Integration of Reinforcement Learning and Discrete Event Simulation Using the Concept of Experimental Frame

Thorsten Pawletta, Jan Bartelt

Research Group Computational Engineering and Automation (CEA), Wismar University of Applied Sciences, Philipp-Müller-Str. 14, 23966 Wismar, Germany; {thorsten.pawletta, jan.bartelt}@hs-wismar.de

SNE 33(3), 2023, 101-109, DOI: 10.11128/sne.33.tn.10651
Received (EUROSIM 2023): 2023-02-01; Acc.Conf.: 2023-03-30
Received SNE: 2023-08-15; Accepted: 2023-08-31
SNE - Simulation Notes Europe, ARGESIM Publisher Vienna
ISSN Print 2305-9974, Online 2306-0271, www.sne-journal.org

Abstract. Reinforcement Learning (RL) is an optimization method from the field of Machine Learning. It is characterized by two interacting entities referred to as the agent and the environment. The goal of RL is to learn how an agent should act to achieve a maximum cumulative reward in the long-term. A Discrete Event Simulation Model (DESM) maps the temporal behavior of a dynamic system. The execution of a DESM is done via a simulator.
The concept of an Experimental Frame (EF) defines the general structure used to separate the DESM into the dynamic system, called the Model Under Study (MUS), and its application context. This supports the diverse use of a MUS in different experimental contexts. This paper explores the generalized integration of discrete event simulation and RL using the concept of EF. The introduced approach is illustrated by a case study that has been implemented using MATLAB/Simulink and the SimEvents blockset.

Introduction

In modeling and simulation (M&S) theory [19], a model describes the dynamic behavior of a real or virtual system. A discrete event model is characterized by a finite number of states over a continuous time base. The execution of the model, i.e. the calculation of trajectories, is performed using a simulator. In the versatile use of a model, it should be developed independently from the context of use. The reference to a concrete experiment can be mapped by way of an Experimental Frame (EF).

An EF specifies the conditions under which a system is observed or a model experimented with [19, 17]. The model used is called the Model under Study (MUS). Depending on the experiments to be performed using a MUS, the corresponding EFs must be specified. Depending on the EF, the same model can be used in a parameter study, sensitivity analysis, optimization, etc. A context-specific EF and the MUS form the simulation model (SM) to be executed by the simulator. The concept of EF can be applied to all dynamic system models and their simulators but this paper focuses on discrete event simulation models (DESM).

The execution of a goal-directed experiment using a DESM and a simulator requires an *Experiment Control* (EC) [20]. The EC defines the goals and constraints of an experiment and structures the experiment process. Inspired by Breitenecker's [1] approach to structuring *simulation-based experiments* (SBE), Pawletta et al. [12] and Schmidt [13] concretized the concept of EC by introducing a *Simulation Method* (SimMeth) and *Experiment Method* (ExpMeth). The SimMeth controls the execution of the simulation runs and ExpMeth consists of arbitrary numerical methods. ExpMeth are used for the pre- and post-processing or to control the SimMeth, such as in simulation-based optimization experiments [3, 13].

Reinforcement Learning (RL) [15] in combination with a dynamic system simulation can be considered a specific SBE. According to Gosavi [8], RL is a simulation-based optimization of Markov Decision Processes (MDPs). In terms of RL, the MDP is modeled as an environment and the agent acts as a controller for the MDP. The agent influences the environment by actions, while the environment performs state transitions and responds with the new states and reward values for each transition. The optimization goal is to learn how the agent should act to achieve a maximum cumulative reward in the long-term.

In contrast to a discrete event dynamic system, an MDP is a discrete time process where the time base is only used for the sequential ordering of states. In addition, not all states of the MUS are usually of interest to the RL. Accordingly, the states of the MUS must be converted into MDP-compliant states. Due to the methodological differences, the combination of the two methods, RL and discrete event simulation, often lead in practice to implementations that are difficult to maintain and MUS that are not generally usable.

After discussing related work in section one, section two presents a general concept of SBE using the EF, and the basics of RL. In section three, an EF-based approach to integrate the RL method and DESM for performing SBE is presented. In addition, the approach is demonstrated using a case study.

1 Related Work

The combination of discrete event simulation and RL is part of numerous works in the field of M&S. The works can be roughly divided into four categories: (i) mathematically-oriented basics, (ii) application-specific solutions, (iii) extensions of the simulation environments, and (iv) generic M&S oriented approaches. The first category focuses on the mathematical principles for combining the two methods without addressing software implementations such as in the work of Gosavi [8].

Work of the second category often uses simplified models that are mapped as RL-compliant MDP environments [14, 10]. The models are not a general-purpose MUS. Other works use their own or proprietary simulation environments to represent the MUS and implement the RL-specific part in Python, often using AI libraries such as TensorFlow [6, 5]. In the case of Feldkamp et al. [6], the coupling is done using a client-server approach with the RL part acting as the server and the simulation environment as the client. The necessary state transformations for the coupling of the two methods are implemented on the RL side in Python.

The coupling of different software systems requires advanced programming skills. To simplify the application of RL techniques, the manufacturers of simulation environments have started to integrate RL-specific elements into their software systems such as described in the work Mahdavi and Tyler [11], Greasley [9], and The MathWorks [16] (third category).

The basic principle is to provide configurable RL agent objects that have a typical RL input/output interface (action as output, observation and reward as inputs). They can be used as parts to build a simulation model. In addition, RL-specific methods are provided that support, for example, the training of an agent. These supplements facilitate the integration of the two methods for users. Nevertheless, the method integration remains a challenge. Aspects such as a clear and maintainable structuring of the simulation model in a multi-purpose MUS, the reward calculation, the conversion of MUS system states or outputs into RL observations and vice versa RL actions into MUS inputs etc. have to be solved by the user.

Work of the fourth category deal with approaches to solving the challenges outlined above. Capocchi and Santucci [2] describe a structuring approach for integrating RL and DESM based on the Discrete Event System Specification (DEVS) [19]. They show the specification of agents with DEVS and how DEVS-based agents can communicate with an environment specified in DEVS. The focus is on the specification of agents. Choo et al. [4] analyzed the necessary transformations in the communication between an RL agent and an environment implemented as a DESM. Both agent and environment form the DESM. To structure the communication between both parts of the DESM, they introduce specific components called a decoder and encoder.

On the basis of Choo et al. [4], the concept of simulation-based experiments in the work of Pawletta et al. [12] and Schmidt [13], and the concept of EF, in the following, refers to a more advanced approach for the integration of RL and discrete event simulation is developed.

2 Basics of SBE, RL and EF

2.1 SBE and Concept of Experimental Frame

Schmidt [13] divides simulation-based experiments (SBEs) into three classes. In the following, only the first two classes are considered. The execution of one or more simulation runs by a SimMeth constitutes a *simple SBE* if the SimMeth is invoked directly by the user or an EC. The SimMeth sets the input values for the DESM, the execution parameters for the simulator, and controls the simulation runs. As previously mentioned, the EC specifies the experiment goals and experiment process. Besides the SimMeth, the EC can invoke further methods for pre- and post-processing.

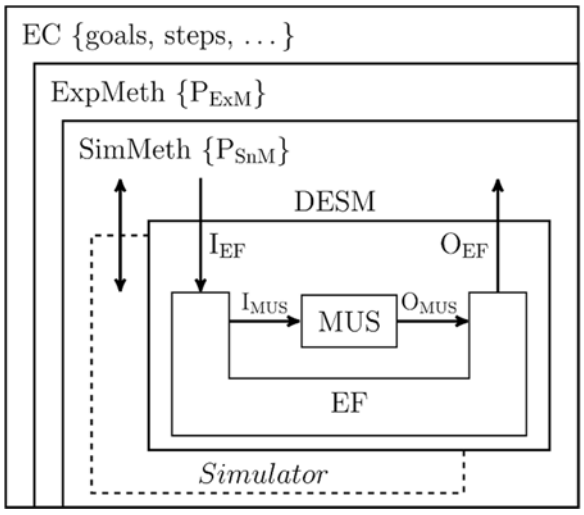

Figure 1: Basic structure of a complex SBE.

In a *complex SBE*, the SimMeth is controlled by an ExpMeth, for example, by a numerical optimization method. Figure 1 shows the basic structure of a complex SBE. Both the SimMeth and ExpMeth define the process parameters (P_{ExM}, P_{SnM}).

An SBE uses a DESM in a specific context. It defines certain experimental goals, conditions, and parameters. The concept of the EF separates the MUS from a specific context of use to improve the reusability of the MUS. Formally, Zeigler [18] defines the function of an EF using a 7-tuple.

$$EF = < T, I, C, O, \Omega_I, \Omega_C, SU >$$

Here T represents the time base, I and O the set of input and output variables of the MUS (equivalent to I_{MUS} and O_{MUS} in Figure 1), C the set of run control variables, Ω_I the set of admissible input segments, Ω_C the set of admissible control segments, and SU the set of summary mappings.

Set Ω_I refers to the input variables of the MUS and to the input/output relationships in the EF. Set Ω_C defines the experimental constraints which is a subset of C×T.

The experiment objectives are mapped to the variables, which are called *interest variables*. The set SU defines the determination of the interest variables based on the MUS output segments. The interest variables are the typical output variables of the EF (O_{EF}).

The implementation of an EF is done using three types of component, called *generator (Gen)*, *acceptor (Acc)* and *transducer (Trans)* [18, 19], as illustrated in Figure 2.

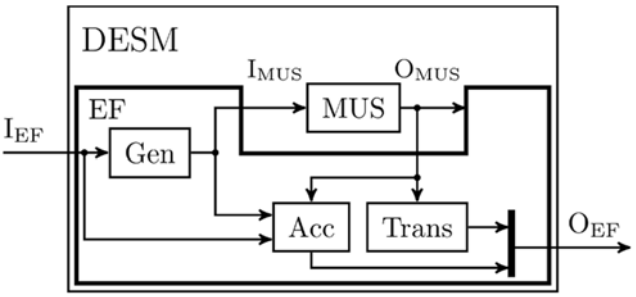

Figure 2: Basic structure of a DESM with MUS and EF. An EF does not necessarily have to contain all three components and the coupling relationships are not fixed.

Gen initializes the configurable parameters of the MUS and calculates the input segments for the MUS which can also be inputs of the Trans or Acc.

The Acc defines the admissible control segments and monitors their compliance. The output of the Acc is run control information. The Trans calculates the SU.

2.2 Reinforcement Learning

According to Sutton and Barto [15], RL focuses on the sequential decision-making by an agent that interacts with a real or virtual environment. The agent is trained by its interactions with the environment. The goal of RL is to learn a behavioral strategy π: $S \rightarrow A$ for the agent that assigns an action $a \in A$ to each state $s \in S$ of the environment. Thus, the agent can act as a controller for the environment. Using RL, a distinction is made between the training and deployment of an agent, although the agent can continue learning during deployment. The basic RL framework is shown in Figure 3.

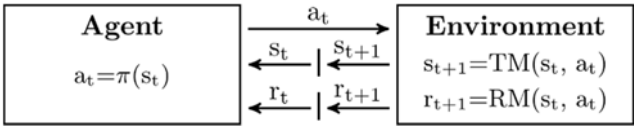

Figure 3: Basic RL framework.

In model-free RL, the agent only knows the allowed action set A at the start of training. The states $s \in S$ of the environment are unknown to the agent.

When an action $a_t \in A$ takes effect, the environment determines its next state s_{t+1} as well as a reward value r_{t+1} using a state transition model $TM: S \times A \rightarrow S$ and reward model $RM: S \times A \rightarrow R$.

The next state and the reward value are sent back to the agent. The index t marks a sequence of states in the sense of a MDP.

Through iterative interactions with the environment, the agent obtains information about possible states of the environment and the benefits of actions, gradually improving its behavioral strategy π. The goal of learning is to maximize the sum of the rewards until a goal is reached.

A variety of different learning strategies have been developed for RL agents such as Q-learning (QL), Deep Q Networks (DQN) etc.

We briefly consider Q-learning that uses formula (1) to learn a strategy π using a table function called the Q-matrix. A matrix element $Q(s,a)$ represents the estimated benefit of an action a_t when it is performed in the state s_t of the environment. The updated $Q(s_t,a_t)$ value of the current state/action tuple (s_t,a_t) is calculated from the previous $Q(s_t,a_t)$ value, the currently received reward r_{t+1}, and the maximum Q-value $(\max_a Q(s_{t+1},a))$ of all possible actions in the currently received next state s_{t+1}. The variables α and γ are process parameters, called hyperparameters.

$$Q(s_t a_t) \leftarrow Q(s_t a_t) + \alpha\,[r_{t+1} + \gamma \cdot \max_a Q(s_{t+1},a) - Q(s_t,a_t)] \qquad (1)$$

The training takes place in episodes. Episodes are independent of each other. Each episode starts in an initial state s_0 of the environment and ends when a target state s_{target} or abort state s_{abort} is reached. At the beginning of the training, the agent selects an action $a \in A$ randomly. This is called *exploration*. As the learning process progresses, the agent increasingly uses the knowledge it has acquired to select an action which is called *exploitation*. The ratio ε of exploration to exploitation is adjusted over the course of the training. After the completion of a defined number of training episodes, the behavioral strategy $a = \pi(s)$ is derived from the training data.

3 Integration of Reinforcement Learning and Discrete Event Simulations

3.1 Experimental Frame for Reinforcement Learning in the Training Phase

The basic structure of a DESM with an EF for RL in the training phase is shown in Figure 4. Although the approach is not limited to DESM, we will only focus on it.

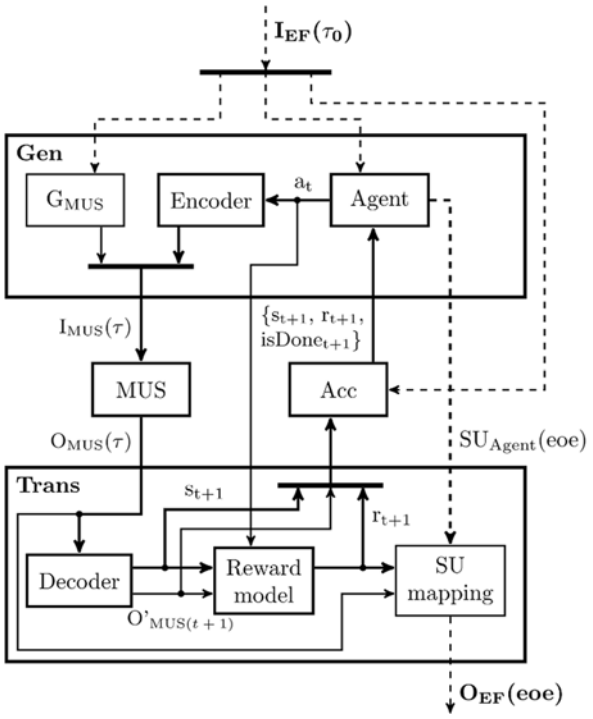

Figure 4: Basic structure of a DESM with MUS and EF for the training phase of a RL experiment.

The DESM consists of the three EF-components Gen, Trans, and Acc as well as the MUS. I_{EF} and O_{EF} represent the input/output interface of the DESM. The variables τ and t represent different time bases where τ is the continuous time base of the dynamic MUS and t the discrete time base for ordering the sequential states of the RL method.

The input variables I_{EF} are initialized by the EC or SimMeth (see Fig 1) at the simulation start time τ_0. The SimMeth executes single or multiple simulation runs.

Each simulation run corresponds to one episode of the RL method. The results of an episode get back via O_{EF} at the *end of an episode (eoe)*.

The Gen component is composed of three subsystems. *Gen.*G_{MUS} is a classical generator that initializes the parameters of the MUS at the beginning of an episode (τ_0) and calculates input segments Ω_I for the MUS inputs $I_{MUS}(\tau)$ over the course of an episode.

The RL agent is also mapped as a subsystem of the Gen (*Gen.Agent*) because the generated actions $a_t = \pi(s_t)$ are inputs of the MUS. The agent's hyperparameters are initialized at the beginning of an episode using the input interface $I_{EF}(\tau_0)$. The typical initialization parameters of component *Gen.Agent* are action set *A*, exploration rate ε and, in Q-learning, the current configuration of the *Q-matrix*.

Action a_t must often be transformed into MUS compatible inputs $i(\tau) \in I_{MUS}(\tau)$. For this data transformation, the encoder approach introduced by Choo et al. [4] is used. The subcomponent *Gen.Encoder* defines an application-specific transformation $i(\tau)=h(a_t)$. A typical example of such a data transformation is the generation of multiple input segments for the MUS based on a single action value. In addition to the inputs' new state s_{t+1} and the reward value r_{t+1} of the agent, a third input $isDone_{t+1}$ is defined.

The *isDone* information is a Boolean value that signals the end or cancellation of an episode to the agent. At the end of an episode*(eoe)*, the agent creates summary mapping $SU_{Agent}(eoe)$ that contains values such as the *number of steps in the episode*, the *total reward of an episode*, or the strategy learned so far (e.g. the *Q-matrix*). The $SU_{Agent}(eoe)$ is passed to the *Trans* component to create an overall summary mapping of the episode.

The *Trans* component is also composed of three subsystems. *Trans.Decoder* is a data transformation component. First, it defines the calculation of the *interest values* ($O'_{MUS(t+1)}$) from the current outputs of the MUS ($O_{MUS}(\tau)$) related to the time base of the RL, i.e.

$$O'_{MUS(t+1)}=f(O_{MUS}(\tau))$$

An example of such a transformation would be the calculation of the maximum queue length based on the previous queue occupancy. Second, it defines the transformation of the interest values $O'_{MUS(t+1)}$ to a state s_{t+1} in the RL space, that is

$$s_{t+1}=g(O'_{MUS(t+1)}).$$

Choo et al. [4] characterized this transformation form as

(i) *State Exhaustiveness* and
(ii) *State Mutual Exclusiveness.*

Here, (i) means that all interest values of the MUS are mapped into one state for the RL and (ii) that for each particular interest value of the MUS, there is only one corresponding state in the RL space.

The RM according to Section 2.2 is mapped in the component *Trans.Rewardmodel* because the reward value is an interest value based on the output variables of the MUS. The reward value characterizes a state tran-sition $s_t \rightarrow s_{t+1}$ in the RL space and is calculated by

$r_{t+1}=RM(s_t, a_t)$ or $r_{t+1}=RM(s_t, s_{t+1})$.

Defining the RM is sometimes a difficult problem. Our own experiments showed that the reward calculation can often be defined more efficiently based on the $O'_{MUS(t+1)}$ values, i.e.

$r_{t+1}=RM(O'_{MUS(t+1)}, a_t)$, or only $r_{t+1}=RM(O'_{MUS(t+1)})$.

The third subcomponent *Trans.SUmapping* implements the overall SU of an episode and passes it at the end of an episode to the EF output O_{EF}. In addition to the summary mapping of the agent (SU_{Agent}), the overall SU may also include the trajectories of the MUS outputs, a cumulative reward record etc.

In accordance with the concept of the EF in Section 2.1, the *Acc* component checks the compliance with the restrictions and termination conditions for the episode based on defined run control information.

The run control variables can be initialized at the start of an episode via the EF input $I_{EF}(\tau_0)$. Typical run conditions to be monitored include (i) the simulation interval $[\tau_0, \tau_{final}]$ of the MUS and thus the maximum duration of an episode and (ii) the detection of illegal system states or the reaching of a target state based on the $O'_{MUS(t+1)}$ or RL-related s_{t+1} values.

Accordingly, the *Acc* checks the newly calculated states s_{t+1} of the RL space as well as the reward values r_{t+1} before sending them to the *Gen.Agent* component. Furthermore, the *Acc* sets the Boolean *isDone* value which signals the continuation or the end of an episode according to the *Gen.Agent*.

3.2 Reinforcement Learning as a Simulation-based Experiment

According to the classification of SBE in Section 2.1, the RL is a complex SBE and has the general structure shown in Figure 1.

The goal of the experiment is

(i) to learn the best possible behavioral strategy π of an agent,
(ii) to extract the best strategy from the training data, and
(iii) to deploy the strategy.

When deploying, we have to distinguish whether a strategy is used with or without the further learning of the agent. The EC has to define these experimental steps. The steps involved in training and deploying the strategy require an ExpMeth that controls a SimMeth.

The ExpMeth *training* contains the following basic steps:

(1) Set the RL process parameters P_{ExM}, such as the learning rate, exploration rate, maximum number of episodes, Q-matrix etc.
(2) Set the simulation execution parameters P_{SnM} for the SimMeth, such as the simulator to be used, the simulation time interval etc.

(3) Set the DESM parameters for the EF components and the MUS and prepare the DESM for executing using a SimMeth.
(4) Initialize the statistical variables, such as those used to record the total reward per episode etc,
(5) Compute the defined number of episodes, i.e. call the SimMeth into a loop to execute the DESM, update the statistical variables after each episode, and check whether to abort the training or continue with another episode.
(6) Determine and save the best policy π, and plot essential learning results.

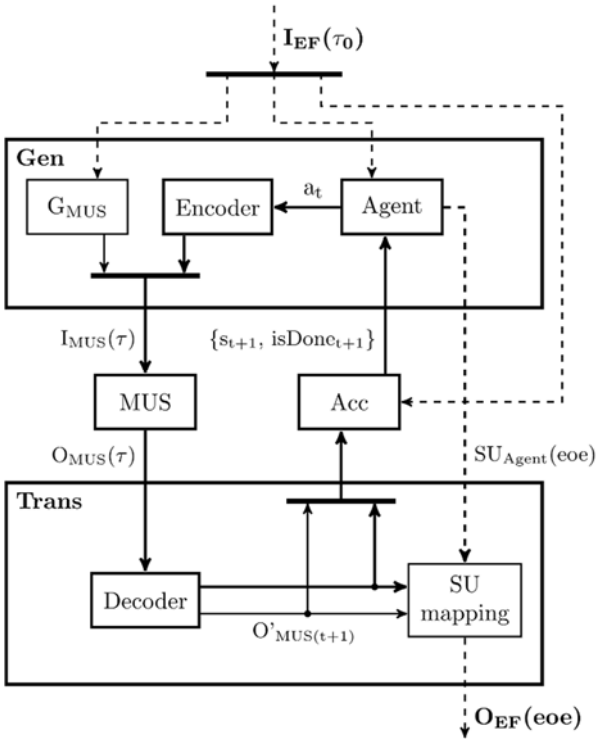

Figure 5: Basic structure of a DESM with MUS and EF for an experiment deployment without training.

An ExpMeth *deployment_with_training* has to implement nearly the same steps as the method *training* described before. The DESM with MUS and EF corresponds to the structure in Figure 4.
In contrast, the procedure and EF for an experimental step *deployment_without_training* is simplified. No explicit ExpMeth is required.

In the EC, the experiment parameters are defined and SimMeth is called on directly according to the number of simulation runs to be executed. Figure 5 shows the reduced structure of the DESM.

3.3 Case Study

The presented approach for integrating dynamic system simulations and RL was investigated on the basis of various case studies using MATLAB/Simulink and SimEvents.
Since the MathWorks' RL toolbox (release R2022a) supports integration with DESM – implemented using the SimEvents blockset – only via workarounds, we used a self-programmed Q-learning agent. In the following, we discuss some basic aspects of a case study without going into implementation details. The full implement-tation is available on Github [7].

The MUS is a simple server line consisting of an entity generator, a convertible operating unit, and two downstream servers connected in parallel with separate input queues as shown in Figure 6.

The operating unit can process two types of entity (*jobType=1|2*). A separate processing time can be defined for each entity type (*procT1, procT2*). A retooling time (*retoolingT*) is necessary when the entity type is changed in the operation unit. The calculation of the entity type and retooling time dependent processing time is done in the simulation runtime using two Simulink functions (not shown in Figure 6).

After processing, branching into one of the two FiFo queues of the downstream servers takes place depending on the entity type. The downstream servers have different processing times (*saleT1, saleT2*).

The definition of the different time values is determined by a value vector *param=[procT1, procT2, retoolingT, saleT1, saleT2]* at input port3 at the simulation start time τ_0. Entities are generated via input events (*msgGenJob*) at input port1. The entity type (*jobType*) to be generated follows on from the value at input port2.

After an entity has been processed in the operating unit, the MUS generates an output event (*y_msgFinish*) at output port1. Furthermore, the current tool setting (*sSetting*) of the operating unit, the current queue lengths (*y_#jobsQ1*, *y_#jobsQ2*), and the number of completed entities on the downstream servers (*y_#jobs1sold*, *y_#jobs2sold*) are output as data from port2 to port6.

Hence, input set I_{MUS} and output set O_{MUS} are defined by:

I_{MUS} = {msgGenJob(τ), type(τ), param(τ_0)}
O_{MUS} = {y_msgFinish(τ), y_sSetting(τ), y_#jobsQ1(τ), y_#jobsQ2(τ), y_#jobs1sold (τ), y_#jobs2sold(τ)}

Obviously, the MUS models the dynamic system behavior independent of a concrete experiment. The goal of the RL experiment is to learn the best possible injection strategy of the two entity types into the MUS so then the queues have the most balanced stock of both types available for the downstream servers.

After training is completed, the best strategy $\pi: S \rightarrow A$ should be extracted from the training data so then, subsequently, the agent can act as a controller of the MUS.

The EC is implemented as a MATLAB script. It defines the parameter sets P_{ExM} and P_{SnM}, such as:

(1) *action set={1, 2}*, coding the two entity types
(2) learning rate $\alpha=0.8$
(3) *sim. time interval=[*$\tau_0=0$, $\tau_{final}=480$*]* per episode
(4) The *number of episodes=20000*, etc.

and calls the ExpMeth *training*. This calls the SimMeth into a loop to execute the DESM for one episode.

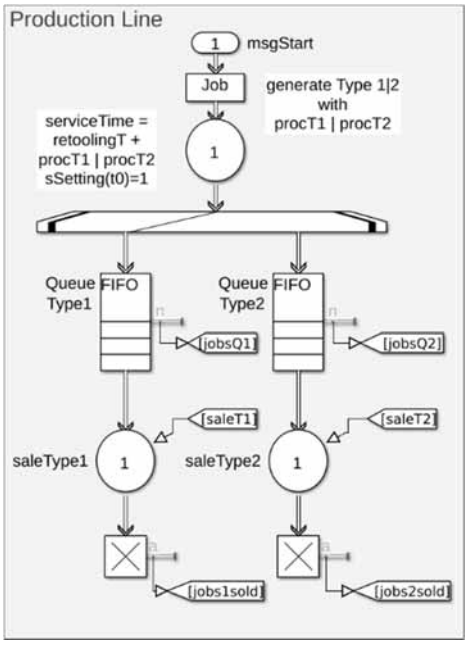

Figure 6: Structure of the MUS in SimEvents.

The MATLAB built-in function *sim* is used as SimMeth. The data transfer between the different methods and the DESM is mainly done via data workspaces.

Figure 7 shows the top-level structure of the DESM for the training phase. It contains all components according to the general approach shown in Figure 4. The MUS named *Prodline* provides the input-output interface described above with $I_{MUS}(\tau)$ and $O_{MUS}(\tau)$. The I_{EF} and O_{EF} of the EF are not visible on the top-level structure of the DESM.

This interface is realized via workspace variables. The encapsulation of the Gen and Trans subcomponents has been omitted. *Parameters* is a G_{MUS}. It generates the constant input segments for the MUS input vector *param* for initializing the MUS parameters. The initialization of parameters for the *Agent*, such as the *Q-matrix*, and for the *Acc* at the beginning of each episode is encoded directly in these components. Analogously, the components *Agent* and *SU.Mapping* output the O_{EF} at the end of an episode.

At simulation start time τ_0, an episode is started by the Agent sending an event *msgGenJob* and setting an action $a_t=\{1 \mid 2\}$ at the *action* port. In this case, the outputs of the Agent are compatible with the inputs of the MUS in value and timestamp with respect to the global simulation clock. Hence, the Agent's outputs are only forwarded by the *Encoder* to the MUS *ProdLine* that generates a new entity with *jobType=action value*.

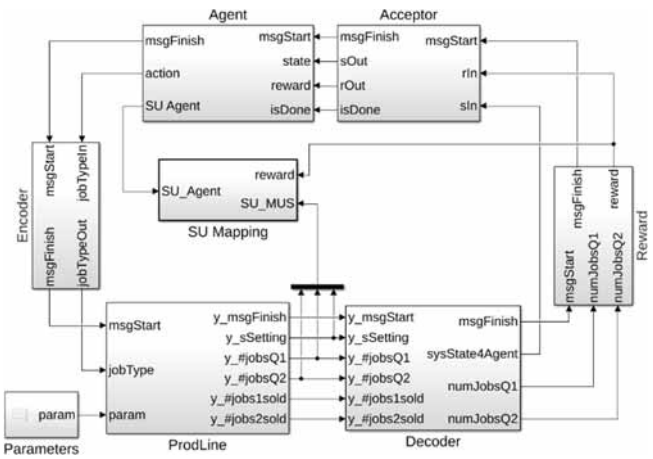

Figure 7: Top-level structure of the DESM with MUS (named *ProdLine*) and EF for the training phase of an RL experiment.

When an entity has completed on the operation unit and been forwarded to one of the two queues, an output event *y_msgFinish(τ)* is sent to activate the *Decoder* and the output data of the MUS is updated. Study-specific *output data(τ)* is passed to the *SU_Mapping* for trajectory recording.

The *Decoder* selects the information relevant to the RL from the MUS *output data(τ)* and calculates the new state s_{t+1} of the RL space (output port *sysState4Agent*). To limit the RL space, the state set of the two queues is limited to a maximum length ($qlength_{max}$) to be considered (2), (3).

The new state s_{t+1} is calculated based on the current tool setting (*sSetting*) of the operating unit and the two limited queue lengths (4).

$$qlQ1 = \max(y_{\#jobsQ1}, qlength_{max}) \quad (2)$$

$$qlQ2 = \max(y_{\#jobsQ2}, qlength_{max}) \quad (3)$$

$$s_{t+1} = (sSetting - 1) \cdot (qlength_{max} + 1)^2 + \\ + qlQ1 \cdot (qlength_{max} + 1) + qlQ2 + 1 \quad (4)$$

An experiment, with *sSetting={1,2}* and $qlength_{max}=30$, results in set of RL states *S={1,2,3,…,1921,1922}*.

After decoding, the reward calculation is activated by an event (*msgFinish*). Contrary to the general approach, the reward is not calculated using the RL-related state $s \in S$ but on the basis of MUS-related variables $O'_{MUS}(t+1)$. In terms of content, both approaches are identical but the second one resulted in a much better structured reward computation (5).

$$r_{t+1} = \begin{Bmatrix} 100 \mid qlQ1 \geq 10 \; and \; qlQ2 \geq 10 \\ qlQ2^2 \mid qlQ1 \geq 10 \; and \; qlQ2 < 10 \\ qlQ1^2 \mid qlQ1 < 10 \; and \; qlQ2 \geq 10 \\ \frac{qlQ2^2 \cdot qlQ1^2}{100} \mid else \end{Bmatrix} \quad (5)$$

After the reward calculation, the *Acceptor* is activated by an event (*msgFinish*). No constraints are defined for s_{t+1} and r_{t+1}, so they are only passed to the *Agent* (*rIn* to *rOut* and *sIn* to *sOut*).

The *Acceptor* defines only a control segment for the simulation time interval $[\tau_0, \tau_{final}]$, which defines the length of an episode. Moreover, restrictions can be defined depending on the queue lengths, for example, thus the premature termination of an episode. At the termination of an episode, the *Acceptor* schedules an internal event with an infinitesimal time advance. The time delay is necessary for data updates in the *Agent* and *SU_Mapping* at the end of an episode.

The *Acceptor* activates the *Agent* via an event (*msgFinish*) and signals using the Boolean variable *isDone* whether the end of an episode has been reached or not.

The *Agent* executes its learning rules and, depending on the *isDone* value, calculates a new *action* value or performs a final data update.

When training the Q-learning agent, the total reward, i.e. the sum of the rewards of an episode, converged on its final value after about 5000 episodes.

The time trajectories of the queue lengths computed using the learned policy π: $S \rightarrow A$ shown in Figure 8 prove that the agent can act as a controller of the MUS.

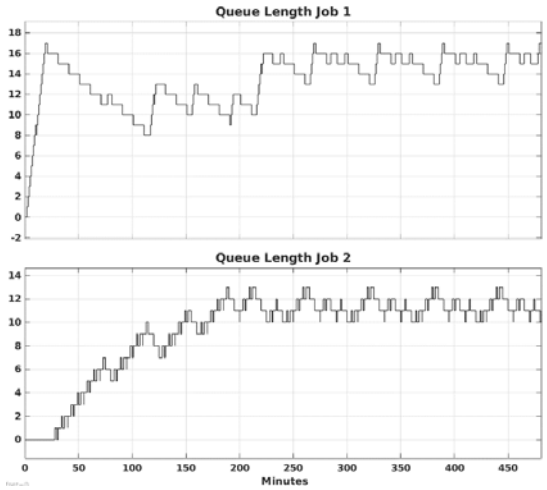

Figure 8: Time trajectories of the queue lengths computed using the learned policy.

4 Conclusions

The integration of dynamic system simulation and RL methods has a high application potential for both M&S and AI applications.

On the basis of the concept of EF and the general structure of complex SBE, it has been shown how a clear methodological separation can be made so then the dynamic system models, simulation methods, simulators and AI methods can be developed independently and reused in different contexts. The methodological considerations have been practically underpinned using a case study.

In the next steps, the methodological approach will be formalized using the Discrete Event System Specification (DEVS). Furthermore, the specification and automation of simulation-based RL experiments based on an extended System Entity Structure and Model Base (SES/MB) framework will be investigated.

References

[1] Breitenecker F. Models, methods and experiments - a new structure for simulation systems. *Mathematics and Computers in Simulation*; 1992. 34(3). 231–260.

[2] Capocchi L, Santucci JF Discrete event modeling and simulation for reinforcement learning system design. *Information*; 2022. 13(3). 1-13

[3] Carson Y, Maria A. Simulation optimization: methods and applications. In: *Proceedings of the 1997 Winter Simulation Conference*; 1997. 118-126.

[4] Choo B, Graham C, Stephen A, Dadgostari F, Beling PA. Reinforcement learning from simulated environments: an encoder decoder framework. In: *Proceedings of the SCS SpringSim'20 (Virtual) Conference;* 2020. 12 pages.

[5] Ehn G, Werner H. Reinforcement learning for planning of a simulated production line. *Master thesis*, Lund University; 2018.

[6] Feldkamp N, Bergmann S, Strassburger S. Simulation-based deep reinforcement learning for modular production systems. In: *Proceedings of the 2020 Winter Simulation Conference*; 2020. 1596-1607.

[7] FG CEA Integration of RL and Discrete Event Simulation: A Case study using MATLAB/ Simulink/ SimEvents. Wismar Univ. of Applied Sciences; Wismar. https://github.com/cea-wismar.

[8] Gosavi A. Solving markov decision processes via simulation. In Fu, M.C. (ed.), *Handbook of Simulation Optimization.* Springer; 2015. 341-374.

[9] Greasley A. Implementing Reinforcement Learning in Simio discrete-event simulation software. In *Proceedings of the SCS SummerSim'20 (Virtual) Conference*; 2020. 11 pages.

[10] Leng J, Jin C, Vogl A, Lui H. Deep reinforcement learning for color-batching resequencing problem. *Journal of Manufacturing Systems*, Elsevier, 56(2020); 2020. 175-187.

[11] Mahdavi A, Wolfe-Adam T. Artificial Intelligence and Simulation in Business [White Paper]; 2020. (reading date 10/20/2022).

[12] Pawletta T. Specification and execution of simulation models and experiments. *MS Workshop 'One simulation model is not enough'*, Univ. of Rostock, Dep. of Computer Science; 2019. https://www.cea-wismar.de/pawel/Forschung/Poster_Slides/2019-04-23-Presi_FG-CEA_UnivRo-WS_reducedSize.pdf.

[13] Schmidt A. Variant management in modeling and simulation using the SES/MB framework [Dissertation]. In: *Advances in Simulation.* Bd. 30. TU Publisher Vienna (in German); 2019.

[14] Shuhui Q, Wang J, Shivani G. Learning adaptive dispatching rules for a manufacturing process system by using reinforcement learning approach. In: *2016 IEEE - 21st International Conference on Emerging Technologies and Factory Automation (ETFA);* 2016. 1-8.

[15] Sutton RS, Barto AG. *Reinforcement learning: an introduction – 2nd edition*. MIT Press; 2018.

[16] The MathWorks. Reinforcement Learning Toolbox. *https://mathworks.com/products/reinforcement-learning.html*, ©1994-2022 The MathWorks, Inc.; 2022.

[17] Traore MK, Muzy A. Capturing the dual relationship between simulation models and their context. *Simulation Modeling Practice and Theory, Elsevier*, 14(2006); 2006. 126-142.

[18] Zeigler BP. *Multifacetted Modelling and Discrete Event Simulation.* Academic Press; 1984.

[19] Zeigler BP, et al. *Theory of modeling and simulation – 3rd edition.* Elsevier, Academic Press; 2018.