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

DEEP REINFORCEMENT LEARNING FOR QUEUE-TIME MANAGEMENT IN SEMICONDUCTOR MANUFACTURING

Harel Yedidsion
Prafulla Dawadi
David Norman
Emrah Zarifoglu

Applied Materials Inc.
AI/ML Team

ABSTRACT

Queue-time constraints (QTC) define a limit on the time that a lot can wait between two process steps in its flow. In semiconductor manufacturing, lots that exceed that time limit experience yield loss, need rework, or get scraped. QTCs are difficult to schedule, since a lot needs to wait to be released to the first process step until there is available capacity to process the final step. However, exactly calculating if there is enough capacity is computationally expensive. In this work we propose a deep Reinforcement Learning (RL) method to manage releasing lots into the queue time constraint. We analyze the performance of our RL method and compare it to seven baseline solutions. Our empirical evaluation shows that the RL method outperforms the baselines in five performance metrics including the number of queue-time violations and makespan, while requiring negligible online compute time.

1 INTRODUCTION

Scheduling lots in a semiconductor manufacturing fab can be represented as a job-shop with reentry. Minimizing the makespan in such a setting is a known NP-Hard scheduling problem (Pinedo 2012).
Any scheduling decision requires considering many constraints such as the lots' critical ratio, due date, priority, and the stations' processing times, batching requirements, qualifications, down-time, bottleneck utilization, and more. In practice, real-time dispatching (RTD) rules use heuristic logic to aggregate all the constraints and considerations, and to prioritize the lots in each queue. However, relying entirely on dispatching can result in many lots experiencing queue-time violations (Cho 2014).

Queue-Time Constraints (QTC) define a maximum time limit between two processing steps, for example, between the wet bench and the furnace steps. Lots that finish processing on the wet bench station, must start processing on the furnace station within a certain time interval before chemical reactions such as oxidation, or contamination reduce the wafers' yield to a level that requires rework or even scraping the lots completely.

In order to prevent QTC violations, A Queue-Time Management System (QMS) is used to manage releasing lots into queue-time constraints via a pre-gate step – a virtual step upstream of the wet bench step. The QMS considers all the QTCs in the fab, and produces a release plan, which determines how many lots of each part type to release into the queue time loop at every timestep. The rationale for separating the QMS from dispatching is that dispatching logic must run very quickly whenever a station becomes idle, while the QMS solution takes time to compute. The separation allows the QMS system to run asynchronously every fixed time interval and produce a release plan.

Figure 1 shows the route we will use in this paper. It includes a virtual pre-gate step and two process steps with a QTC between them. Lots wait at the pre-gate step until they are released into the queue-time constraint, after which they process the wet bench and furnace steps. The furnace step must start within the queue time limit after completing the wet-bench step.

There are different approaches to control releasing lots, each offering different tradeoffs between violations, makespan, and computational complexity. Conservative heuristics such as the fixed-queue Kanban method (Scholl and Domaschke 2000) can minimize the number of violations with simple online compute, however optimally determining the length of the queue for minimizing makespan is challenging (Kopp et al. 2020). More complicated methods employ techniques such as constraint optimization (Cho 2014), or mixed integer programming (Klemmt and Mönch 2012) to accurately calculate a schedule which minimizes makespan without causing any violations, however due to the combinatorial complexity of the problem, finding an optimal solution and validating that no QTCs are violated is computationally intractable for large instances.

In this paper we propose a deep Reinforcement Learning (RL) based approach to solve the QTC problem which produces near optimal solutions quickly at runtime and can be used with any wet bench and furnace dispatching rules. We compare our method to seven other heuristic methods and show that it outperforms all the other methods in terms of the makespan and the number of violations.

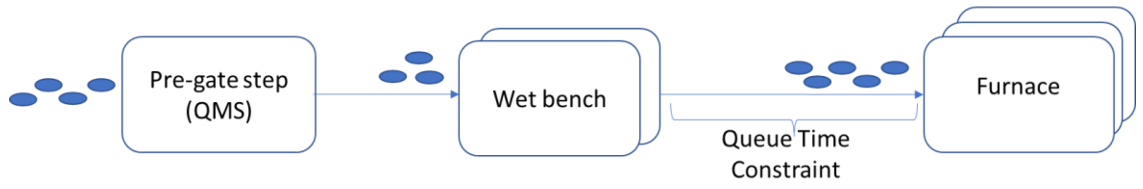

Figure 1: The Queue-time Management System Queue Time Constraint (QTC).

1.1 Reinforcement Learning

Reinforcement learning (RL) is a set of machine learning techniques that have been successfully applied to domains such as computer games (Wurman et al. 2022) complex games (Silver et al. 2017), robotics (Hanna and Stone 2017, Park et al. 2020), and control (Cui et al. 2021).

The RL problem consists of an agent and an environment. The agent takes actions which affect the environment and change its state. Following each action, the agent transitions to the next state and receives a reward. Over time the agent learns a behavior policy that maximizes the total accumulated reward. The environment is typically stated in the form of a Markov Decision Process (MDP) (Puterman 2014).

The queue time management problem we aim to solve in this paper can be modeled as a discrete-time, finite-horizon MDP which is a tuple $M = (S, A, P, R, \rho^0, \gamma, T)$, where S is a state set, A an action set, $P : S \times A \times S \rightarrow \mathrm{R}+$ a transition probability distribution, $R : S \times A \rightarrow \mathrm{R}$ a reward function, $\rho^0 : S \rightarrow [0, 1]$ an initial state distribution, γ is the discount factor, and T the time horizon.

A solution policy is a probability distribution $\pi : S \times A \rightarrow [0, 1]$ that maps states to actions. To find a solution policy, we train a reinforcement learning agent to learn a policy which maximizes the expected return $E_\tau \sum_{t=0}^{T} \gamma^t R(s^t, a^t)$ where $\tau := (s^0, a^0, s^1, a^1 \ldots)$ denotes a trajectory, $s^0 \sim \rho^0$, $a^t \sim \pi(s^t)$, , $s^{t+1} \sim P(s^t, a^t)$.

2 RELATED WORK

QTCs have been controlled using various computational approaches. In the paper by (Scholl and Domaschke 2000) the Kanban approach which maintains a fixed queue size was used. The authors of (Klemmt and Mönch 2012) proposed a mixed integer programming approach which decomposes the problem to manageable sub-problems with a maximal number of jobs. Constraint optimization was proposed as a solution method in (Choung et al.), while in (Mason et al. 2007) a genetic algorithm was developed. (Kim et al. 2020) proposed a supervised learning approach to train a deep neural network to predict lot dispatching in order to maximize station utilization. The labeled data required to train the model was obtained by domain experts through simulation.

In recent years, RL has been increasingly used for production planning and control (Panzer et al. 2021), and specifically in semiconductor fab scheduling in the following research applications:

(Lin et al. 2019) used the Deep Q Network algorithm (DQN) to choose a dispatch rule for optimizing makespan. (Park et al. 2019) used DQN to schedule setup changes to the fab's stations.

(Park et al. 2021, Waschneck, et al. 2018, Zhang et al. 2021, Zhou et al. 2020, and Shi et al. 2020) all used deep RL to perform dispatching to minimize metrics such as makespan, cycle-time, and cycle-time deviation. However, none of these papers considered QTC.

(Altenmüller et al. 2020) used DQN to perform task dispatching with QTC, however they did not compare their approach to existing QMS systems such as the Kanban approach (Scholl and Domaschke 2000).

To the best of our knowledge ours is the first work which develops a deep RL agent which is dedicated to controlling a queue time management system.

3 SOLUTION METHOD

In this section we describe the general solution framework, and components of the RL method; the state space, the action space, the reward function, and the RL algorithm that we used.

3.1 RL Training and Policy Updates

During training, the agent takes an action, the simulator applies that action and simulates one timestep into the future. The agent then receives a new state observation, and a reward. The state transition in our case is deterministic. The state-action-reward sequence is saved, and periodically the RL algorithm uses this experience to update the weights of the neural network which represents the policy. The policy is used to pick the next action. The policy updates aim to maximize the cumulative reward over the time horizon.

3.2 Testing

Once the learning curve stabilizes and the policy stops improving, we save the policy and use it to test the performance of the RL agent on a mix of seen and unseen environments. See Appendix A for an example of the RL algorithm's learning curve.

3.3 State Observation

At each timestep, the agent receives a state observation. The state of observation is comprised of the following components.

- Fab properties: step processing times, queue time constraints.
- Fab observations: number of lots processing per step, and per station.
- Queue time observations: number of successful lots, number of lots in violation, number of lots in process.
- Capacity observation: an estimation of the time to complete all the work in progress (WIP)

The state features are normalized to values in [0,1] and concatenated into a single observation vector.

3.4 Action Space

At each timestep, the agent can decide either to release or not to release a lot. The agent can release a lot of one of the N part types. Thus, the agent can choose a discrete action between 0 to N. Choosing an action 0 does not release any lots and action a_i releases a lot of type $Part_i$.

3.5 Reward Function

We designed a deterministic reward structure which encourages the agent to minimize the number of queue time violations while optimizing for makespan and the number of successful lots.

3.6 RL Algorithm - PPO

The Proximal Policy Optimization (PPO) algorithm (Schulman et al. 2017) is a popular deep RL algorithm which uses a policy gradient method to train a stochastic policy in an on-policy way. Also, it utilizes the actor critic method. In this paper we used the PPO implementation from StableBaselines3 (https://stable-baselines3.readthedocs.io/en/master/modules/ppo.html) and tuned its default parameters to fit the QMS application (please refer to Table 2 in the Appendix for more details on PPO's parameter setup).

4 EVALUATION

4.1 Baseline agents

We compare the performance of the reinforcement learning agent to that of seven baseline agents.

1. *Kanban* agent: Proposed in the paper by (Scholl and Domaschke 2000), this agent aims to maintain a fixed queue size in front of the furnace station family. The agent releases a lot if the number of lots in the queue is less than the desired queue size. The desired queue size is calculated as the maximal queue size which can be processed without causing any QTC violations.
2. *Capacity* based agent: At each time step, this agent computes the currently available capacity of the furnace station family. The capacity estimation is based on the Work In Progress (WIP) in the furnace family, including the steps that precede it. The agent releases a lot if the capacity is smaller than the predetermined threshold. This agent is somewhat similar to the *Kanban* agent, however, it allows for finer control since it is not limited to a fixed-size queue.
3. *Frequency* agent: This agent calculates the unified processing time of the furnace family and releases a lot at a frequency which is closest to that processing time, given the limitation that it can release a lot once every timestep.
4. *Random* agent: At each time step, the *Random* agent randomly chooses an action between releasing a lot or not releasing a lot with equal probability.
5. *Always* release agent: The *Always* release agent releases a lot at every time step, regardless of the current state. This agent provides a bound on the minimal makespan (although at the cost of many violations)
6. *Never* release agent: Contrary to the *Always* release agent, this agent never releases a lot. This agent is used to assess the number of violations that are caused by the initial starting conditions and provides a bound on the minimal number of violations possible.
7. *Q-learning* based agent: This agent uses the Q-learning algorithm to learn a policy to map current state to action. Q-learning (Sutton and Barto 2018) is a classic reinforcement learning algorithm that learns optimal action-value functions by iteratively updating Q-values using the update $Q(s^t, a^t) = Q(s^t, a^t) + \alpha[\, R^t + \gamma \max_a \underset{a}{} Q(s^{t+1}, a) - Q(s^t, a^t)]$,

where $Q(s^t, a^t)$ is the value of the action a^t at the state s^t, γ is the discount factor, and α is the learning rate. We implemented this RL algorithm to benchmark the performance of PPO against another RL algorithm.

4.2 Experimental Setup

In this subsection we describe the implementation details of our experiment framework.

4.2.1 Simulator and environment setup

We designed a custom-built simulator to simulate a fab with QTCs. The simulator allows us to flexibly define a fab environment with any number of stations, station families, and lots of multiple part types. For each part type we define a route, which is a set of processing steps. Each step is assigned to a station family and has its own processing time. Any pair of steps in a route can have a QTC between them. The simulator supports releasing lots of multiple part types.

The experimental setup that was used for the evaluation is similar to the one in Figure 1 where the system has one pre-gate step, a wet bench and two furnaces. The pre-gate step has 0min processing time and controls the release of lots into the wet bench queue. The wet bench has a constant processing time of 20min, and the two furnaces have different processing times (600min, and 700min). Each station has a capacity of one lot at a time. Lots start at the pre-gate step and must go through the wet bench and the furnace to finish their route. There is a QTC between the wet bench step and the furnace step which is set to 200min. This means that any lot that finished processing on the wet bench must start processing on a furnace within 200min to be considered successful, otherwise it violates the QTC. We follow the convention in (Altenmüller 2020) where if a constraint is violated the violating lot continues to the end of the route.

Note that in an actual semiconductor factory wet benches and furnaces process batches of potentially more than one lot at a time. Our method could be extended to this case by releasing batches of lots at the pre-gate step instead of individual lots.

4.2.2 Dispatch rules

For the furnace station family, we used a queue-time based dispatch rule that orders lots based on their remaining queue time, so that the lot that has the least amount of time to violate its QTC will be scheduled first. For the wet bench station family we used a first-in-first-out (FIFO) dispatch rule.

4.2.3 Warm-up

At the beginning of every experiment the queues in front of the station families are empty. To avoid these unrealistic and uniform starting conditions, and bring the system to steady production state, we implement a warm-up stage. Each experiment has a different warm-up stage where we randomly release lots into the system for a given number of timesteps in order to create different starting conditions. We used a different warm-up period for training and testing (3 and 6 timesteps respectively) to prevent overfitting in the RL algorithms. Because the warm-up doesn't consider QTC, the released lots may not be able to process without violations. In the results section we do not consider the violations that were caused by the lots that were released during the warm-up period.

4.2.4 Episode termination conditions

Each experiment (or episode) has at most 100 timesteps, and each timestep takes 100 minutes. At each timestep the agent can take a single action. Following that, the RL agents receive an observation reflecting the system's state at the end of the timestep. An episode terminates when 100 timesteps have passed, or when at least 10 lots complete the route, whichever happens first.

4.2.5 RL integration, training, and testing

In order to train and evaluate the RL agents, we integrated the simulator with OpenAI Gym (Brockman et. al 2016). OpenAI Gym is an open-source Python library for developing and comparing RL algorithms by providing a standard API to communicate between learning algorithms and environments. The RL agents were trained for 5000 episodes.

For testing we let each agent control the pre-gate step for 30 episodes. Each episode had starting WIP randomly created during the warm-up. We set the same random seed for each agent so that they will all face the same 30 randomly generated environments. We averaged the evaluation metrics for each agent and summarized them in Table 1.

4.3 Results

In this section we analyze the results of the empirical evaluation. We compare the agents on five metrics including:

- the average number of violations per episode,
- the average number of successes per episode,
- the average makespan in minutes to complete processing 10 lots, with a maximum time of the episode length (including warm-up),
- the average utilization of the furnace family, which is computed as the fraction of time that the furnace stations were processing out of the total episode time,
- and the average cycle time of the finished lots in minutes.

The *Never* agent gets the minimal number of violations (0.97 on average) which are created from the warm-up. Other agents (*Capacity*, *Kanban*, *Q-Learning*, and *PPO*) are also able to reach that minimal number of violations. In Table 1 we present the adjusted number of violations after subtracting 0.97 from each entry.

Out of the agents that got the minimal number of violations, *PPO* has the highest number of successes with 11.83 successes on average.

The *Always* agent has the shortest makespan with 31.33, nearly matched by *PPO* with 32.1 timesteps on average. Note that the *Always* agent's short makespan comes at the cost of many violations as the *Always* agent has the highest number of them with 29.9 on average.
PPO also achieves near optimal furnace utilization with 0.93 compared to 0.94 achieved by the *Always* agent, and near optimal cycle time with only 1 percent more than the cycle time of the *Always* agent.

The average cumulative reward, which is used for training the RL agents, is not a metric in its own right but is useful in ranking the agents' performance.

In terms of compute time, the *PPO* method can quickly make the decision on the next action given a state observation, and we do not expect that time to grow exponentially with the size of the problem (number of part types, number of lots, number of stations) as opposed to exact solution methods such as constraint optimization. However, we do expect the training time to increase as the state and action space increase.

Table 1: Empirical evaluation results. Each entry is averaged over 30 evaluation runs. Cells with green background indicate the best possible value for the given metric as achieved by the *Always* and *Never* agents. Cells with yellow background highlight the agents that achieved the best value for a given metric.

	Metrics					
Methods	**#Violations above minimum**	**#Successes**	**Makespan** (timesteps)	**Reward**	**Utilization**	**Cycle Time** (Minutes)
Always	29.9	2.4	**31.33**	-30.83	**0.94**	**2182**
Random	14.1	3.13	31.47	-14.96	0.93	2235
Never	**0**	1.97	93	-1.59	0.1	--
Frequency	2.9	7.77	33.3	-3.32	0.87	2324
Kanban	**0**	9.03	55.4	-0.51	0.56	3194
Capacity	**0**	10.03	46.43	-0.32	0.63	2855
Q-Learning	**0**	11.03	33.4	-0.09	0.88	2303
PPO	**0**	**11.83**	**32.1**	**0**	**0.93**	**2207**

5 CONCLUSIONS

In this work we developed a deep RL agent that can efficiently control a QMS achieving optimal performance in terms of minimizing the number of violations, while maintaining near optimal makespan, without having to run time-consuming computation during deployment. Our approach outperformed seven other tested benchmarks including the Kanban method.

Directions for future work include examining more complex environments which represent real world fab conditions more accurately. For example, considering more parts, longer routes, batching, station dedication, and multiple QTCs.

APPENDIX A – PPO PARAMETERS AND LEARNING CURVE

Table 2 details the PPO parameters' values that were used in this study. The values were chosen following a hyper-parameter tunning phase in which the default PPO parameter values were gradually modified in both directions until a better value was found or, if not then the default values were preserved.

Table 2: List of PPO parameters and values.

Parameter Name	Parameter Value
batch_size	50
gae_lambda	0.99
learning_rate	0.00025
n_steps	50
pi net_arch	[8,10]
vf net_arch	[8,10]
activation_fn	ReLU

Figure 2 shows the learning curve of the PPO algorithm. The cumulative episodic reward stables out at 0 after 500,000 timesteps.

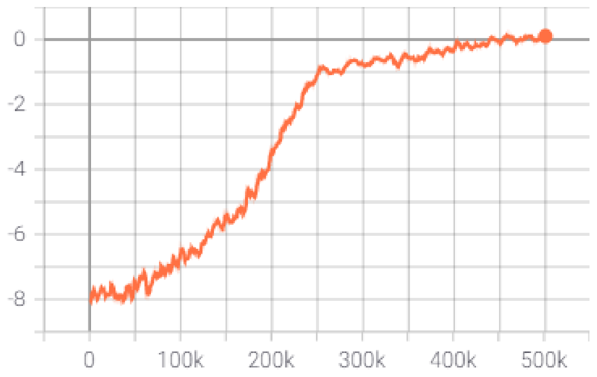

Figure 2: PPO's learning curve taken from Tensorboard.

REFERENCES

Altenmüller, T., Stüker, T., Waschneck, B., Kuhnle, A. and Lanza, G., 2020. *Reinforcement learning for an intelligent and autonomous production control of complex job-shops under time constraints*. Production Engineering, 14(3), pp.319-328.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). *Openai gym*. ArXiv Preprint ArXiv:1606.01540.

Cho, L., Park, H.M., Ryan, J.K., Sharkey, T.C., Jung, C. and Pabst, D., 2014. *Production scheduling with queue-time constraints: Alternative formulations*. In IIE Annual Conference. Proceedings (p. 282). Institute of Industrial and Systems Engineers (IISE).

Choung, Y.I., Jun, K.S., Han, D.S., Jang, Y.C., Lee, T.E. and Leachman, R.C., 2002. *Design of a scheduling system for diffusion processes*. In SMOMS (pp. 1-6).

Cui, J., Macke, W., Yedidsion, H., Goyal, A., Urieli, D. and Stone, P., 2021. *Scalable Multiagent Driving Policies for Reducing Traffic Congestion*. In Proceedings of the 20th International Conference on Autonomous Agents and Multi Agent Systems (pp. 386-394).

Hanna, J.P. and Stone, P., 2017, February. *Grounded action transformation for robot learning in simulation*. In Thirty-first AAAI conference on artificial intelligence.

Kim, H., Lim, D.E. and Lee, S., 2020. *Deep learning-based dynamic scheduling for semiconductor manufacturing with high uncertainty of automated material handling system capability*. IEEE Transactions on Semiconductor Manufacturing, 33(1), pp.13-22.

Klemmt, A. and Mönch, L., 2012, December. *Scheduling jobs with time constraints between consecutive process steps in semiconductor manufacturing*. In Proceedings of the 2012 winter simulation conference (WSC) (pp. 1-10). IEEE.

Kopp, D., Hassoun, M., Kalir, A. and Mönch, L., 2020, December. *Integrating critical queue time constraints into SMT2020 simulation models*. In 2020 Winter Simulation Conference (WSC) (pp. 1813-1824). IEEE.

Lin, C.C., Deng, D.J., Chih, Y.L. and Chiu, H.T., 2019. *Smart manufacturing scheduling with edge computing using multiclass deep Q network*. IEEE Transactions on Industrial Informatics, 15(7), pp.4276-4284.

Mason, S.J., Kurz, M.E., Pohl, L.M., Fowler, J.W. and Pfund, M.E., 2007. *Random keys implementation of NSGA-II for semiconductor manufacturing scheduling. International Journal of Information Technology and Intelligent Computing*, 2(3).

Panzer, M., Bender, B. and Gronau, N., 2021. *Deep Reinforcement Learning In Production Planning And Control: A Systematic Literature Review*. ESSN: 2701-6277.

Park, I.B., Huh, J., Kim, J. and Park, J., 2019. A reinforcement learning approach to robust scheduling of semiconductor manufacturing facilities. IEEE Transactions on Automation Science and Engineering, 17(3), pp.1420-1431.

Park, J., Chun, J., Kim, S.H., Kim, Y. and Park, J., 2021. *Learning to schedule job-shop problems: representation and policy learning using graph neural network and reinforcement learning*. International Journal of Production Research, 59(11), pp.3360-3377.

Park, J.S., Tsang, B., Yedidsion, H., Warnell, G., Kyoung, D., Stone, P. and Sony, A.I., 2020. *Learning to improve multi-robot hallway navigation*. In Proceedings of the Conference on Robot Learning, (pp. 1883-1895). PMLR.

Pinedo, M.L., 2012. *Scheduling* (Vol. 29). New York: Springer.

Scholl, W. and Domaschke, J., 2000. *Implementation of modeling and simulation in semiconductor wafer fabrication with time constraints between wet etch and furnace operations*. IEEE Transactions on Semiconductor Manufacturing, 13(3), pp.273-277.

Puterman, M.L., 2014. Markov decision processes: discrete stochastic dynamic programming. John Wiley & Sons.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A. and Klimov, O., 2017. Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347.

Shi, D., Fan, W., Xiao, Y., Lin, T. and Xing, C., 2020. *Intelligent scheduling of discrete automated production line via deep reinforcement learning. International journal of production research*, 58(11), pp.3362-3380.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A. and Chen, Y., 2017. *Mastering the game of go without human knowledge*. nature, 550(7676), pp.354-359.

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT press.

Wurman, P.R., Barrett, S., Kawamoto, K., MacGlashan, J., Subramanian, K., Walsh, T.J., Capobianco, R., Devlic, A., Eckert, F., Fuchs, F. and Gilpin, L., 2022. *Outracing champion Gran Turismo drivers with deep reinforcement learning*. Nature, 602(7896), pp.223-228.

Waschneck, B., Reichstaller, A., Belzner, L., Altenmüller, T., Bauernhansl, T., Knapp, A. and Kyek, A., 2018, April. *Deep reinforcement learning for semiconductor production scheduling*. In 2018 29th annual SEMI advanced semiconductor manufacturing conference (ASMC) (pp. 301-306). IEEE.

Zhang, C., Song, W., Cao, Z., Zhang, J., Tan, P.S. and Chi, X., 2020. *Learning to dispatch for job shop scheduling via deep reinforcement learning*. Advances in Neural Information Processing Systems, 33, pp.1621-1632.

Zhou, L., Zhang, L., & Horn, B. K. (2020). *Deep reinforcement learning-based dynamic scheduling in smart manufacturing*. Procedia Cirp, 93, 383-388.

AUTHOR BIOGRAPHIES

HAREL YEDIDSION is a research scientist at the AI/ML team at Applied Materials. He earned his Ph.D. form the departement of Industrial Engineering and Management at Ben Gurion University in Israel, and was a postdoc fellow at the department of Computer Science at the University of Texas at Austin. His research interests include multi-agent systems, robotics, and reinforcement learning. His e-mail is harel_yedidsion@amat.com.

PRAFULLA DAWADI is a data scientist at AI/ML team at Applied Materials. He earned his Ph.D. in computer science from the School of Electrical Engineering and Computer Science at Washington State University, Pullman,WA. His research interests includes data science, machine learning systems and reinforcement learning. His email address is prafulla_dawadi@amat.com.

DAVID NORMAN a distinguished member of technical staff at Applied Materials. He earned his Ph.D. in mathematics at the University of Minnesota. His interests include applying mathematical optimization, machine learning, and reinforcment learning to scheduling and planning problems in manufacturing and supply-chain managmement. His e-mail address is david_norman@amat.com.

EMRAH ZARIFOGLU is Head of R&D for APG group in Applied Materials. His research and work interest is focused in the area of cloud computing, artificial intelligence, machine learning, supply chain, semiconductor manufacturing, operations research, simulation. He holds a PhD degree in Operations Research and Industrial Engineering from The University of Texas at Austin. His email address is emrah.zarifoglu@amat.com.