Proceedings of the 2022 Winter Simulation Conference
B. Feng, G. Pedrielli, Y.Peng, S. Shashaani, E. Song, C.G. Corlu, L. H. Lee., E. P. Chew, T. Roeder and P. Lendermann, eds.

REAL-TIME SCHEDULING BASED ON SIMULATION AND DEEP REINFORCEMENT LEARNING WITH FEATURED ACTION SPACE

Shufang Xie
Tao Zhang
Oliver Rose

Universität der Bundeswehr München
Werner-Heisenberg-Weg 39
Neubiberg, 85577, GERMANY

ABSTRACT

In this study, real-time scheduling is narrowed to the selection of one job to be processed from the queue of a machine when the machine becomes idle. It is considered to be one kind of sequential decision-making. Deep reinforcement learning with simulation has been widely used to make such decisions for most environments where the action space is either continuous or discrete but limited in size. However, in the real-time scheduling environment, the number of actions is the number of jobs in the queue which are varying over time. Moreover, if jobs arrive randomly, it is impossible to fix the actions. The action space is dynamic and stochastic. To overcome the difficulties raised by this, the action space is transformed into a featured action space. Actions are distinguished by their features. To apply the featured action space, three innovative structures of neural networks are proposed and compared with each other.

1 INTRODUCTION

Whenever a machine becomes idle during manufacturing, we need to decide which job in the queue should be processed first. Also, once a job arrives at a preemptive machine, we need to decide if the job on the machine can be interrupted. These are real-time scheduling. The goal is that all decisions together will result in a good performance of the manufacturing system, such as shorter cycle times, less tardiness, and so on. In reinforcement learning, agents learn how to make decisions in different situations through interacting with the environment to maximize a numerical reward signal received from the environment. The agents are not told which decisions to take but instead must discover which decisions yield the most reward by trying them or by the experience of the past trials. Deep reinforcement learning is the most popular algorithm in reinforcement learning now, which replaces the Q table with one or more neural networks and extends state space to featured state space in which the states are described by features. Because the states of a manufacturing site cannot be easily cataloged and put into the Q table due to the complexity of the manufacturing, deep reinforcement learning is much more suitable for real-time scheduling environments with complex and implicit state space. The factors concerned most can be included in the state features, like current WIP (work in process), work loading, resource availability, and so on.

In our previous study by Zhang et al. (2017), we already used simulation and reinforcement learning to do real-time scheduling. Jobs in the queue form an action set. Selecting one job to process is regarded as taking action from the set. The reward function comprises the critical ratio of the selected job and the global job holding cost. Since it is difficult to connect to a real system, a simulation environment is built. The learning procedure interacts with the simulation and learns the scheduling knowledge. Two simulation-based algorithms are introduced: simulation-based value iteration and simulation-based Q-learning. The simulation explores the state space and accomplishes state transitions. The value function is parameterized and estimated by using a feedforward neural network. In another study by Xie et al. (2019), we tried to answer two more detailed questions: how to quantify states and actions and how to define reward functions. Many experiments are carried out to investigate the influence of their definitions on the algorithm. The experimental results show that unnecessary features can only make the state space even larger and worsen the performance. So, taking only the necessary information as state features is always the right choice.

However, in both studies, we didn't mention too much about the action space. For different action spaces, we need to design different structures of neural networks. The learning algorithms are also different correspondingly. For example, if the action space is discrete, one neural network is needed with the state as inputs and Q values of all actions as outputs. The number of outputs equals the number of actions. Q-learning is usually used in this case (Hester et al., 2018). For continuous action space, two neural networks are created usually. One neural network inputs the state and outputs the distribution of actions. Actions are sampled from the distribution. Another neural network inputs the state and the sampled action and outputs the Q value. The most used learning algorithm for this is actor-critic learning (Grondman et al., 2012).

In real-time scheduling, the actions are jobs in the queue. Since the jobs arrive in the system randomly and the queue changes over time, the action set is dynamic and stochastic. If we want to include all jobs in the action set, it can be very huge in the long run as jobs keep arriving. In this study, we are going to tackle the problem raised by this kind of action space in the real-time scheduling environment and design the corresponding neural networks. The remaining contents of this paper are structured as follows: some common action spaces are discussed in Section 2; a featured action space similar to the featured state space is introduced to address the actions in the real-time scheduling environment in Section 3; In Section 4, we will propose several neural networks structures which are just fit to the featured action space. The experiments are carried out in Section 5 to evaluate the structures. The study is concluded in the last section.

2 TYPES OF ACTION SPACES

There are many different types of action spaces from different environment domains. The three most common types will be discussed here.

2.1 Discrete action space

Discrete action space is the most used space. It is usually just a set of actions, $A = \{a_1, a_2, \dots, a_p\}$. The size of the set p is fixed and quite limited. For example, in the grid world game, the action set A = {up, down, left, right}. Deep Q-learning is often used to handle such action space (Mnih et al., 2013). A more general form is a multi-dimension discrete action space that consists of a series of discrete action spaces with a different number of actions in each, $a = (b_1, b_2, \dots)$. For instance, a game controller has four keys K1, K2, K3, and K4, and one handler H. The keys can be either pressed or not pressed. The handler can switch in four directions. The action on the game controller $a = (b_{\mathrm{K1}}, b_{\mathrm{K2}}, b_{\mathrm{K3}}, b_{\mathrm{K4}}, b_{\mathrm{H}})$, where $b_{\mathrm{K1}}, b_{\mathrm{K2}}, b_{\mathrm{K3}}, b_{\mathrm{K4}} \in B_K$, $b_{\mathrm{H}} \in B_H$, $B_K = \{nope, pressed\}$ and $B_H = \{nope, up, down, left, right\}$. An asynchronous and deterministic variant of the asynchronous advantage actor-critic learning can deal with this kind of action space (Mnih et al., 2016).

2.2 Continuous action space

In continuous action space, an action is a real number. The action set $A = \{a | min \leq a \leq max\}$. The number can also be unbounded. Obviously, the size of the action set is infinite. For example, in a "Moving" game where an agent tries to find a target area starting from a random point in a sandbox environment, the actions are the turn angle, $0 \leq a < 360$. If we consider the movement in a 3D box, the action space becomes multi-dimensional. The action will be a vector $a = (x, y, z)$, and the action set $A = \{(x, y, z) | min_x \leq x \leq max_x, min_y \leq y \leq max_y, min_z \leq z \leq max_z\}$. A deep deterministic policy gradient (DDPG) agent is dedicated to such action space (Lillicrap et al., 2015).

2.3 Parameterized action space

For Parameterized action space, the action space is discrete, and the action set $A = \{a_1, a_2, \ldots, a_p\}$. The difference is that each action has parameters. After selecting one action, we still need to determine its parameters. The parameters can be either discrete or continuous. For example, in a soccer game, there are four mutually exclusive discrete actions: Dash, Turn, Tackle, and Kick. Agents must select one of these four to execute. Each action has 1-2 continuously valued parameters which must also be specified. An agent must select both the discrete action it wishes to execute as well as the continuously valued parameters required by that action. Hausknecht et al. (2015) proposed some algorithms to solve the problem with such action space.

3 FEATURED ACTION SPACE

As mentioned before, the action set is dynamic and stochastic in a real-time scheduling environment. The three common action spaces cannot represent such an action set and the standard deep reinforcement learning cannot solve problems with such an action set too.

To make the action set deterministic and reduce its size, we propose to convert the action space to a featured action space where actions are described by their features $a = \{y_1^a, y_2^a, \ldots, y_n^a\}$. y is one feature and n is the number of features. The features can be extracted by domain knowledge. For instance, the processing time and waiting time of jobs are the most important factors in job selection, both can be the features of actions. The number of features n is fixed. Moreover, the actions with the same features will be considered to be the same action. This can reduce the size of the action set. Figure 1 demonstrates the transformation.

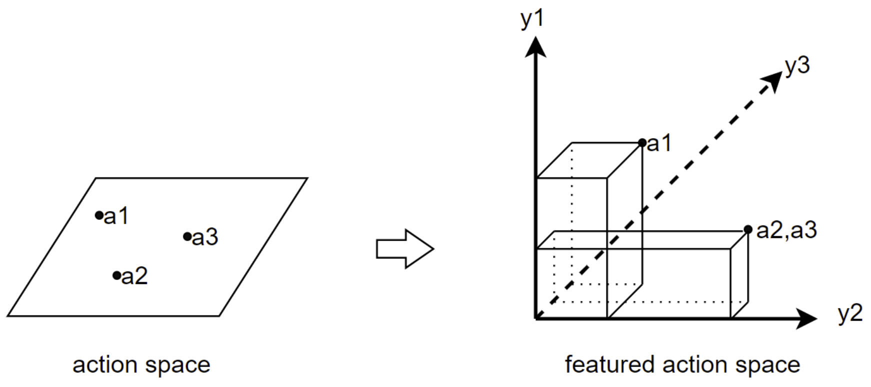

Figure 1: Conversion from action space to featured action space.

The featured action space is like the multi-dimensional action space. Each feature refers to one dimension, but the meaning is different. In multi-dimensional spaces, each dimension represents one sub-action. When making decisions, we must determine each sub-action. The sets or range of sub-actions are fixed and known in advance. However, features are only the information to describe actions in the featured action space. We use actions' features to decide which action should be taken. The values of features may only be obtained just before the decision-making, e.g., waiting times of jobs. Because decisions are made only at some discrete time points, the waiting times are discrete too. It is impossible to know the waiting times in advance, thus, we cannot create a set of waiting times. This is the main difference between the featured action space and the multi-dimensional action space where a finite or infinite set is always able to be created ahead.

4 STRUCTURES OF NEURAL NETWORKS

To introduce the featured action space to deep reinforcement learning, we either transform the featured action space into a fixed action space where the number of actions is fixed or modify the structure of neural networks, especially the inputs and outputs to adapt the space. If the space is transformed into a fixed action space, a neural network with the normal structure can be applied, as detailed in Section 4.1. Without the transformation, the number of actions in the featured action space is not fixed. The normal structure is not applicable. Thus, a new structure (Structure I) is presented in Section 4.2 where both state and action are the inputs of neural networks and Q value of the state and action is the output. To reduce the computation time of Structure I, a two-phase structure (Structure II) is addressed in Section 4.3 where a neural network outputs weights of action features according to a state (input) and then an equation calculates the Q value of one action in the state from the weights and values of the action's features. Based on Structure II, Structure III is introduced in Section 4.4. Structure III is also a two-phase structure, but the equation (the second phase) in Structure II is now replaced by another neural network.

4.1 Normal structure

As mentioned before, the number of actions is fixed in most use cases of reinforcement learning. Thus, the normal structure of neural networks is like Figure 2, in which the input number equals the number (m) of features X in the state space where state $s = \{x_1^s, x_2^s, \dots, x_m^s\}$. The output number is the number of actions. The output refers to the Q values of each action.

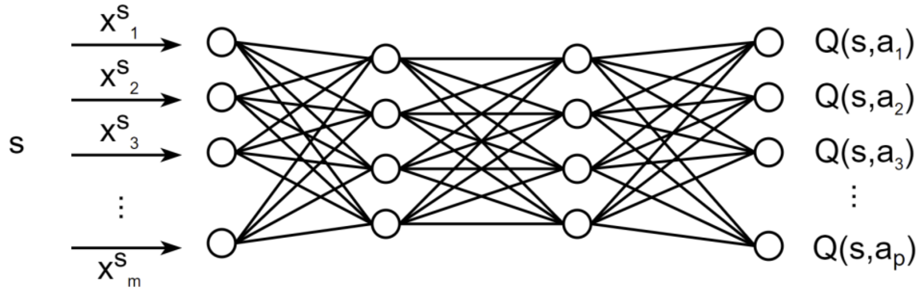

Figure 2: A normal structure of neural networks.

To apply this structure in our study, the featured action space must be transformed into a fixed action space. We can just divide the featured action space into several areas. Each area represents one theoretical action. Jobs whose features are all within one area will be the same action. To divide the space, <min, max, step > are usually defined on each feature and cut the feature to small pieces by the step. Any combination among the pieces of features denotes one theoretical action. If the size of one feature, i.e., the number of possible values, is quite limited, the values can be used directly without cutting. We can also use any cluster algorithms to divide the featured action space. In practice, it is very often that not all theoretical actions appear in decision-making. The invalid action masking technique can handle this problem (Huang and Ontañón, 2020).

4.2 Structure I

Instead of space transformation, we can also adjust the structure of neural networks to fit the featured action space. We can just input state s and action a together into the neural networks and output the value Q(s, a), shown in Figure 3. This can avoid the division of the action space and the invalid action masking because we just select among the valid actions. In this structure, the number of inputs is $m+n$ and the number of outputs is 1. The number of jobs in the queue has nothing to do with the structure.

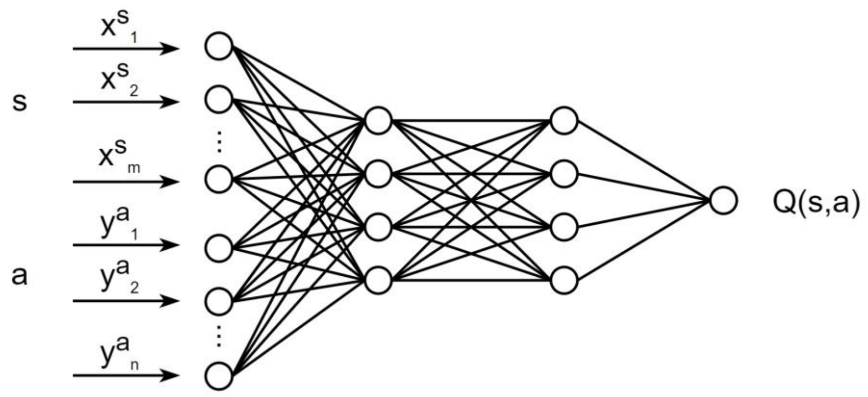

Figure 3: Demonstration of structure I.

When selecting one job from a queue, we just need to calculate the Q value for each action and select the action with the greatest value. When estimating the reward received from the future, we can just input the next state and each action in the next action set into the neural network and adopt the greatest Q value as the future reward. The disadvantage is that more computation is required.

4.3 Structure II

From Structure I, we can see that state s must be input into the neural network every time that we compute the Q value of an action. To prevent this from happening, we assume that the state influences Q value indirectly while action influences directly. The Q value is a sum of weighted action features $Q(s,a) = w_{y_1}y_1^a + w_{y_2}y_2^a + \cdots + w_{y_n}y_n^a$, where the weight of each feature is determined by the state. Thus, we create a neural network. The input is a state, and the output is the weights of all features in that state. The structure is shown in Figure 4. The features are standardized by removing the mean and scaling to unit variance. In this structure, the number of inputs is m and the number of outputs is n. When comparing different actions in a state, the weights are calculated only once and used in the equation for every action. We can also use this structure to figure out which features in action space dominate decision-making.

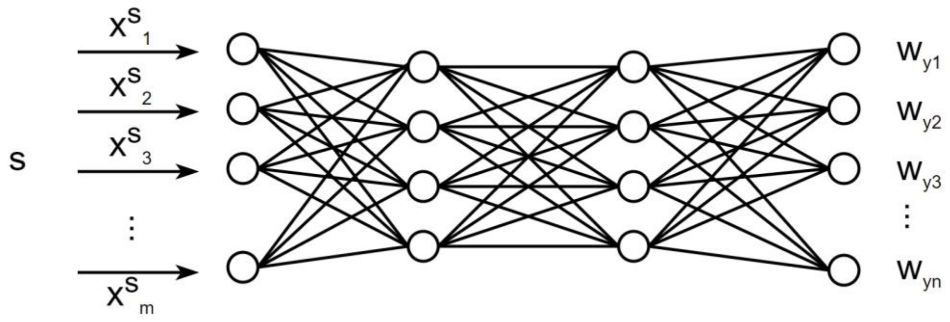

Figure 4: Demonstration of structure II.

4.4 Structure III

In this structure, instead of the linear summation of weighted action features in Structure II, the weighted action features are input into another neural network and the neural network outputs the Q value, shown in Figure 5. The Q value and action features have nonlinear relations. The input number is *n* and the output number is 1. To use this structure, we need to use the neural network in Structure II to calculate the weights first.

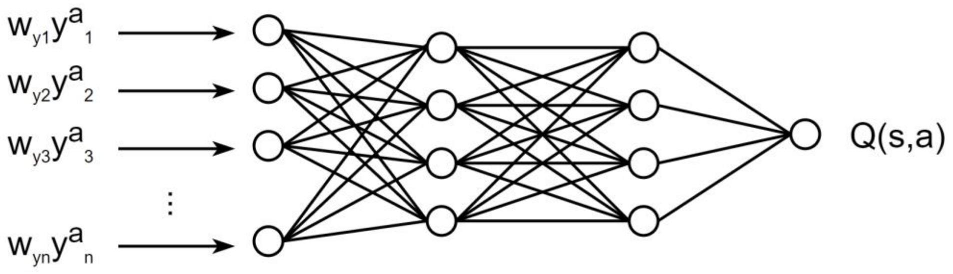

Figure 5: Demonstration of structure III.

5 ALGORITHM

Luckily, deep Q-learning agents are suitable for all four types of structures. A simple algorithm is shown in Figure 6. The simulation will run many times for updating the neural network. Simulation is advanced from one decision-making point to the next. At each point, state *s*, action set *A*(*s*), selected action *a*, and reward *r* are recorded. The neural networks are updated based on the previous state, previous action, current state, current action set, and current reward.

```
initialize neural network Q arbitrarily
$i \leftarrow 0$
loop ($i < Episode\_Num$)
    set previous state $s^- = null$ and previous action $a^- = null$
    initialize simulation
    while ( not sim_end)
        advance simulation to the next decision-making point
        $s \leftarrow$ current state, $A(s) \leftarrow$ jobs in the queue
        If $s^-$ is not null, Then
            $Q(s^-, a^-) \leftarrow r(s \mid s^-, a^-) + \gamma \max_{a' \in A(s)} Q(s, a')$,
            Update neural networks Q with pattern
            $< s^-, a^-, Q(s^-, a^-) >$,
        End If
        Select a job through $\varepsilon$-greedy,
            $a = \begin{cases} \arg\max_{a^* \in A(s)} Q(s, a^*) & \zeta < \varepsilon \\ random(A(s)) & otherwise \end{cases}$
        $s^- \leftarrow s$, $a^- \leftarrow a$

    $i \leftarrow i + 1$
```

Figure 6: Algorithm of deep Q-learning based on simulation.

$Episode\_Num$ denotes the number of simulation runs. γ is the discount ratio. ε is the probability to select an action by the current Q values. ζ is a random number within [0,1] generated before every selection.

6 EXPERIMENTS

6.1 Simulation environment

A simple discrete-event simulator is developed in Python for the experiments. Two types of special events: decision-making events and decision-made events, facilitate the learning process between the simulator and agent. Once a decision needs to be made in the simulation, a decision-making event is triggered and the simulation engine is paused. The agent responds to the event and makes decisions. Once the decision is made, the agent adds a decision-made event to the top of the event list of the simulator and resumes the engine. The simulator will respond to the event and take the decision and continue running to the next decision-making point. This implementation enables the simulator to be wrapped into a Gym environment which is the most popular environment interface for reinforcement learning. The algorithms developed for the Gym environment can be potentially used to solve our problem by using the simulator.

A deterministic and dynamic single-machine model is used to validate the structures and algorithm proposed in Sections 4 and 5. In the model, two types of products are produced on one machine and interarrival times of jobs are 10 and 100 minutes. Processing times for two types of products are 9 and 10 minutes. Preemption is allowed on the machine, i.e., a started job on the machine can be interrupted by another more urgent job and resumed later. If we assume that the current job is always put back into the queue when new lots arrive, the preemption decisions are merged into the job selection. The objective is to minimize the average cycle time. According to the scheduling theory, the optimal rule is always to select jobs with Shortest (remaining) Process Time (SPT). This is also the reason we selected this model. Because we can always see how much room is left for improvement.

Three features are considered in the state space: queue length, queue time, and average WIP. Two features are in the action space: remaining processing time and waiting time. Immediate rewards are calculated from a sum of time-weighted WIP levels, $r = B - \sum_{i=1}^{L} t_i \, WIP_i$, where B is a big enough number that makes sure r is always positive; t is the duration in which WIP does not change; L is the number of WIP changes between two decision-making. The reason we give the constant interarrival times is to use the standard learning algorithms and compare them with others. In the standard algorithm, 110 jobs released in 1000 minutes form the action set and are filtered by the release times in the algorithm.

6.2 Experimental results

The following scenarios are designed for the experiment, shown in Table 1. Scenario 1 just uses the optimal decision rule SPT to generate the optimal scheduling. Scenario 2 adopts the normal neural network structure and the fixed action space. Scenario 3 transformed the featured action space into another new action space to apply the normal structure. Scenarios 4-6 use the featured action space directly with structures I- III.

Deep Q-learning agents interact with the simulation environment and learn scheduling in Scenarios 2-6 respectively. $Episode\_Num$ is 1000 and the simulation length is 1000 minutes for all scenarios. The well-trained agents are then used as decision rules in the simulation. For all scenarios, an average cycle time is calculated and listed in Table 2. From the result, we can see that the average cycle times from Structure I, II, and III almost reach the optimal value from SPT. However, the normal structure with the transformed action space is the worst one because of the transformation from the featured action space to the new action space. The normal structure with the fixed action space can also reach the optimal value. This means reinforcement learning can solve the deterministic scheduling very well. However, the action space can never be fixed in real-time scheduling in practice. Addition work is done for Structure III to check the weights of each action feature at all decision-making points. At 96.3% of decision-making points, the weights of processing times are greater than the weights of waiting times. The behavior of the agent follows the SPT rule in most states.

Table 1: Experimental scenarios.

Scenario	Decision Rule	Action Space	Structure
1	SPT	-	-
2	agent	action space (fixed)	normal
3	agent	action space (transformed)	normal
4	agent	featured	structure I
5	agent	featured	structure II
6	agent	featured	structure III

Table 2: Experimental results.

Scenario	Key Info	Avg. Cycle Time (min)
1	SPT (optimal)	13.62
2	normal (fixed)	13.78
3	normal (transformed)	14.87
4	structure I (featured)	13.70
5	structure II (featured)	13.91
6	structure III (featured)	13.70

7 CONCLUSIONS

Because the action space in the real-time scheduling environment is stochastic and dynamic, the standard algorithms cannot be used to solve the problem. We transform the action space into the featured action space and proposed three types of neural network structures to calculate Q values from the featured action space. The structures are no longer dependent on the number of actions. The experiment shows that deep Q-learning with all three structures can achieve almost optimal scheduling. In addition, we need domain knowledge to extract features from the real system, this makes the agents not a black box anymore. Especially, we can know from Structure II and III which features are the most important for decision-making. This knowledge can be used directly in practice.

REFERENCES

Grondman, I., L. Busoniu, G. A. Lopes, R. Babuska. 2012. “A survey of actor-critic reinforcement learning: Standard and natural policy gradients”. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 42(6):1291-1307.

Hausknecht, M., P. Stone, 2015, “Deep reinforcement learning in parameterized action space”. https://arxiv.org/abs/1511.04143, accessed 21st April 2022.

Hester, T., M. Vecerik, O. Pietquin, M. Lanctot, T. Schaul, B. Piot, A. Gruslys. 2018. “Deep q-learning from demonstrations”. In *Proceedings of the AAAI Conference on Artificial Intelligence,* February 2nd-7th, New Orleans, USA, 3223-3230.

Huang, S., S. Ontañón, 2020, “A closer look at invalid action masking in policy gradient algorithms”. https://arxiv.org/abs/2006.14171, accessed 21st April 2022.

Lillicrap, T.P., J.J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, D. Wierstra, 2015, “Continuous control with deep reinforcement learning”. https://arxiv.org/abs/1509.02971, accessed 21st April 2022.

Mnih, V., K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, M. Riedmiller, 2013, “Playing atari with deep reinforcement learning”. https://arxiv.org/abs/1312.5602, accessed 21st April 2022.

Mnih, V., A.P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, K. Kavukcuoglu, 2016, “Asynchronous methods for deep reinforcement learning”. https://arxiv.org/abs/1602.01783, accessed 21st April 2022.

Xie, S., T. Zhang, O. Rose. 2019. “Online single machine scheduling based on simulation and reinforcement learning”. In *Proceedings of 18. ASIM-Fachtagung Simulation in Produktion und Logistik*. September 19th-20th, Chemnitz, Germany, 59-68.

Zhang, T., S. Xie, O. Rose. 2017. “Real-time job shop scheduling based on simulation and Markov decision processes”. In *Proceedings of the 2017 Winter Simulation Conference*, edited by W. K. V. Chan, A. D'Ambrogio, G. Zacharewicz, N. Mustafee, G. Wainer, and E. Page, 3899-3907. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

AUTHOR BIOGRAPHIES

SHUFANG XIE is a Research Assistant and Ph.D. student at Universität der Bundeswehr at the Chair of Modeling and Simulation. Her focus is on simulation-based scheduling and optimization of production systems. She received her M.S. degree in Metallurgical Engineering from Chongqing University, China. Her email address is shufang.xie@unibw.de.

TAO ZHANG is a Research Assistant at the Universität der Bundeswehr München, and he holds an M.S. in Metallurgical Engineering from Chongqing University, China and a Ph.D. in Computer Science from the Universität der Bundeswehr München, Germany. His research interest is working on production planning and scheduling, the main focus of his research is on the modeling and simulation of complex systems and intelligent optimization algorithms. His email address is tao.zhang@unibw.de.

OLIVER ROSE holds the Chair for Modeling and Simulation at the Department of Computer Science of the Universität der Bundeswehr, Germany. He received an M.S. degree in applied mathematics and a Ph.D. degree in computer science from Würzburg University, Germany. His research focuses on the operational modeling, analysis, and material flow control of complex manufacturing facilities semiconductor factories. He is a member of INFORMS Simulation Society, ASIM, and GI. His email address is oliver.rose@unibw.de.