Integrating Reinforcement Learning and Discrete Event Simulation Using the Concept of Experimental Frame: A Case Study With MATLAB/SimEvents

Thorsten Pawletta*, Jan Bartelt

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

SNE 33(4), 2023, 167-174, DOI: 10.11128/sne.33.sw.10664
Selected ASIM WS 2023 Postconf. Publication: 2023-10-15
Received Revised Improved: 2023-11-22; Accepted: 2023-11-25
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 characterized by two interacting entities, the agent and the environment. The environment is a Markov Decision Process (MDP). The goal of RL is to learn how an agent should act to achieve a maximum cumulative reward in the long-term. In discrete-event simulation (DES), the dynamic behavior of a system is represented in a model (DESM) that is executed via a simulator. The concept of Experimental Frame (EF) provides a structural approach to separating the DESM into the Model Under Study (MUS) and its experimental context. Here, we explore the integration of a discrete event MUS as an environment for RL using the concept of EF. After discussing the methodological framework, a case study using MATLAB/Simulink and the SimEvents blockset is considered. The case study starts with an introduction of the discrete-event MUS for which a control strategy shall be developed. The MUS is reused in three experiments using specific EFs. First, an EF for the design of a heuristic control strategy with ordinary simulation runs is presented. Then, based on the methodological approach, specifics of the EF are considered when using a self-implemented Q-agent and the RL toolbox of MATLAB/Simulink.

Introduction

In modeling and simulation (M&S), a model describes the dynamic behaviour of a real or virtual system. The execution of the model 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 an Experimental Frame (EF).

An EF specifies the conditions under which a system is observed or a model experimented with (Zeigler [12], Zeigler et al. [14], Traore and Muzy [11]). The model used is called the Model under Study (MUS). Depending on the EF, the same MUS can be used in different experimental contexts, such as a parameter study, sensitivity analysis, optimization, etc. The EF and MUS form the simulation model (SM). Discrete event simulation models (DESM) are characterized by a finite number of states over a continuous time base.

The EF implements the interface for a Simulation-Based Experiment (SBE). Inspired by Breitenecker's [1] approach to structuring SBEs, Pawletta et al. [5] and Schmidt [7] introduced the concept of Simulation Method (SimMeth) and Experiment Method (ExpMeth). The SimMeth controls the execution of the simulation runs via a simulator and ExpMeths are arbitrary numerical methods. ExpMeths are used for pre- and post-processing or to control the SimMeth, such as in simulation-based optimization experiments (Carson and Maria [2]; Schmidt [7]).

Reinforcement Learning (RL) (Sutton and Barto [8]) in combination with a dynamic system simulation can be considered as a SBE. However, RL is an optimization method for Markov Decision Processes (MDPs). The MDP is modeled as an environment and an agent acts as a controller. The goal is to learn how the agent should act to achieve a maximum cumulative reward in the long-term.

In contrast to a DESM, an MDP is a discrete time process and the time base is only used for the sequential ordering of states. 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.

Here, the practical integration of both methods using the concept of EF is explored by means of a case study and using MATLAB/Simulink as well as the SimEvents blockset (MathWorks [9]). We start with some basics to SBEs, EF, RL and the usage of RL in a SBE. Then, the MUS is introduced for which a control strategy is developed. To present the reusability of the MUS in the context of different experiments using specific EFs, we start with the design of a heuristic controller. This is followed by two experiments on RL-based controller design.

This work is based on Pawletta and Bartelt [6]. More details on the theoretical background and related work is provided there.

1 Basics

Based on Pawletta and Bartelt [6], we briefly review the basics of structuring SBEs, the RL method, and the use of RL as a method of an SBE.

1.1 Structuring Simulation-Based Experiments

Schmidt [7] divides SBEs into three classes. We consider only the first two classes. 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 a supervisory Experiment Control (EC). An EC defines the goals and steps of an experiment and automates the experiment execution.

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 process parameters (P_{ExM}, P_{SnM}).

The EF separates the MUS from a specific context of use to improve the reusability of the MUS. Formally, Zeigler [13] defines the function of an EF with the tuple.

$$EF = \langle T, I, C, O, \Omega_I, \Omega_C, SU \rangle \quad (1)$$

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.

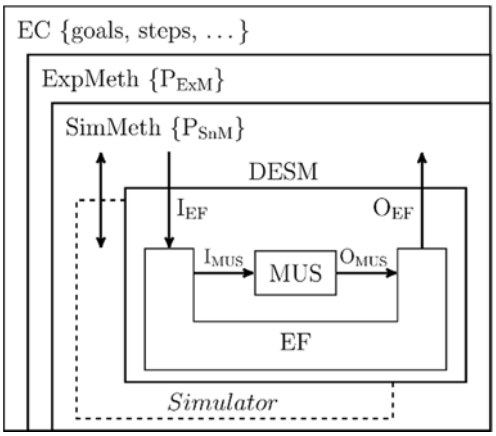

Figure 1. Basic structure of a complex SBE.

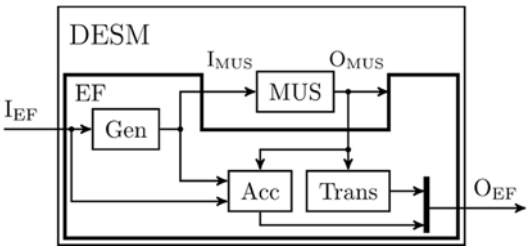

Figure 2. Basic structure of a DESM with MUS and EF. An EF does not necessarily have to contain all three components and the coupling relationships are not fixed.

Set Ω_C defines the experimental constraints. The experiment objectives are mapped to *interest variables*. Set SU defines the determination of the interest variables based on the MUS outputs. The interest variables are typical output variables of the EF. The implementation of an EF is done using three types of components, as illustrated in Figure 2 (Zeigler [13]; Zeigler et al. [14]). The *generator* (Gen) initializes the configurable parameters of the MUS and calculates the input segments for the MUS which can also be inputs of the *Acceptor* (Acc) or *Transducer* (Trans). 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.

1.2 The Method of Reinforcement Learning

According to Sutton and Barto [8], 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.

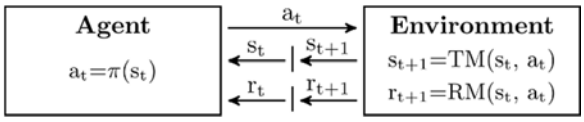

Figure 3. Basic RL framework.

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.

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

We briefly consider Q-learning that uses formula (2) 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 hyperparameters, i.e. they must be defined before training, but can still be adjusted during training.

$$Q(s_t a_t) \leftarrow Q(s_t a_t) + \alpha\,[r_{t+1} + \gamma \cdot \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \qquad (2)$$

The training takes place in independent episodes. 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 episodes, the behavioral strategy $a = \pi(s)$ is derived from the training data.

1.3 Integrating Reinforcement Learning into a Simulation-Based Experiment

When integrating RL and dynamic system simulation, the MUS forms the environment for the RL agent. The goal of such an SBE is

- to learn the best possible behavioral strategy of the agent,
- to extract this strategy from the training data, and
- to use it as a controller for the MUS.

The first two items are defined with an ExpMeth *training* that controls a SimMeth to execute simulation runs. The ExpMeth training contains the following basic steps:

- Set the RL-specific experiment parameters PExM such as the learning rate, exploration rate, maximum number of episodes, Q-matrix etc.
- Set the simulation execution parameters PSnM for the SimMeth, such as the simulator to be used, the simulation time interval etc.
- Set the DESM parameters for the EF and the MUS and prepare the DESM for executing using a SimMeth.
- Initialize statistical variables, such as those used to record the total reward per episode etc,
- 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.
- Determine and save the best policy π, and plot essential learning results.

Figure 4 shows the basic structure of a DESM with an EF for RL in the training phase. The variables τ and t represent the different time bases. τ is the continuous time of the MUS and t the discrete time for ordering the sequential states of the RL.

The input variables I_{EF} are initialized at the simulation start time τ_0, at the beginning of an episode. Results are get back via O_{EF} at the *end of an episode (eoe)*.

The *Gen* is subdivided into three subcomponents. *Gen.*G_{MUS} initializes the parameters of the MUS at τ_0 and calculates input segments Ω_I for the MUS inputs $I_{MUS}(\tau)$ over the course of an episode.

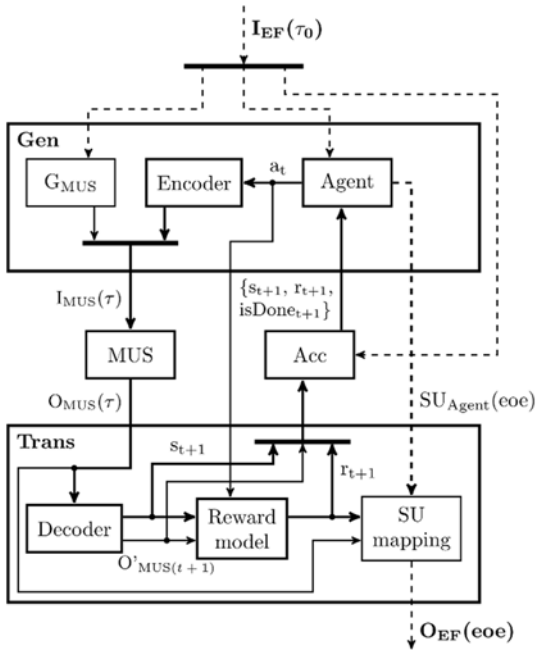

Figure 4. Basic structure of a DESM with a MUS and EF for RL in the training phase.

Gen.Agent maps the RL agent. It is part of the *Gen* because its output, an action a_t, produces inputs of the MUS. The agent's parameters are initialized via the input interface $I_{EF}(\tau_0)$. In addition to the ordinary agent inputs' new state s_{t+1} and the reward value r_{t+1}, the third input $isDone_{t+1}$ is a Boolean value that signals the end or cancellation of an episode. At the *eoe*, the agent creates a summary mapping SU_{Agent} that contains RL-specific values such as the *number of steps*, the *total reward*, or the strategy learned so far (e.g. the *Q-matrix*). The SU_{Agent} is passed to the *Trans*. *Gen.Encoder* defines a mapping $i(\tau)=h(a_t)$ to transform a single action value a_t into MUS compatible input values $i(\tau) \in I_{MUS}(\tau)$ as introduced by Choo et al. [3].

The *Trans* is also subdivided into three subcomponents. *Trans.Decoder* defines (i) the calculation of the *interest values* $O'_{MUS(t+1)}$ from the current outputs $O_{MUS}(\tau)$ of the MUS related to the time base of the RL (i.e. $O'_{MUS(t+1)}=f(O_{MUS}(\tau))$), and transformation of the interest values $O'_{MUS(t+1)}$ to a state s_{t+1} in the RL space (i.e. $s_{t+1}=g(O'_{MUS(t+1)})$). Thus, all interest values of the MUS are mapped into one state for the RL and for each particular interest value there is only one corresponding state in the RL space (Choo et al. [3]). The *Trans.Reward-model* maps the reward calculation. The reward value characterizes a state transition $s_t \rightarrow s_{t+1}$ in the RL space.

Defining the reward calculation is sometimes a difficult problem. Our own experiments showed that the reward value can sometimes be computed very efficiently based on the $O'_{MUS(t+1)}$ values. The component *Trans.SUmapping* implements the overall SU of an episode and passes it at the *eoe* to the output O_{EF}.

The *Acc* checks compliance with the constraints for the episode using defined run control information. Run control variables can be initialized via $I_{EF}(\tau_0)$. Typical run conditions to be monitored include the simulation interval $[\tau_0, \tau_{final}]$ of the MUS and thus the maximum duration of an episode, the detection of illegal states or the reaching of a target state. The *Acc* checks all the relevant quantities and sets the *isDone* value, before sending the tuple $(s_{t+1}, r_{t+1}, isDone_{t+1})$ to the *Gen.Agent*.

When deploying a learned strategy, we have to distinguish whether it is used with or without further learning of the agent. For an experiment *deployment without training* the EF simplifies as shown in Pawletta and Bartelt [6]. No explicit ExpMeth is required. The SimMeth is called directly in the EC according to the number of simulation runs to be executed.

2 Case Study

The basic implementation of the approach to integrate RL and discrete-event simulation introduced in Section 1.3 will be demonstrated by a case study using MATLAB/Simulink and the SimEvents blockset. The objective is to develop a control strategy for a MUS with discrete-event dynamics. First, the most general possible modeling of the MUS, i.e. without concrete references to an experiment, is discussed. Then, the same MUS is used in three experiments using different EFs: (i) to design a heuristic strategy, (ii) to learn a strategy with a self-implemented Q-agent, and (iii) to learn a strategy using MathWorks' dedicated RL toolbox (MathWorks [10]).

2.1 Model Under Study and General Objectives of the Control Design

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 5. 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.

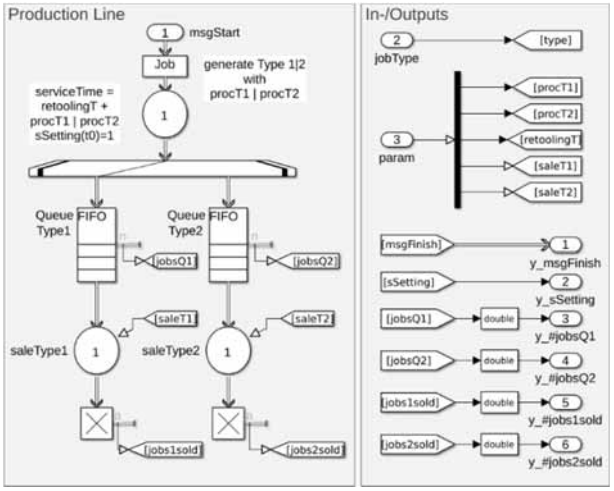

Figure 5. Structure of the MUS in SimEvents.

The calculation of the type of entity and retooling time dependent processing time is done during simulation runtime using two Simulink functions (not shown in Figure 5). 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(τ₀)}*
- O_{MUS}*={y_msgFinish(τ), y_sSetting(τ), y_#jobsQ1(τ), y_#jobsQ2(τ), y_#jobs1sold (τ), y_#jobs2sold(τ)}*

The MUS represents the dynamic system behavior independent of a concrete experiment. The goal of the following experiments is to design a controller with 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.

2.2 Designing a Heuristic Strategy

The top-level structure of the DESM for designing a heuristic control strategy is shown in Figure 6. The MUS named *Prodline* implements the input and output interface described in Section 2.1 with $I_{MUS}(\tau)$ and $O_{MUS}(\tau)$. The I_{EF} / O_{EF} of the EF are not visible on the top-level structure of the DESM.

This interface is realized via workspace variables. The EF consists of five components, of which *Parameters*, *Controller* and *Encoder* form the generator *Gen* according to Figure 2. With the exception of the *Encoder*, the components of the EF operate purely signal-oriented.

The *Transducer* monitors the signal-oriented outputs of the MUS, maps the variables of interest O'_{MUS} for this experiment in a vector *yDec=[sSetting, #jobsQ1, #jobsQ2]*, and provides the vector as output variable. Moreover, the *Transducer* generates the SU mapping, by providing the time trajectories of the O'_{MUS} quantities as EF outputs O_{EF} via the data workspace.

The *Acceptor* controls the termination of the simulation after a specified time interval $[\tau_0, \tau_{final}]$ has elapsed. It evaluates the interest variables *#jobsQ1* and *#jobsQ2* and terminates the simulation run abnormally if the difference between the two quantities exceeds a maximum value.

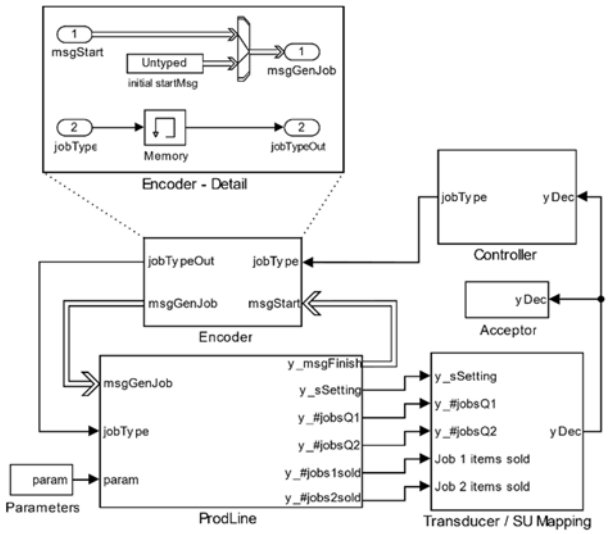

Figure 6. Top-level structure of the DESM with MUS and EF for designing a heuristic control strategy, and substructure of the Encoder block

The *Controller* implements the heuristic strategy. It determines the next entity type to be generated based on the current values of the interest variables received by the Transducer.

The goal is to minimize the difference between the two queue contents (*#jobsQ1* and *#jobsQ2*) while respecting the current tool setting (*sSetting*). The result is passed on as a signal (*jobType*) to the *Encoder*.

The *Encoder* works event-driven (see Figure 6). If the operation server of the MUS is free, it sends an event *y_msgFinish* to the *Encoder*. Thereupon the *Encoder* sends an event *msgGenJob* to the MUS and forwards the signal *jobType* that codes the entity type to be generated. At the start of a simulation run, the *Encoder* generates the initial input event *msgGenJob* and sets the entity type *(jobType=1)* to be generated for the MUS. *Parameters* generates the constant input segments for the MUS vector *param* for initializing the MUS parameters.

This is a simple SBE. The EC defines the parameters P_{SnM} to be varied and directly calls the SimMeth to execute simulation runs.

2.3 Learning a Strategy Using a Self-Implemented Q-Agent

The top-level structure of the DESM for this SBE is shown in Figure 7. The identical MUS named *Prodline* is integrated into an RL-specific EF according to Figure 4. As in the previous experiment, the EF interface (I_{EF}/O_{EF}) is implemented via Workspace variables. *Parameters*, *Agent* and *Encoder* form the generator *Gen*, and *Decoder*, *Reward* as well as *SU Mapping* form the transducer *Trans* (cf. Figure 2). To learn a strategy, the agent requires unique state-action tuples (s_t, a_t) as well as associated next state s_{t+1} and reward values r_{t+1}. Hence, the two time bases t and τ were introduced in Subsection 1.3 for the EF and the MUS. Accordingly, the components of the EF are implemented event-oriented, with the exception of *Parameters* and *SU Mapping*. The component *Parameters* is identical to the previous experiment.

At simulation start time τ_0, an episode is started by the *Agent* sending an event *msgGenJob* and setting an action value $a_t=\{1|2\}$ at the output port *action*. 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*.

When an entity has completed on the operation unit, an output event *y_msgFinish(τ)* is sent from the MUS to activate the *Decoder* and study-specific *output data(τ)* is passed signal-oriented to the *SU_Mapping* for trajectory recording.

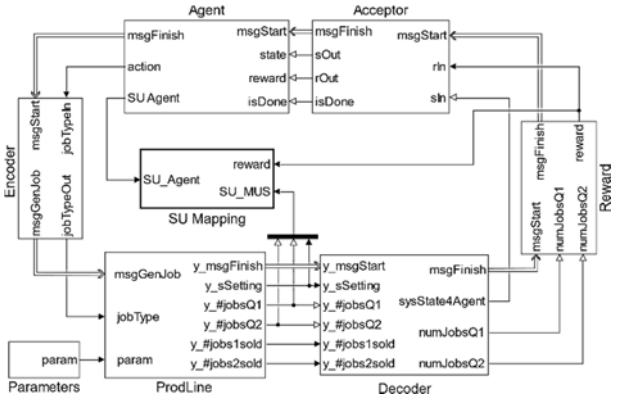

Figure 7. Top-level structure of the DESM with MUS and EF for learning a strategy using a self-implemented agent component.

The *Decoder* selects the information relevant to the RL from the *MUS outputs(τ)* and calculates the new state s_{t+1} of the RL space. To limit the RL space, the Decoder truncates the two queue contents (*#jobsQ1* and *#jobsQ2*) to a maximum length. The new state s_{t+1} is thus calculated from the two limited queue contents and the current tool setting (*sSetting*) of the operating unit, and output at the port *sysState4Agent*.

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 interest variables $O'_{MUS}(t+1)$, in this case *#jobsQ1* and *#jobsQ2*. In terms of content, both approaches are identical but the second one resulted in a much better structured reward computation.

After the reward calculation, the *Acceptor* is activated by an event *msgFinish*. In this experiment, no constraints are defined for s_{t+1} and r_{t+1}, so they are only passed to the *Agent*. Only a control segment is defined for the simulation time interval $[\tau_0, \tau_{final}]$, which specifies the length 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 (*eoe*) has been reached or not.

The *Agent* evaluates the boolean *isDone* value. If *isDone* is false, it executes its learning rules, calculates a new *action* value, and generates an output message *msgFinish* to activate the *Encoder*. In case of *isDone* is true it performs a data update *SU_Agent(eoe)* and the episode is terminated by the *Acceptor*.

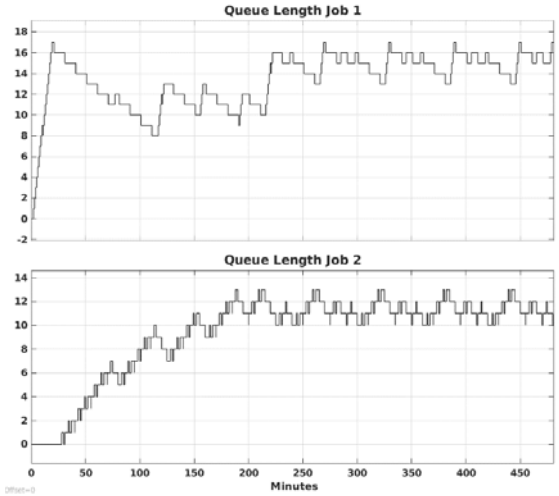

Figure 8. Time trajectories of the queue lengths computed using the learned strategy of a self-implemented Q-agent after 5000 episodes.

This is a *complex SBE*. The EC defines the parameters P_{ExM} and P_{SnM} (cf. Fig. 1), and calls an ExpMeth *training* as described in Subsection 1.3, which calls the SimMeth in a loop to execute the DESM for one episode at a time. Figure 8 shows the simulation results of the MUS using a learned control strategy after 5000 episodes.

The event-oriented implementation of the EF components was done using SimEvents' *Discrete Event Charts*, which call MATLAB functions. This makes the algorithms of the components, such as the learning approach of the agent, easily interchangeable. In Pawletta and Bartelt [6], the algorithms of this experiment are presented in more detail and the full implementation is available on Github (FG CEA [4]).

2.4 Learning a Strategy Using a Dedicated RL Toolbox

The MathWorks offers a dedicated RL toolbox for MATLAB/Simulink (MathWorks [10]). This provides an agent block for Simulink, which is configured from MATLAB. Different learning approaches can be configured in the form of agent types as well as hyperparameter settings. Furthermore, the toolbox provides different methods, such as a training method called *train*. In this experiment we use the Q-learning agent and the *train* method of the RL toolbox. It must be noted that the documentation of the toolbox does not contain any hints or examples for the use with event-oriented MUS implemented with the SimEvents blockset. According to the documentation, the agent block of the RL toolbox works signal-oriented.

A signal is in Simulink a time-varying quantity that has values at all points in time. Accordingly, the agent block is designed for continuous or discrete-time models with equidistant sampling.

The top-level structure of the DESM for this SBE is shown in Figure 9. With the exception of the *Triggered Agent* block, the DESM corresponds completely to the DESM in Figure 7, i.e., all other components of the EF as well as the MUS were adopted unchanged. Hence, only the *Triggered Agent* is discussed below.

The *Triggered Agent*, implements a so-called triggered subsystem and encapsulates the RL agent block of the toolbox. The inner structure of the *Triggered Agent* and the input/output interface of the encapsulated RL agent are also shown in Figure 9. Analogous to the *Agent* in the previous model (cf. Figure 7), the *Triggered Agent* is activated by the *Acceptor* per event (*msgStart*) when a new state s_{t+1} of the RL space (input port *sIn*) and a new reward value (input port *rIn*) as well as the boolean *isDone* value have been calculated. If *isDone* is false, the encapsulated RL agent calculates the next *action value* a_t as well as updates the *cumulative reward value*, and then the *Triggered Agent* activates the *Encoder* by event (*msgFinish*).

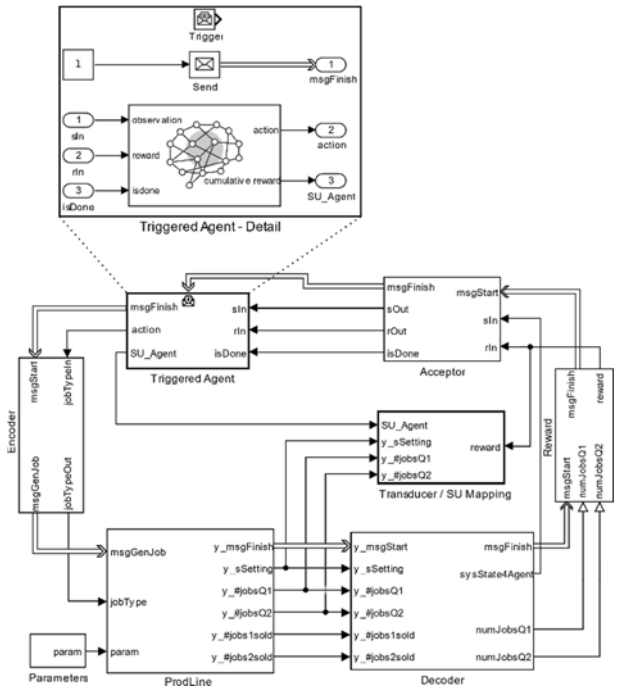

Figure 9. Top-level structure of the DESM with MUS and EF for learning a strategy using the RL Toolbox of MATLAB/Simulink, and substructure of the Triggered Agent.

In the other case, if *isDone* is true, the encapsulated RL agent computes the complete data update of the episode *SU_Agent(eoe)* and, in contrast to the previous model, immediately terminates the episode. I.e., the termination of the episode by the acceptor according to the previous example is skipped.

The EC defines the parameters P_{ExM} and P_{SnM}, and calls the RL toolbox specific ExpMeth *train*, which uses an RL toolbox specific SimMeth.

3 Conclusions

The integration of discrete-event 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 that the MUS, EF, SimMeth, simulator and AI methods – as ExpMeth – can be developed independently and reused in different contexts. The methodological considerations have been practically underpinned by a case study implemented with MATLAB/Simulink and the SimEvents blockset.

In particular, the three experiments of the case study demonstrate that MUS can be developed independently of their experimental context. As shown, this is also true when integrating with the RL method. The adaptation to a concrete experiment can be done by a specific EF, higher-level ExpMeths and a supervisory EC. The basic structure of an EF and the communication relationships in SBE using the RL method were presented.

SBEs in combination with the RL method are characterized by a large number of methodological parameters and variants of agents. Accordingly, the specification of such experiment variants and their automated execution based on the System Entity Structure and Model Base (SES/MB) approach will be investigated in a next step.

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

[3] 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.

[4] FG CEA. Integration of RL and Discrete Event Simulation: A Case study using MATLAB/ Simulink/ SimEvents, Wismar Univ. of Applied Sciences Wismar, 2022, *https://github.com/cea-wismar*.

[5] Pawletta T. Specification and execution of simulation models and experiments. *Discussion talk at 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.*

[6] Pawletta T, Bartelt J. Integration of Reinforcement Learning and Discrete Event Simulation Using the Concept of Experimental Frame. In Mota M.M., editor. *Eurosim Congress 2023*; 2023 Jul; Amsterdam, Netherlands. 8 pages (submitted 2022-Dec-30).

[7] Schmidt A. Variantenmanagement in der Modellbildung und Simulation unter Verwendung des SES/MB Frameworks (Variant Management in Modeling and Simulation using the SES/MB Framework). *Advances in Simulation – Fortschritssberichte Simulation*, Bd. 30, ARGESIM Publisher, Vienna. ISBN ebook 978-3-903347-30-4. DOI 10.11128/fbs.30

[8] Sutton RS, Barto, AG. *Reinforcement Learning: an Introduction – 2nd edition*. 2018, MIT Press.

[9] The MathWorks (2022-1). SimEvents. *https://mathworks.com/simevents/reinforcement-learning.html*, ©1994-2022 The MathWorks, Inc.

[10] The MathWorks. Reinforcement Learning Toolbox. (2022-2); *https://mathworks.com/products/reinforcement-learning.html*, ©1994-2022 The MathWorks, Inc.

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

[12] Zeigler BP. *Theory of Modeling and Simulation – 1st edition*. 1976, John Wiley & Sons.

[13] Zeigler BP. *Multifacetted Modelling and Discrete Event Simulation*. 1984, Academic Press.

[14] Zeigler BP, Muzy A, Kofman E. *Theory of Modeling and Smulation – 3rd edition*. 2018, Elsevier, Academic Press.