Integrating Life-Like Action Selection into Cycle-Based Agent Simulation Environments

Joanna J. Bryson, Tristan J. Caulfield and Jan Drugowitsch
University of Bath
Department of Computer Science
Artificial models of natural Intelligence (AmonI)
http://www.bath.ac.uk/comp-sci/ai/AmonI.html

July 30, 2006

Abstract

Standardised simulation platforms such as RePast, Swarm, MASON and NetLogo are making Agent-Based Modelling accessible to ever-widening audiences. Some proportion of these modellers have good reason to want their agents to express relatively complex behaviour, or they may wish to describe their agents' actions in terms of real time. Agents of increasing complexity may often be better (more simply) described using hierarchical constructs which express the priorities and goals of their actions, and the contexts in which sets of actions may be applicable (Bryson, 2003a). Describing an agent's behaviour clearly and succinctly in this way might seem at odds with the iterative, cycle-based nature of most simulation platforms. Because each agent is known to act in lock-step synchrony with the others, describing the individual's behaviour in terms of fluid, coherent long-term plans may seem difficult.

In this paper we describe how an action-selection system designed for more conventionally-humanoid AI such as robotics and virtual reality can be incorporated into a cycle-based ABM simulation platform. We integrate a Python-language version of the action selection for Bryson's Behavior Oriented Design (BOD) into a fairly standard cycle-based simulation platform, MASON (Luke et al., 2003). The resulting system is currently being used as a research platform in our group, and has been used for laboratories in the European Agent Systems Summer School.

1 Introduction

Standardised simulation platforms such as RePast, Swarm, MASON and NetLogo are making Agent-Based Modelling accessible to ever widening audiences. Some proportion of these modellers have good reason to want to express the actions of their agents in terms of real time. For example, they may want to describe continuous, durative actions such as walking or fighting wars. Such actions can either have pre-specified durations or they can continue occurring until some event consummates or interrupts them. Modellers may also need to increase the complexity of their agents to include relatively complex variable state, such as memories of past interactions with other agents, or conflicting theories of relationships between third parties.

Bryson (2003a) describes how agents of increasing complexity may often be better (that is, more simply) described using hierarchical constructions which express the priorities and goals of their actions, and the contexts in which sets of actions may be applicable. Describing an agent's behaviour clearly and succinctly in this way can seem at odds though with the iterative, cycle-based nature of most simulation platforms, which seem to specify that each agent proposes its own action in lock-step synchrony.

In the present paper we describe a Python-based version of the action selection for Bryson's Behavior Oriented Design (BOD). BOD splits an agent's control into two sorts of entities: modular *behavior libraries* which are written in conventional object-oriented languages such as Python, and Parallel-rooted, Ordered, Slip-stack Hierarchical (POSH) *action selection scripts*. Behavior modules enable action by:

1. containing the code describing *how* an action is performed,
2. providing a place to store any state / memory needed to conduct those actions, and
3. containing code describing any sensing that must be conducted in order to acquire that state / knowledge.

Action selection scripts (also called *plans*) then order these actions, and also the sensing events necessary to support decision making in a way that reflects the individual agent's priorities. In other words, action selection determines *when* an action should be expressed.

The Python version of the core action-selection system, jyPOSH, is currently being used for two different research projects. One is as a real-time system controlling a virtual-reality game agent in Unreal Tournament (Partington and Bryson, 2005; Bída et al., 2006), and the other is for agent-based social simulations. We are currently using MASON (Luke et al., 2003) for the social simulations, but our solution should be easily extendible to other platforms. We have used an early demo of the BOD/MASON system involving dogs herding sheep to teach BOD at the European Agent Systems Summer School in July 2005, where students were encouraged to extend the behaviour of the dogs and sheep, including turning the dogs into wolves that ate the sheep. We are now also building research projects on primate social behaviour on this BOD/MASON platform, although earlier versions of this work were run in NetLogo and SmallTalk (Bryson et al., 2006).

In this paper, we begin by describing the MASON platform and why we are using it. We then give a brief overview of BOD and its POSH action-selection framework. The main purpose of this paper is to outline the relatively simple steps necessary in the abstract for integrating POSH action selection into a cycle-based simulation tool, and the relatively more complicated process of actually integrating a Python planning system and behavior modules into MASON.

2 Background: Component Technologies

2.1 MASON and Other Platforms

MASON (Luke et al., 2003) is multi-agent simulation environment written in Java. It is designed to provide a generic platform with core functionality upon which many different types of simulation can be built. The philosophy behind MASON is to provide a fast, portable, and non-domain-specific framework that contains features commonly required by different types of multi-agent models. Other simulation environments, such as NetLogo and RePast, are more domain-specific

but may also provide a great deal more support for writing simulations in their respective domains. NetLogo, for instance, is designed for simulating social interactions and this is supported by the existence of fairly high-level, competent agents (called turtles) natively within NetLogo and a large number of functions (moving, turning, finding other turtles, etc.) to manipulate these agents. NetLogo provides its own high-level language which features the turtle's actions and ways to manipulate the turtles as special primitives. It is a relatively simple and constrained language intended to help inexperienced programmers build simulations quickly.

In contrast, MASON does not restrict itself to a particular type of agent but provides the flexibility to completely customise its agents. As a result, the primitives provided by MASON are on a lower level. If developers want to manipulate the agent's behaviour at a higher level of abstraction, they are required to program custom high-level primitives themselves. The group that developed MASON has a particular interest in evolutionary simulations, and as such were willing to sacrifice greater development time in exchange for rapid execution time. All entities must be implemented in MASON's native Java language.

Over the last four years, we have had approximately twelve student dissertations (undergraduate and one-year taught masters) on social simulation conducted in our group. Each of these projects has contained a phase where students evaluate a number of platforms and then choose the one on which they build their project. By far the most popular platform has been NetLogo (Wilensky, 1999). This is primarily because of the ease of developing both agents and UIs for assisting in running experiments (and, more recently, behavior-space parameter sweeps.) MASON is the second most popular tool. When it wins over NetLogo, it does so because it runs faster and is more programmable, e.g. it allows linking to Java libraries [1]. We also have had two students select RePast and one student select SeSAm[2].

MASON is divided into separate layers (see Figure 1). The inner layers — *the core* — can function without the outer layers, which provides visualisations and domain-specific functions. At the core are two essential layers, the model and utilities layers. The model layer contains classes that represent two- and three-dimensional fields in both bounded and toroidal forms, both discrete and continuous, along with methods to place, locate, and determine the distance between objects in the field. A scheduler is also supplied, allowing events to be run at various frequencies of iterations and even in a particular order during one iteration. The other core layer implements classes that are useful when writing simulations, such as optimised collections and a random number generator.

Besides the core layers, MASON includes a visualisation layer. This layer is designed in the basic MASON architecture as optional — simulations can be run with or without it, and performance increases when it is turned off. The layer allows for 2D and 3D visualisations using a sophisticated class model, making the display of model data highly customisable. This method also isolates the simulation in a separate thread from the GUI and displays, increasing performance, and enabling the visualisations to be turned on or off as the simulation runs. Information on the state of objects in the model can also be observed and altered at runtime through the use of the inspector classes in this layer.

Beyond these layers, MASON does not provide anything. It is intended as a generic simulation environment and leaves the implementation of domain-specific layers and of the actual models up to others. Our contribution is that we have extended the MASON environment by integrating it with a POSH engine in a compatible language. This allows modellers to use the BOD development

[1] Although NetLogo now also has a Java API, we have not yet tried to exploit this.

[2] The SeSAm student did not complete her dissertation, but this was probably not a consequence of the platform.

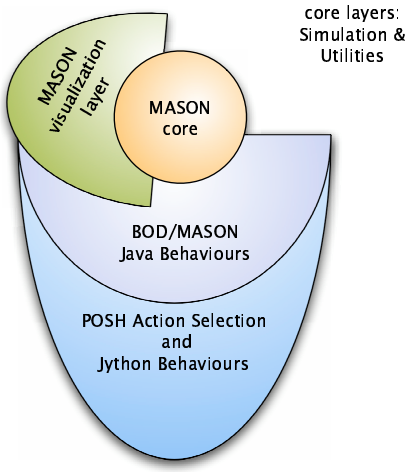

Figure 1: The basic BOD/MASON architecture. We have added extra layers of functionality on top of MASON's default layers (in keeping with MASON's design principles) which provide the user a simpler programming interface, including access to specialist action-selection code.

methodology in MASON, and as a by-product supports their describing their agents in Python rather than Java. This is an advantage because Python is a far quicker and easier language to code in.

2.2 BOD and POSH Action Selection

Behavior Oriented Design (BOD) is a methodology for constructing relatively complex, animal-like agents (Bryson, 2001, 2003b). The purpose of the methodology is to help ordinary programmers produce agents that generate behaviour to meet multiple, possibly conflicting goals that operate continuously and in parallel. Most programmers — and indeed, most ordinary people who simply make plans — are used to thinking in terms of sequences of actions. The challenge of making an autonomous agent is that many goals are held at the same time and must be met concurrently e.g. one agent can simultaneously be entertaining the desire to get a promotion, the desire to have dinner and the desire to wear a clean shirt in the morning. Further, these goals must be achieved in the face of an unpredictable, dynamic environment, which may provide both unexpected opportunities and challenges. Steps that might ordinarily be considered necessary for achieving a goal may not be required in some contexts, but may have to be repeated or even abandoned for other strategies in others.

BOD splits the problem of agent intelligence into two parts. One part is the construction of a *behavior library*. The behavior library consists of relatively ordinary code for describing individual actions, including active, goal-directed sensing and evaluation of the environment. The behaviour library is typically modular, based on the established development methodologies of object-oriented design (Parnas et al., 1985; Coad et al., 1997) and Behavior-Based AI (Brooks, 1991; Matarić, 1997). The other part of a BOD agent is a hierarchical structure known as a *POSH plan*

which organises primitive actions, including sensing[3]. The POSH plan determines an individual agent's capabilities and priorities. Many very different agents can share the same library of behavior modules provided each has its own POSH action selection. A single behaviour library is typically developed for any one research platform, but multiple types of agents can be designed which reuse aspects of this code yet have different priorities or 'personalities' specified in their POSH plans.

The primitives of POSH plans are *acts* and *senses*. These are the interface to the behaviour library — each primitive is typically a method call to one of the objects used to represent a behaviour module. Senses report on conditions in the simulation to inform decision points in the plan, while acts actually change some aspect of the simulation (possibly just the agent).

Picking appropriate primitives is a challenge. Deciding the correct granularity at which action selection should occur is half of the problem of action selection for AI. BOD does this through a set of heuristics which are run iteratively over the development period. Essentially, if a POSH plan is becoming too complicated, then current granularity may be too small and new, more abstract primitives should be built. On the other hand, if there is redundancy in the behaviour library code, then probably a primitive needs to be decomposed into smaller elements to facilitate reuse. If there is redundancy in a plan, then additional memory should be introduced into the agent in the form of state in a behaviour module, which the plans can the refer to, making them more general. Details of the BOD methodology can be found elsewhere (Bryson, 2001, 2003b).

Besides the plan primitives, POSH also provides three aggregate types which provide the plans' order and hierarchy. These are *action patterns*, which are simple sequences; *competences* which are prioritised sets of productions (pairs of sensory preconditions and their action consequences); and a *drive collection* — a special competence which serves as the root of the action-selection hierarchy and specifies the main drives or motivations for the agent. Details of POSH can also be found elsewhere (Bryson and Stein, 2001; Bryson, 2001, 2003a).

Action selection is where the programmer or planner's established ability to sequence actions and prioritise goals can be expressed. The underlying sequential structure can be encoded in the plan hierarchy, then the action selection mechanisms is able to manipulate the actual order of action expression in response to motivational and environmental context. These acts and context-checking senses are then the plan primitives which must be supported by the behaviour library.

The version of POSH we use with MASON is a derivative of pyPOSH. PyPOSH was originally implemented by Kwong (2003). It is a Python version derived from the (Bryson, 2001) lisp version of POSH action selection. As documented by Kwong (2003), we chose to implement a version of POSH in Python because Python is (like lisp) a high-level loosely-typed language which allows rapid code development. However, Python is also a scripting language with relatively familiar syntax and structure, making it more accessible for programmers familiar with languages such as Java, C or perl. Python is also strongly object-oriented, which makes it amenable to both contemporary software engineering in general and to Behavior Oriented Design in particular. Previous to this project, the main pyPOSH behavior library was for Unreal Tournament (Kwong, 2003; Partington and Bryson, 2005).

[3]POSH is an adjective which stands for "Parallel-rooted, Ordered, Slip-stack, Hierarchical"

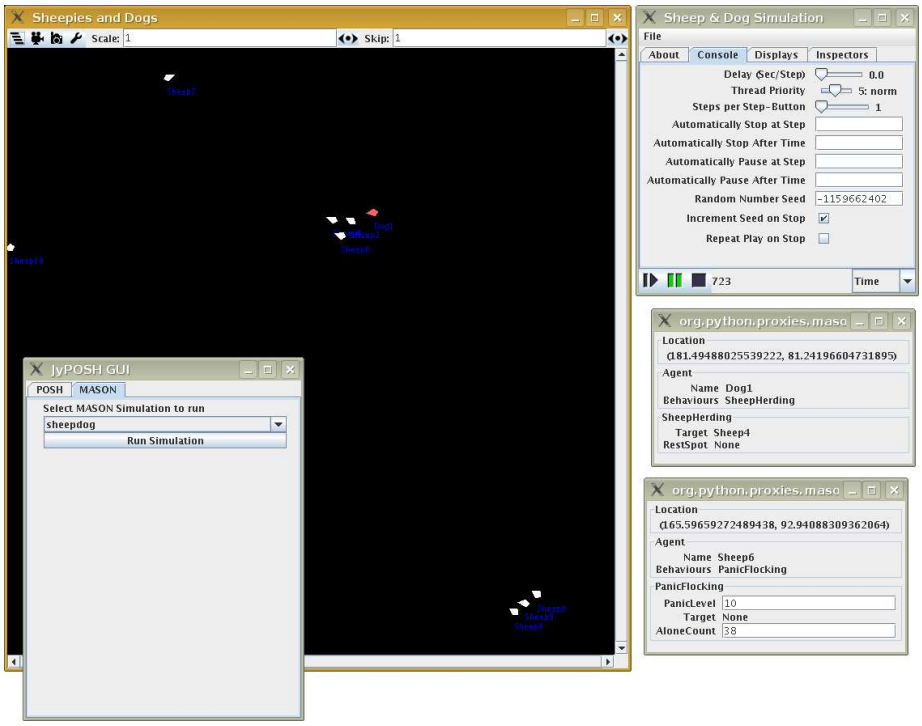

Figure 2: A screenshot of BOD/MASON running the sheep/dog demo. The MASON simulation window shows the agents, while the MASON control panel (upper right) is shows the console tab — this panel controls the simulation. The smaller windows are inspectors. The simulation can be started from the command line, or from the jyPOSH GUI (overlapping the simulation window on the lower left.)

3 Example: Of Sheep and Dogs

Before detailing the technical issues involved in bringing POSH to MASON, we will start by describing a complete BOD/MASON simulation. Let us consider a simple example: we want to simulate a bunch of sheep that are herded by one or more dogs. This gives us two types of agents to model, the sheep and the dog, both situated in the same environment.

Even though we think of the behaviour of sheep and dogs as very different, when we consider the problem of describing intelligence from first principles (e.g. physics), we realize that all agents in a particular environment will share a large amount of basic behaviour. For example, consider navigation. Currently we restrict ourselves to using the continuous toroidal environment MASON provides since this simplifies navigation. MASON is itself in Java, and does not provide many primitives, and so we have implemented some Java primitives on top of MASON that simplify the control of agents in these environments (e.g. the actions `move` and `setDirection` and the senses `location`, `orientation` and `distanceTo`). These primitives are implemented in Java to provide sufficient speed in the simulation. An additional layer in Python links the MASON agents to POSH action selection, and provides all the primitives implemented in the Java layer,

like, for example, information about the agent's location. Once implemented, all these primitives can be used and re-used by all agents that reside in that environment.

The sheep's behaviour is determined by the combination of its POSH plan and its behaviour library. The plan tells them to graze (i.e. do nothing) as long as they are calm and close enough to fellow sheep. If a sheep is not close enough to other sheep (where *enough* is determined in a behaviour and is dependent on just how calm they are feeling) it will move towards the others. When moving to other sheep agents, their motion is determined by simple flocking behaviour, balancing cohesion and aversion between sheep, and some momentum and random motion (Reynolds, 1987). If the sheep sense a nearby dog, their plan says that this is a higher priority concern than grazing. When a dog approaches, the sheep's panic level rises and influences the weight of the different motion components, causing the sheep to increase their speed and flock more densely. Grazing, moving towards other sheep, and flocking motion are implemented as a POSH behaviour library, written in Python. The act/sense primitives provide a thin layer on top of the MASON primitives. The different behaviours are then integrated by the POSH plan.

The dogs' behaviour is designed in the same manner, by linking a set of POSH primitives provided by the dog behaviour library by a POSH plan. The plan tells a dog to rest as long as the herd width does not exceed a certain threshold, and otherwise to approach the closest sheep to make them move closer together. The dogs can share the same resting behaviour as the sheep, but their approaching behaviour is different, since they pick just one sheep to hassle first. The agents' plans determine which behaviour from the library is expressed by those particular agents.

Having implemented the sheep and dog behaviour, all that is left for the user to do is to specify the environment and where the sheep and dogs are located. A simulation master file, also written in Python, takes this role by defining a set of agent classes by their POSH plan, behaviour libraries, and their appearance in the GUI. It also specifies the size of the environment, and how many sheep and dogs are initially randomly located in the environment.

The sheep/dog demo has been implemented for teaching purposes, to show the power of the BOD approach. Students are encouraged to think about how to change the behaviour of the agents. For example, how would you change the sheep into deer, which are smart enough to scatter if a predator attacks rather than to clump? How would you change a dog into a wolf which catches a sheep rather than just frightening them? Could you make a smarter dog that herds the sheep more quickly? Can you make a team of dogs?

To summarise, the only parts that have to be written by the users are:

POSH Behaviour Libraries containing a set of actions and senses that are usable by POSH plans. These actions and senses provide agent-specific higher-level functionality than what is provided by the simulation environment. The behaviour library for sheep, for example, provides amongst others the actions `set_panic_to_max`, `move_to_target`, `flock_move`, and senses `alone`, `predator_close`. The behaviour libraries are written in Python.

POSH Plans linking the actions and senses of the behaviour libraries to form the behaviour of the agent. The syntax of these plans is POSH-specific; the complete grammar can be found on the POSH web page or in the pyPOSH documentation. There is also an integrated development environment (IDE) for developing these plans, called the Advanced Behaviour Oriented Design Environment (ABODE).

Main Simulation Script to set up the simulation, specify the environment size, and initialise the agents of the environment. In our Sheep/Dog example, this script defines the sheep and dog

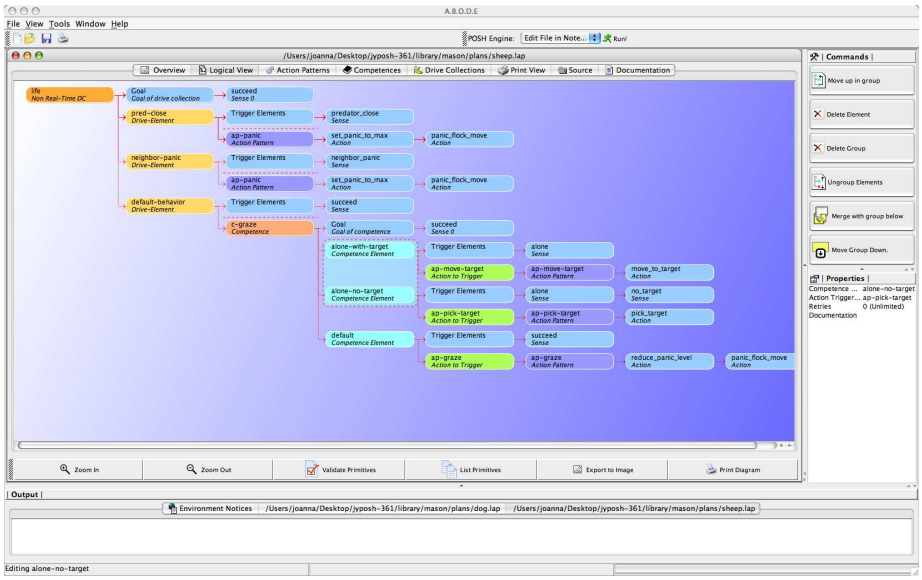

Figure 3: A screenshot of ABODE editing a sheep's POSH plan (see text.)

agents by linking their behaviours with their plans, specifies their appearance in the simulation, and sets the number of sheep and dogs in the simulation, and their initial location. The simulation script is written in Python.

The simulation is displayed and controlled by MASON, which can in turn be started from the main pyPOSH GUI. MASON uses pyPOSH and the user-provided behaviours and plans to let the sheep flock and the dog hunt them. The details of how exactly POSH is controlled from within a simulation environment like MASON — all hidden away from the user — is given in the next section.

4 Under the Hood

Our aim was to keep the tasks of the user developers simple while still maintaining the flexibility of POSH action selection, and the power of the MASON simulation toolkit. All the user needs to do is to write or extend Python behaviour modules, POSH scripts and a master simulation file. How we have provide such simplicity, and how MASON actually interfaces with pyPOSH is described below.

4.1 Language Issues

The POSH system was already written in the Python language. Rather than making it compatible with MASON by re-writing it in Java we decided to use Jython to integrate the two. Jython is an implementation of Python written in Java, which compiles the Python code to Java bytecode which is then run on the Java virtual machine. Jython has the extra advantage over ordinary Python that

Java classes can be used from within Python code, and vice-versa (with a few caveats). Jython also supports multiple-inheritance, and more importantly for us, multiple-inheritance from Python and Java classes simultaneously.

Jython implements an earlier version of the Python language than the one that POSH was written in, meaning that the code initially would not run under Jython. To fix this it was necessary to go through the POSH code and change any parts that used features only found in the newer version of the language. For instance, in more recent versions of Python, there are "true" and "false" keywords in the language, whereas in earlier versions these were represented by "1" and "0" respectively. Another problem that was at first more difficult to solve was the fact that the way Python did variable scoping changed between versions. This presented a problem, particularly with nested functions, where inner functions could not access the members of the enclosing class. Luckily, the new scoping was accessible in the version of Python that Jython implements, so instead of rewriting all of the existing POSH code we only needed to add an import statement ("from __future__ import nested_scopes") at the top of each Python file to enable the new scoping.

4.2 The Layer on Top of MASON

Jython allows the direct use of Java classes from Python scripts. Hence, MASON can be directly used from Python scripts, but then the user still has to cope with the complexity of the MASON framework. To avoid that, we have written a thin layer around MASON that removes some of its complexity and tailors the API specifically for a certain sets of environments to be used in combination with POSH actions selection[4].

For now we have committed ourselves to continuous toroidal environments of arbitrary size. Consequently, we provide a simulation class that only provides such environments, and a simulation GUI class that visualises them. In the master simulation scripts, the only tasks required to be done by the user are now:

- Override the simulation class to specify the size of the environment and initialise the agents at the start of the simulation. That means, the user has to specify the agent type, its initial state, and where it is located in the environment.
- Override the simulation GUI class to set the name of the simulation, and the colour of the environment.

All the rest of the set up for the MASON simulation and GUI is performed in the thin layer between the user and MASON.

4.3 Entities and Agents

As MASON provides only very limited agent functionality, we have constructed an additional layer for entities and agents that is similar to that which can be found in NetLogo. MASON provides the concept of *fields* of the environment that can be seen as a set of different layers, each of which covers the whole environment. Objects in the environment are located in one or several of those fields, and all the functions we have provided for these objects can be restricted to certain

[4]Early users of BOD/MASON complained a great deal that they needed to look in both Java and Python behaviors. That was never really our intention but more of a teething problem. In the current version of BOD/MASON we seem to have been successful at completely encapsulating the Java away from the user.

fields. Going back to the previous sheep/dog example, all the sheep are located on the "sheep" field, and all dogs are on the "dog" field. If a sheep wants to find the closest other sheep, then it just needs to look for the closest object in the "sheep" field. Hence, using different fields allows grouping of agents and entities. This is will speed up certain functions, particularly those exponential on the number of agents, such as comparisons. However, it adds a level of complexity for the MASON developer, since the fields need to kept track of.

The simplest objects we provide are *entities*, which have a location and a certain appearance in the environment, but no orientation. They can represent any stationary object in the simulation, for example, a tree or pond. In addition to its properties, we provide a set of sense primitives for these entities, like its distance to other objects, or the approximate "centre" of all objects in a certain field. These senses apply either to all fields, or just to a given subset.

Another class *oriented entity* inherits the basic entities and provides an additional "orientation" property, and some methods that allow it to move in the environment. Even though it does not yet provide autonomous control, this class provides all that is required from the MASON side to interface it with POSH action selection, as described in the next section.

4.4 Synchronous Asynchrony

BOD agents are asynchronous by design. New actions are performed in response to events in the environment or changes of internal motivation of the agent, not at regularly scheduled intervals. On the other hand, most simulation environments are stepped, and all agents have to choose their actions at each step. How can we combine those two modes of operation?

In fact, this problem is more simple to solve than it appears. Although BOD agents may have multiple threads or even devices supporting the behaviour generated by the behaviour modules, the actual POSH action selection is sequential and cycle-based. The responsiveness is a consequence of the rapid cycle rate of POSH, which typically can operate at hundreds of cycles per second, providing that no primitive action it calls has blocked it for any length of time. In fact, to sustain the illusion of asynchrony, one of the requisite properties for BOD behaviour libraries is that the method calls to behaviours should *not* block while they wait for a protracted action to happen. Where protracted actions occur (e.g. motion in a robot) methods should only serve to initialise or reparameterise an action which is then sustained by the behavior module itself.

When a BOD agent appears to be engaged in a sustained activity what actually happens with respect to action selection is the following. At each cycle, POSH checks if the conjunction of its sensed internal or external states is unchanged, which causes it to perform the same 'act' as in the last cycle. The expressed behaviour is continuous because the POSH cycles are performed at a high frequency, and the action primitives are carefully designed so they show no disruption in expresses behaviour when they recur.

This architecture was originally designed to allow an agent with hierarchical action selection to still be a fully responsive and reactive real-time autonomous agent — in fact, POSH was originally designed for and implemented on autonomous robots. Fortunately, this structure also allows POSH to be easily integrated into stepped simulation environments. Rather than having the POSH agent continuously call the internal cycle, the control is now given to the simulation environment. Its role is to signal the POSH agent at each simulation step to perform one internal cycle, which causes the agent to perform actions according to its POSH plan. The agent's environment becomes the simulation environment, and its behaviour libraries use the simulation environment primitives to sense and act within this environment. Because some cycles in POSH are dedicated to details of

the decision making, a new expressed action may not be chosen on every cycle of the simulator. But this is not really a problem if we are trying to simulate realistic real-time agents. In fact, faking asynchrony in cyclic environments has been shown to be important in simulating animal-like behaviour (Hemelrijk, 2000).

What does that mean for the specific case of using MASON as a simulation environment? Firstly, we need to modify the MASON agent class we have described in the last section to use POSH to perform its action selection. A new class, *AgentBase*, which inherits from both the POSH agent and our MASON agent, is the foundation for all agents that perform POSH action selection in a MASON simulation (e.g. the sheep and the dog in our previous example). This class overrides the standard POSH cycle control, and instantiates a separate MASON control class. At each step of the simulation a method of this control class is called by the MASON stepped scheduler, and that method subsequently calls the POSH core cycle. Hence, at each step of the simulation causes one POSH cycle to be performed. Having several agents in the same simulation, the MASON scheduler calls the POSH cycle of each agent one after the other.

The final question is what implementation of POSH the agent will use. There are currently two versions of POSH action selection encoded in jyPOSH — a scheduled version, *ScheduledAgent*, which maintains a little more decision state and therefore is more cognitively plausible, and a strict slip-stack, *StrictAgent*, which runs faster. Bryson (2001, section 4.6) details the differences between these POSH implementations.

The only thing that is left to the user is to specify for each agent class the set of behaviour libraries to use, and which plan to run. The agent class has to inherit out combine POSH/MASON agent class, and the behaviour/plan information is given by overriding its constructor.

4.5 Intra-Module and Intra-Agent Communication

The behavior library for BOD agents is usually modular. Modular decomposition driven by the sorts of memory an agent needs, just like in standard object-oriented design (Coad et al., 1997). BOD however does not require the modules to be fully encapsulated. Just as for the action selection, one behavior module may poll another for information the second module is specialist in. Similarly, sometimes agents will need to communicate to each other in the simulation. Even if in the real world that communication would have been tactile or visual (for example if one agent hugs another or glares at another), in a simulation this information has to be transmitted between agents. Since in a BOD system all sensing is done in behavior modules, in a BOD agent such information needs to be transmitted between two agents' behavior modules.

BOD/MASON framework provides a unified solution for both of these problems. All `Behaviour` instances for a single agent are stored in a behaviour *dictionary* (the Python version of a hash table), which is maintained by the POSH `Agent` object. The `Agent` object also provides the method `getBehaviour` which allows any process having access to the agent object to get the state of any of the agent's behaviour module objects. Additionally, each behaviour object keeps a reference to the agent object that owns that behaviour. Thus different behaviours can exchange state information by calling `self.agent.getBehaviour(''behaviour_name'')`, and can get information about the internal states of other agents (like the closest other object on the "agents" field) by `self.agent.closest('agents').getBehaviour(''behaviour_name'')`.

4.6 Agent Visualisation

The idea of separating the visualisation from the simulation core in MASON is also applied on the visualisation of agents. While the agent only exists in the simulation, it is linked to a portrayal object that is responsible for its visual appearance in the MASON GUI. Directly translating that to POSH/MASON, the user would have to perform several additional (non-trivial) steps to define the appearance of an agent.

We have simplified this procedure by removing the separation between the agent object and its visualisation. The user now has to call a class method in the overridden constructor of the POSH/MASON agent to set the appearance of the agent. This method creates the agent's portrayal object and links it to the simulation visualisation as soon as it is created. This is another measure for the user to simplify the use of MASON.

4.7 Inspecting Agent States

In complex simulations with many agents it is hard to keep the overview over the variables which specify the memory and identity of the agents. Fortunately, MASON provides some tools that allow the display of changes in the agent's internal states while the simulation is running. Its implementation is based on using Java *reflection* on the methods of the agent to identify accessor and mutator methods, and display them in the GUI's inspector window.

In BOD, the state of an agent is determined by the state of the the behaviour modules it uses from the behaviour library. These behaviours are implemented in Python. Unfortunately, Jython creates Java proxy objects for Python objects, so its methods are not accessible through Java reflection. Hence, the standard inspectors do not work for POSH/MASON agents.

To regain access to the MASON inspectors for BOD/MASON agents, we have added an additional method to the `Behaviour` class (the base class for all behaviours) which lets the user register Python accessor and mutator methods to be used by MASON inspectors. These methods are collected in the agent's inspection objects. Whenever the user requests inspection of a certain agent, some Python code in the BOD/MASON agent's base class adds the inspection GUI components and queries the state accessor methods to display the current state of the agent's behaviours. The user can then modify that state through text fields in the GUI, and these modifications are communicated to the agent's behaviours via the provided mutator methods.

Again, all the implementation detail has been hidden from the users. The only thing they have to do is to provide and register accessor/mutator methods for all the behaviour she defines.

5 Summary

This paper has presented the BOD/MASON agent based modelling tool suite. We have described both in general how complex agent action selection can be incorporated into a stepped ABM simulator, and in particular the problems we encountered and the solutions we have implemented in building POSH capabilities into MASON. We have also briefly outlined the Behavior Oriented Design methodology and provided an overview of MASON, including a brief comparison between it and other simulation platforms.

Since BOD/MASON is one of the development platforms for our research into the evolution of social behaviour, we expect it will continue to grow and improve. We have now created a

mailing list and a bug-tracking system as well as having alpha released an IDE for POSH plans. We encourage our colleagues to consider using this system.

6 Acknowledgements

BOD/MASON was originally built by Tristan Caulfield and has been under continuing development and refinement by Jan Drugowitsch. Development time for both researchers was funded from a grant from The UK Engineering and Physical Sciences Research Council (EPSRC), Grant GR/S79299/01 (AIBACS). ABODE is being funded as contract work by an anonymous industrial benefactor. Thanks also to Ivana Čače and the students of the 2005 European Agent-based Systems Summer School (EASSS) who provided useful feedback. The greatest thanks goes to our primary user, Hagen Lehmann, for suffering through all stages of BOD/MASON development — an interesting way to learn to program for the first time.

References

Bída, M., Burket, O., Brom, C., and Gemrot, J. (2006). Pogamut — platform for prototyping bots in unreal tournament (*pogamut — platforma pro prototypování botøu v unreal tournamentu*). In Kelemen, J. and Kvasnicka, V., editors, *Proceedings of Sixth Czech-Slovak workshop on Cognition and Artificial Life*, pages 67–74. Slezská Univerzita v Opave. in Czech.

Brooks, R. A. (1991). Intelligence without representation. *Artificial Intelligence*, 47:139–159.

Bryson, J. J. (2001). *Intelligence by Design: Principles of Modularity and Coordination for Engineering Complex Adaptive Agents*. PhD thesis, MIT, Department of EECS, Cambridge, MA. AI Technical Report 2001-003.

Bryson, J. J. (2003a). Action selection and individuation in agent based modelling. In Sallach, D. L. and Macal, C., editors, *Proceedings of Agent 2003: Challenges in Social Simulation*, pages 317–330, Argonne, IL. Argonne National Laboratory.

Bryson, J. J. (2003b). The behavior-oriented design of modular agent intelligence. In Kowalszyk, R., Müller, J. P., Tianfield, H., and Unland, R., editors, *Agent Technologies, Infrastructures, Tools, and Applications for e-Services*, pages 61–76. Springer.

Bryson, J. J. and Stein, L. A. (2001). Architectures and idioms: Making progress in agent design. In Castelfranchi, C. and Lespérance, Y., editors, *The Seventh International Workshop on Agent Theories, Architectures, and Languages (ATAL2000)*. Springer.

Bryson, J. J., Yasushi, A., and Lehmann, H. (2006). Agent-based models as scientific methodology: A case study analysing primate social behaviour. *Philosophical Transactions of the Royal Society, B*. in press.

Coad, P., North, D., and Mayfield, M. (1997). *Object Models: Strategies, Patterns and Applications*. Prentice Hall, 2nd edition.

Hemelrijk, C. K. (2000). Towards the integration of social dominance and spatial structure. *Animal Behaviour*, 59(5):1035–1048.

Kwong, A. (2003). A framework for reactive intelligence through agile component-based behaviors. Master's thesis, University of Bath. Department of Computer Science.

Luke, S., Balan, G. C., Panait, L., Cioffi-Revilla, C., and Paus, S. (2003). MASON: A Java multi-agent simulation library. In Sallach, D. L. and Macal, C., editors, *Proceedings of Agent 2003: Challenges in Social Simulation*, pages 49–64, Argonne, IL. Argonne National Laboratory.

Matarić, M. J. (1997). Behavior-based control: Examples from navigation, learning, and group behavior. *Journal of Experimental and Theoretical Artificial Intelligence*, 9(2/3):323–336.

Parnas, D. L., Clements, P. C., and Weiss, D. M. (1985). The modular structure of complex systems. *IEEE Transactions on Software Engineering*, SE-11(3):259–266.

Partington, S. J. and Bryson, J. J. (2005). The behavior oriented design of an unreal tournament character. In Panayiotopoulos, T., Gratch, J., Aylett, R., Ballin, D., Olivier, P., and Rist, T., editors, *The Fifth International Working Conference on Intelligent Virtual Agents*, pages 466–477, Kos, Greece. Springer.

Reynolds, C. W. (1987). Flocks, herds, and schools: A distributed behavioral model. *Computer Graphics*, 21(4):25–34.

Wilensky, U. (1999). *NetLogo*. Evanston, IL, USA.