DEVS MODEL CONSTRUCTION AS A REINFORCEMENT LEARNING PROBLEM

Istvan David
Eugene Syriani

Department of Computer Science and Operations Research (DIRO)
Université de Montréal
2920 Chemin de la Tour, Montreal (Quebec), CANADA
istvan.david@umontreal.ca, syriani@iro.umontreal.ca

ABSTRACT

Simulators are crucial components in many software-intensive systems, such as cyber-physical systems and digital twins. The inherent complexity of such systems renders the manual construction of simulators an error-prone and costly endeavor, and automation techniques are much sought after. However, current automation techniques are typically tailored to a particular system and cannot be easily transposed to other settings. In this paper, we propose an approach for the automated construction of simulators that can overcome this limitation, based on the inference of Discrete Event System Specifications (DEVS) models by reinforcement learning. Reinforcement learning allows inferring knowledge on the construction process of the simulator, instead of inferring the simulator itself. This, in turn, fosters reuse across different systems. DEVS further improves the reusability of this knowledge, as the vast majority of simulation formalisms can be efficiently translated to DEVS. We demonstrate the performance and generalizability of our approach on an illustrative example implemented in Python and Tensorforce.

Keywords: DEVS, reinforcement learning, machine learning, automation.

1 INTRODUCTION

Nowadays, engineered systems have reached a previously unprecedented complexity. Systems are becoming faster, safer, more autonomous, reliable, and durable. To cope with this complexity, engineering such systems is best approached through modeling and simulation. However, the complexity and inherent heterogeneity of these systems render their manual modeling an error-prone, time-consuming, and costly endeavor. This challenge is exacerbated by the role of simulation gradually shifting from the design phase to operation since the early 2000s, and simulators becoming first-class components in today's complex systems (Boschert and Rosen 2016). Automation of model construction can significantly alleviate these problems.

To support the automation of constructing simulation models of complex systems, we propose an approach for the automated inference of atomic discrete event system specification (DEVS) models (Zeigler et al. 2018) by reinforcement learning, outlined in our previous work (David et al. 2021).

The effort and costs associated with practical machine learning can only be justified if the inferred artificial intelligence is versatile enough to cover a sufficiently large set of features that complex systems might possess. DEVS is an appropriate formalism for this purpose, as it allows modeling the reactive behavior of real systems in great detail, including timing and interactions with the environment—two aspects predominantly present in complex systems such as cyber-physical systems and digital twins (Mittal et al. 2019). In addition,

©2022 Society for Modeling & Simulation International (SCS)

DEVS has been shown to be the common denominator of many other simulation formalisms (Vangheluwe 2000), further improving the reusability of the inferred knowledge. Therefore, building an inference framework of DEVS models is a promising solution to maximize the reusability of the approach.

Reinforcement learning (Sutton and Barto 2018) is a particularly appropriate machine learning approach in engineering problems where the set of engineering actions is finite and effectively enumerable, such as in the construction of DEVS models. As a learning approach of the unsupervised kind, reinforcement learning does not require *a priori* labeled data. Instead, the learning agent is provided with the set of engineering actions, and by trial-and-error, the agent learns to organize these actions into sequences of high utility. Relying on an explicitly enumerated set of actions bodes well with well-defined formalisms, such as DEVS. The inferred artificial intelligence—referred to as the *policy*—approximates the tacit knowledge of the domain expert and is able to solve not only the specific problem at hand but other congruent problems as well. Therefore, reinforcement learning can be used to learn how to solve a *class* of problems, rather than learning a solution to a particular case, if provided with a problem at the appropriate level of abstraction.

The main contribution of this paper is the sound mapping of DEVS model construction onto the concepts and mechanisms of reinforcement learning. This mapping allows for learning DEVS model features, such as states, transitions, events, etc. The feasibility and utility of the approach are demonstrated through an illustrative example. Finally, we discuss the main challenges and opportunities.

2 BACKGROUND

In this section, we review the two main topics our work is based on: DEVS and reinforcement learning.

2.1 Discrete Event System Specification (DEVS)

DEVS (Zeigler et al. 2018) is a formalism for specifying discrete event systems and their interactions. An **Atomic DEVS** model is defined as

$$M = \langle X, Y, S, q_{init}, \Delta_{int}, \Delta_{ext}, \lambda, ta \rangle. \tag{1}$$

S is the set of states in the model. $\Delta_{int} : S \to S$ is the internal transition function to the next state. State transitions can be time-based or based on the observation of external events from other models. $ta : S \to \mathbb{R}^{+}_{0,+\infty}$ is the time advance function for each state. It determines the timing of internal state transitions. The model is initialized in the $q_{init} \in Q$ initial state, with $Q = \{(s, e) | s \in S,\ 0 \leq e \leq ta(s)\}$ being the set of total states, i.e., states with the time elapsed in them. X denotes the set of input events the model can react to. $\Delta_{ext} : Q \times X \to S$ is the external transition function to the next state when an event $x \in X$ occurs. Y denotes the set of output events other models can react to. $\lambda : S \to Y$ is the output function.

Additionally, DEVS allows Atomic DEVS models to be recursively composed into **Coupled DEVS** models, allowing arbitrarily complex hierarchies of DEVS models. Based on the notation by Van Tendeloo and Vangheluwe (2017), a Coupled DEVS model is defined as

$$C = \langle X_{self}, Y_{self}, D, \{M\}, \{I\}, \{Z\}, select \rangle. \tag{2}$$

D is the set of model instances that are included in the coupled model. $\{M\}$ is the set of model specifications such that $\forall d \in D \exists M_d : M_d = \langle X, Y, S, q_{init}, \Delta_{int}, \Delta_{ext}, \lambda, ta \rangle_d$, as defined in (1). To connect model instances within a coupled model, $\{I\}$ defines the set of model influences such that $\forall d \in D \exists I_d : d \to D' \subseteq D \setminus \{d\}$. Similarly to Atomic DEVS models, Coupled DEVS models need to define their input and output events to interface with other models too. X_{self} and Y_{self} denote the input and output events, respectively.

Due to the independence of models D within the coupled DEVS model, events within the coupled models might happen in parallel. To retain the unambiguous execution semantics, the *select*: $2^D \to D$ tie-breaking function defines which model to treat with priority in case of conflicting models. To foster the reuse of DEVS models, the set of $\{Z\}$ translation functions allows translating the output events of model $d_1 \in D$ onto the input events of $d_2 \in D$, potentially modifying their content.

2.2 Reinforcement Learning

Reinforcement learning is a machine learning technique that aims at learning what actions to apply in specific situations to maximize a numerical reward signal (Sutton and Barto 2018). The learning *agent* explores its environment sequentially, and at each step, carries out an action, observes the change in the environment, and receives a reward signal. Through a repeated sequence of actions, the agent learns the best actions to perform in the different states. The product of the reinforcement learning process is the *policy* π that maps situations to actions. Thus, the policy is defined as $\pi(a|\sigma)$ of choosing action *a* given the exhibited state σ. The learning process is best defined as a Markov Decision Process by:

$$\textit{MDP} = \langle \Sigma, A, P, R \rangle;\ \pi : \Sigma \to A. \tag{3}$$

Σ is the set of states. A the set of actions. $P : Pr(\sigma'|\sigma, a)$ is the probability of state transitions for a specific action. R is a reward function. The reward function is typically cumulative, rendering the behavior of the agent congruent with a Markov chain. However, the martingale property of non-cumulative reward functions can be utilized in uncertain environments (Vadori et al. 2020).

3 DEVS CONSTRUCTION AS A REINFORCEMENT LEARNING PROBLEM

In this section, we formulate the construction of DEVS models as a reinforcement learning problem based on structures (1) and (3). Given the Markov Decision Process $\textit{MDP} = \langle \Sigma, A, P, R \rangle$ and the Atomic DEVS specification $M = \langle X, Y, S, q_{init}, \Delta_{int}, \Delta_{ext}, \lambda, ta \rangle$, the mapping $\textit{MDP} \to M$ is given as follows:

$$\forall \sigma \in \Sigma : \sigma \to M; \tag{4}$$

$$\forall a \in A, \sigma_k, \sigma_{k+1} \in \Sigma : P_a(\sigma_k, \sigma_{k+1}) \in P \to Pr(\sigma_{k+1}|\sigma_k, a), Pr(\sigma_{k+1}|\sigma_k, a) \mapsto (0, 1); \tag{5}$$

$$R : \Sigma \to \mathbb{R}. \tag{6}$$

Every state $\sigma \in \Sigma$ encodes one DEVS configuration M (4). Actions $a \in A$ are the elementary valid modifications of DEVS models the learning agent can apply, bringing the agent from state σ_i (encoding model M_i) to state σ_j (encoding model M_j) (5). The probability of applying a specific action falls in the range (0, 1). The reward function R maps each state to a metric (6).

3.1 Actions

The actions the learning agent can apply are derived from the elements of the definitions of Atomic DEVS (1) and Coupled DEVS (2). Each action results in a new DEVS configuration, stored as different states of the *MDP*. Thus, the application of action $a \in A$ to the DEVS model M $\textit{apply}(a, M)$ is defined as follows:

$$\begin{aligned} M' &= a(M) \\ \Sigma &\mapsto \Sigma \cup \{\sigma\} \\ \sigma &\mapsto M' \end{aligned} \tag{7}$$

That is, first, the action is applied to the model, resulting in a new model M'. Subsequently, a new state σ is added to the states Σ of the *MDP*. Finally, the new model M' is embedded into the new state σ.

Table 1 lists the actions for constructing Atomic DEVS models. The execution of actions follows (7).

Table 1: Actions the learning agent can execute on Atomic DEVS models

	State actions	
Add DEVS state	$addState(M, s) : M' = M \mid S \cup \{s\},\ ta(s) = \inf$	(8)
Remove DEVS state	$removeState(M, s) : M' = M \mid S \setminus \{s\}$	(9)
	State transition actions	
Add internal transition	$addIntTransition(M, \delta = (s_i, s_j)) : M' = M \mid \Delta_{int} \cup \{\delta\}$	(10)
Remove internal transition	$removeIntTransition(M, \delta \in \Delta_{int}) : M' = M \mid \Delta_{int} \setminus \{\delta\}$	(11)
Add external transition	$addExtTransition(M, \delta = (s_i, s_j)) : M' = M \mid \Delta_{ext} \cup \{\delta\}$	(12)
Remove external transition	$removeExtTransition(M, \delta \in \Delta_{ext}) : M' = M \mid \Delta_{int} \setminus \{\delta\}$	(13)
	Time advance actions	
Set time advance	$updateTa(M, s, t' \in \mathbb{R}^{+}_{0,+\infty}) : M' = M \mid ta(s) = t'$	(14)
	Interacting with other models	
Add output	$addOutput(M, y) : M' = M \mid Y \cup \{y\}$	(15)
Remove output	$removeOutput(M, y \in Y) : M' = M \mid Y \setminus \{y\}$	(16)
Add input	$addInput(M, x) : M' = M \mid X \cup \{x\}$	(17)
Remove input	$removeInput(M, x \in X) : M' = M \mid X \setminus \{x\}$	(18)
	Model initialization	
Set initial state	$setInitial(M, s \in S, 0 \leq e \leq ta(s)) : M' = M \mid q_{init} = (s, e)$	(19)

Additional actions for Coupled DEVS can be defined for (i) coupling, including influence relationships and I/O translation; and (ii) refinement. Table 2 summarizes the actions for Coupled DEVS. Similarly to Atomic DEVS, the execution of Coupled DEVS actions follows (7) as well. Invariant (24) states that applying action (20) requires applying action (21), and applying action (23) requires applying action (22) first.

3.2 Reward Function

The reward function is a crucial element of reinforcement learning, as it guides the learning agent in exploring and eventually finding the most appropriate configuration. The reward function must ensure that the trial-and-error approach of the learning agent converges to the solution in a timely fashion. A lower reward is associated with less appropriate configurations and a higher reward with more appropriate ones. The agent aims to maximize the cumulative rewards of its sequence of actions, thus converging toward the solution. The reward signal is provided by the environment and is generated by the reward function R.

Since our goal is to learn the DEVS model that best simulates the *behavior* of the system, we reward the agent based on the similarity of the *traces* of the constructed DEVS model and the traces of the system.

Definition 1 (Trace)**.** *The trace of DEVS model M is a sequence of timestamped (τ) DEVS events $t(M)$ = {(τ_1, y_1), (τ_2, y_2), ... (τ_n, y_n), ...}, so that each event $y_i \in M.Y$.*

Table 2: Actions the learning agent can execute for coupling DEVS models

	Basic coupling actions	
Add new model instance	$addModelInstance(C, d) : C' = C \mid D \cup \{d\}$	(20)
Add specification	$addModelSpecification(C, M_d) : C' = C \mid M \cup \{M_d\}$	(21)
Remove model instance	$removeModelInstance(C, d \in D) : C' = C \mid D \setminus \{d\}$	(22)
Remove specification	$removeModelSpecification(C, M_d \in \{M\}) : C' = C \mid M \setminus \{M_d\}$	(23)
	Invariants	
Typed model instances	$\forall D, M_d, C : D \in C \Rightarrow M_d \in C.M$	(24)
	Connecting models	
Add influencee	$addInfluencee(C, d_i, d_j \in D) : C' = C \mid I_{d_i} \cup \{d_j\}$	(25)
Remove influencee	$removeInfluencee(C, d_j \in \{I_{d_i}\}) : C' = C \mid I_{d_i} \setminus \{d_j\}$	(26)
	Interacting with other models	
Add output	$addOutput(C, y) : C' = C \mid Y_{self} \cup \{y\}$	(27)
Remove output	$removeOutput(C, y \in Y) : C' = C \mid Y_{self} \setminus \{y\}$	(28)
Add input	$addInput(C, x) : C' = C \mid X_{self} \cup \{x\}$	(29)
Remove input	$removeInput(C, x \in X) : C' = C \mid X_{self} \setminus \{x\}$	(30)
	Tie-breaking	
Add select condition	$addSelect(C, D' \subseteq D, d \in D) : C' = C \mid select := select \cup (D', d)$	(31)

Definition 2 (Distance metric of traces)**.** *The distance metric on the set of traces T is a non-negative real-valued function $d : T \times T \to \mathbb{R}$, such that $d(t_1, t_2) = 0 \Leftrightarrow t_1 = t_2$; and $d(t_1, t_2) = d(t_2, t_1)$.*

Definition 3 (Reward function)**.** *The reward function is defined as $R = -|d(t(M), t(S))|$, i.e., the negative distance between the trace of the model under construction M and the trace of the system to be modeled S.*

4 DEMONSTRATION OF THE APPROACH: INFERRING TIME ADVANCE FUNCTIONS

In this section, we demonstrate how the formal framework of Section 3 can be used.

4.1 Demonstrative Example

Our running example is loosely based on the work of Van Tendeloo and Vangheluwe (2017) and entails a traffic light system in which an autonomous three-color traffic light and a policeman coordinate traffic. In *automatic mode*, the traffic light is active, and changes its active state—its color—sequentially in a time-based fashion. The red light is active for 60 seconds; followed by the green light, active for 50 seconds; followed by the yellow light, active for 10 seconds before switching back to red. This sequence continues until the policeman interrupts it and switches from automatic to *manual mode*. In the running example, the policeman is idle for 200 seconds, then interrupts the traffic light, works for 100 seconds, and then gives the control back to the traffic light and goes idle again for 200 seconds. The goal is to learn the behavior of the system by observing its output and constructing a DEVS model that behaves as similarly as possible. Specifically, in this demonstration, we attempt to learn the time advance function ta of the DEVS model. We implement the example as shown in Figure 1. We first develop an appropriate simulator of the physical

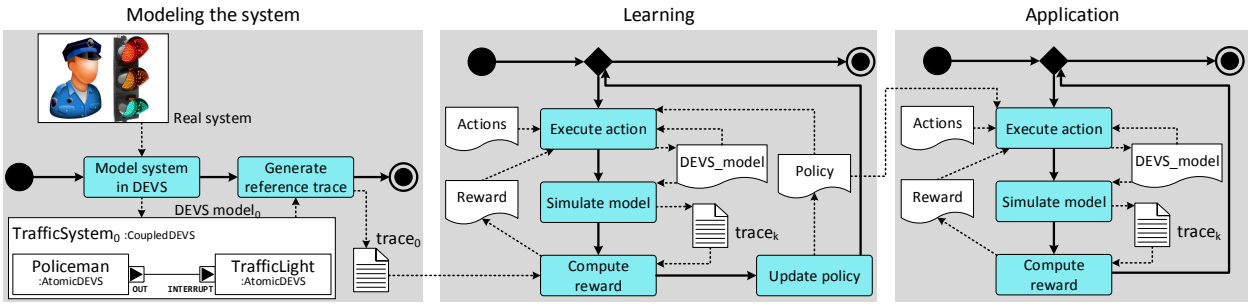


Figure 1: Overview of the approach.

system via DEVS and generate a reference trace (Section 4.2). Subsequently, we use this reference trace in the learning process (Section 4.3). Finally, we evaluate the approach by applying the policy (Section 4.4).

4.2 Modeling the System

We rely on a simulated system (Matulis and Harvey 2021) for the agent to interact with the system to be modeled. This is a convenient choice to avoid costly physical prototyping. To this end, as shown in Figure 1, we first model the system in DEVS. The resulting model is a Coupled DEVS model with two Atomic DEVS components: the *Policeman* and the *TrafficLight*, with the former being able to interrupt the latter and take over control. For modeling purposes, we rely on the PythonPDEVS library(Van Tendeloo and Vangheluwe 2015). We generate the reference trace by executing the model with an appropriate tracer. The resulting $trace_0$ will be used in the learning phase as the input to the learning agent to compare the trace of the candidate model.

4.3 Learning Phase

As shown in Figure 1, the learning agent iteratively chooses actions and executes them, thereby updating the candidate DEVS model. The learning environment contains an instance of a PythonPDEVS simulator to simulate the candidate model. The output of the simulation in iteration k is $trace_k$, which is used to compute the reward for the latest action. This cycle is repeated multiple times, ensuring enough time for the agent to infer a well-performing policy by continuously updating it.

We use the Tensorforce deep reinforcement learning framework to implement our example. Table 3 lists the main parameters and hyperparameters of our setting.

Agent. We use a proximal policy optimization (PPO) agent (Schulman et al. 2017). PPO is a family of policy gradient methods that alternate between interacting with the environment and optimizing an objective function by stochastic gradient ascent or descent. PPO allows for multiple gradient updates of the objective function in each epoch of batches (i.e., one forward and one backward pass of the training examples in the batch), sufficiently improving the performance of standard policy gradient methods while being easier to tune. The number of updates is defined by the *multi_step* parameter, set to 10 in our experiments. The gradient ascent or descent update is performed on batches of action trajectories defined by the *batch size*. We use a batch size of 10 in our experiments. A crucial element of policy update is the update step. If the policy is updated in too large steps, the policy performance can deteriorate. We use a conservative *likelihood ratio clipping* of 0.2 that removes the incentive for changing the objective function beyond the [0.8, 1.2] interval. To allow the agent to collect trajectories up to the horizon without degrading the weight of later decisions, we set the *discount* factor of the reward function to 1.0. To allow some flexibility to randomly explore, the

Table 3: Agent settings

Parameter	Value	Parameter	Value	Parameter	Value
agent	PPO	learning rate	1e-3	actions	type='int'
multi_step	10	l2_regularization	0.0		shape=(3,)
batch_size	10	entropy_regularization	0.0		num_values=5
likelihood_ratio_clipping	0.2	states	type='float'	episodes	2500-10000
discount	1.0		shape=(3,)	max_episode_timesteps	100
exploration	0.01		min_value=0	sim. events generated	50
subsampling_fraction	0.33		max_value=120		

exploration parameter is set to 0.01. That is, the agent can deviate from the policy in 1% of the cases. We keep the *subsampling fraction*, i.e., the ratio of steps used for computing back-propagation within a batch at the default 0.33. We do not use *L2* and *entropy regularizations*. L2 regularization adds random noise to the loss to prevent overfitting, but we have not experienced such problems. Entropy regularization would help balance the probabilities of the policy to keep the entropy at maximum, but we assume equally likely actions at the beginning of the training. The neural network storing the policy is composed of five layers: three dense layers with two register layers in between them, as suggested for Tensorforce settings with multiple actions.

States. We encode the system under study in a 3×1 tensor of decimal numbers (floats) as follows: [ta_{red}, ta_{green}, ta_{yellow}]. We set the following meaningful constraints to keep the demonstrative example feasibly solvable for the purposes of this paper: $0 \leq ta_{red}, ta_{green}, ta_{yellow} \leq 120$.

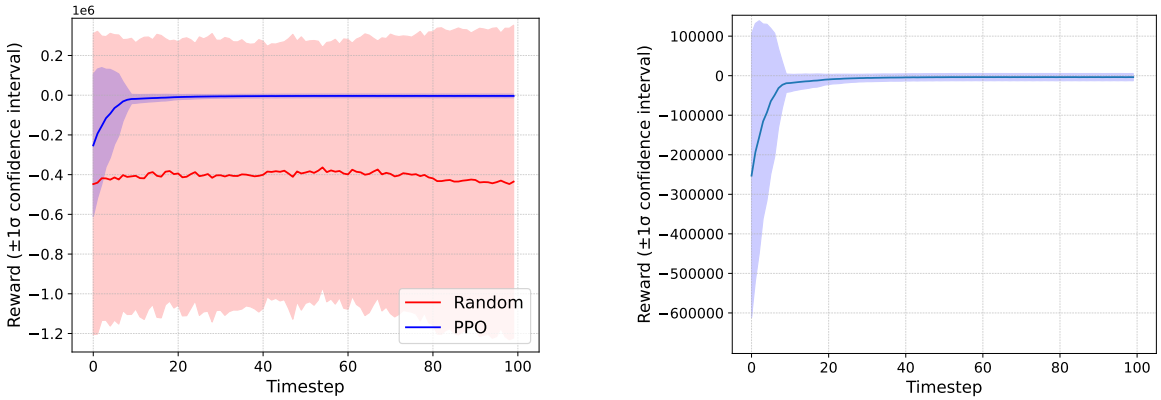
Actions. For each ta, we define five actions by Rule (14). Micro-increase (decrease) actions increase (decrease) the specific ta by 1; macro-increase (decrease) actions increase (decrease) the ta by 5; and the fifth action leaves the ta value intact. The actions are encoded as a 3×5 tensor, which allows the agent to choose one action for each *ta* in each step and execute these actions in parallel. Action tensors are masked in a way that the agent's actions cannot bring the ta values outside the [0, 120] interval.

Reward. By Definition (3), the reward function is formulated as the distance between the trace of the candidate model and the reference trace. Formally, following the notations of Figure 1, the reward in step k is defined as $r_k = -|d(trace_k, trace_0)|$. The goal of the agent is to maximize the expected cumulative reward, i.e., $max\Big\{E\{\sum_{k=0}^{H} a_k r_k\}\Big\}$, where H denotes the horizon of the action trajectory, a_k denotes the kth action, and r_k denotes the kth reward. To ensure faster convergence to the learning objective, we used $-r^2$ in our experiments. The distance metric is the Euclidean distance between the traces. Accordingly, we use the 2-norm (or Euclidean norm) of the trace vectors in our calculations, i.e., $r_k = \big(\sum_{n=1}^{|trace_k|} |x_n|^2\big)^{\frac{1}{2}} - \big(\sum_{m=1}^{|trace_0|} |x_m|^2\big)^{\frac{1}{2}}$, where $|\cdot|$ denotes length with vector arguments $trace_k$ and $trace_0$, and absolute with scalar arguments x_n and x_m.

Training. In our experiments, we used $2\,500 - 10\,000$ episodes to train the agent. Each training episode runs for 100 timesteps. In each timestep, the agent simulates the candidate DEVS model for 50 output events. The best performing agent was trained for $5\,000$ iterations.

4.4 Application and Results

To evaluate the performance of the agent, we sample elements from the $[0, 120]^3$ state-space as initial values and observe how the trained agent performs with these initial values. Specifically, we observe how



(a) Performance of the Random and PPO agents

(b) Closer look at the performance of the PPO agent

Figure 2: Random agent vs. PPO agent

the stepwise reward function evolves and whether it converges to the desired value of 0. We sample a few thousand initial value vectors and calculate their mean and standard deviation at each step. We compare the performance of the agent to the performance of a similarly trained random agent—one that returns random action values—on the same problem and same random sample of initial values.

Figure 2 shows the results of our experiments. As shown in Figure 2a, the performance of the random agent earns consistently worst mean rewards than the performance of the PPO agent. A Mann-Whitney U-test of the reward vectors produced by the two agents reveals a significant difference ($\alpha = 0.05$) between them at every timestep. Figure 2b shows that the rewards are rapidly improving in timesteps 1–9, and the PPO agent's performance becomes steady after timestep 20, remaining close to the ideal reward. Although the moment statistics of these reward vectors do not carry absolute information (e.g., a negative reward does not always mean worst performance), we still report them in Table 4 for relative comparison. We calculate the mean (μ) and standard deviation (σ) of the two reward vectors for the full interval; and specifically for the PPO agent, we additionally calculate (i) for the interval after timestep 9, (ii) for the interval after timestep 20, and (iii) for the last 10 timesteps to show how the agent converges to the solution. As shown by the figures in the table, the PPO agent gradually improves its rewards as the mean and standard deviation improve.

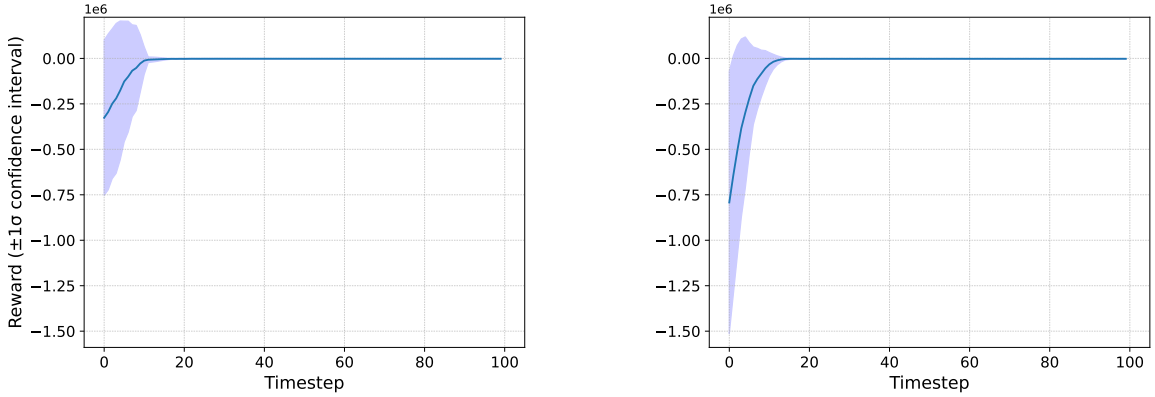
Table 4: Moment statistics of the reward vectors of the Random and PPO agents

	Random	PPO	PPO, $\tau \geq 9$	PPO, $\tau \geq 20$	PPO, $\tau \in [90, 100]$
μ	−406 077.38	−14 867.40	−5 598.93	−4 351.62	−3 644.23
σ	700 702.78	24 577.17	3 041.47	9 418.41	1 601.49

4.5 Generalizability of the Approach

To test the generalizability of the approach, we have tested it on different systems, introducing a change to the time advance functions of *DEVS model*$_0$ in Figure 1. In *System A*, we change the ta values to $[50, 40, 15]$, which is an average of 28.9% of change compared to the original [60, 50, 10]. In *System B*, we change the ta values to $[30, 20, 15]$, an average change of 46.7%. Subsequently, we train two agents—PPO-A and PPO-B—with the same settings shown in Table 3, and the same reward function defined in Definition (3). Figure 3 shows how the two agents perform on their respective systems with randomly sampled initial

conditions. Table 5 shows how the mean and standard deviation statistics change as the underlying system changes. The overall means and standard deviations of the reward vectors generated by PPO-A and PPO-B agents are comparable. In the last 10 timesteps, the two agents behave almost similarly.



(a) PPO-A: Average change of 28.9%.

(b) PPO-B: Average change of 46.7%.

Figure 3: Performance of the approach on alternative underlying systems

Table 5: Moment statistics of the reward vectors of agents trained on alternative underlying systems

	PPO-A	PPO-A, $\tau > 9$	PPO-A, $\tau \in [90, 100]$	PPO-B	PPO-B, $\tau > 9$	PPO-B, $\tau \in [90, 100]$
μ	−18 385.64	−2 468.84	−1 834.81	−34 904.21	−3 276.26	−2 033.27
σ	124 080.44	18 932.22	1 496.93	184 644.55	14 879.08	1 398.03

5 DISCUSSION

As demonstrated in Section 4, our approach (i) performs well on the original problem (Section 4.4); and (ii) the configuration (Table 3) and general reward function (Definition (3)) generalizes appropriately to a class of similar problems (Section 4.5). This class is characterized by changes in the time advance function ta of the original system's DEVS model. Since the definition of the reward function is generic and does not rely on model-specific or system-specific information, this result promises potential generalizability to classes of systems wider than shown here, such as systems with different states and internal transition functions. These directions will be explored in future work.

While the eventually inferred modeled indeed behaves similarly to the system to be modeled, some threats to validity apply. First, it is not always obvious whether learning a similarly behaving model is sufficient, or structural constraints should be respected as well. This is a threat to construct validity. We took multiple steps that mitigate this threat, namely: (i) approximating the similarity of models by their behavior, (ii) encoding the behavior as traces, and (iii) using a Euclidean distance for measuring the similarity of traces. We mitigated the threats to validity by ensuring that behavioral congruence is indeed sufficient in our demonstrative example, by defining the trace information appropriately, and by verifying that the learning agents run with appropriate parameters. Alternative distance measures can be considered to further improve the reward function in our approach, such as dynamic time warping and various kernel methods. In many practical settings, ensuring behavioral congruence is sufficient and this threat to validity may not apply.

We opted for learning DEVS models because by choosing DEVS, a wide range of systems can be naturally modeled by our approach. While this choice makes our approach fairly generic, the approach is reusable

only if appropriate congruence is ensured between applications. This is a threat to external validity. Our paper showed the settings of the agent and the reward function are reusable for a class of models with varying time advance functions. However, we do not claim generalizability of the policy, as our approach might be limited to structurally similar models. Transposing the originally inferred policy to other problems without retraining the agent is a challenge to be investigated in future work. One of the main directions to explore is the ability of validity frames (Van Mierlo et al. 2020) to capture the contextual information of the physical asset and its environment. By that, the conditions under which the policy π is transferable to other problems could be expressed in a formal way.

Although this paper focused on machine learning based inference of simulation models, we foresee our approach to be best employed in combination with a human in the loop. A human expert can provide the machine with meaningful approximate solutions, oversee the activities of the agent, and guide it by hints. While humans perform poorly in specifying optimal models, they perform considerably well in specifying reasonable ones (Wiering and Van Otterlo 2012). Despite the high costs of human reward function in naive reinforcement learning approaches, learning from the human has been shown to be feasible in numerous settings (Knox and Stone 2012, Loftin et al. 2016).

Potential applications of the approach include the development of simulation models for cyber-physical systems and digital twins (Cronrath et al. 2019), and approaches with operational semantics modeled in DEVS, such as process optimization (David et al. 2018), programmed graph transformations (Syriani and Vangheluwe 2010), and service interactions in software toolchains (Van Mierlo et al. 2018).

6 RELATED WORK

The complexity of constructing valid and sound simulators tends to increase rapidly with the growth of the underlying system. Spiegel et al. (2005) showed that identifying assumptions in simple models such as Newton's second law might be infeasible. Page and Opper (1999) identify various sources of complexity that hinder simulator composability and component reusability and might offset their benefits. Such sources of complexity include horizontal challenges, stemming from models situated at different levels of abstraction; vertical challenges, stemming from inappropriate abstraction mechanisms; and scalability challenges, stemming from the increased search friction due to the abundance of information. Our approach manages these challenges by automating the abstraction steps required to build a faithful model of the real system. By that, the limitations of human reasoning can be effectively bypassed.

Cronrath et al. (2019) use contextual bandits—a special form of reinforcement learning—to fine-tune digital twins used for control of manufacturing processes. Digital twin settings typically entail high-performance simulators for the automated optimization of the physical asset. The tuning of such simulation models is crucial in achieving appropriate precision and performance. However, the approach focuses on the run-time adaptation of simulation models and does not address their design phase. In contrast, our approach supports the full lifecycle of simulations models, from their construction to their run-time adaptation.

Floyd and Wainer (2010) use machine learning to learn the behavior of DEVS models. The approach is similar to ours as its learning approach is based on behavioral information based on the output of the observed system. However, while our approach relies on reinforcement learning and infers the DEVS model by trial and error without human intervention, the approach of Floyd and Wainer (2010) uses transfer learning, a special form of supervised learning. This necessitates a human in the loop to provide feedback upon the learning artifacts. The approaches are complementary to each other and their utility depends on the problem at hand. Reinforcement learning aligns better with learning DEVS models of autonomous systems, while transfer learning might perform better in learning human behavior and eventually replacing the human.

Choi and Kim (2002) provide means for extracting DEVS models from trained neural networks. The explainability of learning artifacts is a challenge in the latest incarnation of machine learning techniques, especially in complex ones, such as deep neural networks. Explainability of the learning artifacts allows the proper validation, verification, and modification of the learned artifacts. Such learning architectures are tailored to the specific simulation problem at hand and do not aim at the generality our approach aims at.

7 CONCLUSION

In this paper, we have presented an approach for the automated construction of DEVS models by reinforcement learning. The approach primarily aids the design phase of simulators and is of a particular utility in the context of systems that are challenging to model due to their complexity. To ensure the formal soundness of the approach, we have provided a mapping of DEVS models to reinforcement learning, which allowed us to exhaustively list the model construction actions the learning agent can employ while traversing the design space through a trial-and-error Markov Decision Process. We have evaluated our approach through an illustrative example and demonstrated its promising performance and generalizability properties.

Future work will primarily focus on the reusability of policies in wider classes of problems. Additional lines of research will focus on augmenting our approach with transfer learning techniques for incorporating a human in the loop, and on evaluating our theoretical framework on larger problems. Finally, we plan to generalize our current machinery (available at https://github.com/geodes-sms/devsrl) into a flexible framework for the benefit of the larger DEVS community.

ACKNOWLEDGEMENTS

The authors would like to thank Jessie Galasso and Houari Sahraoui for their valuable insights.

REFERENCES

Boschert, S., and R. Rosen. 2016. *Digital Twin—The Simulation Aspect*, pp. 59–74. Cham, Springer International Publishing.

Choi, S. J., and T. G. Kim. 2002. "Identification of discrete event systems using the compound recurrent neural network: Extracting DEVS from trained network". *Simulation* vol. 78 (2), pp. 90–104.

Cronrath, C., A. R. Aderiani, and B. Lennartson. 2019. "Enhancing digital twins through reinforcement learning". In *Automation Science and Engineering*, pp. 293–298. IEEE.

David, I., J. Galasso, and E. Syriani. 2021. "Inference of Simulation Models in Digital Twins by Reinforcement Learning". In *Proceedings of the 24th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, Fukuoka, Japan*, pp. 221–224, ACM.

David, I., H. Vangheluwe, and Y. Van Tendeloo. 2018. "Translating engineering workflow models to DEVS for performance evaluation". In *Winter Sim. Conference, Gothenburg, Sweden*, pp. 616–627. IEEE.

Floyd, M. W., and G. A. Wainer. 2010. "Creation of devs models using imitation learning". In *Proceedings of the 2010 Summer Computer Simulation Conference*, pp. 334–341. Citeseer.

Knox, W. B., and P. Stone. 2012. "Reinforcement learning from simultaneous human and MDP reward.". In *AAMAS*, pp. 475–482.

Loftin, R. et al. 2016. "Learning behaviors via human-delivered discrete feedback: modeling implicit feedback strategies to speed up learning". *Autonomous agents and multi-agent systems* vol. 30 (1), pp. 30–59.

Matulis, M., and C. Harvey. 2021. "A robot arm digital twin utilising reinforcement learning". *Computers & Graphics* vol. 95, pp. 106–114.

Mittal, S. et al. 2019. "Digital twin modeling, co-simulation and cyber use-case inclusion methodology for IoT systems". In *Winter Simulation Conference*, pp. 2653–2664. IEEE.

Page, E. H., and J. M. Opper. 1999. "Observations on the complexity of composable simulation". In *Proceedings of the 31st conference on Winter simulation*, pp. 553–560, WSC.

Schulman, J., F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. 2017. "Proximal policy optimization algorithms". *arXiv preprint arXiv:1707.06347*.

Spiegel, M., P. F. R. Jr., and D. C. Brogan. 2005. "A case study of model context for simulation composability and reusability". In *Proceedings of the 37th Winter Simulation Conference*, pp. 437–444, IEEE Computer Society.

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

Syriani, E., and H. Vangheluwe. 2010. *Discrete-Event Modeling and Simulation: Theory and Applications*, Book section DEVS as a Semantic Domain for Programmed Graph Transformation, pp. pp. 3–28. Boca Raton, CRC Press.

Vadori, N. et al. 2020. "Risk-sensitive reinforcement learning: a martingale approach to reward uncertainty". In *Proceedings of the First ACM International Conference on AI in Finance*, pp. 1–9.

Van Mierlo, S. et al. 2018. "A multi-paradigm approach for modelling service interactions in model-driven engineering processes". In *Proceedings of the Model-driven Approaches for Simulation Engineering Symposium, SpringSim (Mod4Sim) 2018*, pp. 6:1–6:12, ACM.

Van Mierlo, S., B. J. Oakes, B. Van Acker, R. Eslampanah, J. Denil, and H. Vangheluwe. 2020. "Exploring Validity Frames in Practice". In *Systems Modelling and Management*, pp. 131–148. Springer.

Van Tendeloo, Y., and H. Vangheluwe. 2015. "PythonPDEVS: A distributed parallel DEVS simulator". In *SpringSim (TMS-DEVS)*, pp. 91–98.

Van Tendeloo, Y., and H. Vangheluwe. 2017. "An introduction to Classic DEVS". *arXiv preprint arXiv:1701.07697*.

Vangheluwe, H. 2000. "DEVS as a common denominator for multi-formalism hybrid systems modelling". In *Symposium on computer-aided control system design*, pp. 129–134. IEEE.

Wiering, M. A., and M. Van Otterlo. 2012. "Reinforcement learning". *Adaptation, learning, and optimization* vol. 12 (3).

Zeigler, B. P., A. Muzy, and E. Kofman. 2018. *Theory of modeling and simulation: discrete event & iterative system computational foundations*. Academic press.

AUTHOR BIOGRAPHIES

ISTVAN DAVID is a Postdoctoral Research Fellow at the Université de Montréal, Canada. He received his Ph.D. in Computer Science from the University of Antwerp, Belgium. His research focuses on model-driven engineering, modeling of complex heterogeneous systems, and digital twins. He is an IVADO Postdoctoral Research Scholarship laureate on the topic of inference of simulation models in digital twins by reinforcement learning. Contact: https://istvandavid.com. Email address: istvan.david@umontreal.ca.

EUGENE SYRIANI is an Associate Professor at the department of computer science and operations research at University of Montreal. He received his Ph.D. in Computer Science in 2011 from McGill University. His main research interests fall in software design based on the model-driven engineering approach, the engineering of domain-specific languages, model transformation and code generation, simulation-based design, collaborative modeling, and user experience. Contact: http://www-ens.iro.umontreal.ca/~syriani. Email address: syriani@iro.umontreal.ca.