Proceedings of the 2021 Winter Simulation Conference
S. Kim, B. Feng, K. Smith, S. Masoud, Z. Zheng, C. Szabo, and M. Loper, eds.

FABRICATIO-RL: A REINFORCEMENT LEARNING SIMULATION FRAMEWORK FOR PRODUCTION SCHEDULING

Alexandru Rinciog

Department of Mechanical Engineering
TU Dortmund University
Leonhard-Euler-Straße 5
44227 Dortmund, GERMANY

Anne Meyer

Department of Mechanical Engineering
TU Dortmund University
Leonhard-Euler-Straße 5
44227 Dortmund, GERMANY

ABSTRACT

Production scheduling is the task of assigning job operations to processing resources such that a target goal is optimized. constraints on job structure and resource capabilities, including stochastic influences, e.g. job arrivals, define individual problems. Reinforcement learning (RL) solvers are adaptive and potentially robust in highly stochastic settings. However, benchmarking RL solutions for stochastic problems is challenging, requiring the simulation of complex production settings while guaranteeing reproducible stochasticity. No such simulation is currently available. To cover this gap, we introduce FabricatioRL, an RL compatible, customizable and extensible benchmarking simulation framework. Our contribution is twofold: We first derive requirements to ensure that generic production setups can be covered, the simulation framework can interface with both traditional approaches and RL, and experiments are reproducible. Then, we detail the FabricatioRL design and implementation satisfying the obtained requirements in terms of framework input, core simulation process, and the interface with different scheduling systems.

1 INTRODUCTION

Production scheduling is an NP hard problem increasingly tackled using reinforcement learning (RL), given the growing availability of data and computational resources. However, these methods are still in the concept phase and in urgent need of validation against the more established approaches such as a priori planning using exact solvers, meta-heuristics or simple priority rules. In particular, stochastic problems may benefit from RL application.

1.1 Motivation

RL approaches position themselves somewhere between (near-)optimal and simple heuristic solutions. As with (near-)optimal solutions, RL schedulers make decisions in the current state based on estimates of future conditions. RL schedulers are similar to simple heuristics in that they can react adaptively to changes in the production state (Waschneck et al. 2018). RL solutions may be preferable to exact planning and extensive search for three reasons. First, a priori planning assuming deterministic inputs may be thwarted by stochastic events occurring during production, such as resource availability issues or new job arrivals. Secondly, finding optimal or near-optimal solutions for the planning problem is computationally taxing for large production instances. Finally, such solutions require exact mathematical descriptions of the problem at hand, which are sometimes difficult to formulate for complex setups, e.g. (Rinciog et al. 2020). As opposed to simple priority rules, RL approaches could learn to leverage patterns in the scheduling problem. The priority rule approach disregards any structures that may be inherent to the given problem and is, by

978-1-6654-3311-2/21/$31.00 ©2021 IEEE

design, not optimal. On the flip side, priority rules still work well in uncertain environments and require no expensive computation.

Although studies have shown RL to be promising, there is a validation gap calling for a simulation framework enabling researchers to robustly embed RL methods within the state of the art. RL has been deployed for solving both deterministic and stochastic production scheduling problems with varying degrees of success. In standard deterministic setups, RL generally fails to outperform the state of the art (Gabel and Riedmiller 2012; Reyna et al. 2015; Arviv et al. 2016; Fonseca-Reyna et al. 2018; Méndez-Hernández et al. 2019; Zhang et al. 2020). For stochastic setups (Hofmann et al. 2020; Hu et al. 2020; Liu et al. 2020; Luo 2020; Kuhnle et al. 2020) and highly complex static setups (Zhang and Dietterich 1996; Rinciog et al. 2020) RL shows more promise, although here the experiments presented are difficult to reproduce and hence to validate because of the implementation overhead associated with writing a simulation and the lack of controlled stochasticity. As we focus on reproducible simulation, a comprehensive literature review for such approaches is beyond the scope of this paper. However, in (Rinciog and Meyer 2021b) we discuss current RL scheduling experiments on stochastic setups in more detail.

Furthermore, a well designed benchmarking framework would allow production scheduling researchers to fully explore the potential of already implemented RL libraries. Production scheduling literature mainly employs variations of Q-Learning or Deep Q Networks (DQN) as scheduling agents (Luo 2020; Park et al. 2019; Stricker et al. 2018; Shahrabi et al. 2017; Qu et al. 2015; Jiménez 2012; Aydin and Öztemel 2000). While popular and easy to implement, Q-Learning and DQN are by no means the only options. A simulation framework compatible with RL libraries such as KerasRL (Plappert 2016), Stable-Baselines (Hill et al. 2018), Horizon (Gauci et al. 2018) or Tensorforce (Kuhnle et al. 2017) would help increase the diversity of RL methods employed for scheduling.

1.2 Related Work

To benchmark solvers for deterministic scheduling problems, reporting the inputs and associated results is sufficient, since these problems are fully defined by their inputs, e.g. the job operation precedence and duration for job-shop scheduling problems (JSSP). Inputs for standard open/job/flow-shop problems are available through Beasley's OR library (Beasley 1990). Additionally, input collections can be extracted from dedicated publications, e.g. job-shops in (Demirkol et al. 1998) or flexible job-shops (FJSSP) in (Barnes and Chambers 1996)

Conversely, event discrete simulations are required to benchmark solvers for stochastic scheduling problems. Such simulations are additionally required for RL agent training and testing irrespective of the scheduling problem type. While there are many general simulation frameworks available which allow the modeling of generic material flows (e.g Simulink (MathWorks 2020), AnyLogic (Anylogic 2021), or PlantSimulation (Siemens 2021)), these are proprietary software systems, are not tailored to RL and do not have a mechanism allowing the simulation engineer to easily control stochasticity and guarantee experiment reproducibility.

The de facto standard for RL environments is provided by OpenAI Gym (Brockman et al. 2016). Herein, a general RL environment application programming interface (API) is defined. The Gym API is assumed by the aforementioned RL libraries, and has also found adoption in the OR and engineering community with projects such as ORGym (Hubbs et al. 2020) and SimPyRLFab (Kuhnle et al. 2020). ORGym provides simulations for varied combinatorial optimization problems from the field of Operations Research (OR), e.g. the Bin Packing or Traveling Salesman Problem. SimPyRLFab is the only available production simulation implementing the Gym API. However, SimPyFab is tailored to the semiconductor industry material flow, uses fixed heuristics for some of the decisions that might be taken by RL agents and does not guarantee reproducibility. Hence, for benchmarking a broader range of problem types, a more flexible simulation framework guaranteeing reproducibility is required.

1.3 This Work

By means of this study, we take steps towards alleviating three problems, namely (1) the validation gap for RL methods applied to (stochastic) production scheduling, (2) the reproducibility issues in stochastic setups and (3) the lack of employed RL method diversity. Our contribution is twofold:

1. We derive requirements for an RL benchmarking simulation framework for production scheduling problems in terms of scheduling setup, (RL) control interface and reproducibility (Section 2);
2. We show how these requirements can be satisfied by describing the design and implementation of our simulation framework, FabricatioRL, with a particular emphasis on its inputs, API, and core logic (Section 3).

In keeping with the scope of this study, we cannot provide a detailed account of all production scheduling problem, RL algorithm or framework implementation minutia. Instead, we will introduce the main ideas and provide examples from the RL production scheduling literature. To maintain a connecting thread we also do not elaborate on how the framework can be used with multi-agent RL solution approaches, although we note here, that this is entirely possible.

2 REQUIREMENTS

Production scheduling is the task of sequencing a number of operations forming distinct jobs onto processing resources so as to optimize a particular goal. Operations have individually defined durations, every job is associated with release and due dates and machines are limited to processing one operation at a time without preemption if not otherwise specified. Additional setup parameters pertaining to jobs (e.g. operation precedence constraints) and machines (e.g. limited resource buffer capacities) coupled with different optimization goals lead to distinct scheduling problems.

To solve any production scheduling problem with RL, a Markov Decision Process (MDP) defining the agent-environment interaction at discrete time-steps must be formulated. An MDP describes the state, action and reward sets, together with a state transition function and a reward signal. The agent senses the current environment state and takes an action, whereby the environment is moved to a new state, as defined by the state transition function. The reward function provides feedback for the state now reached.

2.1 Scheduling Setup: Generality, γ-Traceability, Extensibility

Pinedo describes scheduling problems in terms of machine setup α, additional setup parameters β and optimization goal γ in (Pinedo 2012). α defines problem categories based on operation precedence constraints within jobs, operation processing speed and number of machines capable of executing a particular type of operation. β values describe all other setup features, e.g. recirculation, setup times, machine capabilities, machine buffers, machine breakdowns or stochastic release dates. Fueled by the intricacies of real world production, the RL literature introduces β values not covered by Pinedo, e.g. transport times, limited number of transport vehicles (Kuhnle et al. 2020) or stochastic processing times (Jiménez 2012).

Setups defined through α and β are not independent of each other. Rather, setups with more generic constraints subsume setups with stricter constraints. Since such complex setups exist in both literature, e.g. (Jiménez 2012; Qu et al. 2015; Kuhnle et al. 2020; Luo 2020; Hofmann et al. 2020), and practice, we require the benchmarking framework to cover a large number of typical scheduling setups, or, stated differently, to be as general as possible.

We deem the following setup to be sufficiently general, since, without loss of generality, it subsumes most of the setups in (Pinedo 2012) and RL literature: The job operation sequence is constrained by a (directed acyclic) precedence graph and operations can be executed on one ore more available machines. The number of operations in every graph can differ, and particular operations can occur more than once within the same job. The processing speed is machine dependent and is used to scale the duration associated

with every operation. Every operation has an associated tool set, and the tool switching times are sequence dependent. Transport times are modeled explicitly but it is assumed that sufficient transport resources are available so as to immediately start every transport task. Buffers of a certain capacity (including 0 and ∞) are placed before each processing resource. Processing resource breakdowns, job arrivals and operation processing times are stochastic. For a better readability, we refer to this scheduling setup as generalized flexible job shop (GenFJS).

γ defines either job- or resource-centric optimization goals. The most frequent job-centric goals are aggregates (e.g. maximum, average, minimum) of job completion times, flow times, throughput times, lateness, tardiness, earliness and job idle times. Resource-centric goals aggregate resource utilization, number of operations in buffers, buffered processing times, incurred setup times, machine failures and inventory levels. The simulation should track all of these intermediary variables at every time-step, such that the desired target, e.g. minimizing the maximum completion time (makespan), can easily be measured by applying a corresponding aggregation function. To this we refer as γ-traceability.

The scheduling setups considered in RL literature show just how varied production is. Because of this, the simulation framework should be constructed in a modular fashion, so as to allow extensibility.

2.2 RL: Configurability, Extended Gym Compatibility, Runtime Efficiency

The most important event to a production scheduling MDP is that of a resource finishing its current task. For GenFJS, two questions arise on such an occurrence: "What operation should be processed next on the resource just freed?"(i), "To which downstream resource should the job just processed be sent?"(ii).

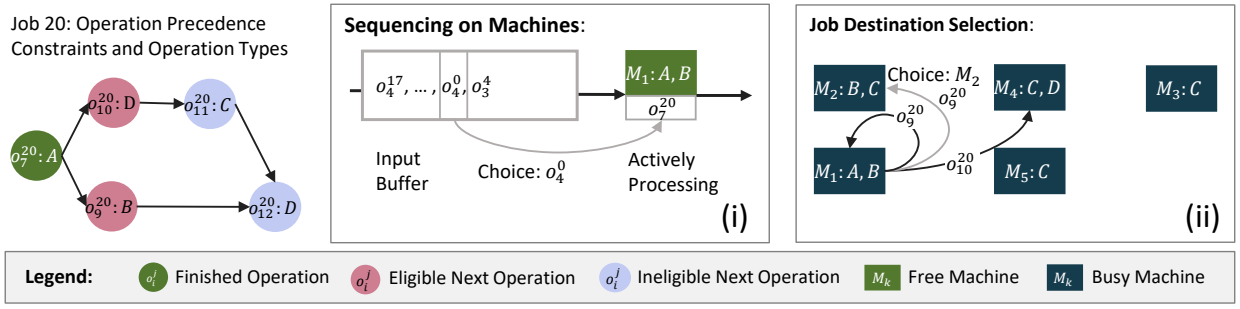

Figure 1: The two decision types in a production setup with operation precedence graphs (top left). Operation and machine types are indicated with capital letters. In (i) and (ii), orange arrows model decisions made, and blue arrows represent alternatives.

The (i + ii) scheduling decomposition leads to the process flow depicted in Figure 1: Whenever the operation of a job is finished on a machine, an RL agent selects the next operation for this machine from the input buffer (i). Then, the agent selects the transport destination (ii). The possible destinations for the job just processed depend on the next eligible operations and the machines capable of processing them. The operation feasibility can be derived from the precedence graph depicted in the top left corner. Depending on the particular scheduling problem and its assumptions, not all decision types will be encountered. For standard job-shop scheduling problems, for instance, only decisions of type (i) will be required.

Action Space Configurability: There are two main approaches to encoding the actions described above. When faced with a decision, an agent can either select an action directly or indirectly. Indirect actions are chosen by selecting an optimizer from a fixed set, which then determines the next direct action. Direct action for (i) comes in the form of an operation index, i.e. the tuple (job number, operation number), from the set of all available operations in production. For (ii) the action is given by a machine index. Additionally, the action can be deferred by outputting a wait signal. Examples of direct decisions can be found in (Jiménez 2012; Qu et al. 2015; Kuhnle et al. 2020), while indirect decisions are covered by

(Aydin and Öztemel 2000; Luo 2020), for instance. Optimizers are often, but not always, e.g. (Shahrabi et al. 2017), simple priority rules.

For the simulation framework, these alternative approaches lead to the action space configurability requirement. The simulation should allow both direct and indirect actions. Additionally, it should be configurable whether the RL agent is responsible for all decision types, or just a subset, deferring action to fixed optimizers for the rest. The optimizer sets for indirect actions and RL complementary action should be easily customizable and extensible.

State Space Configurability: The information needed for state transition depends on the production setup considered: The state tracks the jobs currently in production together with the operations still in need of processing, their remaining processing time, position (machine index) and status (in transport, waiting for processing, processing). Additional fixed information such as transport times, tool switching times, precedence constraints, machine capabilities, buffer capacities, operation types and tool sets should be available, such that the setup constraints can be enforced by the simulation and learned by the agent.

In RL, a distinction is made between agent state and environment state. The latter contains all the information required for the environment to implement its logic, while the former is an agent view of the latter. For production scheduling, the agent state often only contains a subset of the information available in the environment state. This is because the environment state space may be too large for the agent algorithm to handle. Furthermore, simulated stochasticity is transparent to the environment, but should not be transparent to the agent. Lastly, it may be useful to have agents make good decisions based solely on environment state features that are independent of problem size. This would improve the agent's transferablility and generalization qualities.

The exact information comprising the agent state needs to be established through experimentation. On the one hand, raw environment state information as listed above, can be used. On the other hand, environment state information can be condensed into features. Features fall into three categories, namely job features, resource features or target features. The first category aggregates job information, e.g. remaining job processing time or remaining job operations. The second aggregates machine or vehicle related information, e.g. remaining processing time in machine buffers or number of operations queued for processing or transport. Information in these categories can be stored per job/machine or aggregated further into single scalars using centrality (e.g. mean, median) and variance measures (e.g. standard deviation, gini). The last feature category contains optimization goal related variables such as estimated total tardiness (Wang and Usher 2005; Luo 2020) or average machine utilization (Thomas et al. 2018; Luo 2020).

The simulation framework should allow the selection of the agent state components and allow extensions thereof. These can be either raw state information such as the current operation duration matrix or operation position, as well as different features including goal oriented metrics. Moreover, it should be made possible to accommodate user defined state features.

Reward Configurability: There is no generally accepted scheme for reward design, which means that appropriate signals have to be found through experimentation. Since RL agents try to maximize future reward, it stands to reason that, for production scheduling, the reward is often a function of the optimization goal or a goal-related intermediary variable, e.g. (Qu et al. 2015; Wang 2020; Luo 2020; Kuhnle et al. 2020). Important choices in reward design include the time points at which the reward is returned (at every step, every k steps for some k or at the end of the game), whether the reward is continuous or discrete, strictly positive, strictly negative or both, bounded or unbounded (Sutton and Barto 2018).

Extended Gym Compatibility: In terms of the agent interface, the simulation should respect the OpenAI Gym standard, such that external agent libraries such as keras-rl can be used. This allows the application of different pre-implemented RL agents, whether they be policy- or value-based or actor-critic systems, to be trained and tested within the environment in a convenient fashion. Additionally, the simulation should address two supplementary RL techniques not currently covered by the gym standard, namely illegal action masking and offering an environment clone for model-based RL approaches such as AlphaZero (Silver et al. 2017). To construct action masks, the environment has to provide the agent with a list of

legal actions at every step. Environment clones can be used by agents to directly see the effects of one's actions a few steps ahead.

Runtime Efficiency: A simulation requirement that does not follow directly from the details laid down until now is that of simulation performance in terms of runtime. RL is sample-inefficient (Haarnoja et al. 2018), which implies that many simulation runs will be required for the agent(s) to converge.

2.3 Reproducibility: Reproducible Stochasticity, Input Separation, Planning Compatibility

While reproducibility requirements are not bound to the domain of stochastic scheduling, in RL literature for scheduling these are often neglected. We identify four aspects as essential for ensuring reproducibility.

First, the stochasticity embedded in the scheduling process should be exactly reproducible. To that end, reporting the distributions used to simulate stochastic influences is not sufficient. Consider, for instance, a scheduling problem with 10 jobs and 5 machines. Let the job operation sequence be defined by a random permutation of numbers 1 to 5. This leads to $5!^{10}$ possible scheduling instances, which makes the reliable comparison of approaches based on randomly sampled instances virtually impossible if no variance control scheme is implemented. For (online) stochastic problems, the general literature approach is to sample stochastic events on demand during the simulation.

To control stochasticity, the common random numbers (CRN) (Glasserman and Yao 1992) approach can be used. This means that (a) the sequence of random numbers required during the simulation is a function of a single independent variable and (b) the meaning of drawn random numbers is the same between simulations. We refer to the requirement of implementing CRM as reproducible stochasticity.

Secondly, the simulation inputs, including the instantiation of online events, need to be clearly separated from control algorithms. This is needed to allow alternative implementations to run experiments on the exact problem on which results are reported on. We refer to this requirement as input separation. A simulation implementing this requirement has the added benefit of being backwards compatible, in that scheduling problem inputs from preexisting works and benchmark sets, e.g. OR library, can be used to define the simulation setup.

Lastly, the simulation should allow following an externally computed production plan, where the operation start times are listed for every resource. Such an execution plan is often the output of "standard" scheduling approaches. While such an a priori plan may be affected by stochastic events, it may still yield better results than RL in many situations. The exact situations where RL outperforms such "classic" OR approaches have yet to be established. Our simulation framework should enable researchers to cover this gap, which leads to the planning compatibility requirement.

3 FABRICATIO-RL: ARCHITECTURE AND IMPLEMENTATION

Figure 2 shows the main components of the layered architecture of FabricatioRL following the Gym API. The Gym API consists of three core methods namely `init`, `step` and `reset`. `init` is used for parameterizing and instantiating the simulation. `step` takes an action as an argument and returns a state, the reward for the particular state, a flag signaling whether the simulation has ended and a dictionary with debugging information. `reset` is called to move the simulation back to its starting state. The `render` method is additionally defined by Gym to create a visual representation of a state.

FabricatioRL separates the main simulation functionality located in `SchedulingEnvironmentCore` module from the `SchedulingEnvironment` wrapper, which implements the `Gym` interface. The layer components are highlighted in green. The yellow classes pertain to the definition and storage of simulation inputs. The blue and gray highlighted classes represent the two main data structures of the environment. We detail the architecture and outline its functions in terms of input handling, API customization and core functionality.

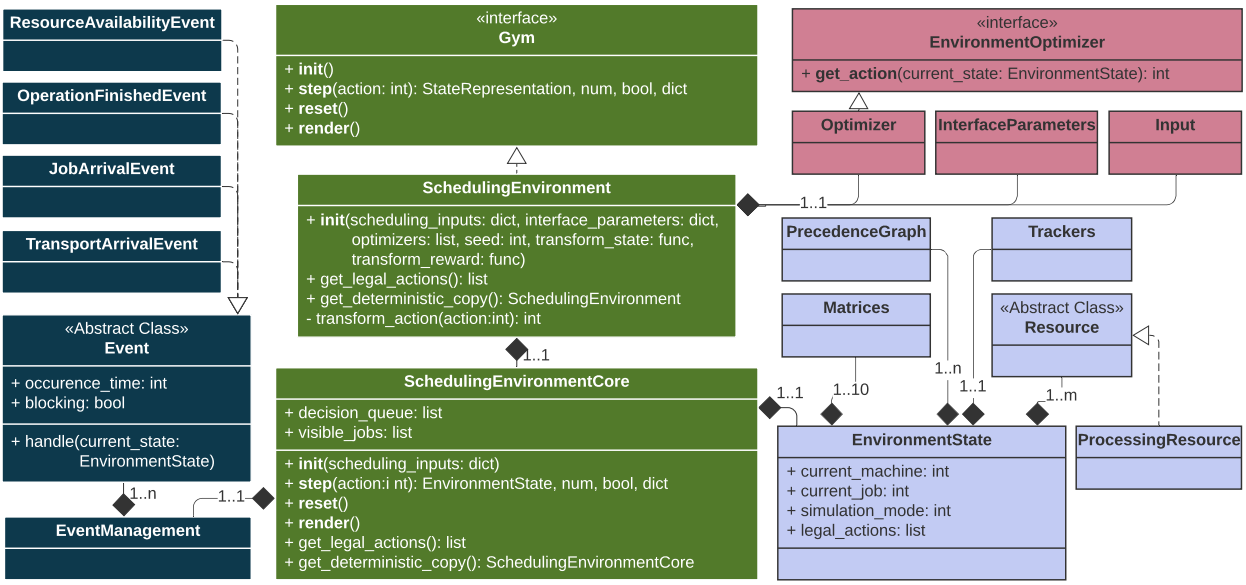

Figure 2: Class diagram of the environment architecture

3.1 Inputs

Let n, m, o, l, t be the number of jobs, number of machines, maximum number of operations per job, number of system tool sets and number of operation types respectively. The inputs for GenFJS are given by the operation precedence graphs $O^P \in \{0,1\}^{n\times o\times o}$, the operation type matrix $O^{Ty} \in \{1..t\}^{n\times o}$, the operation duration matrix $O^D \in \mathbb{N}_+^{n\times o}$, the operation tool set $O^{Tl} \in \{1..l\}^{n\times o}$, the machine speed vector $M^S \in \mathbb{R}^m$, the machine distance matrix $M^{Tr} \in \mathbb{N}_+^{m\times m}$, the tool switching time matrix $M^{Tl} \in \mathbb{N}_+^{l\times l}$, the processing resource buffer size vector $M^{Bf} \in \mathbb{N}_+^m$, machine capability matrix $M^{Ty} \in \{0,1\}^{m\times t}$, the maximum number of failures per machine f, the distributions for the time between failure ϕ, the operation duration noise δ and the job inter-arrival time ι.

The simulation user can choose to pass all the listed matrices and vectors directly to the SchedulingEnvironment.init function. Alternatively, the different input dimensions together with a corresponding sampling function can be provided. init initializes the Input object, where the sampling of the listed matrices and vectors is done if necessary. Additionally, the operation duration noise $O^{D'} \in (0,2)^{n\times o}$, job release times $R \in \mathbb{R}_+^n$ and machine breakdown times $B \in \mathbb{R}_+^{m\times f}$ get sampled in accordance with δ, ϕ and ι respectively. B and R are used to create the stochastic events to be added to the event queue in SchedulingEnvironmentCore during its instantiation.

To ensure **reproducible stochasticity**, random number generator (RNG) seeds are used. All sampling happens after seeding but before the beginning of the simulation execution. In doing so, running the simulation on a particular input set with the corresponding seed would always yield the same stochastic events irrespective of the scheduling method employed. Thus, a dataset of input seed pairs could be used for the validation step, analogous to the benchmarks of supervised learning.

In EnvironmentState.Matrices, $(O^P, O^{Ty}, O^D, O^{Tl}, M^S, M^{Tr}, M^{Tl}, M^{Bf}, M^{Ty}, O^{D'})$ define the initial values. These are used throughout the simulation to determine the next processing steps and their duration. For example, the processing duration d of operation i of job j on the machine k having a current tool set of p can be calculated as $d = M^{Tl}_{pO^{Tl}_{ji}} + O^D_{ji} \cdot M^S_k \cdot O^{D'}_{ji}$.

A copy of the input matrices together with $O^{D'}$, R and B is stored in the Input object. This information is used by the reset function to recreate the initial environment state by calling init with the stored parameters. More importantly, the Input module clearly separates the scheduling problem instance from the rest of the simulation. Logging these inputs leads to the implementation of the **input separation** requirement.

3.2 Interface Module

Using the InterfaceParameters object, the decisions that the RL algorithm covers can be configured. RL agents can, for instance, focus solely on machine sequencing decisions, deferring the job destination selection to Optimizers objects passed to the SchedulingEnvironment constructor. Such is the approach in (Kuhnle et al. 2020), where agents solely take routing decisions, with all sequencing decisions being done by a hard-wired optimizer, namely the First-In-First-Out heuristic. The architecture presented here is more flexible, with EnvironmentOptimizer objects being any object exposing the get_action method. Said method is offered a read-only EnvironmentState object and is expected to return a direct action adequate for the decision type.

Allowing the use of external optimizers for different decision types also fulfills the action space configurability requirement. In such a design, the RL agents select their preferred algorithm based on the current state. For this, the external step method is called with an optimizer from a fixed set associated with that particular decision type. Within step, transform_action is called, which in turn uses the corresponding optimizer from the Optimizer set to make a direct decision (either a machine or operation index) based on the EnvironmentState of the SchedulingEnvironmentCore object.

Both the state and the reward are configurable by passing two transformation functions to init which are stored in InterfaceParameters. The transformation functions select and transform information from the EnvironmentState object to create the desired state and reward representations respectively. The saved transformation functions are called just before step returns to present the agent with the desired representations. The transformation functions both take an EnvironmentState object as a parameter. As such, all the required raw information, including tracking variables, is transparent, whereby full configurability is enabled.

3.3 Core Module

The SchedulingEnvironmentCore class contains the simulation logic expressed through the task of managing the event queue and state objects. Correspondingly, the main attributes of the class are the state of type EnvironmentState and the event management object, EventManagement. Additionally, two attributes, namely the pending_decisions queue and the visible_jobs list are defined. The former tracks the machines for which decisions are needed at the current time. The latter is necessary since the environment knows the exact job structure of future jobs (see Section 3.1), which needs to be hidden from the agent.

The environment state separates raw information that can be directly offered to an agent (Matrices and Trackers) from inner structures (PrecedenceGraph and Resources) used to advance the state in a computationally efficient fashion. Additionally, the current legal_actions, current_machine and simulation_mode information is maintained. simulation_mode indicates how the next action should be interpreted, i.e. either as a sequencing or destination selection action, current_machine is the machine index for which the mode dependent decision is taken (see Figure 1). legal_actions are, as the name indicate, simply a list of valid actions for the current state.

The Matrices object contains the tensors, matrices and vectors described in Section 3.1. As operations are completed, the corresponding matrix entries are zeroed out, thus tracking the state progression. Additionally, two matrices L and S of size $n \times o$ tracking the last processing resource of operations and their status (queued, processing, in transport) are maintained. The Trackers class maintains intermediary variables used for target (γ) computation including the current system time, thus guaranteeing **traceability**.

The PrecedenceGraph is used together with the list of Resource objects to quickly infer legal actions, and create state representations that are more easily understandable by humans. PrecedenceGraph is a graph representation of O^P through which next job operations can be extracted in $O(1)$. The elements of Resource are objects modeling transport and processing resources including input and output buffers with their contents.

The event queue contained by the EventManagement module consists of any number of self handling events sorted ascendingly by their occurrence time. Queued events all implement the Event interface and fall into one of four categories, namely MachineAvailability, OperationFinished, JobArrival and TransportArrival. Whenever an event is triggered, the handle method is called to modify the state accordingly. If the event is non-blocking, the next event in the queue will be triggered upon the return of the first event's handle method. For instance, When jobs arrive, the corresponding entry is added to visible_jobs and the tracker variables are updated. Since new job arrivals do not impact the resource availability directly, the next event triggers. Conversely, when an OperationFinishedEvent occurs, the event processing is halted upon marking the corresponding processing resource as free, so as to allow the agent to make the next decision.

The SchedulingEnvironmentCore methods mirror those of the Gym interface, although the class does not implement it directly, so as to keep the composition relation between SchedulingEnvironment and SchedulingEnvironmentCore clean. As opposed to the outer layer (SchedulingEnvironment), the inner object (SchedulingEnvironmentCore) is not configurable. Here, step only understands direct actions, i.e. operation indices for sequencing and resource indices for job destination selection decisions.

At the heart of the simulation is the step method. Depending on the simulation mode, step interprets the action differently and correspondingly updates the simulation state. Before returning, the method creates and queues the event resulting from the decision, triggers and handles the next events, computes the next legal actions, queues the next required decisions and changes the mode, if necessary. Queued events only trigger if the decision queue is empty. The only events that are created as a result of an agent's action are of type OperationFinishedEvent and TransportArrivalEvent. The former is created when the simulation is in the sequencing mode ((i) in Section 2.2), and reflects the agent's choice of operation to start processing on the machine indicated by the current_resource variable in the EnvironmentState. The latter event ((ii) in Section 2.2) is created in job destination selection mode and reflects the agent's dual choice of next job operation and machine that is to process the chosen operation.

The mechanics of the core module described here enables an **efficient runtime**, which depends mainly on the event queue and state update procedure. During simulation execution, events are created dynamically depending on agent decisions and stochastic influences, and added to a heap-queue. The events are handled in order of their occurrence depending on their type. With a careful implementation, the event handling runtime can be constant, $O(1)$, since the exact positions in the EnvironmentState that need to be updated on occurrence can be saved on creation, e.g. the index of the operation and the machine it is processing on, for OperationFinishedEvents. Let n be the total number of operations to be scheduled during a simulation run. Every operation first needs to be transported then processed. That means that there are $O(2n)$ events to be processed. Using a priority queue with $O(\log(n))$ insertion time and $O(1)$ head retrieval time, leads to an asymptotic runtime of $O(2n\log(n))$.

The **planning method compatibility** requirement is enabled by the core module allowing agents to defer actions by means of a wait signal at the first cycle through the decision_queue. Without the wait signal, it is not possible to iteratively build - and hence simulate - all possible schedules.

4 API USAGE EXAMPLES

A Python implementation of the architecture presented here can be found in (Rinciog and Meyer 2021a). We chose Python since this integrates seamlessly with both Gym and the RL libraries listed in the introduction. The repository additionally contains an initial set of tests and examples of the simulation API configuration. The current tests demonstrate correct seeding functionality. Furthermore, we used ORTools to generate optimal schedules on JSSP benchmark instances, run them through the simulation, and show that the results reported by the simulation match. In what follows, we briefly describe two of the usage examples, namely a simple heuristic control and a KerasRL DQN Agent control, in an effort to familiarize the reader with the API customization process.

Simple heuristic: Running the simulation with a simple heuristic, e.g. least processing time (LPT) (Benoit, Canon, Elghazi, and Heam 2021) is done in three steps.

(1) A ReturnTransformer object needs to be implemented. LPT only operates on processing times for the buffered operations and does not consider the return signal. As such, the transform_reward function can simply return None. The transform_state function should return the legal_actions list and the operation duration matrix from the Matrices object. To be able to compute the values of different optimization goals at the end of the simulation, the Trackers object should also be returned.

(2) The setup parameters need to be defined. Since standard JSSPs are fully described by the operation type and operation duration matrix, the other simulation parameters retain their default values.

(3) The simulation step function is called on a loop with an action indicated by a select_action function. The latter takes the legal_actions and operation duration matrix information and simply selects the operation from legal_actions with the least value in the the duration matrix. The desired optimization metrics can be computed from the Trackers object of the last state returned by step.

KerasRL Agent: The second scenario is a DQN training with KerasRL on randomly generated FJSSPs with graph precedence constraints. To showcase interface customization, we use indirect heuristic actions, e.g. (Luo 2020), raw state information and average machine utilization as a reward. We show how to train and test the agent using seeds. This can be done in six steps.

(1) The Optimizer objects for machine sequencing and job destination selection need to be implemented. Assuming these are simple priority rules, this boils down to implementing the get_action method within an Optimizer object. Said method can be implemented analogously to select_action in the LPT example above, with the distinction that the full state structure is now transparent to the method.

(2) The ReturnTransformer object needs to be implemented. The transform_state method takes $O^D, O^P, O^T, M^{Tr}, M^{Ty}$, together with the current machine number, current job number and the simulation mode from Matrices, flattens the multidimensional information and returns it. The transform_reward method returns the average of all the machine utilization tracker values.

(3) The setup parameters including the optimizer lists and ReturnTransformer object are used to instantiate the environment.

(4) The DQN Agent's neural network (NN) architecture, is defined using Keras. The NN in- and output dimensions are obtained from the observation_space and action_space environment attributes. The NN is then passed to the constructor of the DQNAgent implemented in KerasRL.

(5) The agent is trained by calling its fit method with the environment as a parameter and a number of decisions to execute before training completes. To train on a specific group of inputs, the set_seeds method should be called on the environment before the call to fit. The environment will cyclically use these seeds when re-initializing the environment on the reset background calls by the agent.

(6) test can be called on the agent with the environment and number of episodes as parameters. A different seed set can be set for testing.

5 CONCLUSION

This work is motivated by the increasing interest in RL solutions for production scheduling problems and the validation gap associated with them. We took steps towards covering this gap by first deriving requirements for an RL benchmarking simulation framework and then providing the description of an implementation able to satisfy them. In terms of setup, the simulation framework should be general, extensible and γ-traceable. With respect to RL control, the framework should allow MDP configuration, be gym compatible and be asymptotically efficient in terms of runtime. With respect to experiment reproduciblity, the framework should enable the exact reproduction of stochasticity, clearly separate inputs from control, be compatible with planning methods, and run on traditional inputs from the literature.

In a next step we have shown how these requirements can be fulfilled by using a layered architecture implementing Gym. The inputs reflected by the state and separated by means of a dedicated class, allow for the coverage of many production setups. The simulation logic is centered around self-handling events

and a specialised state structure allowing for an efficient runtime. The key to RL configurability is creating an interface for externally implemented objects to affect the simulation in terms of action interpretation and state and reward representation. By using RNG seeding and sampling everything before the main simulation loop, stochasticity is made stochasticity exactly reproducible, while simultaneously providing flexible sampling functionality for the generating simulation inputs. We provided a preliminary version of the code at (Rinciog and Meyer 2021a) where some example usages can be found.

While this work is a decisive step in the right direction for the task of validating RL approaches for production scheduling, much remains to be done in terms of framework validation, extension and actual RL benchmarking. Thorough testing of the provided framework is necessary for simulation validation. Additionally, a systematic review of scheduling literature is required to reveal and prioritize further setup parameters to be implemented. In terms of RL benchmarking, both exact/search approaches and simple priority rules should be used as baselines for diverse stochastic setups in an effort to infer the exact situations where these individual approaches work best. Last but not least, the framework could be extended to function as a digital twin, thereby enabling the exploitation of the increasingly available real world production data.

REFERENCES

Anylogic 2021. "AnyLogic Documentation". *The AnyLogic Company*. https://help.anylogic.com/index.jsp.

Arviv, K., H. Stern, and Y. Edan. 2016. "Collaborative reinforcement learning for a two-robot job transfer flow-shop scheduling problem". *International Journal of Production Research* 54(4):1196–1209.

Aydin, M. E., and E. Öztemel. 2000. "Dynamic job-shop scheduling using reinforcement learning agents". *Robotics and Autonomous Systems* 33(2-3):169–178.

Barnes, J., and J. Chambers. 1996. "Flexible job shop scheduling by tabu search". *Graduate Program in Operations and Industrial Engineering, The University of Texas at Austin, Technical Report Series, ORP96-09*.

Beasley, J. E. 1990. "OR-Library: distributing test problems by electronic mail". *Journal of the operational research society* 41(11):1069–1072.

Benoit, A., L.-C. Canon, R. Elghazi, and P.-C. Heam. 2021. *Update on the Asymptotic Optimality of LPT*. Ph. D. thesis, Inria Grenoble-Rhône-Alpes.

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

Demirkol, E., S. Mehta, and R. Uzsoy. 1998. "Benchmarks for shop scheduling problems". *European Journal of Operational Research* 109(1):137–141.

Fonseca-Reyna, Y. C., Y. Martínez-Jiménez, and A. Nowé. 2018. "Q-learning algorithm performance for m-machine, n-jobs flow shop scheduling problems to minimize makespan". *Investigación Operacional* 38(3):281–290.

Gabel, T., and M. Riedmiller. 2012. "Distributed policy search reinforcement learning for job-shop scheduling tasks". *International Journal of production research* 50(1):41–61.

Gauci, J., E. Conti, Y. Liang, K. Virochsiri, Z. Chen, Y. He, Z. Kaden, V. Narayanan, and X. Ye. 2018. "Horizon: Facebook's Open Source Applied Reinforcement Learning Platform". *arXiv preprint arXiv:1811.00260*.

Glasserman, P., and D. D. Yao. 1992. "Some guidelines and guarantees for common random numbers". *Management Science* 38(6):884–908.

Haarnoja, T., A. Zhou, P. Abbeel, and S. Levine. 2018. "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor". In *International Conference on Machine Learning*, 1861–1870. PMLR.

Hill, A., A. Raffin, M. Ernestus, A. Gleave, A. Kanervisto, R. Traore, P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu. 2018. "Stable Baselines". *GitHub repository*. https://github.com/hill-a/stable-baselines.

Hofmann, C., C. Krahe, N. Stricker, and G. Lanza. 2020. "Autonomous production control for matrix production based on deep Q-learning". *Procedia CIRP* 88:25–30.

Hu, L., Z. Liu, W. Hu, Y. Wang, J. Tan, and F. Wu. 2020. "Petri-net-based dynamic scheduling of flexible manufacturing system via deep reinforcement learning with graph convolutional network". *Journal of Manufacturing Systems* 55:1–14.

Hubbs, C. D., H. D. Perez, O. Sarwar, N. V. Sahinidis, I. E. Grossmann, and J. M. Wassick. 2020. "OR-Gym: A Reinforcement Learning Library for Operations Research Problem". *arXiv preprint arXiv:2008.06319*.

Jiménez, Y. M. 2012. "A generic multi-agent reinforcement learning approach for scheduling problems". *PhD, Vrije Universiteit Brussel*:128.

Kuhnle, A., J.-P. Kaiser, F. Theiß, N. Stricker, and G. Lanza. 2020. "Designing an adaptive production control system using reinforcement learning". *Journal of Intelligent Manufacturing*:1–22.

Kuhnle, A., M. Schaarschmidt, and K. Fricke. 2017. "Tensorforce: a TensorFlow library for applied reinforcement learning". *GitHub repository*. https://github.com/tensorforce/tensorforce.

Liu, C.-L., C.-C. Chang, and C.-J. Tseng. 2020. "Actor-Critic Deep Reinforcement Learning for Solving Job Shop Scheduling Problems". *IEEE Access* 8:71752–71762.

Luo, S. 2020. "Dynamic scheduling for flexible job shop with new job insertions by deep reinforcement learning". *Applied Soft Computing*:106208.

MathWorks 2020. "Simulation and Model-Based Design". *MathWorks*. https://www.mathworks.com/products/simulink.html.

Méndez-Hernández, B. M., E. D. Rodríguez-Bazan, Y. Martinez-Jimenez, P. Libin, and A. Nowé. 2019. "A Multi-objective Reinforcement Learning Algorithm for JSSP". In *International Conference on Artificial Neural Networks*, 567–584. Springer.

Park, I.-B., J. Huh, J. Kim, and J. Park. 2019. "A Reinforcement Learning Approach to Robust Scheduling of Semiconductor Manufacturing Facilities". *IEEE Transactions on Automation Science and Engineering*.

Pinedo, M. 2012. *Scheduling*, Volume 29. Springer.

Plappert, M. 2016. "keras-rl". *GitHub repository*. https://github.com/keras-rl/keras-rl.

Qu, S., T. Chu, J. Wang, J. Leckie, and W. Jian. 2015. "A centralized reinforcement learning approach for proactive scheduling in manufacturing". In *2015 IEEE 20th Conference on Emerging Technologies & Factory Automation (ETFA)*, 1–8. IEEE.

Reyna, Y. C. F., Y. M. Jiménez, J. M. B. Cabrera, and B. M. M. Hernández. 2015. "A reinforcement learning approach for scheduling problems". *Investigación Operacional* 36(3):225–231.

Rinciog, A., and A. Meyer. 2021a. "FabricatioRL". *GitHub Repository*. https://github.com/malerinc/fabricatio-rl.git.

Rinciog, A., and A. Meyer. 2021b. "Towards Standardizing Reinforcement Learning Approaches for Stochastic Production Scheduling". *CoRR* abs/2104.08196.

Rinciog, A., C. Mieth, P. M. Scheikl, and A. Meyer. 2020. "Sheet-Metal Production Scheduling Using AlphaGo Zero". *https://doi.org/10.15488/9640*.

Shahrabi, J., M. A. Adibi, and M. Mahootchi. 2017. "A reinforcement learning approach to parameter estimation in dynamic job shop scheduling". *Computers & Industrial Engineering* 110:75–82.

Siemens 2021. "Plant Simulation Documentation". *Siemens*. https://plant-simulation.de.

Silver, D., J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton et al. 2017. "Mastering the game of go without human knowledge". *Nature* 550(7676):354.

Stricker, N., A. Kuhnle, R. Sturm, and S. Friess. 2018. "Reinforcement learning for adaptive order dispatching in the semiconductor industry". *CIRP Annals* 67(1):511–514.

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

Thomas, T. E., J. Koo, S. Chaterji, and S. Bagchi. 2018. "Minerva: A reinforcement learning-based technique for optimal scheduling and bottleneck detection in distributed factory operations". In *2018 10th International Conference on Communication Systems & Networks (COMSNETS)*, 129–136. IEEE.

Wang, Y.-C., and J. M. Usher. 2005. "Application of reinforcement learning for agent-based production scheduling". *Engineering Applications of Artificial Intelligence* 18(1):73–82.

Wang, Y.-F. 2020. "Adaptive job shop scheduling strategy based on weighted Q-learning algorithm". *Journal of Intelligent Manufacturing* 31(2):417–432.

Waschneck, B., A. Reichstaller, L. Belzner, T. Altenmüller, T. Bauernhansl, A. Knapp, and A. Kyek. 2018. "Optimization of global production scheduling with deep reinforcement learning". *Procedia CIRP* 72(1):1264–1269.

Zhang, C., W. Song, Z. Cao, J. Zhang, P. S. Tan, and X. Chi. 2020. "Learning to Dispatch for Job Shop Scheduling via Deep Reinforcement Learning". *Advances in Neural Information Processing Systems* 33.

Zhang, W., and T. G. Dietterich. 1996. "High-performance job-shop scheduling with a time-delay TD (λ) network". In *Advances in neural information processing systems*, 1024–1030.

AUTHOR BIOGRAPHIES

Alexandru Rinciog is a PhD Candidate at the Mechanical Engineering Department of the TU Dortmund University. He earned his M.Sc. in Computer Science from Karlsruhe Institute of Technology in 2019 focusing on machine learning and software engineering. As part of the DFG project "Smart Production Control", his main research focus is the combined application of reinforcement learning and constraint programming on the combinatorial optimization problems of production scheduling. Email: alexandru.rinciog@tu-dortmund.de. Site: https://dls.mb.tu-dortmund.de.

Anne Meyer is a junior professor for digitalization in logistics and supply-chain management at the TU Dortmund University. She obtained her Ph.D. from Karlsruhe Institute of Technology in 2015. The 10-year collaboration with the FZI Research Center for Information Technology, served to grow her hands-on expertise in the fields of logistics, optimization and data analytics. anne2.meyer@tu-dortmund.de. Site: https://dls.mb.tu-dortmund.de.