Modeling Self-Adaptive Software Systems With Learning Petri Nets

Zuohua Ding, Member, IEEE, Yuan Zhou, and Mengchu Zhou, Fellow, IEEE

Abstract—Traditional models unable to model adaptive software systems since they deal with fixed requirements only, but cannot handle the behaviors that change at runtime in response to environmental changes. In this paper, an adaptive Petri net (APN) is proposed to model a self-adaptive software system. It is an extension of hybrid Petri nets by embedding a neural network algorithm into them at some special transitions. The proposed net has the following advantages: 1) it can model a runtime environment; 2) the components in the model can collaborate to make adaption decisions while the system is running; and 3) the computation is done at the local component, while the adaption is for the whole system. We illustrate the proposed APN by modeling a manufacturing system.

Index Terms—Adaptive Petri net (APN), adaptive software system, neural network, requirement modeling.

I. INTRODUCTION

A SELF-adaptive software system is one that can autonomously modify its behavior at runtime in response to changes in the system and its environment. It has been used in many fields, such as fault-tolerant computing, distributed systems, biologically inspired computing, and control theory [6].

Models of adaptive systems allow verification and validation during the development process and can support self-adaptation at runtime. However, it is not easy to model them, because: 1) only limited requirements can be used for modeling, and new requirements may occur due to environmental changes; 2) it is not possible to anticipate the behavior in all cases, and the system needs to be adapted dynamically, in a fine-grained way, without disturbing existing behavior; and 3) although certain kinds of anticipated adaptations can be built in by design as runtime configuration parameters, there are many kinds of dynamic adaptation that cannot be anticipated as easily. Hence, the modeling of adaptive systems is much harder than that of nonadaptive systems. A general discussions about the challenges for the systematic software engineering of self-adaptive systems can be found in [6].

Efforts have been made to formally model self-adaptive software systems [5]–[7], [9], [11], [12], [14], [17], [23], [26], [40], [42]. For example, through a finite state automaton with probabilities, the adaption of the system happens among alternative possible paths of the automaton to meet a system’s nonfunctional requirements [15]. Through reevaluating dynamic decision networks, the adaption happens once the conditional probabilities change for different design alternatives over nonfunctional requirements [2]. With a Petri net-based programming model, namely context Petri nets, the behavioral adaptations take place based on the sensed context of execution [4]. With Rainbow, an architecture-based framework, the adaptation is enabled if the operational conditions are unfavorable [13]. A framework named Methodologies and Architectures for Runtime Adaptive Systems is proposed [29], which uses labeled state transition systems directly at a low level of abstraction to model and verify embedded adaptive systems, and adaptation is done by changing the active configuration. And external software mechanisms are used to maintain a form of closed-loop control over the target system [1], [26], thereby allowing a system to self-adapt dynamically, with reduced human oversight.

Salehie and Tahvildari [28] presented a general discussion of different adaptation types. They are classified as structure and behavior-based types. The former focuses on adapting behavior by changing a system’s architecture, while the latter on modifying the functionalities of computational entities. However, they all suffer from the following drawbacks.

1) The adaption mechanisms behave like a control node, switching from one mode to another in a sequential order, thus, lacking collaborative adaption by using their outputs at the same time. For example, mode switching takes place at an architectural level in [3].

2) The computing is not local, a local change at a component requires computing for the whole system, thereby leading to high cost. For example, the overlap adaption [40] is accomplished by a sequence of adaptation transitions.

3) The models are unable to optimize adaption at run time such that the system can find a desired
solution in all situations. In addition, the unknown environment at runtime is not explicitly given in some models [5], [9], [11].

To address the above issues, this paper introduces a new method to create a formal model for the behavior of adaptive systems. The proposed model is an extension to a hybrid Petri net (HPN) by adding special transitions with neural network algorithms. Its adaptation is realized through the learning ability of neural networks. The components can work together to make adaption decisions based on their outputs. The environment is modeled by adding a continuous transition, which generates environmental values at running time. We rigorously show that our model has the adaption ability. A manufacturing process example is used to illustrate how to apply the proposed model.

Our model is better than traditional optimization solutions since it has the ability to not only process runtime data and make decisions, but also model the behaviors of software systems, which can be used for property analysis by model checking tools.

The proposed technique can be used in situations where some decisions can be made only when the runtime environment data is available and cannot be done at the software design stage. The model is scalable since we can design a learning mechanism for each component independently, while collaboration decisions are wisely made.

The rest of this paper is organized as follows. Section II gives a motivating example for self-adaptive systems. Section III describes what needs to be modeled for self-adaptive systems, such as the modeling of learning and corporation abilities. Section IV presents a new Petri net that can be used to model self-adaptive systems. It gives its definition and semantics, and an algorithm to build an evolution graph (EG). Section IV discusses the adaptation ability of the proposed Petri net. Section VI analyzes the motivating example. Section VII is a discussion of our model. Section VIII concludes this paper.

II. MOTIVATING EXAMPLE

A manufacturing system contains customer ordering, raw material purchasing, manufacturing, and customer acceptance as shown in Fig. 1.

There are many branch factories that can finish a contract and many suppliers that can provide required material supplies. Their selection process is in real time to quickly respond to market changes. We assume that there are four branch factories: $B_1$–$B_4$ and two different suppliers, $S_1$ and $S_2$. One appropriate branch factory and one supplier should be selected at a time. The following factors affect the selection of a branch factory: raw material price, transport cost of materials, delivery time, production cycle time, order quantity, production capability, climate, and location, while raw material price, transport cost of materials, delivery time, and location affect the selection of a supplier.

The decision to select a supplier and factory is made based on the above 12 factors. These factors form the environment of the system. Hence, if the market changes, then these factors change, i.e., the environment changes. Hence, we need to find another good combination to complete manufacturing in response to changes. Thus, the system should have the following performance requirements.

R1 Corporation: Select one branch factory and one supplier for the minimum cost.

R2 Learning: The selection process is conducted in real time to quickly respond to market changes.

R3 Adaptation: If the market changes, the system needs to dynamically find a good solution to complete the manufacturing.

III. WHAT TO BE MODELED

From the above requirements, we see that the system has three processes, namely customer, factory, and supplier. For customer, it can be modeled as a regular finite state machine. For the other two, each of which can sense the changes of the environment, learn from the changes to alter its behavior, and finally they cooperate together to complete the manufacturing task with the minimum cost. It is shown in Fig. 2. Thus, our model needs to have the following abilities: 1) it can describe discrete states of the system behavior; 2) it can describe continuous states of environment changes; 3) it can learn; and 4) it can dynamically change its behavior.

Thus, our model should be able to model a hybrid system from 1) and 2). There are many languages that can be
used for this purpose such as transition systems and process algebras. Nevertheless, Petri nets seem a good candidate. On the one hand, they have been used extensively as tools for the modeling, analysis, and synthesis of discrete event systems [20], [21], [34]–[38], [41], [42]. They offer advantages over finite automata, particularly when the issues of model complexity and concurrency of processes are of concern. On the other hand, a continuous system can be approximated by a Petri net [27] and a Petri net model is used as discrete event representation of the continuous variable system [22]. However, Petri nets lack the ability to model learning, adaptation, and corporation.

Next, we present a new Petri net with adaption ability to model a manufacturing system. It contains adaption transitions with learning ability and continuous one for describing environmental changes.

A. Modeling Discrete and Continuous States

Classical Petri nets can model discrete states. In fact, they can be regarded as an extension to finite state machines to model concurrency. To model continuous states, we need to add new places and transitions to the model [10]. We use discrete places to describe discrete states of the system: a discrete place (D-place), represented by a single circle, is a place whose marking or the number of tokens is a nonnegative integer. We use a continuous place to describe environment changes: a continuous place (C-place), represented by a double circle, is a place whose marking is a nonnegative real number. Accordingly, we define discrete and continuous transitions to change the states. A discrete transition (D-transition), represented by a single line, is a transition that moves tokens from its input places to its output places; a continuous transition (C-transition), represented by a hollow rectangle, is a transition that moves real number tokens from its input places to its output places.

B. Modeling Environment

Environment modeling is based on a special C-transition with one input and one output C-place. For example, to model an environmental factor $X$, we can design the weight to an arc from the input C-place to C-transition as $1/\alpha(\tau)$, where

$$\alpha(\tau) = \frac{x - \min}{\max - \min}$$

and $x$ is an observed value of $X$, $\min$ and $\max$ are its minimum and maximum values, respectively, and $\tau$ is a continuous time on some defined time interval. The environment in the model is reflected through the changing weight, which is controlled by a random variable $s(\tau)$ that characterizes the environment $X$. The distribution of $s(\tau)$ is application dependent.

After firing a $C$-transition, we can obtain a normalized value of $X$. As shown in Fig. 3, place $p_1$ has a marking with value 1.0, while $p_2$ represents factor $X$. The marking changes are reflected by the firing quantity of a continuous transition to be defined later.

C. Modeling Learning Ability

Neural networks have learning ability, which have been widely used to model intelligent systems. In this paper, we associate neural networks to our Petri net model. The neural network to be used has one hidden layer, for which the learning law is the back propagation (BP) algorithm [24]. The number of neurons in the hidden layer is determined by Kolmogorov theorem [19]. The simulation of neural activities with Petri net is defined as follows.

1) Places of the Petri net obtained from the neural network are $C$ places since the markings of the Petri net are used to represent the values of signals in the neural network.

2) In order to describe the neural activities, we introduce two different kinds of transitions in the net, i.e., $\alpha$- and $\beta$-transition, where $\alpha$-transition is a special $C$-transition used to model the behavior of the adder in a neuron, while $\beta$-transition associated with a function is used to model the behavior of the active function in a neuron. In the Petri net, $\alpha$ transitions are represented by dashed rectangles and $\beta$ transitions by rectangles with two diagonals in each one.

3) In order to model the behaviors of the neural network accurately by the Petri net, we need to control the firing of these transitions. We thus introduce control places, denoted by dashed circles. The control places are only used to perform control but not participating in the computing.

For example, Fig. 4(a) shows a neuron, where $x_1, \ldots, x_l$ are its inputs, $\Sigma$ is the adder and $\varphi$ is the active function that can be a threshold function, linear function, piecewise-linear function, or log-sigmoid function. In this paper, we choose log-sigmoid function ($\varphi(x) = c \times 1/(1 + e^{-x})$) and hyperbolic tangent sigmoid function ($\varphi(x) = (e^x - e^{-x})/(e^x + e^{-x})$) with appropriate coefficients as active functions. In our example, the coefficient of hyperbolic tangent sigmoid function of the neural network in the factory process is 4, and the other coefficients in the example is 2. Fig. 4(b) is the corresponding Petri net, where $p_{i_1} - p_{i_2}, p_{i_3}, p_{i_4}$ are $C$ places, $p_{i_1}' - p_{i_4}'$ are control places, $r_{i}^\alpha - r_{i}^\beta$ are $\alpha$ transitions, $r_{i}^\beta$ is a $\beta$-transition, and $w_1, \ldots,$ and $w_l$ are the synaptic weights which are adjustable during the training process. The initial markings of $p_{i_1}, p_{i_2}, \ldots,$ and $p_{i_l}$ are $x_1, x_2, \ldots,$ and $x_l$, respectively. The firings of transitions
Fig. 4. Modeling of neural activities with a Petri net. (a) Simple neuron. (b) Corresponding Petri net.

\[ t^\alpha_{1} - t^\alpha_{l} \text{ and } t^\beta_{1} \text{ map the markings of } p_{i1} - p_{il} \text{ to the marking of } p_{o}. \] Note that, sometimes we present this mapping by a function symbol \( f(\cdot) \), i.e., \( M(p_{o}) = f(M(p_{i1}), \ldots, M(p_{il})) \).

Based on the Petri net model of a single neuron, we can easily obtain a Petri net for any neural network. For example, Fig. 5(a) is a neural network with two inputs, five hidden neurons, and one output. Thus, the corresponding Petri net is shown in Fig. 5(b).

In order to simplify the Petri net, we may use a transition, called an \( A \)-transition, to represent the net that is used to model a neural network. It is a transition representing a subnet which behaviors like a neural network. For example, the Petri net is shown in Fig. 5(b) can be simplified by an \( A \)-transition as shown in Fig. 5(c). It has two input places and an output place, representing a neural network. The inputs of this neural network are the markings of \( p_{1} \) and \( p_{2} \) and the output is the marking of \( p_{3} \). There are five neurons in the hidden layer.

Note that, when we transform a neural network to a special Petri net, we do have negative markings and weights in the net. However, the markings of the inputs and outputs of this net are nonnegative. So, if we use an \( A \)-transition to represent this special net, then we do not have any negative markings. Since the semantics of this special net does not affect later discussion, we omit it.

D. Modeling Collaboration

For two \( A \) transitions, if their outputs can eventually reach the same place, they are called to be independent. This structure can be used to model two components that complete adaption separately and independently. If their outputs eventually reach a same transition, they are called to be dependent. This structure can be used to model that two components are combined together to complete adaption. Fig. 6 shows an example of two independent \( A \) transitions, and Fig. 7 shows an example of two dependent ones.

Note that, the concepts of independent and dependent \( A \) transitions can be extended to multiple ones. The algorithm to train two independent \( A \) transitions is direct: just apply the BP algorithm since they are independent. However, the algorithm for two dependent \( A \) transitions needs extra work.

First, we need to consider how to obtain the sample data to build a neural network. These data can be classified into two sets: the set for training the neural network and the other to test it. For a new system, such data can be historical data generated by similar systems in a similar environment, or generated according to domain experts. According to the input, we need to compute the total cost of each factory and each supplier and then select the combination which has the lowest cost as the output. Since the outputs of neural networks are real numbers, the indices of each factory and supplier need to be continuous. Thus, we use an interval to represent a factory or a supplier, and the sample data of each factory and supplier is the midpoint of the corresponding interval.
Second, we need to train the neural networks. We use the BP algorithm, with the objective to minimize the total error of all neural networks. The detail is shown next.

We define \( x^* \) as a set of inputs (input places, transitions) of \( x \) and \( x^* \) as a set of outputs. For example, given \( r^4 \) in Fig. 6(c), \( r^4 = \{ p_1, p_2 \} \) and \( r^4* = \{ p_3 \} \), which is often written as \( r^4* = p_3 \).

Let \( t_1 \) and \( t_2 \) be two \( A \) transitions. Assume that \( |r_1| = q_1 \), \( |r_2| = q_2 \), and \( |r_1*| = s_1 \), \( |r_2*| = s_2 \). Then their associated neural network \( k \), \( k = 1, 2 \), has \( q_k \) inputs and \( s_k \) outputs. Suppose that \( t_1 \) has \( l_1 \) neurons in the hidden layer and \( t_2 \) has \( l_2 \) ones. The sample datasets used to train the networks are \( S_1 \) and \( S_2 \). For neural network \( k \), \( w_{hk}^i \) is the weight from input neuron \( i \) to the hidden neuron \( h \) and \( W_{ok}^i \) is the weight from the hidden neuron \( h \) to output neuron \( o \), where \( i = 1, 2, \ldots, q_k \), \( h = 1, 2, \ldots, l_k \), and \( o = 1, 2, \ldots, s_k \). The activation functions of hidden and output neurons are \( \varphi_1^i(\cdot) \) and \( \varphi_2^i(\cdot) \), respectively; \( x\{ \} = (x_1, \ldots, x_q) \) is a sample input datum, \( y\{ \} = (y_1, \ldots, y_q) \) is the corresponding sample output datum, and \( \tilde{y}\{ \} = (\tilde{y}_1, \ldots, \tilde{y}_q) \) is the corresponding output of the neural network. Besides, \( h_k = 1, \ldots, l_k \) and \( o_k \) denotes the indices of the input neurons; \( h_k = 1, \ldots, l_k \) and \( o_k \) are the indices of hidden neurons of neural network \( k \) and \( o_k = 1, \ldots, s_k \) are the indices of the output neurons of neural network \( k \).

The training algorithm has the following steps.

Step 1: Set accuracy \( \varepsilon \), learning rate \( \eta \), and the maximum number of iterations \( N \).

Step 2: Initialize the weights of two networks. Assign each weight a random number. There are two ways to assign the weights: one is to assign a random number between \(-1, 1\), and the other is to use Nguyen–Widrow algorithm [25], which chooses the values in order to distribute the active region of each neuron in the layer randomly but evenly across the layer’s input space. In our case, we adopt the second way.

Step 3: Select a sample datum \((x^{[1]}, y^{[1]})\) from \( S_1 \) and \((x^{[2]}, y^{[2]})\) from \( S_2 \) randomly.

Step 4: Compute the output of the hidden layer and output layer of each network, and the local error

\[
\begin{align*}
\tilde{z}\{ \} &= \{ \sum_{i=1}^{q_k} w_{hk}^i x_i^k \} \cdot z_{h_k}^k = \varphi_1^k(\tilde{z}_{h_k}^k) \\
y\{ \} &= \{ \sum_{h=1}^{l_k} W_{ok}^h \tilde{y}_h \} \cdot \tilde{y}_{o_k}^k = \varphi_2^k(y_{o_k}^k) \\
e_k &= \sum_{o=1}^{s_k} (\tilde{y}_o^k - y_o^k)^2, \quad e = 1/2(e_1 + e_2). \quad (3)
\end{align*}
\]

Step 5: Compute the gradient of \( e \) in the direction of \( W_{ok}^h \)

\[
\begin{align*}
\frac{\partial e}{\partial W_{ok}^h} &= \frac{\partial e}{\partial y_o^k} \cdot \frac{\partial y_o^k}{\partial \tilde{y}_o^k} \cdot \frac{\partial \tilde{y}_o^k}{\partial W_{ok}^h} \\
&= -\sum_{o=1}^{s_k} (\tilde{y}_o^k - y_o^k) \cdot \varphi_2'(\tilde{y}_o^k) \cdot \tilde{y}_o^k \\
&\quad \cdot \frac{\partial W_{ok}^h}{\partial W_{ok}^h} \\
&= \sum_{o=1}^{s_k} (\varphi_1'(\tilde{z}_{h_k}^k) \cdot \tilde{z}_{h_k}^k) \cdot \delta_{h_k}^k \cdot \varphi_2'(\tilde{y}_o^k) \cdot \tilde{y}_o^k \\
&\quad \cdot \frac{\partial W_{ok}^h}{\partial W_{ok}^h} \\
&\triangleq -\delta_{h_k}^k \cdot \tilde{x}_o^k. \quad (4)
\end{align*}
\]

Step 6: Compute the gradient of \( e \) in the direction of \( w_{hk}^i \)

\[
\begin{align*}
\frac{\partial e}{\partial w_{hk}^i} &= \sum_{o=1}^{s_k} \left( \frac{\partial e}{\partial \tilde{y}_o^k} \cdot \frac{\partial \tilde{y}_o^k}{\partial \tilde{y}_o^k} \cdot \frac{\partial \tilde{y}_o^k}{\partial z_{h_k}^k} \cdot \frac{\partial z_{h_k}^k}{\partial w_{hk}^i} \right) \\
&= \sum_{o=1}^{s_k} \left( \varphi_1'(\tilde{z}_{h_k}^k) \cdot \tilde{z}_{h_k}^k \cdot \tilde{x}_o^k \right) \\
&\triangleq -\delta_{h_k}^k \cdot \tilde{x}_o^k. \quad (5)
\end{align*}
\]

Step 7: Update the weights according to the gradients of \( e \) and learning rate \( \eta \)

\[
\begin{align*}
\Delta w_{ok}^h &= -\eta \frac{\partial e}{\partial w_{ok}^h} = \eta \delta_{h_k}^k \cdot \tilde{x}_o^k \\
\left( w_{ok}^h \right)^{n+1} &= \left( w_{ok}^h \right)^n + \Delta w_{ok}^h \\
\Delta W_{ok}^h &= -\eta \frac{\partial e}{\partial W_{ok}^h} = \eta \delta_{h_k}^k \cdot \tilde{y}_o^k \\
\left( W_{ok}^h \right)^{n+1} &= \left( W_{ok}^h \right)^n + \Delta W_{ok}^h. \quad (6)
\end{align*}
\]
Step 8: Compute the outputs of the neural network according to the inputs of sample data and then compute error $E$:

$$
E = \frac{1}{2|S_1| + 2|S_2|} \left\{ \sum_{(x^{(1)}, y^{(1)}) \in S_1} \| y^{(1)} - \tilde{y}^{(1)} \|^2 + \sum_{(x^{(2)}, y^{(2)}) \in S_2} \| y^{(2)} - \tilde{y}^{(2)} \|^2 \right\}
= \frac{1}{2|S_1| + 2|S_2|} \left\{ \sum_{(x^{(1)}, y^{(1)}) \in S_1} \sum_{o=1}^{s_1} (\tilde{y}_o^{(1)} - \tilde{y}_o^{(1)})^2 + \sum_{(x^{(2)}, y^{(2)}) \in S_2} \sum_{o=1}^{s_2} (\tilde{y}_o^{(2)} - \tilde{y}_o^{(2)})^2 \right\}.
$$

(8)

Step 9: $n = n + 1$. If $n \leq N$ or $E < \varepsilon$, terminate; else reselect a sample datum and go back to Step 4.

For more than two neural networks, we can train them similarly. The difference is that we need to modify (3) and (8) since the numbers of neural networks are different. Suppose there are $Z$ neural networks, then (3) and (8) are rewritten as:

$$
e = \frac{1}{Z} \sum_{i=1}^{Z} e_i
$$

(9)

$$
E = \frac{1}{2Z^2} \sum_{j=1}^{Z} \sum_{i=1}^{Z} \sum_{o=1}^{s} (\tilde{y}_o^{(j)} - \tilde{y}_o^{(i)})^2.
$$

(10)

IV. ADAPTIVE PETRI NETS

Based on the discussion in the above section, we define a new type of Petri nets to model self-adaptive systems, namely adaptive Petri nets (APNs), which have learning and adaptation ability, and can model environmental changes.

A. Definition of Adaptive Petri Net

Definition 1: A Learning Net is a tuple $PN^L = (P = P_c \cup \{p^*_t, p^*_e\}, T = \{t_d, t_a\} \cup T_c, F, w, \gamma)$, where $P_c$ set of continuous places and is classified into two parts: $P^d_c$ and $P^o_c$ with $|P^d_c| = |P^o_c|$. Moreover, $P^d_c = t_d^*$ and $P^o_c = t_a^*$.

- $p^*_s = \emptyset$ and $p^*_t = t_d$; $p^*_e = t_a$ and $p^*_e = \emptyset$;
- $t_d$ discrete transition and $t_a$ is an adaptive transition; $T_c$ is a set of continuous transitions, and $T_e = P^d_c$ and $T_e = P^o_c$;
- $F$ set of arcs such that $F \subseteq (P \times T) \cup (T \times P)$;
- $w$ set of weights assigned to each arc. $w(p, t) \in R^+$ and $w(t, p) \in R^+$, where $R^+$ is a set of nonnegative real numbers. Other weights are equal to 1.

Definition 2: A Branch Net is a tuple $PN^B = (P, T, F)$, where $P$ finite set of places, for each $p \in P$, $|p^*| \geq 1$, and $|p^*_t| \geq 1$;

$T$ finite set of transitions, for each $t \in T$, $|t^*_w| = 1$, and $|t^*_e| = 1$;

$F$ set of arcs such that $F \subseteq (P \times T) \cup (T \times P)$.

Definition 3: A Learning Branch Net is a combination of a Branch Net $PN^B = (P, T, F)$ and a Learning Net $PN^L = (P_c \cup \{p^*_t, p^*_e\}, \{t_d, t_a\} \cup T_c, F, w, \gamma)$, where there are two transitions $t_1, t_2 \in T$ such that $t_1 = t_1$ and $t_2 = t_2$.

In the combination of a branch net and a learning net, each arc from $p^*_t$ to a transition $t$ is assigned a condition described by an inequality. Only the marking of $p^*_t$ satisfies this condition can the transition $t_1$ be fired.

Definition 4: A Closed Process Net is a tuple $PN^C = (P \cup \{p_s, p_e\}, T \cup \{t_e\}, F)$, where $P$ finite set of Internal places, for each $p \in P$, $|p^*| \geq 1$, and $|p^*_t| \geq 1$;

$T$ finite set of Internal transitions, for each $t \in T$, $|t^*_w| = 1$, and $|t^*_e| = 1$;

- $p_s$ start place, $|p^*_s| = 1$, and $|p^*_e| \geq 1$;
- $p_e$ end place, $|p^*_e| \geq 1$, and $|p^*_e^*| = 1$;

- $t_e = t_e$ and $t_e = p_s$;

$F$ set of arcs such that $F \subseteq (P \times T) \cup (T \times P)$.

Definition 5: A Closed Learning Process Net $PN^{C,L}$ is a combination of a Closed Process Net $PN^C = (P \cup \{p_s, p_e\}, T \cup \{t_e\}, F)$ and a Learning Net $PN^L = (P_c \cup \{p^*_t, p^*_e\}, \{t_d, t_a\} \cup T_c, F, w, \gamma)$, where there are two transitions $t_1, t_2 \in T$ such that $t_1 = t_1$ and $t_2 = t_2$.

It is easy to see that start place $p_s$ of a closed (learning) process net leads one or many (learning) branch nets that end at $p_e$ exemplified in Fig. 8.

The communications among processes are through Rendezvous communications, which rely on mutual exclusion places $p_{\text{ack}}$ to the net as introduced in [30]. Such place guarantees that only one closed process net may send a token at any time to a receiving net. No other nets will send a token to that receiver until the current sender is acknowledged by $p_{\text{ack}}$. Places $p_{\text{send}}$ and $p_{\text{ack}}$ are called buffer places, and the connected transitions are called communication transitions, as shown in Fig. 9.

Fig. 8. (a) Closed (learning) process net. (b) Example of a closed (learning) process net with two branch nets.
A Rendezvous Communication Mechanism is a tuple $CM^Q = \langle t_1, t_2, t_3, t_4, p_{12}, p_{34}, p_{send}, p_{ack}, p_{mcp} \rangle$, where

- $p_{12}, p_{34}, p_{send},$ and $p_{ack}$ are places such that $|p_{12}| = |p_{12}^*| = 1$, $|p_{34}| = |p_{34}^*| = 1$, $|p_{send}| = |p_{send}^*| = 1$, and $|p_{ack}| = |p_{ack}^*| = 1$; and $p_{mcp}$ is a place such that $|p_{mcp}| = |p_{mcp}^*|$
- $t_1$ transition such that $|t_1| = |t_1^*| = 2$.
- $t_2$ transition such that $|t_2| = |t_2^*| = 2$.
- $t_3$ transition such that $|t_3| = |t_3^*| = 1$.
- $t_4$ transition such that $|t_4| = |t_4^*| = 2$.

$p_{12}$ and $p_{34}$ are called intermediate states, $p_{send}$ and $p_{ack}$ are called Buffer places, and $p_{mcp}$ is called a Control place, $t_1$ and $t_4$ are called output communication transition, and $t_2$ and $t_3$ are called input communication transition.

Definition 7: Let $PN^C_1$ and $PN^C_2$ be two closed process nets. If there exists a Rendezvous communication mechanism $CM^Q = \langle t_1, t_2, t_3, t_4, p_{12}, p_{34}, p_{send}, p_{ack}, p_{mcp} \rangle$ such that $PN^C_1$ and $CM^Q$ have common parts $\{t_1, t_2, p_{12}\}$, and $PN^C_2$ and $CM^Q$ have common parts $\{t_3, t_4, p_{34}\}$, then $PN^C_1$ is Communicating with $PN^C_2$ through communication mechanism $CM^Q$, or simply Communicating.

This definition describes that two closed process nets communicate with each other through a Rendezvous communication mechanism. It is also true if $PN^C_1$ and $PN^C_2$ are replaced by two closed learning process nets $PN^C_{1,\text{LC}}$ and $PN^C_{2,\text{LC}}$. If more closed process nets are communicating to the same other closed process net, then Rendezvous communication mechanism guarantees that only one closed process net may send a token at any time to the receiving one. The receiver cannot accept any tokens from other closed process nets until the current sender is acknowledged.

Definition 8: Let $A-C$ be three closed (learning) process nets. $A$ and $B$ are communicating with $C$ at the same time, if their Rendezvous communication mechanisms have the same $\{p_{\text{send}}, p_{\text{ack}}, p_{\text{mcp}}\}$, i.e., only $A$ or $B$ can send a message to $C$, but not both at the same time.

For example, Fig. 10 shows that two closed process nets $A$ and $B$ mutually send messages to the same closed process net $C$.

Our design can be extended to the case that one process sends requests to more than one process capable of servicing the requests and waits for at least one response as shown in Fig. 11. No additional details about it will be given due to space limitation.

Definition 9: An APN is a subclass of Petri net $\langle P, T, F \rangle$, consisting of a set of closed process nets $\{PN^C_i, o = 1, 2, \ldots, g\}$, closed learning process nets $\{PN^C_{i,\text{LC}}, i = 1, 2, \ldots, k\}$, and a set of Rendezvous communication mechanisms $\{CM^Q_j, j = 1, 2, \ldots, l\}$ is as follows.

1) Each closed (learning) process net $PN^C_i$ ($PN^C_{i,\text{LC}}$) is communicating through at least one communication mechanism.

2) Each communication mechanism $CM^Q_j$ has been used for one and only one pair of process nets.

An APN net is denoted by $PN^N$. In an APN net, places are classified as idle denoted as $P^S = \{p_s\}$, buffer denoted as $P^B = \{p_{\text{send}}, p_{\text{ack}}\}$, control denoted as $P^C = \{p_{\text{mcp}}\}$, and activity for the rest denoted as $P^A$. While transitions are classified as activity denoted as $T^A = \{t|t^* = 1, |t| = 1\}$, input communication denoted as $T^I = \{t|\exists p \in P^B, \text{s.t.} p \in t\}$, and output communication denoted as $T^O = \{t|\exists p \in P^B, \text{s.t.} p \in t^*\}$. Thus, in an
Fig. 12. \( P^N_\text{APN}^\emptyset \) model for the manufacturing system.

APN net, we can say that a closed process net has input and output communication transitions.

**Definition 10:** A marked APN net is a tuple \( \langle P^N_\text{APN}, M_0 \rangle \), where

\[
P^N_\text{APN} = (P, T, F) \text{ APN net};
\]

\[
M_0 : P \rightarrow \{0, 1\} \text{ initial marking: } \forall p \in P^S, M_0(p) = 1; \forall p \in P^A, M_0(p) = 0; \forall p \in P^B, M_0(p) = 0; \text{ and } \forall p \in P^C, M_0(p) = 1.
\]

From the construction of APN, we see that the proposed APN can be used to model a system consisting of a set of processes that communicate through Rendezvous communication mechanisms.

**Remark 1:** Note that, colored Petri nets assign data values to the tokens, and expressions are attached to the arcs. These define the constraints on the token values in the input places to enable the transitions, and define the token values, produced by the firings, in the output places. HPNs allow the coexistence of both continuous and discrete processes. They include discrete places (marked with tokens) and continuous places associated with real variables. Discrete transitions fire after a determined delay, while enabled continuous transitions fire continuously at a given rate. Hence, our APN is an extension to HPN.

A \( P^N_\text{APN}^\emptyset \) model of the manufacturing system is shown in Fig. 12. In this model, \( P^S = \{p_1, p_6, p_22\} \), \( P^A = \{p_4, p_5, p_19, p_20, p_21\} \), and \( P^C = \emptyset \). Thus, the initial marking is \( M_0(p_1) = M_0(p_6) = M_0(p_22) = 1 \) while others are zero. \( r_1^A \) and \( r_2^B \) are two \( A \) transitions. Each has a three-layer BP neural network, denoted by \( NN_1 \) and \( NN_2 \), respectively.

For convenience, sometimes we may also classify the places as discrete (denoted as \( P_D \)) and continuous (denoted as \( P_C \)). While the transitions as discrete (\( T_D \)), continuous (\( T_C \)), and adaptive (\( T_A \)). According to these definitions,
we find that $P_C, T_C, T_A$ can appear in the structure of learning Petri nets only.

B. Semantics of $PN^N$

A transition is fired if it is enabled. There are three enabling conditions for it are as follows.

1) A $D$-transition $t$ is enabled by marking $M_t$ if $\forall p \in \mathbf{\cdot} \cap (PD \cup PC), M_t(p) \geq W(p, t), \text{ and } \forall p \in \mathbf{\cdot} \cap P_{Ctl}, M_t(p) = 1$.

2) A $C$-transition or $\alpha$-transition $t$ is enabled by marking $M_t$ if $\forall p \in \mathbf{\cdot} \cap (PD \cup PC), M_t(p) > 0, \text{ and } \forall p \in \mathbf{\cdot} \cap P_{Ctl}, M_t(p) = 1$.

3) A $A$-transition $t$ is enabled by marking $M_t$ if $\forall p \in \mathbf{\cdot} \cap P_{Ctl}, M_t(p) = 1$.

4) An $A$-transition $t$ is enabled by marking $M_t$ if $\forall p \in \mathbf{\cdot}, M_t(p) > 0$, where $P_{Ctl}$ are control places in the net representing a neural network.

In a marked $PN^N$, the enabling degree of transition $t$ at marking $M$, denoted by $q(t, M)$ or $q$, measures the maximal amount the transition can be fired in one time enabling, defined as follows.

1) If $t \in T_D$, its enabling degree is the greatest integral number such that

$$q \leq \min_{p_1: p_2 \in \mathbf{\cdot} \cap (P_D \cup P_C)} \frac{M(p_1)}{W(p_1, t)} < q + 1.$$

2) If $t \in T_C$, the enabling degree is a real number such that

$$q = \min_{p_1: p_2 \in \mathbf{\cdot} \cap (P_D \cup P_C)} \frac{M(p_1)}{W(p_1, t)}.$$

If a $C$-transition $t$ is enabled by marking $M_t$, its firing quantity, written as $\delta(t, M)$, is a real number between 0 and $q(t, M)$, describing the number of firing times. In other words, a $C$-transition fires $\delta$ times in a row, and we denote it as $[\delta]^\delta$. As mentioned above, $T_D$ is a kind of special continuous transitions because the firing quantities of the transitions in this set are always equal to their enabling degree, i.e., $\forall t \in T_D, \delta(t, M) = q(t, M)$.

Remark 2: It worth to mention that a transition has the similar firing ability in [39]. Analogy with queuing systems, a transition is designed that has an unlimited number of channels like a many-channel server, where place markings specify the actual restrictions on the number of channels.

Based on the firing quantity, the semantics of $PN^N$, i.e., the rules to compute the marking at each place, are defined based on the following basic net structures. According to the definition of $PN^N$, we have the following structures.

1) Input and Output Places Are D Places: This structure is the same as that in classical discrete Petri nets. For example, in Fig. 13, $p_1$ and $p_2$ are two discrete places, $t$ is a $D$-transition. $M(p_1) = 1$ and $W(p_1, t) = 1$. After $t$ fires, $M(p_1) = 0$ and $M(p_2) = 1$.

2) Input Places Are C Places and Output Places Are D Places: This structure converts continuous marking into discrete marking. If the $D$-transition fires, a real number marking, which is the product of the enabling degree and the weight of the arc from the place to transition, is removed from the input place.

Meanwhile, an integer number marking, which is the product of the enabling degree and the weight of the arc from the transition to the place, is added to the output place.

For example, in Fig. 14, $p_1$ is a $D$-place and $p_2$ is a $C$-place. $t$ is a $D$-transition. Assume $M(p_2) = 2.5, W(p_2, t) = 2$, and $M(p_2) > W(p_2, t), t$ can be enabled. The firing of $t$ consists of removing a marking quantity 2 from place $p_2$ and adding a token (integer) to $p_1$. This procedure can be represented as $(0, 2.5) \rightarrow (1, 0.5)$. Thus, $t$ converts a continuous marking (the one in $p_2$) into a discrete one (the one in $p_1$).

3) Input Places Are D Places and Output Places Are C Places: D- and C-place are connected by a $D$-transition. When the $D$-transition fires, an integer number of tokens are moved from its input places and a real number of marking is added to its output places. The number of moved tokens is the product of its enabling degree and weight of its input arc, and the number of added marking is the product of its enabling degree and weight of its output arc.

For example, in Fig. 15, $p_1$ is a $D$-place and $p_2$ is a $C$-place. $t$ is a $D$-transition. Assume that $p_1$ has a token and the weight $W(p_1, t) = 1$, then $t$ is enabled. When $t$ fires, the token in $p_1$ is removed and a making quantity 1.5 is added to $p_2$ since $W(t, p_2) = 1.5$ and $W(p_2, t) = 0$.

4) Input Places and Output Places Are All C Places Connected by C-Transition: In this case, a marking with a value equal to the product of firing quantity and the weight of the input arc is removed from each input places. Meanwhile, a marking with a value equal to the product of firing quantity and
the weight of the output arc is added to each output place. For example, in Fig. 16, p1, p2, and p3 are C places, and t is a C-transition. The initial marking is 1.5, 1.2, and 0. According to the enabling condition, t is enabled with enabling degree 1.2. The firing quantity of t is assumed to be 0.8, thus, the markings becomes 0.7, 0.4, and 0.96 after t fires.

5) Input Places and Output Places Are All C Places Connected by α- and β-transition: This structure in our net is used to model the activities of a neural network. It models input–output mappings. The weights can be adjusted during the training process of a neural network, which adds the adaptation ability to the net.

The above semantics can be summarized as follows. Suppose that t is enabled and fires at marking $M_i$, then we have the following.

a) If $h(t) = D$, $q$ is the enabling degree, then

$$M_{i+1}(p) = \begin{cases} M_i(p) + qW(t, p), & p \in t^* \cap (P_D \cup P_C) \\ M_i(p) - qW(p, t), & p \in t^* \cap P_{Ctl} \\ 0, & p \in t^* \cap (P_D \cup P_C) \\ M_i(p), & \text{otherwise.} \end{cases}$$

b) If $h(t) = C$ or $\alpha$ with firing quantity $\delta(t, M_i)$, then

$$M_{i+1}(p) = \begin{cases} M_i(p) + \delta(t, M_i)W(t, p), & \text{ otherwise.} \end{cases}$$

$$M_{i+1}(p) = \begin{cases} M_i(p), & \text{otherwise.} \end{cases}$$

where $\varphi(\cdot)$ is a given active function.

c) If $h(t) = \beta$ with associated function $\varphi(\cdot)$, then

$$M_{i+1}(p) = \begin{cases} 0, & p \in t^* \\ \varphi(M_i(p)), & p \in t^* \cap P_C \\ 1, & p \in t^* \cap P_{Ctl} \\ M_i(p), & \text{otherwise.} \end{cases}$$

C. Building Evolution Graph

As building a reachability graph (RG) for a traditional Petri net [41], we can also build a behavior EG for an APN. The following is the algorithm.

The algorithm to compute the EG of an APN

Input: $apn = (P, T, A, h, \varphi, M_0)$;
Output: reachable states $RS$, reachability graph $RG$;
Initial: $RS = RG = [M_0]$, $new_states = [M_0]$, $i = 0$;
1. begin
2. if(new_states != NULL) then
3. $T_{enabled} = \text{the transitions enabled by new_states}$;
4. $old_states$-new_states, $new_states$=new_states;
5. while($T_{enabled} \neq NULL$) do
6. select a transition $t$ from $T_{enabled}$;
7. $M'$ = the marking after firing $t$;
8. if($M'$ is not belong to old_states) then
9. $RS := RS+[M']$ and $RG[RG[M_i-(t)->M']]$;
10. $new_states$=new_states+$M_i-(t)->M'$;
11. end
12. $T_{enabled} := T_{enabled}-\{t\}$;
13. end
14. end
15. end

In the algorithm, $new_states$ collects all the reachable states that have no successors. The main idea to compute the reachable states is to identify all the enabled transitions enabled by $new_states$, then fire them to generate new states and update the set $new_states$. Then repeat this process until there is no new state generated. Since there exist C- and A-transition in an APN, the number of reachable states is infinite.

However, we have the following observation. If we overlap the nodes by firing the same sequence of transitions regardless of the markings at the nodes, we obtain a new graph, namely EG, which has the same structure as that of RG of a normal Petri net. To be more precisely, we have the following.

Proposition 1: For an APN, assume that we replace C- and A-transition by D transitions without changing the priority (the right to take precedence in the firing of transition) to obtain a normal Petri net. Assume that the weights of the arcs connected to D transitions are set to 1. Then the EG of the APN has the same structure as the RGS of the normal Petri net, and the difference is that in EG a node may have an infinite number of markings while in RG a node has only one marking.

The following is a sketch of the proof.
First, consider the classical algorithm for producing RG [41]. It has the following steps.

Step 1: Identify all the enabled transitions.
Step 2: Fire all of them resulting in “new” markings.
Step 3: For each marking, if it is an old or dead marking, go to the next marking. Otherwise, identify all the enabled transitions and fire them to generate the new markings, and then go to the next marking until the markings at the same level are exhausted. Then start the next level of markings.

Compare our algorithm with this algorithm, we find that the process of producing RG and EG are similar except that in the algorithm for EG there are no stops since the number of reachable states is infinity. But once we overlap the nodes, both of them are determined by the firing sequences of transitions only. Since, we only change the C- and A-transition rather than the structure of the Petri net, two nets have the same
For example, the environment space of the above manufacturing system has multi-environments. \( E_1 = \{ e \mid 2 < f_1(e) \leq 3, 0 < f_2(e) \leq 1 \} \) and \( E_2 = \{ e \mid 2 < f_1(e) \leq 3, 1 < f_2(e) \leq 2 \} \) are two of them, where \( f_1 \) and \( f_2 \) are the functions representing the two parts of the net that model two neural networks, respectively. These inequalities are derived from transitions \( t_1 \) and \( t_2 \) in Fig. 12. The inputs of them construct the environment spaces while the outputs are used to partition the environment spaces.

**Proof:**

**Case 1:** The Net Contains Independent Neural Networks Only: Without loss of generality, we assume that the net contains just one neural network, and the embedded function is \( f \). If the network has just one output, then \( E \) has one environment. In fact, if it has more than one, say two: \( E_1 \) and \( E_2 \), then \( f(E_1) \not= \emptyset \), \( f(E_2) \not= \emptyset \), and \( f(E_1) \cap f(E_2) = \emptyset \). Thus, the output of the network has two branches, which is a contradiction.

If the network has more than one outputs, say two, \( c_1 \) and \( c_2 \) with \( c_1 \cap c_2 = \emptyset \). Then there must exist two environments \( E_1 \) and \( E_2 \), such that \( f(E_1 \cup E_2) = c_1 \cup c_2 \), and \( f(E_1) = c_1 \) and \( f(E_2) = c_2 \). In fact, \( E_1 \) and \( E_2 \) must be nonempty. We only need to prove that \( E_1 \cap E_2 = \emptyset \).

If not, there exists a point \( e \in E_1 \cap E_2 \), then \( f(e) \in f(E_1) \cap f(E_2) = c_1 \cap c_2 = \emptyset \), which is a contradiction [from (H), at least one choice can be reached].

**Case 2:** The Net Contains Dependent Neural Networks: Assume that there are two neural networks \( f_1 \) and \( f_2 \) that are dependent. If their outputs do not contain a choice, then \( E \) has just one environment, which can be proved as the first case. If their outputs have a choice, say each has two described by two conditions \( c_{1i} \) and \( c_{2i} \), \( c_{1i} \cap c_{2i} = \emptyset \), \( i = 1, 2 \). Thus, there at most exist four choices based on two neural networks: \( c_{11} \times c_{21}, c_{11} \times c_{22}, c_{12} \times c_{21}, \) and \( c_{12} \times c_{22} \). Note that, the symbol \( \times \) means a Cartesian product. Accordingly, there are four environments \( E_1 \cup E_4 \) such that \( f_1(f_2)(E_1) = c_{11} \times c_{21}, (f_1,f_2)(E_2) = c_{11} \times c_{22}, (f_1,f_2)(E_3) = c_{12} \times c_{21}, \) and \( (f_1,f_2)(E_4) = c_{12} \times c_{22} \). Obviously, each \( E_i \) is nonempty. We need to prove that every two \( E_i \)’s have empty intersection. In fact, for example, if \( E_1 \cap E_2 \neq \emptyset \), then there is an element \( e \in E_1 \cap E_2 \). It follows that \( (f_1,f_2)(e) \in (f_1,f_2)(E_1) \) and \( (f_1,f_2)(e) \in (f_1,f_2)(E_2) \), i.e., \( (f_1,f_2)(e) \in (c_{11} \times c_{21}) \cap (c_{11} \times c_{22}) = \emptyset \). Hence, \( E = \cup E_i \) and \( E_1 \cap E_2 = \emptyset \).

**Definition 13 (Behavior):** Let \( p \) be a place of \( PN^N \) and \( m \) is a marking of place \( p \), \( (p,m) \) represents a local state of \( PN^N \). An execution of \( PN^N \) is a sequence of firings of transitions such that local states \( \{ (p_j,m_j), j = 1, 2, \ldots, l \} \) change with an order. If in the ordered local states, some or all \( p_j \) can be repeatedly visited, then the ordered local states is called a behavior of \( PN^N \).
Lemma 2: A behavior is a subgraph of the EG of $PN^N$ containing one or more cycles.

Proof: From the definition of behavior, for the local states in the subset of $PN^N$ that can be reached, the markings of the places in the subset are reachable markings. Hence, these markings are in the nodes in the EG. All such nodes form a subgraph. Since the local states in the subset can be repeatedly reached, then the markings of those places will repeatedly visit this subgraph, in other words, the subgraph has a cycle. If the subnet contains conflicting conditions, then there are more branches in EG. Again since the local states can be repeatedly reached, we imply that at least one condition will be repeatedly executed. If one is repeatedly executed, then the subgraph repeatedly reached, then the markings of those places will repeatedly execute. If more conditions are repeatedly executed, then this subgraph contains more cycles.

A subgraph of an EG that covers a behavior is named as $b$-graph. If a $b$-graph has at most one cycle, then we say the behavior is a simple behavior, otherwise, we call it a complex behavior. For example, in Fig. 17, there are eight $b$-graphs which are shown in Fig. 18 representing eight complex behaviors. Among them Fig. 18(e) and (f) corresponds to complex behaviors. Among them Fig. 18(e) and (f) corresponds to complex behaviors.

Lemma 3: Let $PN^N$ be an APN and $G$ is its EG. Then, we have: 1) for an environment $E_i$, it has one and only one corresponding subgraph $G_i \subseteq G$; 2) for two environments $E_i$ and $E_j$ and their corresponding subgraphs $G_i$ and $G_j$, if $E_i \neq E_j$, then $G_i \neq G_j$; and 3) $G = \bigcup G_i$, $G_i \cap G_j = \emptyset$, $G_i$ are subgraphs.

Proof:
1) Let $E_i$ be an environment. For any $e_i \in E_i$, it must be part of some reachable marking, thus, there is a node $q$ in $E_i$ that corresponds to $e_i$. From Lemma 2, $q$ is in some subgraph, say $G_i$. We say that there is only one subgraph for it. If not, there is another subgraph, say $G_j$ that contains $q$. So, both subgraphs have an overlap. Since $G_i$ and $G_j$ are different, there exists a choice from some node, say $\omega$ in both subgraphs. If the marking in $\omega$ contains the output of an $A$-transition, then there exists an environment $E_j$ different from $E_i$, such that $e_i \in E_j$. This is a contradiction to the hypothesis that $e_i \in E_i$. It follows that all the nodes in $G_i$ would be in $G_j$, i.e., $G_i \subseteq G_j$. This is a contradiction to the definition of $G_i$. We can also imply that if $E_i \cap E_j = \emptyset$, then $G_i \cap G_j = \emptyset$.

2) Let $E_i$ and $E_j$ be two different environments. From Lemma 2, let $G_i$ and $G_j$ be their corresponding subgraphs in the EG. If $G_i = G_j$, this means that under environments $E_i$ and $E_j$, the net has the same reachable markings. Thus $E_i = E_j$, leading to a contradiction.

3) Let $E$ be the environment. From Lemma 1, $E = \bigcup E_i$, where $E_i \cap E_j = \emptyset$. From 1) and 2), $G_i$ is the corresponding subgraph of $E_i$. Thus, $\bigcup G_i \subseteq G$. We say that all these $G_i$ would cover $G$. In fact, if not, assume that some node in $G$ cannot be covered. This means that the markings in this node cannot be reached by any environments. It is a contradiction to the assumption (H). Moreover, from the end of proof 1), $G_i \cap G_j = \emptyset$.
Another two sample data are: 1) for the environment factors 
\((0.308, 0.085, 0.876, 0.59, 0.303, 0.168, 0.34, 0.067, 0.22, 0.345, 0.092, 0.632)\) and the selection is \(B_1, S_1\), and 2) for the environment factors \((0.653, 0.244, 0.758, 0.299, 0.419, 0.056, 0.031, 0.186, 0.224, 0.797, 0.858, 0.427)\) and the selection is \(B_1, S_2\).

1) **Running Time Environment:** The markings of the continuous places \(p_{34} - p_{41}\) and \(p_{47} - p_{50}\) consists the environmental vector, i.e., \(e = (m_{34}, \ldots, m_{41}, m_{47}, \ldots, m_{50})\). The continuous transitions \(t_{C1} - t_{C12}\) set the values of \(e\).

2) **Local Computing:** Assume that the current environment vector is \(e_1 = (0.592, 0.378, 0.952, 0.742, 0.887, 0.598, 0.689, 0.391, 0.492, 0.71, 0.988, 0.334)\), where the first eight numbers are the inputs of \(NN_1\) of transition \(t_{1}^A\) in a component factory, and the rest four numbers are the inputs of \(NN_2\) of transition \(t_{2}^A\) in a supplier. The output of \(NN_1\) is 2.249 and the output of \(NN_2\) is 1.498. Then based on the adaptive rules, \(B_3\) is selected as a factory and \(S_2\) is selected as a supplier. We see that the computing is done locally, while the system reaches a global decision. The system is running with behavior \(B_1\) \([b\text{-}graph \text{ } G_1 \text{ in Fig. 18(f)}]\).

3) **Collaborative Adaptation:** Now assume that the environment changes from \(e_1\) to \(e_2 = (0.308, 0.085, 0.876, 0.59, 0.303, 0.168, 0.34, 0.067, 0.22, 0.345, 0.092, 0.632)\). Based on Theorem 1, the system changes from
behavior $B_1$ to other behavior, say $B_2$. Two components in the system then recompute the outputs of their neural networks and obtain the outputs as 0.618 and 0.5, respectively. Finally, $B_1$ and $S_1$ are selected, and the system is switched to run under $b$-graph $G_2$ as shown in Fig. 18(a). Fig. 19 displays the simulation of adaptation. At the beginning, the selection position is at point ($B_3, S_2$). When time goes to 19, the selection position is adapted to point ($B_1, S_1$), then to ($B_1, S_2$).

VII. DISCUSSION

A. Model Language

There are many languages that have been used to model self-adaptive systems, such as regular algebra, transition systems, automata, state machines, Markov models, graphs, process algebras, and Petri nets. For a survey, see [33]. The reason why we choose Petri nets as the model in this paper is that they can be used to model collaboration between components and can be converted into different input languages [32] of the existing analysis tools, such as SMV [8], SPIN [18], and SMC [31].

B. Scalability

It is a big adaptation concern. Our model language can model component-based systems. Each component can be modeled with a closed Process net with one or zero neural network. If a component does not require adaptation ability, then the closed process net is not needed; otherwise it has one. Two closed process nets with neural networks can cooperate together to model two components that cooperate. Our model can be extended to more components’ cooperation while still owns adaptation ability. For example, in our example, the manufacturing may have other components such as delivery besides factory and supplier. They can cooperate together to make decision and the adaptation is similar as before.

C. Adaptation

Our adaption is based on behavior switching. However, this switching is through the computing of neural networks with the unknown input range, which is different from existing mode switching, which is designed based on a known input range. Hence, in our self-adaptation, the learning algorithm is used to optimize adaptation decisions at runtime. The complexity of our adaptation is the complexity of online learning algorithm complexity, which is $O(w^2)$, $w = (m + n)(2m + 1)$, where $m$ is the number of the inputs of the neural network, i.e., the number of environment factors, and $n$ is the number of outputs, i.e., the sum of branch factories and suppliers.

D. Collecting Environment Data

In order to train a neural network, we need to collect the data from the environment. We need to care about the following three requirements.

1) Determine a time window: when to start and when to stop. If it is too small, we cannot catch the changes of the environment, if it is too big, then the trained neural networks may not offer the accurate outputs. We can define such window based on the experience for the similar systems or based on the real-time training results.

2) The training of the neural networks can be done online. All the parameters (weights) of neural networks can be reconfigured based on the real-time data to fit more realistic environments. In other words, if we expand $A$-transition in the Petri net, then its weight can be online updated.

3) The adaption capability of the model depends on the data, the more the better, but the more time is needed. It is required that the sample data can cover the scope of the input space.

VIII. CONCLUSION

This paper presents a new Petri net based on neural networks to model self-adaptive software systems. The proposed model can handle runtime environments and let components collaborate to make desired adaption decisions while the system is running.

In the future, we intend to consider the following issues.

1) Improving the Negotiation Ability: In the current version, the cooperation among components is linear, i.e., the cost of manufacturing is the linear combination of the costs of a factory and a supplier, later we need to consider the nonlinear case. Note that, in an APN, if a
Petri net representing a neural network needs the outputs of another Petri net representing another neural network, then two nets can be integrated together, i.e., two neural networks can be regarded as one and can be described by one learning Petri net.

2) **Model Checking of Properties**: The properties can be classified as local and global ones. The former are for the steady-state models that contain no adaptive behavior, i.e., a b-graph in our model. To check these properties, a solution is mapping our APNs to nonlinear hybrid automata, and then applies model checking tool, e.g., HyTech. Global properties happen when a model switches from one b-graph to another. For example, a global property could be: the system should find as good suppliers and factories as possible to fit the changes of 12 factors. Also, we can employ synchronous Petri net to model environment to restrict possible state space explosion in model checking.

3) **Extending Adaption Domains**: In addition to the current performance-based adaption, we may add a mechanism to the model for reliability-based adaption. Thus, we need to concern online bugs which might cover interference freedom, responsiveness, mismatch, and loss tolerance.

ACKNOWLEDGMENT

The authors would like to thank the Associate Editor and the reviewers for their many useful comments that helped them to improve this paper.

REFERENCES


**Zuohua Ding** (M’11) received the M.S. degree in computer science and the Ph.D. degree in mathematics, both from the University of South Florida, Tampa, FL, USA, in 1996 and 1998, respectively.

From 1998 to 2001, he was a Senior Software Engineer with Advanced Fiber Communication, Petaluma, CA, USA. He has been a Research Professor with the National Institute for Systems Test and Productivity, Vail, CO, USA, since 2001. He is currently a Professor and the Director with the Laboratory of Intelligent Computing and Software Engineering, Zhejiang Sci-Tech University, Hangzhou, China. His current research interests include system modeling, program analysis, service computing, software reliability prediction, and Petri nets. He has authored and coauthored over 70 papers.

**Yuan Zhou** is currently pursuing the M.S. degree with the Laboratory of Intelligent Computing and Software Engineering, Zhejiang Sci-Tech University, Hangzhou, China.

His current research interests include program analysis and hybrid system modeling. He has published seven research papers.

**Mengchu Zhou** (S’88–M’90–SM’93–F’03) received the B.S. degree in control engineering from the Nanjing University of Science and Technology, Nanjing, China, in 1983, the M.S. degree in automatic control from the Beijing Institute of Technology, Beijing, China, in 1986, and the Ph.D. degree in computer and systems engineering from Rensselaer Polytechnic Institute, Troy, NY, USA, in 1990.

He joined the New Jersey Institute of Technology, Newark, NJ, USA, in 1990, where he is currently a Distinguished Professor of electrical and computer engineering. His current research interests include Petri nets, Internet of things, semiconductor manufacturing, transportation, and energy systems. He has over 600 publications including 12 books, 290 journal papers (majority in the IEEE TRANSACTIONS), and 28 book chapters.

Prof. Zhou is the Founding Editor of the IEEE Press Book Series on Systems Science and Engineering. He is a Fellow of the International Federation of Automatic Control and the American Association for the Advancement of Science.