

The Petri Net Radial Basis Function Perceptron

Copyright © 2002, Andrew Seely

Abstract—This paper introduces the Petri Net Radial Basis Function Perceptron (PNRBFP), a modified Petri Net that exhibits behavior equivalent to that of a typical radial basis function Perceptron when used in neural networking applications under certain domain restrictions. The PNRBFP makes use of modified transitions to perform basis function calculations and 'fuzzy' style tokens to transport values of basis function outputs. In all other respects the PNRBFP is a standard Petri Net and will benefit from established Petri Net analysis and implementation tools.

Index Terms—Petri Nets, Neural Networks, Radial Basis Functions, Perceptrons.

I. INTRODUCTION

This paper introduces the Petri Net Radial Basis Function Perceptron (PNRBFP), a modified Petri Net that functions as a radial basis function (RBF) Perceptron. The PNRBFP draws on features of simple Petri Nets, Fuzzy Petri Nets, Fuzzy Neural Petri Nets, and RBF Perceptrons. These technologies are briefly defined below.

A. Petri Nets

A simple Petri Net is a directed graph structure formally defined as a five-tuple [1], $N = \{ P, T, I, O, M \}$, where P is the set of *places* $\{p_1 \dots p_n\}$, T is the set of *transitions* $\{t_1 \dots t_m\}$, I is the set of *arcs* from a subset of places to each transition, defined per transition as $I(t_k) = \{ P_k \}$, where $P_k \subset P$. O is the set of arcs from each transition to a subset of places, defined per transition as $O(t_k) = \{ P_k \}$, where $P_k \subset P$. M is the set of *token counts* per place and may be written $M(p_k) = [\text{token count at } p_k]$. Additionally, M may be written as the ordered set of token counts for all of P , $M = \{ \text{count}_1, \text{count}_2, \dots, \text{count}_n \}$

Data flows through a Petri Net in the form of tokens. The state of a Petri Net is written in terms of M . Tokens move within the net through activity of the transitions; a transition t_k is considered *activated* when every place in $I(t_k)$ has at least one token. An activated transition will *fire*, which removes a

single token from every place in $I(t_k)$ and creates a single token in every place found in $O(t_k)$. A Petri Net is considered *non-deterministic*, since there are no rules or laws governing which of a group of simultaneously activated transitions will fire first [2]. Transitions are considered to be *in conflict* when the firing of one activated transition will cause another to become deactivated.

B. Fuzzy Petri Nets and Fuzzy Neural Petri Nets

Fuzzy Petri Nets (FPN) use modified Petri Nets to create logic networks capable of representing 'fuzzy rules' [3]. An FPN is built using the same basic structures as the basic Petri Net described above, with several important modifications. A fundamental change is the creation of a *fuzzy token*, a token capable of conveying a real number value n : $0 \leq n \leq 1$. These fuzzy tokens are passed through the FPN by transitions that incorporate a thresholding feature to simulate decision-making; if a transition t_k is to fire, the fuzzy tokens on $I(t_k)$ must be greater than some threshold value [3].

Fuzzy Neural Petri Nets (FNPN) expand the FPN concept to allow the Petri Net structure to function not only as a decision-making tool but also as a learning tool. The FNPN uses fuzzy tokens like the FPN, but adds *weighted arcs* to change the fuzzy values as they pass between nodes. In addition, the FNPN employs *thresholding transitions* to simulate the behavior of a simple neural network node, effectively allowing the FNPN to create an output of *true* or *false* based on the 'input' tokens it receives [4].

C. Perceptrons

The Perceptron is a basic neural networking element that accepts input typically as a set of boolean numbers, optionally pre-processes the input set into different forms, adjusts these modified inputs with weight multipliers, and compares the sum of these values against a threshold θ . Sums greater than or equal to θ cause the Perceptron to be *true*, while sums less than θ cause the Perceptron to be *false* [5]. The set of inputs for a Perceptron is typically written as X , where X is an ordered set of n values presented to the Perceptron as input: $X = \{ x_1, x_2, \dots, x_n \}$. The set of weights associated with input is typically written as W . In the Perceptron, there is not necessarily a one-to-one correlation between inputs and weights, so W is the ordered set of m multiplier values applied

Manuscript received October 10, 2002.

Andrew Seely is a lecturer for the computer studies department of the University of Maryland University College, European Division, Im Bosseldorn 30, Heidelberg 69126, Germany (e-mail: aseely@faculty.ed.umuc.edu).

to post-processed Perceptron input: $W = \{w_1, w_2, \dots, w_m\}$. The Perceptron's input X is determined during practical application. The nature of the pre-processing stage is determined when the Perceptron is designed and is generally static. W and θ are adjustable as part of the Perceptron training process [6].

D. Radial Basis Function Perceptrons

The Perceptron can be modified to enhance its effectiveness given a decision space that is non-linear. The addition of a basis function layer between pre-processing and weight multiplication will allow the Perceptron to be *true* over a much more varied input. It has been shown that a Perceptron can be used to model any function using an arbitrary number of basis functions [7]. A basis function is simply one of potentially many functions that may be summed to give a single coherent answer. A special class of basis functions is the radial basis function (RBF), a function with output that increases or decreases monotonically with the distance from a central point [8]. A function displaying this property and typically found used in Perceptron context is the Gaussian function shown in Formula 1. A Perceptron that uses a basis function layer with RBFs is called an RBF Perceptron.

$$h(x) = \frac{1}{e^{-\frac{(x-c)^2}{r^2}}} \quad (1)$$

II. THE PETRI NET RADIAL BASIS FUNCTION PERCEPTRON

The PNRBFP is a Petri Net-based Perceptron that mirrors the functionality of the Perceptron described in section I by making modifications to Petri Net primitives. These modifications allow the PNRBFP to maintain the basic Petri Net structure while increasing its inherent processing power to be comparable to that of an RBF Perceptron.

A. Transition Types

The transitions used in the PNRBFP are of four different types, shown graphically in Fig. 1. Regular, unmodified Petri Net transitions are used alongside three special-purpose transitions: The RBF Transition, the Concentrator Transition, and the Threshold Transition. The RBF Transition requires a single place connected on a single input arc. When activated, the RBF Transition will fire until this place is empty. Internally, a 'token counter' will record the number of tokens removed from the input place. The RBF Transition creates a *stage one token* from the output of the RBF function, using the token count as the RBF function input. The RBF Transition defined

here uses the Gaussian function from Formula 1 and has adjustable attributes for the Gaussian radius and center.

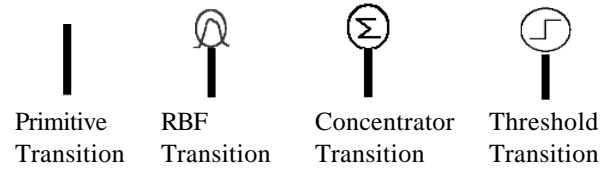


Fig. 1. Types of transitions used by the PNRBFP

The Concentrator Transition requires a single input arc, like the RBF Transition, but each token in its input place is expected to be a stage one token. The Concentrator Transition removes all stage one tokens from its input place and internally sums their values. A single *stage two token* with this summation is created in the concentrator's single output place.

The Threshold Transition requires a single input place, which should only hold stage two tokens. The Threshold Transition has a single attribute θ , equivalent to θ in a Perceptron or threshold learning unit [5]. If the stage two token value is greater than or equal to θ , the Threshold Transition outputs a single primitive token to its output place or places to represent a *true* value result. The Threshold Transition in the PNRBFP behaves in a fashion similar to the Fuzzy Neural Petri Net thresholding transition described by [4].

B. Places and Tokens

Places in the PNRBFP are ordinary primitive Petri Net places. Tokens are of three varieties: Primitive, as described in [2], *stage one*, and *stage two*. The difference between the three types of PNRBFP tokens is a matter of value. A primitive token carries no value other than its presence and is therefore limited to the domain of true (present) and false (absent). A stage one token can carry a real number value n , where $1 \geq n \geq 0$, in the same way that the tokens in both the Fuzzy Petri Net [3] and Fuzzy Neural Petri Net [4] carry fuzzy truth values. A stage two token can carry any real number n , where $n \geq 0$. Conceptually, the purposes of the two types of value-carrying tokens are distinct. The stage one token exists to transmit the output of the RBF Transition to the concentrator. The stage two token serves to carry the Concentrator's output value, which is the summation of all stage one tokens and could potentially be very large.

Fig. 2 shows a fully connected PNRBFP. This PNRBFP is functionally equivalent to the RBF Perceptron described in [5]. The input set X is represented by $p_{1..3}$, the weight set W is represented by the number of arcs leading into p_{10} : $W = \{w_1, w_2, w_3\} \sim \{O(t_9), O(t_8), O(t_7)\} = \{3, 1, 2\}$. For purposes of this discussion, RBF Transitions $t_{4..6}$ will have their radius and center values set as: $H = \{(5,3), (7,2), (9,9)\}$.

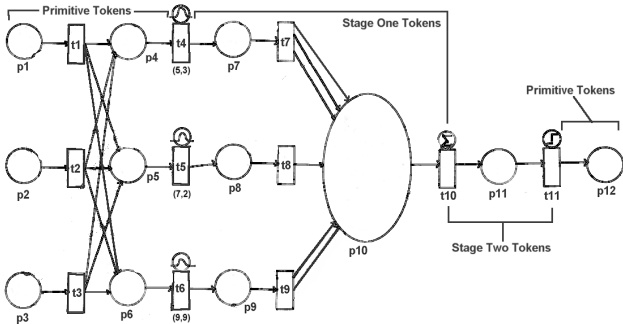


Fig. 2. PNRBFP showing token type zones and RBF radius and center settings

C. Basic Behavior

Logically, the PNRBFP behaves exactly like the RBF Perceptron. Boolean or integer inputs are entered in the input places as primitive tokens. In the case of Figure 2 the input places are $p_{1..3}$. Integer inputs are represented by multiple tokens in a place. The primitive transitions $t_{1..3}$ read the number of tokens in each input place and populate the next layer of places with these tokens. This first layer of transitions can be considered the point of interconnection, distributing the input across all basis functions. It follows that less-than-full interconnectivity is achieved by simply removing select output arcs from these transitions.

After all inputs have been distributed to the intermediate places, each RBF Transition read the number of tokens in its intermediate places, introduces this count to its basis functions, and creates the stage one token with the RBF result in its respective output place. The stage one token is then removed by a primitive transition that acts as a weight by multiplying the stage one token's overall influence with multiple output arcs. Weights are limited to integer values; a weight of three, for example, is achieved by connecting three output arcs from this transition.

All weight-enacting primitive transitions connect their output arcs to a single, common place. It is important to remember that this place is expected to contain only stage one tokens. The Concentrator Transition reads the aggregate collection of stage one tokens and computes the sum of all their values. This sum is used to create a stage two token, which is deposited in a single output place. The stage two token is read by the Threshold Transition; if the stage two value is greater than or equal to the Threshold Transition attribute θ , a primitive token is placed in the output places of the Threshold Transition to represent a *true* processing result.

D. Synchronization

PNRBFP processing requires synchronization between its stages in order to accurately process its data. For example, the

non-deterministic nature of Petri Nets would allow t_4 in Fig. 2 to fire and process input to its RBF before $t_{1..3}$ were finished transferring their input values to the place at $I(t_4)$. A very simple solution is the addition of negation arcs as defined in [1] and [2]. Negation arcs prevent transitions at each processing point from firing until their predecessor transitions complete their firing cycles. Fig. 3 shows the PNRBFP from Fig. 2 with negation arcs for synchronization control.

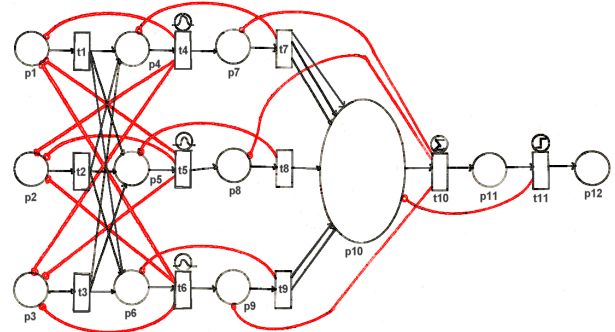


Fig. 3. PNRBFP with negation arcs for synchronization

The negation arcs ensure synchronization by preventing premature transition firing. Since a transition is considered enabled whenever all places on the transitions $I(t)$ set have tokens, the negation arcs ensure that the transition will not fire until the $I(t)$ places of its predecessors have been fully processed. In the context of PNRBFP flow, we can see that the RBF transitions $t_{4..6}$ will not fire until the input places $p_{1..3}$ have been completely emptied by transitions $t_{1..3}$. If $p_{1..3}$ are empty, it is assured that places $p_{4..6}$ have been populated since the firing of a transition is logically a non-divisible event.

Similarly, transitions $t_{7..9}$ will not fire until $p_{4..6}$ are empty, respectively. This prevents the weight-enacting transitions from prematurely accepting RBF Transition output. The Concentrator Transition at t_{10} will not fire until places $p_{7..9}$ are empty, assuring a full aggregation of stage one tokens in place p_{10} . Finally, the Threshold Transition at t_{11} will not fire until p_{10} is empty, enforcing a complete summary from the concentrator transition. For purposes of this paper, the PNRBFP with synchronization control shown in Fig. 3 will be considered the standard PNRBFP.

III. TRAINING THE PNRBFP

The PNRBFP is adjustable by modification of the attributes of the various non-primitive transitions. Change in the number of internal processing nodes would require significant labor, but this type of work is more in the realm of neural network design.

Training the PNRBFP is a matter of adjusting the center and radius for each RBF transition, the number of output arcs for each weight-enabling primitive transition, and θ for the threshold transition. Changing the number of weight arcs is a structural change to the underlying Petri Net, but simply

entails changing the $O(t)$ arc set for the weight-enabling primitive transition. Changing the radius and center for each RBF Transition and θ for the Threshold Transition are attribute changes rather than structure changes, and so are conceptually and programmatically very easy. Learning techniques for determining actual change values for these attributes are beyond the scope of this paper but are discussed in detail in the references [5], [6], [7], [8].

IV. PNRBFP BY EXAMPLE

The processing flow of the PNRBFP may be easily demonstrated using the example scenario in Table 1.

TABLE 1
RBF PERCEPTRON SAMPLE VALUES

$X = \{ 1, 0, 1 \}$
 $W = \{ 3, 1, 2 \}$
 $H = \{ (5,3), (7,2), (9,9) \}$
 $\theta = 3.000$

Input $X = \{ 1, 0, 1 \}$, which equates to an initial state marking of $M = \{ 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0 \}$ for the PNRBFP as shown in Fig. 4. Notice that the standard definition of a Petri Net still applies to the PNRBFP; in practical implementation care must be taken to differentiate between the various non-primitive structures, but for purposes of analysis and Petri Net-context definition the PNRBFP remains a five-tuple as defined in section I.

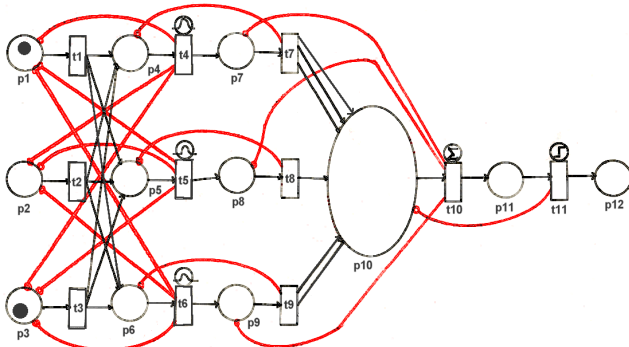


Fig. 4. PNRBFP example, part one: Input

After the initial tokens have been placed in $p_{1..3}$, the transitions $t_{1..3}$ will fire if they are enabled. In this example, Transitions t_1 and t_3 will each fire one time and t_2 will not fire at all. The firing of t_1 will place a single token in each place $p_{4..6}$, and the same will occur for the firing of t_3 causing $p_{4..6}$ to each have two tokens as shown in Fig. 5. All three RBF Transitions $t_{4..6}$ are enabled based on having tokens on their respective $I(t)$ sets, and all three are synchronized by nature of their negation arcs connecting to the input places. Since all input places are now empty, the RBF Transitions are able to fire.

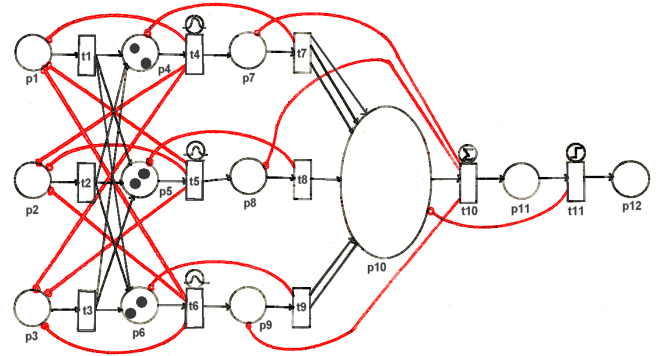


Fig. 5. PNRBFP example, part two: Interconnectivity

Each RBF Transition removes and counts the transitions from its respective $I(t)$ place. Since each place $p_{4..6}$ has two tokens, the RBF for each RBF Transition is given an input $x = 2$. The RBF Transition t_4 now has values $x = 2, r = 5, c = 3$. Using Formula 1, the stage one token deposited in p_7 has a value of 0.852. The RBF Transition t_5 has values $x = 2, r = 7, c = 2$, creating a stage one token in p_8 with a value of 1.000. The RBF Transition t_6 has values $x = 2, r = 9, c = 9$, creating a stage one token in p_9 with a value of 0.141. Fig. 6 demonstrates the output of the RBF Transitions.

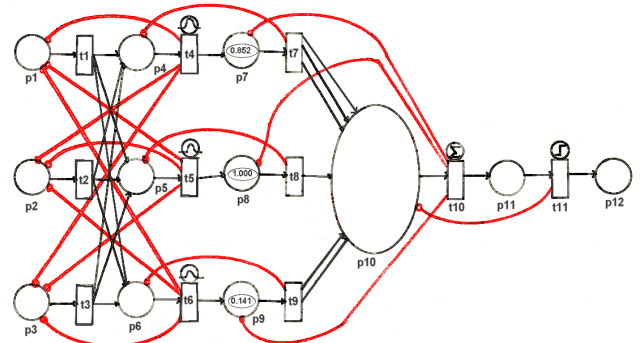


Fig. 6. PNRBFP example, part three: RBF output

The weight-enacting primitive transitions at $t_{7..9}$ each read the stage one tokens from places $p_{7..9}$, respectively, when their synchronizing negation arcs show that the input arcs of the RBF transitions have indeed been emptied. The transition t_7 removes the stage one token with value 0.852 from p_7 and deposits two copies of it in p_{10} , representing the weight w_1 of 2. The transition t_8 removes the stage one token with value of 1.000 from p_8 and deposits a single copy of it in p_{10} , representing the weight w_2 of 1. The transition t_9 removes the stage one token with value 0.141 from p_9 and deposits three copies of it in p_{10} , representing the weight w_3 of 3. The resulting marking with stage one tokens is shown in Fig. 7.

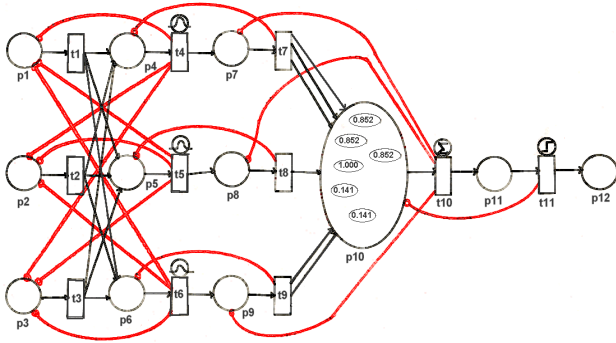


Fig. 7. PNRBFP example, part four: Influence of Weights

When all RBF Transition outputs have been read and transferred, with weights, to p_{10} , the Concentrator Transition is clear to fire. The Concentrator Transition t_{10} will repeatedly remove tokens from p_{10} until p_{10} is empty, adding the value of each stage one token to the value of a single stage two token. When p_{10} is empty, t_{10} deposits the stage two token with value $(0.852 * 2) + (1.000 * 1) + (0.141 * 3) = 3.127$ in p_{11} for reading by the Threshold Transition, as shown in Fig. 8.

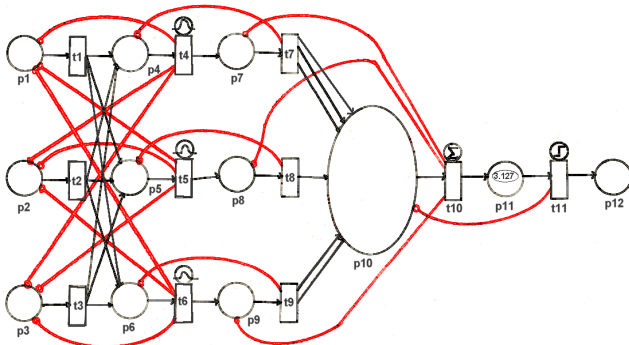


Fig. 8. PNRBFP example, part five: Concentrator output

The Threshold Transition t_{11} is free to fire when place p_{11} has been completely emptied by the Concentrator Transition. The Threshold Transition removes the stage two token from p_{11} and compares its value to θ . In this example, the stage two token value 3.127 is greater than $\theta = 3.000$, so a primitive token is deposited in the output place p_{12} . Thus, the PNRBFP as defined in Table 1 has an output of *true*.

V. PNRBFP COMPLEXITY

The number of computations a PNRBFP would take for a given input is directly related to the sum of all input values and can be demonstrated in seven steps. Returning to Fig. 3 for reference and assuming a fully connected PNRBFP in general: Input transitions $t_{1..3}$ will all execute a remove token operation for each input token in their respective input places; step one is the total number of remove token operations is thus equal to the total number of tokens placed in $p_{1..3}$.

Each transition $t_{1..3}$ will then deposit the same number of tokens it read into every place in its output $O(t)$ set. Since in a

fully connected Perceptron the same aggregate input quantity is presented to all basis functions, step two consists of the value of step one multiplied by the number of basis functions in the PNRBFP.

Step three consists of the RBF Transitions each reading the individual tokens in their respective input places. Since each RBF input place contains the quantity from step one, step three consists of the number of computations from step one multiplied by the number of basis functions. Step three has the same value as step two.

Step four is in two parts. First, the input is given to the RBF function for computation. Second, the RBF output is used for a single create token operation. Step four consists of two operations multiplied by the number of basis functions.

Step five involves the weight transitions. Each weight transition executes a single remove token to read the stage one token created by the RBF Transition. Then each weight transition executes as many create token operations as it has weight arcs. Thus, step five consists of the number of basis functions plus the total number of weights.

In step six, the Concentrator Transition executes a remove token operation for every token in its input place. One token is created for each weight arc from step five, so step six consists of the total number of weights plus a single create token operation.

Step seven removes the single token that is the output from step six, executes a comparison of the token value against the threshold value, and may or may not create a single token as output. Step seven thus consists of two or three operations.

Assigning M as the number of input tokens to the PNRBFP, N as the number of basis functions, and W as the total number of weights, the total number of computations for a PNRBFP can be stated: M (step one) + $M * N$ (step two) + $M * N$ (step three) + $2N$ (step four) + $M + N$ (step five) + $W + 1$ (step six) + 3 (step seven) = $M + MN + MN + 2N + M + N + W + 1 + 3 = 3N + 2MN + 2M + W + 4$. Since M and W are constant after the PNRBFP has been built, it remains that the number of computations required by the PNRBFP grows in a linear fashion as the number of inputs N increases.

REFERENCES

- [1] T. Murata, "Petri Nets: Properties, Analysis, and Applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541-580, 1989.
- [2] J. L. Peterson, "Petri Nets," *ACM Computing Surveys*, vol. 9, no. 3, pp. 221-252, 1977.
- [3] C. Looney, "Fuzzy Petri Nets for Rule-Based Decision Making," *IEEE Transactions on Systems, Man, and Cybernetics*, vol.18, no. 1, pp. 178-183, 1988.
- [4] S. I. Ahson. "Petri Net Models of Fuzzy Neural Networks," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 25, no. 6, pp. 926-932, 1995.

- [5] K. Gurney. *An Introduction to Neural Networks*, London: Routledge, 1997.
- [6] J.C. Principe, N. R. Euliano, and W. C. Lefebvre. *Neural and Adaptive Systems*. New York: John Wiley & Sons, Inc., 2000.
- [7] T. Dean, J. Allen, and Y. Aloimonos, *Artificial Intelligence: Theory and Practice*. Menlo Park, CA: Addison-Wesley Publishing Company, 1995.
- [8] M. J. L. Orr, *Introduction to Radial Basis Function Networks*. Technical Report, Centre for Cognitive Science, University of Edinburgh, Scotland, 1996.

Andrew R. Seely became a Member of IEEE in 1998. He has a Master of Science in computer science from Nova Southeastern University, Fort Lauderdale, Florida (2002), and a Bachelor of Science in computer and information science from the University of Maryland University College, European Division, Heidelberg, Germany (1999).

He spent seven years serving in the U.S. Air Force, and is currently a lecturer with the University of Maryland University College, European Division. His current research projects involve studying the applicability of Petri Nets to knowledge representation problems.

Mr. Seely is a member of the Association for Computing Machinery (ACM), the American Association for Artificial Intelligence (AAAI), and the Upsilon Pi Epsilon honor society for the computing sciences.