *cohesion*, especially when under attack by predators. This is something over which our boids simulation does not have perfect control (sections 1.4 and 4.1.4).

To improve this engineering, we need to have good definitions for flocking and cohesion – and *metrics* for measuring them. We do not attempt to engineer these behaviours directly, because we do not know how to do this! However, armed with such metrics, we can put on a scientific basis experiments that explore the effect of aspects in the system we *do* engineer (the individual behaviour of agents, their adjustable parameters etc.) on the behaviours we seek.

Before that, there are numerous parameters of the system that govern our *low-level* engineering: for example, factors controlling the strength of forces for collision avoidance, vector alignment, mass attraction, hoik repulsion; factors in the weighted sum that integrate all of the above; favoured flying speeds and speed/acceleration limits; vision radius and angle. All these need tuning – a process akin to the *calibration* of a scientific instrument (which is what the simulation has become).

Once this is done and we are satisfied with the low-level behaviours of *individual* boids we observe, we can attend higher-level behaviours. Sections 4.1.2 and 4.1.3 note that *vision angle* impacts on the varying densities of flock (greatest at the front), squabbling (at the rear) and the kinds of flock that emerge (shaped slow-moving patches or fast-moving undulating streams). The relationships describing this impact can now, and should be, formalised and tested by experiment.

However, we are concerned about flock cohesion. The sequence of events in figures 8(d) through 8(f) show a hoik harassing a small flock, with the flock behaviour eventually causing it to split in two – not the reaction we want. That flock has boids with all-round vision (i.e. 360 degrees). Reducing this angle, we observe even less stability in flocks under attack.

Section 3.3.4 describes the programmed behaviour of an individual boid to the presence of hoiks. Initially, we only had the rule that boids directly seeing a hoik turned in the opposite direction (as best they could, given their current vectors). Other boids were influenced by those that were panicking just by the standard rules (avoid collision, align vectors, move to centre of flock). The results were as described in section 4.1.4.

Then, we added the *alarm* rule – also given in section 3.3.4. This certainly has the effect of flooding awareness of the danger across the whole flock almost instantly. Boids spotting a hoik go into an alarm state (they *"shriek"*). Boids spotting an alarmed boid also get alarmed. In a few cycles, all boids in the flock become alarmed. Alarmed boids gradually calm down to their non-alarmed state.

Our visualisation shows this by colouring an initially alarmed boid white, holding this colour for a number of cycles, changing to grey, holding this for some more cycles before restoring its natural colour when it has calmed down (its alarm state having reduced to zero). Figure 8(d) shows an alarm *shock wave* sweeping across the small flock in the centre, bowing away from the hoik. In figure 8(e), the shock wave has jumped to the flock above (that the top half of the initially alarmed flock is about to split off and join, figure 8(f)). The visual effects are similar to watching swarms of *firefly* synchronising with sweeps of changing colour.

The rapid spreading of information across a flock – far faster than individual boids can move – and its limitation to the flock is an emergent behaviour that *was* anticipated and engineered. It is an emergent effect because no flocks are programmed (there are no flocks in the program). We are able to do this because the behaviour is a relatively simple emergence and we can see the whole relationship between cause and effect.

A key element of the alarm rule is that an alarmed boid only matches vectors with boids that were alarmed *before itself.* In this way, we tried to engineer a more forceful reaction in the flock away from the predator – alarmed boids would pay more attention to what boids nearer than them to the source of panic were doing. However, not much difference was observed when this rule was added. Possibly, this is because the alarm reaction spreads so fast that there is insufficient gradation of alarm levels across the flock, so that the directional information towards the source of the alarm is lost.

This indicates the difficulty of engineering emergent behaviours. We have to look for simple rules that can be implemented through *local* effects on individual agents. We need to quantify them, along with metrics for the sought behaviours, conduct experiments and analyse the results. Our experiment with the alarm rule is (so far) negative, but this is still valuable – and happens frequently in science.

Of course, we need to look at what other researchers are doing. Many are also working with simulations. If there are promising results, we should try to reproduce them using our own instruments – i.e. the occam-π concurrency infrastructure described in this paper. If the results are reproduced, worries that the emergent effects being constructed may be down to programming bugs will be greatly reduced.

We reference just three (from very many) studies here. Hemelrijk and Hildenbrandt [20] report variations on Reynolds' rules in their (3-D) simulation of fish shoals. Each rule works on a different vision range: collision avoidance being smallest (as normal), then vector alignment, then mass attraction. Further, these ranges change (inversely) with local fish density – the aim being to look further when there are few neighbours and not look *through* fish that are very close. The vision angles for the mass attraction rule is less than those for the other two. The vector alignment rule has a forward, as well as backward, blind region. They report solutions that produce shoals with good cohesion, though they are not challenged by predators. We mention this work to show how the space of parameters for the basic rules can be simply and considerably expanded.

Sayama's Swarm Chemistry model [43] expands the parameter space differently: it features multiple heterogeneous populations of boid-like swarming agents which cooperate and compete to produce a variety of emergent behaviours. Through interactive, directed evolution of the model parameters for the different populations, users can design swarming systems that exhibit striking biological and non-biological characteristics (oscillation, fluid dynamics). Sayama's recent work explores the effects of different evolutionary mechanisms on the robustness and complexity of designed systems when subjected to undirected evolution, showing how engineered complex systems can self-adapt to changing environmental conditions using evolutionary techniques.

Finally, Ballerini et al. [5] report on detailed observations of real starling swarms using 3-D photography (5 frames/second), with computer extraction of 3-D coordinates for each bird. Swarms with over 1000 birds were studied. They conclude that each bird interacts with only its nearest six (or seven) neighbours rather than as many as are in its visual range – a *topological* rather than *metric* neighbourhood. They give arguments from evolutionary biology to explain this, emphasising the benefits this may yield for flock cohesion – especially in the presence of predators. They report some 2-D simulations based on this (with interactions reduced to only three or four birds, because of the reduced dimensionality) that strongly support their conclusions.

4.2.3 Unexpected relationships between phenomena

Science and mathematics have many examples where studies in one field have surprising application in another. Sometimes this leads to the discovery of unsuspected commonalities in the underlying mechanisms, even though those mechanisms operate through different agents and media. The better understanding promoted by such discovery furthers research and development in both fields.

Fluid dynamics and bird flocking may seem to be different fields of study. However, videos of flocks of starlings give the appearance of *"intelligent"* single organisms whose bodies are *fluid*, rather than solid, wheeling and folding across the sky with great speed, flexibility and rational purpose. Artifacts with similar properties may have interesting application (section 4.2.5).

Flocks in our `occoids` simulation turn to streams as the vision angle of their component boids is reduced (section 4.1.3 and figure 9). Below 180 degrees (i.e. forward and some lateral vision only), we are left only with parallel undulating streams – all moving with similar speed.

We can place *"rocks"* in these streams and observe the effects. We reuse our `hoik` agents for these rocks: they repel approaching `occoid`s which bend and flow around them. As these hoiks currently do not move, placing a (roughly square) phalanx of them creates an significant obstruction to stream flow. We are interested in whether any *turbulence* emerges.

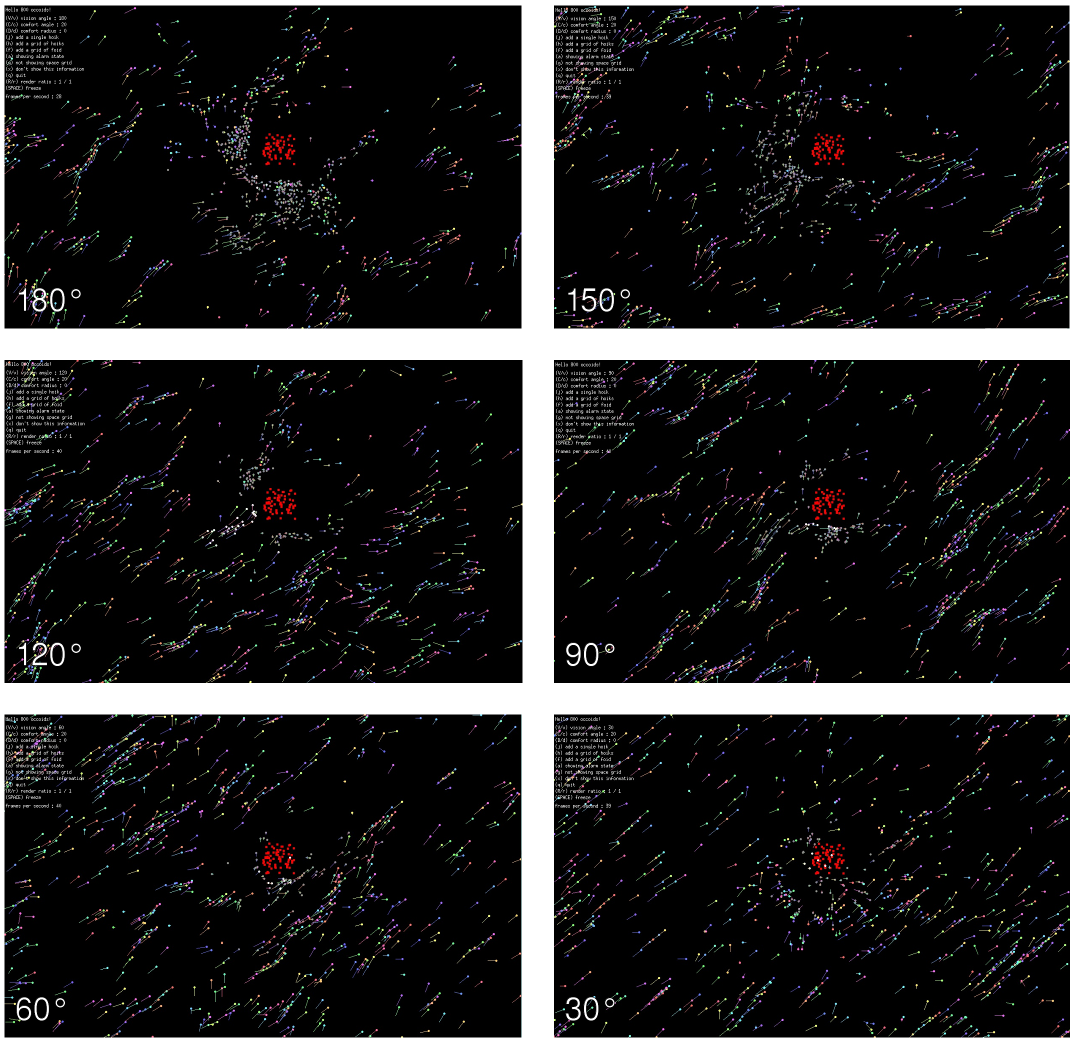
Figure 10 shows snapshots from the `occoid`s simulation with the *"bird"* vision angle reducing from 180 degrees (forwards and sideways only) to 30 degrees (tunnel vision) in steps of 30.

With 180 degree vision, stream agents (i.e. `occoids`) approaching the rock head-on *and on either side* feel back-pressure and turbulent pools form immediately upstream and to the sides of the rock. The pools are very dynamic, continually refreshed with new agents and with older ones eventually working their way around and overflowing back into the running streams. Immediately downstream from the rock is a relatively empty region, sheltered from the prevailing flow. This shelter disappears further on as passing streams, that were pushed aside by the rock pools, re-occupy the empty space (evidenced by the *"new"* streams approaching the rock head-on, which are wrapped around in this toroidal world from the downstream region).

With 150 degree vision, the upstream rock pool is smaller and the streams re-fill the downstream region more quickly. At 120 and 90 degrees, the upstream pool is just a small splash area – but the downstream area is more active with streams pushing back earlier and colliding. At 60 and 30 degrees, the undulations in the streams have mostly gone and close-passing streams flow by, with deviations only caused by stream agents hitting rock agents head-on and splashing back – seen especially in the 30 degree vision image.

Here, the vision angle seems to be reflecting the *viscosity* of the fluid in the stream. High values give high viscosity and back-pressure from the rock has a wider influence on fluid flow. Low values imply low viscosity and the rock only causes localised splashing.

A new behaviour also emerges from the low vision angles. The "rock" is somewhat *porous* – it was constructed from stationary hoiks placed randomly in a square area, so there are plenty of holes. Indeed, empty space inside the rock forms a *maze* of tunnels. With tunnel vision, directly approaching boids see



```
Hello 800 occoids!
(V/v) vision angle : 180
(C/c) comfort angle : 20
(D/d) comfort radius : 0
(j) add a single hoik
(h) add a grid of hoiks
(f) add a grid of foid
(a) showing alarm state
(g) not showing space grid
(x) don't show this information
(q) quit
(R/r) render ratio : 1 / 1
(SPACE) freeze
frames per second : 28
```

```
Hello 800 occoids!
(V/v) vision angle : 150
(C/c) comfort angle : 20
(D/d) comfort radius : 0
(j) add a single hoik
(h) add a grid of hoiks
(f) add a grid of foid
(a) showing alarm state
(g) not showing space grid
(x) don't show this information
(q) quit
(R/r) render ratio : 1 / 1
(SPACE) freeze
frames per second : 39
```

```
Hello 800 occoids!
(V/v) vision angle : 120
(C/c) comfort angle : 20
(D/d) comfort radius : 0
(j) add a single hoik
(h) add a grid of hoiks
(f) add a grid of foid
(a) showing alarm state
(g) not showing space grid
(x) don't show this information
(q) quit
(R/r) render ratio : 1 / 1
(SPACE) freeze
frames per second : 40
```

```
Hello 800 occoids!
(V/v) vision angle : 90
(C/c) comfort angle : 20
(D/d) comfort radius : 0
(j) add a single hoik
(h) add a grid of hoiks
(f) add a grid of foid
(a) showing alarm state
(g) not showing space grid
(x) don't show this information
(q) quit
(R/r) render ratio : 1 / 1
(SPACE) freeze
frames per second : 40
```

```
Hello 800 occoids!
(V/v) vision angle : 60
(C/c) comfort angle : 20
(D/d) comfort radius : 0
(j) add a single hoik
(h) add a grid of hoiks
(f) add a grid of foid
(a) showing alarm state
(g) not showing space grid
(x) don't show this information
(q) quit
(R/r) render ratio : 1 / 1
(SPACE) freeze
frames per second : 40
```

```
Hello 800 occoids!
(V/v) vision angle : 30
(C/c) comfort angle : 20
(D/d) comfort radius : 0
(j) add a single hoik
(h) add a grid of hoiks
(f) add a grid of foid
(a) showing alarm state
(g) not showing space grid
(x) don't show this information
(q) quit
(R/r) render ratio : 1 / 1
(SPACE) freeze
frames per second : 39
```

Fig. 10 Turbulence: streams and rocks, with reducing *"bird"* vision angles. The streams are flowing diagonally, from bottom-left to top-right.

the rock too late to take avoiding action; they "feel" little back-pressure. If they "hit" something solid, they bounce off. If they find a hole, they dive straight in! They then find themselves in tunnels whose walls are made from hoiks. They are very scared. They find themselves pushed through the maze by large forces and exit (where there is no repelling force) at the earliest opportunity.

So, we have four emergent behaviours – *flocking*, *streaming*, *turbulence* and *maze solving* – arising from the same space-time model populated by the same bird and predator agents. The different behaviours are generated by changing a single parameter (the vision angle of the birds) and the positioning of the predators. We find this interesting and unexpected. Even if these particular experiments find no application, the discovery – through simulation – of common mechanisms leading to apparently different emergent behaviours increases our awareness of the skills needed to engineer them and may lead to advances in understanding of the phenomena involved.

4.2.4 A note on obstacles

Reynolds' original work [30] describes two methods for avoiding obstacles: a *force field* model (where the obstacles radiate force vectors that impact on approaching boids) and *steer-to-avoid* (where the boids look out for imminent obstacles, search for an edge and compute vectors to just miss). The force field idea is not favoured since, if a boid is heading exactly against a completely symmetric force field (e.g. towards the centre of a disc-shaped obstacle or perpendicular to a wall), all that happens is that the boid will slow down (either crashing or going into reverse) but will not deviate.

Reynolds' video (linked from the URL given in the *References* section for [30]) shows a small flock of boids (around 30) flowing *smoothly* around both sides of cylindrical obstacles, rejoining on the other side and continuing in their original direction. Nevertheless, one of the boids manages to crash into a cylinder, bounce back and lose contact with the flock. Presumably, this is caused by the close presence of neighbouring boids, whose influence (through the standard flocking rules) defeated the obstacle avoidance rule.

The objection to the force field model is valid only for a *single* boid approaching a symmetric obstacle head-on. For flocking boids and irregular obstacles, numerous irregular local forces prevent the envisaged problem scenario (straight line braking and reversing) from happening. For occoids, hoik avoidance (section 3.3.4) is by force field – as is collision avoidance between boids, though that field has much lower strength. In (the previous) section 4.2.3, a ragged phalanx of hoiks was used for obstacles (rocks) in streams of boids (with low vision angles). *Turbulent*, not smooth, flow emerges and that seems useful for future research and application.

For occoids with higher vision angles (so that flocking, rather than streaming emerges), the rocks of figure 10 cause approaching flocks to turn away rapidly, if they get too close, and often break into sub-flocks. The smooth flow round the obstacle and re-join shown in Reynolds' video does not happen. The reasons for this are the strength of aversion to the *hoik-rock* by the occoids, the raggedness of the obstacle and, possibly, the alarm rule triggered by hoiks (section 3.3.4).

In [41], the first demonstration video shows occoids in a world with disc obstacles that are *not* made up from a dense mass of hoiks! These obstacles are still implemented by force fields, but with strength similar to those for collision avoidance between occoids. These occoids have high vision angles (300 degrees). The emerging flocks *do* flow smoothly past these obstacles, bouncing gently if they get too close but generally maintaining flock cohesion. They do not flow around both sides and re-join (like the boids in Reynold's video) partly because the obstacle diameters are larger than the emerging flock widths, and partly because no global direction (or target point) for the occoids has been built into their low-level behaviour rules.

We are always concerned to keep things simple – both in concept and computational load. The force field rule for obstacles has this simplicity and, except for non-flocking agents and perfectly symmetric obstacles, yields familiar emergent behaviours. The literature on obstacle avoidance has many examples that stress the computational complexity required (e.g. [26,16]). Reynolds' paper [30] describes the need for different algorithms to implement the steer-to-avoid rule for different obstacle shapes, as well as their implications for computational load. Our occoids have no special rules for individual shapes of obstacle, so their avoidance costs no more than collision avoidance between occoids ... and system perfromance continues to scale well with the number of processor cores and processors (section 3.3.5).

In the next section, a new case study is introduced in which the moving agents *do* use a steer-to-avoid rule. However, the same rule avoids both agent-agent and agent-obstacle collisions. Further, no searching for obstacle edges is required – hence, there is no need for special rules for special shapes. The computational load is proportional to the number of agents (not its square) so, as with the occoids study, the system remains light and simple.

4.2.5 Exploring new physics

Hinted at in section 4.2.3 (second paragraph), is the prospect of constructing new artifacts. Inspired by observing natural phenomena, but constructed using materials with properties we currently do not know how to manufacture, we can use simulation to model those properties and see if something emerges that we can discover how to control (at least, within the simulation). In this way, we are using simulation not just as a scientific instrument with which to understand better complex systems of whose existence we are already aware. We are using simulation as a way of defining a *new* physics, exploring its consistency, developing its theory and considering its application. If something with useful potential emerges, that would be motivation to discover ways to manufacture the needed base materials. If those base materials have no special value *by themselves*, such motivation would be needed.

Figure 11 shows a sequence of images from a world different to the `occoids`. There are *"red"* agents and *"green"* agents and they live in a world of black space (through which they can move) and white walls (they cannot enter). Initially, they are all packed randomly and tightly into a small box in the centre of their world.

The agents are *nanobots* or, possibly, rather clever *ionised plasma particles* following special rules. The red agents spiral around aimlessly, but react when they bump into each other or walls (i.e. they feel and apply *pressure*). The green agents have more purposeful rules:

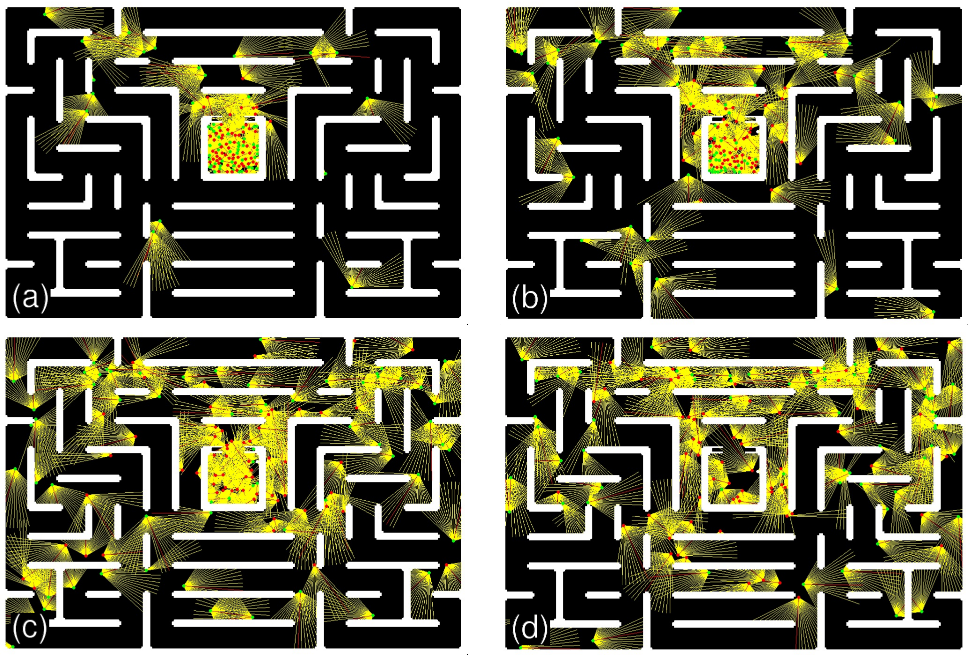


Fig. 11 Nanobots / active plasma: exploring a maze. There are 64 red agents mixed with 64 green.

– *head in the furthest clear direction you can see, with a speed proportional to that distance (which implies collision avoidance). In the case of a tie, choose a direction that keeps you going as straight as possible.*
– *if stuck for too long in one place, rotate by some random amount.*

Both kinds of agent have similar mechanical capabilities: they can move forwards (up to some maximum speed) and can spin clockwise or anti-clockwise (at a fixed rate of rotation). They are equipped with an array of (laser) range finders that fan out over a range of forward pointing angles. These deliver an array of target (i.e. wall or other agent) distances, up to some maximum range. This is the only information each agent has on its environment, from which it determines its movement (forward speed and spin).

The red agent rules lead to emergent global behaviour similar to gaseous diffusion. As they escape their initial packing, they diffuse evenly throughout their world – gradually reducing the density (and, so, partial pressure) of the red cloud.

The green agents, once they escape their entrapment, positively head out to explore their environment – performing *racing turns* at the corners and avoiding collisions. They fill their space much faster than the red agents: their behaviour is not like a slowly diffusing cloud, more like a swarm that seems to have purpose.

This is reflected in figure 11, which shows images from a mixed world of 64 red and 64 green agents. The range-finder beams are shown in yellow, with the central beam in red. Image (a) is soon after release. Mainly green agents have escaped and are racing to all parts of free space. The few red agents outside the box have not wandered far. This pattern continues through images (b) and (c), with red agents gradually reaching the outer limits. Image (d) shows almost all agents escaped from their initial box – just one green and eight red remain. The green agents remain most active, continually searching all areas.

Two potential applications for the *green* agents, at least, are apparent. One is for safe control of driverless automobiles (though a *destination* rule would need to be added). The traffic intersections in the example maze of figure 11 are extremely hazardous – yet traffic flows with no collisions or jams (even with the presence of the red agents, who cause a few bumps).

Note that this is accomplished *without* complex image processing or building a library of special procedures to implement rules of the road, plus rules for every variety of intersection (as is common for entries to the DARPA Grand Challenge [62]). In fact, the *green* logic implementing all control for the agent takes only 73 lines of occam-π. Of course, rules of the road (such as *drive-on-the-left*, traffic lanes, and traffic lights) would need to be abandoned and *all* cars would need to guarantee to abide by the same control. This is not beyond the bounds of the possible though – some experiments have already been successful (in reducing accidents and improving traffic flow) simply by removing traffic lights and lane markings [4].

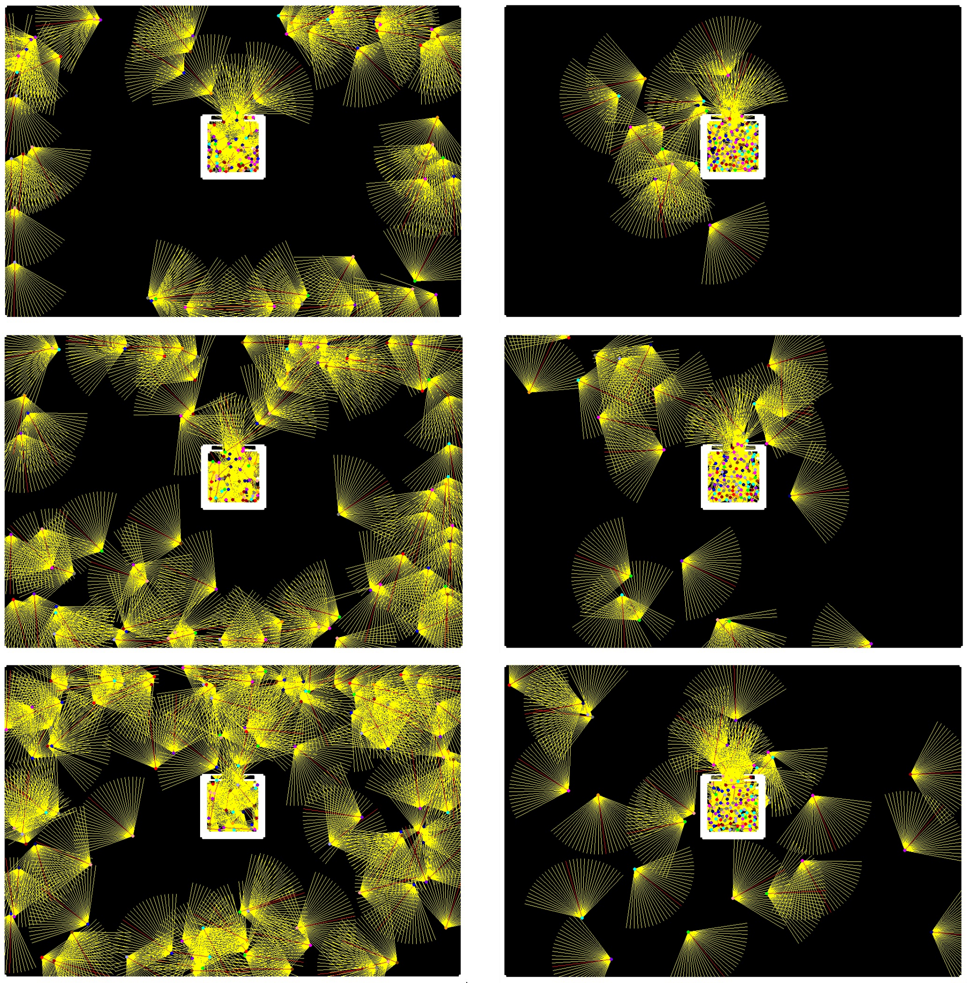


Fig. 12 Nanobots / active plasma: sealing a container. The left column shows a sequence of images with 128 *"green"* agents, though with multiple actual colours. The right column shows the behaviour of 128 *"red"* agents, sampled at similar times to the those on the left.

Figure 12 shows the potential for a second, more exotic application. The world and agents are the same, except for the removal of all walls in the maze – leaving just open space, a sealed outer container and the central box (in which all agents are initially packed). The left column is a sequence of images with just 128 *"green"*-logic agents (though the agents have many actual colours). The right column shows *"red"*-logic agents at similar time intervals.

The left column shows *green* agents, fountaining out of their packing and racing around the edges of the container. Even after some time has passed, the third image shows the green cloud applying greatest pressure to the outer walls of the container and mostly ignoring the central area – *emergent unprogrammed behaviour*. This is not the case in the right column, which shows *red* agents diffusing much more slowly and displaying no special preference for any area. With a little assistance from engineers of the future, the *green* cloud becomes a *shaped force field.*

5 Summary and Conclusions

We have explained our motivation for the *training* of emergent behaviours within complex systems to meet future engineering needs. We have presented a *massively concurrent* infrastructure for the modelling of dynamic space-time environments and the agents (mobile processes) that populate them – from which unprogrammed, but required, activities emerge. This infrastructure is *rationally* engineered to the extent that we have verified guarantees for the specified behaviour of individual agents and that the system is safe from low-level error (e.g. deadlock, livelock, data race hazard, starvation). The use of *process-oriented* concurrency, based on the CSP and π-calculus process algebrae and implemented through the occam-π multiprocessing language, greatly assisted here – and also yielded automatic and significant performance speed-up on today's multicore processors.

Everything else is *emergent* engineering. A case study based on Reynolds' *boids* model was presented. Into this were stirred agents representing predators (that repel the boids) and food sources (that attract). The difficulties of achieving certain behaviour goals (such as the resilience of flock cohesion under harassment from predators) was described. The sensitivity of switching between particular patterns (such as flocking or streaming) to variations of a single parameter (such as the angle of vision for the boids) was discovered and measured. Further behavioural changes (from orbital inspection of a food source to a sudden feeding frenzy) had other dependencies (such as the size of the flock encountering the food).

We are dealing with systems whose high-level properties we need to control; but we have not *directly* programmed them, do not understand the mechanisms generating them and are often surprised by their very existence. The importance of a rational approach, based on *scientific method*, to researching the emerging phenomena has been emphasised. Observations need to be made and theories developed and tested by experiment. Only then can we begin to have confidence in using such systems in critical applications.

We have presented examples showing that phenomena based on different physics and operating at different scales nevertheless emerge from common logical elements, perhaps differing only by values of key parameters. The commonalities so discovered may promote cross-fertilisation of research and engineering application of those phenomena in their respective fields.

Others have cogently argued, [2], that simulation should be considered as an important *scientific instrument* in the research and development of complex systems in science and engineering. We have extended this view to speculate on its use in pre-emptive studies for the research and exploitation of phenomena that are unknown (but imaginable) at present, and may emerge from low-level entities whose properties we can specify but not build (at least, at the required *nanoscale*) with present engineering skills.

Finally, we convey the satisfaction that emerges simply from watching the agents move and interact. In a world populated with birds, hawks and food in dynamically changing configurations, we see what appear to be *social systems* evolve and adapt as we adjust various parameters. We see small and large flocks moving between food sources, sometimes orbiting nervously (because hawks are watching nearby), sometimes feeding (when in sufficient numbers), sometimes staying in particular regions, sometimes breaking up and streaming away in waves before reforming elsewhere. Of course, there can be no such motivations in the agents themselves. However, with reference to the Turing test for artificial intelligence [49], we might further speculate: *"if it looks like a duck, and quacks like a duck, we have at least to consider the possibility that we have a small aquatic bird of the family Anatidae on our hands"* [1].

Acknowledgements This work would not have been started or completed without the skills, support and enthusiasm of all our colleagues on the CoSMoS project [47], the occam-π and KRoC development teams [40,8], and the Software Engineering Institute (Carnegie Mellon University). We owe special thanks to Fred Barnes, Carl Ritson, Paul Andrews, John Markus Bjørndalen, Linda Northrop and David Wood. We are also very grateful to our Editors and anonymous referees for their constructive and helpful criticism. Finally, we acknowledge some giants of the field on whose shoulders we gratefully stand: Tony Hoare and Bill Roscoe (CSP), Robin Milner (π-calculus) and David May (classical occam).

References

1. Adams, D.: Dirk Gently's Holistic Detective Agency. Gallery Books (1987). See also: http://en.wikipedia.org/wiki/Duck_test
2. Andrews, P., Stepney, S., Winfield, A.: Simulation as an Experimental Design Process. EmergeNET4: Engineering Emergence (2010). URL http://www.cs.york.ac.uk/nature/emergeNET4/andrews.pdf. AF/ST-TR-10-01-PR
3. Andrews, P.S., Sampson, A.T., Bjørndalen, J.M., Stepney, S., Timmis, J., Warren, D.N., Welch, P.H.: Investigating Patterns for the Process-oriented Modelling and Simulation of Space in Complex Systems. In: S. Bullock, J. Noble, R. Watson, M.A. Bedau (eds.) Artificial Life XI: Proceedings of the Eleventh International Conference on the Simulation and Synthesis of Living Systems, pp. 17–24. MIT Press, Cambridge, MA (2008). URL http://www.cosmos-research.org/docs/alife2008-space.pdf
4. Baker, L.: Removing Roads and Traffic Lights Speeds Urban Travel. Scientific American (2009). URL http://www.scientificamerican.com/article.cfm?id=removing-roads-and-traffic-lights

5. Ballerini, M., Cabibbo, N., Candelier, R., Cavagna, A., Cisbani, E., Giardina, I., Lecomte, V., Orlandi, A., Parisi, G., Procaccini, A., Viale, M., Zdravkovic, V.: Interaction Ruling Animal Collective Behavior Depends on Topological rather than Metric Distance: Evidence from a Field Study. Proceedings of the National Academy of Sciences **105**(4) (2008). URL http://www.pnas.org/cgi/doi/10.1073/pnas.0711437105. pp. 1232–1237
6. Barnes, F.R.M., Welch, P.H.: Mobile Data, Dynamic Allocation and Zero Aliasing: an occam Experiment. In: A. Chalmers, M. Mirmehdi, H. Muller (eds.) Communicating Process Architectures 2001, *Concurrent Systems Engineering*, vol. 59, pp. 243–264. WoTUG, IOS Press, Amsterdam, The Netherlands (2001). ISBN: 1-58603-202-X
7. Barnes, F.R.M., Welch, P.H.: Communicating Mobile Processes. In: I. East, J. Martin, P. Welch, D. Duce, M. Green (eds.) Communicating Process Architectures 2004, *Concurrent Systems Engineering Series, ISSN 1383-7575*, vol. 62, WoTUG-27, pp. 201–218. IOS Press, Amsterdam, The Netherlands (2004). ISBN: 1-58603-458-8
8. Barnes, F.R.M., Welch, P.H., Moores, J., Wood, D.C.: The KRoC Home Page. Programming Languages and Systems Research Group, University of Kent, http://www.cs.kent.ac.uk/projects/ofa/kroc/ (2010)
9. Bently, P.J. (ed.): Evolutionary Design by Computers. Morgan Kaufmann (1999)
10. Bjørndalen, J.M., Sampson, A.T.: Process-Oriented Collective Operations. In: P.H. Welch, S. Stepney, F.A. Polack, F.R. Barnes, A.A. McEwan, G.S. Stiles, J.F. Broenink, A.T. Sampson (eds.) Communicating Process Architectures 2008, *Concurrent Systems Engineering*, vol. 66, pp. 309–328. WoTUG, IOS Press, Amsterdam, The Netherlands (2008). URL http://www.cosmos-research.org/docs/cpa2008-poco.pdf
11. Brown, N.C.C.: Communicating Haskell Processes: Composable Explicit Concurrency Using Monads. In: P.H. Welch, S. Stepney, F.A. Polack, F.R. Barnes, A.A. McEwan, G.S. Stiles, J.F. Broenink, A.T. Sampson (eds.) Communicating Process Architectures 2008, *Concurrent Systems Engineering*, vol. 66, pp. 67–83. WoTUG, IOS Press, Amsterdam, The Netherlands (2008). URL http://www.cs.kent.ac.uk/pubs/2008/2914
12. Brown, N.C.C.: C++CSP Home Page. Programming Languages and Systems Research Group, University of Kent, http://www.cs.kent.ac.uk/projects/ofa/c++csp/ (2010)
13. Brown, N.C.C.: Communicating Haskell Processes Home Page. Programming Languages and Systems Research Group, University of Kent, http://www.cs.kent.ac.uk/projects/ofa/chp/ (2010)
14. Brown, N.C.C., Welch, P.H.: An Introduction to the Kent C++CSP Library. In: J. Broenink, G. Hilderink (eds.) Communicating Process Architectures 2003, WoTUG-26, Concurrent Systems Engineering, ISSN 1383-7575, pp. 139–156. IOS Press, Amsterdam, The Netherlands (2003). ISBN: 1-58603-381-6
15. Dahm, W.J.A.: Technology Horizons: A Vision for Air Force Science and Technology during 2010-2030 (Volume I). United States Air Force Chief Scientist (AF/ST) (2010). AF/ST-TR-10-01-PR
16. Davison, A.: Killer Game Programming in Java. O'Reilly Media Inc. (2005). Chapter 22, *Flocking Boids*
17. Dorigo, M., Stützle, T.: Ant Colony Optimization. MIT Press, Cambridge, Massachusetts (2004)
18. Easley, D., Kleinberg, J.: Networks, Crowds, and Markets: Reasoning About a Highly Connected World. Cambridge University Press (2010). URL http://www.cs.cornell.edu/home/kleinber/networks-book/
19. Edelman, B., Ostrovsky, M., Schwarz, M.: Internet Advertising and the Generalized Second Price Auction: Selling Billions of Dollars Worth of Keywords. Working Paper 11765, National Bureau of Economic Research (2005). URL http://www.nber.org/papers/w11765
20. Hemelrijk, C.K., Hildenbrandt, H.: Self-Organized Shape and Frontal Density of Fish Schools. Ethology **114**(3) (2008). URL http://onlinelibrary.wiley.com/doi/10.1111/j.1439-0310.2007.01459.x/abstract. pp. 245–254
21. Hoare, C.A.R.: Monitors: An operating system structuring concept. Communications of the ACM **17**(10), 549–557 (1974)
22. Hoare, C.A.R.: Communicating Sequential Processes. CACM **21**(8), 666–677 (1978)
23. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall (1985)
24. Jones, R.E., Barnes, F.R.M.: MirrorGC: Garbage Collection for Multicore Systems (2010). URL http://www.cs.kent.ac.uk/projects/gc/mirrorgc/. EPSRC EP/H026975/1
25. Klein, M., Moreno, G.A., Parkes, D.C., Plakosh, D., Seuken, S., Wallnau, K.: Handling Interdependent Values in an Auction Mechanism for Bandwidth Allocation in Tactical Data Networks. In: Proceedings of the 3rd international workshop on Economics of networked systems, NetEcon '08, pp. 73–78. ACM, New York, NY, USA (2008). URL http://doi.acm.org/10.1145/1403027.1403044
26. Lorek, H., Informatik, F., White, M.: Parallel bird flocking simulation (1993). Citeseer: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.48.8430
27. Martin, J., Welch, P.: A Design Strategy for Deadlock-free Concurrent Systems. Transputer Communications **3**(4) (1996). pp. 215–232
28. Milner, R.: Communicating and Mobile Systems: the π-Calculus. Cambridge University Press (1999). ISBN-10: 0521658691, ISBN-13: 9780521658690
29. Polack, F.A., Andrews, P.S., Sampson, A.T.: The Engineering of Concurrent Simulations of Complex Systems. In: 2009 IEEE Congress on Evolutionary Computation (CEC 2009), pp. 217–224. IEEE Press (2009)
30. Reynolds, C.W.: Flocks, Herds, and Schools: A Distributed Behavioral Model. Computer Graphics **21**(4), 25–34 (1987). See also: http://www.red3d.com/cwr/boids/
31. Ritson, C.G., Sampson, A.T., Barnes, F.R.M.: Multicore Scheduling for Lightweight Communicating Processes. In: J. Field, V.T. Vasconcelos (eds.) Coordination Models and Languages, COORDINATION 2009, Lisboa, Portugal, June 9-12, 2009. Proceedings, *Lecture Notes in Computer Science*, vol. 5521, pp. 163–183. Springer (2009). URL http://www.cs.kent.ac.uk/pubs/2009/2928
32. Ritson, C.G., Welch, P.H.: A process-oriented architecture for complex system modelling. Concurrency and Computation: Practice and Experience **22**, 965–980 (2010). DOI 10.1002/cpe.1433. URL http://www.cs.kent.ac.uk/pubs/2010/3066
33. Roscoe, A.: The Theory and Practice of Concurrency. Prentice Hall (1997)
34. Roscoe, A.: Understanding Concurrent Systems. Springer (2010)
35. Rosenschein, J.S., Zlotkin, G.: Rules of Encounter: Designing Conventions for Automated Negotiation among Computers. MIT Press, Cambridge, Massachusetts (1994)
36. Roughgarden, T.: Algorithmic Game Theory. Commun. ACM **53**(7), 78–86 (2010). DOI http://dx.doi.org/10.1145/1785414.1785439
37. Sampson, A.T.: Process-Oriented Patterns for Concurrent Software Engineering. Ph.D. thesis, University of Kent (2010). URL http://offog.org/publications/ats-thesis.pdf

38. Sampson, A.T., Bjørndalen, J.M., Andrews, P.S.: Birds on the Wall: Distributing a Process-Oriented Simulation. In: 2009 IEEE Congress on Evolutionary Computation (CEC 2009), pp. 225–231. IEEE Press (2009). URL http://www.cosmos-research.org/docs/cec2009-wall.pdf
39. Sampson, A.T., Ritson, C.G., Barnes, F.R.M., Welch, P.H.: occam-π Download and Install Page. Programming Languages and Systems Research Group, University of Kent, http://projects.cs.kent.ac.uk/projects/kroc/trac/wiki/Installation (2011)
40. Sampson, A.T., Ritson, C.G., Jadud, M.C., Barnes, F.R.M., Welch, P.H.: occam-π Home Page. Programming Languages and Systems Research Group, University of Kent, http://occam-pi.org/ (2010)
41. Sampson, A.T., Welch, P.H., Bjørndalen, J.M., Andrews, P.S.: CoSMoS occoids videos (2010). URL http://www.cosmos-research.org/demos/occoids/
42. Sauter, J.A., Matthews, R., Van Dyke Parunak, H., Brueckner, S.A.: Performance of Digital Pheromones for Swarming Vehicle Control. In: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems, AAMAS '05, pp. 903–910. ACM, New York, NY, USA (2005). URL http://doi.acm.org/10.1145/1082473.1082610
43. Sayama, H.: Seeking Open-Ended Evolution in Swarm Chemistry. In: IEEE Symposium on Artificial Life (ALIFE) 2011, pp. 186–193. IEEE Press (2011). ISBN: 978-1-61284-062-8
44. Schneider, S.: Concurrent and Real-time Systems — The CSP Approach. Wiley and Sons Ltd., UK, Baffins Lane, Chichester, UK (1999). ISBN: 0-471-62373-3
45. Schweigler, M.: A Unified Model for Inter- and Intra-processor Concurrency. Ph.D. thesis, Computing Laboratory, University of Kent, Canterbury, UK, Canterbury, Kent, CT2 7NF, United Kingdom (2006). URL http://www.cs.kent.ac.uk/pubs/2006/2429
46. Smith, A.: An Inquiry into the Nature and Causes of the Wealth of Nations, 5 edn. Methuen and Co., Ltd., London (1904). First published: 1776
47. Stepney, S., Welch, P.H., Polack, F., Timmis, J., Barnes, F.R.M., Tyrrell, A.: CoSMoS Home Page (2010). URL http://www.cosmos-research.org/cosmos/. EPSRC EP/E053505/1 and EP/E049419/1
48. Stonedahl, F., Wilensky, U.: Finding Forms of Flocking: Evolutionary Search in ABM Parameter-Spaces. In: Proceedings of the MABS workshop at the Ninth International Conference on Autonomous Agents and Multi-Agent Systems, Lecture Notes of Artificial Intelligence (LNCS/LNAI). Springer (2010)
49. Turing, A.M.: Computing Machinery and Intelligence. Mind, New Series **59**(236), 433–460 (1950). URL http://www.jstor.org/stable/2251299
50. Valiant, L.: A Bridging Model for Parallel Computation. In: Communications of the ACM, vol. 33 (8), pp. 103 – 111. ACM Press (1990)
51. Welch, P.H.: An occam-π Quick Reference Guide. Programming Languages and Systems Research Group, University of Kent, https://www.cs.kent.ac.uk/research/groups/plas/wiki/OccamPiReference/ (2011)
52. Welch, P.H., Barnes, F.R.M.: Communicating Mobile Processes: introducing occam-π. In: A. Abdallah, C. Jones, J. Sanders (eds.) 25 Years of CSP, *Lecture Notes in Computer Science*, vol. 3525, pp. 175–210. Springer Verlag (2005)
53. Welch, P.H., Barnes, F.R.M.: Mobile Barriers for occam-π: Semantics, Implementation and Application. In: J. Broenink, H. Roebbers, J. Sunter, P. Welch, D. Wood (eds.) Communicating Process Architectures 2005, *Concurrent Systems Engineering Series*, vol. 63, WoTUG-28, pp. 289–316. IOS Press, Amsterdam, The Netherlands (2005). ISBN: 1-58603-561-4
54. Welch, P.H., Barnes, F.R.M.: A CSP Model for Mobile Channels. In: Communicating Process Architectures 2008, *Concurrent Systems Engineering Series*, vol. 66, WoTUG-31, pp. 17–33. IOS Press, Amsterdam, The Netherlands (2008). ISBN: 978-1-58603-907-3
55. Welch, P.H., Brown, N.C.C.: The JCSP (CSP for Java) Home Page (2011). http://www.cs.kent.ac.uk/projects/ofa/jcsp/
56. Welch, P.H., Brown, N.C.C., Moores, J., Chalmers, K., Sputh, B.H.C.: Integrating and Extending JCSP. In: A.A. McEwan, S. Schneider, W. Ifill, P. Welch (eds.) Communicating Process Architectures 2007, *Concurrent Systems Engineering Series*, vol. 65, pp. 349–370. IOS Press, Amsterdam, The Netherlands (2007). ISBN: 978-1-58603-767-3
57. Welch, P.H., Brown, N.C.C., Moores, J., Chalmers, K., Sputh, B.H.C.: Alting barriers: synchronisation with choice in Java using CSP. Concurrency and Computation: Practice and Experience **22**, 1049–1062 (2010)
58. Welch, P.H., Pedersen, J.B.: Santa claus: Formal analysis of a process-oriented solution. ACM Trans. Program. Lang. Syst. **32**(4), 14:1–14:37 (2010). URL http://doi.acm.org/10.1145/1734206.1734211
59. Welch, P.H., Pedersen, J.B., Barnes, F.R.M., Ritson, C.G., Brown, N.C.: Adding Formal Verification to occam-π. In: Communicating Process Architectures 2011, *Concurrent Systems Engineering Series*, vol. 68, WoTUG-33, pp. 379–379. IOS Press, Amsterdam, The Netherlands (2011). URL http://www.wotug.org/papers/CPA-2011/Welch11/Welch11-slides.pdf. ISBN: 978-1-60750-773-4
60. Welch, P.H., Sampson, A.T., Wood, D.C.: "To Boldly Go: an occam-π mission to engineer emergence" – supplementary materials. Programming Languages and Systems Research Group, University of Kent, http://projects.cs.kent.ac.uk/projects/naco-boldly/ (2011)
61. Werfel, J., Bar-Yam, Y., Rus, D., Nagpal, R.: Distributed Construction by Mobile Robots with Enhanced Building Blocks. In: International Conference on Robotics and Automation. IEEE Press (2006)
62. Wikipedia: DARPA Grand Challenge (2010). URL http://en.wikipedia.org/wiki/DARPA_Grand_Challenge
63. Yang, Y., Souissi, S., Défago, X., Takizawa, M.: Fault–tolerant Flocking for a Group of Autonomous Mobile Robots. Journal of Systems and Software **84**(1), 29–36 (2011)