MoGym: Using Formal Models for Training and Verifying Decision-making Agents[⋆]

Timo P. Gros[1], Holger Hermanns[1,2], Jörg Hoffmann[1], Michaela Klauck[1], Maximilian A. Köhl[1], and Verena Wolf [1]

[1]Saarland University, Saarland Informatics Campus, Saarbrücken, Germany
[2]Institute of Intelligent Software, Guangzhou, China
`{timopgros,hermanns,hoffmann,klauck,koehl,wolf}@cs.uni-saarland.de`

Abstract. MoGym, is an integrated toolbox enabling the training and verification of machine-learned decision-making agents based on formal models, for the purpose of sound use in the real world. Given a formal representation of a decision-making problem in the JANI format and a reach-avoid objective, MoGym (a) enables training a decision-making agent with respect to that objective directly on the model using reinforcement learning (RL) techniques, and (b) it supports rigorous assessment of the quality of the induced decision-making agent by means of deep statistical model checking (DSMC). MoGym implements the standard interface for training environments established by OpenAI Gym, thereby connecting to the vast body of existing work in the RL community. In return, it makes accessible the large set of existing JANI model checking benchmarks to machine learning research. It thereby contributes an efficient feedback mechanism for improving in particular reinforcement learning algorithms. The connective part is implemented on top of Momba. For the DSMC quality assurance of the learned decision-making agents, a variant of the statistical model checker modes of the Modest Toolset is leveraged, which has been extended by two new resolution strategies for non-determinism when encountered during statistical evaluation.

Keywords: Formal Methods · Statistical Model Checking · Reinforcement Learning

1 Introduction

Making optimal decisions in an uncertain environment is the crux of many practical problems. Reinforcement Learning (RL) is a popular method to compute near-optimal policies for sequential decision-making problems [60]. In the last years, RL algorithms that approximate optimal decision policies by training deep

[⋆] Authors are listed alphabetically. This work was partially supported by the German Research Foundation (DFG) under grant No. 389792660 as part of TRR 248, by the European Regional Development Fund (ERDF), and by the Key-Area Research and Development Program Grant 2018B010107004 of Guangdong Province.

neural networks have exhibited unprecedented performance in various tasks [47]. However, the expressivity of these models makes them difficult to interpret or to be checked for consistency for some desired properties. This is an impediment to the use of such representations in safety-critical applications [61]. In addition, the environment of the decision-making agent executing the policy during training is typically specified implicitly in the form of simulation code. In the academic context, for instance the Arcade Learning Environment is widely used, which provides game simulators for different ATARI 2006 benchmarks [6].

If one strives for a principled understanding of the power of RL algorithms or of the properties of a specific learned agent in the (possibly uncertain) environment, a formal, mathematically precise and unambiguous description of the *training environment* appears central. The formal methods community has developed appropriate language concepts for the description of such environment models. Their advantage lies in their succinctness and modularity as well as their underlying mathematically rigorous formal semantics based on stochastic process models such as Markov Decision Processes (MDPs) [53], the main semantic object of probabilistic model checking [40]. A widespread format to describe MDP models of environments is the JANI format [14], providing a modular, automata-like syntax, supported by several model checkers, like Storm, the Modest Toolset, ePMC [33,30,29], and via a translation also by PRISM [41].

This paper presents MoGym, a toolbox that bridges the gap between formal methods and RL by enabling (a) formally specified training environments to be used with machine-learned decision-making agents, and (b) the rigorous assessment of the quality of learned agents. For (a), it implements and extends the OpenAI Gym API [11], which is the widely used standard interface for deep reinforcement learning [55,26,16,35,50]. MoGym is based on Momba [39], a Python toolbox for dealing with quantitative models from construction to analysis centered around JANI. MoGym can process JANI models for the description of a training environment and, based on the induced formal MDP semantics, makes it possible to train agents using popular RL algorithms.

For (b), the environment format itself is accessible to state-of-the-art model checkers. This enables probabilistic model checking of a specific agent acting in the environment specified by the model. This can be crucial to determining if further training improves the agent's quality and, whenever synthesis of the optimal agent is feasible, it allows a comparison of the agent's behavior to the optimal one. As such, the environment provides a stable and fully controllable training and checking context to assert the safety risk induced by an agent during and after training. More concrete, MoGym leverages deep statistical model checking (DSMC) [20,21]. As shown in these works on DSMC, the quality assessment of an agent during training is not trivial and can especially not always be derived from the observed training returns. Hence, analyzing the quality of the decision-making agents after training clearly is of interest [20,21], especially for badly interpretable agent structures such as neural networks (NN). In DSMC this is done by using the decision-making agent as an oracle resolving the non-determinism in the MDP specifying the environment. When resolving

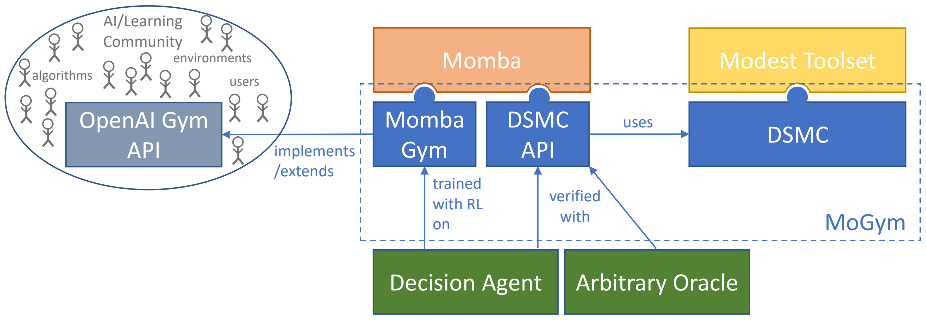


Fig. 1: The architecture of MoGym and its components

the non-determinism, a Markov chain results on which the probability of satisfying a given reach-avoid objective can be calculated. A prominent technique for doing so with very low memory requirements is statistical model checking (SMC) [5,34,44,64,67,7,32]. The satisfaction probability for the reach-avoid objective calculated using statistics based on a set of simulation runs of the resulting Markov chain, can serve as an indicator of the quality of the decision-making agent for solving the reach-avoid task it was originally trained on.
MoGym comprises the following components:

- *Momba Gym*, newly implemented on top of Momba [39]. It implements and extends the OpenAI Gym API [11] for deep reinforcement learning. Momba Gym can be used to load a specified formal model together with a reach-avoid objective given by a JANI file [14] and then train a decision-making agent on it, which interacts in the environment given by the formal model.
- The *DSMC API*, also newly implemented on top of Momba. It includes a Python API to use the DSMC functionality [20,21] of the Modest Toolset [30,13].
- *DSMC* implemented in the Modest Toolset. In prior work [20,21], we implemented Deep Statistical Model Checking for specific networks and purposes, only. With this work, we extend the statistical model checker modes [13] of the Modest Toolset to be able to handle any formal MDP model given in one of the input languages of the toolset, and any neural network of arbitrary structure, as well as arbitrary oracles connected via a function. With the DSMC functionality it is possible to statistically model check the probability with which formal properties, i. e., reach-avoid objectives, are fulfilled by the decision-making agent, respectively oracle.

Figure 1 shows how the different parts of MoGym are interconnected. First, a decision-making agent can be trained on a formal model and a reach-avoid property, defined in a JANI model, against the OpenAI Gym API by using Momba Gym with different reinforcement learning techniques, which can be implemented and defined by the user. Afterwards, the trained agent can be verified w.r.t. reach-avoid objectives by invoking the DSMC API, which makes use of the DSMC extension of the statistical model checker modes. Alternatively, the training step can be skipped, or can be done in any other way, and an arbitrary external oracle can be checked.

We are not aware of any other work that enables a direct connection of formal verification models and reinforcement learning that directly allows the analysis of different RL agents for a variety of verification benchmarks.

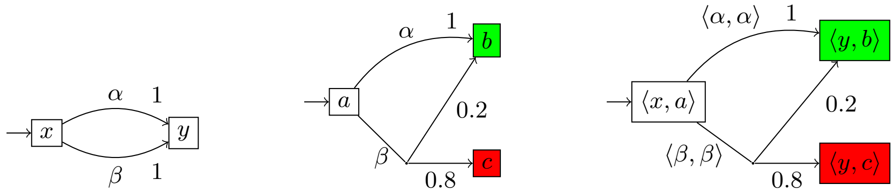
Outline of the paper. In Sect. 2 we describe the Momba Gym Python API and explain how MoGym is used to train agents on existing JANI MDPs. Sect. 3 presents the DSMC API of Momba, and discusses its use to assess the quality of decision-making agents or arbitrary oracles via DSMC, together with the new DSMC functionality of modes. In Sect. 4 we provide empirical insight into the full functionality of MoGym. Sect. 5 concludes the paper.

A preview of the Jupyter Notebook demonstrating the code we used to execute the experiments shown in the paper can be found online. It will later be part of the full artifact for the tool paper.

2 Formal Models as Training Environments

At the heart of MoGym is an implementation of the OpenAI Gym API in *Momba Gym*, which now enables the usage of JANI models as training environments. OpenAI Gym [11] constitutes *the* standard API for interfacing environments with different reinforcement learning algorithms enabling their comparison and fostering development of new techniques. It is widely used by both, algorithms that interact with the interface [55,26,16,35,50], as well as various benchmarks that implement (and sometimes extend) the interface [63,66,62,3,18,15]. With Momba Gym, MoGym provides an extension of this API for general JANI MDP models equipped with reach-avoid properties. JANI is a JSON-based format for exchanging formal models between tools [14]. It is the standard format in the quantitative verification community and directly supported by state-of-the-art tools, like Storm [33], the Modest Toolset [30], and ePMC [29]. Translations from and to other languages such as the PRISM language [41,42], Modest [28] and even the planning language PPDDL [37,36] exist.

A JANI model is a network of interacting automata with variables. Each automaton consists of a set of *locations* and a set of probabilistic *edges* from a *source* location to possibly multiple *destination* locations. Edges can be labeled with *edge labels* and annotated, depending on the destination, with assignments to variables. The *transitions* of the network are then obtained by synchronizing the automata, i. e., in every transition, potentially multiple automata participate with one edge, respectively. For our purposes, we assume that a decision-making agent controls a single automaton in the network, i. e., resolves the non-determinism of this automaton. Fig. 2 exemplifies the construction of an automata network from two automata: a controlled automaton (a) and a non-controlled automaton (b). Depending on which of the edges of automaton (b) is taken, the probability of ending up in state b is either 1.0 (action α) or 0.2 (action β). The final composition (c) is then the product of both automata synchronizing over the shared edge labels α and β. Controlling automaton (a) here



(a) Controlled automaton.

(b) A non-controlled automaton.

(c) Composition of both automata.

Fig. 2: Networks of interacting automata.

implies selecting which of the transitions in the final compositional does happen. By choosing the edge labeled with α, the transition $\langle \alpha, \alpha \rangle$ in the composition is selected and analogously for β. The choice of α in the controlled automaton obviously is the one maximizing the probability of reaching the green state $\langle y, b \rangle$ in the composition (c). In fact, the state is reached with certainty. Technically, this approach would extend to a multi-agent setting where different agents resolve the non-determinism in different parts of the model. We plan to provide a multi-agent setting in future work and assume here that all non-determinism not resolved by the controlled automaton is resolved uniformly.[1]

For training an agent in an environment, the OpenAI Gym API requires the definition of an *action space* and an *observation space*. In response to receiving observations from the observation space, the trained agent makes a decision from the action space. To enable the usage of general JANI MDP models as environments, an action space and observation space have to be extracted from the model. Depending on the model, there are multiple ways to do so. Momba Gym implements different strategies for this extraction. For the action space, edges of the controlled automaton can be selected by index or by label. For the observation space, (i) only global variables, (ii) global variables and local variables of the controlled automaton, or (iii) all variables can be declared as observable.[2] Other strategies can easily be added to Momba Gym.

Whenever the agent makes a decision in response to an observation, the decision is mapped to an edge of the controlled automaton and then to a transition of the network. If present, other non-deterministic influences are resolved uniformly at random, as mentioned above. In this case, the user receives a warning message so that this is taken into account when inspecting the results. After taking the respective transition, the environment continues the trace through the model until a state is reached where the agent can make a decision again.

Momba Gym supports reach-avoid properties of the form $\phi \,\mathbf{U}\, \psi$ where ϕ and ψ are propositional logic formulas over the model's states. $\phi \,\mathbf{U}\, \psi$ encodes the property that a state satisfying ψ is reached eventually and that ϕ holds on all states prior to reaching ψ. In a *bad state*, which should be avoided, ψ is not

[1] That is, each of the remaining non-deterministic options is considered equiprobable. MoGym can easily be extended with other mechanisms to resolve non-determinism.
[2] For more details about those strategies see `https://momba.dev/gym/`.

satisfied and (i) there are no remaining transitions or (ii) ϕ is violated. In a *goal state* ψ is satisfied. To apply RL techniques, Momba Gym supports providing a *reward structure* specifying the reward for reaching a goal, the (usually negative) reward for reaching a bad state, the reward for taking a decision neither leading to a goal nor to a bad state (usually zero), and the reward for taking a non-applicable decision. Using the Momba Gym API integrated in Momba, one can create a training environment from an arbitrary JANI MDP model as follows:

```python
from momba import jani, gym
model = jani.load_model(JANI_SOURCE)
# ...
env = gym.create_generic_env(model, automaton)
```

In this command, `automaton` is the automaton the agent controls. The function `create_generic_env` takes additional optional parameters specifying the strategy for the extraction of the action and observation space (i. e., by index or by label, see above) as well as the reward structure (by defining the four reward values indicated above) and parameters of the JANI model. The resulting `env` implements the OpenAI Gym API such that it can be directly used to train an agent for the given property using arbitrary RL algorithms based on the OpenAI Gym API. Thereby, Momba Gym makes JANI MDP models accessible to the RL community to train and evaluate their algorithms on. The implementation of the Momba Gym environment uses the explicit state space exploration engine of Momba which is written in Rust. It is sufficiently performant such that it can be used to train different agents using state-of-the-art RL algorithms.

Momba Gym extends the OpenAI Gym API with the ability to *fork* the environment and query the applicable actions. The former is useful for algorithms based on *Monte-Carlo Tree Search (MTS)* [12], known to act favorably on prominent benchmarks, like Atari Games [24]. Further, MTS forms the basis of DeepMind's famous algorithms around AlphaGo and AlphaZero [57].

In addition to the general Momba Gym API, we provide exemplary code to train an agent for an arbitrary formal model. While we ourselves implemented *deep Q-learning* [47], MoGym is open to any (deep) reinforcement learning algorithm. Using our implementation of deep Q-learning, enables training of a decision-making agent for an arbitrary JANI MDP model.[3] We note however that deep RL is known to be hyperparameter sensitive [45], so intensive tweaking of hyperparameters might be needed for the learning to work. In this regard our deep Q-learning implementation is no exception.

3 Verifying Agents Using Statistical Model Checking

If given a formal model and a decision-making agent trained on it, MoGym supports verification by deep statistical model checking. To this end, the DSMC API of MoGym implements two functions, one for verifying arbitrary agents in the form of Python functions and one for verifying PyTorch neural networks.

[3] Details will be included in the artifact of the paper.

Both functions rely on our DSMC extension of the statistical model checker modes [13] of the Modest Toolset [30], which accepts both forms of decision entities, and returns the reach-avoid probability calculated by the model checker.

Statistical model checking is based on Monte-Carlo simulation [56,65]. Using statistics, a probability estimate is derived from a set of simulation runs, regarding the satisfaction of a reach-avoid property, the error of which is bounded by a confidence interval. This is determined by the probability of the error in the computation being larger than ϵ is smaller than δ: $P(\mathit{error} > \epsilon) < \delta$. For SMC to be applicable, the non-determinism of the model needs to be resolved [8,13]. In our DSMC setting this is done by the agent and otherwise resolved uniformly, i. e., equiprobable across all options (see Sect. 2). The computed reach-avoid probability can serve as an indicator of the overall quality of the decisions made by the agent [20]. The DSMC implementation in modes provides the same functionality regarding the observation space (global and/or local variables) and action space (select by index or label) as the Momba Gym training infrastructure described in Sect. 2.

As mentioned above, modes can deal with two variants of decision-making agents. An arbitrary Python function mapping observations to decisions can be checked with the DSMC API of MoGym by executing:

```
gym.checker.check_oracle(oracle, model, automaton)
```

Here, `oracle` is the Python function implementing the decision-making agent. Notably, this is not limited to trained agents in any way. Any arbitrary Python function with an appropriate signature can be used. The other parameters are analogous to `create_generic_env`. In particular, `check_oracle` also allows optionally specifying a strategy for extracting the action and observation spaces (see above).

While `check_oracle` involves executing Python code, a more efficient approach is available when the decision-making agent is a PyTorch neural network. In this case, the network can directly be verified with `check_nn`:

```
gym.checker.check_nn(nn, model, automaton)
```

To this end, we assume that the network is a sequence of layers. The function `check_nn` extracts these layers from the provided neural network `nn` and exports them in a JSON-based format. The neural network is then loaded by modes and used for model checking without calling back into the Python runtime. With the help of TorchSharp [25] (a .NET library providing access to the library that powers PyTorch) our extension of modes supports networks with arbitrary dimensions and activation functions.

Alternatively to the DSMC API provided by MoGym, it is also possible to invoke modes on the command line to check a NN or to connect it to an arbitrary decision-making agent via a socket connection. The agent could be any program taking the information of the observation space as input and sending an action decision back.

4 Experimental Insights

With MoGym it is now possible to train agents and assess their quality for arbitrary JANI MDP models by evaluating them using the DSMC extension of the statistical model checker modes. In the following, we demonstrate all parts of the workflow when MoGym is used from training to evaluation. For our case studies, the training was performed by using a well-established standard RL algorithm, the deep Q-learning algorithm [47].

Benchmarks. Working with MoGym starts with devising a formal model to train a decision-making agent on. For example, the *Quantitative Verification Benchmark Set* (QVBS) [31] contains JANI models originally collected for competitions among quantitative verification tools. With the help of MoGym they are now accessible for use in the learning community. For our case studies, we selected three MDP benchmarks from the QVBS: *cdrive.2*, *elevators* and *firewire_dl*. With respect to the observation spaces, we use the Momba Gym API default setting, in which only global variables are observable.

In *cdrive.2* a car drives in a city modeled using locations connected by roads with traffic lights. The car should reach a destination without an accident [10]. In the *elevators* case, a certain number of elevators is available to transport coins to a predefined level. An elevator can fall down on a lower level [38,10]. The *firewire_dl* benchmark models the leader election protocol in the Tree Identify Protocol of the IEEE 1394 High Performance Serial Bus [59,43].

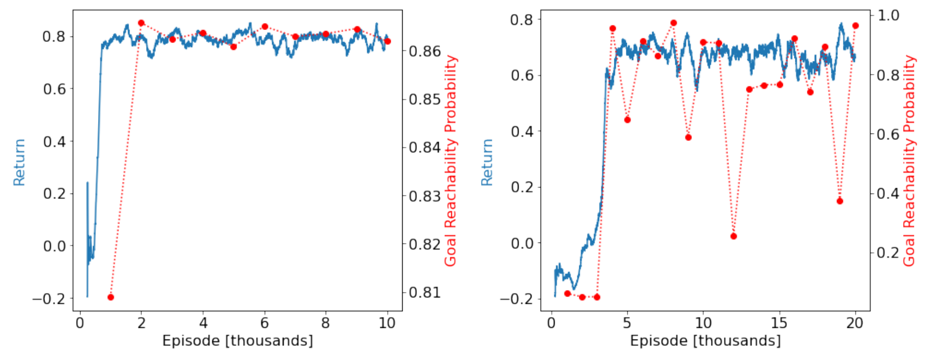
Another popular benchmark is Racetrack, which has been adopted for decision making under uncertainty in many works [2,19,4,9,46,51,52]. In Racetrack, a vehicle needs to be driven on a discretized grid track towards a goal as fast as possible without crashing. A preview of the Jupyter Notebook showing the code we used for the experiments, which will later be part of the tool paper's artifact, is available online.

Training. We trained agents for all of the considered benchmarks by using the calls to the Momba Gym API as introduced in Sect. 2, which can be inspected in Sect. 2.1 and 2.2 of the Jupyter notebook.

Fig. 3 (a) and (b) shows the training progress of *cdrive.2* and *Racetrack*, respectively, depicted in blue. The training for *cdrive.2* took around 1 *min*, and for *Racetrack* about 22 *min*, on a standard laptop. In contrast to these two benchmarks, learning for *elevators* and *firewire_dl* failed. During training, the agent was able to reach the goal, but the NN was not able to generalize.

Verification. For *cdrive.2* and *Racetrack*, the training return increases over the number of training episodes and is quite stable at the end. The training return is commonly regarded as an estimator of the training progress [48,47]. Here it appears to indicate that the quality of the trained neural networks does neither increase nor decrease from a certain episode on.

However, we now can use DSMC to check the actual quality of the trained agents, i. e., we can determine how high the probability is that they indeed



(a) (b)

Fig. 3: Blue: Training curve showing sliding mean of the training return, i. e., the accumulated discounted reward over the last 500 training episodes, on the left y-axis. Note the different scale for (a) and (b). Red: Goal reachability probability on the right y-axis. Both are plotted over the number of training episodes on the x-axis. (a) Shows results for *cdrive.2* and (b) for *Racetrack*.

reach the goal in their respective environments defined by the MDP model. We do so by making use of the DSMC API of MoGym, introduced in Sect. 3 using modes as backend. We check the goal reachability probability of the NN policies extracted every 1000 training episodes as shown in Sect. 2.1 and 2.2 of the Jupyter notebook.

As depicted by Fig. 3, the return during training is not as expressive as expected. While the training return is relatively consistent for both *cdrive.2* and *Racetrack*, the goal reachability probability (depicted in the red points) over training is not. In contrast, it both increases and decreases over the training episodes. So, the training return alone turns out not to be a good indicator for deciding which of the extracted policies actually is the best one. For *cdrive.2* (Fig. 3 (a)), this can be considered as fine tuning, as most of the policies perform near-optimal. In contrast, for *Racetrack* (Fig. 3 (b)), we observe a huge difference between the policies, including near-optimal policies as well as policies with a goal reachability probability of only about 20%. These deeper insights regarding the neural networks' quality are only possible by using DSMC.

Having selected the best policy for each benchmark, the analysis yields a goal reachability probability of 86.57% for *cdrive.2*, where a policy acting optimally would reach the goal with a probability of 86.45%.[4] The optimal value has been calculated with the exhaustiv probabilistic model checking engine mcsta [27] of the Modest Toolset. The goal reachability probability of the best NN policy

[4] Note that the goal reachability probability of the NN policies is estimated by statistical model checking. Thus, even though it might seem surprising at first sight, it is of course possible that the analysis of our policy yields a slightly higher goal reachability probability than optimally possible as long as this is within the given confidence interval. We use $P(error > \epsilon) < \delta$, where $\epsilon = 0.01$ and $\delta = 0.05$, i. e., a confidence of 95%.

of the trained agent for *Racetrack* is 97.30% where the optimal policy reaches the goal with a probability of 99.99%.

5 Conclusion and Future Work

We presented MoGym, an integrated toolbox to train, analyze and verify decision-making agents on formal models. These formal models are made available through Momba Gym, which implements and extends the well-established OpenAI Gym API for arbitrary reinforcement learning techniques. Using these techniques to obtain NNs or, alternatively, some general decision-making agents, they can then be rigorously verified with DSMC using the new extension of modes. The approach is open to all JANI MDPs and modes can in principle handle arbitrary fully connected and even convolutional networks.

On the basis of the QVBS and *Racetrack*, we showed how the toolchain of MoGym works. As presented, our formal-model-based approach enables deeper insights for specified properties than non-formal, implicitly defined simulation-based environments.

In the future, we want to address the problem which caused the training for *elevators* and *firewire_dl* to fail. Given the successes of deep RL across many diverse environments [58,57,54,23,49,1,47], one is tempted to expect it to work well on the considered environments [31,22], too. Still, deep reinforcement learning is known to perform badly in domains with large action spaces [17], and we suspect this to be the root of the problem we observe. The action structures arising in networks of automata are of a specific kind. Rooted in process algebra, their main role is to enable and orchestrate synchronization across automata, and this is indeed the case for the JANI models *elevators* and *firewire_dl*. A more meaningful construction of an action space of compositional models suitable for learning appears needed.

Furthermore, the extension of our tool to other model types and an extension to control all of the modeled automata, making the learning task a multi-agent one, would clearly be of interest. Apart from that, we plan to build upon MoGym to develop DSMC techniques further. With DSMC Evaluation Stages [21] it has already been shown that DSMC can be applied during deep RL to determine state space regions with weak performance to concentrate on them during the learning process. With the help of MoGym this technique can now be done much more integrated and there is room for further implementations into this direction in our tool chain.

References

1. Agostinelli, F., McAleer, S., Shmakov, A., Baldi, P.: Solving the Rubiks Cube with Deep Reinforcement Learning and Search. Nature M. Intel. pp. 356–363 (2019)
2. Baier, C., Christakis, M., Gros, T.P., Groß, D., Gumhold, S., Hermanns, H., Hoffmann, J., Klauck, M.: Lab conditions for research on explainable automated decisions. In: TAILOR 2020. pp. 83–90 (2020)

3. Bard, N., et al.: The hanabi challenge: A new frontier for ai research. Artificial Intelligence **280**, 103216 (2020)
4. Barto, A.G., Bradtke, S.J., Singh, S.P.: Learning to act using real-time dynamic programming. Artificial Intelligence **72**(1), 81 – 138 (1995)
5. Basu, A., Bensalem, S., Bozga, M., Caillaud, B., Delahaye, B., Legay, A.: Statistical Abstraction and Model-Checking of Large Heterogeneous Systems. In: FORTE 2010. vol. 6117, pp. 32–46. Springer (2010)
6. Bellemare, M.G., Naddaf, Y., Veness, J., Bowling, M.: The arcade learning environment: An evaluation platform for general agents. JAIR **47**, 253–279 (2013)
7. Bogdoll, J., Fioriti, L.M.F., Hartmanns, A., Hermanns, H.: Partial order methods for statistical model checking and simulation. In: FORTE 2011. vol. 6722, pp. 59–74. Springer (2011)
8. Bogdoll, J., Hartmanns, A., Hermanns, H.: Simulation and Statistical Model Checking for Modestly Nondeterministic Models. In: GI/ITG Conf. Measurement, Modelling, and Eval. Comp. Sys. Depend. Fault Tol. pp. 249–252. Springer (2012)
9. Bonet, B., Geffner, H.: Labeled RTDP: improving the convergence of real-time dynamic programming. In: ICAPS. pp. 12–21 (2003)
10. Bonet, B., Givan, B.: Non-Deterministic Planning Track of the 2006 IPC. http://idm-lab.org/wiki/icaps/ipc2006/probabilistic/ (2006), acc. Oct., 13, 2021
11. Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., Zaremba, W.: Openai gym. CoRR **abs/1606.01540** (2016)
12. Browne, C.B., et al.: A survey of monte carlo tree search methods. IEEE Trans. Comp. Intel. and AI in Games **4**(1), 1–43 (2012)
13. Budde, C.E., D'Argenio, P.R., Hartmanns, A., Sedwards, S.: A statistical model checker for nondeterminism and rare events. In: TACAS. pp. 340–358 (2018)
14. Budde, C.E., Dehnert, C., Hahn, E.M., Hartmanns, A., Junges, S., Turrini, A.: JANI: Quantitative model and tool interaction. In: TACAS. pp. 151–168 (2017)
15. Côté, M.A., et al.: Textworld: A learning environment for text-based games. In: Workshop on Computer Games. pp. 41–75. Springer (2018)
16. Doshi-Velez, F., Kim, B.: Towards a rigorous science of interpretable machine learning. arXiv preprint arXiv:1702.08608 (2017)
17. Dulac-Arnold, G., et al.: Deep reinforcement learning in large discrete action spaces. arXiv preprint arXiv:1512.07679 (2015)
18. Fan, L., Zhu, Y., Zhu, J., Liu, Z., Zeng, O., Gupta, A., Creus-Costa, J., Savarese, S., Fei-Fei, L.: Surreal: Open-source reinforcement learning framework and robot manipulation benchmark. In: Conf. Robot Learning. pp. 767–782. PMLR (2018)
19. Gros, T.P., Groß, D., Gumhold, S., Hoffmann, J., Klauck, M., Steinmetz, M.: TraceVis: Towards Visualization for Deep Statistical Model Checking. In: Int. Symp. Leveraging Applications of Formal Methods, Verification and Validation (2020)
20. Gros, T.P., Hermanns, H., Hoffmann, J., Klauck, M., Steinmetz, M.: Deep statistical model checking. In: FORTE 2020. pp. 96–114 (2020)
21. Gros, T.P., Höller, D., Hoffmann, J., Klauck, M., Meerkamp, H., Wolf, V.: DSMC evaluation stages: Fostering robust and safe behavior in deep reinforcement learning. In: QEST. pp. 197–216 (2021)
22. Gros, T.P., Höller, D., Hoffmann, J., Wolf, V.: Tracking the race between deep reinforcement learning and imitation learning. In: QEST 2020. vol. 12289, pp. 11–17. Springer (2020)
23. Gu, S., Holly, E., Lillicrap, T., Levine, S.: Deep Reinforcement Learning for Robotic Manipulation with Asynchronous Off-policy Updates. In: 2017 IEEE Int. Conf. robotics and automation (ICRA). pp. 3389–3396. IEEE (2017)

24. Guo, X., Singh, S., Lee, H., Lewis, R.L., Wang, X.: Deep learning for real-time atari game play using offline monte-carlo tree search planning. In: Advances in neural information processing systems. pp. 3338–3346 (2014)
25. Gustafsson, N., et al.: TorchSharp. https://github.com/dotnet/TorchSharp (2021), accessed on Sept., 22, 2021
26. Haarnoja, T., Zhou, A., Abbeel, P., Levine, S.: Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In: Int. conf. ML. pp. 1861–1870. PMLR (2018)
27. Hahn, E.M., Hartmanns, A.: A comparison of time- and reward-bounded probabilistic model checking techniques. In: SETTA 2016. pp. 85–100 (2016)
28. Hahn, E.M., Hartmanns, A., Hermanns, H., Katoen, J.: A compositional modelling and analysis framework for stochastic hybrid systems. Formal Methods Syst. Des. **43**(2), 191–232 (2013)
29. Hahn, E.M., Li, Y., Schewe, S., Turrini, A., Zhang, L.: iscasmc: A web-based probabilistic model checker. In: FM 2014. pp. 312–317 (2014)
30. Hartmanns, A., Hermanns, H.: The Modest Toolset: An integrated environment for quantitative modelling and verification. In: TACAS 2014. pp. 593–598 (2014)
31. Hartmanns, A., Klauck, M., Parker, D., Quatmann, T., Ruijters, E.: The Quantitative Verification Benchmark Set. In: TACAS 2019. pp. 344–350 (2019)
32. Hartmanns, A., Timmer, M.: On-the-Fly Confluence Detection for Statistical Model Checking. In: NFM 2013
33. Hensel, C., Junges, S., Katoen, J.P., Quatmann, T., Volk, M.: The probabilistic model checker storm. Int. Jour. on Software Tools for Technology Transfer (2021)
34. Hérault, T., Lassaigne, R., Magniette, F., Peyronnet, S.: Approximate probabilistic model checking. In: VMCAI 2004. vol. 2937, pp. 73–84. Springer (2004)
35. Ho, J., Ermon, S.: Generative adversarial imitation learning. Advances in neural information processing systems **29**, 4565–4573 (2016)
36. Hoffmann, J., Hermanns, H., Klauck, M., Steinmetz, M., Karpas, E., Magazzeni, D.: Let's learn their language? A case for planning with automata-network languages from model checking. In: AAAI 2020. pp. 13569–13575 (2020)
37. Klauck, M., Steinmetz, M., Hoffmann, J., Hermanns, H.: Bridging the gap between probabilistic model checking and probabilistic planning: Survey, compilations, and empirical comparison. J. Artif. Intell. Res. **68**, 247–310 (2020)
38. Koehler, J., Schuster, K.: Elevator control as a planning problem. In: 5. Int. Conf. Art. Intel. Planning Sys. pp. 331–338. AAAI (2000)
39. Köhl, M.A., Klauck, M., Hermanns, H.: Momba: JANI meets python. In: TACAS. pp. 389–398 (2021)
40. Kwiatkowska, M.Z., Norman, G., Parker, D.: Stochastic model checking. In: SFM 2007, Advanced Lectures. pp. 220–270. LNCS 4486 (2007)
41. Kwiatkowska, M.Z., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: 23. CAV 2011. pp. 585–591 (2011)
42. Kwiatkowska, M.Z., Norman, G., Parker, D.: The PRISM benchmark suite. In: 9. QEST 2012. pp. 203–204 (2012)
43. Kwiatkowska, M.Z., Norman, G., Sproston, J.: Probabilistic model checking of deadline properties in the IEEE 1394 firewire root contention protocol. Formal Aspects Comput. **14**(3), 295–318 (2003)
44. Legay, A., Delahaye, B., Bensalem, S.: Statistical Model Checking: An Overview. In: Runtime Verification - 1. RV 2010. vol. 6418, pp. 122–135. Springer (2010)
45. Liessner, R., Schmitt, J., Dietermann, A., Bäker, B.: Hyperparameter optimization for deep reinforcement learning in vehicle energy management. In: ICAART (2). pp. 134–144 (2019)

46. McMahan, H.B., Gordon, G.J.: Fast exact planning in Markov decision processes. In: ICAPS. pp. 151–160 (2005)
47. Mnih, V., et al.: Human-level Control through Deep Reinforcement Learning. Nature **518**, 529–533 (2015)
48. Mnih, V., et al.: Asynchronous methods for deep reinforcement learning. In: Int. conf. machine learning. pp. 1928–1937. PMLR (2016)
49. Nazari, M., Oroojlooy, A., Snyder, L., Takac, M.: Reinforcement learning for solving the vehicle routing problem. In: Advances in Neural Inf. Proc. Sys. 31, pp. 9839–9849. Curran Associates, Inc. (2018)
50. Pathak, D., Agrawal, P., Efros, A.A., Darrell, T.: Curiosity-driven exploration by self-supervised prediction. In: Int. conf. ML. pp. 2778–2787. PMLR (2017)
51. Pineda, L.E., Lu, Y., Zilberstein, S., Goldman, C.V.: Fault-tolerant planning under uncertainty. In: IJCAI. pp. 2350–2356 (2013)
52. Pineda, L.E., Zilberstein, S.: Planning under uncertainty using reduced models: Revisiting determinization. In: ICAPS 2014 (2014)
53. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. Wiley (1994)
54. Sallab, A.E., Abdou, M., Perot, E., Yogamani, S.: Deep Reinforcement Learning Framework for Autonomous Driving. Electronic Imaging **2017**(19), 70–76 (2017)
55. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O.: Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347 (2017)
56. Sen, K., Viswanathan, M., Agha, G.: On Statistical Model Checking of Stochastic Systems. In: CAV. pp. 266–280 (2005)
57. Silver, D., et al.: Mastering the Game of Go Without Human Knowledge. Nature **550**(7676), 354–359 (2017)
58. Silver, D., et al.: A General Reinforcement Learning Algorithm That Masters Chess, Shogi, and Go Through Self-play. Science **362**(6419), 1140–1144 (2018)
59. Stoelinga, M., Vaandrager, F.W.: Root contention in IEEE 1394. In: 5. AMAST Workshop, ARTS'99. vol. 1601, pp. 53–74. Springer (1999)
60. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. Adaptive computation and machine learning, The MIT Press, second edn. (2018)
61. Verma, A., Murali, V., Singh, R., Kohli, P., Chaudhuri, S.: Programmatically interpretable reinforcement learning. In: Int. Conf. on ML. PMLR (2018)
62. Waschneck, B., Reichstaller, A., Belzner, L., Altenmüller, T., Bauernhansl, T., Knapp, A., Kyek, A.: Optimization of global production scheduling with deep reinforcement learning. Procedia Cirp **72**, 1264–1269 (2018)
63. Xia, F., Zamir, A.R., He, Z., Sax, A., Malik, J., Savarese, S.: Gibson env: Real-world perception for embodied agents. In: IEEE Conf. Computer Vision and Pattern Recognition. pp. 9068–9079 (2018)
64. Younes, H.L.S., Simmons, R.G.: Probabilistic verification of discrete event systems using acceptance sampling. In: CAV 2002. vol. 2404, pp. 223–235. Springer (2002)
65. Younes, H.L., Kwiatkowska, M., Norman, G., Parker, D.: Numerical vs. Statistical Probabilistic Model Checking: An Empirical Study. In: TACAS. pp. 46–60. Springer (2004)
66. Yu, T., Quillen, D., He, Z., Julian, R., Hausman, K., Finn, C., Levine, S.: Meta-world: A benchmark and evaluation for multi-task and meta reinforcement learning. In: Conf. Robot Learning. pp. 1094–1100. PMLR (2020)
67. Zuliani, P., Platzer, A., Clarke, E.M.: Bayesian statistical model checking with application to Stateflow/Simulink verification. FM Sys. Des. **43**(2), 338–367 (2013)