Object Store Based Simulation Interworking

Carl G. RITSON [a], Paul S. ANDREWS [b] and Adam T. SAMPSON [c]
[a] *School of Computing, University of Kent*
[b] *Department of Computer Science, University of York*
[c] *Institute of Arts, Media and Computer Games, University of Abertay Dundee*

Abstract. The CoSMoS project is building generic modelling tools and simulation techniques for complex systems. As part of this project a number of simulations have been developed in many programming languages. This paper describes a framework for interconnecting simulation components written in different programming languages. These simulation components are synchronised and coupled using a shared object space. This approach allows us to combine highly concurrent agent-based simulations written in occam-π, with visualisation and analysis components written in flexible scripting languages such as Python and domain specific languages such as MATLAB.

Keywords. complexity, CoSMoS, flocking, occam-pi, Python

Introduction

The CoSMoS project[1] is investigating techniques for building and using computer simulations of complex systems to enable scientific research. Ultimately, we envisage such simulations to be used as tools to support theory exploration, hypothesis generation, and design of real-world experimentation. *Complex systems* is a term often used to describe real-world phenomena that display behaviours that are not obviously deducible as the combination of the behaviours of the individual system components. Consequently, complex systems are synonymous with the property of *emergence*.

We have focussed the majority of our simulation efforts on the technique of *agent-based simulation* (ABS). Components of the system under study (the agents) are represented explicitly in the simulation as separate computational units. Populations of these agents then interact with each other in a programmed environment. A classic example of a complex system behaviour is bird flocking whereby the individual behaviour of birds flying together can result in the emergence of a flock at the population level. Reynold's [1] created an ABS of bird flocking called *boids*, which has been used by the CoSMoS project to investigate various complex systems modelling and simulation issue [2,3].

A major focus of CoSMoS is to develop a simulation framework for running highly-concurrent and parallel agent-based simulations. As a step towards this goal, we have previously proposed a prototype technique for integrating simulations written in different programming languages [4], which we will refer to as the *CoSMoS driver*. This paper builds on this initial work, describing a specific implementation that allows us to integrate our bird-flocking simulation written in occam-pi with visualisation and analysis tools written in Python.

[1] www.cosmos-research.org

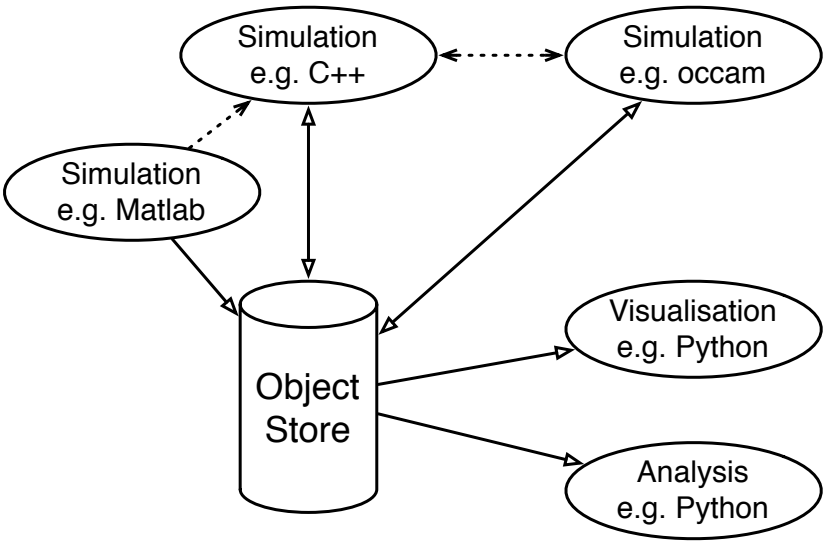

Figure 1. Example connectivity through the CoSMoS Driver Object Store. Solid lines show connections and dashed lines data interactions betweens simulations mediated by the driver.

1. CoSMoS Driver

The CoSMoS driver presented in [4] proposes a prototype infrastructure for creating multilingual simulations. A shared store of simulation objects allows simulation entities to be shared amongst any number of different components of the simulation such as visualisation, interaction and analysis. Objects in the store are created, destroyed and updated by simulation components (such as ABS), while other components can query the object store for information about individual objects. As the object store is external to the different components, it can serve any language for which a client interface has been written (in this paper we describe will details of interfaces for occam-pi and Python). Figure 1 shows how simulation components might interact.

Conceptually, the CoSMoS driver object store combines properties of a database and a publish/subscribe system and allows each component to be written in the programming language of choice, ranging from object-oriented and process-oriented to scripting and specialised domain specific languages such as MATLAB or R. We have outlined our motivations for wishing to construct multilingual simulations in [4], which are summarised here:

Work-flow : *in silico* experimentation with an ABS does not lend itself well to a single programming language or paradigm. In the majority of cases we want to run a core simulation with or without a visualisation, and to capture data for immediate or post-simulation analysis. Ideally we would like to be able to choose the most appropriate languages for the different components of a our simulations, hopefully leading to increased productivity and maintainability. Without the ability to integrate different languages easily into a simulation work-flow, language choice can be a compromise.

Prototyping : we often want to develop prototypes of our simulations to understand and explore the effectiveness of our models and to ascertain whether the time and effort of engineering a high-performance simulation is justified. A multilingual simulation framework would allow us to develop analysis and visualisation tools in parallel with any prototype simulation, and then reuse these components in final simulations. Whilst saving time and effort, this would also allow us to compare the results of the prototype and final simulation as a comparison check. This would also allow us easily to swap out different versions of the simulation model and compare between them.

Accessibility : we often wish to make simulation more accessible to those who are not expert programmers. Many scientists are highly skilled in the use of domain specific lan-

guages as MATLAB or R, but they do not have skills or experience to develop detailed simulations. A multilingual framework such as the CoSMoS driver could allow such scientists to interact with simulation without the need to program in high-performance simulation languages like occam-π.

As previously noted, the CoSMoS driver object store shares many elements with a database and a publish/subscribe system. Wherever possible our explorations draw heavily on existing research and implementations. The architecture adopts aspects of the Linda tuple-space, such as a global name space and temporal elements [5], and the underlying data model is serialised to a type-independent key-value store [6]. While existing systems for distributed data interchange (such as the Apache Project's Hadoop [7]) fulfill many of the requirements we describe, the dependency graph along with setup and maintenance burdens is typically too high for the environment we are targeting. We are striving for a solution with no dependencies or embedded assumptions about the computational model in use.

In this paper, we describe a newer prototype of the CoSMoS driver which improves on that described in [4]. The following subsections describe the significant changes.

1.1. Distributed Simulation

The previous CoSMoS Driver prototype was a Python application in to which plugins were loaded. Operating system pipes were used to connect a single external simulation which created and queried data. Visualisations and analysis components were Python code loaded into and synchronised with the driver.

The version detailed in this work uses network sockets via which all components connect. There is no limit on the number of simulations or types of component that can connect and interact via the driver. Data consistency is maintained as objects are owned by a single component at a time and may only be updated from that component. Objects can be read by any component at any time.

1.2. Time Travel

The previous CoSMoS Driver prototype did not store data, it maintained a data set for the present time step only of the simulation. This prevented exhaustive post-analysis of entire simulation runs such those explained in section 3.

The version used in this paper stores all data changes and provides access to all versions of an object's state. This allows a component to read all simulation data and perform complete analysis on it. Previous to this, analysis components would need to maintain their own copy of data by recording it with each time step.

Additionally, by prefixing object names with a simulation run name, multiple versions of a dataset can be stored. This provides for multiple runs of the same simulation (for averaging), and multiple runs with different parameters (parameter space exploration). A component accessing the driver can perform visualisation or analysis over all these datasets.

2. Driver Interface

This section describes the driver interface and its implementation in occam-π.

2.1. Protocol

The CoSMoS driver interface runs over standard network sockets and uses a simple record based binary protocol. Where objects, their field names and data are untyped binary data. The protocol has eight requests:

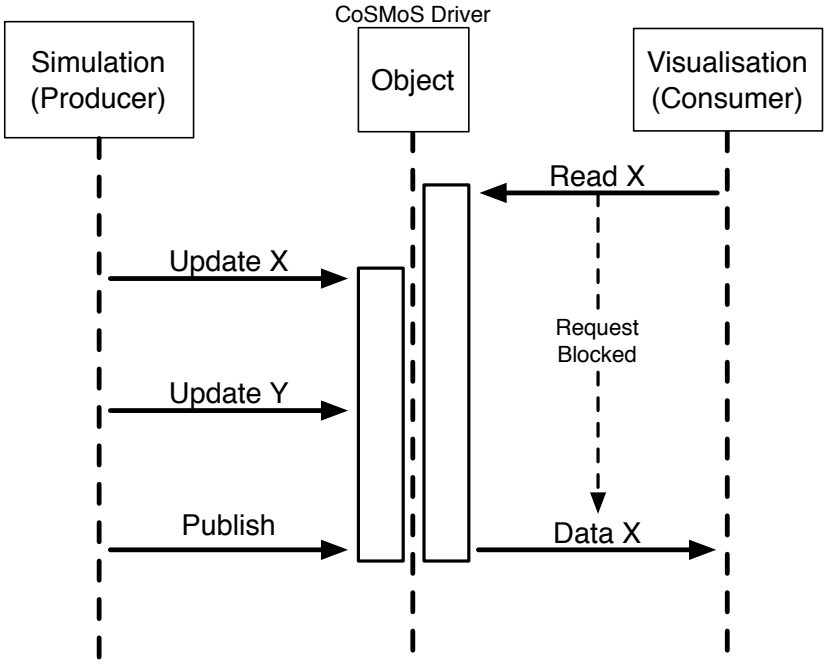

Figure 2. Example of interaction between two simulation components via the CoSMoS Driver. A read request for field X in the object is blocked until an update is published against it.

create an object (start object timeline)
delete an object (end object timeline)
update an object field with a value
publish pending updates to an object
read an object field at a time (blocking)
test an object field at a time (non-blocking read)
query for objects that match a regular expression
stop an active request

Each request has an identifier and multiple requests can be active simultaneously. This allows batching of requests and internal concurrency in the client. Simulations producing data send a series of update and publish requests (potentially batched). Clients consuming the data send a number of reads and wait for responses (again batching where appropriate). Synchronisation is maintained as reads are blocked until data for the requested read has been published. Figure 2 shows this interaction.

2.2. Process Model

In occam-π a driver process provides access to the CoSMoS driver socket via a client/server protocol over a shared channel bundle. The driver process multiplexes and marshals requests from object proxy processes. Figure 3 shows the network diagram between the driver process and the object proxy processes.

An object proxy process is forked off for each object the simulation is accessing or updating. It stores the object name and adds it to requests, it also acts as a decoder and buffer for responses to requests on the object. Internally an array of response channels is stored by the driver process and responses from the network are forwarded down these to the associated object proxy.

Figures 4 shows the protocol definitions for the driver protocol. The `open` request is used by an object proxy process to register with the driver process, it provides a mobile channel (bundle) on to which network response will be sent. An `opened` response is returned with the specific request identifier for that object proxy (which is the array index of the channel within

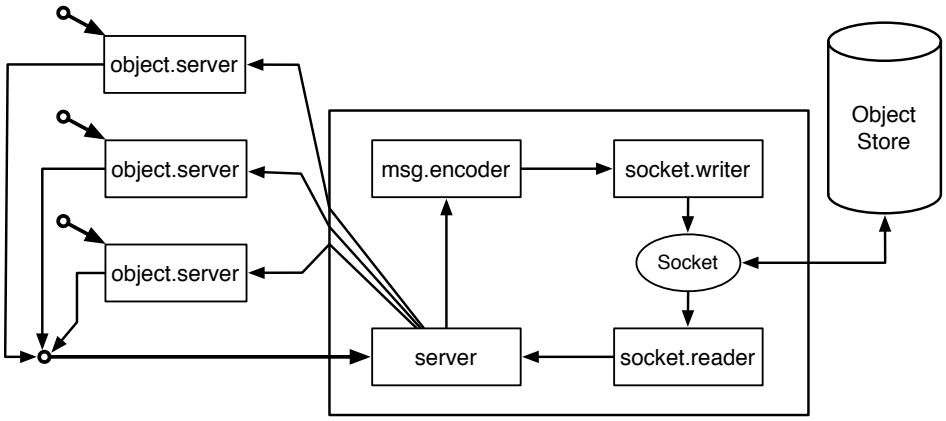

Figure 3. occam-π process network for CoSMoS driver interface. On the left are object proxies (`object.server`) which provide access to objects. The server process multiplexes access to the message encoder (`msg.encoder`) and demultiplexes response back to the object proxies.

```
PROTOCOL DRIVER.REQ
  CASE
    open; DRIVER.RESP.CT!
    close; INT
    create; MOBILE []BYTE; REAL32
    delete; MOBILE []BYTE
    update; MOBILE []BYTE; MOBILE []BYTE; MOBILE []BYTE
    update.int; MOBILE []BYTE; MOBILE []BYTE; INT
    update.real32; MOBILE []BYTE; MOBILE []BYTE; REAL32
    publish; MOBILE []BYTE; REAL32
    read; INT; MOBILE []BYTE; MOBILE []BYTE; REAL32
:

PROTOCOL DRIVER.RESP
  CASE
    ok
    error; MOBILE []BYTE
    opened; INT
    msg; MOBILE []BYTE
:
```

Figure 4. occam-π driver server client/server protocol.

the driver process). The `close` request unregisters an object proxy with the driver process. Object proxies send `create` on startup and `delete`, `update`, `publish` in response to requests on the proxy. The object proxy protocol (Figure 5) is a simplification of the driver protocol.

2.3. Python API

The Python client to the CoSMoS driver is approximately 300 lines of source code and provides a simple synchronous interface through a `SimulationClient` class. Each of the class methods, shown in Figure 6, mirror a request in the driver protocol. No object proxies were used due to the synchronous nature of the client and the ease of manipulating strings and complex data structures in Python.

3. Case Study

We have used our implementation of Reynold's boids flocking as our case-study. The simulation agent's movement is governed by three basic rules:

```
PROTOCOL OBJECT.REQ
  CASE
    delete
    update; MOBILE []BYTE; MOBILE []BYTE
    update.int; MOBILE []BYTE; INT
    update.real32; MOBILE []BYTE; REAL32
    publish; REAL32
    read.int; MOBILE []BYTE; REAL32
    read.real32; MOBILE []BYTE; REAL32
:

PROTOCOL OBJECT.RESP
  CASE
    int; INT
    real32; REAL32
    error; MOBILE []BYTE
:
```

Figure 5. occam-π object proxy client/server protocol.

```
class SimulationClient(SimSocket):
    @classmethod
    def connect(cls, host, port):
    def create(self, obj_id, time):
    def delete(self, obj_id):
    def update(self, obj_id, field, data):
    def publish(self, obj_id, time):
    def read(self, obj_id, field, time):
    def test(self, obj_id, field, time):
    def query(self, regexp):
```

Figure 6. Abstracted class definition for the Python CoSMoS driver client.

Collision Avoidance : avoid collisions with nearby objects
Velocity Matching : try to match velocity with nearby boids
Flock Centring : try to stay close to nearby boids

The emergence of flocking behaviour depends on parameters within the basic rules. Which neighbours are visible to an agent is critical and governed by two parameters: the vision angle and vision radius. Modifying the vision radius (distance an agent can see) directly affects this. Fundamentally if an agent cannot see other agents, it cannot flock with them.

To test the present CoSMoS driver design we have reproduced previous work exploring the effect of the vision radius on the emergence of the flocking behaviour [8]. We use Singular Value Decomposition (SVD) to measure the system entropy [9]. The system entropy provides a description of emergent properties such as flocking behaviour. Generally speaking the system entropy changes as boids interact and flocking behaviour emerges.

To calculate the SVD, a matrix Z is constructed containing all the boid $(1..n)$ positions and velocities, for all time steps $(1..m)$ of the simulation.

$$Z = \begin{bmatrix} \overbrace{P.x_1^1\ P.y_1^1\ V.x_1^1\ V.y_1^1}^{Boid_1} & \cdots & \overbrace{P.x_1^n\ P.y_1^n\ V.x_1^n\ V.y_1^n}^{Boid_n} \\ \vdots & \vdots & \vdots \\ \underbrace{P.x_m^1\ P.y_m^1\ V.x_m^1\ V.y_m^1}_{Boid_1} & \cdots & \underbrace{P.x_m^n\ P.y_m^n\ V.x_m^n\ V.y_m^n}_{Boid_n} \end{bmatrix} \tag{1}$$

```
VAL []AGENT.INFO infos IS [boid.infos FOR num.boids]:
INITIAL MOBILE []BYTE ids IS MOBILE [num.boids * BYTESIN(INT)]BYTE
SEQ
  []INT ids RETYPES [ ids FROM 0 FOR (SIZE ids) ]:
  SEQ i = 0 FOR SIZE infos
    ids[i] := infos[i][id]
  object[req] ! update.real32; "velocity_x"; info[velocity][x]
  object[req] ! update.real32; "velocity_y"; info[velocity][y]
  object[req] ! update; "neighbours"; ids
  object[req] ! publish; time
  time := time + 1.0
```

Figure 7. Source code modification to boid process in order to record position and velocity.

Each boid belongs to a group. The group depends on the analysis being performed. For a global analysis the boids all belong to the same group; however, for a local analysis the grouping will depend on the vision radius. At each time step the mean position and velocity of a each group is calculated.

$$\bar{P}_{group_n} = \frac{\sum_{i \in group_n} P_t^i}{n} \wedge \bar{V}_{group_n} = \frac{\sum_{i \in group_n} V_t^i}{n} \tag{2}$$

The position and velocity of each boid in the matrix is then made relative to the respective group mean at the given time step. Notionally this is done as a simple difference; however, to deal with wrapping of spatial coordinates in the toroidal boids simulation we may use more complex rules.

$$\forall_i Z.P_t^i \leftarrow \bar{P}_t - Z.P_t^i \wedge \forall_i Z.V_t^i \leftarrow \bar{V}_t - V.P_t^i \tag{3}$$

SVD is then computed for a number of time steps (matrix rows).

$$\Sigma_t = \mathrm{SVD}(Z_t) \tag{4}$$

The results are normalised with respect to the sum of the singular values.

$$\Sigma'_t = \frac{\Sigma_t}{(\sum_i \sigma_{i,t})} \text{ where } \sigma_{i,t} \in \Sigma_t \tag{5}$$

Finally the entropy is calculated from the singular values for the time steps. This gives a single entropy value for a range of m time steps, t_i to t_{i+m}.

$$S_t = -\sum_i \sigma'_{i,t}.\log_2(\sigma'_{i,t}) \tag{6}$$

This kind of statistical analysis can be rapid prototyped in Python using existing libraries for scientific and numeric computation (SciPy/NumPy) [10]. The results can also be quickly graphed and visualised using plotting libraries such as *matplotlib* [11].

To generate the data for processing we modified the *occoids* simulation to connect to the CoSMoS driver and report the boids initial position and velocity at each time step. The position of each boid can be calculated incrementally from its initial position and intervening velocities. We also added code to report the neighbours of a boid (other boids within its vision radius) at each time step. The neighbours can be used to construct groupings for doing local SVD [8]. The source code modification in order to capture data were less than 30 lines of code, the key implementation change in the *boid* process can be seen in Figure 7.

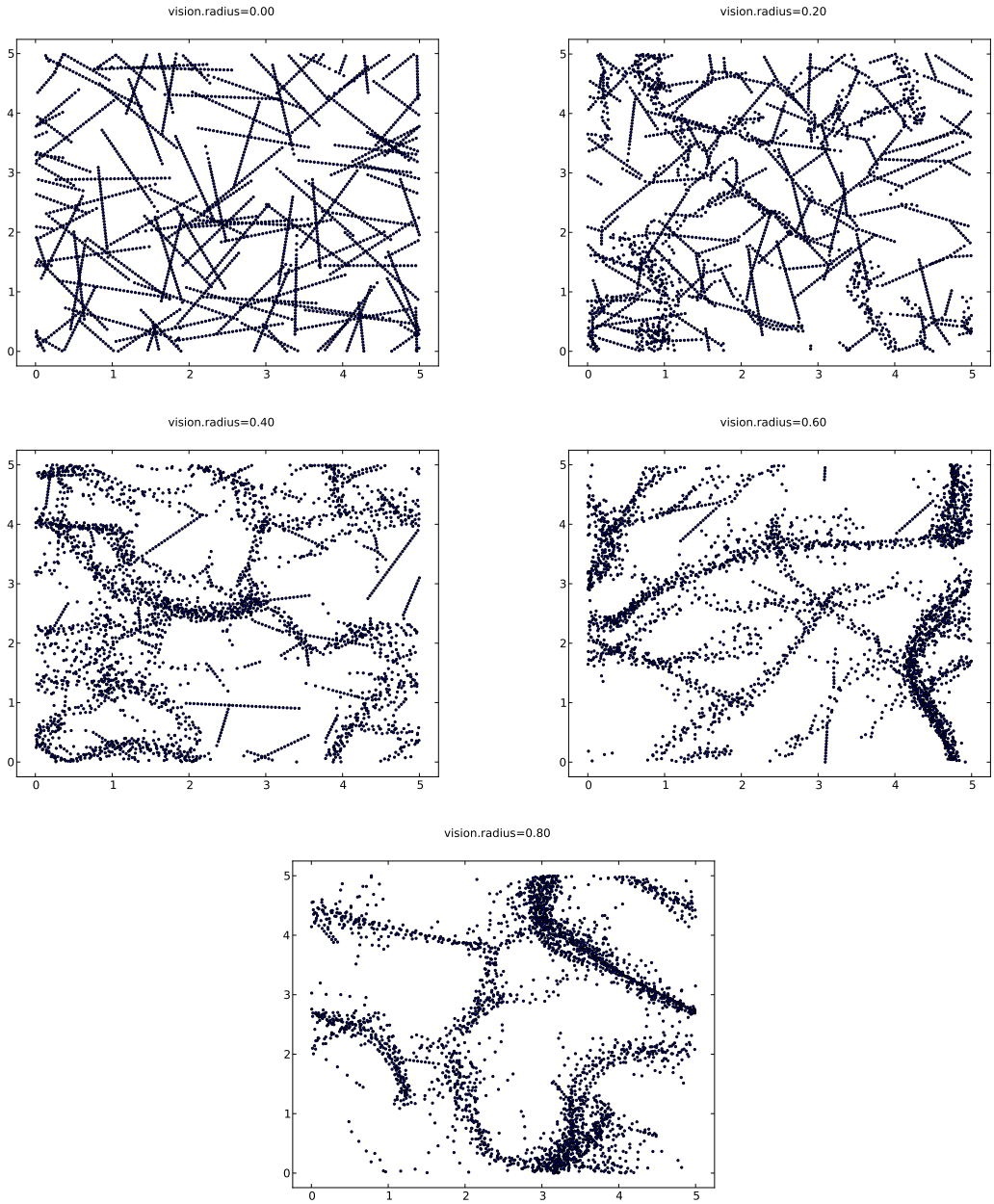

Figure 8. Scatter plots of boid locations over the course of simulation runs varying the vision radius.

3.1. Results

In this section we discuss a selection of results exploring the impact of boids vision radius on their flocking behaviour. The CoSMoS space modelling framework divides simulation space in to a set of locations [2]. A boid's vision radius is expressed relative to the size of locations, continuously from 0.0 to 1.0. For the results presented in this paper we have explored the range $0.0-0.8$ at 0.2 step intervals. Each simulation run uses 100 boids on a 5x5 grid for 300 time steps. The initial position and velocity of each boid is generated by a pseudo-random number generator seeded from the system clock.

Figure 8 shows scatter plots of the boid locations for sample simulation runs. For a vision radius of 0.0, the boids move in straight lines from their initial position maintaining their initial velocities. As they never see any other boids, their motion can never be influenced by them. This is in contrast with plots for vision radii of 0.4, 0.6 and 0.8, in which smooth flight paths can be seen where boids interact and form small flocks. Some straight line paths

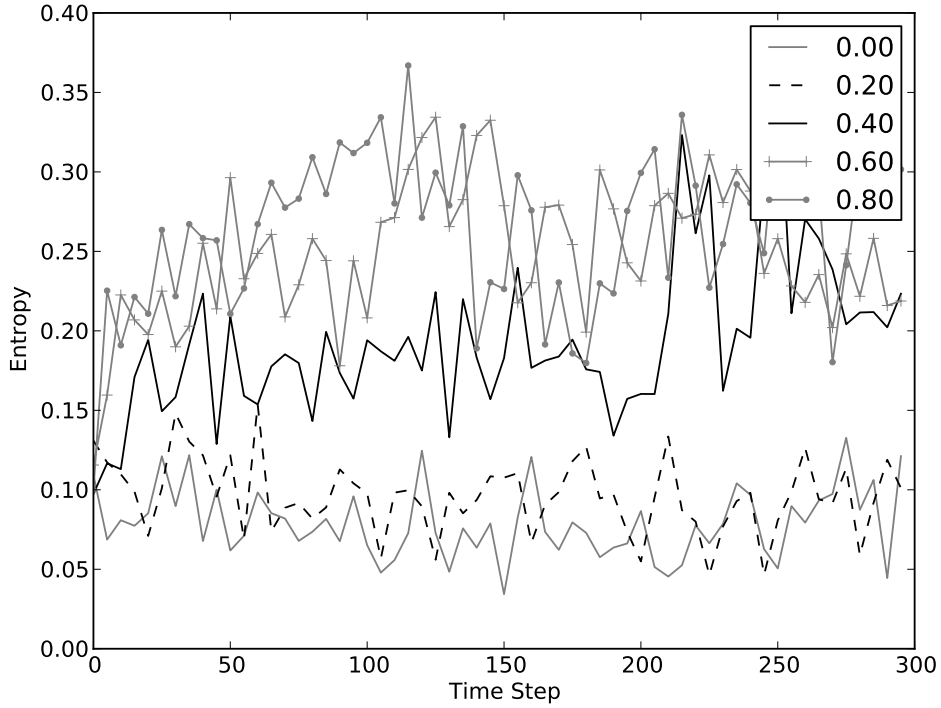

Figure 9. Analysis of boids and velocities with varying vision radius. All boids are treated as the same group (global analysis). Increasing entropy characterises emergence.

remain where a boid has been sufficiently isolated for it to remain uninfluenced by other boids during the simulation time. However as the vision radius is increased the flocking convergence occurs more rapidly and the number of straight line paths tends to zero.

Figure 9 shows the SVD derived system entropy plotted for a number of different vision radii. Boids are all treated as a single group. As the vision radius is increased the system entropy increases observably. In the case where boids can not see each other they move consistently in a single direction, so their mean position and velocity is fairly static and entropy is low. Therefore the entropy is seen to increase as the vision radius increases. However, there is a significant jump in system entropy levels when the vision radius moves between 0.2 and 0.4, this suggests there is a tipping point for the flocking behaviour in between these values.

Local SVD analysis where boids are grouped with respect to their neighbours (boids in their vision radius and vision arc) is shown in Figure 10. In this case the system entropy is seen to decrease with increasing vision radii. More specifically system entropy decreases as flocking occurs. This is a result of a boid's position and velocity becoming more uniform with respect to its neighbours as flocking occurs. The results for a vision radius of 0.4 show this convergence as the flocking behaviour builds momentum after time step 100. For a radius of 0.2 flocking barely occurs (evidenced by Figure 8), whereas for 0.6 and 0.8 flocking occurs almost immediately.

It has been suggested that emergent properties are characterised by abrupt changes in system entropy in response to gradual small changes in system parameters [9]. These results for flocking appear to fit with this definition.

4. Conclusions

We have demonstrated the integration of an analysis component written in Python with agent based complex systems simulation written in occam-π. This approach combines the benefits

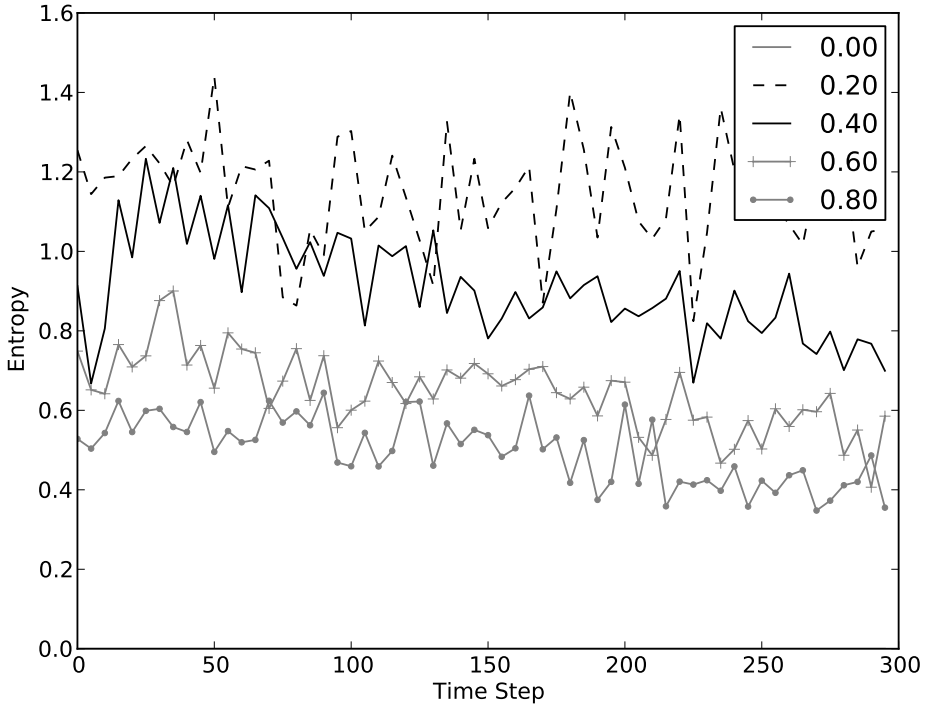

Figure 10. Analysis of boids and velocities with varying vision radius. Boids are grouped by the neighbours they can see. Decreasing entropy characterises emergence. No values are present for a vision radius of 0.0 as it is not possible to characterise the entropy of agents those position and velocity are constant with respect to their grouping (themselves).

of both, rapid and flexible prototyping in Python using existing libraries for numeric computation with a high degree of concurrent expression in occam-π. Furthermore the analysis code produced can be applied to a range of spatial oriented simulations which can be interfaced with the CoSMoS driver through minimal modification.

5. Future Work

We plan to continue prototyping and developing the CoSMoS driver. In this section we discuss specific issues arising from this work.

Performance is a continuing issue with our present CoSMoS driver server prototype. Specifically the internal data structures within the server appear to swell with large numbers of data sets, even though the bulk of this data is stored on disk. To overcome this we intend to rewrite the server in a compiled language and use specific designed in-memory data structures to help.

Request handling is another for performance improvement. Our analysis component requests almost all of the data stored within the driver, this generates many millions of requests. Where possible these requests are batched together so further requests are not waiting on responses yet to arrive. Network connectivity is not an issue as the server and client are running on the same computer system. Still we see a bottlenecking effect with requests which return data (read, query) compared to those which just store data (update, publish). Further work is needed to investigate whether this is to do with the overheads of network sockets, or simply poor performance in our server implementation. Using a shared memory or memory mapped interface between the client and server is another area for exploration.

Acknowledgements

This work is part of the CoSMoS project, funded by EPSRC grants EP/E053505/1 and EP/E049419/1.

References

[1] Craig W. Reynolds. Flocks, herds and schools: A distributed behavioral model. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 25–34, New York, NY, USA, 1987. ACM.

[2] Paul S. Andrews, Adam T. Sampson, John Markus Bjørndalen, Susan Stepney, Jon Timmis, Douglas N. Warren, and Peter H. Welch. Investigating patterns for the process-oriented modelling and simulation of space in complex systems. In S. Bullock, J. Noble, R. Watson, and M. A. Bedau, editors, *Artificial Life XI: Proceedings of the Eleventh International Conference on the Simulation and Synthesis of Living Systems*, pages 17–24. MIT Press, Cambridge, MA, August 2008.

[3] Adam T. Sampson, John Markus Bjørndalen, and Paul S. Andrews. Birds on the wall: Distributing a process-oriented simulation. In *2009 IEEE Congress on Evolutionary Computation (CEC 2009)*, pages 225–231. IEEE Press, May 2009.

[4] Adam T. Sampson and Paul S. Andrews. The best of most worlds: Shared objects for multilingual simulation. In *Proceedings of Parallel/High-Performance Object-Oriented Scientific Computing (POOSC'10)*, October 2010. to appear.

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

[6] Tokyo cabinet: a modern implementation of DBM. `http://fallabs.com/tokyocabinet/`.

[7] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[8] Peter Nash. A simulation of a complex system using occam-π: an investigation of emergent behaviour and entropy. Master's thesis, University of York, 2008.

[9] W.A. Wright, R.E. Smith, M. Danek, and P. Greenway. A generalisable measure of self-organisation and emergence. In *Proceedings of the International Conference on Artificial Neural Networks*, ICANN '01, pages 857–864, London, UK, 2001. Springer-Verlag.

[10] SciPy: Scientific Tools for Python. `http://www.scipy.org`.

[11] Python 2D plotting library. `http://matplotlib.sourceforge.net/`.