Automated Inference of Simulators in Digital Twins

Book chapter

ISTVAN DAVID, DIRO, Université de Montréal, Canada
EUGENE SYRIANI, DIRO, Université de Montréal, Canada

1 INTRODUCTION

Digital Twins are real-time and high-fidelity virtual representations of physical assets [35]. The intent of a Digital Twin is to provide data-intensive software applications with a proxy interface for the underlying asset. This is achieved by mirroring the prevalent state of the underlying system through the continuous processing of real-time data originating from the sensors of the physical asset. This one-directional dependent virtual representation is often referred to as a *Digital Shadow*. Digital Twins are composed of possibly multiple Digital Shadows and additional facilities that enable exerting control over the physical asset through its actuators.

This bi-directional coupling of physical and digital counterparts enables advanced engineering scenarios, such as automated optimization, real-time reconfiguration, and intelligent adaptation of physical systems. Such scenarios are typically enabled by simulators [9]. Simulators are programs that encode a probabilistic mechanism on a computer and enact its changes over a sufficiently long period of time [36]. Statistical methods are readily available to process the results of simulations, known as simulation *traces*.

However, the inherent complexity of systems subject to digital twinning renders the construction of these simulators an error-prone, time-consuming, and costly endeavor [5]. The complexity of constructing precise models tends to increase rapidly with the growth of the underlying system. Spiegel et al. [43] show that identifying assumptions even in simple models such as Newton's second law of motion ($\vec{p} = m\vec{v}$) might be infeasible. This is particularly concerning when considering the size and heterogeneity of systems subject to digital twinning. The inherent uncertainty and stochastic features of some domains—such as bio-physical systems [15]—further exacerbate this problem.

Therefore, managing the efforts and costs associated with simulator construction is a matter of paramount importance. Traditionally, reusability and composability of simulator components have been seen as an enabler to scaling up simulation engineering. However, these techniques are severely hindered by vertical challenges (stemming from inappropriate abstraction mechanisms), horizontal challenges (stemming from different points of views), and increased search friction due to the abundance of information [32]. In addition, Malak and Paredis [23] point out that the ability to reuse a model does not necessarily mean that it should be reused. However, model reuse often results in the segmentation of the knowledge required for validation across models of possibly different domains. Substantial efforts have been made to enable sound reuse and composition of simulator components. For instance, one can port the results of component-based software engineering to simulator engineering [3]. Also, one can rely on validity frames [50] to capture the context in which assumptions of a model are considered valid. However, these techniques still fall short of appropriately scaling up the engineering of simulators.

This chapter introduces the reader to an alternative approach to automating simulator construction by machine learning. Machine learning is a particularly appropriate technique for the inference and configuration of simulators because of the high volume of the data generated in systems subject to digital twinning. Specifically, in this chapter,

Authors' addresses: Istvan David, DIRO, Université de Montréal, Montréal, Canada, istvan.david@umontreal.ca; Eugene Syriani, DIRO, Université de Montréal, Montréal, Canada, syriani@iro.umontreal.ca.

we rely on the Discrete Event System Specification (DEVS) [54] as the formalism of simulation foundations; and we choose reinforcement learning [44] as the machine learning method for inference. The approach infers DEVS models by observing the system to be modeled. As the system traverses its state space, the reinforcement learning agent learns to react to these states by adapting the DEVS model to produce the same behavior as the system to be modeled. The versatility of DEVS vastly increases the reusability of the inferred knowledge.

Structure. The rest of this chapter is structured as follows. Section 2 provides a general introduction to the foundations of Digital Twin simulators (Section 2.1), the Discrete Event System Specification formalism (Section 2.2), and reinforcement learning (Section 2.3). Based on these preliminaries, Section 3 presents an approach for the automated inference of Digital Twin simulators by reinforcement learning. Section 4 provides a general discussion of the approach and contextualizes it within the broader topic of Digital Twins. Finally, Section 5 concludes the chapter and outlines potential directions for future work.

2 FOUNDATIONS

This section provides the reader with an overview of selected foundational topics relevant to the automated inference of simulators in Digital Twins.

2.1 Simulators in Digital Twins

The complexity of engineered systems has been at a steady pace for decades. Not only is the number of system components increasing—as illustrated by Moore's law [38] and its variants—but the heterogeneity of the components is on the rise as well [33]. To cope with this ever-increasing complexity, the engineering of intricate systems, such as Digital Twins, is best approached through modeling and simulation.

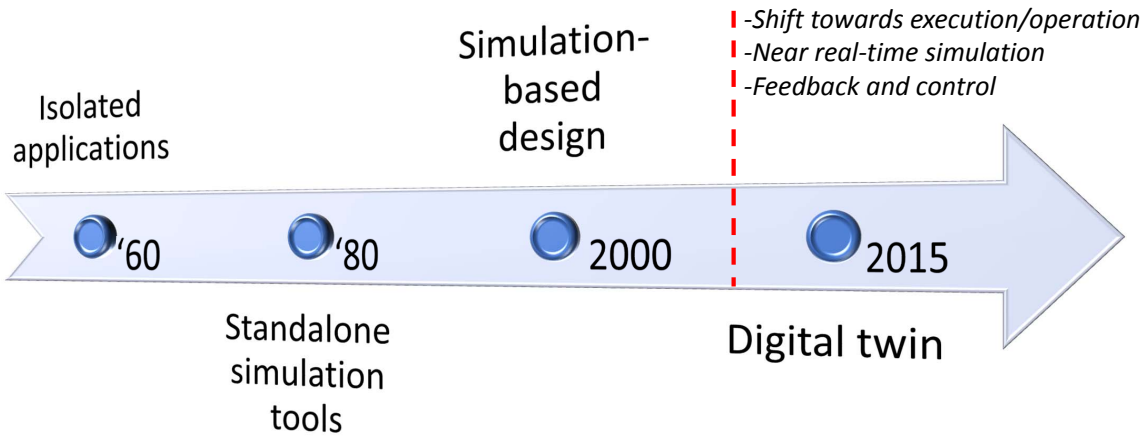


Fig. 1. The role of simulation in the engineering of complex systems – adopted from Boschert and Rosen [6]

The typical role of simulation has shifted from the design phase of complex systems to their operational phase. Boschert and Rosen [6] identify four significant phases of this evolution, shown in Figure 1. In the early 1960s, simulation was used in isolated cases of select domains employing the few experts in simulation, such as mechanical engineering and military intelligence. About two decades later, the first wave of simulation tools arrived, with a particular focus

on engineering problems. With the advent of multi-disciplinary systems engineering—highlighted by domains such as mechatronics and early incarnations of today's cyber-physical systems—the role of simulation has been firmly positioned in the design phase of systems engineering. Simulation has become a mainstay of engineering complex systems. Finally, the emergence of Digital Twins shifted the usage of simulators towards the operational phase of systems.

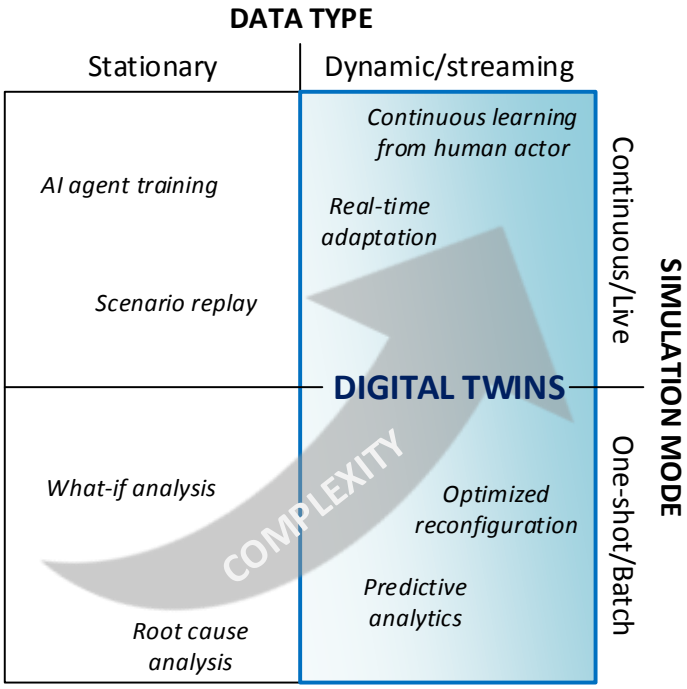


Fig. 2. Characteristic Digital Twin services and the source of their complexity

Today, simulators are first-class components of Digital Twins and enablers of the sophisticated features and services Digital Twins provide. As shown in Figure 2, the complexity of these features and services is a result of two factors. First, data in Digital Twins tends to be more dynamic than in traditional software applications. Often, models cannot be materialized in the memory to support reasoning because data is dispersed across the temporal dimension. Streams of sensor readings are prime examples of this challenge. Second, and related to the previous point, simulation scenarios in Digital Twins are often of a continuous or live nature. This is contrasted with one-shot scenarios in which a simulator is instantiated on demand, used to simulate a quantitative property, and then disposed. Pertinent examples of these complex services include:

- real-time adaptation of the system to a changing environmental context;
- optimized re-configuration to align with changing business goals;
- predictive analytics for the early identification of threats, such as failures and errors;
- what-if analysis to evaluate human decisions to be made;
- providing a learning environment for training purposes of human and computer agents.

At the core of the simulator, the physical asset is represented by a formal model, from which complex algorithms calculate the metrics of interest. This model has to capture the specificities of the physical asset in appropriate detail to consider the results of the simulation representative of the physical asset. Typically, simulation models reuse or rely on models already present in the Digital Twin, e.g., the models defined by Digital Shadows (Figure 3).

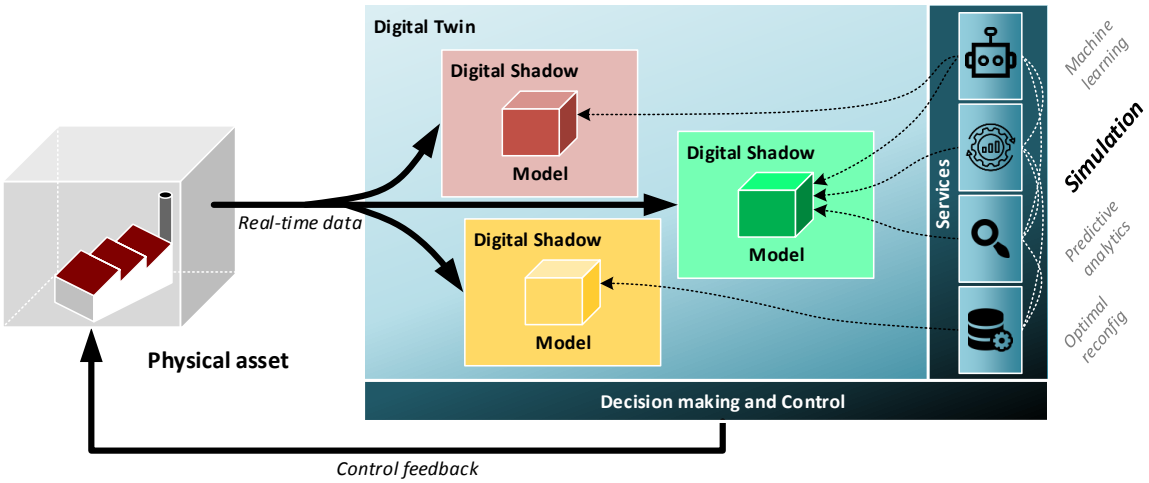


Fig. 3. Simulators are first-class citizens in Digital Twins

A model [26] is an abstraction of reality and allows its users to focus on the essential aspects of the problem at hand while still providing a basis for rigorous methods to analyze, verify, and operationalize models. How to model the problem at hand depends on the specific use case. For example, a safety analysis might rely on discrete and abstract models, such as a Petri net; or rely on acausal continuous models, such as bond graphs or differential equations. The following section introduces the reader to a versatile modeling formalism with exceptionally high utility and potential in the simulation of complex systems, such as Digital Twins.

2.2 Discrete Event System Specification (DEVS)

Investing resources into the learning process only pays off if the learned simulation model is representative of a class of problems rather than a specific case. For example, after learning how to construct the Digital Twin of a robot arm, one would like to use this knowledge to construct the Digital Twin of a robot leg. This generalization can be achieved at the level of the simulation model by choosing the appropriate simulation formalism. The Discrete Event System Specification (DEVS) formalism [54] is a prime candidate for this purpose, as the versatility of DEVS allows modeling the reactive behavior of real systems in great detail, including timing and interactions with the environment—two aspects predominantly present in the physical assets of Digital Twins [27].

DEVS is a formalism for modeling and simulation of systems. DEVS is a hierarchically compositional formalism. That is, it allows for constructing system specifications of arbitrary complexity through the sound composition of more primitive DEVS models. DEVS is closed under composition, i.e., composing two DEVS models results in a new DEVS model. Although discrete by nature, DEVS also supports modeling and simulating continuous state systems and hybrid continuous-discrete state systems. As Vangheluwe [51] points out, DEVS can serve as an effective assembly language

between different simulation formalisms. Due to its ability to tackle the complexity and heterogeneity of the modeled system, DEVS is especially suitable for building simulators for Digital Twins.

The discrete event abstraction of DEVS allows computing the prevalent state of the modeled system based on its previous state and the latest observed event. The state of a DEVS model is changed only upon observing events but remains constant in between. Events in DEVS are timed, i.e., they allow omitting intermediate points in time when no interesting events occur. This is in contrast with simulation formalisms with a fixed time step, where time increments with a fixed value. Flexibility in choosing the appropriate time granularity of DEVS events is especially useful in complex simulation and co-simulation scenarios of Digital Twins [17].

Events can originate from the DEVS model itself or from other DEVS models. This allows DEVS models to be reactive to other components' changes and to events generated by the modeled system. Being able to ingest real-time data is a distinguishing requirement against simulators of Digital Twins, and DEVS is an appropriate choice in this respect as well.

Due to its remarkable expressiveness, DEVS has been a particularly popular choice in academic and commercial simulation tools, such as PythonPDEVS[1] and MathWorks' SimEvents[2].

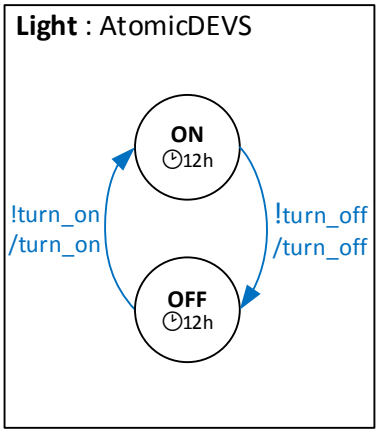
2.2.1 Running example. To illustrate the capabilities of DEVS modeling, we use the running example of a Digital Twin of a biological production room. The production room is isolated from the external environment, and every physical property is artificial. The lighting is turned on and off periodically, simulating twelve hours of daylight and twelve hours of night. Simultaneously, a heating, ventilation, and air conditioning (HVAC) system takes care of the proper temperature and humidity in the production room. The HVAC system is either in a heating state or in a cooling state, and its behavior follows the cycles of the lighting system. When the lights turn on, the HVAC system switches to heating mode to increase the temperature of the air. When the lights turn off, the HVAC system switches to cooling mode to decrease the temperature of the air. For safety reasons, the HVAC system is allowed two hours to prepare for the new operation mode after the switch is requested.

The Digital Twin supports two operation modes: observation and simulation. In observation mode, the Digital Twin provides human and machine agents with the prevalent state of the physical room in real time. In simulation mode, different performance indicators of the room are calculated based on scenarios enacted by a simulator. For example, one might calculate the overall energy consumption of the room for an extended period of time by simulating the behavior of the lighting and HVAC components.

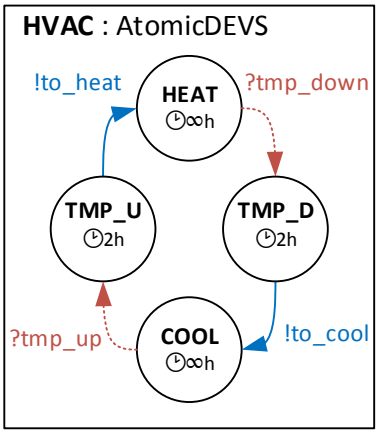
In this chapter, we use DEVS to construct models serving as the basis of simulation.

2.2.2 Atomic DEVS. Atomic DEVS models are the simplest DEVS models. They describe the autonomous behavior of a system or its component. Figure 4 represents two atomic DEVS models of the running example as timed automata. The model of the lighting system in Figure 4a shows that the lights are turned on and off periodically, every twelve hours. Timed transitions take the lighting system from one state to another. These transitions are independent of the surroundings of the lighting system; hence, they are referred to as internal transitions. In contrast, the model of the HVAC system in Figure 4b relies on external transitions. Once the HVAC system is in the heating () or cooling () state, the system cannot advance to the next state in a timed fashion. Transitions occur upon receiving an event, or the system remains in its current state. By convention, this behavior is expressed by a time advance set to infinity (∞) for each state. Upon receiving the `tmp_up` event in thestate, the system transitions into thestate to prepare for thestate.

[1] https://msdl.uantwerpen.be/documentation/PythonPDEVS
[2] https://www.mathworks.com/products/simevents.html



(a) Atomic DEVS model with internal transitions



(b) Atomic DEVS model with internal and external transitions

Fig. 4. Atomic DEVS models from the Digital Twin of a biological production room

The system then proceeds to thestate for two hours (2h) before transitioning to thestate in an automatic fashion. The transition fromtovia thefollows the same logic.

We use the following definition of atomic DEVS to provide formal grounds.

Definition 1 (Atomic DEVS). *An atomic DEVS model is defined as the eight-tuple* $M = \langle X, Y, S, q_{init}, \Delta_{int}, \Delta_{ext}, \lambda, ta \rangle$.

S is the set of states in the DEVS model. In the *Light* model in Figure 4a, S = {,}. $\Delta_{int} : S \to S$ is the internal transition function that maps the current state to the next state. In the *Light* model, Δ_{int} = {→,→}. Internal transitions capture the autonomous behavioral dynamics of a DEVS model as they are driven by internal factors and are independent of external factors. The internal factor determining *when* state transitions occur is the $ta : S \to \mathbb{R}^{+}_{0,+\infty}$ time advance function. In the *Light* model, ta = {→ 12h,→ 12h}. That is, after exhibiting thestate for twelve hours, the system transitions to the next state. According to Δ_{int}, this next state will be thestate. 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. In the *Light* model, q_{init} = {, `0.0`}. That is, the room is initialized with thestate, with 0.0 elapsed time at the beginning of the simulation.

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. In Figure 4, these events are prefixed with a question mark (?). For example, in the *HVAC* DEVS model, X={`tmp_up`, `tmp_down`}. These events are provided by external event sources, for example, other DEVS models. To this end, Y denotes the set of output events of the DEVS model that other models can react to. In Figure 4 these events are prefixed with an exclamation point (!). For example, in the *HVAC* model, Y={`to_heat`, `to_cool`}. Finally, the $\lambda : S \to Y$ output function specifies how output events are generated based on the state of the model. Output events are generated before the new state is reached. That is, the output function associates an output event with the state that is being left. For example, in the *HVAC* model, λ = {→ `to_heat`,→ `to_cool`}.

2.2.3 Coupled DEVS. DEVS supports the hierarchical composition of models into more complex DEVS models. Such complex DEVS models are called coupled DEVS models. As discussed previously, the lighting and HVAC components

of the running example can be modeled as atomic DEVS models. However, the tmp_up and tmp_down events in the *HVAC* model need to be supported by the transitions in the *Light* model. The coupled DEVS formalism provides the means to model this connection.

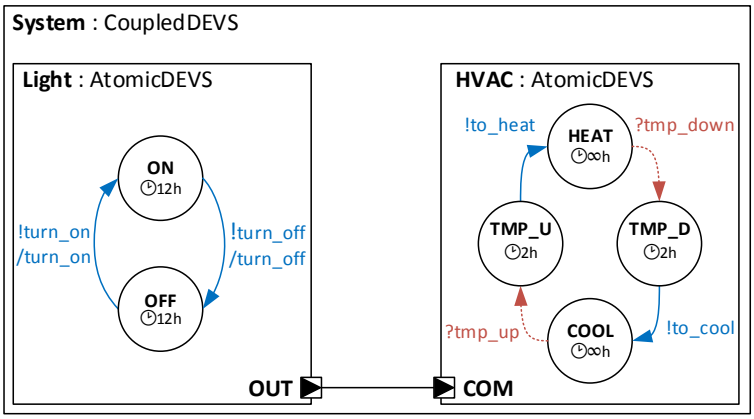


Fig. 5. Coupled DEVS model, integrating the atomic DEVS models in Figure 4

Figure 5 shows a DEVS model coupling the two atomic DEVS models previously seen in Figure 4. In this coupled DEVS model, the *HVAC* model reacts to state transitions in the *Light* model by observing its output events arriving to its command (COM) port from the output (OUT) port of the *Light* model. As the *Light* model is turned on, the turn_on event notifies the *HVAC* model to activate the transition from thestate to thestate.

We use the following definition of coupled DEVS to provide formal grounds.

Definition 2 (Coupled DEVS). *A coupled DEVS model is defined as the seven-tuple* $C = \langle X_{self}, Y_{self}, D, \{M\}, \{I\}, \{Z\}, select \rangle$.

D is the set of model instances that are included in the coupled model. In the coupled *System* model, D = {light, hvac}. $\{M\}$ 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 Definition 1. In the coupled *System* model, M_{light} is the model in Figure 4a, and M_{hvac} is the model in Figure 4b. To connect model instances within a coupled model, $\{I\}$ defines the set of model influences such that $\forall d \in D \exists I_d : d \rightarrow D' \subseteq D \setminus \{d\}$. In the *System* model, it is the *Light* model influencing the *HVAC* model. That is, I_{light} = {hvac}, and $I_{hvac} = \varnothing$. Similar to atomic DEVS models, coupled DEVS models can define input and output events to interface with other models. X_{self} and Y_{self} denote the input and output events, respectively. In the example Figure 5, the coupled DEVS model does not define its own events. Thus, $X = \varnothing$, and $Y = \varnothing$. Due to the independence of models $d \in D$ within the coupled DEVS model, events within the coupled models might happen in parallel. To retain the unambiguous execution semantics, the *select*: $2^D \rightarrow D$ tie-breaking function defines which model to treat with priority in case of conflicting models. In the *System* DEVS model, we choose to prioritize the *Light* component over the *HVAC* component, i.e., *select* = {{light, hvac} → light, {light} → light, {hvac} → hvac}. Finally, 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. In the coupled *System* DEVS model, the turn_on output event of the *Light* model should trigger the transition from theto thestate in the HVAC model. That is, the turn_on output event should be translated to the tmp_up input event. Similarly, turn_off should be translated to tmp_down. That is, Z = {turn_on → tmp_up, turn_off → tmp_down}.

2.3 Reinforcement learning

Reinforcement learning [44] is a machine learning paradigm and a family of algorithms implementing goal-directed learning from interactions with the environment. Reinforcement learning focuses on learning how to react to specific states of the environment—that is, how to map situations to actions—–in order to maximize a reward signal.

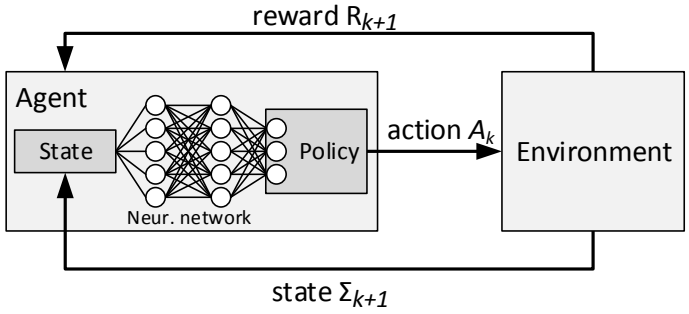


Fig. 6. Deep reinforcement learning – conceptual overview

Figure 6 shows the conceptual overview of reinforcement learning. In a typical reinforcement learning setting, the *agent* explores its *environment* sequentially and exploits it to achieve a result. At each step, the agent carries out an *action*, observes the new *state* Σ_{k+1} of the environment, and receives a *reward* R_{k+1}. 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 the agent can encounter to actions the agent can execute in those situations. Thus, the policy is defined as $\pi(a \in A | \sigma \in \Sigma)$ to choose action a given the current state σ. The policy can be represented in a simple lookup table or can be as intricate as a neural network. The latter case is known as *deep reinforcement learning*, where the eventual decision about the next action is encoded in the output layer of a neural network.

Markov decision processes [34] and Markov chains [29] are particularly apt choices to formalize the process of reinforcement learning and the subsequent behavior of the trained agent.

Definition 3 (Markov decision process). *A Markov decision process is defined as four-tuple* $\langle \Sigma, A, P_A, R_A \rangle$.

Here, Σ denotes the set of states of the environment the agent can observe. A denotes the set of actions the agent can perform on the environment. $P_A : (\sigma \in \Sigma, a \in A) \mapsto Pr(\sigma' | \sigma, a)\rangle$ is the policy, that is, the probability of the agent executing action a in state σ, which will result in the new state σ'. $R_A : (\sigma \in \Sigma, a \in A) \mapsto \mathbb{R}$ is the numeric reward received for choosing action a in state σ.

Markov chains [29] are the formal structures underpinning Markov decision processes. A Markov chain is a stochastic state transition system with a martingale property, i.e., they describe sequences of events in which the probability of an event depends solely on the prevalent state and not on previous states. A Markov decision process extends the underlying Markov chain by the notion of *choice* (through the concept of actions) and *motivation* (through the notion of rewards). Conversely, by reducing the number of actions to one—i.e., $A = \{a\}$—and the reward to a constant—e.g., $R_A \mapsto 1$—the Markov decision process reduces to a Markov chain. Similarly, a Markov decision process with a policy emulates the underlying Markov chain as action $a \in A$ in state $\sigma in \Sigma$ is determined by the policy acting as the Markov transition matrix.

The aim of a Markov decision process is, therefore, to identify the best policy P_A that maximizes the reward (in policy-based methods) or a more long-term, accumulative value (in value-based methods). The behavior of a reinforcement learning agent can be formally captured in this compact formal structure. This formalization helps define how reinforcement learning can be programmed to learn simulators, and in particular, simulation models conforming to DEVS.

2.3.1 Various flavors of reinforcement learning. While the reward function quantifies the immediate reward associated with an action, it is often more useful to optimize the agent's behavior for a long-term *value* function. The value function V is the expected sum of rewards along a trajectory of actions. *Value-based* reinforcement learning methods aim to maximize this value function. In contrast, *policy-based* methods aim to maximize the reward function in every state. *Actor-critic* methods provide a trade-off between value-based and policy-based methods, in which the actor implements a policy-based strategy, and the critic criticizes the actions made by the actor based on a value function. Different methods offer different benefits and perform with different utility in specific learning problems. Value-based approaches are deterministic because they select actions greedily when maximizing the value function; however, this might lead to under-exploration. In contrast, policy-based methods explore the state space more thoroughly but produce more noisy estimates, potentially resulting in unstable learning processes. Recently, non-cumulative reward functions with a martingale property—that is, $\mathbb{E}(V|a_1, ...a_n) = R(s_n, a_n)$—have been successfully utilized to address these problems [49].

The above methods belong to the *active* type of reinforcement learning methods, in which the goal is to learn an optimal policy. In contrast, *passive* reinforcement learning methods aim to evaluate how good a policy is. Finally, in *inverse* reinforcement learning settings, the goal is to learn the reward function under a fixed policy or value function and state space.

Specifically, this chapter illustrates an approach for learning DEVS models by a policy optimization method called Proximal Policy Optimization (PPO) [41]. PPO is a policy gradient method that improves upon its predecessor, Trust Region Policy Optimization (TRPO) [40], by eliminating performance drops while simplifying implementation. Due to its performance, PPO has become a popular choice recently for reinforcement learning tasks. Structurally, PPO is implemented in an actor-critic fashion, where the actor optimizes the policy by immediate rewards, and the critic drives the decisions of the actor by considering the value function of long-term rewards. The value function maintained by the critic is used in actor updates, thereby allowing the critic to influence the actor. PPO alternates between interacting with the environment and optimizing an objective function by stochastic gradient ascent or descent. The agent updates the actor and critic in a batch fashion, allowing for multiple gradient updates in each epoch (i.e., one forward and one backward pass of the training examples in the batch). At the end of each batch, the agent updates its actor and critic networks.

2.3.2 Reinforcement learning in simulator engineering. Reinforcement learning has been successfully employed for various problems of simulator engineering before. Imitation learning has been shown to be a feasible approach to infer DEVS models by Floyd and Wainer [16]. In an imitation learning setting, the agent learns from another agent or human by observing their reactions to a sequence of inputs. Such techniques show an immense utility when eliciting the tacit knowledge of an expert is a particularly challenging task and expert demonstrations are easier to carry out. A substantial challenge in imitation learning is the low number of training samples, and therefore, the quality of demonstrations is paramount for success. For example, demonstrations must sufficiently cover the characteristic cases and edge-cases of problem-solving process. The amount of learning input is not a problem when learning DEVS models of autonomous systems, and therefore, the techniques of reinforcement learning can be fully leveraged.

A less interactive way to learn from examples has been provided by Otero et al. [31], who use inductive learning to infer general knowledge about the system in the form of DEVS models. The approach relies on a behavior description language by which the system has to be described in appropriate detail, serving as the training input to the learning algorithm.

Deep reinforcement learning has been used by Sapronov et al. [37] to learn and store the parameterization of discrete event models. The eventual policy encodes an optimal control strategy for the parameters of the simulator. Such a fusion of a reinforcement learning trained neural network with discrete event simulations offers better customizability of the simulator and the ability to optimize its configuration from an ingested data stream by updating the neural network.

A special form of classic reinforcement learning, contextual bandits [8] have been shown to be an appropriate choice in the run-time adaptation and reconfiguration of Digital Twins with continuous control [10, 47]. Contextual bandits reduce the environment to a one-state Markov decision process and focus solely on mapping action-reward combinations. This simplification is admissible when the results of an action can be assessed immediately. Continuous control problems of Digital Twins are prime examples of such settings. 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.

3 INFERENCE OF SIMULATORS BY REINFORCEMENT LEARNING

In our approach, simulators are inferred by observing the states of the physical system. As shown in Figure 7, Digital Twin settings typically provide advanced facilities to observe the system to build a simulator for. Particularly, the models of the corresponding Digital Shadow can be used to gather real-time information about the physical asset. In many cases, interacting with the environment is not feasible or raises safety concerns. In such cases, simulated environments can be used to infer simulators [24]. The advantage of simulated environments is that the learning agent can learn long trajectories (sequences of actions) in a short time as the responses of the environment are simulated. In addition, machine learning capabilities are already present in a typical Digital Twin setting, further simplifying the task of inferring simulators.

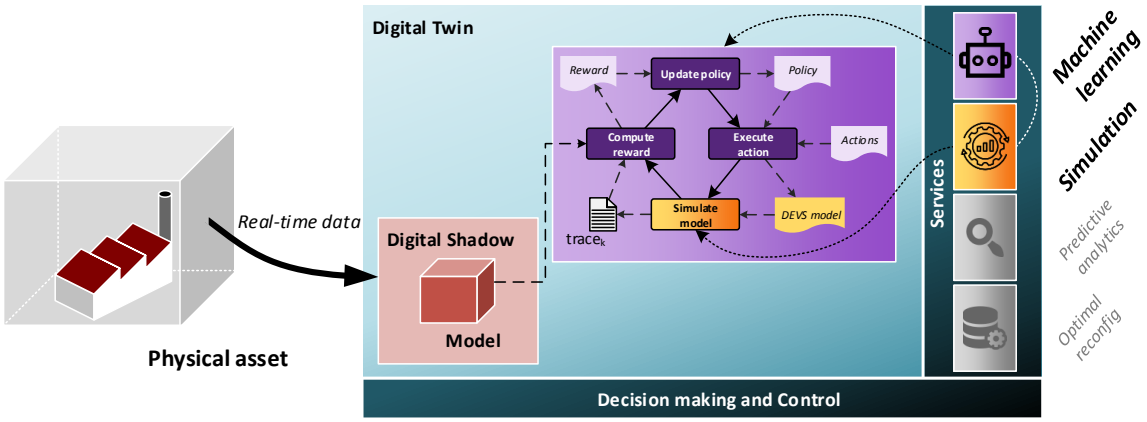


Fig. 7. Overview of the approach

Reinforcement learning excels in solving sequential decision-making problems based on an explicitly modeled set of actions. Thus, reinforcement learning is a particularly appropriate choice in engineering problems where the set of engineering actions is finite and effectively enumerable, such as DEVS model construction. 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. The learning agent iteratively chooses actions and executes them, thereby updating the candidate DEVS model. The candidate DEVS model is simulated with an appropriate simulator. The learning environment can be equipped with a standalone DEVS simulator for this purpose, or the Digital Twin can provide it as a service (as shown in Figure 7). The output of the simulation in iteration k is trace_k. The trace stores behavioral information about the simulation, which provides means for comparing the behavior of the simulator with that of the real system. The reward is computed by comparing the trace of the simulation and the trace (history) of the real system. This cycle is repeated multiple times, ensuring enough time for the agent to infer a well-performing policy by continuously updating it.

This training session results in two important artifacts. First, the final DEVS model which will be used in the simulator service of the Digital Twin. Second, the inferred intelligence persisted in the policy of the agent. This intelligence 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. Thus, this intelligence can be reused across Digital Twins built for different physical assets.

In the rest of this section, we outline a formal framework for integrating DEVS and reinforcement learning, and whenever necessary, we illustrate concepts through the illustrative example introduced in Section 2.2.1.

3.1 Formal framework

To utilize reinforcement learning in the inference of DEVS models, DEVS construction operations are needed to be translated to reinforcement learning terms. In the following, we briefly review such a translation, provided by David and Syriani [13] and David et al. [11].

Given a Markov Decision Process $MDP = \langle \Sigma, A, P_A, R_A \rangle$ (Definition 3) and the atomic DEVS specification $M = \langle X, Y, S, q_{init}, \Delta_{int}, \Delta_{ext}, \lambda, ta \rangle$ (Definition 1), the mapping $MDP \rightarrow M$ is briefly given as follows.

States. Every state $\sigma \in \Sigma$ of the Markov Decision Process encodes one DEVS configuration M. That is, $\forall \sigma \in \Sigma : \sigma \mapsto M$. We use the notation σ_M to denote the state that encodes M.

Actions. Intuitively, actions in the Markov Decision Process that learns to construct DEVS models, are DEVS operators. Every action is applied to a DEVS model M, resulting in a new model M'. Subsequently, a new state encoding M', $\sigma_{M'}$ is added to the states of the *MDP*. That is, $\forall a \in A : a(M) \mapsto M', \Sigma \mapsto \Sigma \cup \{\sigma_{M'}\}$. In accordance with the Markov property of the underlying search process, in each state, the probability of executing action $a \in A$ is determined by policy P_A.

Here, we only show one example used later in this chapter. Setting the time advance function to a specific value is defined as

$$updateTa(M, s, t' \in \mathbb{R}^{+}_{0,+\infty}) : M' = M \mid ta(s) = t'. \tag{1}$$

The detailed list of atomic and coupled DEVS actions is provided in [13].

Reward function. To guide the learning agent towards learning the right DEVS model, the reward function R_A maps each state to a metric. That is, $R_A : \Sigma \rightarrow \mathbb{R}$. The reward is calculated by simulating the candidate model and evaluating how similar its behavior is to the behavior of the modeled system.

The behavior of a simulation is reflected in its trace. Roughly speaking, a trace of a simulation $\mathcal{S}(M)$ is a sequence of timestamped DEVS events $t(\mathcal{S}(M)) = \{(\tau_1, y_1), (\tau_2, y_2), ... (\tau_n, y_n), ...\}$, so that each event $y_i \in M.Y$. ($M.Y$ denotes the set of output events – see Definition 1).

To measure the similarity of traces or the lack thereof, we need an appropriate distance metric. Roughly, a distance metric on the set of traces T is a non-negative real-valued function $d : T \times T \rightarrow \mathbb{R}$, such that $d(t_1, t_2) = 0 \Longleftrightarrow t_1 = t_2$ (identity); and $d(t_1, t_2) = d(t_2, t_1)$ (commutativity).

Based on the above, we are ready to define the general reward function for learning DEVS models as $R = -|d(t(M), t(S))|$. That is, the reward is inversely proportional to the distance between the trace of the model under construction M and the trace of the system to be modeled S.

3.2 Agent setup

Every reinforcement learning approach requires a specific setup. In this section, we show how a proximal policy optimization (PPO) agent (see Section 2.3.1) can be used to implement a reinforcement learning mechanism. Table 1 shows an example parameterization of the PPO agent implementation of Tensorforce, the reinforcement learning library of TensorFlow[3]. These settings have been shown to be successful in the inference of DEVS models previously [13].

Table 1. Example agent settings

Parameter	**Value**
agent	PPO
multi_step	10
batch_size	10
likelihood_ratio_clipping	0.2
discount	1.0
exploration	0.01
subsampling_fraction	0.33
learning rate	1e-3
l2_regularization	0.0
entropy_regularization	0.0
states	type='float' shape=(4,) min_value=0 max_value=24
actions	type='int' shape=(4,) num_values=5

The number of updates per batch is defined by the *multi_step* parameter. The gradient ascent or descent update is performed on batches of action trajectories defined by the *batch size*. A crucial element of policy update is the update step. If the policy is updated in too large steps, the policy performance can deteriorate. A conservative *likelihood ratio clipping* of 0.2 removes the incentive for changing the objective function beyond the [0.8, 1.2] interval. Setting the

[3] https://www.tensorflow.org/

discount factor of the reward function to 1.0 allows the agent to collect trajectories up to the horizon without degrading the weight of later decisions. Setting the *exploration* parameter to 0.01 allows the agent some flexibility to randomly explore. That is, the agent can deviate from the policy in 1% of the cases. The *subsampling fraction* parameter controls the ratio of steps taken in an episode used for computing back-propagation. *L2 regularization* adds random noise to the loss to prevent overfitting. This parameter should be set appropriately based on some manual experimentation of the agent's tuning. *Entropy regularization* would help balance the probabilities of the policy to keep the entropy at maximum, but in the context of this book chapter, we can safely assume equally likely actions at the beginning of the training. The neural network storing the policy is composed of seven layers: four dense layers with three register layers in between them, as suggested for Tensorforce settings with multiple actions.

3.3 States

The state of the agent has to store the four time advance values. This information can be captured in a 4×1 tensor of decimal numbers (floats) as follows: $[ta_{off}, ta_{on}, ta_{tmp\_u}, ta_{tmp\_d}]$. In the running example, the Digital Twin has to simulate cycles of days. Therefore, the following reasonable constraints can be specified: $0 \leq ta_{on}, ta_{off}, ta_{tmp\_u}, ta_{tmp\_d} \leq 24$.

3.4 Actions

To allow the agent to manipulate the time advance values of the DEVS model under construction, actions should be specified in accordance with Equation 1. Below, we show five typical actions.

Macro-increase $Inc(ta) : ta \mapsto ta + 0.1$
Micro-increase $inc(ta) : ta \mapsto ta + 0.01$
No change $noc(ta) : ta \mapsto ta$
Micro-decrease $dec(ta) : ta \mapsto ta - 0.01$
Macro-decrease $Dec(ta) : ta \mapsto ta - 0.1$

Macro-steps improve the convergence of the agent to a rudimentary solution, while micro-steps allow the agent to refine the solution within a solution area. To speed up the learning process, the agent should be able to change each of the four time advance values encoded in the state in each step. Therefore, it is a good idea to encode the five actions operating on the four time advance values as a 4×5 tensor.

The response to a chosen action is then obtained, as shown in Listing 1. Here, we only show the update of the first state variable, i.e., ta_{off}. The response mechanism of the agent expects the chosen actions on its input (`action`). The first element of the `action` tensor (`action[0]`) corresponds to the action chosen for the first element of the state, i.e., ta_{off}. The chosen action is represented by an integer. The switch case in Lines 10–21 interprets the value under `action[0]` and reacts accordingly. Changes to the rest of the state variables are calculated in a similar way. Finally, the new state is returned in Line 30.

Tensorforce provides means for specifying action masks which allow for controlling which actions are enabled in specific states. The action mask in Listing 2 ensures that the agent's actions cannot bring the time advance values outside the $[0, 24]$ interval. Element $[i, j]$ in the action mask tensor corresponds to element $[i, j]$ of the action tensor and element $[i]$ of the state tensor. For example, mask element $[1, 1]$ (`self.ta['off'] <= 23.9`) determines that the first action in the action tensor ($Inc(ta)$) action can be executed on the first value of the state (ta_{off}) only if $ta_{off} \leq 23.9$. Clearly, this means that in each timestep k (see Figure 7), $ta_{off}(k) \leq 24$.

```
def response(self, action):
  on_action = action[0]
  off_action = action[1]
  tmp_u_action = action[2]
  tmp_d_action = action[3]
  update = np.array([0, 0, 0, 0])

  #Compute ON action
  if on_action == 0: update += np.array([0.1, 0, 0, 0])
  elif on_action == 1: update += np.array([0.01, 0, 0, 0])
  elif on_action == 2: update += np.array([0, 0, 0, 0])
  elif on_action == 3: update += np.array([-0.01, 0, 0, 0])
  elif on_action == 4: update += np.array([-0.1, 0, 0, 0])
  else: raise Exception('Unknown action')

  #Compute OFF action
  ...
  #Compute TMP_U action
  ...
  #Compute TMP_D action
  ...
  return self.ta+update
```

Listing 1. Action mask controlling the applicability of actions

```
np.array([
  [self.ta['off'] <= 23.9, self.ta['off'] <= 23.99,
        True, self.ta['off'] >= 0.01, self.ta['off'] >= 0.1],
  [self.ta['on'] <= 23.9, self.ta['on'] <= 23.99,
        True, self.ta['on'] >= 0.01, self.ta['on'] >= 0.1],
  [self.ta['tmp_u'] <= 23.9, self.ta['tmp_u'] <= 23.99,
        True, self.ta['tmp_u'] >= 0.01, self.ta['tmp_u'] >= 0.1],
  [self.ta['tmp_d'] <= 23.9, self.ta['tmp_d'] <= 23.99,
        True, self.ta['tmp_d'] >= 0.01, self.ta['tmp_d'] >= 0.1],
])
```

Listing 2. Action mask controlling the applicability of actions

3.5 Reward

As per Section 3.1, the reward function is formulated as the distance between the trace of the candidate model and the reference trace. Following the notations of Figure 7, 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., $max\left\{E\{\sum_{k=0}^{H} a_k r_k\}\right\}$, 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, a more aggressively increasing penalty can be used, e.g., $-r^2$. The distance metric is the Euclidean distance between the traces. Accordingly, the 2-norm (or Euclidean norm) of the trace vectors can be used, i.e., $r_k = \left(\sum_{n=1}^{|trace_k|} |x_n|^2\right)^{\frac{1}{2}} - \left(\sum_{m=1}^{|trace_0|} |x_m|^2\right)^{\frac{1}{2}}$, where $|\cdot|$ denotes length with vector arguments $trace_k$ and $trace_0$, and absolute with scalar arguments x_n and x_m.

4 DISCUSSION

We discuss the advantages and limitations of the presented approach, as well as outline some challenges.

4.1 Advantages of this approach

As we have shown, reinforcement learning aligns well with the construction of simulators. As reported by previous work [13], the illustrative setup in Section 3 performs well on DEVS construction problems. The state encoding and the reward function in are general enough to accommodate simulation formalisms other than DEVS. Basically, it is the set of actions (Section 3.4) that make the presented approach specific to DEVS. By the separation of formalism-specific concerns, the approach promises high reusability across various formalisms used in Digital Twins.

Our approach illustrates that the complexity of simulator construction reduces to the essential complexity of setting up and tuning reinforcement learning agents. Accidental complexity is largely mitigated, mostly due to the high automation and minimized need for human intervention. The engineering of Digital Twins is challenged by the enormous complexity of the systems subject to twinning, and therefore, approaches such as the one outlined in this book chapter project to be key enablers in the next generation of Digital Twins.

With proper automation, these approaches reduce the costs of developing simulators. As a consequence, the development costs of Digital Twin services relying on simulation will decrease as well. Reduced costs, in turn, enable more sophisticated services and features. For example, training machine learning agents in the simulated worlds of Digital Twins allows the agent to explore potentially unsafe or hazardous settings without actual physical risk.

As emphasized previously, DEVS is a versatile simulation formalism. Consequently, it is not surprising to see numerous Digital Twin use cases relying on DEVS. Such use cases with substantial upside include the optimization of manufacturing processes through simulation [14], explicitly modeled interactions between services and with data APIs that allow for better validation and verification of interaction protocols [25], and explicit models at runtime driven by programmed graph transformations [45].

4.2 Limitations

While the eventually inferred model behaves similarly to the system to be modeled, this approach faces limitations.

It is not always obvious whether learning a similarly behaving model is sufficient or structural constraints should be respected as well. To overcome this limit, one needs to ensure that behavioral congruence is sufficient for the problem at hand. In addition, the trace information must be rich enough to allow reasoning about the property on which the similarity between the underlying asset and the simulation model is based on. Alternative distance measures can be considered to further improve the reward function, such as dynamic time warping and various kernel methods [4, 39]. In many practical settings, ensuring behavioral congruence is sufficient, and this threat to validity may be negligible.

While opting for DEVS makes the outlined approach fairly generic, the approach is reusable only if appropriate congruence is ensured between applications. As demonstrated in [13], it is feasible to parameterize the outlined approach in a way that the agent and the reward function are reusable for a class of models. However, the generalizability of the policy 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. One promising direction to explore is the ability of validity frames [50] to capture the contextual information of the physical asset and its environment. The conditions under which the policy is transferable to other problems could be expressed in a formal way.

4.3 Open challenges

Below, we outline some of the challenges and opportunities we find the most important currently.

4.3.1 Human-machine and machine-machine learning. We see opportunities in learning settings with the human or other machine learning agents involved. While humans perform poorly in specifying optimal models, they perform considerably well in specifying reasonable ones [53]. As a consequence, rudimentary models or model templates provided by domain experts would increase the performance of the approach. Furthermore, domain experts can be employed in the oversight of the learning process as well. In such interactive settings, the human expert can be queried by the machine at the points of the exploration where the search algorithm indicates tough choices for the machine. Learning from the human expert can be enabled to further improve the agent's performance. Despite the high costs of human reward function in naive reinforcement learning approaches [7], learning from the human has been shown to be feasible in numerous settings [19, 21].

Alternative techniques, such as learning from demonstration [2] and active learning [42], should be considered to augment the approach with. Once reinforcement agents become experienced enough with different instances and flavors of the same class of Digital Twins, new agents can be trained by experienced agents. Such settings would allow removing the human from the loop and replacing the manually assembled default policies with the ones the agents themselves inferred from previous cases.

4.3.2 Policy reuse in congruent problems. One of the main benefits of building inference methods specifically for DEVS is that the wide range of problems that can be mapped onto DEVS can effectively leverage the approach presented in this book chapter. This facilitates the much sought-after reuse of simulators and simulator components [23, 32]. Reusing the policy of the agent is different from simulator reuse. When a policy is reused, previously learned patterns of simulator inference are used to solve slightly different simulator inference problems.

Policy reuse is typically achieved by transfer learning [48], in which the output layer of the neural network encoding the policy is removed, and the rest of the network is retrained on the new problem. This, in effect, associates the previously learned solution patterns with the tokens of the problem at hand. The effectiveness of transfer learning in simulator inference is determined by the similarity of the original problem and the new one.

4.3.3 Ensembles of inverse learning methods. The approach presented in this book chapter uses a generic reward function based on behavioral traces. However, some problems might necessitate more specialized reward functions. As discussed in Section 4, appropriately capturing the reward function becomes challenging as the complexity of the underlying system increases. Similarly, we used a set of generic DEVS model engineering operations, but domain-specific approaches might prefer to leverage a more refined set of actions. For example, the exact value by with the time advance is increased or decreased in Section 3.4 was an arbitrary choice on our end.

To approach the estimation of these parameters in a sound way, inverse reinforcement learning methods [28] can be employed. Inverse reinforcement learning aims to learn the reward function under a fixed policy. Similarly, the action space can be inferred under a fixed policy and reward function. We foresee advanced learning settings in which the reward function, value function, and actions are inferred in an iterative fashion by an ensemble of inverse reinforcement learning algorithms. This will enable simulator inference processes that are better tailored to the problem at hand.

4.3.4 Co-design of digital and physical capabilities. The feasibility of using machine learning methods is heavily influenced by the amount and quality of data that can be used to train the machine learning agent. The same holds

for the services provided by Digital Twins. In the context of the approach presented in this book chapter, data quality and quantity can be improved by considering data concerns in the early design phases of Digital Twins. We anticipate research directions focusing on the co-design of Digital Twin services and the instrumentation of the physical asset (such as the sensor and actuator architecture, data management infrastructure, and appropriate network configuration). Such integrated approaches will enable more efficient harvesting of data on the physical asset, give rise to more sophisticated real-time data processing capabilities [12], and improve the costs associated with the engineering of Digital Twins.

4.3.5 The need for unified Digital Twin frameworks. The lack of unified Digital Twin frameworks adversely affects the engineering of their simulators [46], even if automated as discussed in this chapter. Limitations of reports on Digital Twin have been recently identified as a factor that hinders understanding and development of Digital Twin tools and techniques [30]. Experience reports often omit important characteristics of the Digital Twin, such as the scope of system-under-study, the time-scale of processing, and life-cycle stages. Standards such as the Reference Architectural Model Industry 4.0 (RAMI 4.0) [18] and various academic frameworks [1, 22] are the first steps towards alleviating this issue, although they only address select aspects of Digital Twins.

The benefits of ontology-based semantic knowledge management have been demonstrated in the design of complex heterogeneous systems [52]. The success of ontology-based techniques in the realm of cyber-physical systems [20] sets promising future directions in their application in the design of Digital Twins.

5 CONCLUSION

As the complexity of services provided by Digital Twins keeps increasing, the importance of advanced simulation capabilities to support those services grows as well. Therefore, proper automation is paramount in enabling the construction of simulators.

This chapter outlined how machine learning can be used to automate the construction of simulators in Digital Twins. We illustrated the concepts of such an approach based on DEVS and reinforcement learning. Depending on the problem at hand, other simulation formalisms and other machine learning approaches might be appropriate choices as well. However, DEVS projects to be an appropriate choice for a wide range of problems as it is commonly considered the assembly language of simulation formalisms [51]. Reinforcement learning is a natural fit for problems in which previously labeled data is not available, but model operations can be exhaustively listed. The formal foundations of DEVS provide a good starting point for the latter, allowing reinforcement learning to be leveraged for inferring DEVS models.

The approach discussed in this chapter is of particular utility in the context of systems that are challenging to model due to their complexity, such as Digital Twins. We anticipate numerous lines of research on this topic unfolding in the upcoming years.

ACKNOWLEDGEMENT

This work was partially supported by the Institute for Data Valorization (IVADO).

REFERENCES

[1] Shohin Aheleroff, Xun Xu, Ray Y. Zhong, and Yuqian Lu. 2021. Digital Twin as a Service (DTaaS) in Industry 4.0: An Architecture Reference Model. *Adv. Eng. Informatics* 47 (2021), 101225. https://doi.org/10.1016/j.aei.2020.101225

[2] Brenna D Argall, Sonia Chernova, Manuela Veloso, and Brett Browning. 2009. A survey of robot learning from demonstration. *Robotics and autonomous systems* 57, 5 (2009), 469–483.

[3] Robert G. Bartholet, David C. Brogan, Paul F. Reynolds, and Joseph C. Carnahan. 2004. In search of the philosopher's stone: Simulation composability versus component-based software design. In *Proceedings of the Fall Simulation Interoperability Workshop*.

[4] Donald J. Berndt and James Clifford. 1994. Using Dynamic Time Warping to Find Patterns in Time Series. In *Knowledge Discovery in Databases: Papers from the 1994 AAAI Workshop, Seattle, Washington, USA, July 1994. Technical Report WS-94-03*, Usama M. Fayyad and Ramasamy Uthurusamy (Eds.). AAAI Press, 359–370.

[5] Francis Bordeleau, Benoit Combemale, Romina Eramo, Mark van den Brand, and Manuel Wimmer. 2020. Towards Model-Driven Digital Twin Engineering: Current Opportunities and Future Challenges. In *International Conference on Systems Modelling and Management*. Springer, 43–54.

[6] Stefan Boschert and Roland Rosen. 2016. Digital twin—the simulation aspect. In *Mechatronic futures*. Springer, 59–74.

[7] Paul Christiano, Jan Leike, Tom B Brown, Miljan Martic, Shane Legg, and Dario Amodei. 2017. Deep reinforcement learning from human preferences. *arXiv preprint arXiv:1706.03741* (2017).

[8] Wei Chu, Lihong Li, Lev Reyzin, and Robert E. Schapire. 2011. Contextual Bandits with Linear Payoff Functions. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2011, Fort Lauderdale, USA, 2011 (JMLR Proceedings, Vol. 15)*. JMLR.org, 208–214.

[9] Benoit Combemale et al. 2020. A Hitchhiker's Guide to Model-Driven Engineering for Data-Centric Systems. *IEEE Software* (2020).

[10] Constantin Cronrath, Abolfazl R Aderiani, and Bengt Lennartson. 2019. Enhancing digital twins through reinforcement learning. In *Automation Science and Engineering*. IEEE, 293–298.

[11] Istvan David, Jessie Galasso, and Eugene 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*. ACM, 221–224.

[12] Istvan David, István Ráth, and Dániel Varró. 2018. Foundations for Streaming Model Transformations by Complex Event Processing. *Softw. Syst. Model.* 17, 1 (2018), 135–162. https://doi.org/10.1007/s10270-016-0533-1

[13] Istvan David and Eugene Syriani. 2022. DEVS Model Construction as a Reinforcement Learning Problem. In *2022 Annual Modeling and Simulation Conference (ANNSIM)*. IEEE.

[14] Istvan David, Hans Vangheluwe, and Yentl Van Tendeloo. 2018. Translating engineering workflow models to DEVS for performance evaluation. In *Winter Simulation Conference*. IEEE, 616–627.

[15] Thijs Defraeye, Chandrima Shrivastava, Tarl Berry, Pieter Verboven, Daniel Onwude, Seraina Schudel, Andreas Bühlmann, Paul Cronje, and René M. Rossi. 2021. Digital twins are coming: Will we need them in supply chains of fresh horticultural produce? *Trends in Food Science & Technology* 109 (2021), 245–258. https://doi.org/10.1016/j.tifs.2021.01.025

[16] Michael W Floyd and Gabriel A Wainer. 2010. Creation of DEVS models using imitation learning. In *Proceedings of the 2010 Summer Computer Simulation Conference*. Citeseer, 334–341.

[17] Cláudio Gomes, Casper Thule, David Broman, Peter Gorm Larsen, and Hans Vangheluwe. 2018. Co-Simulation: A Survey. *ACM Comput. Surv.* 51, 3 (2018), 49:1–49:33. https://doi.org/10.1145/3179993

[18] Martin Hankel and Bosch Rexroth. 2015. The reference architectural model industrie 4.0 (rami 4.0). *ZVEI* 2, 2 (2015), 4–9.

[19] W Bradley Knox and Peter Stone. 2012. Reinforcement learning from simultaneous human and MDP reward.. In *AAMAS*. 475–482.

[20] Zihang Li, Guoxin Wang, Jinzhi Lu, Didem Gürdür Broo, Dimitris Kiritsis, and Yan Yan. 2022. Bibliometric Analysis of Model-Based Systems Engineering: Past, Current, and Future. *IEEE Transactions on Engineering Management* (2022), 1–18. https://doi.org/10.1109/TEM.2022.3186637

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

[22] Yuqian Lu, Chao Liu, Kevin I-Kai Wang, Huiyue Huang, and Xun Xu. 2020. Digital Twin-driven smart manufacturing: Connotation, reference model, applications and research issues. *Robotics Comput. Integr. Manuf.* 61 (2020), 101837. https://doi.org/10.1016/j.rcim.2019.101837

[23] Richard J. Malak and Chris J. J. Paredis. 2004. Foundations of Validating Reusable Behavioral Models in Engineering Design Problems. In *Proceedings of the 36th conference on Winter simulation*. IEEE Computer Society, 420–428.

[24] Marius Matulis and Carlo Harvey. 2021. A robot arm digital twin utilising reinforcement learning. *Computers & Graphics* 95 (2021), 106–114.

[25] Simon Van Mierlo, Yentl Van Tendeloo, Istvan David, Bart Meyers, Addis Gebremichael, and Hans Vangheluwe. 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, Baltimore, MD, USA, April 15-18, 2018*. ACM, 6:1–6:12.

[26] Marvin Minsky. 1965. Matter, mind and models. (1965).

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

[28] Andrew Y. Ng and Stuart Russell. 2000. Algorithms for Inverse Reinforcement Learning. In *Proceedings of the Seventeenth International Conference on Machine Learning (ICML 2000), Stanford University, Stanford, CA, USA, June 29 - July 2, 2000*. Morgan Kaufmann, 663–670.

[29] James R Norris and James Robert Norris. 1998. *Markov chains*. Number 2. Cambridge university press.

[30] Bentley James Oakes, Ali Parsai, Bart Meyers, Istvan David, Simon Van Mierlo, Serge Demeyer, Joachim Denil, Paul De Meuleneare, and Hans Vangheluwe. 2023. *A Digital Twin Description Framework and its Mapping to Asset Administration Shell*. Springer. To appear. Preprint available: https://arxiv.org/abs/2209.12661.

[31] Ramón P Otero, David Lorenzo, and Pedro Cabalar. 1995. Automatic induction of DEVS structures. In *International Conference on Computer Aided Systems Theory*. Springer, 305–313.

[32] Ernest H. Page and Jeffrey M. Opper. 1999. Observations on the complexity of composable simulation. In *Proceedings of the 31st conference on Winter simulation*. WSC, 553–560.

[33] Magnus Persson, Martin Törngren, Ahsan Qamar, Jonas Westman, Matthias Biehl, Stavros Tripakis, Hans Vangheluwe, and Joachim Denil. 2013. A characterization of integrated multi-view modeling in the context of embedded and cyber-physical systems. In *Proceedings of the International Conference on Embedded Software, EMSOFT 2013, Montreal, QC, Canada, September 29 - Oct. 4, 2013*. IEEE, 10:1–10:10. https://doi.org/10.1109/EMSOFT.2013.6658588

[34] Martin L Puterman. 1990. Markov decision processes. *Handbooks in operations research and management science* 2 (1990), 331–434.

[35] Adil Rasheed, Omer San, and Trond Kvamsdal. 2020. Digital twin: Values, challenges and enablers from a modeling perspective. *IEEE Access* 8 (2020), 21980–22012.

[36] Sheldon M. Ross. 2012. *Simulation (5. ed.)*. Academic Press.

[37] Andrey Sapronov, Vladislav Belavin, Kenenbek Arzymatov, Maksim Karpov, Andrey Nevolin, and Andrey Ustyuzhanin. 2018. Tuning hybrid distributed storage system digital twins by reinforcement learning. *Advances in Systems Science and Applications* 18, 4 (2018), 1–12.

[38] Robert R Schaller. 1997. Moore's law: past, present and future. *IEEE spectrum* 34, 6 (1997), 52–59.

[39] Bernhard Schölkopf. 2000. The Kernel Trick for Distances. In *Advances in Neural Information Processing Systems 13, Papers from Neural Information Processing Systems (NIPS) 2000, Denver, CO, USA*. MIT Press, 301–307.

[40] John Schulman, Sergey Levine, Pieter Abbeel, Michael I. Jordan, and Philipp Moritz. 2015. Trust Region Policy Optimization. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015 (JMLR Workshop and Conference Proceedings, Vol. 37)*. JMLR.org, 1889–1897.

[41] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).

[42] Burr Settles. 2012. Active learning. *Synthesis lectures on artificial intelligence and machine learning* 6, 1 (2012), 1–114.

[43] Michael Spiegel, Paul F. Reynolds Jr., and David C. Brogan. 2005. A case study of model context for simulation composability and reusability. In *Proceedings of the 37th Winter Simulation Conference*. IEEE Computer Society, 437–444.

[44] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction*. MIT press.

[45] Eugene Syriani and Hans Vangheluwe. 2010. *Discrete-Event Modeling and Simulation: Theory and Applications*. CRC Press, Boca Raton, Book section DEVS as a Semantic Domain for Programmed Graph Transformation, 3–28. https://doi.org/Discrete-Event-Modeling-and-Simulation-Theory-and-Applications/Wainer-Mosterman/9781420072334

[46] Fei Tao, He Zhang, Ang Liu, and Andrew Y. C. Nee. 2019. Digital Twin in Industry: State-of-the-Art. *IEEE Trans. Ind. Informatics* 15, 4 (2019), 2405–2415. https://doi.org/10.1109/TII.2018.2873186

[47] Nikita Tomin et al. 2020. Development of Digital Twin for Load Center on the Example of Distribution Network of an Urban District. In *E3S Web of Conferences*, Vol. 209.

[48] Lisa Torrey and Jude Shavlik. 2010. Transfer learning. In *Handbook of research on machine learning applications and trends: algorithms, methods, and techniques*. IGI global, 242–264.

[49] Nelson Vadori 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*. 1–9.

[50] Simon Van Mierlo, Bentley James Oakes, Bert Van Acker, Raheleh Eslampanah, Joachim Denil, and Hans Vangheluwe. 2020. Exploring Validity Frames in Practice. In *Systems Modelling and Management*. Springer, 131–148.

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

[52] Ken Vanherpen, Joachim Denil, Istvan David, Paul De Meulenaere, Pieter J. Mosterman, Martin Törngren, Ahsan Qamar, and Hans Vangheluwe. 2016. Ontological reasoning for consistency in the design of cyber-physical systems. In *1st International Workshop on Cyber-Physical Production Systems, CPPS@CPSWeek 2016, Vienna, Austria, April 12, 2016*. IEEE Computer Society, 1–8. https://doi.org/10.1109/CPPS.2016.7483922

[53] Marco A Wiering and Martijn Van Otterlo. 2012. Reinforcement learning. *Adaptation, learning, and optimization* 12, 3 (2012).

[54] Bernard P Zeigler, Alexandre Muzy, and Ernesto Kofman. 2018. *Theory of modeling and simulation: discrete event & iterative system computational foundations*. Academic press.