The Best of Most Worlds

Shared Objects for Multilingual Simulation

Adam T. Sampson
University of Abertay Dundee, UK
ats@offog.org

Paul S. Andrews
University of York, UK
psa@cs.york.ac.uk

Abstract

Computing techniques are increasingly being used in scientific research to tackle a diverse set of problems. An example is complex systems research, which focuses on the use of computer simulations to explore, understand and describe the real-world system under study. These simulations are often sophisticated pieces of software with numerous design trade-offs between performance and ease of development and use. We propose a simulation framework for complex systems simulation that allows each component of a simulation—for example visualisation, or data analysis—to be developed in the most appropriate language. The framework uses the concept of shared objects to communicate data between simulation components. We present here a detailed motivation for multilingual simulations, an outline design and prototype for the simulation framework, and discuss future plans for the framework.

1. Introduction

Numerous computing techniques exist to help scientists tackle a wide variety of research problems. One technique that is becoming increasingly popular in *complex systems* research is computer *simulation*. Complex systems are characterised by elaborate behaviours at the system level that are a consequence of the more simple behaviours of the system components. Importantly, the high-level system behaviours are not obviously deducible as the sum of the low-level component behaviours. This property is often described in complex systems research (complexity science) as *emergence*.

Complex systems simulation is used to explore, understand or describe emergent system behaviours. Typically this involves simulating real-world complex phenomena from any number of scientific disciplines including biology, chemistry, physics and sociology. Complex systems simulation may also be applied to investigate complex systems themes and behaviours that transcend any specific subject, for example the property of self-organisation.

One popular approach to constructing simulations of complex systems is *agent-based simulation* (ABS). To construct an ABS, components of the real-world system are first modelled as (populations of) individuals (the agents) that interact with each other in an environment. In the ABS, the agents are expressed as separate computational entities within an encoding of the identified environment. A classic example is the simulation of bird flocking. Here each agent is a simulated bird that adjusts its behaviour depending on neighbouring bird agents and environmental conditions such as the weather. Execution of the ABS aims to generate the system-level emergent behaviours that are the object of study, in this case, a bird flock.

The CoSMoS project[1] is working towards developing tools and techniques to assist the simulation of complex systems. Specifically, the project aims to develop two main outputs: an agile process to capture best-practice in simulation construction and use [1]; and a simulation framework for running highly-concurrent and parallel agent-based simulations. This latter output forms the subject for this paper, which examines how shared object data can be used to construct simulations that exploit the relative advantages of different programming models and languages. In order to achieve our aims, we are working with real scientists on a number of complex systems case studies to ensure that our approach is useful on real-world systems. Examples of the case studies we are working are summarised in [4], and include numerous immune systems investigations, plant growth, social modelling, and electricity networks.

This paper is concerned with the complex systems subset of scientific computing and explores the rationale for wanting multilingual simulations. Using concepts from object-orientation (OO) we can produce scientific simulations that use any number of programming languages, both OO languages and others. Here we use both process-oriented and OO languages. The paper describes ongoing work to develop

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

a generic framework uses shared objects to achieve communication between different components of the simulation, which can be implemented in any suitable programming language. Using our numerous case-studies we have been exploring general requirements for complex systems simulations. The desire for multilingual simulations has arisen from developing these case studies and wishing to use the most appropriate tools for the job.

The paper is organised as follows: section 2 examines our motivation for wanting to develop multilingual simulation; section 3 outlines a general design for our simulation framework, and then describes and critiques our current prototype implementation; section 4 outlines the benefit of being able to develop multiple implementations of complex systems simulations; and section 5 concludes the paper by exploring the future directions of our simulation framework.

2. Motivation

Simulators to explore complex systems are most often developed by individuals or small teams as bespoke tools that attempt to address a specific set of research questions. The simulators are used to perform *in-silico* experimentation that provide insight into the real-world complex systems domain.

Like many pieces of complicated software, the simulations we develop to study complex systems consist of numerous components. The *core simulation* represents the model of the science of which we are interested. In the majority of our case studies, this is implemented as an ABS (outlined above). Other simulation components allow us to visualise and interact with the core simulation. For example a graphical user interface can contain any number of visualisations of executing ABS and an analysis of the data it generates. We also need to interact to configure the core simulation both on initialisation of the simulation and when changing parameters during execution. Another major component of complex systems simulations is the extraction of different types of data that describe its execution. This data (the simulation results) can be used to analyse the behaviour of the simulation and subsequently draw conclusions about the underlying scientific system in the real world.

Analysis of simulation data can either happen whilst the simulation is running, but is more often reserved for a post-simulation activity Agent-based simulations often make use of stochastic techniques, with behaviours expressed using weighted random choices. Consequently, in-silico simulation involves the collection of data from multiple simulation runs to build up a statistical picture of the overall behaviour of the system. Coupled with the fact that these models are heavily parameterised, exploration of the simulation's behaviour may involve thousands of individual runs. These runs are independent so we can make use of clusters to achieve parallelism (an *embarrassingly parallel* problem), and achieve results in a sensible time frame.

The different simulation components allow us to interact with the simulation in different ways, at different stages of our research. Three of the main ways we interact are:

Development: many of our simulations are representative of systems that have spatial and temporal dimensions that make them open to visualisation. During development, visualisation is a natural and time saving debugging tool. As we are interested in emergent behaviour that results from the interaction of numerous agents in complex sytems, small errors in agent behaviour can produce macroscopic errors in the system as a whole during runtime. Being able to see both the individual agent behaviours and macroscopic behaviours in a single visualisation highlights such errors.

Interactive investigation: simulations are often highly configurable with a vast space of possible parameters and solutions. Interacting with simulations (for example changing parameters on the fly) can help explore this parameter space and narrow the space of solutions to be explored. Such simulation runs are *explorative* and help us gain knowledge of the system through visual feedback.

Batch experimentation: in the majority of instances working with simulation, we wish to create a statistical picture of certain behaviours to present as results that can then inform our scientific investigations with respect to the real domain. This requires numerous simulation runs to generate the necessary data. The types of complex systems simulation we deal with often include stochasitc elements increasing the requirement for batch experimentation. The key to these types of experiments is that they are run without a GUI, but concentrate on generating data. This data must be collected and suitably analysed.

Programming language choice for complex systems simulations depends on numerous factors. Often we rely on a single choice of language to develop all simulation components from the ground up with little code reuse. The choice of language may be a general-purpose programming language such as Java, Python, Fortran or C++, or it may be a language with features designed specifically to support simulation, such as NetLogo or the stochastic π-calculus. Generic simulation environments and libraries also exist for a number of general purpose languages that support the rapid development of simulations. Examples include the Mason libraries for Java and the Breve simulation environment.

In other circumstances, we might employ more than one language in an ad hoc simulation pipeline, especially in the case of batch experimentation. Here the simulation core is written in a general purpose or domain specific language, and a scripting or dynamic language (shell script, Perl, Python, Ruby etc.) used to control multiple simulation runs and collection of data. This data is then analysed with these languages or other languages designed for processing such data such as Matlab and R.

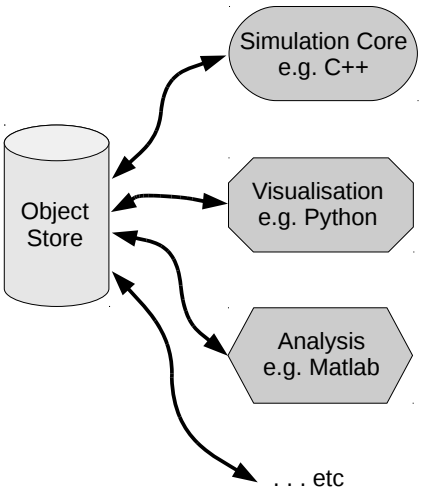

Figure 1. Object store

It is obvious that different languages have different strengths and weaknesses; C++ and Fortran offer high performance for numerical operations, whereas Python, Perl and Ruby are expressive and have great IO capabilities enabling more rapid development. Different languages also have differing levels of support for libraries and interfacing with other tools.

Owing to the differing component requirements of our simulations it is infeasible to select a single language that is equally good for each simulation component. Without access to tools that make developing multilingual simulations a reality, the choice of languages for developing simulations may depending on the developers preference for a particular language or the language that offers the best compromise.

Ultimately, it is our intention, that we should be able to choose the most appropriate languages for the different components that our simulations will entail. This should lead to increased productivity and a move away from the ad hoc ways we have mixed languages in simulation design to date.

3. A Simulation Framework

3.1 Proposed Design

We propose that simulations should be built to communicate with a shared store of *simulation objects*: entities within the simulation that are of interest for interaction, visualisation or analysis, such as agents or environmental properties (figure 1). Simulations can create and destroy objects in the store, and set their properties. Other tools can query the store for information about objects, extracting the information needed for analysis and visualisation. While the store only holds the current state of the objects—not their history—it is aware of the virtual time within the simulation, allowing clients to obtain a consistent view of the state of the simulation.

The store therefore combines properties of a database and a publish/subscribe system: it can be thought of as a conduit for information about simulation objects that flows out of the simulation and into other tools. Objects have unique identifiers, allowing references between objects. The structure of objects within the store may be entirely dynamic, or make use of datatypes defined ahead of time; the latter would allow more efficient, nuanced queries of the store (e.g. "find all the pine trees"), although techniques developed for NoSQL databases may be applicable.

Since the store is external to the simulation, it can serve simulation and analysis clients written in any language; it is only necessary to provide a client library for the desired language. As the majority of programming languages provide facilities for binding to native libraries, the easiest approach is to write a C or C++ client library, then write thin wrappers around that for target languages—although greater efficiency or convenience may be achieveable in some languages using a "pure" approach without a native library.

We believe that this approach can make simulations more accessible to users who are not expert programmers. A scientific simulation exists to serve the purposes of a *domain expert*: a scientist with expertise in the system under study. While domain experts are often highly skilled in the use of specialised data analysis tools such as MATLAB or R, most have little or no experience of software development [3]. Using a multilingual framework for simulation allows scientists to interact with and extract data from their simulations without needing to be able to program in the languages normally used for high-performance simulation. They can instead interact with the system using simpler scripting languages (such as Python), or directly using the tools with which they are already familiar (such as MATLAB)—removing a significant barrier to entry.

The separation of the objects in the store from the components that read and modify them is essentially an implementation of the Model-View-Controller pattern. It allows us to view the results of the simulation in several different ways concurrently by using multiple visualisations—or to view different simulations through the same visualisation. We will explore the implications of this later. The model in this case is not the *platform model* that the simulation is an implementation of, but a data model that only contains the output from the simulation.

3.2 Prototype Implementation

At present, the prototype implementation of our framework allows us to integrate simulations written in occam-π with visualisation and analysis components written in Python. We have used both languages in a number of CoSMoS case studies.

occam-π is a programming language based on the principles of CSP and the π-calculus [7]. occam-π's excellent support for safe, lightweight, message-passing concurrency and its highly-efficient multicore runtime system make

it straightforward to build agent-based simulations where agents are implemented directly as lightweight processes. Such *process-oriented* simulations automatically make good use of modern multicore processors, and can be distributed over clusters of machines [5, 6]. However, occam-π's limited set of data types and lack of bindings for external libraries makes data analysis and visualisation more complex than in other languages.

Python, on the other hand, is a "scripting" language: it trades lower performance for greater expressivity and ease of interfacing to other tools and libraries. Python's dynamic type system and well-stocked standard library allows programming in imperative, object-oriented and functional styles using sophisticated data structures. Furthermore, there are a wide range of Python add-on modules that are useful for scientific computing: for example, SciPy and NumPy provide high-performance mathematical operations based upon bindings to standard libraries such as LINPACK, matplotlib allows interactive plotting of data, and RPy allows direct interfacing to the R statistical computing system.

The prototype provides a simple, unstructured store of simulation objects, implemented in Python (figure 2). Each object is represented as a Python dictionary; the only structure imposed is that all objects must have an `id` field containing their unique identifier, and a `type` field containing a string (such as `tree`). The simulation sends messages to the store across a pipe in order to update the object database according to a simple protocol:

Register: add a simulation object to the store, taking a type and (optional) identifier;

Unregister: remove a simulation object from the store;

Attribute: add or update an attribute on an object;

Timestep: advance the current virtual time in the simulation.

Visualisation and analysis components are currently written as Python plugin modules that are loaded by the store. Any number of components (including zero) can be supplied. Each component is notified when the timestep changes, and has access to a consistent snapshot of the objects in the store as if they were Python objects—for example, an object `obj`'s type is accessible as `obj.type`. Components can retrieve an object with a particular identifier, iterate over all objects, or iterate over objects of a specific type. The simulation can continue to run while components analyse the data from the previous timestep, with property changes only becoming visible once the analysis for the next timestep starts.

The occam-π client interface makes use of a number of concurrent processes to allow object property encoding to be parallelised (figure 3). A simulation typically contains many agent processes, representing entities in the simulation. For each agent that wishes to appear as a simulation object in the store, a corresponding *object server* is created; local variables are bound into object properties using occam-π's

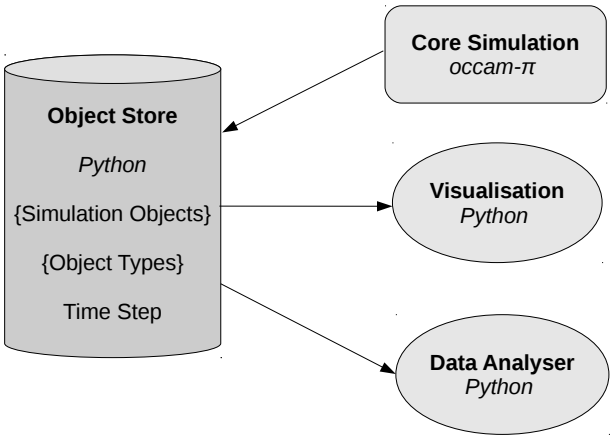

Figure 2. Current design of simulation framework

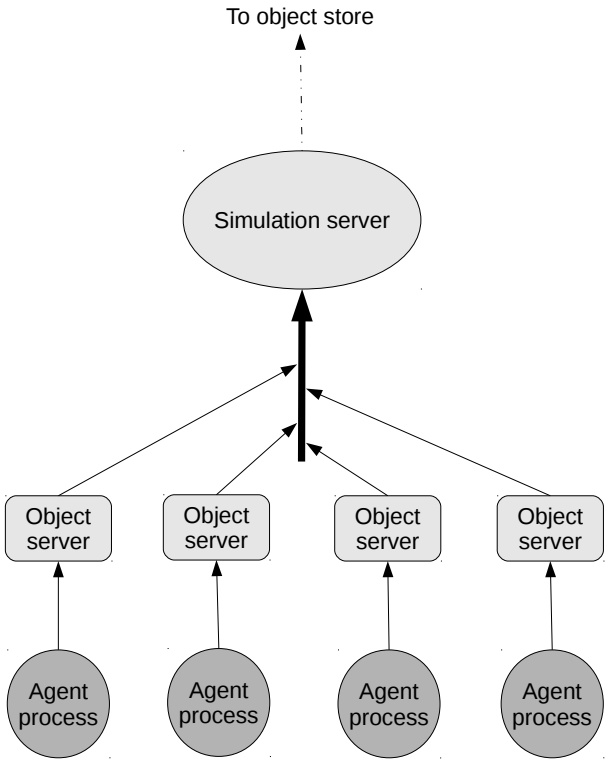

Figure 3. occam-π interface to the object store

abbreviation system. This encodes property values from the agent into store protocol messages, and passes them to the *simulation server* over a shared communication channel, which batches them together and sends them over the pipe to the store. The simulation server engages in the simulation's virtual time mechanism, and inserts timestep messages into the data stream at the appropriate points.

3.3 Evaluation

We have used the prototype framework successfully in a real-world scientific simulation of granuloma formation in the liver, with a high-performance simulation in occam-π coupled to analysis components in Python. We used PyGame—which provides Python bindings for the SDL cross-platform graphics library—to produce a live visualisation, with the visualisation code being significantly shorter and simpler than a previous attempt at an occam-π visualisation. During this work we have identified some shortcomings with the prototype.

The simulation includes many thousands of simulation objects representing segments of blood vessel, all of which report state changes to the store on each timestep. Profiling shows that the store is actually the performance bottleneck here: the message parsing code, written in Python, takes up the bulk of the time in the simulation as a whole. This is obviously not ideal, since we would normally expect the computation-heavy simulation to be the bottleneck; we would also prefer that the simulation programmer not have to worry about how often their results are made visible to the store.

The unstructured nature of the object store makes it very simple to put data in the store, but complicates the process of retrieving results for analysis: a visualisation component may retrieve an object without knowing exactly which fields it includes. We could make objects return a dummy value for requested fields that have not yet been provided, but this would only push the problem further down the chain; instead, we could require the simulation author to specify the *results model* for the simulation explicitly, which would not be especially arduous, and would offer advantages for documentation and debugging too.

4. Multiple Implementations

Engineering a correct, high-performance simulation of a complex system often involves a considerable amount of work. Before investing time in creating a final version of a simulation, it is often useful to write a quick prototype, which can be used to verify that the model has the desired properties, and to perform initial calibration of parameter values. In agile software engineering terms, this kind of proof-of-concept prototype is a *spike solution*.

As performance is not a great concern for a prototype, an expressive language such as Python can be used, allowing rapid development and easy modification of the prototype. Developing a prototype often eases the development of a final implementation by allowing multiple implementation approaches to be explored. The prototype may even make the development of the final simulation unnecessary—for example, if a fatal flaw is found in the model, or if the performance of the prototype is shown, after testing, to be adequate for the required experimentation.

Using our simulation framework, a prototype simulation can be treated in exactly the same way as the final simulation: analysis and visualisation tools can be developed in parallel with the prototype, and then used with the final simulation. This allows direct comparison between the results from the prototype and the final simulation, giving the developer additional confidence in the correctness of their simulation.

This kind of validation between multiple simulations is also useful when modifying an existing simulation—for example, when updating it to match a new version of the underlying model, or when reimplementing it using a different language or environment. Deliberately implementing a simulation using different approaches can help to reveal underspecified aspects of the underlying model [2].

In some cases, it may even be possible to feed real-world data into the framework, allowing direct comparison between the simulation and the original domain. This is only possible when the data in question exists in the same form in the domain and the simulation; that is, the model is not too far abstracted from the domain.

5. Future Work

In the near future, we plan to develop our prototype framework into a reliable, easy-to-use substrate for the future development of complex systems simulations. Projects that will make use of the framework over the next few months cover a variety of areas including plant biology, immunology, sociology and electrical engineering. The framework will be developed and maintained as a sustainable open-source project under the aegis of CoSMoS.

The biggest problem with the prototype is its poor performance in the face of very large numbers of simulation objects. This will be addressed by the use of existing technologies for efficient data storage and communication, and by the implementation of smart client- and server-side filtering techniques to minimise the amount of data sent while avoiding round-trips in network communication—the greatest performance concern for distributed simulations. These filtering techniques could make use of fuzzy, predictive approaches to estimate which data will be required by which simulation components.

We also need to provide client bindings to the system for more languages, including all those used in our existing simulations. These will take advantage of language idioms where appropriate. In OO languages such as Java and Python, annotations and reflection could be used to allow properties of simulation objects to be easily exported. In concurrent languages such as occam-π, a message-passing interface could be provided to allow an efficient, event-driven programming style for simulation components.

Looking further ahead, the framework could offer a number of more sophisticated facilities:

Distributed simulation: in its simplest sense, the framework could provide a communication mechanism for dis-

tributing a single simulation across a cluster, subsuming the technologies already developed for this in CoSMoS. However, it could also allow co-simulation, where different parts of a simulation are simulated using different tools, or multiscale simulation, where different parts of the system are modelled at different scales in space or time. The framework would allow consistent control, visualisation and analysis of a distributed simulation.

Cluster integration: the framework could be integrated with cluster management or cloud computing systems, allowing the automatic scheduling and management of distributed simulations. In cases where many simulations must be run with different parameter settings—for example, when performing sensitivity analysis—the framework could launch simulations and collate results automatically.

Property manipulation: it would be straightforward to allow simulation objects' properties to be altered as well as read, with the alterations directly affecting the parameters' values in the simulation. This would allow the construction of visualisations that allowed the simulation user to interact directly with objects—in the same way that MASON, NetLogo and other tightly-integrated simulation environments already do—which is useful for debugging and experimenting with a system. Global properties, or properties that affect a class of simulatiion objects, could be placed in a shared simulation object that is read by all interested parties.

Time travel: at present the store's idea of time only follows the simulation, but it would also be possible for the store to keep a (limited) history of the simulation, allowing the user to wind the visualisation back and forth in time. Further integration with the simulation could allow the simulation to be paused or single-stepped through interactions with the store.

Meta-analysis: since the object store contains information about relationships between simulation objects, the framework could attempt to analyse these relationships to automatically identify clusters of related objects and patterns of interaction. These could be used for profiling the simulation—to identify performance problems with the simulation design, or automatically load-balance a distributed simulation—or could be used to automatically identify potential emergent properties of the complex system under study.

Acknowledgments

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

References

[1] P. S. Andrews, F. A. C. Polack, A. T. Sampson, S. Stepney, and J. Timmis. The CoSMoS process version 0.1: A process for the modelling and simulation of complex systems. Technical Report YCS-2010-453, Department of Computer Science, University of York, 2010.

[2] T. Ghetiu, R. D. Alexander, P. S. Andrews, F. A. C. Polack, and J. Bown. Equivalence arguments for complex systems simulations - a case-study. In S. Stepney, P. H. Welch, P. S. Andrews, and J. Timmis, editors, *Complex Systems Simulation and Modelling Workshop (CoSMoS 2009)*, pages 101–140. Luniver Press, Aug. 2009. ISBN 978-1-905986-22-4.

[3] J. E. Hannay, C. MacLeod, J. Singer, H. P. Langtangen, D. Pfahl, and G. Wilson. How do scientists develop and use scientific software? In *SECSE '09: Proceedings of the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, pages 1–8, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3737-5. doi: http://dx.doi.org/10.1109/SECSE.2009.5069155.

[4] F. A. C. Polack, P. S. Andrews, T. Ghetiu, M. Read, S. Stepney, J. Timmis, and A. T. Sampson. Reflections on the simulation of complex systems for science. In *ICECCS 2010: Fifteenth IEEE International Conference on Engineering of Complex Computer Systems*, pages 276–285. IEEE Press, 2010.

[5] C. G. Ritson, A. T. Sampson, and F. R. M. Barnes. Multicore Scheduling for Lightweight Communicating Processes. In J. Field and V. T. Vasconcelos, editors, *Coordination Models and Languages, 11th International Conference, COORDINATION 2009, Lisboa, Portugal, June 9-12, 2009. Proceedings*, volume 5521 of *Lecture Notes in Computer Science*, pages 163–183. Springer, June 2009. ISBN 978-3-642-02052-0.

[6] A. T. Sampson, J. M. Bjørndalen, and P. 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. ISBN 978-1-4244-2959-2.

[7] P. H. Welch and F. R. Barnes. Communicating mobile processes: introducing occam-π. In A. Abdallah, C. Jones, and J. Sanders, editors, *25 Years of CSP*, volume 3525 of *Lecture Notes in Computer Science*, pages 175–210. Springer Verlag, April 2005. ISBN 3-540-25813-2.