A DEVS-Based Framework for Integrating Reinforcement Learning Agents into Discrete Event Simulation Environments

Ezequiel Beccaria[1], Verónica Bogado[2], and Jorge A. Palombarini[1,2]

[1] Facultad Regional Villa María, UTN, Villa María, Cordoba, Argentina
[2] IMITAB - CONICET-UNVM, Villa María, Cordoba, Argentina

Abstract. Discrete event simulation increasingly requires the incorporation of intelligent decision-making agents into simulation models, yet no standardized formal approach exists for doing so within the Discrete Event System Specification (DEVS) formalism. In this paper, we propose a framework that provides reusable, formally specified DEVS building blocks—atomic and coupled models—for constructing simulation environments, formally modeled as Generalized Semi-Markov Decision Processes (GSMDP), that expose standardized interfaces for interacting with Reinforcement Learning (RL) agents. The framework defines a set of composable DEVS components (StateObserver, EventGenerator, Agent) that encapsulate the RL interaction protocol, allowing domain-specific modeling to remain the central focus. We validate our approach through a real-world railway rescheduling case study, demonstrating that three fundamentally different RL algorithms can be evaluated on the same DEVS environment without any modifications to the simulation model.

Keywords: DEVS , Sequential Decision Problems , RL-Integrated Simulation Environments , Reusable DEVS Building Blocks , GSMDP

Un Framework Basado en DEVS para la Integración de Agentes de Aprendizaje por Refuerzo en Entornos de Simulación de Eventos Discretos

Abstract. La simulación de eventos discretos requiere cada vez más la incorporación de agentes inteligentes de toma de decisiones en los modelos de simulación, pero no existe un enfoque formal estandarizado para hacerlo dentro del formalismo de Especificación de Sistemas de Eventos Discretos (DEVS). En este artículo, proponemos un framework que provee building blocks DEVS reutilizables y formalmente especificados—modelos atómicos y acoplados—para construir entornos de simulación

modelados como Procesos de Decisión Semi-Markovianos Generalizados (GSMDP) capaces de interactuar con agentes de Aprendizaje por Refuerzo (RL). El framework define un conjunto de componentes DEVS componibles (StateObserver, EventGenerator, Agent) que encapsulan el protocolo de interacción RL, permitiendo que el modelado específico del dominio permanezca como foco central. Validamos nuestro enfoque mediante un caso de estudio de replanificación ferroviaria del mundo real, demostrando cómo los building blocks propuestos pueden componerse para crear un entorno de simulación DEVS completamente funcional e integrado con tres algoritmos RL fundamentalmente distintos.

Keywords: DEVS , Problemas de Decisión Secuencial , Entornos de Simulación Integrados con RL , Building Blocks DEVS Reutilizables , GSMDP

1 Introduction

Discrete EVent System Specification (DEVS, (Zeigler et al., 2019)) is a well-established formal framework for modeling and simulating complex systems. Based on general systems theory, DEVS has been successfully applied to a wide range of domains, including evaluating software architectures (Bogado et al., 2017) and modeling routing processes (Blas et al., 2022), among others.

Incorporating intelligent decision-making capabilities into DEVS models can extend their applicability to complex *Sequential Decision Problems (SDP)*. Reinforcement Learning (RL, (Arulkumaran et al., 2017)) has emerged as a powerful methodology for learning control policies in such problems, with Deep Reinforcement Learning (DRL) achieving remarkable success in domains such as game playing (Mnih et al., 2015), among others. These advances have created a growing demand for DEVS simulation models that can seamlessly interact with RL agents, leveraging RL capabilities without abandoning the formal guarantees and compositional advantages of DEVS.

However, despite the natural suitability of DEVS for modeling the dynamics, uncertainty, and concurrency inherent in real-world decision problems, the formalism currently lacks standardized components for exposing simulation models to RL agents: no formal mechanisms exist for action reception, state observation, reward signaling, or episode management within DEVS models. While DEVS already provides modular system representation, time as a fundamental modeling element, and asynchronous event processing, no existing work offers reusable, formally specified DEVS components that can be composed to create RL-ready simulation environments.

In this work, we propose a DEVS-based framework that provides formally specified, reusable building blocks for constructing simulation environments capable of interacting with RL agents. The framework models sequential decision

problems as Generalized Semi-Markov Decision Processes (GSMDP) within the DEVS formalism, defining composable atomic models—StateObserver, EventGenerator, and Agent—that encapsulate the RL interaction protocol. The modular and hierarchical nature of DEVS enables these blocks to be composed into environments of varying complexity, keeping the focus on domain-specific modeling while the framework handles the RL integration infrastructure. A case study on a real-world railway rescheduling problem validates the proposal.

While many control problems can be modeled as Markov Decision Processes (MDP, (Puterman, 1990)), complex scenarios involving concurrent uncontrollable stochastic events require Generalized Semi-Markov Decision Processes (GSMDP, (Rachelson et al., 2008)). DEVS provides the formal tools to specify such GSMDP environments within the simulation framework.

The remaining work is structured as follows: Section 2 analyzes related works; Section 3 describes the conceptual model of the RL-integrated simulation framework; Section 4 defines the formal DEVS specification of the proposed building blocks; Section 5 describes the implementation aspects of the framework, which is subsequently validated through the development of a case study in Section 6; Section 7 presents conclusions and outlines future work.

2 Related Work

Several formal approaches have addressed the modeling of stochastic and decision processes within DEVS. *DEVS Markov Models* (Seo et al., 2018) formalize stochastic DEVS as a subclass but do not address the decision dimension. A hierarchical MDP framework within DEVS is proposed in (Kessler et al., 2017), exploiting DEVS compositional structure, yet it does not extend to GSMDPs nor provide reusable building blocks for general RL integration. In (Rachelson et al., 2008), an online simulation-based method for solving GSMDPs is presented, but it adopts an algorithmic perspective without providing reusable DEVS models that can be instantiated for specific domains. None of these works offer composable, formally specified DEVS components for constructing RL-ready simulation environments.

From the RL community, Gymnasium (Towers et al., 2024) provides a standardized API for agent-environment interaction, demonstrating the demand for such interfaces. However, it operates outside any formal simulation framework and provides no mechanisms for integrating RL agents with pre-existing DEVS models.

In (David & Syriani, 2022), RL is used to create DEVS models that replicate a target system's behavior, rather than providing building blocks for RL-integrated environments. Finally, a methodological framework for RL-DEVS integration was proposed in (Beccaria et al., 2021), focusing on "what" to do (activities, inputs/outputs, roles). In contrast, the present work focuses on "how"—providing formally specified DEVS building blocks that can be composed to construct RL-ready simulation environments for GSMDPs.

3 Conceptual Model of the RL-Integrated Simulation Framework

Figure 1 depicts the architecture of the proposed framework. The DEVS simulation environment models the control problem as a GSMDP, where state transitions are driven by both the agent's actions (a_t) and exogenous events (e_t), producing observations (s_t) and rewards (r_t). The RL Agent is a pluggable component that interacts with the simulation through standardized DEVS ports. This separation provides a modular architecture: different RL algorithms can be plugged into the same DEVS model and the same agent can interact with different environments, without modifying the simulation model.

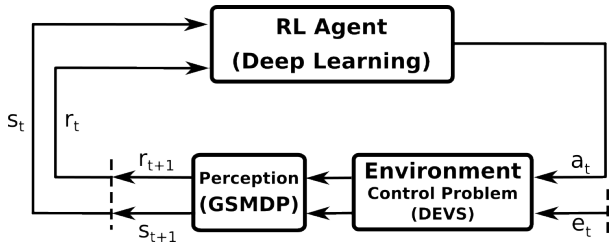

Fig. 1. Architecture of the proposed DEVS-based framework for RL-integrated simulation.

While Fig. 1 presents the high-level agent–environment interaction loop, the conceptual model in Fig. 2 details the entities and packages that realize this architecture. The proposed framework is organized around a set of conceptual entities that define how a DEVS simulation environment interacts with RL agents. These concepts and their relationships are structured into three packages: *Environment* (the core contribution), *Agent*, and *Training*.

The *Environment* package is the core of the framework, grouping all the required functionality to model and simulate a GSMDP as the simulation environment. The *GSMDP* concept is associated with a *TransitionFunction*, which manages state transitions of the environment given the occurrence of one of the available transitions, represented with the *Transition* concept, in the current state. The *State* concept represents all possible states in which the environment can be found, and the *Event* concept captures events that can occur in the environment. In a GSMDP, there are two important specialized events: *Exogenous* events, which are activated following a given distribution, or *Action* events, which are activated by the agent. In some states, certain transitions may be prohibited or banned. This capacity of prohibiting or allowing system transitions is managed by the *EventFunction* concept. The *RewardFunction* concept generates the reward signal sent to the Agent given a state transition. These concepts define the interface that must be implemented for each specific domain.

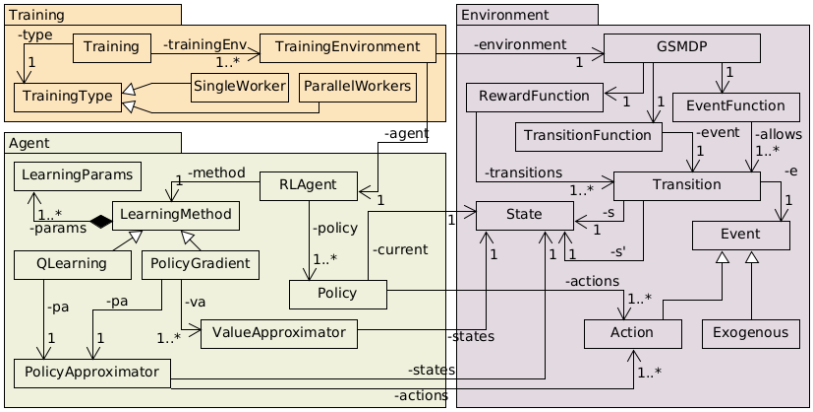

Fig. 2. Conceptual model of the RL-integrated DEVS simulation framework.

The *Agent* package defines the pluggable RL component that interacts with the simulation environment. The *RL Agent* concept is associated with a *LearningMethod* (with configurable *LearningParam*) to learn a *Policy* that defines which action to take according to the current environment state. The framework supports both value-based methods (*QLearning* with a *PolicyApproximator*) and policy gradient methods (with a *ValueApproximator*), using deep learning approximators that enable RL in high-dimensional state and action spaces, and can be used as-is with different RL algorithms.

The *Training* package represents the deployment configurations for the framework. Two configurations are supported: *SingleWorker*, where one agent interacts with one simulation environment (e.g., DDQN architecture, (van Hasselt et al., 2016)), or *ParallelWorkers*, where multiple agents interact with several simulation environments simultaneously (e.g., A3C (Mnih et al., 2016) or PPO (Schulman et al., 2017) architectures). These configurations determine how the environment is instantiated and orchestrated, not the environment model itself.

4 Formal Model

Based on the concepts defined in Section 3, a formal DEVS model is defined (Figure 3). The *SimulationEnvironment* coupled model has two main components: the Agent and the GSMDP. The *GSMDP* model is a Coupled DEVS composed of components that model the control problem, exposing ports “action" (in) and “step" (out) for agent interaction.

Regarding its relationship with Fig. 1: while the agent is conceptually a pluggable block interacting with the environment, for simulation purposes it is coupled *within* the *SimulationEnvironment* model so the whole system runs under a single DEVS simulator. This structuring choice preserves the logical separation, as agent and GSMDP still communicate exclusively through the “action" and

"step" ports, allowing different RL algorithms to be swapped without modifying the GSMDP.

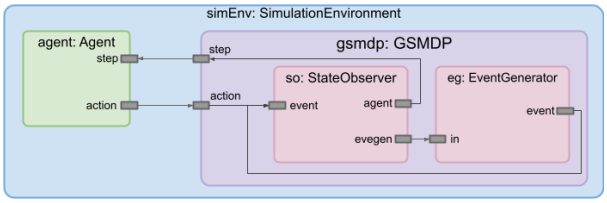

Fig. 3. DEVS coupled model architecture for RL-integrated simulation.

Its core component is the *stateObserver* –Equation (1)– a reusable Parallel Atomic DEVS model that maintains the global state, updating it upon arrival of exogenous events or agent actions via the "event" input port. It is instantiated with domain-specific implementations of the transition function (tf), reward function (rf), enabled actions function ($enabledA$), and active events function ($activeE$). Its output ports "agent" and "generator" send the current state to the agent and activate/passivate *EventGenerator* components, respectively.

$$
\begin{aligned}
StateObserver &= (X, Y, S, ta, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda) \\
Event &= \{event_i | i = 1, \ldots, n\} \\
InPorts &= \{\text{“}event\text{”}\};\ X_{event} = Event \\
X &= \{(p, v) \mid p \in InPorts, v \in X_p\} \\
OutPorts &= \{\text{“}generator\text{”}, \text{“}agent\text{”}\} \\
Y_{generator} &= \{true, false\} \\
A &= \text{An environment action set} \\
Y_{agent} &= \{(o, r, d, A_o) \mid o \in \mathbb{R}^*, r \in \mathbb{R}, d \in \{true, false\}, A_o \subseteq A\} \\
Y &= \{(p, v) \mid p \in OutPorts, v \in Y_p\} \\
S &= \{(o, r, \sigma) \mid o \in \mathbb{R}^*, r \in \mathbb{R}, \sigma \in \mathbb{R}_0^+\} \\
tf &: \mathbb{R}^* \times Event \to \mathbb{R}^*;\ done: \mathbb{R}^* \to \{true, false\} \\
rf &: \mathbb{R}^* \times Event \times \mathbb{R}^* \times \mathbb{R}_0^+ \to \mathbb{R} \\
enabledA &: \mathbb{R}^* \to \{a \mid a \subseteq A\};\ activeE: \mathbb{R}^* \to \{true, false\} \\
\delta_{ext}(o, r, \sigma, e, (p, v)) &= \begin{cases} (tf(o, v), rf(o,\ v,\ tf(o, v),\ e), 0) \\ \qquad \text{if } p = \text{“}event\text{”} \wedge v \in X_p \\ (o, r, \sigma - e) \quad \text{otherwise} \end{cases} \\
\delta_{int}(o, r, \sigma) &= (o, 0, \infty) \\
\delta_{con}(o, r, \sigma, e, x) &= \delta_{ext}(\delta_{int}(o, r, \sigma), 0, x) \\
\lambda(o, r, \sigma) &= \begin{cases} (\text{“}generator\text{”}, activeE(o)) \\ (\text{“}agent\text{”}, \{o, r, done(o), enabledA(o)\}) \end{cases} \\
ta(s) &= \sigma
\end{aligned}
\tag{1}
$$

The state $s = (o, r, \sigma) \in S$ holds the current observation o, the last reward r, and the time advance σ. Upon receiving an event, the model computes the new state via tf, the reward via rf, determines available actions ($enabledA$), activates/passivates event generators ($activeE$), and checks for terminal states ($done$).

The *EventGenerator* –Equation (2)– is a reusable Parallel Atomic DEVS model that generates time-dependent exogenous events, one per type of concurrent stochastic event. It alternates between passive and active phases: when activated via the “in" port, it samples the next event time from a probability distribution and outputs the event through the “out" port. Although the specification uses a normal distribution for illustration, the sampling mechanism is encapsulated within the atomic model, allowing it to be replaced by any probability distribution (e.g. exponential, Poisson, or empirical) without affecting the rest of the model.

$$
\begin{aligned}
EventGenerator &= (X, Y, S, ta, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda) \\
NextEventSigma &\sim \mathcal{N}(eventMean, eventStd) \\
Event &= \{event_i | i = 1, \ldots, n\} \\
EvtActivation &= \text{An arbitrary set} \\
InPorts &= \{\text{“}in\text{”}\};\ X_{in} = EvtActivation \\
X &= \{(p, v) \mid p \in InPorts, v \in X_p\} \\
OutPorts &= \{\text{“}out\text{”}\};\ Y_{out} = Event \\
Y &= \{(p, v) \mid p \in OutPorts, v \in Y_p\} \\
S &= \{(phase, ev, \sigma) \mid phase \in \{\text{“}passive\text{”}, \text{“}active\text{”}\}, ev \in Event, \sigma \in \mathbb{R}_0^+\} \\
s_0 &= (\text{“}passive\text{”}, null, \infty) \\
\delta_{ext}(phase, ev, \sigma, e, (\text{“}in\text{”}, v)) &= \begin{cases} (\text{“}active\text{”}, ev, NextEventSigma) \\ \quad \text{if } phase = \text{“}passive\text{”} \wedge v \in X_p \wedge v = 1 \\ (\text{“}active\text{”}, ev, \sigma - e) \\ \quad \text{if } phase = \text{“}active\text{”} \wedge v \in X_p \wedge v = 1 \\ (\text{“}passive\text{”}, ev, \infty) \quad \text{otherwise} \end{cases} \\
\delta_{int}(phase, ev, \sigma) &= \begin{cases} (\text{“}active\text{”}, ev, NextEventSigma) \\ \quad \text{if } phase = \text{“}active\text{”} \\ (\text{“}passive\text{”}, ev, \infty) \quad \text{otherwise} \end{cases} \\
\delta_{con}(phase, ev, \sigma, e, x) &= \delta_{ext}(\delta_{int}(phase, ev, \sigma), 0, x) \\
\lambda(phase, ev, \sigma) &= (\text{“}out\text{”}, ev) \quad \text{if } phase = \text{“}active\text{”} \\
ta(s) &= \sigma
\end{aligned}
\tag{2}
$$

The *Agent* component –Equation (3)– is a pluggable Parallel Atomic DEVS model that wraps any RL algorithm behind a standardized DEVS interface. It receives a *Step* tuple (current state o, reward r, done flag, and allowed actions A_o) via the “step" port, processes it through the *observation* function to select a control action, and outputs it via the “action" port.

$$
\begin{aligned}
Agent &= (X, Y, S, ta, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda) \\
Event &= \{event_i | i = 1, \ldots, n\} \\
A &= \text{An environment action set} \\
Step &= \{(o, r, done, a) \mid o \in \mathbb{R}^*, r \in \mathbb{R}, \\
&\qquad done \in \{true, false\}, a \subset A\} \\
InPorts &= \{\text{“}step\text{”}\};\ X_{step} = Step \\
X &= \{(p, v) \mid p \in InPorts, v \in X_p\} \\
OutPorts &= \{\text{“}action\text{”}\};\ Y_{action} = A \\
Y &= \{(p, v) \mid p \in OutPorts, v \in Y_p\} \\
S &= \{(ev, \sigma) \mid ev \in Event, \sigma \in \mathbb{R}_0^+\}; \qquad s_0 = (null, \infty) \\
obs &: Step \to A \\
\delta_{ext}(ev, \sigma, e, (p, v)) &= \begin{cases} (obs(v), 0) & \text{if } p = \text{“}step\text{”} \\ (ev, \infty) & \text{otherwise} \end{cases} \\
\delta_{int}(ev, \sigma) &= (null, \infty) \\
\delta_{con}(ev, \sigma, e, x) &= \delta_{ext}(\delta_{int}(ev, \sigma), 0, x) \\
\lambda(ev, \sigma) &= (\text{“}action\text{”}, ev) \quad \text{if } ev \neq null \\
ta(s) &= \sigma
\end{aligned}
\tag{3}
$$

5 Framework Implementation

In Fig. 4 the design diagram of the RL-integrated simulation environment is defined. The implementation of the presented framework, as well as the programming of the experiments, was carried out using *Java* as the programming language.

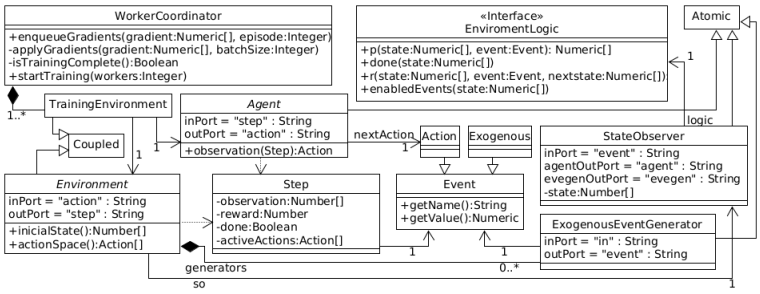

Fig. 4. Design model of the RL-integrated DEVS simulation framework.

The design model targets the *DEVSJAVA*[3] simulator engine. Its components are organized into two groups: domain-specific components, configured for each application, and framework-provided components, used as ready-to-use infrastructure.

The domain-specific components include the *Environment* class, which extends a Coupled DEVS class establishing the required ports ($InPorts$=$\{$“$action$”$\}$

[3] https://sourceforge.net/projects/devs-suitesim/

and $OutPorts$={"*step*"}), and the *Event* class, representing action events (controlled by the agent) and exogenous events (uncontrollable).

The framework-provided components are used as-is: the *Agent* abstract class extends Atomic DEVS with RL interaction ports and includes pre-built implementations for DDQN, A3C, and PPO; the *SimulationEnvironment* class serves as the Coupled DEVS container for the whole model; the *Step* class carries observation, reward, and status information from the environment to the agent; and the *WorkerCoordinator* class manages global approximator updates for parallel deployments (A3C, PPO).

The *Environment* class can be implemented either as a monolithic atomic DEVS model holding all the simulation behavior, or as a coupled DEVS model built from related components (the design in Fig. 4), where the *StateObserver* class realizes the formal model of Section 4 (Eq. 1) and the *ExogenousEventGenerator* instances trigger exogenous events following a statistical distribution. To obtain a fully functional RL-integrated environment, the only domain-specific code required is an implementation of the *EnvironmentLogic* interface plus the configuration of the control-problem exogenous events; the rest of the RL integration infrastructure is provided by the framework. The *EnvironmentLogic* interface defines four domain-specific methods: the reward function "r" (reward for the last transition), the transition function "p" (state transition from s to s'), the enabled events function "enabledEvents" (activation/passivation of event generators based on s'), and the termination function "done" (episode end condition based on s').

6 Case Study

To validate the proposed framework, we use the rescheduling problem presented in (Šemrov et al., 2016), called *A simple three-stop railway*, following the methodology in (Beccaria et al., 2021). In this control problem, an agent decides train departure times to avoid delays and deadlocks. The layout consists of three stations (A, B, C) connected by a single-track railway divided into five block sections, each with capacity for one train, and stations hold up to three trains simultaneously. The state and event spaces are detailed in Tables 1 and 2.

Table 1. Simple Three-Stop Railway Environment State Space

Variable	Description
$pos_i \in [0, 15200] \subset \mathbb{R}$	Position of train i in meters ($i \in \{1, 2, 3\}$)
$speed_i \in [-27, 27] \subset \mathbb{R}$	Speed of train i in m/s
$trains_j \in \{0, 1, 2, 3\}$	Number of trains in block section j ($j \in \{1, \ldots, 8\}$)
$available_j \in \{0, 1\}$	Availability of block section j
$t \in [0, 3000] \subset \mathbb{R}^+$	Simulation time in seconds

Three event types occur in this environment (Table 2): uncontrollable train arrivals, and two agent-controlled actions—adding delay to a train's departure and No Operation (NOP).

Table 2. Simple Three-Stop Railway Environment Event Space

Event	Action	Event Description
$arrival_i$	No	This event is triggered every time that one of the running trains arrives at one of the stations.
$add_i(m)$	Yes	m minutes added to departure time of the subsequent train i.
NOP	Yes	No OPeration action.

The reward function (Equation 4) penalizes each delay action by its magnitude m, encouraging minimal delays. At simulation end ($s'_t = 3000$), a penalty of -1000 is applied for each train that failed to reach its destination position obj_i (i.e., $pos_i \neq obj_i$), whether due to deadlock or excessive delay.

$$r(add_i(m), s') = \begin{cases} -m & \text{if } s'_t < 3000 \\ -\sum_{i=1}^{3} (s'_{pos_i} \neq obj_i) * 1000 & \text{otherwise} \end{cases} \tag{4}$$

6.1 Experimental Setup

Using the proposed building blocks, the simulation environment was composed for this problem (Fig. 5). The *EnvironmentLogic* interface was implemented for the railway domain, and the environment was composed with one *StateObserver* and three *ExogenousEventGenerator* components (one per train arrival event), plus an additional generator (*FinalEvent*) ensuring a final reward perception at simulation end. The *Agent* component was then coupled to the environment, completing the RL-integrated simulation model.

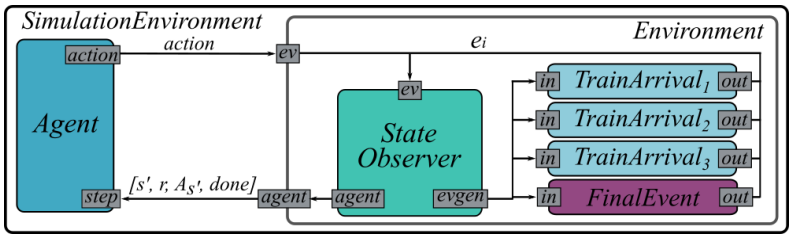

Fig. 5. DEVS simulation environment composed from the proposed building blocks, with its components and their couplings.

Following (Šemrov et al., 2016), the timetable defines three trains with speeds of 70 and 100 km/h. At each episode start, a random train receives a uniform delay of 7–10 minutes that, without intervention, causes a deadlock. The agent must learn to reschedule departures while minimizing total delay.

To evaluate the modularity of the framework, three different pre-built RL agent implementations were selected, using the DeepLearning4j[4] library: *DDQN*, *A3C*, and *PPO*. Notably, switching between these fundamentally different RL architectures (value-based, actor-critic, and policy gradient) required no changes to the DEVS simulation environment—only the Agent component was swapped.

Each agent was trained for 10000 episodes (3000 steps each) across five sessions with different random seeds, using the hyper-parameters from the original papers. The implementation[5] uses Java, executed on an AMD Ryzen 7 1800 (8 cores, 32 GB RAM).

To validate the simulation environment, traces were compared against an equivalent Gymnasium implementation using Dynamic Time Warping (DTW) (Bringmann et al., 2024) across four scenarios (combining no/random action with no/added delay), yielding a mean distance of $\mu = 0.31$ ($\sigma = 1.51$). The high standard deviation reflects variability across scenarios with different event densities, yet all individual distances stayed below thresholds consistent with equivalent qualitative behavior, confirming equivalent dynamics between both implementations.

6.2 Computational Results

Subsequently, to demonstrate the framework's ability to support the evaluation of multiple RL algorithms on the same simulation environment, the policy performance learned by each agent (A3C, DDQN, PPO) is compared across twelve different testing scenarios that introduce from 7 to 10 minutes delay on each train. In table 3, the performance of each agent in each of the predefined test scenarios can be visualized. The numbers in bold indicate which agent obtained the best performance in a given scenario.

The results demonstrate the framework's key advantage: the same DEVS simulation environment was used to evaluate three fundamentally different RL architectures without any modifications to the simulation model. The Mean and Std rows in Table 3 summarize performance over five independent runs with different random seeds, indicating robustness across runs.

The learning curves in Fig. 6 explain the observed differences. A3C converges the fastest and remains the most stable, consistent with its lowest test variance. PPO also converges to near-optimal reward but with a noisier trajectory, while DDQN is still improving when the training budget is exhausted, indicating it has not yet converged. This accounts for its higher variance: the differences reflect convergence speed under a fixed budget of 10,000 episodes rather than asymptotic policy quality.

[4] https://deeplearning4j.konduit.ai

[5] https://github.com/ezequielbeccaria/RLDEVS4j-SingleTrackRailway

Table 3. Random Delay Scenario Test Results

Scenario	DDQN	A3C	PPO
T1-7D	-420	-120	**-0**
T2-7D	**-240**	-360	-3360
T3-7D	-420	-480	**-300**
T1-8D	-480	-120	**-0**
T2-8D	**-240**	-360	-360
T3-8D	-3360	-480	**-300**
T1-9D	-600	-120	**-60**
T2-9D	**-360**	**-360**	**-360**
T3-9D	-3420	-480	**-300**
T1-10D	-3240	**-120**	**-120**
T2-10D	-840	**-360**	**-360**
T3-10D	**-360**	-480	-3360
Mean	-1165.0	**-320.0**	-740.0
Std	1265.57	**149.66**	1179.15

7 Conclusions and Future Work

In this work, a DEVS-based framework has been proposed for constructing RL-ready simulation environments for complex sequential decision problems. It provides reusable, formally specified DEVS building blocks—StateObserver, EventGenerator, and Agent—that encapsulate the RL interaction protocol, support both single-worker and parallel deployment configurations, and enable the evaluation of multiple RL algorithms on the same simulation environment without modifications. The framework was implemented and validated on the railway rescheduling problem *A simple three-stop railway*, where multiple concurrent stochastic processes compete to alter the system state, akin to GSMDP-like environments. By eliminating the need to implement RL integration infrastructure from scratch, it shifts the focus towards modeling aspects such as the stochastic concurrent processes and reward functions of the domain. Moreover, the building blocks can be incorporated into pre-existing DEVS models, enabling the integration of RL agents into already validated simulation environments without redesign.

Future work includes incorporating formal verification techniques to validate the correctness and consistency of the constructed environments, and exploring generative AI models to automatically generate new environment configurations from existing ones. Additionally, a bridge layer or a Python-based reimplementation of the building blocks would improve interoperability with the dominant Python-based RL ecosystem. Finally, evaluating the framework on more complex scenarios—with larger state spaces, higher event concurrency, or multi-agent coordination—would further validate its scalability.

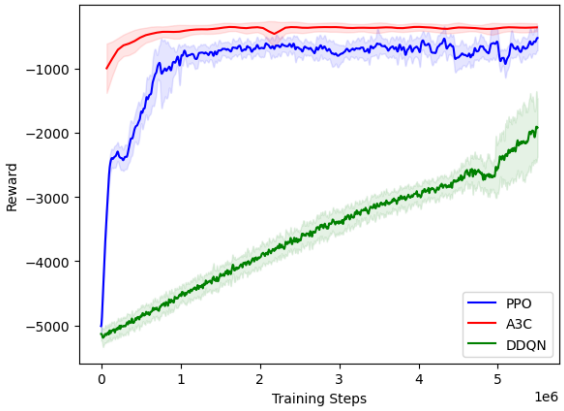

Fig. 6. Training reward curves (mean ± standard deviation over five random seeds) for DDQN, A3C, and PPO.

References

Arulkumaran, K., Deisenroth, M. P., Brundage, M., & Bharath, A. A. (2017). Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, *34*(6), 26–38. https://doi.org/10.1109/MSP.2017.2743240

Beccaria, E., Bogado, V., & Palombarini, J. A. (2021). A devs based methodological framework for reinforcement learning agent training. *IEEE Latin America Transactions*, *19*(4), 679–687. https://doi.org/10.1109/TLA.2021.9448551

Blas, M. J., Leone, H., & Gonnet, S. (2022). Devs-based formalism for the modeling of routing processes. *Software and Systems Modeling*, 1179–1208. https://doi.org/10.1007/s10270-021-00928-4

Bogado, V., Gonnet, S., & Leone, H. (2017). Devs-based methodological framework for multi-quality attribute evaluation using software architectures. *2017 XLIII Latin American Computer Conference (CLEI)*, 1–10. https://doi.org/10.1109/CLEI.2017.8226435

Bringmann, K., Fischer, N., van der Hoog, I., Kipouridis, E., Kociumaka, T., & Rotenberg, E. (2024). Dynamic dynamic time warping. *Proceedings of the 2024 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 208–242.

David, I., & Syriani, E. (2022). Devs model construction as a reinforcement learning problem. *2022 Annual Modeling and Simulation Conference (ANNSIM)*, 30–41. https://doi.org/10.23919/ANNSIM55834.2022.9859369

Kessler, C., Capocchi, L., Santucci, J.-F., & Zeigler, B. (2017). Hierarchical Markov decision process based on DEVS formalism. *2017 Winter Simulation Conference (WSC)*, 1001–1012. https://doi.org/10.1109/WSC.2017.8247850

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., & Kavukcuoglu, K. (2016, 20–22 Jun). Asynchronous methods for deep reinforcement learning. In M. F. Balcan & K. Q. Weinberger (Eds.), *Proceedings of the 33rd international conference on machine learning* (pp. 1928–1937, Vol. 48). PMLR. https://doi.org/10.48550/arXiv.1602.01783

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., & Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, *518*(7540), 529–533. https://doi.org/10.1038/nature14236

Puterman, M. L. (1990). Markov decision processes. *Handbooks in operations research and management science*, *2*, 331–434. https://doi.org/10.1016/S0927-0507(05)80172-0

Rachelson, E., Quesnel, G., Garcia, F., & Fabiani, P. (2008). A Simulation-based Approach for Solving Generalized Semi-Markov Decision Processes. *Frontiers in Artificial Intelligence and Applications*, 583–587. https://doi.org/10.3233/978-1-58603-891-5-583

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. https://doi.org/10.48550/arXiv.1707.06347

Šemrov, D., Marsetič, R., Žura, M., Todorovski, L., & Srdic, A. (2016). Reinforcement learning approach for train rescheduling on a single-track railway. *Transportation Research Part B: Methodological*, *86*, 250–267. https://doi.org/10.1016/j.trb.2016.01.004

Seo, C., Zeigler, B. P., & Kim, D. (2018). DEVS Markov Modeling and Simulation: Formal Definition and Implementation. *Proceedings of the Theory of Modeling and Simulation Symposium*, 1:1–1:12. https://doi.org/10.1145/3213187.3213188

Towers, M., Kwiatkowski, A., Terry, J., Balis, J. U., De Cola, G., Deleu, T., Goulão, M., Kallinteris, A., Krimmel, M., KG, A., et al. (2024). Gymnasium: A standard interface for reinforcement learning environments. *arXiv preprint arXiv:2407.17032*.

van Hasselt, H., Guez, A., & Silver, D. (2016). Deep reinforcement learning with double q-learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, *30*(1). https://doi.org/10.1609/aaai.v30i1.10295

Zeigler, B. P., Muzy, A., & Kofman, E. (2019). Chapter 1 - introduction to systems modeling concepts. In B. P. Zeigler, A. Muzy, & E. Kofman (Eds.), *Theory of modeling and simulation (third edition)* (Third Edition, pp. 3–25). Academic Press. https://doi.org/10.1016/B978-0-12-813370-5.00009-2