Programming with simulated neurons: a first design pattern

Carl Evans
Dept. of Computer Science
Middlesex University
C.Evans@mdx.ac.uk

Ian Mitchell
Dept. of Computer Science
Middlesex University
I.Mitchell@mdx.ac.uk

Chris Huyck
Dept. of Computer Science
Middlesex University
C.Huyck@mdx.ac.uk

Abstract

An investigation has been carried out with regard to programming a form of deterministic logic based entirely in terms of biologically plausible neurons. To this end, a prototype has been successfully developed that incorporates a neuron version of the classic state design pattern. This neuron version is based on a novel programming technique, which models logical states as persistently active cell assemblies. These are populations of intra-connected neurons that have been triggered to continually fire until programmatically suppressed, thus enabling a neural form of state-transition logic. These neural-state cell assemblies have been developed using a specialist neuron simulation software library that is commonly employed by neuroscientists and is the adopted software protocol for the hardware platforms currently being developed for the Human Brain Project. An underlying inspiration of the work is to look forward to the possibility of a programming paradigm based entirely on biologically plausible neurons. It is envisaged that such a neural programming paradigm would benefit from established techniques, and that the neural cell assembly state pattern that has been developed and described in this report is a next step in that direction. In addition, a new graphical notation has been formulated in order to visualise the prototype. Whilst not a primary focus of the research to date, this visualisation notation may prove beneficial to the computational neuroscience community who work with similar neuron simulation software as that employed for the prototype presented here.

1. Introduction

This report describes a first design pattern for programming with simulated neurons. It is, in part, an adaptation of a simulated neuron programming technique pioneered by Huyck (2009). It is a novel approach that intersects the typical perspectives of computer scientists specialising in artificial intelligence (AI) and computational neuroscientists for whom a primary concern is modelling the biochemical function of the brain. That is to say, in practical terms computer scientists working with standard network topologies tend to focus their interest on the design of artificial neural networks (ANNs) for learning (and classification) tasks, whereas neuroscientists tend to investigate and measure response to interconnected populations of neurons, via the use of specialist neuron simulators, based on a variety of mathematical models. Members of the AI community often develop ANNs using general purpose, symbolic, programming languages. One could suggest that a desirable goal of the connectionist approach to programming AI systems would be to have a programming language based entirely on neurons. Perhaps this would be a domain-specific language rather than a general purpose language, but central to its approach would be a much more biologically plausible mapping to biochemical neuron activity than can be achieved with the general purpose symbolic languages that currently exist. However, whilst a purely neuron-based programming language does not yet exist, there are hardware platforms that are based on a neuron architecture rather than a traditional Von Neumann structure, e.g., the SpiNNaker architecture (Furber et al. 2013). Furthermore, a number of such neuron hardware platforms support neuron simulator libraries such as PyNN (Davison et al. 2008). The research groups employing neuron simulation libraries generally belong to the neuroscience community. Huyck and Mitchell (2014), on the other hand, are computer scientists who have demonstrated the use of this combination of neuron-based hardware and neuron simulator software to develop a classification system based on the earlier simulated neuron programming technique developed by Huyck. That work has a broader goal of machine learning rather than promoting a novel programming paradigm per se. However, the focus of the work reported here is to adapt this programming model and to make some further steps towards a simulated neuron design pattern that aims to provide a blueprint for, at least, one aspect of programming deterministic logic in simulated neurons. In particular, this work has mapped the concept of programming with neuron populations to the classic *state design pattern* of Gamma et al (1995). The state design pattern

is specifically an object-oriented design pattern that has a distinct model of state object transitions. This pattern does lend itself to a neuron model, which represents state as a firing neuron populations, but such a design is less straightforward than the object-oriented version. Addressing this design problem is a key feature of the work described in this report. In addition, the authors have a desire to demonstrate integration of a simulated neuron programmed module within a broader software framework and demonstrate more general utility. Accordingly, a small prototype has been successfully developed, which incorporates the neuron state model as a component within a model-view-controller (MVC) architectural pattern, whereby another of the classical design patterns, *the strategy pattern*, is employed to interchange the underlying model component between an object implementation (using object-oriented Python) and a neuron implementation (using PyNN).

The remainder of this report is structured as follows. Section 2 provides some brief background to aspects of computational neuroscience and design patterns that underpin this research. Section 3 describes the neuron-based state pattern of programming which is the main focus of this work. In presenting this, the report describes a new graphical notation that has been developed in order to visualise the implementation of the prototype. Section 4 discusses the integration of the neuron model within the context of an MVC framework, and Section 5 provides conclusions and proposes future evolution of the research.

2. Background

The following background sections aim to convey some basic underpinnings to the work. They mainly relate to a few key neuroscience concepts, the fundamentals of design patterns, and refer to the software technologies that have most relevance.

2.1 Spiking neuron models

The concept of a 'spiking' neuron model is a fundamental feature. When one (pre-synaptic) neuron signals another (post-synaptic) neuron across their synapse, a change in the neuron's electrical (membrane) potential occurs. If the potential is large enough, the charge on the post-synaptic neuron rapidly changes. This is known as an 'action potential', which reaches a peak and resets, thus forming a voltage pulse, or 'spike' in the membrane potential.
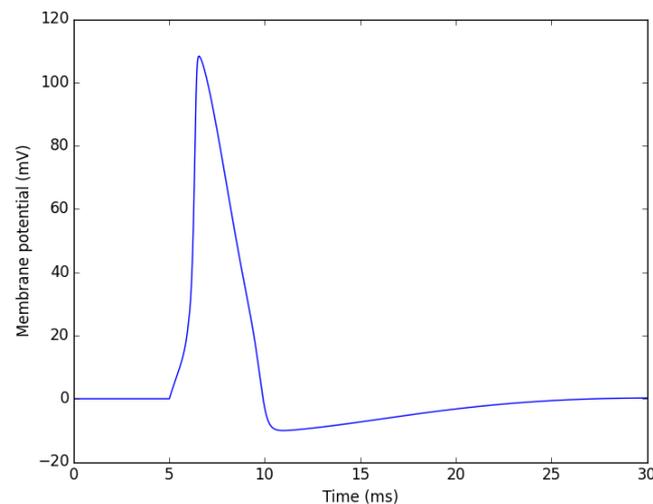
Figure 1 – Characteristic plot of an integrate and fire spiking neuron

Figure 1 illustrates the characteristic sharp rise and peak of potential, which then 'leaks' away and resets to a baseline voltage (typically achieving a negative charge prior to settling to the baseline). This represents a classic neuron model that is typically represented in computational neuroscience as a numerical integration, hence the term 'integrate and fire' neuron model (Brette and Gerstner, 2005). There are numerous mathematical neuron models based on this theme.

2.2 Neuron simulation and hardware

Several neural simulation tools have been developed with the purpose of allowing neuroscientists to simulate biologically plausible networks of neurons based on the types of spiking neuron model described above. These simulation systems allow computational neuroscientists to build neuronal networks at a high level of abstraction and their application programming interfaces (APIs) allow researchers to vary the parameters of the mathematical formulae on which the chosen neuron model is based. Furthermore, these simulation technologies enable large scale neuron networks, and developers often think in terms of a network of neuron 'populations' rather than a network of single neurons (although, of course, still possible and potentially of interest). The simulator software libraries provide support for creating sets of neurons that can be both intra-connected within a given population, as well as being inter-connected with other sets of neurons. They also provide numerous algorithms to define different types of cell population connection, along with various other features such as the generation of electrical inputs (such as spikes), and facilities to examine network activity. Examples of popular spiking neuron simulators are NEST (Plesser et al. 2015) and Brian (Goodman and Brette, 2013). When creating neural networks with these simulators, developers can take advantage of the Python programming language. PyNN (pronounced 'pine') is a Python package that provides a simulator-independent library for building neuronal network models (Davison et al. 2008). PyNN operates at a higher level of abstraction such that the (same) Python code used to create a network will run on several underlying simulator implementations that are supported, effectively providing a type of 'write once, run anywhere' meta-language. Currently NEST and Brian are both supported by PyNN. Like the supported simulators, PyNN provides a high-level API for neuron populations, synapse models, connectivity algorithms etc., particularly for large-scale networks, whilst still allowing a lower-level API that may be suited to smaller networks but allows more flexibility. An important aspect of the PyNN library is that it is the adopted software API for the European Human Brian Project[1] (HBP). In particular, two (complementary) neuromorphic hardware projects are under development: BrainScaleS[2] at Heidelberg, Germany, and SpiNNaker[3] at Manchester, UK. These projects are developing semiconductor computer chips based on novel neural architectures rather than the traditional Von Neumann architecture. Both the BrainScaleS and SpiNNaker systems have an interface, designed for neuroscience researchers, based on Python scripts using the PyNN API. An HBP funded project being conducted by the AI group at Middlesex University, entitled 'Neuromorphic Embodied Agents that Learn' (NEAL) aims to develop an agent system that learns, specifically using a test environment of PyNN in combination with the HBP Neuromorphic Platforms (Huyck et al. 2015). The NEAL project provides a context for the work reported here, and this has been influential in the adoption of PyNN and Python technologies for this research.

2.3 Cell assemblies

Using a neural simulation middleware, such as PyNN, one can create connections between neuron populations using various synapse models. Synapses are either excitatory or inhibitory. Put very simply, the type of synapse model can determine whether or not the action potential of a pre-synaptic neuron stimulates the post-synaptic neuron (excitatory) or negates its existing activity to some degree (inhibitory). An important biological aspect is that the strength of a synapse can change over time, either in the short term or long term. With repeated modification, the strength of the synapse excitation (or inhibition) becomes increased (or decreased), and semi-permanent, an effect known (biologically) as long-term potentiation (LTP) or long-term depression (LTD). LTP is thought to be the basis of learning and memory, a process known as 'plasticity'. Furthermore, it is understood that the proximity of connected neurons is very influential, such that, if cells (neurons) that are near to each other repeatedly fire together, their firing efficiency is increased, or reinforced, a phenomenon sometimes referred to as Hebbian learning, named after Donald Hebb, the father of neuropsychology (Hebb, 1949). Hebb was also particularly interested in how neurons acted together in groups, or 'cell assemblies', which has been a focus of work by Huyck and his co-researchers. In particular, Fan and

---

[1] https://www.humanbrainproject.eu/

[2] https://brainscales.kip.uni-heidelberg.de

[3] http://apt.cs.manchester.ac.uk/projects/SpiNNaker

Huyck (2008) developed a variation using a 'fatiguing leaking integrate and fire' (FLIF) neuron model to form cell assemblies. This FLIF neuron model has been employed to demonstrate how a population of intra-connected neurons (i.e., a cell assembly) can be configured to fire in a persistent manner. That is to say, the neurons within a given assembly can be wired to maintain a continuous firing state, and this has been used to demonstrate that a network of cell assemblies can be arranged to represent a finite state automaton (FSA). Further discussion on this model is provided in Section 3. With regard to the use of cell assemblies, a priority of the work of Huyck at al. has been on learning, with NEAL being one of the more recent projects. The research described in this report is orthogonal to this theme. The focus is not on learning, as such, but rather to demonstrate how the cell assembly approach can be used to represent deterministic logic by way of a neuron-based state-transition model. In particular, a blueprint for programming with simulated neurons is proposed that adapts the cell assembly model to represent the concepts behind a classic object-oriented design pattern, namely the *state* design pattern.

2.4 Design patterns

The world of software design patterns has broadened considerably since the landmark text by Gamma et al. (1995). The 'gang of four' (GoF) authors, as they are frequently referred to, proposed specifically twenty-three object-oriented patterns, which have become regarded as classic software design patterns. It is generally accepted that some of these classic patterns are, perhaps, more pervasive than others. For example, the *observer* design pattern is fairly ubiquitous, whereas a pattern such as the *flyweight* pattern tends to be seen in quite specialised domains such as computer graphics. Buschmann et al. (1996) extended the patterns concept to an architectural level (commencing a series of texts describing software architectures, the so-called Pattern-Oriented Software Architecture, or POSA, patterns), and since then a large patterns community has evolved and continues to grow[4]. A fundamental concept of design patterns is that they provide a solution template to a recognised design problem. Very importantly, design patterns are not *invented* specifically to solve a given design problem. Rather, they are *recognised* for what they really are, i.e., acquired wisdom with regard to an existing approach to solving a recurring design problem. It is the categorisation and documentation of that solution blueprint that constitutes the design pattern. Gamma et al. did not invent their patterns, rather they documented and formalised known techniques to solve known problems, and it is also particularly important to acknowledge that they aim only to represent *guidance* rather than absolute frameworks. As an example, consider the state design pattern. This is one of the original GoF patterns and has applicability when a component's dynamics should vary when its internal state is modified (it is thus regarded as an example of a *behavioural* pattern). The configuration of an object-oriented pattern is typically reflected in a UML (Unified Modelling Language) class diagram, which presents the general template of the design solution. Figure 3 illustrates the standard UML class diagram for the state design pattern. There is no absolute requirement to follow this exact model in order to call one's implementation a state pattern. But key to the class configuration is a hierarchy of state classes that extend from a common parent class, which in turn defines a consistent protocol for the client component (the 'context' class). The basic arrangement is that, at any point in time, the context would be linked with only one of the implemented state class objects, and the currently-referenced object would represent the current state of the system. Hence, the specific implementation of the operations of the current object define the dynamics of the current state. However, a key aspect that the general GoF state design pattern UML class diagram, as shown in Figure 3, does not explicitly illustrate is that the *spirit* of the pattern is for an individual state object to determine, or control, which state it transits to. That is to say, it is a particular feature of the object-oriented state design pattern that the implemented state sub-classes take responsibility for *changing the class* of the current state object (as referenced by the context object) whilst adhering to the abstract protocol specification.

Referring to Figure 3, if the context object (i.e., an object of ContextClass) is currently linked to an object of class ConcreteStateA, and a request to execute an operation on that object is made, then the implementation of that object's operation may determine whether or not the reference from the context object to a state object should be changed, and if it is to be changed, which state object (of a

---

sibling sub-class of the abstract State class) it should be changed to (in which case it would typically 'call-back' to the context object to set its new link).
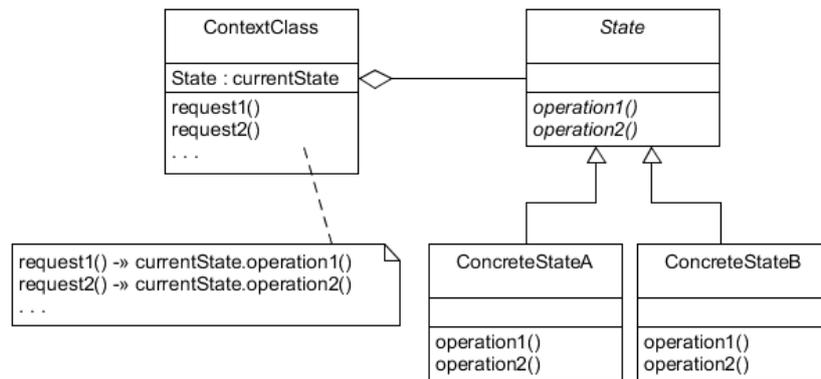

Figure 3: State design pattern (object oriented)

This state sub-class control is particular to the object-oriented design pattern, rather than the more general FSA concept. Furthermore, different operation implementations may determine different state change decisions, thus promoting a potentially complex object state transition model. For the described prototype, however, the relatively simple model of a traffic light simulation has been selected. This system has only four discrete states, each with a single operation to essentially move to the next state in the sequence (hence only one possible state transition per state). In terms of an object model, the typical implementation is for each concrete state object to set the reference of the context object (acting effectively as a *state-machine* object) to point to the next state object in the sequence, with the traffic signal system having a cyclic state transition model as illustrated by the UML state diagram in Figure 4.
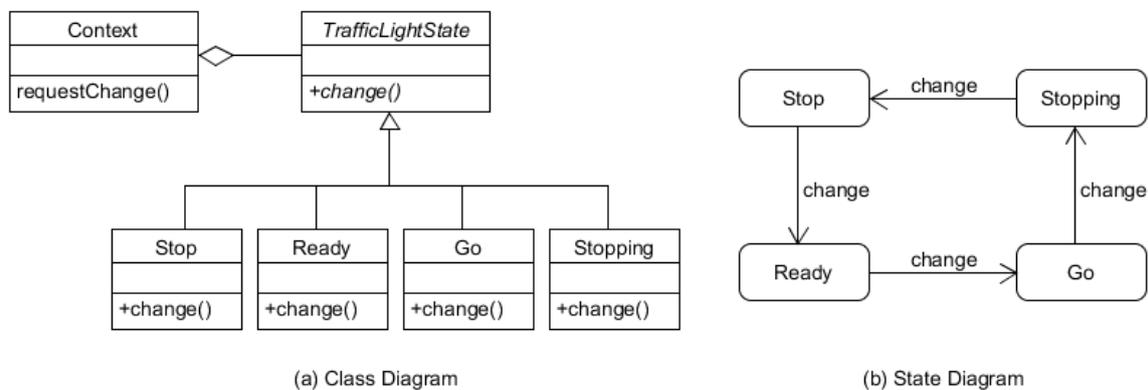

Figure 4: Object-oriented state pattern representing a traffic signal

Figure 4 (a) illustrates the structural (class) configuration of the general GoF state pattern applied to the traffic signal example. This only requires one abstract operation that each of the four concrete (light) states implement, and that is to change the reference of the context object. The dynamics of this are represented in the UML state diagram illustrated by Figure 4 (b).

2.5 Summary
The thrust of the work described in the remainder of this report addresses the premise that simulated neurons can be used to program in a general purpose manner. Such a programming paradigm would naturally appeal to those working within the field of artificial intelligence and could be employed to complement typical machine learning and classification tasks (Huyck and Fan, 2007). The underlying theme of this position is to adopt the connectionist approach to AI and aim to develop 'intelligent' programs in a biologically plausible manner. Some connectionists might argue that this cannot

ultimately be achieved with symbolic programming alone. It is envisaged that this will take on more importance with the emergence of biologically inspired computer architectures that will, perhaps one day, support a programming language that is predominately neural. In addressing this challenge, much of the work of Huyck and his co-researchers has had a strong focus on the use of cell assemblies as a biologically plausible model. Whilst, in the main, the work of Huyck et al. has investigated numerous aspects of learning and memory, the idea of being able to program more generally, via the cell assembly model, underpins a number of their projects and provides the inspiration for this investigation. This branch of the research deviates from a specific focus on learning and memory, and attempts to validate the cell assembly based FSA design by adapting it to a blueprint for, at least, one aspect of deterministic programming in neurons. That is to say, a first design pattern that is specific to programming with simulated neurons.

3. A neuron-based state pattern

This section will describe the prototype software that has been developed in order to demonstrate the neuron-state model described in the report. The source code is available to download directly from the following url: http://www.cwa.mdx.ac.uk/NEAL/code/neural-state-simulation.zip.

3.1 Visualisation of PyNN code

Rather than presenting Python code, this report will convey the structure of the program using a visual representation of the network, and in particular, visualisations of specific PyNN programming structures that are critical to the model. These visualisation are, in essence, a novel feature in their own right. They have been designed to aid in comprehension (and documentation) of some earlier projects that employed similar cell assemblies, such as NEAL. There is clear potential to extend this visualisation aspect further, but this is not a primary focus of this report.

A PyNN program is a timed simulation. Following initial set up (mainly related to neural timing parameters), a simulation typically comprises the following aspects: a spike generator, one or more populations of neurons, and a set of *projections* that link populations of neurons. Any given population can be based on a single cell type, or a combination of cell types. The populations created for the prototype that has been developed comprise a single cell type, which is an integrate-and-fire model. The PyNN class representing a cell type provides numerous parameters to configure those cells according to the mathematical model upon which they are based. A projection makes synapses from the neurons of one population to another population, and does so using a specified *connector*. Connectors comprise a combination of a synapse type (which specifies excitatory and inhibitory weight values), and a connector algorithm. Various standard connector algorithms are provided via the PyNN API. For example, one can connect a single neuron from one population to all neurons in another population, make 'all-to-all' mappings, make random connections, or provide bespoke mappings.

Whilst PyNN provides a wealth of neuron component classes within its API, only a few visual artefacts are required to represent the neuronal state model that has been developed: a neuron population, an intra-population projection, and inter-population projection (which specifies connector or synapse type) and a spike-source component. Examples of these are illustrated in Figures 5a and 5b. PyNN provides a series of specialised cell types (as programming constructs) to represent a range of electrical inputs to a neuron (or a neuron population). Collectively, these are termed 'spike sources' and are typically used to stimulate (i.e., add excitatory or inhibitory weight to) neurons within a given simulation. The prototype employs a straightforward 'spike source array' cell type. This essentially allows the programmer to code an array of specific 'spike times' throughout the simulation (in milliseconds). The spike input is achieved by connecting the spike array to a population of neurons. Figure 5a (i) illustrates a spike source of one neuron. A key feature of the cell assembly approach is to create a population of neurons that are intra-connected. So, for each neuron in a given population, the programmer can 'project' a connection from that neuron to its neighbours within the same population. Furthermore, the programmer can specify to which neighbours to project the connection. Figure 5a (ii) illustrates a visualisation that is used to represent a population of neurons, with the number of neurons in the population indicated by the number enclosed within the triad of connected nodes. Figure 5a (ii) is intended to illustrate that each of the 10 neurons in the population is connected to

each of the other 9 neurons in the population, but not self-connected. Figure 5a (iii) illustrates the same type of population with neurons self-connected as well as to all neighbouring neurons.
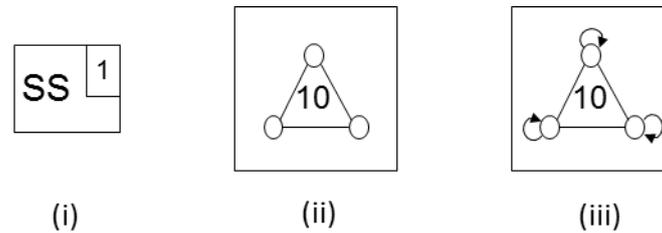

Figure 5a: Visualisation of neuron populations

With populations defined, one can use several variations of connector to establish the overall network. Some of these are illustrated in Figure 5b. For example, Figure 5b (i) illustrates a connector that is of type 'all to all' with a net excitatory weight (indicated by the plus symbol). The arrows indicate the direction of the connection.
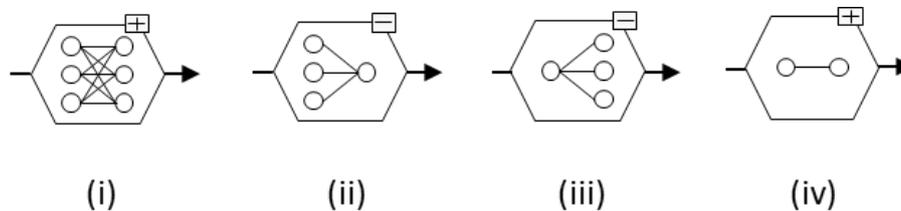

Figure 5b: Visualisation of population connectors

Figure 5b (ii) indicates an 'all to one' connection algorithm with net inhibitory weight (indicated by the minus symbol), Figure 5b (iii) shows one neuron connected to all in a given population with net inhibitory weight, and Figure 5b (iv) illustrates a one-to-one neuron connection with positive weight (excitatory), which is employed in the prototype for connecting spike sources to a cell assembly.

3.2 The model

The characteristic operation of a PyNN script is that it is executed for a predetermined time specified in milliseconds. The selected spike mechanism provides network input, and the response of the network to those inputs is recorded. There are several options for determining how data relating to network activity is recorded and examined but it is typically plotted in the form of a graph. The goal of the described prototype, however, is to represent firing states in real-time, and so requires an ability to interpret population activity during the simulation run rather than inspecting recorded data after the program has completed. The PyNN API does not specifically provide for this requirement, so it was a small design challenge that needed to be overcome. This was manageable, however, because the PyNN simulation control does allow for repeated invocations of Python 'callable objects'. This allows timed repetition of a section of code that can record the current activity of the network, inspect the recording during the same program iteration, and then reset the recording parameters ready for the next 'call back' from the simulation controller. Key to the implementation solution is the capability to project a population onto itself (i.e., an intra-projection) and that the cell assemblies of the defined states essentially all exist in a 'primed' state. In particular, Huyck and Fan (2008) demonstrated how two active cell assemblies could ignite a third, and effectively spread their activation to that third cell population. In addition, they were able to control how a cell assembly could supress the activation of another population so that its neurons stop firing. A similar design is used with regard to the implementation of the neural-based traffic-light state model implementation that is illustrated in Figure 6. It should be noted that the population, connector and spike source network illustrated in Figure 6 is one configuration that achieves a solution, but there are alternative configurations that can also achieve the desired results. For example, it is possible to employ populations with fewer neurons and configure the connectors with alternative algorithms and replicate the overall network activity.
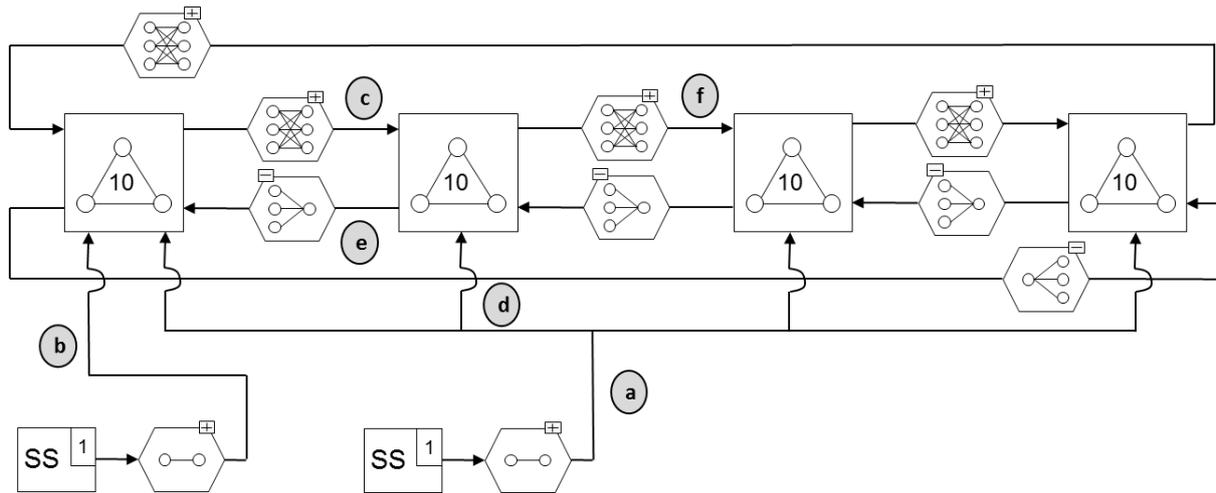
Figure 6: Visualisation of the PyNN neuron state model

In other words, Figure 6 illustrates an overall structural pattern that can use different population sizes and connector algorithms. The network visualisation in Figure 6 includes labels (a) through (f) to aid explanation of how the model operates. With respect to the state populations (indicated as containing 10 neurons) illustrated in Figure 6, state transitions operate from left to right. A spike-source generates a timed series of positive pulses throughout the simulation to represent triggers for state changes. This is indicated by label (a) in the diagram. This source is connected to a single neuron in each of the four cell assemblies (i.e., intra-projected neuron populations). The synapse weight is sufficient to 'prime' the populations, but not to make them fire. The network is essentially started by a second (single) pulse injected to the first cell assembly, indicated by label (b). The combined excitatory weight is configured to reach a threshold such that the first cell assembly now fires and maintains an active state. This active state is connected to the next cell assembly in the chain via an all-to-all projection as illustrated by label (c), but the net excitatory weight is insufficient for the second cell assembly to fire and it remains primed. After a predetermined interval, the next spike is injected into all four cell assemblies at point (a). The combined excitatory synapse weights that feed into the second cell assembly at points (c) and (d) are sufficient to reach a threshold to make the second cell assembly fire and maintain an active state. A one-to-many projection goes from that cell assembly back to its predecessor, which has sufficient inhibitory weight to suppress its activity, effectively taking it back to the primed state. The second cell assembly is projected forward to the third cell assembly, denoted at position (f) in the network, but again, there is insufficient weight to fire the third cell assembly until the next timed spike from position (a), and the sequence continues such that the network switches between all four states with only one cell assembly firing at a time. Deterministic control of the state transitions between cell assemblies is effectively under the control of the spike (trigger) input at position (a). In the prototype, this is a timed series of pulses, but this could be event-driven. Each timed pulse is transmitted to all four cell assemblies in the system such that the spike input is not targeted at any one specific population. In effect, the network of cell assemblies itself, via its wired projections, determines the active neuron state transition. Thus the spirit of the state design pattern is captured in the model.

4. Prototype architecture

As noted above, PyNN developers are typically neuroscientists who visualise the operation of their neural models via several types of graphical plots at the end of a given simulation. Certainly, the activity of the connected neuron populations in Figure 6 can be presented in a plot. However, the work described here has a more specific goal of conveying the utility of this network model as a neural programming pattern, and as such there is a desire to demonstrate the pattern as an implementation model within the context of a broader software system. To this end, the prototype has been integrated within a model view controller (MVC) architecture. What is, perhaps, interesting about this approach is that it is the model component that is interchangeable. MVC is a fairly ubiquitous architectural pattern, but the vast majority of implementation descriptions that one finds

reference to focus on the interchangeability of the view component. For example, many web frameworks utilise MVC in this manner. However, the interchangeability of the model component was always a key aspect to the MVC pattern described by Gamma et al. (1995). Although MVC was not listed as one of their 23 classic patterns, Gamma et al. made specific reference to the importance of their *strategy design pattern* within MVC. The strategy design pattern facilitates the encapsulation and exchange of an implementation algorithm at runtime for a given controller, and the prototype described here has adopted this architectural approach, which is illustrated as a UML class diagram in Figure 7.
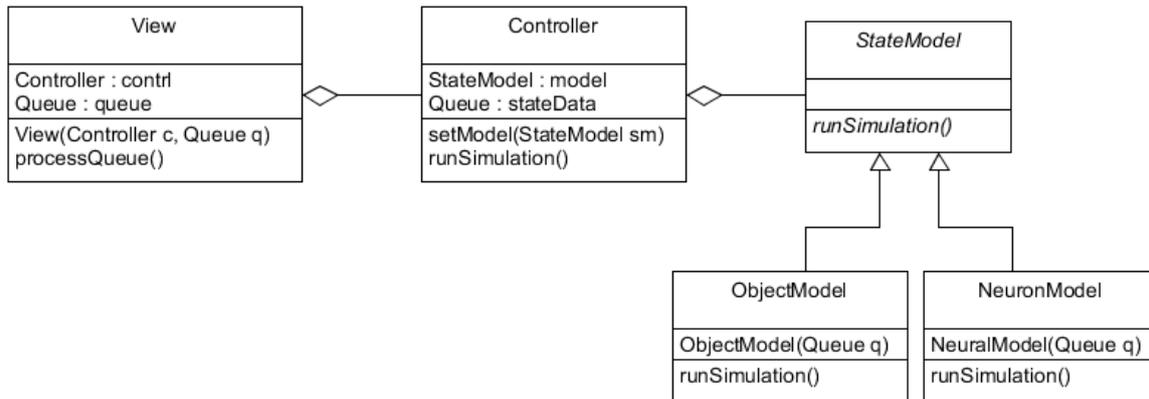

Figure 7: Neuron state model as an implementation strategy within an MVC framework

The view component of the prototype is a simple Python GUI from which the user can run a simulation of a timed sequence of transitions of a standard UK traffic signal (represented as simple coloured graphical widgets). In the prototype, this can be achieved with either a classic object-oriented state design pattern (object model) or via the new neuron-based state model. The GUI provides a selection widget to change the implementation model from the object-oriented version to the neuron-based PyNN implementation, and vice versa, and does so whilst the program is running (thus satisfying the strategy pattern requirement to be dynamically interchangeable). This is managed by the controller component, which employs the strategy pattern to assign the specific implementation model of the simulation and delegate to that state model sub-type accordingly. In fact, structurally the strategy pattern looks similar to the state pattern in that a context component (the controller in this case) delegates to one of a number of model implementation strategies that satisfy an abstract parent class protocol. There is, however, a distinct difference in dynamics between state and strategy patterns, with the intent of the strategy pattern being focussed only on encapsulation of the algorithms of a model. In this prototype MVC implementation, communication of state transitions from model to view, via controller, is facilitated via a publish-and-subscribe model using a queue data structure. In order to allow for both models to be interchanged during a single run of the prototype, this queue stores an abstracted enumeration of the four states of the traffic signal.

5. Conclusions

Programming with simulated neurons has two main benefits: neurocognitive modelling, and neuron inspired AI. Neurocognitive modelling of the brain using a simulated version of a relatively low-level and well understood primitive, i.e., neurons, will help advance understanding of neural and cognitive functioning. The benefits of neuron inspired AI are perhaps less obvious, but more immediately important. The waning of Moore's Law can be partially compensated with parallelism, so concurrency is significant. Programming in simulated neurons gives a very fine grained parallelism, using billions of processors. The advent of neuromorphic hardware takes advantage of this, but the software community does not really know how to take full advantage of these systems. Currently, there is no neuron-specific programming language. At some point in the future, dedicated neuron-oriented programming languages may exist, and the developers of such languages might look to base their APIs on established neural programming patterns.

The work described in this report is still in its early stages but a working prototype has been achieved as a proof of concept demonstration. At this time, a neuron-based equivalent of the classic state design pattern is tentatively proposed. There is some further work to be achieved before, for example, announcing this to the patterns community. For one thing, the prototype is based on a very simple model with a single predetermined transition from one state to another single state. The next stage is to develop the existing prototype to operate with a more complex state model in which the currently firing cell assembly has a choice of more than one possible transition to alternative cell assemblies. The authors are confident in achieving this, but only at that point might one accurately suggest a full state design pattern equivalent.

A very important aspect of this work is that this neuron-based pattern of programming has not simply been invented. Rather, it has been adapted from a technique that was developed by a computer scientist and his co-workers specialising in machine learning with neural networks, and who have not been consciously operating within the world of patterns. Key to the idea of proposing a design pattern is in the recognition of a technically sound, and repeatable, programming method and its representation. Uniquely, in this case, its representation is not object-oriented but neural. The authors view this as the beginning of an interesting strand of investigation. Mapping the neural prototype described in this report to a classic design pattern is a first step. Possibly, the authors may investigate the mapping of other well-known patterns from symbolic programming paradigms in terms of their usefulness in neural programming. However, there may well be neuron-specific patterns to be discovered, and it is reasonable to suspect that concurrency will be at the heart of some of them.

6. References

Brette R and Gerstner W (2005) Adaptive exponential integrate and fire model as an effective description of neuronal activity. Journal of Neurophysiology, 94: 3637-3642.

Buschmann F, Meunier R, Rohnert H, Sommerlad P, Stal M (1996) Pattern-Oriented Software Architecture: A System of Patterns. Wiley.

Byrne E and Huyck C (2010) Processing with cell assembles. Neurocomputing, 74: 76-83

Davison AP, Brüderle D, Eppler JM, Kremkow J, Muller E, Pecevski DA, Perrinet L, Yger P (2008) PyNN: a common interface for neuronal network simulators. Frontiers in Neuroinformatics, 2:11.

Fan Y and Huyck C (2008) Implementation of finite state automata using fLIF neurons. In: IEEE Systems, Man and Cybernetics Society, pp 74-78.

Furber S, Lester D, Plana L, Garside J, Painkras E, Temple S, Brown S (2013) Overview of the spinnaker system architecture. IEEE Transactions on Computers, 62(12): 2454-2467.

Gamma E, Helm R, Johnson R, Vlissides J (1995) Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.

Goodman D and Brette R (2013) Brian simulator. Scholarpedia 8(1): 10883.

Hebb D (1949) The Organization Of Behaviour. John Wiley & Sons.

Huyck C. (2009) A psycholinguistic model of natural language parsing implemented in simulated neurons. Cognitive Neurodynamics, 3, 317-330.

Huyck C, Evans C, Mitchell I (2015) A comparison of simple agents implemented in simulated neurons. Biologically Inspired Cognitive Architecture, 12: 9-19.

Huyck C and Fan Y (2007) Parsing with fLIF neurons. In: IEEE Systems, Man and Cybernetics Society, pp 35-40.

Huyck C and Mitchell I (2014) Post and pre-compensatory Hebbian learning for categorisation. Computational Neurodynamics, 8(4):299-311.

Plesser H, Diesmann M, Gewaltig M, Morrison A (2015) Nest: the neural simulation tool. Encyclopedia of Computational Neuroscience, 1849-1852.