Implementing Rules with Artificial Neurons

Christian Huyck[1] and Dainius Kreivenas[1]

Middlesex University, London NW4 4BT UK c.huyck@mdx.ac.uk
http://www.cwa.mdx.ac.uk/chris/chrisroot.html

Abstract. Rule based systems are an important class of computer languages. The brain, and more recently neuromorphic systems, is based on neurons. This paper describes a mechanism that converts a rule based system, specified by a user, to spiking neurons. The system can then be run in simulated neurons, producing the same output. The conversion is done making use of binary cell assemblies, and finite state automata. The binary cell assemblies, eventually implemented in neurons, implement the states. The rules are converted to a dictionary of facts, and simple finite state automata. This is then cached out to neurons. The neurons can be simulated on standard simulators, like NEST, or on neuromorphic hardware. Parallelism is a benefit of neural system, and rule based systems can take advantage of this parallelism. It is hoped that this work will support further exploration of parallel neural and rule based systems, and support further work in cognitive modelling and cognitive architecture.

Keywords: Rule Based System · Simulated Neurons · Cognitive Architecture · Compilation To Neurons

1 Introduction

Simulated and emulated neural systems can be used for practical applications, have massive parallelism, and can be used for exploring human and other animal neural processing. Rule based systems are a standard programming paradigm that has been widely used for expert systems and for cognitive modelling. Consequently, the ability to easily translate a rule based system into a neural system that can execute the same system has many potential uses in modern AI and cognitive science. This paper describes such a translation mechanism, and the neural execution of two particular rule bases (Monkeys and Bananas section 4.1, and the Tower of Hanoi section 4.2).

The idea is based around finite state automata (FSAs) as each rule is a simple FSA moving from one set of facts to another. Facts are implemented neurally as binary cell assemblies (CAs) (see section 2.2).

This translation ability expands the potential easy use of large neural systems, as large rule based systems can be directly translated to them. This supports the exploration of large, parallel rule based systems, and the exploration of neural cognitive architectures.

2 Literature Review

Rule based systems are powerful, have a long history, and are widely used in modern cognitive science (see section 2.1). FSAs are an important standard computational theoretical construct that can be readily implemented in neurons using binary CAs; CAs more broadly are important standard neuropsychological constructs that form the neural basis of concepts (see section 2.2). Biological neurons can be accurately simulated and emulated in modern computers; moreover, standard mechanisms are available to increase the reusability of neural systems (see sections 2.3 and 3.2).

2.1 Rule Based Systems

Rule based systems have been in use since at least the 1940s [17], and are Turing complete. They were important in early AI systems (e.g. [16]). Many expert systems in the 1970s and since have been implemented in rule based systems, in part because experts can explain their reasoning in rules, and thus it is relatively simple for a programmer to translate this expertise into a rule based program. Rules are also the most common mechanism for procedural knowledge in cognitive architectures, including ACT-R [2], Soar [13], and EPIC [12].

As the name suggests, rule based systems are built around if then rules. For example **if** *Banana at A, and Monkey at A* **then** *Monkey has Banana*. Facts, like *Banana at A* are boolean. Rule based systems are readily implemented on standard computers. While developing the system described in this paper, CLIPS [18] was used to develop particular systems, and as a test that the same results were produced. Rule based systems are often easier for people to learn to program than standard programming languages like Java.

In most rule based systems, one rule is applied in each time step. However, it is also possible to apply all supported rules in parallel. For example, the rule based component of EPIC [12] applies many rules in each cycle. This parallel application may be useful in cognitive modelling, but it is also useful as parallel processing. It is possible to have a parallel rule based system that fires thousands or even millions of rules in parallel, allowing a rapid processing speed increase.

While rules are simple, an FSA is perhaps simpler.

2.2 Finite State Automata and Cell Assemblies

Finite state automata are standard computational theoretical models based around states [14]. They can be used to recognise that an input is valid, so starting from a state, they will move to another state depending on the next input. They are powerful and widely used mechanisms, in for example compilers. They are, however, not Turing complete.

A long standing theory of neuropsychology is that concepts are represented by reverberating circuits of neurons called cell assemblies (CAs) [10]. These neurons have a large number of synapses to each other, and the weights of these synapses are large. So, if enough of the neurons start to fire, the neurons will

continue to fire causing a reverberating circuit. Once the CA starts to fire, it is said to be ignited. An ignited CA is a psychological short-term memory. When the neurons in the CA stop firing at an elevated rate, the CA is no longer in short-term memory.

While the topology of biological CAs is complex and poorly understood, it is relatively simple to develop binary CAs in simulated neurons. These binary CAs are either firing, or not. They persist indefinitely, unless inhibited by neurons outside the CA, which can switch the CA off.

These binary CAs can act as states in a finite state automata. Once a state is active (firing persistently), it will persist indefinitely. A second state can be activated by a combination of the current state, and input. The second state can inhibit the first, and the neural implementation of the automata has transitioned to the second state. Any FSA can be implemented in this fashion [6].

The attentive reader will note that while rule based systems are Turing complete, FSAs are not. How then can FSAs be used to implement rule based systems. The answer is in the neural facts. An FSA can be combined with an infinite tape to make a Turing machine, which is, as the name suggests, Turing complete. In this case, an infinite number of neurons to implement facts replaces the tape. Do note that any finite calculation can be done with a finite sized tape, and similarly with a finite number of neurons. This should not be particularly surprising as it has been shown that a system based on neurons is Turing complete [4].

2.3 Artificial Neurons

There a many artificial neural models with complexities varying from simple integrate-and-fire neurons [15], to compartmental models [11] and beyond. There is a trade-off between biological accuracy and computational efficiency. This work makes use of leaky-integrate-and-fire (LIF) models [3]. These point models are widely used in biological neural modelling, and are available on neuromorphic hardware [8]. Unlike standard Von Neumann architectures, neuromorphic hardware is not general purpose, but instead emulates neurons; this hardware supports a large degree of parallelism.

A simple description of LIF models is that they collect activation from other neurons. If a neuron collects enough activation to surpass its threshold, it fires, and sends activation to the neurons it is connected to. These can be modelled continuously, but the systems typical run in discrete time steps. If a neuron does not fire in a time step, some if its activation leaks away. The time step used in the simulations described below is 1 ms.

The connections between neurons are weighted uni-directional synapses. These weights may be positive (excitatory) or negative (inhibitory).

3 System

The system described in this paper translates rules to a neural implementation of those same rules. These rules are run in simulated neurons, but could easily be

run on neuromorphic hardware. In the simulations below, execution of the neural rule based system is done by externally stimulating the neural implementation of the initial facts, causing them to become persistently active (putting them into memory). The firing neurons in the fact spreads activation to the rules they support, causing the rules to be applied, putting new facts into memory, and removing old facts.

3.1 Translating Rules To Neurons

The translation system takes as input the rules defined as python structures, and the initial facts. It stores the rules internally as a dictionary of definitions, which determine what facts activate or deactivate other facts. It then takes each rule and translates it to one or more FSAs.

The translation engine uses the rules to create new facts by scanning the existing facts. In essence, these are the facts that may become true during execution. As new facts and rules are added to the system a recursive mapping process ensures that all new facts that are needed are created.

Through the system's recursive translation engine, the facts are mapped to rules. Every time a new fact or rule is added to the system, new facts may be added. For example, equation 1 is a simple rule approximating CLIPS syntax.

$$(MonkeyCanReach\ ?x) \Rightarrow (MonkeyGrab\ ?x) \\ (remove\ (MonkeyCanReach\ ?x)) \tag{1}$$

In the rule 1, the $?x$ is a variable. When it is defined, the system has no facts. If a fact ($MonkeyCanReach\ Banana$) is defined, the system will put the fact into the dictionary. It will also add the ($MonkeyGrab\ Banana$) fact to the dictionary because of rule 1. If the rule had a constant fact (e.g. ($MonkeyCanReach\ Banana$)) as the antecedent, it would have added that fact to the dictionary while reading the rule. The consequent of sets up an FSA that both adds new facts and removes the old ones.

With this rule and the associated two facts, the system will create an FSA that links the two facts so that if the ($MonkeyCanReach\ Banana$) fact becomes active, it will turn on the ($MonkeyGrab\ Banana$) fact, and in turn be turned off.

If a new fact is added, say $MonkeyCanReach\ Apple$, the corresponding $MonkeyGrab\ Apple$ fact will be created and another linking FSA for the rule will be created.

Note that in this context the fact is true if its associated state is on, and is made false by turning that state off. Of course, there is no turning on or off at this stage. The structure is just set up in an internal python dictionary. This structure relates to a Rete net[7] typically used in rule based engines.

Once all of the rules and initial facts are read, the system caches out the fact and rule structures to neurons and synapses. This involves translating each state into a CA, implemented in neurons and synapses. These CAs are linked so that

they implement the FSAs underlying the rules; this is done by adding synapses to turn fact states on and turn them off. There are extra assertion neurons, for turning the fact CAs on, and retraction neurons for turning them off (see figure 1).

If the rule has more than two if clauses, the system makes use of intermediate states. So, the if portion $A\&B\&C$, would be represented by an FSA with A and B turning on the intermediate state AB. The if clause is true if AB and C are both on.

The translation engine needs to make sure that every possibility is exhausted. One way the engine is made more efficient is to find the matching facts for a rule using fact groups. Each fact group is a named item that is stored in a key valued dictionary. Each dictionary value is a list of facts that belong to this group. This concept separates the facts into groups allowing rule conditions to be found and attributes matched faster.

Finally, the translation engine needs to set up external activation for the initial facts. This is done with PyNN spike source generators.

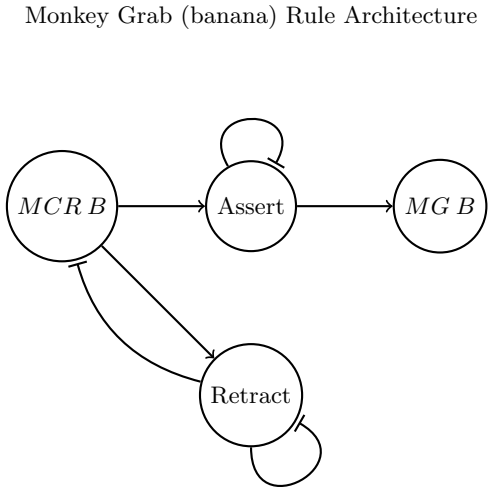

Fig. 1. Neural Architecture for the rule *MonkeyGrab* when a fact *MonkeyCanReach Banana* is added. Where each state is a CA. Arrows represent excitatory connections, and blocks represent inhibitory connecitons; so the MCRB CA turns on the retract CA, which in turn extinguishes MCRB and itself.

Figure 1 illustrates the neural CA structure of the rule from equation 1 along with the fact (*MonkeyCanReach Banana*). This shows that, when ignited, the CA for (*MonkeyCanReach Banana*) starts a spread of neural activation that leaves the (*MonkeyGrab Banana*) CA ignited, while the reach is extinguished.

This system is a simple rule based system translation engine. It converts rules and initial facts to simple FSAs. It then converts these to neurons, synapses, and spike sources. The neural system can then be run in a simulator or emulator (see section 4.1.

3.2 Simulating Neurons

The simulations described in this paper make use of the NEST [9] neuron simulator, and uses PyNN [5] as middleware. That is PyNN, python classes for managing neural net simulations, is used to specify the neural topology, and manages starting the simulation, initial inputs to the neurons, and recording the resulting firing. Once PyNN specifies the topology and initial inputs, it calls NEST to simulate the neurons.

The neurons that are used are adaptive exponential integrate and fire neurons [3]. These are leaky integrate and fire neurons, and the default parameters are used with three exceptions. The refractory period is increased from 1ms. to 2ms; the firing threshold is decreased from -50.0 to -53.0 mV; and the reset after firing is decreased from -65.0 to -70.0 mV. These neurons are a standard set used in the authors' neural FSA work to assure consistent firing behaviour.

Each state consists of 10 neurons, with the first eight being excitatory, and the last two inhibitory. The excitatory neurons connect to the other neurons in the state with the same weight, and the inhibitory neurons connect to the other neurons with the same weight. The supports regular firing speeds (every 5ms), once the state is ignited.

The time step used in these simulations is 1ms. This is to conform with SpiNNaker behaviour, which is closely tied to this 1ms. step. Consequently it should be relatively easy to translate this work to run on SpiNNaker. All code can be found at http://www.cwa.mdx.ac.uk/NEAL/NEAL.html.

3.3 Executing the Rules and Facts in Neurons

Fig. 2 displays the neural structure and the spike times for the *monkey grab banana* rule (equation 1). The vertical axis refers to neuron numbers, and the horizontal axis to time in ms. Each dot represents a neuron firing.

Fig. 2 shows that the initial fact $MonkeyCanReach\,Banana$ fires at 9ms. With no rule, it would repeatedly fire indefinitely. However, the rule is applied, and the assertion CA fires at 15ms and that ignites the $MonkeyGrab\,Banana$ fact CA at 21ms. The fact $MonkeyCanReach\,Banana$ is also retracted at 15ms by being inhibited by the retraction neurons. The whole simulation took only 28ms to progress from initial state to the end state. It will persist in this state indefinitely, unless other rules are available.

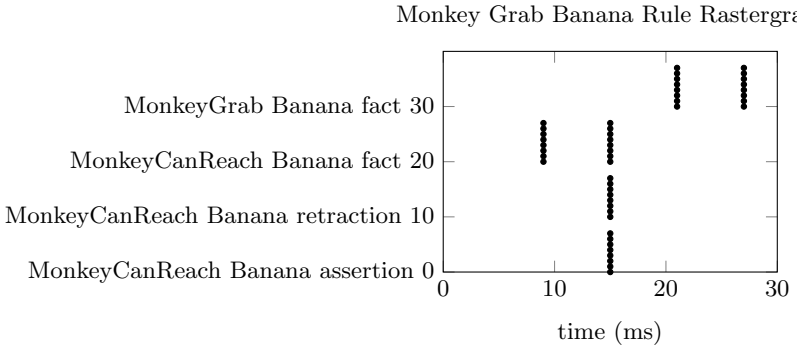

Fig. 2. Spike times for rule $MonkeyGrabBanana$. Neuron numbers represent: 0-9 - $MonkeyGrabBanana$ assertion neurons; 10-19 - $MonkeyCanReachBanana$ retraction neurons; 20-29 - $MonkeyCanReach\ Banana$ fact; 30-39 - $MonkeyGrab\ Banana$ fact

4 Examples

Two examples of complete systems are provided. The first is a simple monkeys and Bananas problem. The second shows the more complex Tower of Hanoi problem.

4.1 Monkeys and Bananas

The monkey and banana problems is a good old fashioned artificial intelligence problem. The problem involves a monkey and bananas suspended from the ceiling. There is also a chair in the room and the only way to reach the fruit is to move the chair, and stand on it to reach the bananas.

The following scenario describes rules and facts in a format approximating CLIPS syntax. There are three initial facts: $(ChairAt\ 2)$, $(Fruit\ banana\ 0)$, $(Fruitapple1)$. The $ChairAt$ fact represents the position of the chair. The $Fruit$ facts represent the type of fruits and their position. As long as the chair position is the same of that of a fruit, it is considered that the fruit can be reached. The scenario also consists of four rules: $MonkeyGrab$ equation 1 defined earlier, $EatFruit$ equation 2, $MonkeyHasFruit$ equation 3 and $PushChair$ equation 4.

$$\begin{aligned}(MonkeyHas\ ?type) \Rightarrow \\ (assert\ (MonkeyAte\ ?type)) \\ (remove\ (MonkeyHas\ ?type))\end{aligned} \tag{2}$$

$$\begin{aligned}(ChairAt\ ?position)\&(Fruit\ ?type\ ?position) \Rightarrow \\ (assert\ (MonkeyHas\ ?type)) \\ (remove\ (Fruit\ ?type\ ?position))\end{aligned} \tag{3}$$

$$(Fruit\ ?type\ ?position)\&(not\ ChairAt\ ?position) \Rightarrow \\ (assert\ (ChairAt\ ?position)) \\ (remove\ (not\ ChairAt\ ?position)) \tag{4}$$

Note that the *PushChair* rule has the *not* operator, meaning the *ChairAt* fact is not in the same position as the *Fruit* fact. This is done by a dictionary process that links facts where the variable *?position* differs between the two antecedent clauses.

The system converts the rules and facts to neurons. When it is run, it completes the task. The spike times are shown in figure 3.

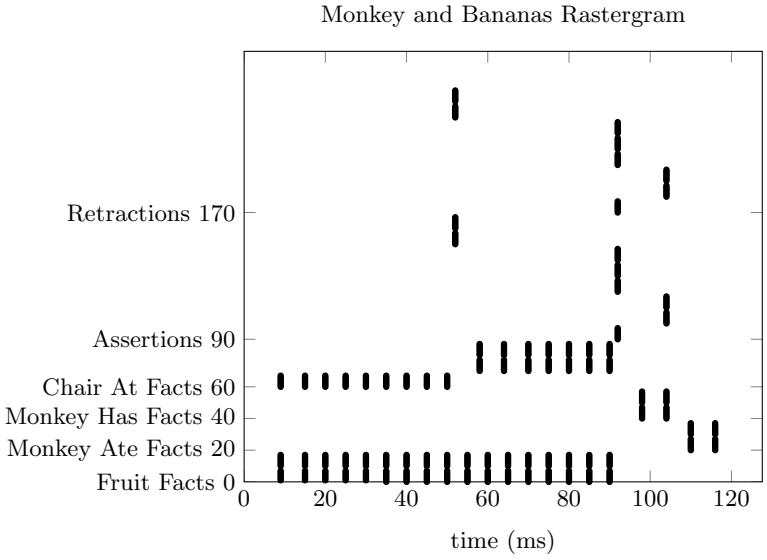

Fig. 3. Spikes of the neurons that implement the Monkeys and Bananas Problem.

The bottom 20 neurons represent the initial *Fruit* facts. As figure 3 shows, the fruit is grabbed at 90 ms. Neurons 21-40 represent *MonkeyAte* facts and 41-60 represent *MonkeyHas* facts. Neurons 61-90 are the *ChairAt* facts. At 58ms, the rule is applied changing the facts. Neurons 230-249 fire and retract the (*ChairAt* 2) fact. Neurons 150-169 fire and assert the (*ChairAt* 0) and (*ChairAt* 2) facts.

The parallel nature of the execution of the rules leads to an unanticipated, and probably unwanted, effect. The system does not understand that the *ChairAt* fact is a single object Therefore, the 61-70 neurons of (*ChairAt*2) fact, after 58ms

are replaced by two facts ($ChairAt1$) and ($ChairAt0$) represented by 71-90 neurons to match both the initial $Fruit$ facts. This triggers the $MonkeyHasFruit$ rule for both apple and banana. From this point on the conflict carries through to the end of the simulation, which takes around 110ms to rest at the end state.

The rest of the spikes belong to retractions and assertions. The 91-170 neurons are the assertions and 171-250 are the retractions, which get activated to transition the simulation through different states. The final state, which will persist indefinitely, is ($MonkeyAte\ apple$) and ($MonkeyAte\ banana$).

4.2 Tower of Hanoi

The Tower of Hanoi is a widely known problem and widely used problem involving three towers and a number of discs of increasing size. Each disc can fit on the base of a tower, or on a larger disc already on a tower, but not on a smaller disc. Only the top disc of a tower can be moved to another tower. Fig. 4 shows an example starting state of the problem with four discs.

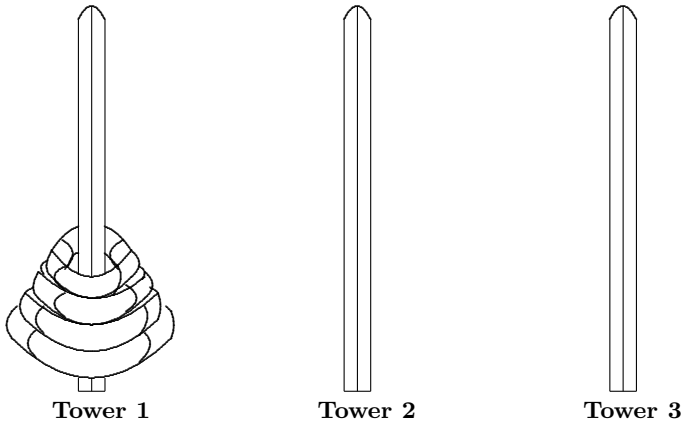

Fig. 4. The Tower of Hanoi four disc start state.

To solve the Tower of Hanoi, a goal stack is used. The system uses a series of facts to represent the stack. This paper presents facts as tuples in parenthesis, like CLIPS. So, the fact ($StackTop\ 3$) says that the stack has three elements in it. The stack contains two types of items: goals, and moves. A move fact is of the form ($Stack\ ?stackLevel\ Move\ ?disc\ ?from\ ?to$). Again, as in rules, variables are prefixed with a question mark. So, the stack at the *stackLevel*, moves, the disc *disc* from tower $from$ to tower *to*. So if the fact were instantiated

as ($Stack$ 3 $Move$ C 1 3), the disc C would move from tower 1 to tower 3 when the third stack item was popped.

The stack can also contain goals. They are of the form ($Stack$?$stackLevelGoal$?$topDisc$?$bottomDisc$?$from$?to). So there is goal to move the discs between $topDisc$ and $bottomDisc$ from tower $from$ to tower to. The first goal that is added to the stack for one three disc problem is ($Stack$ 1 $Goal$ A C 1 3), which states move the discs A to C from tower 1 to tower 3.

The system consists of five rules: addDisc, addFinalDisc, makeMove, goals To-Goals, and goalsToMoves. The first two rules are used to initialize the facts with the initial positions of the discs. Typically this has discs A to N on tower 1. The initial fact that says how large N is must be specified.

The makeMove rule accounts for the primitive disc moves, and is described by equation 5. It is relatively complex, but even very complex rules can be cached out to neurons. The rule only works on the item that is on top of the stack. If this item is a $Move$, it removes it by popping the stack (the bottom two lines), and moves the disc (the first two clauses after the arrow).

$$\begin{aligned}(StackTop\ ?x)\&(Stack\ ?x Move\ ?disc\ ?from\ ?to)\& \\ (Disc?disc\ ?from) \Rightarrow (assert(Disc?disc\ ?to)) \\ (remove(Disc\ ?disc\ ?from)) \\ (remove(StackTop\ ?x))(assert(StackTop\ (-?x\ 1))) \\ (remove(Stack\ ?x Move\ ?disc\ ?from\ ?to))\end{aligned} \tag{5}$$

Note that these rules make use of addition (+) and subtraction (−), and they have to be handled in the code. This is done by mapping out the structure so that the assertion creates the appropriate new fact. So, this is a dictionary process that calculates the range of possible integers, and makes assertions and retractions appropriately. In this code it works for addition and subtrction, but could readily be done for other operations such as multiplication and division.

The remaining two rules, goalsToGoals and goalsToMoves, handle subgoaling. The goalsToGoals rules accounts for moving large amounts of discs. If the current top of the stack has the goal to move more than two discs from one place to another, the goal is replaced with two subgoals and a move. For instance, if the goal is ($stack$?top $Goal$ A C 1 3), it is replaced with ($stack$?top $Goal$ A B 2 3), ($stack$ (+?top 1) $Move$ C 1 3), and ($stack$ (+?top 2) $Goal$ A B 1 2). The goalsToMoves rule accounts for moving two discs. It replaces a goal with two discs with three moves. For instance, if the goal is ($stack$?top $Goal$ A B 1 3), it is replaced with ($stack$?top $Move$ A 2 3), ($stack$ (+?top 1) $Move$ B 1 3), and ($stack$ (+?top 2) $Move$ A 1 2). These five rules can solve any problems with any number of discs by using the minimum amount of moves required.

Figure 5 displays the rastergram of the neuron spike times of the Tower of Hanoi problem with three discs implemented using the rules described above converted to neurons. As the neurons are binary CAs, and all the neurons in the CA behave the same, only one neuron per CA is shown in the rastergram. The neurons represented in the figure shows the full system of neurons.

The bottom 80 neurons represent the *discAt* facts, so the movement of the largest disc from tower 1 to tower 3 can be seen at 600 ms. The neurons from 590-909 are the internal neurons used for combining multiple if clauses. Neurons 450-589 are retraction neurons. Neurons 310-449 are assertion neurons; 210-309 are the goal and move facts; and 180-209 are the towers, which stay on throughout the simulation. Neurons 120-179 are the *stackTop* neurons; 110-119 are the neurons for the initial fact saying there are three discs; and neurons 80-109 are the neurons for adding the discs at the start of the simulation.

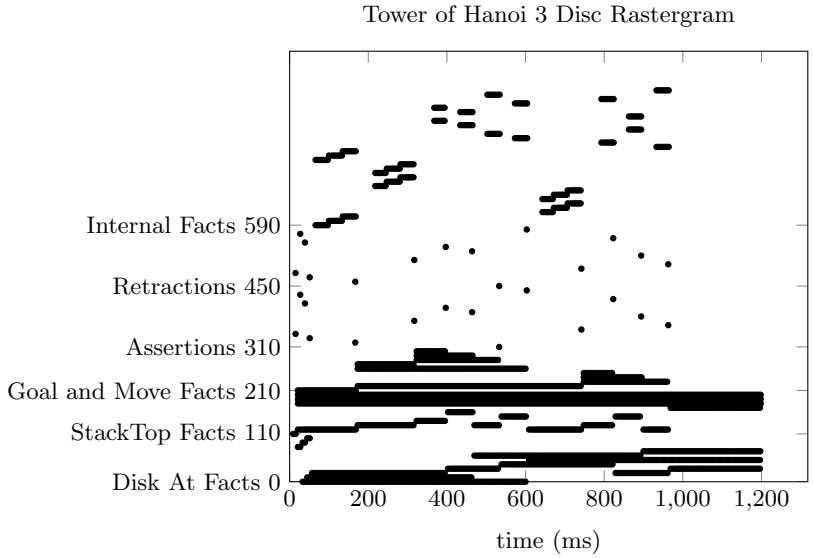

Fig. 5. Tower of Hanoi 3 disc problem neuron spikes. Neuron number on the horizontal axis with many labels omitted.

Figure 6 shows the simulated times between moves on the five disc Tower of Hanoi problem. This closely echoes the human times and cognitive model times reported by Altmann and Trafton [1]. The difference is that times of the neural system are about 20 times faster.

5 Conclusion

This paper has briefly discussed the importance of rule based systems. It has shown how they can be automatically translated into neural systems.

Though this paper shows that rule based systems can be readily implemented in neurons, and indeed can be directly translated, a number of improvements can be made to the system. There are relatively straight forward issues about

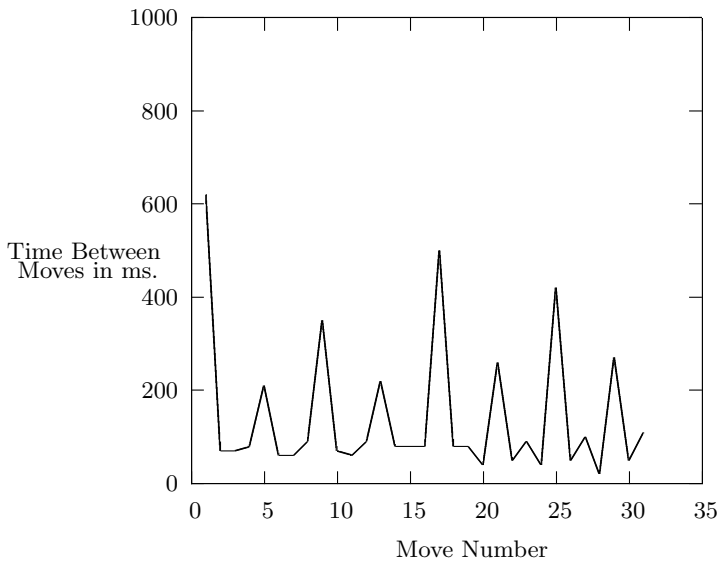

Fig. 6. Latencies for Moves in the 5 Disc Tower of Hanoi Problem

usability, and more complex issues. Relatively straight forward improvements include a parser for the rule based system, a closer link to, for instance, CLIPS, and tools for recognition of issues like the forking of multiple facts as shown in section 4.1.

More complex issues include parallelism, number of neurons, expansion to other types of memory, and cognitive improvements. For instance a large number of neurons might be needed for certain systems; if a rule based system were to use real numbers, it would need an uncountably infinite number of CAs to cope with this. The actual number could be determined at translation time, but it is possible that that number could be very large. The translation system might also note when this number is large, and warn the user. The translation process assumes that possible values are known before hand. Dynamically interacting with a virtual environment via neural facts and neural outputs can lead to the introduction of new facts. In this case, the system would have to specify a range of possible values.

One of the benefits of neural systems is that time emerges naturally. Biological neurons have a time course, and the artificial versions make explicit use of this time. This translation system will support further exploration of large parallel rule based systems, and provide communication with research in standard cognitive architectures.

Acknowledgment

This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 720270 (the Human Brain Project).

References

1. Altmann, G., Trafton, J.: Memory for goals: An activation-based model. Cognitive Science **25(1)**, 39–83 (2002)
2. Anderson, J., Lebiere, C.: The Atomic Components of Thought. Lawrence Erlbaum (1998)
3. Brette, R., Gerstner, W.: Adaptive exponential integrate-and-fire model as an effective description of neuronal activity. J. Neurophysiol. **94**, 3637–3642 (2005)
4. Byrne, E., Huyck, C.: Processing with cell assemblies. Neurocomputing **74**, 76–83 (2010)
5. Davison, A., Brüderle, D., Eppler, J., Muller, E., Pecevski, D., Perrinet, L., Yqer, P.: PyNN: a common interface for neuronal network simulators. Frontiers in neuroinformatics **2** (2008)
6. Fan, Y., Huyck, C.: Implementation of finite state automata using flif neurons. In: IEEE Systems, Man and Cybernetics Society. pp. 74–78 (2008)
7. Forgy, C.: Rete: A fast algorithm for the many pattern/many object pattern match problem. In: Readings in Artificial Intelligence and Databases, pp. 547–559 (1988)
8. Furber, S., Lester, D., Plana, L., Garside, J., Painkras, E., Temple, S., Brown, A.: Overview of the spinnaker system architecture. IEEE Transactions on Computers **62**(12), 2454–2467 (2013)
9. Gewaltig, M., Diesmann, M.: NEST (neural simulation tool). Scholarpedia **2**(4), 1430 (2007)
10. Hebb, D.O.: The Organization of Behavior: A Neuropsychological Theory. J. Wiley & Sons (1949)
11. Hodgkin, A., Huxley, A.: A quantitative description of membrane current and its application to conduction and excitation in nerve. Journal of Physiology **117**, 500–544 (1952)
12. Kieras, D., Wood, S., Meyer, D.: Predictive engineering models based on the epic architecture for a multimodal high-performance human-computer interaction task. ACM Transactions on Computer-Human Interaction **4:3**, 230–275 (1997)
13. Laird, J., Newell, A., Rosenbloom, P.: Soar: An architecture for general cognition. Artificial Intelligence **33,1**, 1–64 (1987)
14. Lewis, H., Papadimitriou, C.: Elements of the Theory of Computation. Prentice-Hall, Inc. Englewood Cliffs, New Jersey (1981)
15. McCulloch, W., Pitts, W.: A logical calculus of ideas immanent in nervous activity. Bulletin of Mathematical Biophysics **5**, 115–133 (1943)
16. Newell, A., Simon, H.: The logic theory machine–a complex information processing system. IRE Transactions on information theory **2(3)**, 61–79 (1956)
17. Post, E.: Formal reductions of the general combinatorial decision problem. American journal of mathematics **65:2**, 197–215 (1943)
18. Riley, G., Culbert, C., Lopez, F.: C language integrated production system. Tech. rep., NASA (1989)