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.

A CUSTOMIZABLE REINFORCEMENT LEARNING ENVIRONMENT FOR SEMICONDUCTOR FAB SIMULATION

Benjamin Kovács
Pierre Tassel
Martin Gebser

Alpen-Adria-Universität Klagenfurt
Universitätsstraße 65-67
Klagenfurt am Wörthersee 9020, AUSTRIA

Georg Seidel

Infineon Technologies Austria AG
Siemensstraße 2
Villach 9500, AUSTRIA

ABSTRACT

Reinforcement learning based methods are increasingly used to solve NP-hard combinatorial optimization problems. By learning from the problem structure, or the characteristics of instances, the approach has high potential compared to alternative techniques solving all instances from scratch. This work introduces a novel framework for creating (deep) reinforcement learning environments simulating up to real-world scale semiconductor fab scheduling problem instances. The highly configurable framework supports creating single- and multi-agent environments where the simulation factory is either partially or fully controlled by the learning agents. The action and observation spaces and the reward function are customizable based on pre-defined features. Our toolkit creates environments with a standard interface that can be integrated with various algorithms in a few minutes. The simulated datasets may involve challenging features like downtimes, batching, rework, and sequence-dependent setups. These can also be turned off and simulated datasets be automatically downscaled during the prototyping phase.

1 INTRODUCTION

Efficient scheduling of semiconductor (SC) fabs remains challenging, as the problem is NP-hard and real-world problem instances tend to be intractable for many methods due to the size of the problem (Bureau et al. 2006; Waschneck et al. 2016). The manufacturing process usually consists of hundreds of steps with possibly different routes for manifold product types, while machines can be allocated dynamically.

The issue became increasingly relevant recently. Due to the effects of the pandemic, multiple sectors are hit by the shortage of semiconductor parts globally. As a result of the lack of chips, the auto industry has to halt production temporarily, while the prices of consumer electronics are constantly rising. Besides costly factory expansions, a better way of increasing production volume is improving the utilization of the current resources by optimizing factory schedules and developing adaptive methods to handle the dynamic circumstances and the enormous number of orders (Attinasi et al. 2021).

Deep reinforcement learning (RL) (Arulkumaran et al. 2017; Kaelbling et al. 1996; Sutton and Barto 2018) based methods demonstrated the capability of tackling large-scale problems that no alternative algorithms managed before. Well-trained RL agents outperformed previous state-of-art algorithms in playing board and video games and even reached superhuman performance for some applications (Mnih et al. 2013; Silver et al. 2016). Recently, solving combinatorial optimization problems (COPs) with RL was also attempted in several areas, aiming to replace approximation and heuristic methods. The capabilities of RL have already been proven for large-scale planning and scheduling problems (Guo et al. 2021; Tassel et al. ; Waschneck et al. 2018), including examples from the semiconductor industry in the chip design process (Mirhoseini et al. 2020). However, there are many open problems, especially in the case of practical

applications. In industrial contexts, algorithms must be robust and trustable in handling a highly dynamic setting with uncertain processing times and unpredictable events like breakdowns and extra work.

However, training RL agents has high sample complexity, and reaching an acceptable policy quality for large-scale problems usually requires collecting billions of samples. Therefore, to avoid disruptions to the production process, initial training and evaluation of the agent take place in simulations. In an industrial context, scheduling techniques tend to be simulated using customized commercial software solutions. Since these software tools and datasets are not publicly available, researchers usually train agents on smaller-scale custom datasets (Kuhnle et al. 2021; Suerich and Young 2020). The lack of a universal benchmark suite makes measuring scientific progress challenging.

We aim to fill the identified gap by creating a RL toolbox based on the semiconductor fab simulator (Kovács et al. 2022) we developed to compare different scheduling strategies. To the best of our knowledge, the resulting artifact is the first scalable open RL environment built for semiconductor fab scheduling. The main contributions of our work are the following:

1. Building on our semiconductor fab simulator, we introduce an open-source tool to create custom RL environments simulating large-scale manufacturing plants.
2. Our tool supports method development end-to-end, from small scale prototyping to training, evaluating and comparing agents on large instances. The bundled dispatching strategies make it easy to evaluate methods against greedy policies.
3. The novel toolkit makes it possible to measure scientific progress on real-world scale problems and compare novel algorithms. We equip our tool with the de facto standard OpenAI gym interface and make our tool freely available on https://github.com/prosysscience/PySCFabSim-release.

The rest of this paper is organized as follows. In Section 2, we provide the background of our work, including a summary of alternative simulation tools and RL-based methods focusing on the semiconductor industry in Section 2.3. Our toolkit is introduced in Section 3, followed by hints for practical applications and benchmark results of an example evaluation in Section 4. Finally, Section 5 concludes the paper and proposes further research directions.

2 BACKGROUND

This sections covers the necessary background on semiconductor fab scheduling and reinforcement learning. Additionally, a brief review of available RL-based scheduling approaches and environments points out the novelty and importance of our tool, compared to the current alternatives.

2.1 Semiconductor Fab Scheduling

Semiconductor manufacturing is one of the most complex production processes (Bureau et al. 2006), taking up to several hundreds of steps to process a lot until completion, while scheduling is subject to various constraints and uncertainty. There are multiple classes of machines requiring different scheduling tactics. For example, batching machines may process multiple lots, while cluster machines process multiple consecutive steps on a single wafer in their chambers. Factories usually manufacture hundreds of different products that also have different routes. Development lots contain products manufactured in small quantities but may have unique routes and more urgent deadlines. Some steps are subject to time coupling requiring capacity pre-allocation or machine dedication constraints, and the processing times depend on the machine assignments. Additionally, factory layouts change dynamically, as companies aim to resolve manufacturing bottlenecks and expand capacities, or machine breakdowns occur. The extent of the problem makes real-world cases intractable for exact methods. In practice, handcrafted dispatching heuristics are popular solving strategies for large instances, while bottlenecks and difficult core problems may be addressed by exact methods (Waschneck et al. 2016; Kopp et al. 2020; Pfund et al. 2006).

2.2 Reinforcement Learning

An RL agent (Sutton and Barto 2018) interacts with the environment, modeled as a Markov Decision Process (MDP), in discrete steps. In each step, the agent receives a state vector from the environment and picks an action. Then, the environment performs the selected action, and the agent receives a reward and a new observation. The procedure is repeated for a fixed number of steps or until the environment reaches a terminal state, ending the episode. The agent's goal is to maximize the return, the sum of the rewards for an episode.

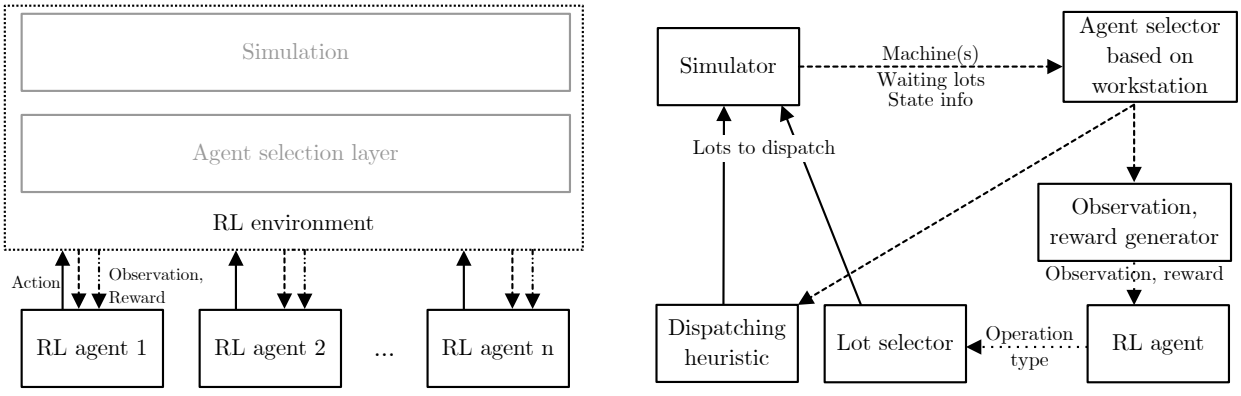
Deep RL (Arulkumaran et al. 2017; Kaelbling et al. 1996; Sutton and Barto 2018) combines deep neural networks' capability of learning representations with reinforcement learning. In deep RL, the agent uses a deep neural network as a function approximator for the policy and value functions. The technique enables solving real-world scale environments with high-dimensional observation spaces.

In multi-agent reinforcement learning (Buşoniu et al. 2010; Zhang et al. 2021), multiple agents interact with a common environment. The agents' goal is to cooperatively maximize the global reward (Figure 1). The environment selects the agent involved in the upcoming decision and presents an observation and a reward to the agent. The agent chooses an action based on the observation and forwards it to the environment. This procedure is repeated until the end of an episode (goal reached, time elapsed, etc.). Depending on the problem, the agents may take actions in parallel, or sequentially, based on the state of the environment.

2.3 Related Work

Deep RL methods have already demonstrated their utility for solving complex COPs. Jacobs et al. (2021) combine an exact method for a max-min problem with deep RL, where the agent has to learn a heuristic for the outer maximization problem. During the training process, the reward is defined as the advantage of the RL agent relative to a default heuristic. The approach proved to solve the capacitated vehicle routing and the traveling salesman problem efficiently.

Scheduling Domains. Guo et al. (2021) present an RL-based approach for a multi-resource scheduling problem concerning the optimization of cloud resources. By combining the method with imitation learning, the training time can be significantly reduced.



(a) Multi-agent RL environment. In our tool, the simulation can be separated from the RL layer. The latter is responsible for selecting an agent and constructing observations, mapping core simulation by a gym interface.

(b) The combined heuristic and RL dispatching framework of our tool. An RL agent can be localized to particular workstations, while dispatching heuristics handle the rest of the factory.

Figure 1: Structure of multi-agent RL problems and our environment.

Tassel et al. () solve the job shop scheduling problem with respect to minimizing the makespan using a deep RL (proximal policy optimization) based method. The authors use a reward function based on machines' scheduled area (utilization) to make the learning process more efficient. The measure shows a hight correlation with the resulting makespan. The trained single agent outperforms several dispatching methods on classical Taillard's instances (Taillard 1993).

Paeng et al. (2021) propose a deep RL-based framework to schedule real-world manufacturing systems, where product demands and due dates are dynamic. The developed state and action spaces are independent of varying parameters, so that an agent can handle diverse instances of the problem class without retraining. A solution using deep Q-networks outperforms meta-heuristics and alternative RL-based methods for scheduling independent jobs (without sub-tasks), where a job can be processed on multiple machines and setup changes may be required for switching between jobs of different families. Compared to application scenarios in semiconductor manufacturing, the solved problem lacks the support of batching machines, and all jobs are available for processing from the beginning, i.e., each job consists of a single step.

Semiconductor Industry. Park et al. (2020) model a part of the semiconductor manufacturing process with jobs consisting of numerous operations. A machine with a prescribed setup should perform each operation, and an RL-based method was proposed to tackle the dispatching problem. When a machine becomes available, the agent observes the number of waiting operations of each operation type, the setup status, and the action- and utilization history. Then, it selects the performable operation type. The reward is designed to ensure that its sum matches the objective function (makespan). Selecting the operation can implicitly lead to a setup, which is automatically performed on demand. The developed solution outperforms a genetic algorithm and a rule-based approach on instances with up to 175 machines and about 3000 operations. Smaller-scale instances are used for training, followed by deployment on larger cases.

Kuhnle et al. (2021) investigates a similar order dispatching problem. However, here the agent focuses on transporting lots between lot sources, and input and output buffers of machines. Then, the machines process the contents of their buffers lot by lot, so the method is limited to non-batching machines. The authors propose an RL framework and demonstrate its usage in simulations of two scenarios (focus on optimizing transport or operation sequence) on small instances with few machines. They conclude that, contrary to rule-based methods, the RL-based agent (using trust region policy optimization) adapts to both scenarios well. Notably, the training was performed on custom simulation tools and closed datasets.

3 THE RL FRAMEWORK

This section presents our toolbox. First, we set up requirements to be fulfilled by the developed software. Second, we describe the base dataset and briefly review our core simulator. Then, we introduce the building blocks our RL environment creator framework, starting with its general structure, followed by customization options regarding the actions, reward functions and observation spaces.

3.1 Relevance of Our Toolbox

The reviewed literature shows that previous works evaluate methods on custom simulators built upon small-scale datasets or toy problems that neglect details essential for practical usage, like batching, or the capability to adapt to factory upgrades. We instead develop a framework targeting applications in real-world factories to be a drop-in replacement for rule-based heuristics. The methods to be developed solve the challenge of adaption to dynamic environments, and use the same interface as the dispatching strategies widely adopted in industry. Contrary to other fields of machine learning and RL that work with standardized datasets, making the performance of methods comparable, the diverse evaluation standards in SC fab scheduling complicate measuring scientific progress and comparing methods against each other. Our toolbox addresses this shortcoming and provides a uniform framework for experimentation and evaluation.

3.2 Requirements

Considering the goal of developing a universal benchmark tool to evaluate RL-based approaches on problems in the semiconductor fab scheduling domain, the design requirements for the devised software artifact are:

1. General:
   (a) Simulated instances should be close to real-world scheduling problems.
   (b) Open license for all components: both datasets and the software must be freely available, and only open-source libraries should be used for the implementation.
   (c) Examples should be provided to demonstrate the creation of custom environments.
2. Reinforcement learning-related:
   (a) Support for various agent configurations: the number of agents and machines families controlled by each agent should be configurable by the developer.
   (b) Customizable environments: the goal of customizability includes the freedom of developers and researchers to access the internal simulation state, process raw data and compute novel metrics based on it, while also quick prototyping should be supported by pre-defined features.
   (c) Usage of standard interfaces: the OpenAI gym interface (Brockman et al. 2016) and accompanying monitoring libraries should be readily integrated.

3.3 Dataset & Factory Model

The semiconductor manufacturing process involves several constraints that make modeling and solving challenging. Since it is one of the most complex real-world scheduling problems, its unique characteristics require certain experience to create realistic models and instances of a problem domain. Our goal is to develop a reinforcement learning framework based on an existing model to ensure correspondence to real-life circumstances and to enable the validation of the simulation tool.

Based on our literature review, the SMT2020 dataset (Kopp et al. 2020) proved to be our top candidate. The dataset is recent, available online, and aims to represent circumstances in real-world factories with routes involving hundreds of steps as well as a large variety of machine families. Additionally, it incorporates the major challenges of wafer fabrication introduced in (Bureau et al. 2006). The dataset contains high volume-low mix, and low volume-high mix problem instances, optionally with development lots.

We use a factory model based on the selected dataset. There are multiple products, and each product has a separate route with hundreds of operations. New lots of a product are introduced periodically based on the order list. Each lot may belong to one of three priority classes (normal, urgent, super urgent). Each operation has a processing time, a maximum batch size and an assigned machine family. The variety of tools includes batching and cascading machines, time coupling and machine dedication constraints as well as uncertain processing times and stochastic events. For a detailed description of the model, refer to (Kopp et al. 2020; Kovács et al. 2022).

3.4 Simulator

Our earlier work introduced PySCFabSim (Kovács et al. 2022), the core simulator of the RL environment.

The PySCFabSim tool is an event-based simulator built in pure Python on the basis of the SMT2020 dataset. The experimental results show that the key performance indicators of our software match those of a reference implementation computed in a commercial simulation solution. In contrast to commercial solutions, PySCFabSim is open source, allowing to access the internal state of the simulator to extract metrics required for more advanced decision making algorithms. Its extensibility with plugins makes data collection easy to implement. With the optimized performance, it is well-applicable for machine learning training, where millions of samples are required. It simulates two years of factory operation within 20 minutes on a desktop machine (including time taken by the FIFO dispatching policy), with about 40,000 completed lots and 10 million dispatching decisions for the period. The memory footprint is as low as 200

MBs and instantiating a simulator instance takes less than three seconds, allowing parallel execution of agents. To enable prototyping and experimenting with smaller-scale instances, an automatic downscaling algorithm was added to the simulator, generating new instances by proportionally reducing the number of machines and production lots. While our earlier work focused on introducing and validating the simulator itself, without detailed information on the specialized interfaces, this work aims to dive deeper into the details and usage of our RL framework.

3.5 RL Interface Implementation

We develop our RL interface in Python, as it enables direct interaction with the core simulator and makes extending the framework with new components practicable. Upon instantiation of an RL environment, the selected dataset is loaded into the memory. Then, a simulator instance is created for each episode of the RL problem after calling the environment's reset function. The simulator is extended with plugins required by the features used in the defined observation space or the reward function. A simulation run is halted upon reaching a decision point, followed by the invocation of the assigned agent to make a dispatching decision. Then, the decision is executed and the whole procedure is repeated until reaching a time limit, or completing all lots (end of the episode). The simulator's Weights & Biases (https://wandb.ai/) and chart plugins can be used for monitoring the trained agent's behavior within the RL framework.

By installing our package from `pypi` and importing it, a few demo environments are automatically registered with gym. To use custom environments the `DynamicSCFabSimEnv` class can be instantiated with the desired parameters.

An overview of our tool's architecture is presented in Figure 1, where Figure 1a illustrates multi-agent interaction with the environment. The agents communicate with a layer transforming the simulator's general interface to a standard gym interface by generating the observation space and reward from the simulator's inner state. In the reverse direction, the mid-layer is translating the agent's decision to a batch of lots and machine to start the lots on.

Figure 1b outlines the behavior of the interface layer. Using the general interface, the simulator asks for decision based on the machine and the lots waiting for the selected machine. Then, based on the machine assignments of the agents received during the construction of the environment, the candidate agent is selected to make the decision. The machine and the waiting lots can be directly forwarded to the dispatching heuristic which directly sends the dispatchable lots to the simulator. However, for the RL agent, an observation and a reward have to be generated. The action taken by the RL agent – either a selection of lot, a lot type or a dispatching strategy – is used to determine the next (batch of) lot(s) to be started on the machine. To avoid bias, the order of observations in the observation space is shuffled before passing it to the agent.

3.5.1 Agent Configuration

First, the user needs to define the agent configuration. The framework provides a general way of declaring both single- and multi-agent setups. When instantiating the environment, a list of agents must to be passed to the constructor. The agent can either be an instance of `RLAgent` or `GreedyAgent`, parametrized by a list of machines and a list of machine families that the agent should handle. For the greedy agent, the sorting criteria (either FIFO or critical ratio) can also be defined. A single machine or machine family should only be assigned to one agent. The machine families not assigned to any agent default to a greedy agent with the critical ratio sorting strategy. Based on the defined configuration, the simulation is executed until a decision needs to be taken by an RL agent. In that timepoint, the simulation is halted and the control is passed on to the RL agent with an observation and reward.

Figure 2 shows possible agent configurations on an example problem instance with three machine groups. It is possible to create a problem where the same agent controls all workstations and machines (top left), while it is possible to limit the RL agent's decision making to fewer machine groups (top right).

In a multi-agent case, several RL agents can interact with the same environment (bottom left). To compute a reference solution, the greedy agent can be used on all machines (bottom right).

3.5.2 Action Space

An RL agent dispatches lots by picking an action from a set of available options. Our framework provides four unique actions spaces, indicated in the upper left part of Figure 3, all representing an entirely different way of thinking. All action spaces are discrete, so that the agent's task is to select one action among a given number of alternatives.

The first option is selecting a heuristic to dispatch lots on a machine when it becomes available. Then, the selected heuristic ranks the lots in the queue, constructs batches if required and dispatches the lots with the highest importance based on the heuristic's priority rule. The available heuristics are FIFO, critical ratio, setup avoidance and most delay first. The set of rules, so the action space itself, can be extended with custom and combined rules when creating the environment. The heuristic selection is an action space with the size independent from the size of the problem itself, making scalability easier compared to other approaches. The observation space may contain information about the candidate machine and aggregate information over the waiting lots or operation types.

In the following cases, the size of the action space depends on the problem size (e.g., count of similar machines or operation types per machine), limiting scalability and adding an extra hyperparameter – the size of the action space – to be determined before training. Action masking is also required when there are fewer options available.

The next action space represents assigning a lot to a queue of a machine. When a lot's state changes to idle, the RL agent is called to select a (possibly utilized) machine. When the machine becomes available it selects the first lot from its queue according to a pre-defined priority rule (i.e., setup avoidance, or hot lot

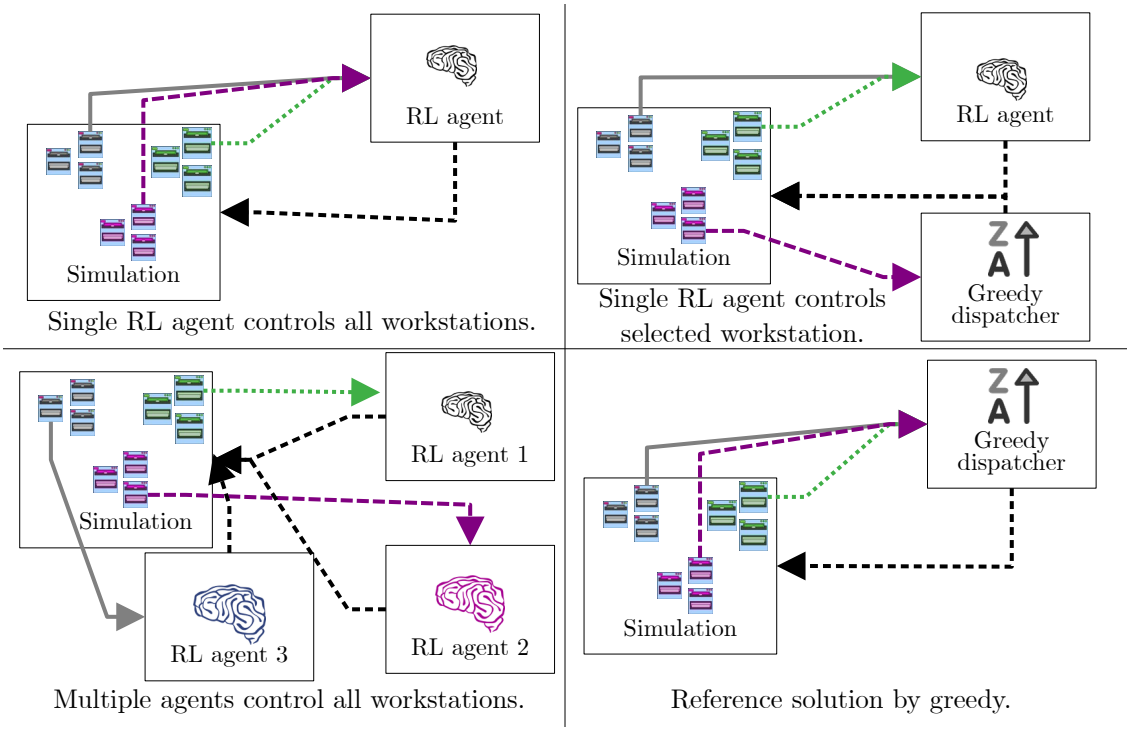


Figure 2: Interaction of the agents with the simulator for single- and multi-agent setups. Each machine family can be controlled by a different agent, but the same agent can also control multiple families. The families not controlled by RL default to a control by the critical ratio greedy strategy.

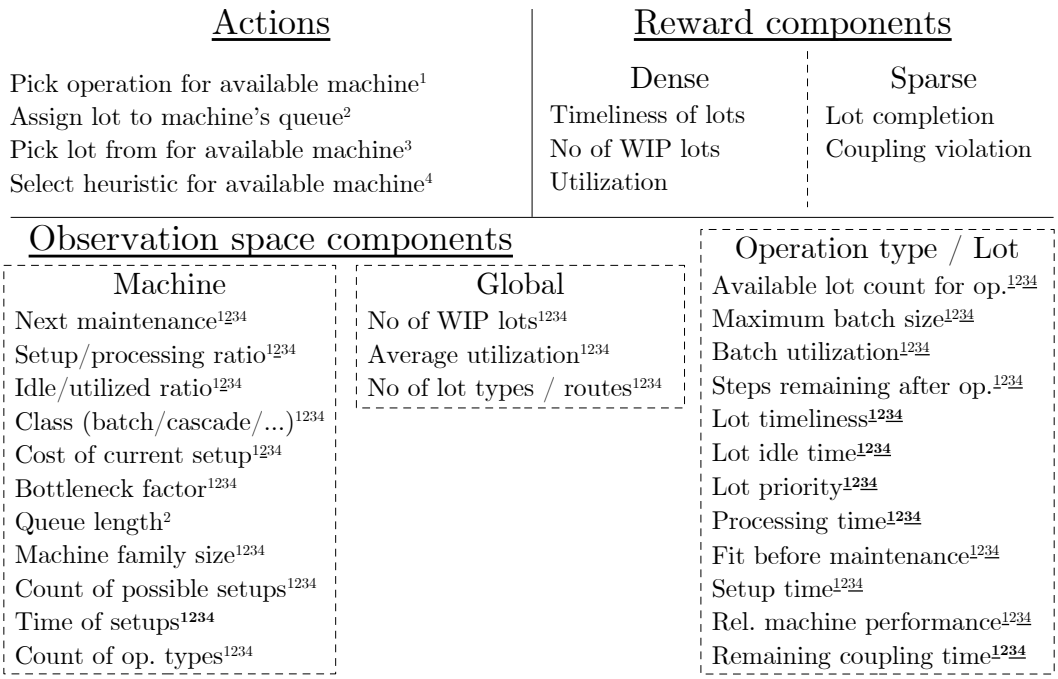


Figure 3: Pool actions, reward- and observation components. Observation space components denoted with numbers in upper index are only available for actions with the same notation. Underline denotes features present for each action, contrary to non-underlined features represented as a single scalar for the action.

first). The agent receives information about a single lot, with information about each selectable machine and its queue.

The remaining two action spaces allow for selecting a lot, or an operation type to dispatch on a machine when it becomes available. The observation space may contain information about the machine affected, and either the metrics of the lots waiting for the machine, or those aggregated by the operation types of the coming step. Depending on the problem instance, differentiating between lots with the same product (and route) may not be needed, e.g., in case of low mix problems. In such cases, it is preferable to use the strategy of selecting an operation type. Then, aggregate metrics are used for each operation type.

3.5.3 Observation Space

The observation space may contain global metrics describing the problem instance or state variables of the simulator that are independent of the actual decision (machine, available lots, etc). The global metrics can be used with any of the action spaces.

The local features are computed based on the actual decision, so their availability relies on the selected action space. For example, when selecting an operation type to start on a machine, only a single machine's properties are visible. However, when a machine has to be selected for a lot, the visible information includes multiple machines. Therefore, for each action space, a different set of observation features is available.

We provide a large set of pre-programmed features for each action type, as listed in Figure 3, making it possible to define environments declaratively, without the need of understanding the core simulator's codebase and writing imperative code. Adding custom features is also possible by creating and installing a simulator plugin that computes the required metric, and adding a (lambda) function to the observation space (line 20 of Figure 4) with the required parameters based on the selected action space.

```
1   from rl.env.action_choose_rule_for_machine import ChooseRuleForMachine
2   from rl.env.actions import SingleObservationFeature
3   from rl.env.agent import RLAgent, GreedyAgent
4   from rl.env.environment import DynamicSCFabSimEnv
5   from rl.env.reward import Reward
6   from simulation.plugins.wandb_plugin import WandBPlugin
7
8   R = Reward
9   O2 = ChooseRuleForMachine.Observation
10  P = GreedyAgent.Policy
11  DEMO_ENV_2 = lambda max_steps=100000000, max_days=730: DynamicSCFabSimEnv(
12      action=ChooseRuleForMachine(
13          alternatives=[P.CriticalRatio, P.FIFODeadline, P.FIFOWaiting, P.AvoidSetup, P.HotLotFirst, P.CombinedFIFO, ],
14          observation_space=[O2.Machine.cascading, O2.Machine.bottleneck_factor, O2.Machine.setup_last_at,
15                             O2.Machine.setup_last_cost, O2.Machine.idle_processing_ratio, O2.Machine.next_maintenance,
16                             SingleObservationFeature(lambda instance, **kwargs: len(instance.done_lots), True), ], ),
17      agents=[
18          RLAgent(idx=0, machine_groups=['Diffusion']), RLAgent(idx=1, machine_families=['DefMet_BE_33', 'DefMet_BE_42']),
19          GreedyAgent(policy=GreedyAgent.Policy.CombinedCR),
20      ],
21      simulator_params=dict(plugins=[WandBPlugin()], run_to=3600 * 24 * max_days, ), dataset='SMT2020_LVHM',
22      reward=5 * R.Dense.LotWipCount() + R.Dense.LotTimeliness() + 20 * R.Sparse.LotCompletion(), max_steps=max_steps, )
```

Figure 4: Definition of a multi-agent reinforcement learning environment. Two RL agents control a full machine group and two machine families, choosing a dispatching strategy from a pre-defined pool. The observation space provides information about the characteristics of the machine. The reward is a linear combination of multiple key performance indicators.

3.5.4 Reward Function

Similarly to the observation space, the reward function can be declaratively defined by combining predefined and custom reward functions, usually as a linear combination. The predefined reward functions in the upper right part of Figure 3 can be classified as sparse and dense. The sparse functions only give feedback for an action when reaching major goals several steps later, e.g., when a lot is completed. In contrast, the dense reward functions give feedback to the agent immediately, making the learning process more efficient. However, they are more difficult to design, compared to the sparse functions. The reward function can be customized by subclassing. Custom and pre-defined reward functions can be combined using operators, as shown on line 31 in Figure 4. For a more sophisticated reward function, a plugin can be installed on the simulator, listening to events influencing the implementable reward function and pre-computing features.

A sample declarative environment definition is presented in Figure 4. Using the documentation and the IDE's code completion, a new environment can easily be constructed from the pre-defined components, with the desired single- or multi-agent configuration.

4 EXPERIMENTAL EVALUATION

To demonstrate the applicability of our tool, we deploy an agent of *Stable-Baselines3* (Raffin et al. 2021). The library contains implementations of performant single-agent RL algorithms and uses the Gym interface. To train and evaluate the agent, two instances of the environment are constructed (Figure 5). The training environment is reset after every 270 days, since it usually takes about 9 months for the simulation to stabilize in terms of KPIs, while evaluation is performed for 730 days for better comparability.

The agent is trained for 1 million steps on a sample environment, limiting the agent's decisions to one machine family (*Diffusion*) and two machines (*DefMet_BE_33*, *DefMet_BE_42*). Due to the limited availability of multi-agent RL methods, we use a shared policy network for the agents. Comparing the performance of the trained RL agent with the critical-ratio (CR) heuristic, the results in Figure 6. show that CR outperforms the RL agent. The latter was trained using the default parameters without tuning. The relatively low performance of the agent can be explained by difficulties of connecting the effects of decisions with long-term sparse rewards. Additionally, due to the uncertainties related to machine breakdowns and rework, the environment only provides partial observability. In the future, we aim to develop agents specialized for the simulated problem, to improve the KPIs of the production process.

```python
from os import path
from simulation.gym.progress_callback import ProgressCallback
from simulation.gym.sample_envs import DEMO_ENV_2
from stable_baselines3 import PPO
def main():
    save_freq = 100000
    save_path, total_training_timesteps, eval_freq = 'test_file', 10 * save_freq, save_freq
    env, eval_env = DEMO_ENV_2(max_steps=100000, max_days=730), DEMO_ENV_2(max_days=270)
    model = PPO("MlpPolicy", env, verbose=1)
    checkpoint_callback = ProgressCallback(env, total_training_timesteps,
                                           save_freq=save_freq, save_path=save_path, name_prefix='checkpoint_')
    model.learn(
        total_timesteps=total_training_timesteps, eval_freq=eval_freq, eval_env=eval_env, n_eval_episodes=1,
        callback=checkpoint_callback,
        log_interval=100,
    )
    model.save(path.join(save_path, 'trained.weights'))
if __name__ == '__main__':
    main()
```

Figure 5: Training a single RL agent on a demo environment with the *Stable-Baselines3* library.

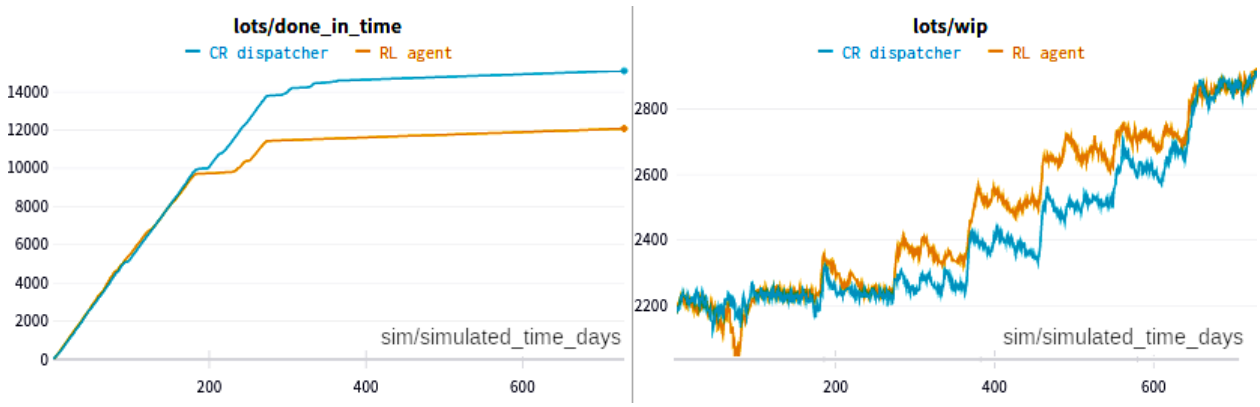


Figure 6: Performance of RL agent compared to CR agent: number of on-time lots and count of work-in-progress lots during the 2-year simulation period. Charts are generated by the tool's monitoring plugin.

5 CONCLUSION

This paper introduced a customizable RL environment integrated into a simulation of the semiconductor manufacturing process, intending to provide a standardized benchmark suite for RL agents concerned with large-scale scheduling problems. The environment can be used for benchmarking and evaluating the performance of different agents within a uniform framework. The importance of the observation space's features can be analyzed by retraining and evaluating them in environments with customized observation space and reward function. We demonstrated the usage of the framework with the definition of a demo environment and its integration with a widely adapted RL method.

Future Work In the next phase of the project, our goal is to develop new scalable, multi-agent RL techniques to control a real-world scale factory. We aim to implement this coming milestone using the RL framework presented in this work. To develop competitive RL-based methods, the reward functions and the observation space will require additional fine-tuning using the introduced plugin framework and algorithms. In cooperation with our industry partner Infineon Technologies, we plan to integrate the models of physical plants into our framework. New transfer learning experiments could be conducted to evaluate the agents' performance on datasets with different characteristics. Finally, agents validated on new datasets can be tested on higher-fidelity simulators used to analyze mature dispatching methods before real-world deployment.

ACKNOWLEDGMENT

This work was partially funded by KWF project 28472, cms electronics GmbH, FunderMax GmbH, Hirsch Armbänder GmbH, incubed IT GmbH, Infineon Technologies Austria AG, Isovolta AG, Kostwein Holding GmbH, and Privatstiftung Kärntner Sparkasse. Martin Gebser is also affiliated with Graz University of Technology, Rechbauerstraße 12, Graz 8010, AUSTRIA. The cliparts are from http://www.clker.com.

REFERENCES

Arulkumaran, K., M. P. Deisenroth, M. Brundage, and A. A. Bharath. 2017. "Deep Reinforcement Learning: A Brief Survey". *IEEE Signal Processing Magazine* 34(6):26–38.

Attinasi, M. G., R. De Stefani, E. Frohm, V. Gunnella, G. Koester, M. Tóth, and A. Melemenidis. 2021. "The Semiconductor Shortage and its Implication for Euro Area Trade, Production and Prices". *Economic Bulletin Boxes* 4:78–82.

Brockman, G., V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. 2016. "OpenAI Gym". http://arxiv.org/abs/1606.01540, accessed 10. June 2022.

Bureau, M., S. Dauzère-Pérès, and Y. Mati. 2006. "Scheduling Challenges and Approaches in Semiconductor Manufacturing". *IFAC Proceedings Volumes* 39(3):739–744.

Buşoniu, L., R. Babuška, and B. De Schutter. 2010. "Multi-agent Reinforcement Learning: An Overview". In *Innovations in Multi-Agent Systems and Applications - 1*, edited by D. Srinivasan and L. C. Jain, 183–221. Berlin, Germany: Springer Berlin Heidelberg.

Guo, W., W. Tian, Y. Ye, L. Xu, and K. Wu. 2021. "Cloud Resource Scheduling With Deep Reinforcement Learning and Imitation Learning". *IEEE Internet of Things Journal* 8(5):3576–3586.

Jacobs, T., F. Alesiani, and G. Ermis. 2021, Aug. "Reinforcement Learning for Route Optimization with Robustness Guarantees". In *Proceedings of the 30th International Joint Conference on Artificial Intelligence, IJCAI-21*, edited by Z.-H. Zhou, 2592–2598. Montreal, Canada: International Joint Conferences on Artificial Intelligence Organization.

Kaelbling, L. P., M. L. Littman, and A. W. Moore. 1996. "Reinforcement Learning: A Survey". *Journal of Artificial Intelligence Research* 4:237–285.

Kopp, D., M. Hassoun, A. Kalir, and L. Mönch. 2020. "SMT2020—A Semiconductor Manufacturing Testbed". *IEEE Transactions on Semiconductor Manufacturing* 33(4):522–531.

Kovács, B., P. Tassel, R. Ali, M. El-Kholany, M. Gebser, and G. Seidel. 2022, May. "A Customizable Simulator for Artificial Intelligence Research to Schedule Semiconductor Fabs". In *2022 33rd Annual SEMI Advanced Semiconductor Manufacturing Conference*, 1–6. Saratoga Springs, New York: SEMI.

Kuhnle, A., J.-P. Kaiser, F. Theiß, N. Stricker, and G. Lanza. 2021, Mar. "Designing an Adaptive Production Control System Using Reinforcement Learning". *Journal of Intelligent Manufacturing* 32(3):855–876.

Mirhoseini, A., A. Goldie, M. Yazgan, J. W. J. Jiang, E. M. Songhori, S. Wang, Y. Lee, E. Johnson, O. Pathak, S. Bae, A. Nazi, J. Pak, A. Tong, K. Srinivasa, W. Hang, E. Tuncer, A. Babu, Q. V. Le, J. Laudon, R. Ho, R. Carpenter, and J. Dean. 2020. "Chip Placement with Deep Reinforcement Learning". https://arxiv.org/abs/2004.10746, accessed 10. June 2022.

Mnih, V., K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller. 2013. "Playing Atari with Deep Reinforcement Learning". http://arxiv.org/abs/1312.5602, accessed June 2022.

Paeng, B., I.-B. Park, and J. Park. 2021. "Deep Reinforcement Learning for Minimizing Tardiness in Parallel Machine Scheduling With Sequence Dependent Family Setups". *IEEE Access* 9:101390–101401.

Park, I.-B., J. Huh, J. Kim, and J. Park. 2020. "A Reinforcement Learning Approach to Robust Scheduling of Semiconductor Manufacturing Facilities". *IEEE Transactions on Automation Science and Engineering* 17(3):1420–1431.

Pfund, M. E., S. J. Mason, and J. W. Fowler. 2006. "Semiconductor Manufacturing Scheduling and Dispatching". In *Handbook of Production Scheduling*, edited by J. W. Herrmann, 213–241. Boston, MA: Springer US.

Raffin, A., A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann. 2021. "Stable-Baselines3: Reliable Reinforcement Learning Implementations". *Journal of Machine Learning Research* 22(268):1.

Silver, D., A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. 2016. "Mastering the Game of Go with Deep Neural Networks and Tree Search". *Nature* 529(7587):484–489.

Suerich, D., and T. Young. 2020. "Reinforcement Learning for Efficient Scheduling in Complex Semiconductor Equipment". In *2020 31st Annual SEMI Advanced Semiconductor Manufacturing Conference*. May 6-9, Saratoga Springs, New York, 1–3.

Sutton, R. S., and A. G. Barto. 2018. *Reinforcement Learning: An Introduction*. Cambridge, MA: MIT press.

Taillard, E. 1993. "Benchmarks for Basic Scheduling Problems". *European Journal of Operational Research* 64(2):278–285.

Tassel, P., M. Gebser, and K. Schekotihin. "A Reinforcement Learning Environment For Job-Shop Scheduling". arXiv preprint arXiv:2104.03760 https://arxiv.org/abs/2104.03760, accessed Apr 8, 2021.

Waschneck, B., T. Altenmüller, T. Bauernhansl, and A. Kyek. 2016. "Production Scheduling in Complex Job Shops from an Industry 4.0 Perspective: A Review and Challenges in the Semiconductor Industry.". In *Proceedings of the 1st International Workshop on Science, Application and Methods in Industry 4.0*, edited by O. B. Roman Kern, Gerald Reiner, 1–12. Graz, Austria: CEUR Workshop Proceedings.

Waschneck, B., A. Reichstaller, L. Belzner, T. Altenmüller, T. Bauernhansl, A. Knapp, and A. Kyek. 2018, April. "Optimization of Global Production Scheduling with Deep Reinforcement Learning". *Procedia Collège International pour la Recherche en Productique* 72:1264–1269.

Zhang, K., Z. Yang, and T. Başar. 2021. "Multi-Agent Reinforcement Learning: A Selective Overview of Theories and Algorithms". In *Handbook of Reinforcement Learning and Control*, edited by K. G. Vamvoudakis, Y. Wan, F. L. Lewis, and D. Cansever, 321–384. Cham: Springer International Publishing.

AUTHOR BIOGRAPHIES

BENJAMIN KOVÁCS is a Ph.D. candidate in the Institute for Artificial Intelligence and Cybersecurity at the University of Klagenfurt, Austria. He focuses on optimization of production processes using simulation techniques and adaptive learning methods. His email address is benjamin.kovacs@aau.at.

PIERRE TASSEL is a Ph.D. candidate in the Institute for Artificial Intelligence and Cybersecurity at the University of Klagenfurt, Austria. His current research interests include reinforcement learning methods for combinatorial optimization problems and constraint programming. His email address is pierre.tassel@aau.at.

MARTIN GEBSER is professor for Production Systems at the University of Klagenfurt and Graz University of Technology. He received his PhD from the University of Potsdam in 2011, where he worked on theoretical and practical aspects of declarative problem solving methods. He works on applications of modern solving technology addresses, e.g., planning and scheduling, product configuration, and system design. His email address is martin.gebser@aau.at and his website can be found at https://ainf.aau.at/prosys/.

GEORG SEIDEL is Senior Manager of Infineon Technologies Austria AG (Villach, Austria). He has been involved in simulation, WIP flow management and Industrial Engineering topics since 2000. He was responsible for WIP flow management, especially for Lot dispatching at Infineon's site in Kulim (Malaysia) from 2012 until 2015. He is now responsible to rollout Fab Simulation at Infineon sites in Kulim, Regensburg and Villach. His email address is georg.seidel@infineon.com.