SN Computer Science (2023) 4:156
https://doi.org/10.1007/s42979-022-01575-2

ORIGINAL RESEARCH

Enhancing Deep Reinforcement Learning with Scenario-Based Modeling

Raz Yerushalmi[1,2] · Guy Amir[2] · Achiya Elyasaf[3] · David Harel[1] · Guy Katz[2] · Assaf Marron[1]

Received: 21 November 2022 / Accepted: 16 December 2022

Abstract
Deep reinforcement learning agents have achieved unprecedented results when learning to generalize from unstructured data. However, the "black-box" nature of the trained DRL agents makes it difficult to ensure that they adhere to various requirements posed by engineers. In this work, we put forth a novel technique for enhancing the reinforcement learning training loop, and specifically—its reward function, in a way that allows engineers to *directly* inject their expert knowledge into the training process. This allows us to make the trained agent adhere to multiple constraints of interest. Moreover, using scenario-based modeling techniques, our method allows users to formulate the defined constraints using advanced, well-established, behavioral modeling methods. This combination of such modeling methods together with ML learning tools produces agents that are both high performing and more likely to adhere to prescribed constraints. Furthermore, the resulting agents are more transparent and hence more maintainable. We demonstrate our technique by evaluating it on a case study from the domain of internet congestion control, and present promising results.

Keywords Machine learning · Deep reinforcement learning · Scenario-based modeling · Rule-based specifications · Domain expertise

Introduction

Deep neural networks (*DNNs*) have been achieving state-of-the-art results in addressing various tasks. Specifically, *deep reinforcement learning* (*DRL*) has recently emerged as a popular method for training DNNs, in the case of learning from unstructured data—e.g., in games [1], autonomous driving [2], smart city communications [3], manufacturing [4], and more [5, 6]. As this trend continues, it is likely that DRL will gain a foothold in many systems of critical importance.

Although DRL-based agents have proven to be highly successful, and demonstrate superior performance to that of hand-crafted agents, problematic aspects of this paradigm have been observed. One such setback is that there might be a discrepancy between the empirical distribution on which the DRL agent is trained on, and the actual distribution that the agent encounters during its deployment [7]. In addition, due to the fact the DRL agents employ DNNs, they are susceptible to the various vulnerabilities that exist in many DNNs [8], such as sensitivity to adversarial perturbations, and more. When such issues are discovered, the DRL agent often needs to be modified in an additional training process;

This article is part of the topical collection "Advances on Model-Driven Engineering and Software Development" guest edited by Luís Ferreira Pires and Slimane Hammoudi.

✉ Raz Yerushalmi
raz.yerushalmi@weizmann.ac.il

Guy Amir
guyam@cs.huji.ac.il

Achiya Elyasaf
achiya@bgu.ac.il

David Harel
dharel@weizmann.ac.il

Guy Katz
guykatz@cs.huji.ac.il

Assaf Marron
assaf.marron@weizmann.ac.il

1 Department of Computer Science and Applied Mathematics, Weizmann Institute of Science, 76100 Rehovot, Israel

2 School of Computer Science and Engineering, The Hebrew University of Jerusalem, 91904 Jerusalem, Israel

3 Software and Information Systems Engineering, Ben-Gurion University of the Negev, 8410501 Beer-Sheva, Israel

Published online: 11 January 2023

however, such actions are extremely difficult, due to DRL systems largely being considered as "black box" models [9]. Specifically, their underlying decision-making process is almost impossible to directly interpret. In addition, the DRL training process itself is highly time-consuming, rendering it infeasible to retrain the agent whenever there is a change in either circumstances, or requirements.

In recent years, many works have been published on DNN interpretability and explainability [10, 11]. These also include tailored techniques that incorporate formal methods to facilitate DRL-driven reasoning [12]. Although these works demonstrate some progress, they are still nascent, and typically suffer from limited scalability, and are not suitable in the case of industrial-used DRL agents.

In this paper, we address this important gap, by integrating *modeling techniques* into the classic DRL training loop. The main idea behind our approach is to leverage the strengths of classical specification frameworks, which are typically applied within procedural or rule-based modeling, and carry these advantages over to the training loops of DRL agents, enabling explicit, classical specifications, to directly affect the DRL training. Specifically, we propose to occasionally override the calculated *reward function* [13] of the DRL agent, which is then reflected in the *return* that the agent attempts to maximize during its training. Using our process, engineers may formulate domain knowledge as a specification in their modeling formalism of choice, and then inject that knowledge into the computed reward so that it reflects the level in which the specification is satisfied. This eventually generates a DRL agent that better conforms to the predefined requirement specifications of the system.

Our proposed approach is highly generalizable, and allows integration of different modeling formalisms into the DRL process. In order to demonstrate a proof-of-concept of our approach, and for evaluation purposes, we focus on the *scenario-based modeling* (*SBM*) [14, 15] scheme. In the SBM modeling scheme, a modeler creates a collection of small, individual scenarios, where each one of these scenarios reflects a specific behavior of the system, either desirable or undesirable. These scenarios are executable, and bring about a fully executable model of the desired global behavior of the system in question when combined together. We note that one of the key features of SBM is the ability of each single scenario to allow the user to specify *forbidden* (unwanted) behavior, which the system as a whole should avoid. SBM methods have demonstrated effectiveness in modeling systems from numerous domains, including webservers [16], cache coherence protocols [17], gaming [18], production control [19], transportation systems [20], biological modelings [21], and others.

During the DRL training loop, the agent may be considered as a reactive system: it receives an input from the environment representing the observation in the current time-step, reacts accordingly, and subsequently receives a reward depending on its actions, based on which the agent may adjust its behavior for the future. In order to integrate SBM and DRL, we developed a novel method, which executes the scenario-based model and the DRL training loop in parallel. Then, we can modify the reward values in order to penalize the agent whenever it performs an action that is forbidden by the scenario-based model. We argue that this process would increase the likelihood that the agent will learn the constraints that are expressed through the scenario-based model, in addition to learning its original goals.

As a case study, we chose to demonstrate our approach on Aurora [22], which is a DRL-based agent, trained to optimize Internet congestion control. The Aurora agent is deployed at the sender node of a communication networking system, and controls the sending rate of packets sent out from this node, with the goal of maximizing effective throughput of packets. As part of our evaluation, we demonstrate how scenario-based specifications can be used in order to improve the training of the Aurora DRL agent, and specifically, encourage the agent to demonstrate *fairness*, by preventing it from repeatedly increasing the sending rate on the possible expense of other senders deployed on the same link. Our experiments show that an agent trained by our SBM-driven method is far less likely to exhibit the unwanted behavior, when compared to an agent trained by DRL alone, thus highlighting the potential of our approach.

The work described in this paper extends our previous work [23], in several ways. Most notably, it introduces a definition of the operational semantics of our new framework; and it also includes an in-depth explanation about the integration of SBM with the AI-Gym [24] DRL training system.

The rest of the paper is organized as follows: in "Background" section, we provide a comprehensive background on scenario-based modeling and on the deep reinforcement learning paradigm. In "Integration of SBM into the Reward" section, we describe our method, which integrates concepts from SBM and DRL, and we describe our experimental results in "Case Study: the Aurora Congestion Controller DRL Agent" section. Related work is described in "Related Work" section, and we conclude our research in "Conclusion and Future Work" section.

Background

Scenario-Based Modeling

Scenario-based modeling (SBM) [14, 15, 25] is a modeling paradigm, which is designed to facilitate the development of reactive systems from components with human-understandable behavior. SBM methods focus on inter-object, system-wide behaviors, and thus differ from object-centric paradigms, that are more conventional. In

SBM, a system is comprised of components called *scenario objects*, each of these components describing a single behavior (desired or undesired) of the system. This behavior can formally be modeled as a sequence of events. We note that scenario-based models are fully executable: when run, all the scenario objects run synchronously in parallel, resulting in a cohesive system behavior.

Thus, the resulting model complies with the constraints and the requirements of all the participating scenario objects [15, 25].

More concretely, each scenario object may be regarded as a transition system, whose states are referred to as *synchronization points*. The scenario object can transition between synchronization points, based on *events* triggered by a global *event-selection mechanism*. At each synchronization point, the scenario object declares events that it *requests* and events that it *blocks*. These declarations encode, respectively, the desirable and forbidden actions of the system, from the perspective of the particular scenario object. In addition, scenario objects can also declare events for which they *wait*, hence asking to be notified once these events occur. After making those declarations, the scenario object is suspended until the occurrence of an event that it requested or waited for, at which point the scenario resumes and may transition to another synchronization point.

During execution, the scenario objects run in parallel, until they all reach a given synchronization point. Then, the event-selection mechanism collects declarations of all requested and blocked events. This mechanism first selects, and then triggers, one of the events requested by at least one of the scenario objects and that is not blocked by any scenario.

Figure 1 (originally from Ref. [26]) depicts a simple scenario-based model of a toy system for controlling the water temperature and fluid levels in a water tank. Each of the scenario objects is depicted as a separate transition system, in which the nodes represent synchronization points. The edges, which represent transitions, are in accordance to the preceding node's requested or waited-for events. The scenarios AddColdWater and AddHotWater repeatedly wait for the WaterLow events to occur, and consecutively

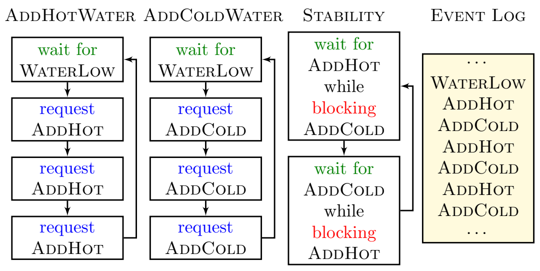

Fig. 1 (Originally borrowed from Ref. [26]) A scheme depicting a SB model for controlling the temperature and fluid levels in a water tank

request the events AddCold or AddHot, respectively, three times. Since all of these six events may be triggered in any arbitrary order, a new scenario Stability is introduced, in order to keep a stable temperature level. This, in turn, alternately blocks AddCold and AddHot, and thus enforces the interleaving of these events. The event log depicts the resulting execution trace.

There are various ways to customize the policies for selecting which event should be triggered next, out of the pool of all events that are requested (and not blocked). Common policies for selecting the next event include randomized selection, arbitrary selection, a selection based on a look-ahead for achieving certain outcomes, or a selection based on some predefined priorities [27, 28].

In practice, SBM is supported in various frameworks, both visual and textual. Such examples include implementations in multiple high-level languages, i.e., C, C++, Java, Erlang, and Python (see, e.g., Ref. [15]); other examples include the language of *live sequence charts* (LSC), where SBM modal sequence diagrams [14, 25] are produced using SBM-based concepts; and different domain-specific languages and extensions [20, 29, 30].

An additional trait that renders SBM to be extremely useful is that the SBM-driven models facilitate compositional verification and is amenable to model checking [17, 18, 31–34]. Thus, formal verification techniques can usually be applied, in order to ensure that a SBM satisfies multiple criteria of choice, either as an independent standalone model, or as a single component within a larger system. When executing scenario-based models in distributed architectures [35–39], these model can automatically be repaired by the use of automated analysis techniques [26, 40, 41]. These techniques are able to update the SBMs in various strategies, i.e., as part of the Wise Computing initiative [42–45].

For our work presented here, namely injecting expert-specific knowledge into the DRL training loop, the choice of SBM is attractive primarily due to its natural alignment with experts' description regarding the requirements specification of the system in hand. Furthermore, SBM's formalism, executability, and facilitating incremental development [46, 47] provides additional benefits.

Indeed, although previous research also demonstrated the use of SBM to complement DRL, there is significant difference—past work has focused mainly on *guarding* an already-trained DRL agent, rather than using SBM to actively affect what the agent actually learns [48, 49], as we demonstrate in this work.

Deep Reinforcement Learning

Deep reinforcement learning [13] is a popular paradigm that is used to automatically train an agent on making decisions for a given task. During training, the agent attempts to maximize an accumulated *reward* (according to some predefined reward function) through consecutive interactions with its environment, in multiple time-steps.

Figure 2 depicts a scheme of the basic DRL training cycle. In the common setting, the agent interacts with an environment at discrete time-steps $t \in \{0, 1, 2, 3, \dots\}$. At each single time-step t, the agent observes a state s_t representing the current environment, and selects a single action a_t accordingly. In the next time-step $t+1$, the agent receives a reward $R_t = R(s_t, a_t)$, as a result of its action a_t at time-step t. Next, the environment moves to the next state s_{t+1}, and the cycle continues. Through this interaction, a policy function $f : s_t \rightarrow a_t$ is gradually learnt by the agent, which attempts to maximize its *return* G_t, the future cumulative discounted reward [13]:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

where R_t is the reward at time-step t, and $\gamma \in [0, 1]$ is a *discount rate* factor, indicating the influence of past decisions on the future.

It is generally accepted that the specification of an appropriate reward function R_t is crucial in order for the agent to succeed in learning an optimal policy during the DRL training cycle. Consequently, this issue has received significant attention [13, 50, 51] in recent years. As we explain later, our approach is in fact complementary to this line of research: we suggest to augment the reward function based on the specifications and constraints that are provided by domain-specific experts.

Integration of SBM into the Reward

Overview of Our Approach

Our proposed method integrates a scenario-based model into the DRL training cycle, in order to instruct the agent to obey (during training) the specifications and constraints that are embodied in the scenarios constituting the model.

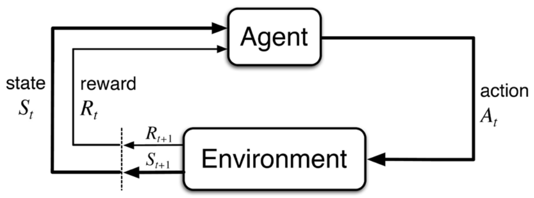

Fig. 2 (Originally borrowed from Ref. [13]) The agent–environment interaction in RL

Specifically, according to SBM modeling principles, we create a mapping from each possible action a_t of the DRL agent, to a dedicated event in the scenario-based model, e_{a_t}. This mapping allows the scenario objects to keep track of, and react to, the agent's actions. We refer to these e_{a_t} events as *external events*, to indicate that their request originates from outside the SBM model.

Once an external event e_{a_t} is triggered by the DRL agent, we allow the scenario objects to react and update their internal states, possibly triggering additional ("internal") events. In other words, the desired outcome is that the triggering of an external event will be followed by a sequence of internal events, and the process will then repeat itself.

To bring this about, we adopt the concept of *super-steps* [27]. The semantics of the scenario-based model is thus altered as follows. First, the scenario-based model runs normally, until it reaches a synchronization point in which no events are *enabled* (i.e., requested and not blocked). It then waits for a new external event e_{a_t} to be triggered. Once the DRL agent performs an action a_t (which is mapped to e_{a_t}), a dedicated scenario in the scenario-based model will request e_{a_t} in order for it to be triggered (unless it is *blocked*). Then, all scenarios that waited for e_{a_t} continue to their next synchronization point, in which other events may be enabled. Execution then continues normally until no events are enabled, at which time the model waits for another external event, and the process repeats itself. We elaborate on these semantics in "Operational Semantics: Scenario-Based Models with External Events" section.

The aforementioned execution scheme allows the scenario-based model to observe actions of the DRL agent, but does not allow any feedback to be sent to the agent. To support this, we make the following adjustment: when an external event is selected by the DRL agent and is passed along to the scenario-based model, the SBM execution engine checks whether that external event is currently *blocked* by any of the scenario objects. If so, information is sent back to the DRL training framework, so that the DRL agent is *penalized* through the reward, as elaborated next.

Affecting the Agent's Reward Value

In the framework we built for executing DRL under SBM constraints, the scenario-based model is executed alongside the agent under training. Let $\tilde{s}_t$ denote the scenario-based model's state at time-step t. The agent's reward function at time-step t, denoted R_t, is computed as follows: (i) at time-step t, the agent selects an action a_t, which is mapped to e_{a_t}; (ii) the environment reacts to the action a_t, transitions to a new state s_{t+1}, and then a candidate reward value $\tilde{R}_t$ is computed; (iii) the scenario-based model receives e_{a_t}, and if e_{a_t} is not blocked, the model executes a transition to a new state $\tilde{s}_{t+1}$; (iv) if e_{a_t} is *blocked* in state $\tilde{s}_t$, then the scenario-based model decreases the reward—and effectively *penalizes* the agent:

$$R_t = \begin{cases} \alpha \cdot \tilde{R}_t - \Delta & ; \text{if } \tilde{s}_t \xrightarrow{e_{a_t}} \tilde{s}_{t+1} \text{ is blocked} \\ \tilde{R}_t & ; \text{otherwise} \end{cases} \tag{1}$$

for some constants $\alpha \in [-1, 1]$ and $\Delta \geq 0$; and (v) the reward R_t at this time-step is returned to the agent.

The novel training procedure is depicted in Fig. 3. These aforementioned changes to the common training loop are motivated by the intention to allow the DRL agent to learn a policy that satisfies the requirements encoded in the original reward function, while still learning to satisfy scenario-encoded constraints. This is done without the need to change the interface between the agent and its original environment. We also point out that although the scenario-based model is unaware of the environment state s_t, it is aware of its internal state, as well as all the agent's performed actions, except the actions mapped to *blocked events*. We leave for future work the task of allowing the scenario objects to view the environment state as well—as this may be required in more complex systems. The DRL agent's execution of undesired actions, associated with events blocked by the SBM, is left for future work as well.

The learnt policy may be significantly affected by the chosen penalty policy, i.e., the selected α and Δ constants. Thus, the delicate balance between encouraging the agent to obey the SBM specifications on one hand, and on the other hand

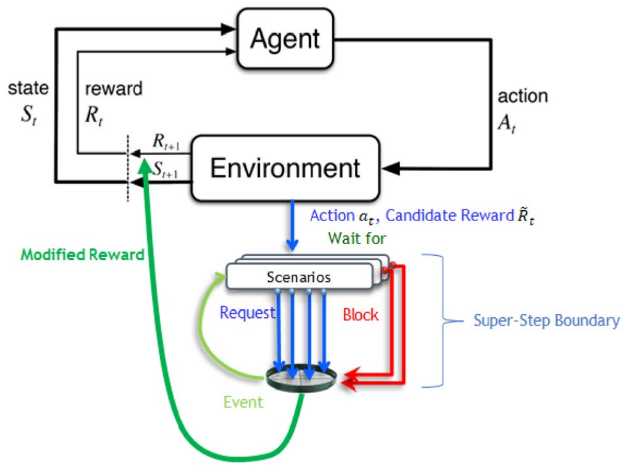

Fig. 3 The DRL-SBM execution framework: at each time-step, a candidate reward $\tilde{R}_t$ is calculated by the environment, based on state s_t, and the agent's action a_t. The SB model is executed in parallel, running a super-step after receiving e_{a_t}, and may reduce the reward $\tilde{R}_t$ if e_{a_t} is a forbidden action (i.e., a blocked event) in the current state $\tilde{s}_t$

allowing it to learn a policy that solves the original problem, should be taken into consideration by the user.

As a toy example, we once again refer the reader to the example depicted in Fig. 1, but this time from the perspective of the DRL agent. Suppose we wish to train a DRL agent to respond to a *WaterLow* event by adding equal amounts of cold water and hot water, and suppose we also wish to inject a constraint indicating that the agent should avoid two consecutive additions of cold water or hot water. This can be achieved by designing a basic scenario-based model that comprises a scenario indicating *Stability* (Fig. 1), and integrating this new scenario into the agent's training cycle. As a result, this new scenario object would penalize the DRL agent when it performs the undesirable sequence of actions, effectively encouraging it towards learning the correct, desired, policy of interleaving.

Operational Semantics: Scenario-Based Models with External Events

We begin by describing a common execution semantics for SBM, which is implemented within the BP-Py framework [52]. Various SBM implementations, like in the original LSC language [25, 53], provide a variety of ways to support the injection of environment events into the scenario-based execution. The BP-Py environment did not include such support initially, and so we added necessary aspects of this support as part of the DRL-SBM execution framework, as detailed later.

The original implementation of BP-Py performs the following steps during the execution of a scenario-based model:

a. Initialize the scenario objects: each scenario object reaches its first synchronization point and declares the sets of events that it requests, waits-for and blocks.
b. Run in an infinite loop:

(i) Check for an enabled event (an event that is requested and not blocked).
(ii) If there are no enabled events, break; the program has finished its run.
(iii) Select one enabled event, either randomly or per some pre-specified event-selection strategy.
(iv) Mark the selected event as triggered.
(v) Inform the scenario objects of the triggered event. Each scenario object that requested or waited for this event wakes up, proceeds to its next synchronization point, and declares new sets of events that it requests, waits-for, and blocks.

Figure 4 depicts pseudo-code for the main loop of the SBM core, using these semantics.

```
# Main loop
while True:
    if noEnabledEvents():
        terminateExecution()

    event = selectEnabledEvent()
    advanceAllBThreads(event)
```

Fig. 4 (Borrowed from Ref. [23]) A pseudo-code block of the main execution loop, in a common SB model. The execution of the main loop is terminated when no enabled events, i.e., requested events that are not blocked, remain. We note the assumption that scenario objects have been initialized prior to the execution of the loop

In order to support integration with an external environment, such as a DRL agent's training environment, we propose to add support for *super-steps* within the BP-Py implementation, as we described earlier with the DRL-SBM execution framework. Recall that a super-step is a set of consecutive steps in the scenario-based model, which begins with an external event and continues for as long as the scenario-based model can execute steps, i.e., has enabled events to choose from. A super-step ends when the scenario-based model reaches a *stable configuration*, i.e., a configuration in which no event is enabled, and consequently no scenario object can progress. The execution environment then waits for the next external event. Observe that the external event does not compete with internal events—it can only be inserted into the model when the model is in a stable configuration. Thus, in fact, the execution cycle assumes that the scenario-based model is a closed system; i.e., the super-step runs once, from start to finish (or ad infinitum), without receiving any external events.

We now define a modified version of the steps performed by the DRL-SBM execution framework, this time with the intention of supporting super-step execution:

1. Initialize the scenario objects: each scenario object reaches its first synchronization point and declares the sets of events that it requests, waits-for and blocks.
2. Run until a stable configuration is reached, i.e., while an enabled event exists:
    (a) Select one enabled event, either randomly or per some pre-specified event-selection strategy.
    (b) Mark the selected event as triggered.
    (c) Inform the scenario objects of the triggered event. Each scenario object that requested or waited for this event wakes up, proceeds to its next synchronization point, and declares new sets of events which it requests, waits-for and blocks.
3. Run in an infinite loop:
    (a) Wait for a request to trigger an external event e_{a_t} (coming from the next action of the DRL agent), and then add the event to the list of requested events, using a dedicated scenario.
    (b) Check whether the external event e_{a_t} is declared as *blocked*.
    (c) If yes, modify the step reward using formula (1).
    (d) If e_{a_t} is not declared as *blocked*, then while an enabled event exists:
        (i) Select one enabled event, either randomly or per some pre-specified event-selection strategy.
        (ii) Mark the selected event as triggered.
        (iii) Inform the scenario objects of the triggered event. Each scenario object that requested or waited for this event wakes up, proceeds to its next synchronization point, and declares new sets of events which it requests, waits-for and blocks.

In this design, we force the scenario-based model to return to a stable configuration before another action of the DRL agent is taken and the associated external event is triggered. Consequently, there cannot be more than a single pending external event. Figure 5 depicts the pseudo-code of the modified, high-level loop of our implementation.

As a small example, consider again the scenario-based model in Fig. 1. Assume that we now wish to use that model to control an actual, physical water tank. To accomplish this, we can define the *WaterLow* as an *external event*, and then run a super-step of the model each time that event is triggered.

Case Study: the Aurora Congestion Controller DRL Agent

As a case study for evaluating our technique, we decided to focus on the Aurora agent [22], which is a DRL-based agent implementing an Internet congestion control algorithm. The agent is deployed at the sender node of a communication system and controls the sending rate of packets sent from the node, with the goal of optimizing the link's throughput. During each time-step, the agent receives a state vector with statistics regarding the link's observed throughput, latency, and the percent of packets that were previously lost. At each time-step, the agent outputs a single action, indicating a change to the sending rate of packets in the next time-step. The Aurora agent is intended to replace earlier, hand-crafted Internet protocols (e.g., TCP) for obtaining similar goals, and has previously demonstrated excellent performance [22].

```
# Modified main loop, with super—steps
while True:
    # Handle an agent action
    action = waitForAgentAction()
    if isBlocked( action ):
        penalizeAgentReward()
    else:
        keepOriginalAgentReward()
        advanceAllBThreads( action )
        # Perform super—step
        while haveEnabledEvents():
            event = selectEnabledEvent()
            advanceAllBThreads( event )
```

Fig. 5 (Adopted from Ref. [23]) A pseudo-code block of the main loop of DRL-SBM execution framework. The model waits for the agent action, penalizes the agent if it chose an action that is blocked, or performs a super-step otherwise. We note the assumption that the model reached a stable configuration after its initialization, and before it entered the main loop

The authors of Aurora raised an important point regarding the system's fairness, asking: *can our RL agent be trained to "play well" with other protocols (TCP, PCC, BBR, Copa)?* [22]. Indeed, in the case of DRL models, including Aurora, it is usually quite hard to train an agent to behave according to various predefined fairness properties. In our case study, we set out to use our SBM-driven training procedure, in order to inject specific fairness constraints to the learning cycle, and effectively train a fair Aurora agent. Specifically, we attempted to train the Aurora agent to achieve high throughput, while avoiding unfair and "aggressive"-like behavior. For example, we encouraged the agent to avoid becoming a "bandwidth hog" that continuously increases its sending rate, and thus prohibits other senders on the same link from having a fair share of the link's bandwidth.

Embedding SBM Super-Step Model Execution into the AI-Gym DRL Training Environment

In order to allow our proof-of-concept implementation to affect the training of an Aurora DRL agent, we leveraged AI-Gym's environment object's *step* and *reset* interfaces. Quoting from AI-Gym documentation of those two interfaces:

- Step: Runs one time-step of the environment's dynamics.
- Reset: Resets the environment to an initial state and returns an initial observation.

We treat each training episode of the DRL agent as a separate run of the scenario-based model. Thus, at the beginning of each episode we create a fresh executable copy of the entire scenario-based model, where all scenario objects are at their initial states.

In order to support the execution of a scenario-based model in super-step mode (per the semantics described in "Operational Semantics: Scenario-Based Models with External Events" section), we extended the BP-Py [52] platform with the following methods:

i. initiate_run(): initializes the scenario objects, bringing them to a stable configuration (per steps 1 and 2 of the semantics in "Operational Semantics: Scenario-Based Models with External Events" section).
ii. is_event_blocked(): checks whether the external event that was triggered is blocked (per step 3.(b) in "Operational Semantics: Scenario-Based Models with External Events" section).
iii. super-step_all_bthreads(): executes a super-step of the scenario objects (per step 3.(c) in "Operational Semantics: Scenario-Based Models with External Events" section).

Finally, in order to support step 3.(a) of the semantics, i.e., waiting for an external event and injecting it into the scenario model as a requested event, we added an external service and a dedicated scenario object to the scenario-based model itself. The service is a piece of code, completely external to the SB model, which is comprised of two methods that provide a mapping from external actions to SBM events. The code for these methods, which is tailored for the Aurora DRL agent but can easily be adapted to other systems, appears in Fig. 6.

The dedicated scenario object that was added to the model is responsible for injecting the corresponding events into the SBM execution engine; its code appears in Fig. 7. Although the code supports a list of such events, by convention we only allow a single such event in each training time-step.

Each training episode starts with a call to the *reset()* method of the customized AI-Gym *env* object that we created. In order to ensure that each training session uses a fresh instantiation of the scenario-based model, a new instance of the model is created by that *reset()* method; and its *initiate_run()* method is immediately invoked to bring it to a stable configuration.

Resetting and restarting the scenario-based model at an episode boundary allows the scenarios to monitor and constrain the agent's behavior with a perspective that includes both state-specific behavior, as well as the sequencing and flow of the agent's behavior throughout each training episode.

During each time-step of the training episode, AI-Gym invokes the *step()* method of the *env* object, which causes the agent to select an action, obtain its reward, and

```
def request_matching_action(act):
    requested_ev = None
    if act>SBP_action_treshold:
        requested_ev = BEvent("SBP_IncreaseRate")
    elif act< -SBP_action_treshold:
        requested_ev = BEvent("SBP_DecreaseRate")
    else:
        requested_ev = BEvent("SBP_KeepRate")
    return requested_ev


def append_request(action=None):
    if not action is None:
        SBP_requested_ev = request_matching_action(action)
        SBP_requested_cache.append(SBP_requested_ev)
        return SBP_requested_ev
    return None
```

Fig. 6 The Python implementation of mapping external actions to SBM events. This code is neither part of the scenario-based model, nor of the SBM execution engine. The *request_matching_action* method is tailored for the Aurora DRL agent: it lists the three possible agent actions SBP_IncreaseRate, SBP_DecreaseRate, and SBP_KeepRate, and maps them to the corresponding SBM events

```
def SBP_request_random_action():
    # global SBP_requested_cache
    while True:
        unique_list = SBP_requested_cache.copy()
        SBP_requested_cache.clear()
        yield {request: unique_list}
```

Fig. 7 The Python implementation for injecting external events into the scenario-based model execution. The scenario first waits for a new event to be placed in a queue by the *request_matching_action* method. Once such an event arrives, the scenario synchronizes and requests it (using the *yield* statement). When the synchronization call returns, the event has been triggered, and the scenario awaits another external event

observe the new environment state. In order to ensure that the scenario-based model remains synchronized with the training steps, we extended the *env* object to allow the scenario-based model to be informed of selected actions, and interfere with the calculation of the reward. The code snippet in Fig. 8 is called within the *step()* method of the customized *env* object, and invokes the service mentioned earlier.

Evaluation Setup

Using the BP-Py environment [52], along with our novel extension (described in "Embedding SBM Super-Step Model Execution into the AI-Gym DRL Training Environment" section), we generated a simplistic scenario-based model. This model, which is comprised of a single scenario, penalizes the Aurora agent in case it increases the chosen

```
 ...
SBP_requested_ev = append_request(action)
if self.b_program.is_event_blocked(SBP_requested_ev):
    reward = self.reward_penalty(reward)
else:
    self.b_program.superstep_all_bthreads()
...
```

Fig. 8 A snippet of the code invoked from within AI-Gym's *env*'s *step()* method, and which alters the agent's step reward in case the selected action is currently not allowed by the scenario-based model

```
def SBP_avoid_k_in_a_row():
    k = 3
    counter = 0
    blockedEvList = []
    waitforEvList = [BEvent("IncreaseRate"),
                     BEvent("DecreaseRate"),
                     BEvent("KeepRate")]
    while True:
        lastEv = yield{ waitFor:waitforEvList,
                        block:blockedEvList }
        if not lastEv is None:
            if lastEv == BEvent("DecreaseRate")
            or lastEv == BEvent("KeepRate"):
                counter = 0
                blockedEvList = []
            else:
                if counter == k - 1:
                    #Blocking!
                    blockedEvList.append(
                      BEvent("IncreaseRate"))
                else:
                    counter += 1
```

Fig. 9 (Borrowed from Ref. [23]) A Python implementation of a scenario used for blocking the *IncreaseRate* event after $k-1$ subsequent occurrences of the same action

sending rate for k consecutive steps. We refer the reader to Fig. 9, which includes the SBM code for three steps (i,e,, $k = 3$).

The scenario waits for three possible events that represent the agent's possible actions at each time-step: *DecreaseRate*, *IncreaseRate*, or *KeepRate*, which, respectively, represent the agent's decision to either decrease the packet sending rate, increase it or keep it as is. Whenever the agent increases the packet sending rate in $k-1$ consecutive time-steps, the *IncreaseRate* event is consecutively triggered $k-1$ times, and thus the scenario will block that event, until eventually the agent selects a different action. When a requested agent action matches one of the blocked events, the execution environment overrides the reward with a penalty, and consequently signals to the agent that the originally chosen behavior is undesirable.

As a first step in our evaluation process, we trained an Aurora agent with the original framework [22] used. Then, we compared the results of two Aurora agents that differ in the training process; one agent ($\mathcal{A}_O$) was trained with the original “vanilla” framework, while the second agent ($\mathcal{A}_E$) was trained using our DRL-SBM execution framework. Excluding these differences in the training cycles, both agents shared the same values for all the remaining configurable parameters.

In order to compute the penalty during the training of $\mathcal{A}_E$, we empirically chose the penalty function hyper-parameters (see Eq. 1) to be $\Delta = -4.5, \alpha = 0$. These values were selected in order to allow the agent to learn not only the main goals of Aurora (optimizing packet throughput on a link), but also additional constraints, as specified by the scenario-based model. We note that for $\Delta \in (-2, 0]$, the agent was unable to address the SBM-driven constraints, whereas when choosing $\Delta \in (-\infty, -10]$ the resulting agent was unable to learn its original goals, as elaborated in Ref. [22].

Evaluation Results

Using the evaluation metrics from Ref. [22], we compared the training performance of both the agents. Figure 10 depicts the exponentially weighted moving average (EWMA) reward of both the agents ($\mathcal{A}_E$ and $\mathcal{A}_O$) during training.

These results clearly demonstrate significant differences in the training times of both agents. Specifically, $\mathcal{A}_E$ needs to be trained for a much larger number of episodes, in order to learn an adequate policy with a reward value that is similar to the reward reached by $\mathcal{A}_O$. We also observe that it takes $\mathcal{A}_E$ 40,000 episodes (slightly more than 4.5 in the logarithmic scale), in order to converge to an optimal policy, compared to about 3000 episodes of $\mathcal{A}_O$.

Next, we checked the empirical frequency in which both agents increase the sending rate for three consecutive time-steps (i.e., violate the SBM constraints). As can be seen in Fig. 11, $\mathcal{A}_O$ demonstrated an average violation frequency of 9–11%.

In contrast, the SBM-trained $\mathcal{A}_E$, barely violated the SBM constraints, and demonstrated an average violation frequency of about 0.34%. For additional details, we refer the reader to Fig. 12.

Summary. The aforementioned results demonstrate a significant difference in the resulting policies of $\mathcal{A}_E$ and $\mathcal{A}_O$, when focusing on the frequency of violations of the property in question: whereas $\mathcal{A}_O$ violates the property in approximately 9–11% of the time-steps, $\mathcal{A}_E$ on the other hand barely violates the property, as expressed in the significantly low violation rate (0.34%). Although the two agents demonstrate a difference in the property in question, it is important to note that they both reach a similar overall reward during training, indicating that they were both able to generalize to an adequate policy, with respect to Aurora’s main goal. While the enhanced agent took longer to converge to this adequate policy (a fact that is not surprising, as it had to learn additional constraints), these evaluations showcase the potential of our approach in introducing SBM-driven methods to the classic DRL training cycle.

Related Work

In recent years, multiple approaches were proposed for using hand-crafted software components in order to improve the DNN training process, and enhancing the performance of DNNs during deployment. One such approach calls for *composing* DNNs and hand-crafted components; for example, see [54], where rules can override decisions made by an autonomous driving system. This approach includes different strategies for composing the DNNs and the hand-crafted components: the composition can be in *parallel* (the DNN and the code run side-by-side, and each handles a different task); *sequential*, where the output of the hand-crafted code is fed into the DNN’s input, or vice-versa; or *ensemble-based*, where the hand-crafted code and the DNN attempt to solve the same task.

Another family of approaches is *reflection-based*, in which the DNN and the hand-crafted code adjust their execution gradually, in accordance with their past performance [55–57]. In this work, we present a novel approach which can be viewed as a marriage between composition-based approaches and reflection-based approaches; a DRL agent runs alongside an SBM-driven hand-crafted model, in order to improve the DRL agent’s training process.

Prior work has explored potential combinations between SBM-driven methods and DRL. In one such attempt, Katz [48, 58] demonstrated the use SBMs to *guard* an existing agent, i.e., to override the agent’s decisions in cases where they violated the SBM. In another attempt, Elyasaf et al. [59] fine-tuned the strategy of an existing SBM, using a DRL agent; using a RoboSoccer game as a case study, they demonstrated that the DRL agent can learn a policy that guides a scenario-based player to grab a soccer ball more effectively.

In this work, we propose a technique that differs from both these approaches, and is complementary to them: instead of using a scenario-based model to guard an existing DRL agent, or using a DRL agent to guide hand-crafted models, we propose to combine scenario-based models into the DRL training procedure, and thus to improve the DRL agent a-priori to its deployment, so that it abides to any arbitrary, user-chosen, specifications injected to the SB model.

Conclusion and Future Work

Deep reinforcement learning is an excellent paradigm for training agents to succeed in many real-world tasks; but it has several strong limitations, including the inability to inject and integrate domain-expert knowledge into the training process. In this work, we propose a novel approach which aims to bridge this gap, and introduce a method for integrating classical modeling techniques into the DRL training cycle. The agents trained by our approach, as we demonstrate here, are much more likely to obey the multiple goals and restrictions which are defined by domain experts, and injected into the system through SBM scenarios. In addition, those defined scenarios increase the explainability of the generated DRL agents, explaining some of their decisions.

Going forward, we plan to study the scalability of our approach by applying it to more complex case studies, with intricate agents and various scenario-based specifications. We also intend to enhance the suggested DRL-SBM interface, which is currently quite simple. We believe this can be achieved either by updating the semantics of the scenario-based model, or by defining an event-based protocol in order to manipulate the agent's reward function in more subtle strategies. Thus, we believe the feedback from the scenario-based model will allow the agent to distinguish between actions that are only *slightly* undesirable from other actions that are *extremely* undesirable. In addition, we plan to find novel methods of encouraging an agent to execute a desirable action, i.e., by introducing new constructs to the SBM (in addition to penalizing the agent for taking undesirable actions). As we previously discussed, we also plan to explore generalizable criteria

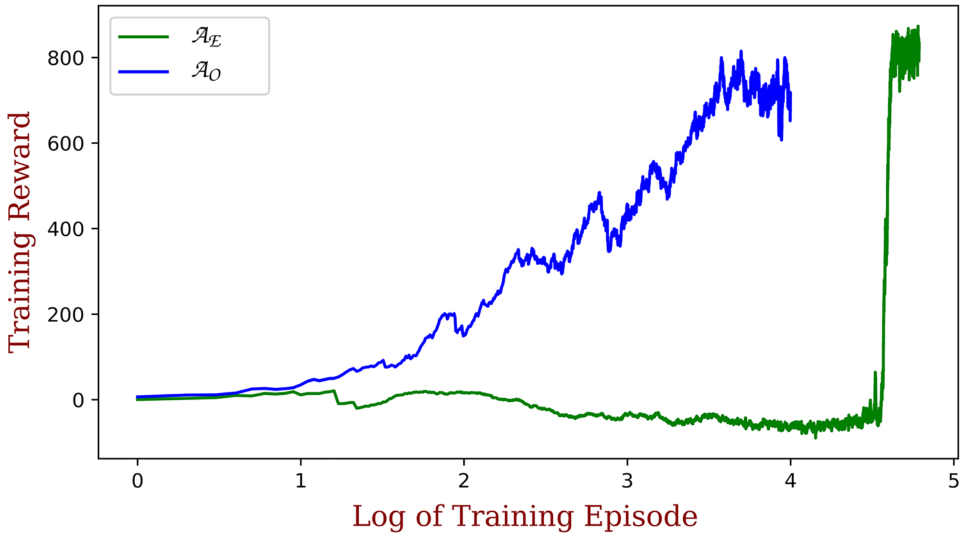

Fig. 10 (Borrowed from Ref. [23]) A comparison of the average reward obtained by both $\mathcal{A}_E$ and $\mathcal{A}_O$, as a function of the time-step (in log scale) during training

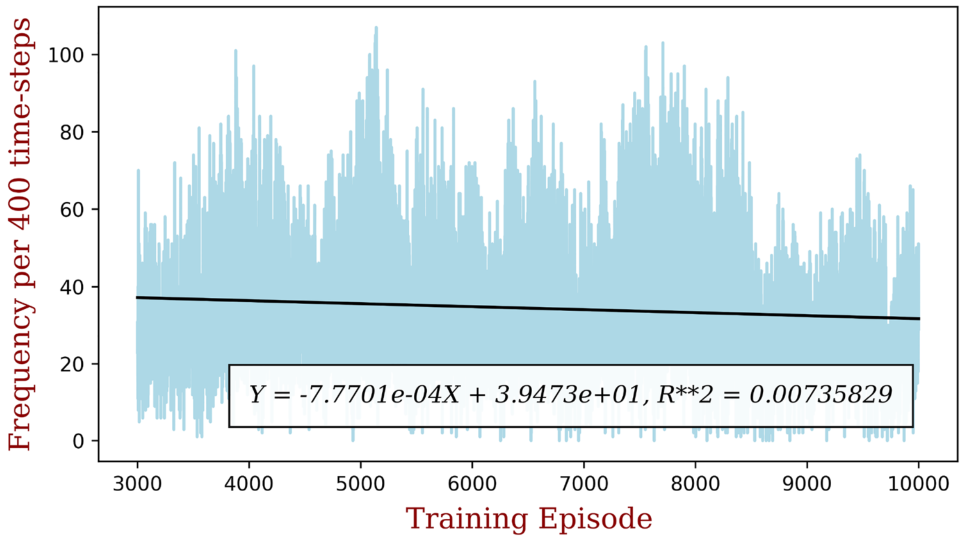

Fig. 11 (Borrowed from Ref. [23]) $\mathcal{A}_O$ performance: Every line indicates the number of times that the "vanilla" $\mathcal{A}_O$ increased the packet sending rate for three consecutive times (at least), during the training cycle. Each of the training episodes consists of 400 time-steps. The agent performed on average 35–45 violations per episode, which is equivalent to a property violation frequency of approximately 9–11%. The linear regression characterizing the violation frequency is $y = -0.00077701x + 39.47343569$

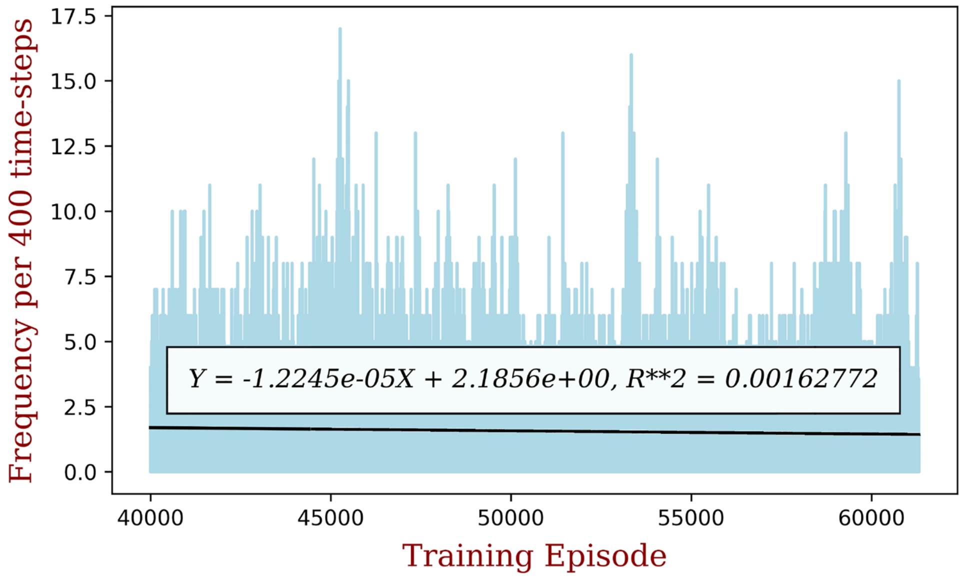

Fig. 12 (Borrowed from Ref. [23]) $\mathcal{A}_E$ performance: The lines state the number of times $\mathcal{A}_E$ increased the packet sending rate for three consecutive times (at least), during the training cycle. The $\mathcal{A}_E$ agent violated the property only 1–2 times per episode, on average. This translates to a violation frequency of about 0.34%, and to a linear regression line $y = -0.000012245x + 2.1856$

for choosing penalty values that will allow the agent to learn the wanted properties, while successfully preserving the original learning task at hand.

Finally, another angle for future research is to measure how scenario-assisted training may affect DRL. For example, by accelerating the learning of properties that could, in general, be learned without such assistance. We believe it is also interesting to evaluate multiple case studies, in order to understand if SBM-specified expert knowledge, aimed to improve system performance (instead of complying with new requirements), indeed accomplishes improvements over what the same system could learn without injecting expert-knowledge into the training cycle.

Funding The work of R. Yerushalmi, G. Amir, A. Elyasaf and G. Katz was partially supported by a grant from the Israeli Smart Transportation Research Center (ISTRC). The work of G. Amir was supported by a scholarship from the Clore Israel Foundation. The work of D. Harel, A. Marron and R. Yerushalmi was partially supported by a research grant from the Estate of Harry Levine, the Estate of Avraham Rothstein, Brenda Gruss and Daniel Hirsch, the One8 Foundation, Rina Mayer, Maurice Levy, and the Estate of Bernice Bernath, a grant 3698/21 from the ISF-NSFC joint to the Israel Science Foundation and the National Science Foundation of China, and a grant from the Minerva foundation.

Declarations

Conflict of interest The authors of this work declare that there are no conflicts of interest.

References

1. Ye D, Liu Z, Sun M, Shi B, Zhao P, Wu H, Yu H, Yang S, Wu X, Guo Q, Chen Q, Yin Y, Zhang H, Shi T, Wang L, Fu Q, Yang W, Huang L. Mastering complex control in MOBA games with deep reinforcement learning. In: Proc. 34th AAAI conf. on artificial intelligence (AAAI); 2020. p. 6672–9.
2. Kiran B, Sobh I, Talpaert V, Mannion P, Sallab A, Yogamani S, Perez P. Deep reinforcement learning for autonomous driving: a survey. IEEE Trans Intell Transp Syst. 2021;1–18.
3. Xia Z, Xue S, Wu J, Chen Y, Chen J, Wu L. Deep reinforcement learning for smart city communication networks. IEEE Trans Ind Inform. 2021;17(6):4188–96.
4. Li J, Pang D, Zheng Y, Guan X, Le X. A flexible manufacturing assembly system with deep reinforcement learning. Control Eng Practice. 2022;118: 104957.
5. Elyasaf A. Inform Softw Technol. Context-oriented behavioral programming. 2021;133: 106504.
6. Mohamad Suhaili S, Salim N, Jambli M. Service chatbots: a systematic review. Exp Syst Appl. 2021;184: 115461.
7. Eliyahu T, Kazak Y, Katz G, Schapira M. Verifying learning-augmented systems. In: Proc. conf. of the ACM special interest group on data communication on the applications, technologies, architectures, and protocols for computer communication (SIGCOMM); 2021. p. 305–18.
8. Szegedy C, Zaremba W, Sutskever I, Bruna J, Erhan D, Goodfellow I, Fergus R. Intriguing properties of neural networks. Technical Report. 2013. Preprint at arXiv:1312.6199
9. Goodfellow I, Bengio Y, Courville A. Deep learning. Cambridge: MIT Press; 2016.
10. Ribeiro M, Singh S, Guestrin C. Why should I trust you?: Explaining the predictions of any classifier. In: Proc. 22nd ACM SIGKDD int. conf. on knowledge discovery and data mining; 2016. p. 1135–44.
11. Samek W, Wiegand T, Müller K. Explainable artificial intelligence: understanding, visualizing and interpreting deep learning models. ITU J: Impact Artif Intell (AI) Commun Netw Serv. 2018;1(1):39–48.
12. Kazak Y, Barrett C, Katz G, Schapira M. Verifying Deep-RL-Driven Systems. In: Proc. 1st ACM SIGCOMM Workshop on Network Meets AI & ML (NetAI); 2019. p. 83–89.
13. Sutton R, Barto A. Introduction to reinforcement learning. Cambridge: MIT Press; 2018.
14. Damm W, Harel D. LSCs: breathing life into message sequence charts. J Form Methods Syst Des (FMSD). 2001;19(1):45–80.

15. Harel D, Marron A, Weiss G. Behavioral programming. Commun ACM (CACM). 2012;55(7):90–100.
16. Harel D, Katz G. Scaling-up behavioral programming: steps from basic principles to application architectures. In: Proc. 4th SPLASH workshop on programming based on actors, agents and decentralized control (AGERE!); 2014. p. 95–108.
17. Katz G, Barrett C, Harel D. Theory-aided model checking of concurrent transition systems. In: Proc. 15th int. conf. on formal methods in computer-aided design (FMCAD); 2015. p. 81–8.
18. Harel D, Lampert R, Marron A, Weiss G. Model-checking behavioral programs. In: Proc. 9th ACM int. conf. on embedded software (EMSOFT); 2011. p. 279–88.
19. Harel D, Kugler H, Weiss G. Some methodological observations resulting from experience using LSCs and the play-in/play-out approach. In: Scenarios: models. Transformations and tools. Berlin: Springer; 2005. p. 26–42.
20. Greenyer J, Gritzner D, Katz G, Marron A. Scenario-based modeling and synthesis for reactive systems with dynamic system structure in scenario tools. In: Proc. 19th ACM/IEEE int. conf. on model driven engineering languages and systems (MODELS); 2016. p. 16–23.
21. Kugler H, Marelly R, Appleby L, Fisher J, Pnueli A, Harel D, Stern M, Hubbard J, et al. A scenario-based approach to modeling development: a prototype model of C. Elegans vulval fate specification. Dev Biol. 2008;323(1):1–5.
22. Jay N, Rotman N, Godfrey B, Schapira M, Tamar A. A deep reinforcement learning perspective on internet congestion control. In: Proc. 36th int. conf. on machine learning (ICML); 2019. p. 3050–9.
23. Yerushalmi R, Amir G, Elyasaf A, Harel D, Katz G, Marron A. Scenario-assisted deep reinforcement learning. In: Proc. 10th int. conf. on model-driven engineering and software development (MODELSWARD); 2022. p. 310–9.
24. Brockman G, Cheung V, Pettersson L, Schneider J, Schulman J, Tang J, Zaremba W. OpenAI Gym. Technical Report. 2016. Preprint at arXiv:1606.01540
25. Harel D, Marelly R. Come, let's play: scenario-based programming using LSCs and the play-engine. Berlin: Springer; 2003. https://doi.org/10.1007/978-3-642-19029-2.
26. Harel D, Katz G, Marron A, Weiss G. Non-intrusive repair of reactive programs. In: Proc. 17th IEEE int. conf. on engineering of complex computer systems (ICECCS); 2012. p. 3–12.
27. Harel D, Kugler H, Marelly R, Pnueli A. Smart play-out of behavioral requirements. In: Proc. 4th int. conf. on formal methods in computer-aided design (FMCAD); 2002. p. 378–98.
28. Elyasaf A, Weinstock M, Weiss G. Chapter 1. Interweaving AI and Behavioral Programming Towards Better Programming Environments, pp. 3–27. https://doi.org/10.1142/9789811239922_0001
29. Harel D, Katz G, Marron A, Sadon A, Weiss G. Executing scenario-based specification with dynamic generation of rich events. Commun Comput Inf Sci (CCIS) 2020;1161.
30. Katz G, Marron A, Sadon A, Weiss G. On-the-fly construction of composite events in scenario-based modeling using constraint solvers. In: Proc. 7th int. conf. on model-driven engineering and software development (MODELSWARD); 2019. p. 143–56.
31. Harel D, Kantor A, Katz G, Marron A, Mizrahi L, Weiss G. On composing and proving the correctness of reactive behavior. In: Proc. 13th int. conf. on embedded software (EMSOFT); 2013. p. 1–10.
32. Harel D, Katz G, Marron A, Weiss G. The effect of concurrent programming idioms on verification. In: Proc. 3rd int. conf. on model-driven engineering and software development (MODELSWARD); 2015. p. 363–9.
33. Katz G. On module-based abstraction and repair of behavioral programs. In: Proc. 19th int. conf. on logic for programming, artificial intelligence and reasoning (LPAR); 2013. p. 518–35.
34. Harel D, Katz G, Lampert R, Marron A, Weiss G. On the succinctness of idioms for concurrent programming. In: Proc. 26th int. conf. on concurrency theory (CONCUR); 2015. p. 85–99.
35. Harel D, Kantor A, Katz G, Marron A, Weiss G, Wiener G. Towards behavioral programming in distributed architectures. J Sci Comput Programm (J SCP). 2015;98:233–67.
36. Steinberg S, Greenyer J, Gritzner D, Harel D, Katz G, Marron A. Efficient distributed execution of multi-component scenario-based models. Commun Comput Inf Sci (CCIS). 2018;880:449–83.
37. Steinberg S, Greenyer J, Gritzner D, Harel D, Katz G, Marron A. Distributing scenario-based models: a replicate-and-project approach. In: Proc. 5th int. conf. on model-driven engineering and software development (MODELSWARD); 2017. p. 182–95.
38. Greenyer J, Gritzner D, Katz G, Marron A, Glade N, Gutjahr T, König F. Distributed execution of scenario-based specifications of structurally dynamic cyber-physical systems. In: Proc. 3rd int. conf. on system-integrated intelligence: new challenges for product and production engineering (SYSINT); 2016. p. 552–9.
39. Harel D, Kantor A, Katz G. Relaxing synchronization constraints in behavioral programs. In: Proc. 19th int. conf. on logic for programming, artificial intelligence and reasoning (LPAR); 2013. p. 355–72.
40. Harel D, Katz G, Marron A, Weiss G. Non-intrusive repair of safety and liveness violations in reactive programs. Trans Comput Collect Intell (TCCI). 2014;16:1–33.
41. Katz G. Towards repairing scenario-based models with rich events. In: Proc. 9th int. conf. on model-driven engineering and software development (MODELSWARD); 2021. p. 362–72.
42. Harel D, Katz G, Marelly R, Marron A. Wise computing: toward endowing system development with proactive wisdom. IEEE Comput. 2018;51(2):14–26.
43. Marron A, Arnon B, Elyasaf A, Gordon M, Katz G, Lapid H, Marelly R, Sherman D, Szekely S, Weiss G, Harel D. Six (im) possible things before breakfast: building-blocks and design-principles for wise computing. In: Proc. 19th ACM/IEEE int. conf. on model driven engineering languages and systems (MODELS); 2016. p. 94–100.
44. Harel D, Katz G, Marelly R, Marron A. An initial wise development environment for behavioral models. In: Proc. 4th int. conf. on model-driven engineering and software development (MODELSWARD); 2016. p. 600–12.
45. Harel D, Katz G, Marelly R, Marron A. First steps towards a wise development environment for behavioral models. Int J Inform Syst Model Des (IJISMD). 2016;7(3):1–22.
46. Gordon M, Marron A, Meerbaum-Salant O. Spaghetti for the main course? Observations on the naturalness of scenario-based programming. In: Proc. 17th ACM annual conf. on innovation and technology in computer science education (ITCSE); 2012. p. 198–203.
47. Alexandron G, Armoni M, Gordon M, Harel D. Scenario-based programming: reducing the cognitive load, fostering abstract thinking. In: Proc 36th int. conf. on software engineering (ICSE); 2014. p. 311–20.
48. Katz G. Guarded deep learning using scenario-based modeling. In: Proc. 8th int. conf. on model-driven engineering and software development (MODELSWARD); 2020. p. 126–36.
49. Katz G, Elyasaf A. Towards combining deep learning, verification, and scenario-based programming. In: Proc. 1st workshop on verification of autonomous and robotic systems (VARS); 2021. p. 1–3.
50. Ng A, Harada D, Russell S. Policy invariance under reward transformations: theory and application to reward shaping. In: Proc. 16th int. conf. on machine learning (ICML); 1999. p. 278–87.

51. Zou H, Ren T, Yan D, Su H, Zhu J. Reward shaping via meta-learning. Technical Report. 2019. Preprint at arXiv:1901.09330
52. Yaacov T. BPPy: behavioral programming in Python. 2020. https://github.com/bThink-BGU/BPPy
53. Harel D, Marron A, Weiss G. Programming coordinated scenarios in Java. In: Proc. 24th European conf. on object-oriented programming (ECOOP); 2010. p. 250–74.
54. Shalev-Shwartz S, Shammah S, Shashua A. On a formal model of safe and scalable self-driving cars. Technical Report. 2017. Preprint at arXiv:1708.06374
55. Kang C, Kim G, Yoo S-I. Detection and recognition of text embedded in online images via neural context models. In: Proc. 31st AAAI conf. on artificial intelligence (AAAI); 2017.
56. Milan A, Rezatofighi H, Dick A, Reid I, Schindler K. Online multi-target tracking using recurrent neural networks. In: Proc. 31st AAAI conf. on artificial intelligence (AAAI); 2017.
57. Ray P, Chakrabarti A. A mixed approach of deep learning method and rule-based method to improve aspect level sentiment analysis. Appl Comput Inform. 2020.
58. Katz G. Augmenting deep neural networks with scenario-based guard rules. Commun Comput Inf Sci (CCIS). 2021;1361:147–72.
59. Elyasaf A, Sadon A, Weiss G, Yaacov T. Using behavioural programming with solver, context, and deep reinforcement learning for playing a simplified RoboCup-Type game. In: Proc. 22nd ACM/IEEE int. conf. on model driven engineering languages and systems companion (MODELS-C); 2019. p. 243–51.