



Formal Specification and Testing for Reinforcement Learning

MAHSA VARSHOSAZ, IT University of Copenhagen, Denmark
 MOHSEN GHAFARI, IT University of Copenhagen, Denmark
 EINAR BROCH JOHNSEN, University of Oslo, Norway
 ANDRZEJ WĄSOWSKI, IT University of Copenhagen, Denmark

The development process for reinforcement learning applications is still exploratory rather than systematic. This exploratory nature reduces reuse of specifications between applications and increases the chances of introducing programming errors. This paper takes a step towards systematizing the development of reinforcement learning applications. We introduce a formal specification of reinforcement learning problems and algorithms, with a particular focus on temporal difference methods and their definitions in backup diagrams. We further develop a test harness for a large class of reinforcement learning applications based on temporal difference learning, including SARSA and Q-learning. The entire development is rooted in functional programming methods; starting with pure specifications and denotational semantics, ending with property-based testing and using compositional interpreters for a domain-specific term language as a test oracle for concrete implementations. We demonstrate the usefulness of this testing method on a number of examples, and evaluate with mutation testing. We show that our test suite is effective in killing mutants (90% mutants killed for 75% of subject agents). More importantly, almost half of all mutants are killed by generic write-once-use-everywhere tests that apply to *any* reinforcement learning problem modeled using our library, without any additional effort from the programmer.

CCS Concepts: • **Theory of computation** → **Program specifications**; • **Software and its engineering** → **Software testing and debugging**.

Additional Key Words and Phrases: specification-based testing, reinforcement learning, Scala

ACM Reference Format:

Mahsa Varshosaz, Mohsen Ghaffari, Einar Broch Johnsen, and Andrzej Wąsowski. 2023. Formal Specification and Testing for Reinforcement Learning. *Proc. ACM Program. Lang.* 7, ICFP, Article 193 (August 2023), 34 pages. <https://doi.org/10.1145/3607835>

1 INTRODUCTION

“Applications of reinforcement learning are still far from routine and typically require as much art as science” (Sutton and Barto [2018]). The development process for reinforcement learning (RL) applications is exploratory rather than systematic, which reduces reuse between applications and increases the chances of introducing errors into particular implementations, lowering trustworthiness and effectiveness. This is especially important in areas where reinforcement learning is used to control physical devices (e.g., embedded or cyber-physical systems, and robots). Techniques and tools for systematic quality assurance of reinforcement learning applications are rare in the field.

Authors’ addresses: Mahsa Varshosaz, IT University of Copenhagen, Copenhagen, Denmark, mahv@itu.dk; Mohsen Ghaffari, IT University of Copenhagen, Copenhagen, Denmark, mohg@itu.dk; Einar Broch Johnsen, University of Oslo, Oslo, Norway, einarj@ifi.uio.no; Andrzej Wąsowski, IT University of Copenhagen, Copenhagen, Denmark, wasowski@itu.dk.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2023 Copyright held by the owner/author(s).

2475-1421/2023/8-ART193

<https://doi.org/10.1145/3607835>

Reinforcement learning is a method of machine learning, in which an agent experiments interactively with an environment, receiving rewards for these interactions. The most popular learning algorithms are *model-free*, as they do not need a complete model of the environment but learn from sampling executions. The goal of a learning algorithm is to find an optimal behavior policy for the agent which maximizes long-term expected reward. Reinforcement learning research often focuses on evaluating the quality of obtained policies. However, the results of even the most carefully designed evaluation experiments have little value, unless the evaluated methods are implemented correctly. Our long-term goal is to make the development and test of reinforcement learning applications more systematic and the outcomes more trustworthy by enabling easily accessible automatic testing. In this paper, we take a step towards a direct formal specification of correctness for reinforcement learning problems (agents) and the *learning algorithms* themselves, as opposed to the policies that they output.

We focus on *temporal difference* (TD) reinforcement learning methods [Sutton 1988], a large and well-established class of model-free methods. The majority of reinforcement-learning algorithms in use are TD algorithms. The TD algorithms update an estimate of a state action value function using a number of sampling steps and an estimate (bootstrapping). The *temporal difference* in the name refers to the fact that an update for a state-and-action expected reward is performed not immediately but after one or more execution steps. As we focus on the correctness of the algorithm and the agent, we settle for simple representations of value estimation—discrete tables. However, the core structure of our specification appears relevant for approximating methods.

We perform a domain analysis of the TD algorithms domain, leading to a formal specification. We characterize commonalities and differences between different reinforcement learning problems and between TD algorithms. We pay special attention to the update step in the algorithms, commonly described using so-called *backup diagrams* [Sutton 1988]. Our domain analysis formally defines *bdl*, or a *back-up diagram language*—a compositional, domain-specific language for describing updates in reinforcement learning. An interpreter for *bdl*, formalized in a denotational style, serves as a specification of correctness for updates in individual TD algorithms. A compositional denotational definition of this interpreter is what allows us to characterize many TD algorithms at once.

The obtained specification may be used for testing and verification (after embedding in a suitable formal system). We use testing to demonstrate and evaluate it here. We implement *Q*-Learning, SARSA, and Expected SARSA along with a number of case studies extracted from text books and papers. We add a number of tests derived from the formal modeling of agents and implement the *bdl* interpreter to serve as an oracle in property-based tests in the style of Quickcheck [Claessen and Hughes 2000]. Using an interpreter as an oracle for testing a concrete implementation is a technique well-known to compiler engineers, but rarely used outside this community. Our applicative strongly-typed implementation of these algorithms in Scala 3 is concise and traceable to the formal definitions in the paper. All code and tests are available online.

The test harness can be imposed on different algorithms and can be extended with properties specific for an agent. We try it on three algorithms and nine agents. We experience that the framework can carry the implementation of various kinds of problems, and that the cost of customizing the tests (especially the cost of providing custom generators for property-based tests of agents) is not high. Furthermore, an experiment based on mutation testing [DeMillo et al. 1978] demonstrates that the test harness kills a vast majority of programs with randomly injected faults. Crucially, about half of all mutants for each problem are killed by *generic tests* that are written once and reused for all new agents, just by invocation. No additional effort from the programmers implementing new agents is required.

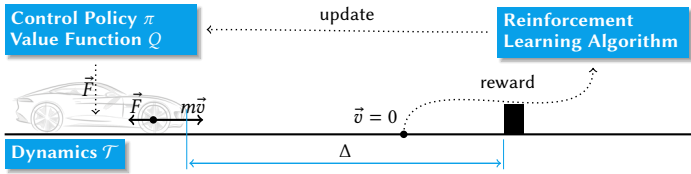


Fig. 1. Example: A car moves with velocity \vec{v} towards a fixed obstacle at distance Δ , learning how to brake. The control policy chooses a deceleration with which to brake. The agent receives a reward based on the location where stopped and updates the policy

We continue with a motivating example in Sect. 2. The technical contributions are:

- A formal specification of key elements of reinforcement learning problems and TD algorithms, including a detailed, self-contained, fully-formal definition of the update step (Sect. 4). The specification is probabilistic, applicative, compositional, and executable. We are not aware of any comparably detailed formal specification of reinforcement learning to date.
- A translation of the above specification into a test harness (Sect. 5), implemented (Sect. 6) following the paradigm of property-based testing. The test harness includes a general interpreter of *bdl* terms—formal models of updates for TD learning algorithms. It is used as an oracle. To our best knowledge this is the first property-based test suite for reinforcement learning.
- An evaluation of the test harness on SARSA and Expected SARSA for several agents using mutation testing (Sect. 7). The evaluation shows that the test harness is able to kill half of the generated mutants “for free,” i.e. without any customization of tests for new agents.

We discuss limitations (Sect. 8) and related work (Sect. 9), and conclude in Sect. 10.

2 MOTIVATING EXAMPLE: AN AGENT AND A LEARNING ALGORITHM

Consider a moving car that needs to stop before reaching a static obstacle (Fig. 1). We want to apply RL techniques to this problem and teach the car’s controller to avoid a collision with the obstacle. Hence, we formulate a reinforcement learning problem to which we can apply a RL algorithm.

In our problem formulation, the car controller (the agent) interacts with the road and the obstacle (the dynamic environment) to learn the control policy from experience. The states describe the possible positions and velocities of the car, and the actions the possible changes in velocity. The transition function then computes the effect of an action on a state. The goal is to avoid collision with the obstacle. We need to formulate a reward function reflecting this goal! Clearly, hitting the obstacle should give a negative reward; for other states and actions we might want to penalize unnecessarily sharp braking actions. How can we check that our transition and reward functions fit our intended reinforcement learning problem?

In this paper, we give a formal definition for a class of reinforcement learning problems, for which we implement a test harness using property-based testing. A good test suite that ensures the basic properties of RL problems, can significantly accelerate the task definition process. In fact, there are many potential pitfalls in defining such reinforcement learning problems. Some tests reflect implicit assumptions about the class of RL problems whereas others are specific to a particular RL problem. An example of the *generic* case is a test expressing that a transition from a so-called observable state (i.e., a state known to the reinforcement learning algorithm) will lead to another observable state. Such domain properties reflect real development problems in debugging RL implementations, based on our experience and on discussions with RL developers, and testing for simple properties helps eliminate the main bugs fast. An example of a *specific* problem property is, for our braking car

example, that the car cannot move backwards by braking. Problem-specific tests are needed to check that the transition and reward functions properly capture the characteristics of the problem domain.

Once the problem has been defined, we can apply a reinforcement learning algorithm to it and obtain a policy. The car should learn to halt before the obstacle and to avoid unnecessary sharp braking. Throughout the learning process, the car repeatedly starts from different states (i.e., positions and velocities), selects different actions (i.e., values of deceleration) and observes the resulting reward and change of state. Sometimes the car stops before the obstacle, sometimes it crashes. A reinforcement learning algorithm estimates the long-term effect of taking an action in a specific state. The estimate is updated by considering new rewards observed from the agent's interactions with the environment. This update step constitutes the core of such RL algorithms; the details of how and when to perform the update vary depending on the specific algorithm.

One example of a RL algorithm is SARSA (Fig. 2). The algorithm considers episodes that are sequences of states S_i and actions A_i that can be selected in those states, leading to a final state. A value function Q assigns a long-term reward estimation to state–action pairs. SARSA is an on-policy learning algorithm because the policy depends on Q when it selects an action in a given state. The rate by which learning affects Q is given by α , while γ defines the long-term reward discount.

Even though the standard presentation of SARSA (Fig. 2) makes an impression of an imperative program, a programmer would quickly notice that one cannot implement it without much additional knowledge. For instance, the meaning of “observe” and “reward” in Line 3 requires understanding what is an agent, what operations it supports, and with what semantics. The link between ε and π in l. 4 is not explicit. The policy π , mapping states to optimal actions, needs knowing in which state to select A_{t+1} . Also value function Q needs to be involved, as it defines which action is the best (so π and Q are dependent). All these problems could be fixed in Fig. 2 by refining the pseudo code, but they get difficult to fix for more advanced temporal difference algorithms, for which the update equation in Line 5 considers multiple states and actions at different times. The monolithic presentation no longer scales. Without a formal compositional specification, it is not only hard to implement the algorithms, but even harder to say whether the implementation is correct. Finally, it is non-obvious that the program is probabilistic and that several identifiers and expressions represent random variables.

As TD learning algorithms follow a common structure, we can capture this structure in a language and build a probabilistic interpreter for it to serve as a specification and an oracle. A run of this interpreter given a term defines the update of a particular TD algorithm. The term describing the SARSA update (l. 5) is $\text{sample}^V \text{Update}^\alpha \text{sample}$, and a denotational semantics precisely defines its meaning.

In this paper, we formalise TD learning algorithms and build a test harness for them. We first define properties that should apply generally to the considered class of TD algorithms. An example of such a property, is that the action selected in any state is distributed according to the specified policy and Q -table, which we check with a statistical test. Similarly, we test the update of the Q -table, using our interpreter as an oracle. An erroneous update will typically not crash, but produce a wrong value; a broken update step might go unnoticed for a long time, only manifesting itself in subpar results from the learning process. Our test suite facilitates the detection of such subpar behavior.

3 PRELIMINARIES

We follow the notation of [Sutton and Barto \[2018\]](#) and conventions of lambda-calculus; e.g., we omit parentheses around function arguments and write fx instead of $f(x)$. Sets and types are boldfaced.

A probability distribution over a finite set \mathbf{A} is a function $\text{Pr}_{\mathbf{A}} \in \mathbf{A} \rightarrow [0; 1]$ that assigns some probability mass to each element $a \in \mathbf{A}$, and that satisfies the usual axioms of probability. We write $\text{pmf } \mathbf{A}$ for the set of all *probability mass functions* (or distributions) over the set \mathbf{A} . If \mathbf{A} is continuous, we write $\text{pdf } \mathbf{A}$ for the set of all *probability density functions* $\text{Pr}_{\mathbf{A}} \in \mathbf{A} \rightarrow \mathbb{R}_+ \cup \{0\}$ over \mathbf{A} . Since the co-domains of functions in this paper will often be such probability mass or density functions, we

- 1: Initialize S_t , select action A_t using the current policy π (ϵ -greedy)
- 2: **while** S_t is not final **do**
- 3: Execute A_t , observe the next state S_{t+1} and reward R_{t+1}
- 4: Select next action A_{t+1} using the ϵ -greedy policy derived from Q
- 5: $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t))$
- 6: $S_t \leftarrow S_{t+1}, A_t \leftarrow A_{t+1}$

Fig. 2. A standard impure presentation of the popular SARSA algorithm following the style adopted in the literature [Sutton and Barto 2018]. The pseudocode shows one learning episode, until the agent reaches a final state. Normally a large number of episodes is run, updating the same global Q -table. The most characteristic part of SARSA is the update rule in Line 5. It uses the current value $Q(S_{t+1}, A_{t+1})$ as an approximation of the future long-run reward. The action A_{t+1} is selected probabilistically according to the current policy.

parenthesize expressions that return distributions, to emphasize this; e.g., if $f x y$ returns a distribution over $Z \ni z$, we write $(f x y) z$ for the probability (or density) assigned to z by the function $f x y$.

A probability mass function Pr_A and a function $f : A \rightarrow \text{pmf } B$ induce a probability distribution over the set B following the chain rule (by picking a value $a \in A$ according to Pr_A , applying f to a , then picking a value in B according to the obtained distribution).

$$(\text{Pr}_A \gg f) \in \text{pmf } B \quad (\text{Pr}_A \gg f) b = \sum_{a \in A} \text{Pr}_A a \cdot (f a) b \quad (1)$$

Some readers might find it useful to know that the left-associative composition operator \gg between a distribution and an into-distribution function is an instance of a monadic bind [Ramsey and Pfeiffer 2002]. If Pr_A is a pdf and $f : A \rightarrow \text{pdf } B$ is measurable, we similarly define:

$$(\text{Pr}_A \gg f) \in \text{pdf } B \quad (\text{Pr}_A \gg f) b = \int_{a \in A} \text{Pr}_A a \cdot (f a) b \quad (2)$$

The integral (technically, the measure) is used to formally describe the behavior of a RL system. Our implementation replaces the computation of probability measures with estimation by sampling (so the source domains are also discrete in any given run).

Given two functions $f \in A \rightarrow \text{pmf } B$ and $g \in B \rightarrow \text{pmf } C$ their Kleisli composition $f \gg g$ is the unique function from $A \rightarrow \text{pmf } C$ given below, Eq. (3). The Kleisli composition “flattens” (multiplies) nested probability distributions during function composition.

$$f \gg g = \lambda a. f a \gg g \quad (3)$$

Given a constant $x \in X$, we write $\text{Det } x$ for the unique deterministic distribution such that $(\text{Det } x) x' = 1$ iff $x = x'$ and zero otherwise (the Dirac distribution).

From the reinforcement learning and testing perspective, it does not matter whether one performs statistical tests following the Bayesian or the frequentist tradition [Gelman and Shalizi 2013]. Somewhat arbitrarily we make the Bayesian choice [Gelman et al. 2013; Kruschke 2014]. Given a prior distribution with parameter θ over a random variable X , and given a number of observations δ of values drawn from X , a Bayesian analysis rests on computing or estimating a posterior belief distribution on θ given the observations. One typically uses Bayes’ theorem: $\text{Pr}(\theta | \delta) \approx \text{Pr}(\delta | \theta) \cdot \text{Pr}(\theta)$, where $\text{Pr}(\delta | \theta)$ is known as the likelihood function. The likelihood is typically easy to formulate as it is generative: for each value of θ it can give a probability of generating δ . In the context of testing probabilistic programs, the likelihood will be given by the program semantics. Once the posterior distribution is established, one queries it for probabilities of relevant facts: for instance what is the belief that $\theta \geq 0.9$. In general, the posterior can be an arbitrarily complex function, but if the prior is specifically selected for the likelihood (a *conjugate* prior), a

kind of closure is obtained: the prior and the posterior have the same format. For conjugate priors, often an easy syntactic rule exists for updating the prior given the observations. We exploit such in this paper for Bernoulli and Gaussian priors to write simple, but computationally efficient tests.

4 A FORMAL SPECIFICATION OF REINFORCEMENT LEARNING

In reinforcement learning, an agent interacts with an environment in order to learn its behavior by trial-and-error. The agent's aim is to find a policy that maximizes a measure of reward [Kaelbling et al. 1996]. A *reinforcement learning problem* defines the agent's state space, dynamics (transition relation), and the rewards. The problem is handed over to a *learning algorithm* to search for the optimal policy, given the problem. The first key technical development of this paper is to formalize both the RL problems and the RL algorithms in order to allow reason and testing about the correctness of learning processes.

4.1 Reinforcement Learning Problems (Agents)

Definition 4.1. A *Reinforcement Learning Problem* is represented by a tuple $(\overline{\mathbf{State}}, \overline{\mathbf{State}}_0, \mathbf{Action}, \mathbf{State}, O, \mathcal{T}, \mathcal{R}, \mathcal{F})$ where:

- \mathbf{r}_1 : $\overline{\mathbf{State}}$ is a possibly infinite set of states of the environment and the agent combined,
- \mathbf{r}_2 : $\overline{\mathbf{State}}_0 \in \text{pdf } \overline{\mathbf{State}}$ is a density function defining probability for initial states,
- \mathbf{r}_3 : \mathbf{Action} is a finite set of actions that an agent can take,
- \mathbf{r}_4 : \mathbf{State} is a finite set of observable states,
- \mathbf{r}_5 : $O \in \overline{\mathbf{State}} \rightarrow \mathbf{State}$ is a total observation function,
- \mathbf{r}_6 : $\mathcal{T} \in \overline{\mathbf{State}} \rightarrow \mathbf{Action} \rightarrow \text{pdf } \overline{\mathbf{State}}$ is the transition probability function,
- \mathbf{r}_7 : $\mathcal{R} \in \overline{\mathbf{State}} \rightarrow \mathbf{Action} \rightarrow \mathbb{R}$ is the reward function, and
- \mathbf{r}_8 : $\mathcal{F} \in \overline{\mathbf{State}} \rightarrow \{0, 1\}$ is a predicate defining which observable states are final for a training episode. Initial states are not final, i.e., if $\overline{\mathbf{State}}_0(\overline{S}) > 0$ then not $\mathcal{F}(O \overline{S})$.

The agent perceives the world through a discrete observation function O (\mathbf{r}_5), mapping the possibly continuous state space $\overline{\mathbf{State}}$ of the environment (\mathbf{r}_1) to a finite state space \mathbf{State} of the learning algorithm (\mathbf{r}_4). For the rest of the paper, it is useful to remember that the identifiers with a line over refer to the actual state space of the environment, while those without refer to the state space abstraction that the agent can observe (the observable state space). The reward function $\mathcal{R} \overline{S} A$ defines the reward received by the agent (\mathbf{r}_7) when arriving at state \overline{S} after taking the action A . The transition function \mathcal{T} defines a distribution of successor states $\mathcal{T} \overline{S} A$ given a source state \overline{S} and an action A , see \mathbf{r}_6 . The function \mathcal{T} captures the stochastic environment behavior (e.g., noise) by returning a distribution. Its values are density functions, to account for the continuous state of the environment. Generally, \mathcal{T} defines a partially observable Markov Decision Process (MDP) with rewards and the composition $\mathcal{T} \gg O$ projects it onto a finite state MDP.

Given a reinforcement learning problem, a *run* is a sequence $\overline{S}_0 A_0 R_1 \cdots \overline{S}_{t-1} A_{t-1} R_t \cdots$, where all transitions have non-zero probability: $(\mathcal{T} \overline{S}_i A_i) \overline{S}_{i+1} > 0$ and $R_{i+1} = \mathcal{R} \overline{S}_{i+1} A_i$. Each state \overline{S}_i in a run marks a discrete time *epoch* in which the system performs the action A_i and receives a reward R_{i+1} .

Definition 4.2. A RL problem is *episodic* iff every run from an initial state eventually reaches some final state \overline{S} , so $\mathcal{F}(O \overline{S}) = 1$, see \mathbf{r}_8 in Def. 4.1. Otherwise the problem is *non-episodic*.

Example 4.3. To exemplify the definitions, we instantiate Def. 4.1 for the car example of Fig. 1.

- \mathbf{r}_1 : The set of states of the environment is $\overline{\mathbf{State}} = [0.0, 15.0] \times [0.0, 10.0]$, where $[0.0, 15.0]$ is the interval of possible positions and $[0.0, 10.0]$ is the interval of possible velocities. For a state $\overline{S} \in \overline{\mathbf{State}}$, we write $\overline{S}.p$ for its position and $\overline{S}.v$ for its velocity. See Fig. 3.

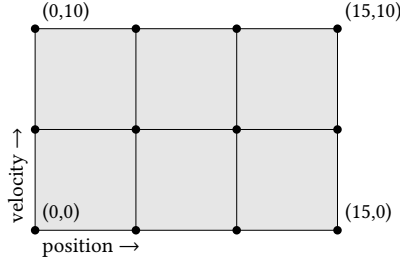


Fig. 3. The state space for the braking car example of Fig. 1. Each state is a pair of numbers representing the position of the car and its forward velocity. The environment state space is the entire gray rectangle. The black bullet points represent the observable states. The observation function \mathcal{O} maps the interior of each of the small squares (including its bottom and left edges) to the observable state in its bottom-left corner.

- r₂**: The car can start in any state with equal probability, so $\overline{\mathbf{State}}_0$ is a uniform distribution over the gray rectangle in Fig. 3: $\overline{\mathbf{State}}_0 = \text{Uniform}(\overline{\mathbf{State}})$.
- r₃**: The set of actions is $\mathbf{Action} = \{-10.0, -5.0, -2.5, -0.5, -0.05, -0.01, -0.001\}$ represents possible deceleration (negative acceleration) rates corresponding to increasingly gentle braking.
- r₄**: The set of observable states is $\mathbf{State} = \{0.0, 5.0, 10.0, 15.0\} \times \{0.0, 5.0, 10.0\}$, so even though the environment admits infinitely many position–velocity combinations, the agent can only observe 12 pairs, represented as black points in Fig. 3.
- r₅**: The observation function discretizes each car state by bringing each of its components to the largest multiple of five smaller than the value (in Fig. 3 each square is discretized to its bottom-left corner):

$$\mathcal{O} \bar{S} = S \quad \text{where} \quad S.p = 5 \cdot \left\lfloor \frac{\bar{S}.p}{5} \right\rfloor \quad \text{and} \quad S.v = 5 \cdot \left\lfloor \frac{\bar{S}.v}{5} \right\rfloor.$$

- r₆**: The transition function \mathcal{T} selects the next state given a predecessor state \bar{S}_t and an action A_t . In this example, the successor state \bar{S}_{t+1} is calculated deterministically based on the car dynamics (assuming that the car moves for a time step δ):

$$\bar{S}_{t+1}.p = \bar{S}_t.p + \bar{S}_t.v\delta + A_t\delta^2/2 \quad \text{and} \quad \bar{S}_{t+1}.v = \bar{S}_t.v + A_t\delta.$$

Since the successor state is computed deterministically we get $\mathcal{T} \bar{S}_t A_t = \text{Det}(\bar{S}_{t+1})$, a Dirac distribution over successor states.

- r₇**: The reward for reaching a state \bar{S}_{t+1} after taking the action A_t is

$$\mathcal{R} \bar{S}_{t+1} A_t = -100 \quad \text{if} \quad \bar{S}_{t+1}.p \geq 10 \quad \text{and} \quad \mathcal{R} \bar{S}_{t+1} A_t = A_t \quad \text{otherwise.}$$

The model assumes that the obstacle is found at position 10 (the rightmost edge in Fig. 3). The first case penalizes a crash. The second case (no crash) says that a sharper deceleration results in a lower reward; the reward is proportional to the action value in this case, to penalize violently abrupt braking.

- r₈**: The braking car problem is episodic. The car is in a final state if either it has come to a full stop or it has crashed, so $\mathcal{F} S$ is true if and only if $S.v = 0.0 \parallel S.p \geq 10$. \square

A *policy* for an agent defines its behavior at any state. It maps observed states to actions that an agent takes in those states. Policies are derived from richer *value functions* estimating the ultimate reward of an execution started by each action in a state, $Q \in \mathbf{State} \times \mathbf{Action} \rightarrow \mathbb{R}$. We let \mathbf{Q} denote the set of all value functions. A policy then becomes a probability function that, given a value

function, represents the distribution of plausible actions in each state.

$$\pi \in \mathbf{Q} \rightarrow \mathbf{State} \rightarrow \text{pmf } \mathbf{Action} \quad (4)$$

A greedy policy returns the action maximizing expected reward for a given state deterministically (with probability one). A greedy policy is not effective for learning, as it may follow globally sub-optimal actions, especially at an early stage of learning. A popular policy in RL algorithms is the ε -greedy policy that forces the agent to try random actions with small probability $\varepsilon \in [0, 1]$ and use the locally best action otherwise:

$$(\pi Q_t S_t) A_t = \begin{cases} 1 - \varepsilon + \varepsilon \cdot |\mathbf{Action}|^{-1} & A_t = \arg \max_A Q_t(S_t, A) \\ \varepsilon \cdot |\mathbf{Action}|^{-1} & \text{otherwise} \end{cases} \quad (5)$$

Solving a RL problem means finding the policy that maximizes the expected reward over many episodes. For non-episodic tasks a discounted expected reward over an infinite run is maximized.

4.2 Temporal Difference Learning Algorithms

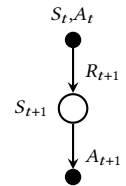
We will now summarize the update methods of RL algorithms following the style adopted in the literature of the field. We will rectify the shortcomings of this style in later sections.

In implementations, the value function is often represented as a table with a value for each state-action pair, a so-called Q -table. (In deep reinforcement learning, it is approximated using a neural network, but we are not concerned with this here, as we are formalising the core structure of the algorithms.) Learning happens by experimentation: an agent tries an action, receives a reward, and updates the Q -table entry for that action. How precisely an update is made is the very essence of each reinforcement learning algorithm. A large class subsuming most common designs are the *temporal-difference* (TD) algorithms, which update the value function after accumulating the reward over a (possibly singleton) sequence of actions. Given a state S_t and action A_t , the general update rule for a TD prediction method is (written as an imperative assignment, as commonly practiced in the literature of the field):

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \cdot (G_t - Q(S_t, A_t)) . \quad (6)$$

The new state-action value estimation is the old estimate $Q(S_t, A_t)$ corrected by its error against the new estimate G_t , discounted by a coefficient α . The *return* G_t is the newly obtained expected reward value. Its calculation varies, depending on the specific TD method used. The *TD error* value $\alpha(G_t - Q(S_t, A_t))$ represents the difference between the new estimate and the old estimate of the expected reward, where $\alpha \in [0, 1]$ is the learning rate defining the trade-off between learning and remembering. If $\alpha = 1$ then the old estimate is entirely forgotten and the new estimate is adopted. If $\alpha = 0$ the new estimate is ignored: no learning happens. Other values of α control the speed with which the new experiences influence the current policy. Concrete update rules for different TD algorithms are instances of the general rule given in Eq. (6) with different returns G_t .

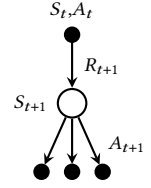
Example 4.4. The popular SARSA algorithm [Rummery and Niranjan 1994] is a TD algorithm. The pseudocode is shown in Fig. 2 (Sect. 2). Given a Q -table and the learning rate α , it performs the following steps for a prescribed number of episodes. The second but last line, performs the update. The term $R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$ defines the return G_t in this case, where $\gamma \in [0, 1]$ is the so-called discount factor, weighing immediate rewards vs future rewards. \square



Reinforcement learning researchers often visualize the calculation of the return in a so called backup diagram. The diagram for SARSA is shown to the right. The top arrow means

that the reward R_{t+1} is obtained first by sampling the environment using action A_t from state S_t . The second arrow means that the action A_{t+1} is sampled from the policy and its current value estimate in $Q(S_{t+1}, A_{t+1})$ is used to compute the return and update $Q(S_t, A_t)$. In general, a backup diagram is a stylized list of steps, where all but last interact with the environment, while the last one uses a prior estimate. The proliferation of backup diagrams in conference presentations, lectures, and teaching materials on reinforcement learning is testimony to the pivotal role of the update and its structure for understanding the RL algorithms.

Example 4.5. The Expected SARSA algorithm [Van Seijen et al. 2009] follows the same steps as SARSA (Example 4.4) just with a different update rule. It uses an estimate for all possible actions in the second step instead of the single best action chosen by the current policy π . The summation below computes an expectation over all actions in the next step discounted by factor γ . A backup diagram for the Expected SARSA update rule is shown to the right. Note that the second step differs from SARSA, reflecting the use of expectation instead of following the current policy greedily.



$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \sum_a \pi(a|S_{t+1})Q(S_{t+1}, a) - Q(S_t, A_t)). \quad \square$$

SARSA and Expected SARSA are *on-policy* algorithms as they improve the same policy that it is following to select subsequent actions (a decision making policy). In other algorithms, the updated policy and the followed policy might differ (off-policy learning).

Example 4.6. Q -learning is a popular and effective learning algorithm similar to SARSA [Watkins 1989]. Its update equation is the same as that of SARSA and Expected SARSA if the updated policy π is greedy ($\epsilon = 0$) [De Asis et al. 2018]. Q -learning can be seen as an off-policy version of SARSA, that allows choosing actions using an ϵ -greedy policy ($\epsilon > 0$), but performing an update using an estimation based on a greedy policy ($\epsilon = 0$). However, in the scope of this paper, where we focus on semantics of the update procedure, both algorithms have the same specification. The difference between Q -learning and SARSA manifests when multiple updates are composed iteratively (see Sect. 8). \square

In all the above algorithms the update rule is similar and the main difference is in the calculation of the return. The update is based on a single step ($n = 1$), the received return, and an estimation of the value for the state-action pair for the next step in the Q -table. The entries in the table serve as a proxy for the remaining achievable long-term reward. All these algorithms can be generalized to perform an update after $n > 1$ steps, however the imperative non-compositional presentation gets very unwieldy. For this reason, we postpone the generalization to the formal compositional definition below.

4.3 Formalizing Temporal Difference Reinforcement Learning Algorithms

Despite the mathematical notation, the standard presentation of these algorithms in the literature remains relatively informal and hides many intricacies. Crucially, in this style, the update equations get more and more complex, eventually spanning several lines for the most complex methods in the classic textbook of Sutton and Barto [2018]. Even though the temporal difference algorithms share semantic similarity, the monolithic presentation makes it difficult to appreciate and exploit this similarity in formal reasoning and testing. This not only makes the reinforcement learning algorithms difficult to implement, but also hinders formal reasoning about and testing of desirable behaviors.

Let us point out a few challenges concretely. For example, R_{t+1} above is the immediate reward when an agent transitions to a state as a result of taking an action. This reward should be calculated using the model of the agent and the environment, so the functions \mathcal{T} , \mathcal{O} , and \mathcal{R} in our formalization of the RL problems, but this is not visible in the standard presentation. Similarly, A_{t+1} denotes an

action selected using the policy π , but the standard presentation does not relate A_{t+1} and π in any way. This relation is non-trivial; at runtime, the policy is not fixed, as it depends on the Q -table, which is regularly updated. In some reinforcement learning algorithms the policy also undergoes temporal changes, for instance, the exploration ratio ϵ is decreased over time. Which version of the policy π should select A_{t+1} ? Similarly, the action selection depends on the current state but you cannot see this in the equation. Finally, while it appears that A_{t+1} is a concrete value, semantically it stands for a *random variable*, as the functions π and \mathcal{T} are stochastic. This is the case for most of the elements in the algorithm, including states S_t and rewards R_t . An additional complication is that for TD algorithms looking beyond one step ahead, the update uses values from past time epochs (past iterations of the loop).

Obviously, a test or a correctness proof has to specify these time dependencies, values, and probability distributions explicitly and precisely. Instead of hiding meanings in variable names and indices, one needs a specification robust to alpha-renaming that makes dependencies explicit. For this reason we will now formalize the calculation of the *return* for a large class of TD algorithms. It is possible to handle many languages simultaneously, by defining a small term language to formally specify them, and defining the semantics of the update rule as an interpreter for this term language. Our term language for update rules is inspired by the backup diagrams. If RL researchers find them important and informative, they must convey important information from the domain expertise perspective. Unfortunately, the diagrams themselves are quite informal, not clearly compositional, and lack details. To address these issues we propose a textual syntax generated by the following grammar.

$$\begin{aligned} est & ::= \text{sample}^\gamma \mid \text{expectation}^\gamma \\ bdl & ::= est^+ \text{Update}^\alpha(\text{sample} \mid \text{expectation}) \ . \end{aligned} \quad (7)$$






An *est* term, corresponding to a single segment in a backup diagram, represents an estimation step parametrized by a discount factor $\gamma \in [0, 1]$. It estimates state-action values by sampling (sample^γ) or by averaging ($\text{expectation}^\gamma$) state-action-value estimates over all possible actions in a state. A *bdl* term, corresponding to an entire diagram, combines a non-empty sequence of estimation steps (est^+) with a final update—the last segment in each diagram. Update^α should be seen as infix operator parametrized by a learning rate $\alpha \in [0, 1]$, one for a sampling update and one for an expectation update.

Example 4.7. Table 1 lists backup diagrams (first column), their *bdl* abstract syntax (second column), and Sutton-and-Barto-style return calculations for five examples of temporal difference algorithms (third column); only the first one was discussed above. The *bdl* expression for 1-step SARSA is: $\text{sample}^\gamma \text{Update}^\alpha \text{sample}$. As shown in Example 4.4, the update step includes updating the Q -value of a state-action pair (S_t, A_t) using the reward and the Q -value of the state-action pair (S_{t+1}, A_{t+1}) resulting from one time policy sampling. Similarly, for the 2-step SARSA, the diagrams represent two sampling steps composed with a sampling update: $\text{sample}^\gamma \text{sample}^\gamma \text{Update}^\alpha \text{sample}$. In contrast, the update in the last step of n -step Expected SARSA is calculated by taking an expectation of values for all possible actions instead of using the value for the sampled action. In the third column, $G_{t:t+n}$ and $\pi(a|s)$ are notations adopted by Sutton and Barto [2018] to denote, respectively, the n -step return, from time step t to $t+n$, and the probability of taking action a in state s following policy π . \square

We can now specify the update step of the TD algorithms compositionally by giving formal semantics to elements of a *bdl* term. We map each basic element type to a function. The semantics of an entire backup diagram (and thus of an update) is given by a function composition. This style can be directly implemented in a functional programming language and used, for instance, for testing.

Table 1. Examples: A representation of updates in TD learning as a backup diagram, a *bdl* term, and a return calculation (G_t in Eq. (6)) after Sutton and Barto [2018]. The index T denotes the final time step in an episode.

Diagram **BDL (abstract syntax)** **Return ($G_{t:t+n}$) in Update formulae [Sutton and Barto 2018]**

	<p>1-step SARSA sample^{γ} Update^{α} sample</p>	$R_{t+1} + \gamma Q_t(S_{t+1}, A_{t+1})$
	<p>2-step SARSA sample^{γ} sample^{γ} Update^{α} sample</p>	$R_{t+1} + \gamma R_{t+2} + \gamma^2 Q_{t+1}(S_{t+2}, A_{t+2})$
	<p>n-step SARSA (sample^{γ})^{n} Update^{α} sample</p>	$\begin{cases} R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{n-1} R_{t+n} \\ \quad + \gamma^n Q_{t+n-1}(S_{t+n}, A_{t+n}) & \text{if } n \geq 1 \wedge t < T - n \\ R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} \dots + \\ \quad \gamma^{T-t-1} R_T & \text{otherwise} \end{cases}$
	<p>n-step expected SARSA (sample^{γ})^{n} Update^{α} expectation</p>	$\begin{cases} R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{n-1} R_{t+n} \\ \quad + \gamma^n \sum_a \pi(a S_{t+1}) Q_{t+n-1}(S_{t+1}, a) & \text{if } t < T - n \\ R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} \dots \\ \quad + \gamma^{T-t-1} R_T & \text{otherwise} \end{cases}$
	<p>n-step tree backup (expectation^{γ})^{n} Update^{α} expectation</p>	$\begin{cases} R_{t+1} + \gamma \sum_{a \neq A_{t+1}} \pi(a S_{t+1}) Q_{t+n-1}(S_{t+1}, a) \\ \quad + \gamma \pi(A_{t+1} S_{t+1}) G_{t+1:t+n} & \text{if } t < T \wedge n \geq 2 \\ R_{t+1} + \gamma \sum_a \pi(a S_{t+1}) Q_t(S_{t+1}, a) & \text{if } n = 1 \\ R_T & \text{if } t = T - 1 \end{cases}$

Even more interestingly, an *interpreter* for *bdl* terms can be implemented in functional style and used for testing updates of direct implementations of TD algorithms.

An estimation step, say sample^y , is based on a current table $Q_t \in \mathbf{Q}$, a source state $\bar{S}_t \in \overline{\mathbf{State}}$, an action $A_t \in \mathbf{Action}$ that the system decided to perform, a return value $G_t \in \mathbb{R}$ from the preceding estimation steps, and a discount factor $\gamma_t \in [0, 1]$ for this and subsequent steps. It runs a step of the environment \mathcal{T} and returns the reached successor state, the obtained reward value, and the action to perform in the next step. To make estimation steps compositional, the current discount factor, which can depend on the step, and the cumulative discounted reward up to this step (current return) are included as a part of the semantic domain. Since the environment may contain stochastic behavior and the action selection may be probabilistic (as we use ϵ -greedy policies), the semantics of an estimation step is actually a multivariate probability distribution over target states, actions, returns, and discount factors, resulting in the following semantic domain.

$$\llbracket \cdot \rrbracket_{est} \in \mathbf{Q} \rightarrow \overline{\mathbf{State}} \times \mathbf{Action} \times \mathbb{R} \times \mathbb{R} \rightarrow \text{pmf}(\overline{\mathbf{State}} \times \mathbf{Action} \times \mathbb{R} \times \mathbb{R}) \quad (8)$$

The semantic function for a sampling estimation step is:

$$\begin{aligned} \llbracket \text{sample}^y \rrbracket_{est} Q_t &= \lambda(\bar{S}_t, A_t, G_t, \gamma_t). \mathcal{T} \bar{S}_t A_t \gg && \text{(run the system in state } \bar{S}_t \text{ executing action } A_t) \\ &\lambda \bar{S}_{t+1}. \text{Det} \left(\mathcal{R} \bar{S}_{t+1} A_t \right) \gg && \text{(reward } R_{t+1} \text{ for the action and the resulting state)} \\ &\lambda R_{t+1}. \text{Det} \left(\mathcal{O} \bar{S}_{t+1} \right) \gg && \text{(observe discrete state } S_{t+1} \text{ reached)} \\ &\lambda S_{t+1}. \pi Q_t S_{t+1} \gg && \text{(select the next action } A_{t+1} \text{ following policy } \pi) \\ &\lambda A_{t+1}. \text{Det} (G_t + \gamma_t R_{t+1}) \gg && \text{(discount accumulated return by } \gamma_t \text{ and add } R_{t+1}) \\ &\lambda G_{t+1}. \text{Det} (\gamma_t \cdot \gamma) \gg && \text{(accumulate the discount factor)} \\ &\lambda \gamma_{t+1}. \text{Det} (\bar{S}_{t+1}, A_{t+1}, G_{t+1}, \gamma_{t+1}) && (9) \end{aligned}$$

First, the step function is applied to the environment state \bar{S}_t and action A_t . This results in a distribution over successor states, and we bind a successor to \bar{S}_{t+1} . The reward is computed deterministically for a pair of successor state \bar{S}_{t+1} and the performed action A_t , the result bound to R_{t+1} . The observable target state S_{t+1} is obtained using a deterministic function \mathcal{O} , and the next action is selected on policy π obtaining a non-trivial distribution again. At this point, we compute the return by adding the prior return G_t with the discounted reward. Finally, the discount factor is updated by this step's discount rate γ . In the last line all the four elements, i.e., next state \bar{S}_{t+1} , next action A_{t+1} , return G_{t+1} , and discount factor γ_{t+1} to be used in the next step, are returned.

An advantage of this presentation is that it makes it explicit where the values of S_{t+1} , A_{t+1} , and R_{t+1} come from, which steps are deterministic, and which result in proper random variables (those not generated by Det). The first four function applications in Eq. (9), show how these values are obtained.

The semantics of an expectation step is defined similarly below. The differences between sampling and expectations are highlighted in blue—the expectation step uses an expected value of the estimated return instead of a sample reward when calculating the return.

$$\begin{aligned}
\llbracket \text{expectation}^Y \rrbracket_{est} Q_t = & \\
\lambda(\bar{S}_t, A_t, G_{:t}, \gamma_t). \mathcal{T} \bar{S}_t A_t \gg & \quad (\text{run the system step}) \\
\lambda \bar{S}_{t+1}. \text{Det} \left(\mathcal{R} \bar{S}_{t+1} A_t \right) \gg & \quad (\text{calculate reward}) \\
\lambda R_{t+1}. \text{Det} (O \bar{S}_{t+1}) \gg & \quad (\text{perform observation}) \\
\lambda S_{t+1}. \pi Q_t S_{t+1} \gg & \quad (\text{select next action}) \\
\lambda A_{t+1}. \text{Det} \left(G_{:t} + \gamma_t \left[R_{t+1} + \sum_{A \neq A_{t+1}} (\pi Q_t S_{t+1}) A \cdot Q_t(S_{t+1}, A) \right] \right) \gg & \quad (\text{update return with expected reward}) \\
\lambda G_{:t+1}. \text{Det} \left(\gamma_t \cdot \gamma \cdot (\pi Q_t S_{t+1}) A_{t+1} \right) \gg & \quad (\text{discount weighed by prob. of } A_{t+1}) \\
\lambda \gamma_{t+1}. \text{Det} \left(\bar{S}_{t+1}, A_{t+1}, G_{:t+1}, \gamma_{t+1} \right) & \quad (10)
\end{aligned}$$

The next return $G_{:t+1}$ in Eq. (10) is computed as a sum of current return $G_{:t}$, the discounted immediate reward R_{t+1} obtained by taking action A_t , and the expected return from alternative actions which are not selected as next action. This expectation is also discounted. The next discount factor is computed by multiplying the current discount factor with the constant discount factor for the step and the probability of taking the next action A_{t+1} . This calculation of discount factor is common for algorithms such as n -step tree backup [Sutton and Barto 2018].

The estimation steps can be composed into larger sequences of the same type.

$$\llbracket est_1 \cdots est_k \rrbracket_{est^+} \in \mathbf{Q} \rightarrow \overline{\mathbf{State}} \times \mathbf{Action} \times \mathbb{R} \times \mathbb{R} \rightarrow \text{pmf} (\overline{\mathbf{State}} \times \mathbf{Action} \times \mathbb{R} \times \mathbb{R}) \quad (11)$$

The composition is computed using the standard Kleisli composition:

$$\begin{aligned}
\llbracket est_1 \cdots est_k \rrbracket_{est^+} Q_t = & \\
\lambda(\bar{S}_t, A_t, G_{:t}, \gamma_t). \llbracket est_1 \rrbracket_{est} (Q_t) (\bar{S}_t, A_t, G_{:t}, \gamma_t) \gg & \quad (\text{the 1. estimation step}) \\
\lambda(\bar{S}_{t+1}, A_{t+1}, G_{:t+1}, \gamma_{t+1}). \llbracket est_2 \rrbracket_{est} (Q_t) (\bar{S}_{t+1}, A_{t+1}, G_{:t+1}, \gamma_{t+1}) \gg & \quad (\text{composed with the 2. estimation}) \\
\vdots & \\
\lambda(\bar{S}_{t+k-1}, A_{t+k-1}, G_{:t+k-1}, \gamma_{t+k-1}). & \quad (\text{composed with the } k\text{th estimation}) \\
\llbracket est_k \rrbracket_{est} (Q_t) (\bar{S}_{t+k-1}, A_{t+k-1}, G_{:t+k-1}, \gamma_{t+k-1}) & \quad (12)
\end{aligned}$$

The same can be stated compactly using the Kleisli composition of functions:

$$\llbracket est_1 \cdots est_k \rrbracket_{est^+} Q_t = \llbracket est_1 \rrbracket_{est} Q_t \gg \llbracket est_2 \rrbracket_{est} Q_t \gg \cdots \gg \llbracket est_k \rrbracket_{est} Q_t. \quad (13)$$

From the functional programming perspective, the above may appear an obvious step, but we are not aware of a similar observation in the reinforcement learning literature, where the update procedures for each algorithm tend to be presented monolithically and implemented from scratch, obscuring the common structure of TD algorithms. This also means that in a correctness proof, it is difficult to generalize from properties of a single estimate step to the entire composed update.

Finally, we define the meaning of an update step, which performs a sequence of estimation steps, given an initial state and action, estimates the value of a final action in the sequence (in the right-hand-side operand) using the Q -table, and finally performs an update of the respective entry of the table. After an update the agent lands in a new target state and has chosen the next action (for on-policy algorithms). The type of the semantics reflects this: we start with a Q -table, a state and an action, and land (with some stochastic disturbance) in a new Q -table, state, and a

subsequent action to execute.

$$\llbracket est^k \text{ Update}^\alpha (\text{sample} \mid \text{expectation}) \rrbracket_{bdl} \in \mathbf{Q} \times \overline{\mathbf{State}} \times \mathbf{Action} \rightarrow \text{pmf}(\mathbf{Q} \times \overline{\mathbf{State}} \times \mathbf{Action}) \quad (14)$$

Let us define this function, for the case of the sampling update first.

$$\begin{aligned} \llbracket est^k \text{ Update}^\alpha \text{ sample} \rrbracket_{bdl} = & \\ & \lambda(Q_t, \bar{S}_t, A_t). \llbracket est^k \rrbracket_{est^*} (Q_t) (\bar{S}_t, A_t, 0, 1) \gg \quad (\text{execute the estimation steps}) \\ & \lambda(\bar{S}_{t+k}, A_{t+k}, G_{t:t+k}, \gamma_{t+k}). \text{Det} \left(\mathcal{O} \bar{S}_t, \mathcal{O} \bar{S}_{t+k} \right) \gg \quad (\text{observe initial \& final state}) \\ & \lambda(S_t, S_{t+k}). \text{Det} (G_{t:t+k} + \gamma_{t+k} Q_t(S_{t+k}, A_{t+k})) \gg \quad (\text{return based on } A_{t+k}) \\ & \lambda G_{t:t+k+1}. \text{Det} (Q_t[(S_t, A_t) \mapsto Q_t(S_t, A_t) + \alpha [G_{t:t+k+1} - Q_t(S_t, A_t)]) \gg \quad (\text{update}) \\ & \lambda Q_{t+1}. \text{Det} (Q_{t+1}, \bar{S}_{t+k}, A_{t+k}) \quad (15) \end{aligned}$$

The notation $Q_t[x \mapsto y]$ in the penultimate line means a table entry substitution. It denotes a new Q -table, obtained from Q_t by replacing the entry at position x to contain the value y . All other entries remain unchanged. The first line of the above semantics executes all the estimation steps of the algorithm (if any), then establishes what is the observable initial and target states, and uses the Q -table entry of the target state to update the entry for the initial state–action pair.

An update with an expectation estimate is defined similarly below in Eq. (16). The highlighted difference is in the computation of the return $G_{t:t+k+1}$. In Eq. (15), the discounted estimated Q_t value for the last pair of state and action (S_{t+k}, A_{t+k}) is added in the update. In Eq. (16), a discounted expectation over the Q_t values over all actions in the next state S_{t+k} is used instead. The discount factor uses the accumulated update and the local contribution (cf. Eqs. (9) and (10)).

$$\begin{aligned} \llbracket est^k \text{ Update}^\alpha \text{ expectation} \rrbracket_{bdl} = & \\ & \lambda(Q_t, \bar{S}_t, A_t). \llbracket est^k \rrbracket_{est^*} (Q_t) (\bar{S}_t, A_t, 0, 1) \gg \quad (\text{execute the estimation steps}) \\ & \lambda(\bar{S}_{t+k}, A_{t+k}, G_{t:t+k}, \gamma_{t+k}). \text{Det} \left(\mathcal{O} \bar{S}_t, \mathcal{O} \bar{S}_{t+k} \right) \gg \quad (\text{observe initial \& final state}) \\ & \lambda(S_t, S_{t+k}). \text{Det} (G_{t:t+k} + \gamma_{t+k} \sum_A (\pi S_{t+k} A) Q_t(S_{t+k}, A)) \gg \quad (\text{expected return}) \\ & \lambda G_{t:t+k+1}. \text{Det} (Q_t[(S_t, A_t) \mapsto Q_t(S_t, A_t) + \alpha [G_{t:t+k+1} - Q_t(S_t, A_t)]) \gg \quad (\text{update}) \\ & \lambda Q_{t+1}. \text{Det} (Q_{t+1}, \bar{S}_{t+k}, A_{t+k}) \quad (16) \end{aligned}$$

The above functions provide compositional specifications for backup diagrams and for TD updates. As the Kleisli composition is associative, the above semantics is insensitive to the order of composing the functions. Since the above definitions are self-contained (all dependencies are explicit in the function constructions), they eliminate ambiguities seen in typical descriptions and can support automatic test of and formal reasoning about implementations. For example:

THEOREM 4.8. *For a greedy policy π (with $\varepsilon = 0$) SARSA and Expected SARSA perform updates in the same manner, i.e., $\llbracket \text{sample}^y \text{ Update}^\alpha \text{ sample} \rrbracket_{bdl} = \llbracket \text{sample}^y \text{ Update}^\alpha \text{ expectation} \rrbracket_{bdl}$.*

PROOF. The action A_{t+k} in Eq. (15) is always bound to the action with the highest entry in the Q -table, as the policy π is greedy and it assigns probability 1 to the highest value action (cf. Eq. (9)). The return calculation in Eq. (15) returns the value in the Q -table discounted by γ_{t+k} . Similarly, as π is greedy, it is a Dirac distribution, so in the return calculation in Eq. (16), it will have value 1 for exactly one action A which is the action with the highest entry of $Q(S_{t+k}, A)$. Consequently, $A = A_{t+k}$ and the sum in Eq. (16) collapses to the same term as in Eq. (9). Ultimately, the proof rests on the fact that the mean and the mode of a Dirac-distributed random variable are the same. \square

Recall that the update of Q -Learning, corresponds to an update of these algorithms if the policy π is greedy. Consequently, we can use the same specification to test the update in the Q -learning implementations. Without a formal specification it would be hard to make a similar statement.

5 SPECIFICATION-BASED TESTING OF REINFORCEMENT LEARNING

The presented properties can be used as a start of a formal verification project for an RL application. However, verification can quickly become intricate with specifics of the chosen learning algorithm. As we want to maintain a broad focus on many algorithms, while keeping the paper accessible and (relatively) short, we turn to automated testing for demonstration of our specification. We use the formal definitions in Sect. 4 to derive tests. Our long-term goal is to build a parameterized test harness and a method to write testable properties, to broadly lay the grounds for automating testing for RL.

The tests are organized in two main categories: the *tests of problem* definitions and the *tests of algorithms*. Problem tests check consistency properties of the agent and environment model, and of their interaction. We further distinguish between the *specific problem* tests, relevant only for a specific RL problem, and *generic tests* for problem definitions—the properties that should hold in general for all RL problem definitions. The algorithm tests check whether the learning algorithm behaves according to its design specification. In the following, we first focus on generic problem tests and then move to problem-specific and algorithm tests. Even though they are practically important, we devote less attention to the specific problem properties, as these are not reusable for larger groups of users. Their discussion quickly gets lost in the intricacies of a specific problem. In contrast, the generic problem properties and the tests of algorithms benefit more users directly, and can be pre-implemented in a library. We demonstrate deriving a selection of tests.

All tests below are described as abstract universal properties, basically logical statements. This way we minimize pollution by details of the programming language and style. We discuss how to concretize them as executable code in Sect. 6.

5.1 Testing Reinforcement Learning Problems (Agents)

Generic Problem Properties. These properties should hold for all implementations of RL problems. They are derived from the formal definition of a reinforcement learning problems in Sect. 4.1. We begin with the totality of the observation function O (Def. 4.1, requirement \mathbf{r}_5), a classic case of a *domain constraint*. The observation function O links the environment and the agent with the state space of the learning algorithm—as the algorithm only ‘sees’ and ‘learns’ about the observable states. The function O should be total in the sense that every system state should have a translation to an observable state; otherwise some system trajectories will lead to crashes or unexpected runs of the learning algorithm. Note that for most applications the domain constraints are not automatically enforced by the type system, as often only subsets of a type’s values are legal (say the set of positive floats as opposed to all floats). Typically, these subsets are not tracked by the type system, thus it is natural to resort to testing.¹

$$\forall \bar{s} \in \overline{\mathbf{State}}. O \bar{s} \in \mathbf{State} , \quad (17)$$

$$\forall \bar{s}_0 \in \overline{\mathbf{State}}. \overline{\mathbf{State}} \bar{s}_0 > 0 \rightarrow O \bar{s}_0 \in \mathbf{State} . \quad (18)$$

Equation (17) is testable if we can generate elements of $\overline{\mathbf{State}}$ and check membership in \mathbf{State} . In discrete RL, observable state spaces are finite and relatively small, so the latter is easily achieved by

¹In a strongly typed programming language totality of functions can be reflected in types. Unfortunately, pragmatics often prevents it. The environment states and observations are generated by a physical system, or a simulator, which may be implemented in another programming language than the learning algorithms or the test harness. For instance, in some of our Scala-based projects we use Java implementations of the environment to leverage the existing infrastructure.

enumerating the observable states. However, in the case of approximate learning, specifically in continuous state spaces, employing a membership predicate becomes more practical and feasible. That is, finding a proper observable state space is not straightforward and requires approximating based on the environment model or problem. This requirement also applies to initial system states, which follows by combining \mathbf{r}_2 and \mathbf{r}_5 . When testing it is useful to specify this requirement separately, to ensure that the property is tested on the initial states. See Eq. (18); recall that $\overline{\mathbf{State}_0}$ is a probability density function. The precondition that a density is positive ensures that the state is attainable.

We further require that the initial state is not final. Equation (19) tests the interaction of the pmf $\overline{\mathbf{State}_0}$ (requirement \mathbf{r}_2) and the predicate \mathcal{F} (requirement \mathbf{r}_8):

$$\forall \bar{S}_0 \in \overline{\mathbf{State}}. \overline{\mathbf{State}_0} \bar{S}_0 > 0 \rightarrow \neg \mathcal{F} (O \bar{S}_0) . \quad (19)$$

For the transition function \mathcal{T} , a domain constraint is obtained by combining requirements \mathbf{r}_5 , \mathbf{r}_6 and \mathbf{r}_2 in Eq. (20). Together with the properties above, the totality of observation leads to a *closure* property—starting in an observable state, we end in an observable state:

$$\forall \bar{S}_t, \bar{S}_{t+1} \in \overline{\mathbf{State}}. \forall A \in \mathbf{Action}. (\mathcal{T} \bar{S}_t A_t) \bar{S}_{t+1} > 0 \rightarrow O \bar{S}_{t+1} \in \mathbf{State} . \quad (20)$$

For episodic agents (Def. 4.2), we also test termination. Termination is difficult to establish by testing, but a random exploration strategy with a timeout is effective for simple RL problems. We implement it as a generic problem property with a timeout parameter (measured in discrete time epochs) in the model of episodic problems. This way different agents can be tested against different time horizons.

Some properties emerge from the composition of an algorithm and the problem definition, so one needs to involve both components (sections 4.1 and 4.3) in testing. One such example is a key property for reinforcement learning—*convergence*. Its violation can be caused by errors in the learning implementation, but also by overly liberal reward functions, a part of the problem definition. While convergence is not testable generally,² some special cases can be tested. In the context of a finite implementation we can test whether the accumulated reward values are representable in the range of double numbers (no overflow errors). An overflow of reward estimation during learning is effectively a sign of divergence. A test for overflow of rewards can be constructed by performing iterations of updates from various initial Q -tables. Given a learning algorithm u and a natural number of iterations n

$$\begin{aligned} &\forall Q_t \in \mathbf{Q}. \forall S_t \in \mathbf{State}. \forall A_t \in \mathbf{Action}. \\ &\forall Q \in \mathbf{Q}. [([u]_{\text{bdl}})^n(Q_t, S_t, A_t)] Q > 0 \\ &\rightarrow \forall S \in \mathbf{State}. \forall A \in \mathbf{Action}. Q(S, A) \text{ is in the floating point range,} \end{aligned} \quad (21)$$

where the n -times iteration composition of $[u]_{\text{bdl}}$ is defined using Kleisli-composition (an iteration with monadic bind). The property states that if a Q -table Q is reachable via n iterations of the learning loop from an initial configuration (q_t, S_t, A_t) , all values stored in its cells should be in the floating point range. The floating point range test can be implemented by tracking overflow exceptions or by checking the Q -table entries for NaN and infinity, depending on the programming language used. Also the range of the generated Q -tables (the first quantifier) needs to be reasonable (as in regularized, realistic), to avoid values very close to the overflow/underflow limit. What is realistic range of reward values is problem-dependent.

The domain properties listed above, although very simple, constitute real development problems in debugging RL implementations. When implementing applications we often encountered these errors and other RL developers confirmed this to us anecdotally. Such simple tests are able to capture and help diagnose many confusions in practice. It appears that testing for simple properties helps

²Convergence is also hard to prove in a formal system, and even otherwise—the convergence of deep reinforcement learning remains an open problem [Sutton and Barto 2018].

to eliminate the main bugs fast. This is in contrast to complex properties of the algorithm, discussed in Sect. 5.2, which tend to cause fewer problems in practice. Algorithms are typically implemented well, and are run (and thus debugged) many times by many users, but problem formulations are new for every RL task one undertakes and thus prone to new bugs. A good test suite ensuring their basic properties significantly accelerates the task definition process.

Specific Problem Properties. These tests capture idiosyncratic properties of the problem domain; they cannot be formulated without a concrete problem. We include a few cases for our running example to complete the picture, starting with the physics of braking. Equation (22) states that the car’s position never becomes negative, Eq. (23) expresses that a stopped car cannot be moved by braking, and Eq. (24) states that a forward-moving car cannot move backward by braking. All three properties reflect actual bugs experienced by us in our first model of physics for this example:

$$\forall \bar{S}_1, \bar{S}_2 \in \overline{\mathbf{State}}. \forall A \in \mathbf{Action}. (\mathcal{T} \bar{S}_1 A) \bar{S}_2 \geq 0 \rightarrow \bar{S}_2.p \geq 0 \quad (22)$$

$$\forall \bar{S}_1, \bar{S}_2 \in \overline{\mathbf{State}}. \forall A \in \mathbf{Action}. [\bar{S}_1.v = 0 \wedge (\mathcal{T} \bar{S}_1 A) \bar{S}_2 > 0] \rightarrow \bar{S}_1.p = \bar{S}_2.p \quad (23)$$

$$\forall \bar{S}_1, \bar{S}_2 \in \overline{\mathbf{State}}. \forall A \in \mathbf{Action}. [\bar{S}_1.v > 0 \wedge (\mathcal{T} \bar{S}_1 A) \bar{S}_2 > 0] \rightarrow \bar{S}_2.p \geq \bar{S}_1.p \quad (24)$$

The next two tests capture basic intuitions about rewards. Equation (25) says that the further the car is from the obstacle the larger the reward; it is then easier to brake in time, if the velocities are the same. Equation (26) states that lower velocity should yield higher rewards, as it is easier to stop by braking from lower velocities, if in the same position.

$$\forall \bar{S}_1, \bar{S}_2 \in \overline{\mathbf{State}}. \forall A \in \mathbf{Action}. \bar{S}_1.p \leq \bar{S}_2.p \wedge \bar{S}_1.v = \bar{S}_2.v \rightarrow \mathcal{R} \bar{S}_1 A \geq \mathcal{R} \bar{S}_2 A \quad (25)$$

$$\forall \bar{S}_1, \bar{S}_2 \in \overline{\mathbf{State}}. \forall A \in \mathbf{Action}. \bar{S}_1.p = \bar{S}_2.p \wedge \bar{S}_1.v \leq \bar{S}_2.v \rightarrow \mathcal{R} \bar{S}_1 A \geq \mathcal{R} \bar{S}_2 A \quad (26)$$

In summary, problem-specific tests are needed to check that the transition function and the reward function capture the domain characteristics and problem objectives.

5.2 Testing Reinforcement Learning Algorithms

The specification in Sect. 4.3 enables us to derive correctness tests for the learning algorithms. We start with algorithm-independent properties that apply widely to TD learning methods. The simplest tests enforce domain constraints on initialization and update steps. Let Q_0 be the initial value of a Q -table. We test whether Q_0 is defined for all state action pairs in Eq. (27) and that all entries are initialized to zero—a common choice—in Eq. (28).

$$\mathbf{dom} Q_0 = \mathbf{State} \times \mathbf{Action} , \quad (27)$$

$$\forall S_0 \in \mathbf{State}. \forall A_0 \in \mathbf{Action}. Q_0(S_0, A_0) = 0.0 . \quad (28)$$

We establish a domain property for the policy used to select actions. Below, π represents some implementation of a policy; we enforce adherence to the specification of Eq. (4).

$$\forall Q_t \in \mathbf{Q}. \forall S_t \in \mathbf{State}. \forall A_t \in \mathbf{Action}. [(\pi Q_t S_t) A_t > 0] \rightarrow A_t \in \mathbf{Action} . \quad (29)$$

While ensuring domain constraints is relevant, the essence of each policy lies in its probabilistic behavior. In an on-policy ϵ -greedy learning algorithm (the class considered in this paper), the policy selects a random action with probability ϵ and otherwise it greedily follows the highest-value action. This requirement can be cast as a probability distribution test. For every state S_t , the selected action A_t should be distributed according to the distribution $\pi Q_t S_t$. In Eq. (30), we derive a Boolean random variable that tracks selecting the highest value action. We check whether this random

variable is distributed according to a Bernoulli distribution with a parameter derived from Eq. (5).

$$\forall Q_t \in \mathbf{Q}. \forall S_t \in \mathbf{State}. \left[(\pi_{Q_t S_t}) \gg (\lambda_{A_t}. A_t \neq \arg \max_A Q_t(S_t, A)) \right] \sim \text{Bern} \left(\varepsilon \cdot \frac{|\mathbf{Action}| - 1}{|\mathbf{Action}|} \right) \quad (30)$$

In practice during testing, the policy is not available as a symbolic representation of a distribution, but as a sampling algorithm. Therefore, testing the above law requires a statistical test. In our test harness for reinforcement learning, we perform a Bayesian test here. We use a weak prior (a Beta distribution) which encodes that the actual parameter of the Bernoulli distribution is essentially unknown. We collect a sample of executions of the policy and estimate the posterior belief in this parameter given the outcomes of these executions, whether the maximum value action or another action has been selected. This can be calculated analytically for a Beta prior using the conjugate update rule for a Bernoulli likelihood [Kruschke 2014]. We check whether in the obtained posterior distribution over values concentrates 0.94 of the probability mass in a small *credible interval* containing $\varepsilon \cdot (1 - |\mathbf{Action}|^{-1})$ (also known as a *high density interval* [Kruschke 2014]).

As argued in the previous section, the Q -table update is the key element of reinforcement learning—an update is the defining piece of logic for every reinforcement learning algorithm. Second, updates are executed with high frequency. A learning procedure often involves hundreds of thousands of episodes, with many epochs per episode, each epoch containing an update. Third, the success criterion for an update is not easily observable: An update would typically not crash, but “just” produce a wrong floating point number. For this reason, a broken update step can remain unnoticed for a long time, only manifesting in hard to explain subpar results from learning, for instance a slower convergence or higher variance (instability)—both very hard to assess. Consequently, a thorough testing of the update step appears prudent.

The specification of an update defines a function that, given a Q -table Q_t , a system state \bar{S}_t , and an action A_t , returns a multivariate density function over successor Q -tables, target states, and subsequent actions (cf. Sect. 4.3). Let `update` denote the implementation of the update function and $\llbracket u \rrbracket$ the prescribed semantics of this function for the algorithm under test (cf. Sect. 4.3, Eqs. (15) and (16)). The following requirement states that the implementation and the specification produce the same multivariate distribution:

$$\forall Q_t \in \mathbf{Q}. \forall \bar{S}_t \in \overline{\mathbf{State}}. \forall A_t \in \mathbf{Action}. \text{update}(Q_t, \bar{S}_t, A_t) = \llbracket u \rrbracket_{\text{bdl}}(Q_t, \bar{S}_t, A_t), \quad (31)$$

where `update` stands for the implementation of the update function for the concrete learning algorithm. Two aspects of Eq. (31) warrant further discussion: first, how is the equality established (recall that this is an equality on multivariate distributions), second, how the specification $\llbracket u \rrbracket_{\text{bdl}}$ is concretely provided to the test. We handle these points in order.

First, an update produces a multivariate distribution over target states, next actions, and Q -tables. Out of the three components, the target state selection belongs to specific problem tests (Sect. 5.1), the next-action selection follows the policy (see Eq. (30)). Let us focus on the marginal representing the change in the Q -table entry $Q_t(O\bar{S}_t, A_t)$, as the most interesting here. We derive the two random variables representing this update using the implementation and the specification.

$$\begin{aligned} \forall Q_t \in \mathbf{Q}. \forall \bar{S}_t \in \overline{\mathbf{State}}. \forall A_t \in \mathbf{Action}. \\ \left[\text{update}(Q_t, \bar{S}_t, A_t) \gg g \right] &= \left[\llbracket u \rrbracket_{\text{bdl}}(Q_t, \bar{S}_t, A_t) \gg g \right] \\ \text{where } g &= \lambda(Q_{t+1}, \bar{S}_{t+1}, A_{t+1}). Q_{t+1}(O\bar{S}_t, A_t). \end{aligned} \quad (32)$$

Then we test statistically whether the two distributions agree. Assuming that the updates are normally distributed, allows us to compare the two samples with a simple Gaussian model. This test can be implemented in many ways. We perform a Bayesian belief propagation again. We generate

a statistical sample of update results from both the implementation and the specification, compute their point-wise difference, and derive a posterior for the difference using an uninformative normal prior (a conjugate). Then we check whether the posterior for the difference is concentrated around zero with high probability (credibility interval 0.94) [Kruschke 2014].

To perform this test we need the specification u . We use three strategies in our library to provide this specification. We list them in the order of increasing trustworthiness. First, we can *directly implement a specification* as a function in a functional language, directly following the equations of Sect. 4.3 instantiated for a concrete algorithm u , and using a sampling-based implementation of the probability monad (we use our own implementation here). This approach might be useful if testing an implementation in a different style, say a Python implementation. When testing a purely functional implementation of a reinforcement learning algorithm, we end up testing essentially the same code against itself, as the specification and implementation are identical. Second, we can use an implementation of the *update function of another algorithm*. For instance, as discussed in Sect. 4, we can test Q -learning using an update function of SARSA, and a greedy policy π . Testing implementations against each other is a well established tradition—this here is a special case of differential testing for an expected update value.

The third strategy is the most interesting one from the programming language perspective. It is crucial that the equations in Sect. 4.3 are executable, given a sampling implementation of the probability monad. We can thus implement an *interpreter* for *ddl* terms, which given a particular specification term, performs the update accordingly following the model of the update. This allows using the interpreter to test many TD algorithms, the same way as, for example, interpreters are used to test compilers and partial evaluators (where interpreted code is also serving as an oracle for a concrete specialized code). We show fragments of our interpreter in Sect. 6.

6 IMPLEMENTATION

We implemented the above test harness in Scala, and used it to test implementations of SARSA, Q -Learning, and Expected SARSA, along with several example reinforcement learning problems extracted from text books and research papers. The entire project is about 2.3K lines of purely functional Scala 3 code using the Cats and ScalaCheck libraries;³ including the algorithms, all example problems, and tests. Our infrastructure is extensible, it facilitates modular development of reinforcement applications, and continuous testing during the development. Many tests are reusable and can be instantiated for new algorithms and problems. The project is open source and available⁴.

We follow the *property-based testing* (PBT) methodology of Claessen and Hughes [2000] to bridge from the formal specification world to the concrete applications. This methodology is useful both for testing and for development of formal specifications, as it facilitates static type checking and immediate randomized execution of formal properties. This way obvious, and even some non-obvious, specification errors can be detected automatically. At the same time the specifications can be used as tests.

In PBT, program properties are written as executable predicates over input data. Testing a property involves generating inputs automatically, evaluating a predicate on the inputs, and checking whether it holds on all the inputs. For each input data variable we need a test case generator. These are typically associated with the types in PBT, at least in strongly typed languages. The PBT testing libraries provide generators for standard types, and since generators are compositional, it is relatively cheap to add custom ones, as we also show below. PBT testing libraries are available for most mainstream programming languages.

³<https://typelevel.org/cats/>

⁴<https://github.com/itu-square/symsim>

```

1 forAll { (s_t: State) =>                               //  $\forall s_t \in \overline{\text{State}}$ 
2   forAll { (a_t: Action) =>                           //  $\forall a_t \in \text{Action}$ 
3     for s_t1 <- agent.step (s_t) (a_t)                //  $s_{t1} \leftarrow \text{sample}(\mathcal{T}_{s_t} a_t)$ 
4       d1 = agent.observe (s_t1)                       //  $d1 \leftarrow \mathcal{O}_{s_{t1}}$ 
5     yield observableStates.contains (d1)              //  $d1 \in \text{State}$ 
6   } }

```

Fig. 4. Testing that a state which can be reached by a step can also be correctly observed (a domain constraint). The comments in the right column relate the test to Eq. (20)

```

1 val positions = Gen.choose[Double] (0.0, 10.0)
2 val velocities = Gen.choose[Double] (0.0, 10.0)
3 val actions = Gen.oneOf (Car.instances.enumAction.membersAscending)
4 forAll (velocities, positions, actions) { (v, p, a) => //  $\forall S_1 = (v, p) \in \overline{\text{State}}. \forall A \in \text{Action}$ 
5   for s_2 <- Car.step (CarState (v, p)) (a)           //  $S_2 \leftarrow \text{sample}(\mathcal{T}_{S_1} A)$ 
6   yield (v > 0 ==> s_2.p >= p)                       //  $v > 0 \text{ implies } S_{2.p} \geq S_{1.p}$ 
7 }

```

Fig. 5. A car shall not move backwards by braking. The comments in the rightmost column trace to Eq. (24)

```

1 forAll { (q: Q, a_t: Action) =>
2   val trials = for s_t <- agent.initialize
3     a_t1 <- chooseAction (q) (agent.discretize (s_t))
4   yield a_t1 != bestAction (q) (agent.discretize (s_t))
5   val successes = trials.take (episodes).count { _ == true }
6   val failures = episodes - successes
7   val cdfEpsilon = Beta (2 + successes, 2 + failures).cdf (epsilon)
8   cdfEpsilon >= 0.94
9 }

```

Fig. 6. An ϵ -greedy on-policy action choice follows the best action greedily with probability below ϵ , cf. Eq. (30)

We now demonstrate how the properties of Sect. 5 can be tested. Let us begin with the basic domain constraint that if a state can be reached, its observation is a valid observable state, as defined in Eq. (20) (a generic problem property). The test is shown in Fig. 4. Notice that the implementation is very close to the logical description of the property—the comments make the link explicit. The biggest difference is that instead of quantifying over S_{t+1} and checking whether it is reachable with positive density, we sample the value of S_{t+1} from the successor state distribution—if the sample is obtained, we are guaranteed that its density was positive. Notably, in an implementation of the test, the logical quantifiers (`forAll`) are implemented as samplers of test data. Instead of proving the property, the framework evaluates it on several hundreds of input cases.

To support generation of test cases in properties like the one of Fig. 4, the test harness requires that the problem definition includes generators for actions and states. Below we show the default generator for the states of the braking car example using ScalaCheck:

```

1 val genCarState: Gen[CarState] = for
2   v <- Gen.choose (0.0, Double.MaxValue)
3   p <- Gen.choose (0.0, Double.MaxValue)
4   yield CarState (v, p)

```

Figure 5 shows an example of a specific problem test for our running example—checking whether the braking car does not go backwards while moving, as specified in Eq. (24). In this example, we use custom generators (defined in lines 1–3) to control the domains of states, velocities, and actions to come from some specific ranges. The actual property derives a Boolean random variable representing test success like before, except that we also use the ScalaCheck pre-condition operator `==>`.

Finally, Fig. 6 shows a simple implementation of the learning algorithm property specified in Eq. (30), checking that when following an ϵ -greedy policy, a random action is selected with probability ϵ with high belief. In this test, random Q -tables are generated using our custom generator (not shown). As we cannot compare distributions directly, we derive a Boolean random variable `trials`, true whenever an action has not been selected greedily. We count the number of successes in a sample and calculate a posterior for the bias parameter (l. 7). Line 8 checks whether this posterior has almost all values below ϵ (probability mass 0.94).⁵

Finally, the test of the update distribution of Eq. (32) follows a similar statistical design. Here the more interesting aspect is how the update oracle is provided—as mentioned before this can be done by interpreting our term language. Figure 7 shows all the sampling cases of our *bdl* interpreter, implementing the semantic rules of Sect. 4.3. The listing has four parts: the *bdl* abstract syntax implementation (lines 1–7), an estimation step definition (lines 8–18), a sequential Kleisli composition of multiple estimation steps (lines 19–23), and finally a definition of an update step (lines 24–34). The reader can convince themselves that the interpreter indeed follows closely and formally the definitions of the algorithms, and it is fairly easy to establish a one-to-one correspondence (in our implementation the transition \mathcal{T} and reward \mathcal{R} are combined in a single call, e.g. in Line 13). A small term in the *bdl* abstract syntax can be provided to the interpreter to simulate an update of any TD learning algorithm. A similar interpreter can be naturally implemented in any functional programming language.

7 EXPERIMENTAL EVALUATION

We now assess the applicability of the specification and evaluate the effectiveness of the resulting test harness. We discuss how these tests can be reused and as a result reduce the cost of testing in such setups. In particular we address the following research questions:

- RQ1** Is the specification general enough to accommodate diverse reinforcement learning problems?
- RQ2** How effective is the test harness in finding bugs in reinforcement learning problems?
- RQ3** To what extent can generic problem properties be used to reduce the cost of testing for reinforcement learning problems?

To answer **RQ1**, we implement a range of small and medium-sized case studies (first 5 columns in Tbl. 2). We give a brief description of each case study in the following. (Our code and case studies for the experiments in this section are open source and publicly available [Varshosaz et al. 2023])

Case studies. The Unit Agent is the smallest problem in the collection, featuring a single state and a constant reward. We have defined and implemented it to facilitate testing and debugging of learning on a minimal case. Braking Car is the running example of this paper [Vardhan and Sztipanovits 2021]. The somewhat similar Golf learns what club and force to use to hit the target in a minimum number of rounds. The Mountain Car aims to learn how to obtain enough momentum to move up a steep climb [Moore 1990]. Maze requires an agent to find a safe path to a goal location in a 2D maze [Russell and Norvig 2016]. Windy Grid is similar, but includes randomized dislocation of the agent (a wind) [Sutton and Barto 2018]. Cliff walking is another example in which the goal of the agent is to move from a point in a map to another without walking into a cliff area [Sutton and

⁵For a Bayesian statistics aficionado, we remark that this test would be better done using a proper region-of-practical-equivalence (ROPE) for ϵ [Kruschke 2014], but we use a simpler test here to avoid longer discussions of statistics.

```

1 // BDL abstract syntax cf. Eq. (7) p. 10
2 enum Est:
3   case Sample (gamma: Double)
4   case Expectation (gamma: Double)
5 enum Upd:
6   case SampleU, ExpectationU
7 case class Update (est: List[Est], alpha: Double, update: Upd)
8 // Semantics of a sampling estimation step cf. Eq. (9) p. 12
9 def sem (est: Est) (q_t: Q) (s_t: State, a_t: Action, g_t: Double, gamma_t: Double)
10 : Randomized[(State, Action, Double, Double)] = est match
11 case Sample (gamma) =>
12   for (s_tt, r_tt) <- agent.step (s_t) (a_t)
13     os_tt          = agent.observe (s_tt)
14     a_tt           <- vf.chooseAction (epsilon) (q_t) (os_tt)
15     g_tt           = g_t + gamma_t * r_tt
16     gamma_tt       = gamma_t * gamma
17   yield (s_tt, a_tt, g_tt, gamma_tt)
18 case Expectation (gamma) => ...
19 // Kleisli composition of multiple estimation steps cf. Eq. (13) p. 13
20 def sem (ests: List[Est]) (q_t: Q) (s_t: State, a_t: Action, g_t: Double, gamma_t: Double)
21 : Randomized[(State, Action, Double, Double)] =
22   ests.foldM (s_t, a_t, g_t, gamma_t)
23   { case ((s_t, a_t, g_t, gamma_t), e) => sem (e) (q_t) (s_t, a_t, g_t, gamma_t) }
24 // Semantics of a sampling update cf. Eq. (15) p. 14
25 def learningEpoch (bdl: Update, q_t: Q, s_t: State, a_t: Action): Randomized[(Q, State, Action)] =
26   bdl.update match
27   case SampleU =>
28     for (s_tk, a_tk, g_tk, gamma_tk) <- sem (bdl.est) (q_t) (s_t, a_t, 0.0, 1.0)
29       (os_t, os_tk)          = (agent.observe (s_t), agent.observe (s_tk))
30       g_tkk                  = g_tk + gamma_tk * q_t (os_tk, a_tk)
31       q_tt_value             = q_t (os_t, a_t) + bdl.alpha * (g_tkk - q_t (os_t, a_t))
32       q_tt                    = q_t.updated (os_t, a_t, q_tt_value)
33     yield (q_tt, s_tk, a_tk)
34   case ExpectationU => ...

```

Fig. 7. The abstract syntax and interpreter for BDL (only the sampling parts; the expectation parts differ minimally as shown in the equations in Sect. 4.3). Given a term representing an update of a particular reinforcement learning algorithm, this interpreter serves as a probabilistic correctness oracle for testing updates.

Barto 2018]. The k -arm bandit represents a class of state-less agents which admit k actions [Sutton and Barto 2018]. Pumping is a larger industrial case study developed together with a public utility company operating pumping stations for drinking water. The controller must satisfy the public water consumption while the water table does not run dry or get polluted. The pumping case has a larger state space with 92160 observable states.

In response to **RQ1**, we note that the types implementing the concepts of our formal specification facilitate reinforcement learning problems of various scale. The framework allows for learning policies by running different learning algorithms and test the applications. Each application has a test suite and generic tests can be reused between applications. In total, all these case studies required as little as 68 lines of code for the generator implementations. Thus the generators are

Table 2. Experiment results. *K-arm bandit is the class of stateless randomized problems. Unit agent was used in testing learning algorithms but it has no tests itself and was unmutated as it represents an artificial problem.

agent	state space size		episodic gen. size	test cases		mutants			time	mutation
	continuous	observable		[LOC]	[#]	killed	survived	invalid	[s]	score [%]
Pumping	$\mathbb{R} \times \mathbb{R} \times \mathbb{R} \times \mathbb{R}$	92160	✓	17	31	43	16	0	1787	73
Mountain Car	$\mathbb{R} \times \mathbb{R}$	121	×	11	20	24	7	1	34	77
Braking Car	$\mathbb{R} \times \mathbb{R}$	12	✓	9	17	25	0	2	43	100
Windy Grid	-	70	✓	6	17	7	0	1	31	100
Cliff Walking	-	38	✓	6	13	31	1	0	5	97
Simple Maze	-	12	✓	7	15	26	2	0	8	93
Golf	-	10	✓	6	12	9	1	0	7	90
K-arm Bandit*	-	2	✓	3	5	2	0	4	7	100
Unit Agent	-	1	✓	3	-	-	-	-	-	-

not hard or expensive to develop. This highlights the advantage of using the types provided by the specification that enables reuse in implementing reinforcement learning problems.

To answer **RQ2** and **RQ3**, we evaluate the adequacy and the effectiveness of the test suite, using mutation testing [DeMillo et al. 1978; Hamlet 1977]. During mutation testing, variants of a program, called *mutants*, are generated by applying syntactic changes, a class of fault injections. The objective of mutation testing is to measure the ability of a test suite in distinguishing between the output of the original program and its mutants. The program output is often defined as the observable return values, thrown exceptions and program crashes (in the context of unit testing). An outcome of mutation testing is a *mutation score* that represents the ability of the test suite to discriminate between mutants and the correct code [Papadakis et al. 2019]. In the following we explain how the experiments are designed and discuss the results.

Experiment design. We generate mutants of reinforcement learning algorithms, agents and environments using Stryker,⁶ a mutation testing tool that supports several programming languages. Stryker4s is the version of the tool for Scala. We use it to inject faults in the implementation of the reinforcement learning problem and run tests. Stryker supports syntactic transformation rules, *mutators*, for *boolean literals*, *conditional expressions*, *equality operator*, *logical operator*, *method expressions*, *regex* and *string literals*. For each mutant Stryker reports a result: *survived*, means that all tests passed, *killed* means that at least one test failed, *timeout* means that the tests have not terminated before timeout, and *no coverage* means the mutation was not detected by tests—no test failed. Besides these, Stryker also reports invalid mutants e.g., mutants causing runtime errors. Stryker computes a mutation score as:

$$\text{mutation score} = \frac{\#killed + \#timeout}{\#valid \times 100} \quad (33)$$

Mutators for *arithmetic operations* are not supported by Stryker for Scala, so it does not generate mutants of reinforcement learning algorithms (other mutators do not change the code as their target operations do not exist in the algorithm). This is a limitation applied conservatively for Scala as arithmetic operators are function calls and, depending on the code, mutating these operators can lead to mutants that are *stillborn* (i.e., syntactically illegal). To mitigate for this weakness, we manually (using a script) mutate our implementations of reinforcement learning algorithms following the

⁶<https://stryker-mutator.io/>

strategy used by Stryker’s arithmetic mutators for other languages. In general, these changes are among the common changes applied by arithmetic mutators [Papadakis et al. 2019]: swapping plus for minus or vice-versa (m_1) and swapping division for multiplication or vice versa (m_2).

Execution and results. We generate mutants for all case studies described in Tbl. 2, except for the unit agent which is not a real agent but rather an algebraic construct used to test the algorithms (it has a singleton state and a singleton policy space). We use Stryker4s version 0.14.3, which is the latest available version that supports Scala 3. The mutants of the reinforcement learning algorithms are generated semi-automatically using a Python script which applies arithmetic mutators to inject faults in the implementation (other mutators are not applicable for these implementations as explained above). To answer **RQ2**, we run Stryker on each of the case studies individually 10 times. The experiments are performed on a Macbook Pro with 2,3 GHz Quad-Core Intel Core i5 processor and 16 GB RAM.

The results can be found in the six right-most columns of Tbl. 2. The test cases column shows the total number of tests that are run against each mutant. This number includes six generic problem tests (laws) that should hold for all agents. The time reported in the table is the average of the time for Stryker running tests on mutated files in 10 experiment runs. As the mutation score can change due to the randomisation in tests, we report the lowest mutation score in case of changes between runs (only in two cases the mutation score changed between runs).

In response to **RQ2**, we note that the results of evaluation show that in 75% of the cases the mutation score is above 90%. One of the common reasons for surviving mutants is lack of tests for extreme values which is due to the limitation of test data generators. Additionally, there are mutants, for example in the pump case, that are result of applying changes in a function in which the outcome is conditionally selected from overlapping intervals. Hence, writing tests that distinguish changes in the conditions is not feasible.

To address **RQ3**, we re-run the experiments excluding the tests specific to the agents and using only generic tests. These tests express properties (laws) that should hold for any reinforcement learning problem, specifically agent and algorithm in this framework. The results of performing mutation testing using generic problem tests are presented in Fig. 8. In this figure the number of killed and survived mutants are depicted (the number of invalid mutants remain the same as in Tbl. 2).

To answer **RQ3**, we note that the results in Fig. 8 (right) show that in all cases except for pump example the mutation score is above 48%. We observe that mutants are detected by tests that perform sanity checks on the output of the functions responsible for discretizing (observing) the state space and identifying when an agent is in a final state. The results show that these tests can be effective in finding a subset of bugs and providing them can give an advantage to the developer to avoid rewriting the tests which reduces the cost of testing as a result.

Figure 8 (left) illustrates the results of performing mutation testing for SARSA and Expected SARSA algorithms. For each algorithm seven tests (laws) are executed. In SARSA algorithm, five faults are injected including three by m_1 and two by m_2 mutations. All mutants with m_1 changes are killed by tests that are generated based on the specification of SARSA algorithm as explained in Sect. 5. The mutants with m_2 changes are stillborn and are caught by the Scala type checker due to a type mismatch introduced by the change. In Expected SARSA algorithm, six faults are injected including three by m_1 and three by m_2 . Similarly the three mutants with m_1 changes are killed and the three mutants with m_2 changes are caught by the Scala type checker. As an additional observation in relation to **RQ3**, we note that in this case all valid mutants are killed by the tests designed for the learning algorithms while the stillborn mutants are caught by Scala type checker. Hence, considering the valid mutants, a full detection of bugs is achieved by the tests designed for

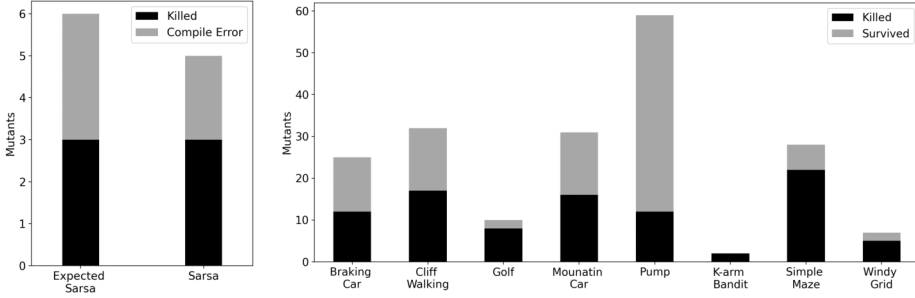


Fig. 8. Mutation results with generic tests for SARSA/Expected SARSA (left) and case studies (right).

testing TD algorithms. These test cases can be parameterized and reused between algorithms and as a result can contribute to reducing the cost of testing the reinforcement learning setups.

8 DISCUSSION

Supporting Other Reinforcement Learning Methods. We focused on testing on-policy TD learning, and exploited an equivalence of updates to test also Q -learning, which is an off-policy algorithm. In general, specifying off-policy learning requires a refinement of *bdl* semantics to support two policies simultaneously. There are no technical obstacles to it, besides maintaining the simplicity of exposition. In Eqs. (9), (10), (15) and (16), one needs to distinguish the policy π used for estimation in the final update from the one that is used to select the next action for execution. Presently, the same policy is used for both. We do intend to implement this in our Scala harness soon.

Support for *Monte Carlo* methods and dynamic programming can be added rather directly. Monte Carlo methods are very similar to n -step TD methods (Tbl. 1). The main difference is that the update only happens at the end of an episode, i.e., when a terminal state is reached. At this point the actual return is known and the state action values can be updated. The sequencing of estimations for episodic systems is a special case of the sequencing of estimations introduced in Eq. (15). To describe the termination of episodes in a final state, without a pre-specified number of steps, a Kleene-star-like variant of the update equation could be introduced in *bdl*. We sketch its definition below. Note that the update-until-termination does not require the final prediction step, as all rewards are known until a termination of a run.

$$\begin{aligned}
 & \llbracket \text{est Update} * \rrbracket_{bdl} S_0 A_0 G_t \gamma_t = \\
 & \lambda \left(Q_t, \bar{S}_t, A_t \right). \text{ if } \mathcal{F} \left(O \bar{S}_t \right) && \text{(at the end of the episode?)} \\
 & \text{ then Det } \left(Q_t \left[(S_0, A_0) \mapsto G_t \right], \bar{S}_t, A_t \right) && \text{(update and stop if in final)} \\
 & \text{ else } \llbracket \text{est} \rrbracket_{est} \left(Q_t \right) \left(\bar{S}_t, A_t, G_t, \gamma_t \right) \gg && \text{(execute the estimation step)} \\
 & \lambda \left(\bar{S}_{t+1}, A_{t+1}, G_{t+1}, \gamma_{t+1} \right). \text{ Det } \left(O \bar{S}_{t+1} \right) \gg && \text{(observe the resulting state)} \\
 & \lambda S_{t+1}. \llbracket \text{est Update} * \rrbracket_{bdl} S_0 A_0 G_{t:t+1} \gamma_{t+1} \left(Q_t, \bar{S}_{t+1}, A_{t+1} \right) && \text{(iterate again)} \quad (34)
 \end{aligned}$$

Here, the state S_0 is the origin state of the Monte Carlo update (initially the same as $O \bar{S}_t$), and A_0 is the initial action (same as the first A_t). The values of G_t and γ_t , the initial accumulated reward and the initial compounded discount factors, should be initialized to zero and one, respectively. The essential difference from Eq. (15) is found in the first three lines. We check whether the agent has arrived at a final state and terminate with an update, if so. Otherwise we iterate again, maintaining

the current accumulated reward and discount factor. The above can be defined as the least fixed point of a Scott-continuous operator (the core of the equation above) in a suitable domain. These are standard steps though, so we omit the details, in favour of a program-like recursive presentation.

For dynamic-programming-based methods (DP), the learner requires an explicit probabilistic model of the environment. Hence, in the update rules one should use an MDP model instead of sampling to obtain the successor states, rewards and their probabilities. The DP algorithms commonly use a state value function $V \in \mathbf{State} \rightarrow \mathbb{R}$, instead of a Q -table. A state value function represents the total amount of reward that an agent can accumulate over the episodes starting in that state. The value of each state is updated iteratively, by accumulating the immediate reward and value of all possible successor states, i.e., the states that are reachable in one step. To formalize this, the same structure of the equations can be kept, but with a unary (not binary) value function. Moreover, one can eliminate the probability monad, as the dynamic programming update is deterministic (or uses Dirac distributions if maintaining the same structure with probabilistic algorithms is desirable).

Testing the TD(λ) algorithm requires modeling *eligibility traces* [Sutton and Barto 2018] in the *bdl* semantics and the interpreter. The eligibility traces are a mechanism which allows to execute multiple TD-updates simultaneously, similarly to pipe-lining in CPUs. Once eligibility traces are handled in the *bdl* interpreter it will graduate from being a test oracle to being a general TD learning algorithm, parameterized by an updated specification, which is interesting for future work.

It is also interesting to generalize to different value function representations, to allow *approximate learning* with neural networks. As *bdl* abstracts from the representation of the Q -table, a neural network can in principle be used; structurally the specification should not change much. For example Actor-critic methods still follow the pattern of TD(0), SARSA, and Q-Learning [Sutton and Barto 2018]. As action selection is already probabilistic in our model (ϵ -exploration), for continuous actions we just need to switch from sampling a discrete distribution to a continuous one. The challenge lies in formalizing gradient-based updates on neural networks, which are much more sophisticated than simple assignments to a table cell, and require non-trivial further work; similarly for popular newer policy iteration methods like PPO [Schulman et al. 2017]. To address this limitation, we plan to investigate using methods taken from the differentiable programming languages field in the future.

Software Testing with Statistics. As machine learning gains popularity, we face more and more programs with probabilistic correctness requirements that have to be tested statistically. Reinforcement learning is one such example. This requires development of new experience in software testing. The tests we used are certainly simple, even simplistic, as we prioritized efficiency and simplicity over strength. Assuming conjugate priors is too strong in general [McElreath 2020]. Monte-Carlo posterior estimation is a possible alternative—unfortunately this would slow the tests down by several orders of magnitude. The performance problem is exacerbated, if more precise tests are used. We tested for equivalence of expectations, but the variance is also relevant for reinforcement learning, especially that learning algorithms may agree on the expectation but differ on variance. Comparing variance would help to kill mutants that maintain the same expectation at the cost of slower convergence, which manifests in a higher variance of reward estimations. Unfortunately, it is computationally much harder to estimate variance as precisely as the mean. In general, the credibility (strength) of statistical tests correlates with their computational cost. More samples are required for stronger conclusions.

Statistical tests are by their very nature flaky [Luo et al. 2014]. They can fail occasionally, disturbing continuous integration, or mutation scoring like in our experiments (mutation runs the tests many times, increasing the chance of failure occurring). There is an inherent tension between flakiness of tests and their ability to kill mutants and to detect bugs. One can set weaker thresholds

for credibility of a test, which will make it fail less or practically never, but it will also make it accept larger deviations from the spec and miss more bugs. One can strengthen the test to kill more mutants, but this will increase the frequency at which the test will fail randomly on the correct code. This trade-off makes it very difficult to set the hyper parameters (thresholds) for statistical tests. We have done this by trial-and-error, but more systematic methods are needed.

9 RELATED WORK

A large body of recent work studies the use of RL and deep RL to improve testing processes. Such techniques [Liu et al. 2022; Romdhana et al. 2022; Su et al. 2022; Tufano et al. 2022; Türker et al. 2021; Zhang et al. 2021; Zheng et al. 2021] are applied for testing a variety of systems (e.g., video games, web applications, and cyber physical systems). In contrast, we are concerned with the opposite problem—applying testing to reinforcement learning.

Many authors focus on testing Machine Learning (ML) algorithms broadly. For example, optimizing stochastic regression tests in ML projects [Dutta et al. 2021], augmenting a deep learning test set to increase its mutation score [Riccio et al. 2021], testing bias in ML software [Chakraborty et al. 2021], pointwise robustness in deep neural networks [Wu et al. 2020], concolic testing for deep neural networks [Sun et al. 2018], formally verifying safety properties of deep reinforcement learning system [Ivanov et al. 2020]. The present paper does not contribute to testing neural networks (even if they are a representation of value functions used in RL) but addresses testing the correctness of RL problems and algorithm implementations by providing specifications for their basic blocks.

In the field of RL, a key topic of focus in prior work is the reliability assessment of a trained agent. Adversarial ML is used to understand the behavior of models and algorithms in the presence of failure inducing contexts and behaviors. Huang et al. investigate impact of the effectiveness of adversarial examples on a deep RL algorithm [Huang et al. 2017]. Lin et al. introduce strategically timed attacks on RL agents [Lin et al. 2017]. Amirloo et al. propose to guide adversarial sampling by a predictor trained along with the agent to predict the probability of failure [Abolfathi et al. 2021]. Vardhan and Sztipanovits use a generative model to find failure scenarios [Vardhan and Sztipanovits 2021]. Ruderman et al. study the worst-case analysis to detect the directions in which agents may have failed to generalize while learning the policy [Ruderman et al. 2019]. To overcome the small adversarial perturbations on the agent’s inputs, Oikarinen et al. propose to train RL agents with improved robustness against l_p -norm bounded adversarial attacks [Oikarinen et al. 2021]. In this line of work, the focus is on optimality and generality of the obtained policies. However, they side step the problem of correctness of the RL implementations used to learn the policies. In contrast, we follow a modular testing strategy, not unlike unit testing, for low-level properties of individual elements in RL applications, hoping that this exposes problems early and close to their origins. Furthermore, this way we also hope to inspire work on formal verification of RL, as properties follow the style more often used in verification.

Another line of work, surveyed by García and Fernández [2015], addresses the synthesis and update of models that preserve safety properties by either modifying the optimality criterion or the exploration process. In particular, safety properties can be ensured for RL algorithms by incorporating a *shielding* mechanism that prevents the algorithm from taking actions that could lead to unsafe outcomes, by means of techniques such as control barrier functions [Alshiekh et al. 2018], logically constrained learning [Hasanbeig et al. 2018] and safe permissive schedulers [Junges et al. 2016]. Justified speculative control proves the shields safe by means of deductive verification [Fulton and Platzer 2018, 2019]. Jansen et al. consider probabilistic shielding to ensure the safety of RL agents [Jansen et al. 2020]. Tappler et al. combine automata learning and shielding into a method that enables RL agents to acquire a model of the environment and enforces safety constraints [Tappler et al. 2022b]. Although both shielding and testing are concerned with the correctness of RL, shielding

techniques are complementary to testing: they aim at enforcing correctness by constraining the learning process rather than at exposing bugs. Whereas shielding techniques mainly focus on constraining the actions of reinforcement learning problems, our work addresses property-based testing both for reinforcement learning problems and for the learning algorithms themselves.

Other software engineering methods have been applied to test and verify RL agents, including black-box fuzzing [Pang et al. 2022], search-based testing [Tappler et al. 2022a], mutation testing [Lu et al. 2021], deductive reasoning [Déletang et al. 2021], using machine learning models and genetic algorithms to test policies [Zolfagharian et al. 2023]. Alur et al. have studied the formal specifications of RL tasks [Jothimurugan et al. 2019] and of multi-agent RL problems [Jothimurugan et al. 2022], transforming task specifications in RL [Alur et al. 2022], RL algorithms in abstract decision processes [Jothimurugan et al. 2021b], and compositional RL from logical specifications [Jothimurugan et al. 2021a] are other cases that software engineering to RL. In contrast, our work is concerned with providing a direct formal specification of correctness for RL problems and algorithms themselves, as opposed to the policies that they output. We develop a property-based test harness for all elements of a RL problem and algorithm. We are not aware of similar formal definitions of RL problems that are precise and self-contained and nor of prior uses of property-based testing for RL.

10 CONCLUSION

We have presented a formal specification of the different components of reinforcement learning, targeting temporal difference methods. The formalization enables us to derive an associated test harness, reusable across a large class of reinforcement learning applications based on Q -learning, SARSA, etc. Somewhat unusually for the reinforcement learning context, the testing harness embeds an interpreter of formal models for update equations as an oracle (a practice well recognized in programming language engineering). The test harness has been evaluated on several algorithms and agents using mutation testing, showing good baseline effectiveness. Our implementations (including all algorithms, laws, case study examples, and test scripts) is available as an open source project.

Formal verification and testing of learning algorithms is a fast growing research field. As specification is the first step towards verification, this paper may help researchers working on verification of probabilistic programs to tackle reinforcement learning. We also hope that the presentation of tests in Sections 5 and 6 can serve as a tutorial on how and what to test when developing a RL system.

ACKNOWLEDGMENTS

Partially funded by DIREC (Digital Research Centre Denmark), a collaboration between the eight Danish universities and the Alexandra Institute supported by the Innovation Fund Denmark.

DATA AVAILABILITY STATEMENT

The experiment data and the scripts to reproduce them are available at Varshosaz et al. [2023]. The implementation of symsim and its test suite are an open source project available at <https://github.com/itu-square/symsim>.

A APPENDIX

In order to facilitate translating our specification to formal verification systems and to other testing frameworks, we include an integrated definition below.

A.1 A Formal Specification of Reinforcement Learning

A.1.1 Reinforcement Learning Problems. In the following we add the definitions for formalising a reinforcement learning problem.

Definition A.1. A Reinforcement Learning Problem is represented by a tuple $(\overline{\mathbf{State}}, \overline{\mathbf{State}}_0, \mathbf{Action}, \mathbf{State}, O, \mathcal{T}, \mathcal{R}, \mathcal{F})$ where:

- r**₁: $\overline{\mathbf{State}}$ is a possibly infinite set of states of the environment and the agent combined,
- r**₂: $\overline{\mathbf{State}}_0 \in \text{pdf } \overline{\mathbf{State}}$ is a density function defining probability for initial states,
- r**₃: \mathbf{Action} is a finite set of actions that an agent can take,
- r**₄: \mathbf{State} is a finite set of observable states,
- r**₅: $O \in \overline{\mathbf{State}} \rightarrow \mathbf{State}$ is a total observation function,
- r**₆: $\mathcal{T} \in \overline{\mathbf{State}} \rightarrow \mathbf{Action} \rightarrow \text{pdf } \overline{\mathbf{State}}$ is the transition probability function,
- r**₇: $\mathcal{R} \in \overline{\mathbf{State}} \rightarrow \mathbf{Action} \rightarrow \mathbb{R}$ is the reward function, and
- r**₈: $\mathcal{F} \in \overline{\mathbf{State}} \rightarrow \{0, 1\}$ is a predicate defining which observable states are final for a training episode. Initial states are not final, i.e., if $\overline{\mathbf{State}}_0(\bar{S}) > 0$ then not $\mathcal{F}(O \bar{S})$.

Definition A.2. A RL problem is *episodic* iff every run from an initial state eventually reaches some final state \bar{S} , so $\mathcal{F}(O \bar{S}) = 1$. Otherwise the problem is *non-episodic*.

Policy π is a probability function that, given a value function, represents the distribution of plausible actions in each state.

$$\pi \in \mathbf{Q} \rightarrow \overline{\mathbf{State}} \rightarrow \text{pmf } \mathbf{Action} \quad (\text{RL } 3)$$

Action selection based on an ε -greedy algorithm is defined as:

$$(\pi Q_t S_t) A_t = \begin{cases} 1 - \varepsilon + \varepsilon \cdot |\mathbf{Action}|^{-1} & A_t = \arg \max_A Q(S_t, A) \\ \varepsilon \cdot |\mathbf{Action}|^{-1} & \text{otherwise} \end{cases} \quad (\text{RL } 4)$$

A.1.2 Temporal Difference Learning Algorithms. The general update rule for a TD prediction method is:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(G_t - Q(S_t, A_t)) . \quad (\text{RL } 5)$$

A.1.3 Formalising Temporal Difference Algorithms. The proposed abstract syntax for backup diagrams is generated by the following grammar:

$$\begin{aligned} est & ::= \text{sample}^Y \mid \text{expectation}^Y \\ bdl & ::= est^+ \text{Update}^\alpha(\text{sample} \mid \text{expectation}) . \end{aligned} \quad (\text{RL } 6)$$

The semantic domain of an estimation step is:

$$\llbracket \cdot \rrbracket_{est} \in \mathbf{Q} \rightarrow \overline{\mathbf{State}} \times \mathbf{Action} \times \mathbb{R} \times \mathbb{R} \rightarrow \text{pmf } (\overline{\mathbf{State}} \times \mathbf{Action} \times \mathbb{R} \times \mathbb{R}) \quad (\text{RL } 7)$$

The semantic function for a sampling estimation step is:

$$\begin{aligned} \llbracket \text{sample}^Y \rrbracket_{est} Q_t = & \\ & \lambda(\bar{S}_t, A_t, G_t, \gamma_t). \mathcal{T} \bar{S}_t A_t \gg= \\ & \lambda \bar{S}_{t+1}. \text{Det} \left(\mathcal{R} \bar{S}_{t+1} A_t \right) \gg= \\ & \lambda R_{t+1}. \text{Det} \left(O \bar{S}_{t+1} \right) \gg= \\ & \lambda S_{t+1}. \pi Q_t S_{t+1} \gg= \\ & \lambda A_{t+1}. \text{Det} (G_{:t} + \gamma_t R_{t+1}) \gg= \\ & \lambda G_{:t+1}. \text{Det} (\gamma_t \cdot \gamma) \gg= \\ & \lambda \gamma_{t+1}. \text{Det} \left(\bar{S}_{t+1}, A_{t+1}, G_{:t+1}, \gamma_{t+1} \right) \end{aligned} \quad (\text{RL } 8)$$

The semantic function for an expectation estimation step is:

$$\begin{aligned}
\llbracket \text{expectation}^y \rrbracket_{est} Q_t = & \\
& \lambda(\bar{S}_t, A_t, G_{:t}, \gamma_t). \mathcal{T} \bar{S}_t A_t \gg \\
& \lambda \bar{S}_{t+1}. \text{Det} \left(\mathcal{R} \bar{S}_{t+1} A_t \right) \gg \\
& \lambda R_{t+1}. \text{Det}(\mathcal{O} \bar{S}_{t+1}) \gg \\
& \lambda S_{t+1}. \pi Q_t S_{t+1} \gg \\
& \lambda A_{t+1}. \text{Det} \left(G_{:t} + \gamma_t \left[R_{t+1} + \sum_{A \neq A_{t+1}} (\pi Q S_{t+1}) A \cdot Q(S_{t+1}, A) \right] \right) \gg \\
& \lambda G_{:t+1}. \text{Det} \left(\gamma_t \cdot \gamma \cdot (\pi Q S_{t+1}) A_{t+1} \right) \gg \\
& \lambda \gamma_{t+1}. \text{Det} \left(\bar{S}_{t+1}, A_{t+1}, G_{:t+1}, \gamma_{t+1} \right) \tag{RL 9}
\end{aligned}$$

The semantic domain for a sequence of composed estimation steps:

$$\llbracket est_1 \cdots est_k \rrbracket_{est^+} \in \mathbf{Q} \rightarrow \overline{\mathbf{State}} \times \mathbf{Action} \times \mathbb{R} \times \mathbb{R} \rightarrow \text{pmf}(\overline{\mathbf{State}} \times \mathbf{Action} \times \mathbb{R} \times \mathbb{R}) \tag{RL 10}$$

The semantics of composed estimation steps is defined as:

$$\begin{aligned}
\llbracket est_1 \cdots est_k \rrbracket_{est^+} Q_t = & \\
& \lambda(\bar{S}_t, A_t, G_{:t}, \gamma_t). \llbracket est_1 \rrbracket_{est} (Q_t) (\bar{S}_t, A_t, G_{:t}, \gamma_t) \gg \\
& \lambda(\bar{S}_{t+1}, A_{t+1}, G_{:t+1}, \gamma_{t+1}). \llbracket est_2 \rrbracket_{est} (Q_t) (\bar{S}_{t+1}, A_{t+1}, G_{:t+1}, \gamma_{t+1}) \gg \\
& \vdots \\
& \lambda(\bar{S}_{t+k-1}, A_{t+k-1}, G_{:t+k-1}, \gamma_{t+k-1}). \\
& \llbracket est_k \rrbracket_{est} (Q_t) (\bar{S}_{t+k-1}, A_{t+k-1}, G_{:t+k-1}, \gamma_{t+k-1}) \tag{RL 11}
\end{aligned}$$

The compact version of the above composition is:

$$\llbracket est_1 \cdots est_k \rrbracket_{est^+} Q_t = \llbracket est_1 \rrbracket_{est} Q_t \gg \llbracket est_2 \rrbracket_{est} Q_t \gg \cdots \gg \llbracket est_k \rrbracket_{est} Q_t \tag{RL 12}$$

The semantic domain of an update step is:

$$\llbracket est^k \text{ Update}^\alpha (\text{sample} \mid \text{expectation}) \rrbracket_{bdl} \in \mathbf{Q} \times \overline{\mathbf{State}} \times \mathbf{Action} \rightarrow \text{pmf}(\mathbf{Q} \times \overline{\mathbf{State}} \times \mathbf{Action}) \tag{RL 13}$$

The semantic function for an update step finalized with a sample is:

$$\begin{aligned}
\llbracket est^k \text{ Update}^\alpha \text{ sample} \rrbracket_{bdl} = & \\
& \lambda(Q_t, \bar{S}_t, A_t). \llbracket est^k \rrbracket_{est^+} (Q_t) (\bar{S}_t, A_t, 0, 1) \gg \\
& \lambda(\bar{S}_{t+k}, A_{t+k}, G_{:t+k}, \gamma_{t+k}). \text{Det} \left(\mathcal{O} \bar{S}_t, \mathcal{O} \bar{S}_{t+k} \right) \gg \\
& \lambda(S_t, S_{t+k}). \text{Det} \left(G_{:t+k} + \gamma_{t+k} \cdot Q(S_{t+k}, A_{t+k}) \right) \gg \\
& \lambda G_{:t+k+1}. \text{Det} \left(Q_t(S_t, A_t) \mapsto Q(S_t, A_t) + \alpha [G_{:t+k+1} - Q(S_t, A_t)] \right) \gg \\
& \lambda Q_{t+1}. \text{Det}(Q_{t+1}, \bar{S}_{t+k}, A_{t+k}) \tag{RL 14}
\end{aligned}$$

The semantic function for an update step finalized with an expectation is:

$$\begin{aligned}
& \llbracket est^k \text{ Update}^\alpha \text{ expectation} \rrbracket_{bdl} = \\
& \lambda(Q_t, \bar{S}_t, A_t). \llbracket est^k \rrbracket_{est^t} (Q_t) (\bar{S}_t, A_t, 0, 1) \gg \\
& \lambda(\bar{S}_{t+k}, A_{t+k}, G_{t:t+k}, Y_{t+k}). \text{Det} \left(\mathcal{O} \bar{S}_t, \mathcal{O} \bar{S}_{t+k} \right) \gg \\
& \lambda(S_t, S_{t+k}). \text{Det} (G_{t:t+k} + Y_{t+k} \sum_A (\pi S_{t+k+1} A) \cdot Q_t(S_{t+k}, A)) \gg \\
& \lambda_{G_{t:t+k+1}}. \text{Det} (Q_t(S_t, A_t) \mapsto Q_t(S_t, A_t) + \alpha [G_{t:t+k+1} - Q_t(S_t, A_t)]) \gg \\
& \lambda_{Q_{t+1}}. \text{Det}(Q_{t+1}, \bar{S}_{t+k}, A_{t+k})
\end{aligned} \tag{RL 15}$$

A.2 Selected Tests for Reinforcement Learning

A.2.1 *Testing RL Problems.* Every system state should have a translation into an observable state.

$$\forall \bar{S} \in \overline{\mathbf{State}}. \mathcal{O} \bar{S} \in \mathbf{State}, \tag{RL 16}$$

$$\forall \bar{S}_0 \in \overline{\mathbf{State}}. \mathbf{State}_0 \bar{S}_0 > 0 \rightarrow \mathcal{O} \bar{S}_0 \in \mathbf{State}. \tag{RL 17}$$

An initial state is never a final state:

$$\forall \bar{S}_0 \in \overline{\mathbf{State}}. \mathbf{State}_0 \bar{S}_0 > 0 \rightarrow \neg \mathcal{F} (\mathcal{O} \bar{S}_0). \tag{RL 18}$$

Starting in an observable state, taking an action, an agent ends in an observable state.

$$\forall \bar{S}_t \in \overline{\mathbf{State}}. \forall A_t \in \mathbf{Action}. \mathcal{O} (\mathcal{T} \bar{S}_t A_t) \in \mathbf{State} \tag{RL 19}$$

A.2.2 *Testing RL Algorithms.* An initial Q -table Q_0 is defined for all state action pairs and all entries are initialized to zero.

$$\mathbf{dom} Q_0 = \mathbf{State} \times \mathbf{Action} \tag{RL 20}$$

$$\forall S_0 \in \mathbf{State}. \forall A_0 \in \mathbf{Action}. Q_0 (S_0, A_0) = 0.0 \tag{RL 21}$$

A policy π includes valid actions.

$$\forall Q_t \in \mathbf{Q}. \forall S_t \in \mathbf{State}. \forall A_t \in \mathbf{Action}. [(\pi Q_t S_t) A_t > 0] \rightarrow A_t \in \mathbf{Action} \tag{RL 22}$$

The probability distribution test for the ε -greedy algorithm.

$$\begin{aligned}
& \forall Q_t \in \mathbf{Q}. \forall S_t \in \mathbf{State}. \\
& \left[(\pi Q_t S_t) \gg (\lambda_{A_t}. A_t \neq \arg \max_A Q_t (S_t, A)) \right] = \text{Bern} \left(\varepsilon \cdot \frac{|\mathbf{Action}| - 1}{|\mathbf{Action}|} \right)
\end{aligned} \tag{RL 23}$$

The implementation and the specification produce the same multivariate distribution.

$$\begin{aligned}
& \forall Q_t \in \mathbf{Q}. \forall \bar{S}_t \in \overline{\mathbf{State}}. \forall A_t \in \mathbf{Action}. \\
& \text{update} (Q_t, \bar{S}_t, A_t) = \llbracket [u] \rrbracket_{bdl} (Q_t, \bar{S}_t, A_t)
\end{aligned} \tag{RL 24}$$

The updates are normally distributed as they are affected by the noise in many executions of the agent.

$$\begin{aligned}
& \forall Q_t \in \mathbf{Q}. \forall \bar{S}_t \in \overline{\mathbf{State}}. \forall A_t \in \mathbf{Action}. \\
& \left[\text{update} (Q_t, \bar{S}_t, A_t) \gg g \right] = \left[\llbracket [u] \rrbracket_{bdl} (Q_t, \bar{S}_t, A_t) \gg g \right] \\
& \text{where } g = \lambda(Q_{t+1}, \bar{S}_{t+1}, A_{t+1}). Q_{t+1} (\mathcal{O} \bar{S}_t, A_t)
\end{aligned} \tag{RL 25}$$

REFERENCES

- Elmira Amirloo Abolfathi, Jun Luo, Peyman Yadmellat, and Kasra Rezaee. 2021. CoachNet: An Adversarial Sampling Approach for Reinforcement Learning. In *NeurIPS2019 Workshop on Safety and Robustness in Decision Making*. arXiv. <https://doi.org/10.48550/ARXIV.2101.02649>
- Mohammed Alshiekh, Roderick Bloem, Rüdiger Ehlers, Bettina Könighofer, Scott Niekum, and Ufuk Topcu. 2018. Safe reinforcement learning via shielding. In *Proc. AAAI Conference on Artificial Intelligence*, Vol. 32. AAAI Press. <https://doi.org/10.1609/aaai.v32i1.11797>
- Rajeev Alur, Suguman Bansal, Osbert Bastani, and Kishor Jothimurugan. 2022. A Framework for Transforming Specifications in Reinforcement Learning. In *Principles of Systems Design: Essays Dedicated to Thomas A. Henzinger on the Occasion of His 60th Birthday (Lecture Notes in Computer Science, Vol. 13660)*, Jean-François Raskin, Krishnendu Chatterjee, Laurent Doyen, and Rupak Majumdar (Eds.). Springer. https://doi.org/10.1007/978-3-031-22337-2_29
- Joymallya Chakraborty, Suvodeep Majumder, and Tim Menzies. 2021. Bias in Machine Learning Software: Why? How? What to Do?. In *Proc. 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (Athens, Greece) (ESEC/FSE 2021)*. ACM Press, 429--440. <https://doi.org/10.1145/3468264.3468537>
- Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *5th ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*. ACM Press, 268–279. <https://doi.org/10.1145/357766.351266>
- Kristopher De Asis, J. Fernando Hernandez-Garcia, G. Zacharias Holland, and Richard S. Sutton. 2018. Multi-Step Reinforcement Learning: A Unifying Algorithm. In *Proc. 32nd AAAI Conference on Artificial Intelligence and Thirtieth Innovative Applications of Artificial Intelligence Conference and Eighth AAAI Symposium on Educational Advances in Artificial Intelligence (AAAI'18/IAAI'18/EAAI'18)*. AAAI Press, Article 354, 8 pages. <https://doi.org/10.1609/aaai.v32i1.11631>
- Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. 1978. Hints on test data selection: Help for the practicing programmer. *Computer* 11, 4 (1978), 34–41. <https://doi.org/doi:10.1109/C-M.1978.218136>
- Saikat Dutta, Jeeva Selvam, Aryaman Jain, and Sasa Misailovic. 2021. TERA: Optimizing Stochastic Regression Tests in Machine Learning Projects. In *Proc. 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (Virtual, Denmark) (ISSTA 2021)*. ACM Press, 413--426. <https://doi.org/10.1145/3460319.3464844>
- Grégoire Déletang, Jordi Grau-Moya, Miljan Martić, Tim Genewein, Tom McGrath, Vladimir Mikulík, Markus Kunesch, Shane Legg, and Pedro A. Ortega. 2021. Causal Analysis of Agent Behavior for AI Safety. arXiv. <https://doi.org/10.48550/ARXIV.2103.03938>
- Nathan Fulton and André Platzer. 2018. Safe Reinforcement Learning via Formal Methods: Toward Safe Control Through Proof and Learning. In *Proc. Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18)*, Sheila A. McIlraith and Kilian Q. Weinberger (Eds.). AAAI Press, 6485–6492. <https://doi.org/10.1609/aaai.v32i1.12107>
- Nathan Fulton and André Platzer. 2019. Verifiably safe off-model reinforcement learning. In *Proc. 25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2019) (Lecture Notes in Computer Science, Vol. 11427)*. Springer, 413–430. https://doi.org/10.1007/978-3-030-17462-0_28
- Javier Garcia and Fernando Fernández. 2015. A comprehensive survey on safe reinforcement learning. *J. Mach. Learn. Res.* 16 (2015), 1437–1480. <https://dl.acm.org/doi/10.5555/2789272.2886795>
- Andrew Gelman, John B Carlin, Hal S Stern, David B Dunson, Aki Vehtari, and Donald B Rubin. 2013. *Bayesian data analysis*. CRC press.
- Andrew Gelman and Cosma Rohilla Shalizi. 2013. Philosophy and the practice of Bayesian statistics. *Brit. J. Math. Statist. Psych.* 66, 1 (2013), 8–38. <https://doi.org/10.1111/j.2044-8317.2011.02037.x>
- Richard G. Hamlet. 1977. Testing Programs with the Aid of a Compiler. *IEEE Transactions on Software Engineering* 3, 4 (1977), 279–290. <https://doi.org/10.1109/TSE.1977.231145>
- Mohammadhosein Hasanbeig, Alessandro Abate, and Daniel Kroening. 2018. Logically-Correct Reinforcement Learning. *CoRR* abs/1801.08099 (2018). arXiv:1801.08099 <http://arxiv.org/abs/1801.08099>
- Sandy Huang, Nicolas Papernot, Ian Goodfellow, Yan Duan, and Pieter Abbeel. 2017. Adversarial Attacks on Neural Network Policies. arXiv. <https://doi.org/10.48550/ARXIV.1702.02284>
- Radoslav Ivanov, Taylor J Carpenter, James Weimer, Rajeev Alur, George J Pappas, and Insup Lee. 2020. Case study: verifying the safety of an autonomous racing car with a neural network controller. In *Proc. 23rd International Conference on Hybrid Systems: Computation and Control*. 1–7. <https://doi.org/10.1145/3365365.3382216>
- Nils Jansen, Bettina Könighofer, Sebastian Junges, Alex Serban, and Roderick Bloem. 2020. Safe Reinforcement Learning Using Probabilistic Shields. In *Proc. 31st International Conference on Concurrency Theory (CONCUR 2020) (LIPIcs, Vol. 171)*, Igor Konnov and Laura Kovács (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 3:1–3:16. <https://doi.org/10.4230/LIPIcs.CONCUR.2020.3>
- Kishor Jothimurugan, Rajeev Alur, and Osbert Bastani. 2019. A Composable Specification Language for Reinforcement Learning Tasks. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc.

- Kishor Jothimurugan, Suguman Bansal, Osbert Bastani, and Rajeev Alur. 2021a. Compositional Reinforcement Learning from Logical Specifications. In *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan (Eds.), Vol. 34. Curran Associates, Inc., 10026–10039.
- Kishor Jothimurugan, Suguman Bansal, Osbert Bastani, and Rajeev Alur. 2022. Specification-Guided Learning of Nash Equilibria with High Social Welfare. In *Proc. 34th International Conference on Computer Aided Verification (CAV 2022) (Lecture Notes in Computer Science, Vol. 13372)*, Sharon Shoham and Yakir Vizel (Eds.). Springer, 343–363. https://doi.org/10.1007/978-3-031-13188-2_17
- Kishor Jothimurugan, Osbert Bastani, and Rajeev Alur. 2021b. Abstract Value Iteration for Hierarchical Reinforcement Learning. In *Proc. 24th International Conference on Artificial Intelligence and Statistics (Proceedings of Machine Learning Research, Vol. 130)*, Arindam Banerjee and Kenji Fukumizu (Eds.). PMLR, 1162–1170.
- Sebastian Junges, Nils Jansen, Christian Dehnert, Ufuk Topcu, and Joost-Pieter Katoen. 2016. Safety-Constrained Reinforcement Learning for MDPs. In *Proc. 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2016) (Lecture Notes in Computer Science, Vol. 9636)*, Marsha Chechik and Jean-François Raskin (Eds.). Springer, 130–146. https://doi.org/10.1007/978-3-662-49674-9_8
- Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. 1996. Reinforcement Learning: A Survey. *J. Artif. Intell. Res.* 4 (1996), 237–285. <https://doi.org/10.1613/jair.301>
- John Kruschke. 2014. *Doing Bayesian data analysis: A tutorial with R, JAGS, and Stan*. Academic Press.
- Yen-Chen Lin, Zhang-Wei Hong, Yuan-Hong Liao, Meng-Li Shih, Ming-Yu Liu, and Min Sun. 2017. Tactics of Adversarial Attack on Deep Reinforcement Learning Agents. In *Proc. 26th International Joint Conference on Artificial Intelligence (IJCAI'17)*. AAAI Press, 3756–3762. <https://doi.org/10.24963/ijcai.2017/525>
- Junrui Liu, Yanju Chen, Bryan Tan, Isil Dillig, and Yu Feng. 2022. Learning Contract Invariants Using Reinforcement Learning. In *Proc. 37th IEEE/ACM International Conference on Automated Software Engineering, (ASE 2022)*. ACM Press, 63:1–63:11. <https://doi.org/10.1145/3551349.3556962>
- Yuteng Lu, Weidi Sun, and Meng Sun. 2021. Mutation Testing of Reinforcement Learning Systems. In *Proc. 7th International Symposium on Dependable Software Engineering: Theories, Tools, and Applications (SETTA 2021) (Lecture Notes in Computer Science, Vol. 13071)*, Shengchao Qin, Jim Woodcock, and Wenhui Zhang (Eds.). Springer, 143–160. https://doi.org/10.1007/978-3-030-91265-9_8
- Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *Proc. 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE-22)*, Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey (Eds.). ACM Press, 643–653. <https://doi.org/10.1145/2635868.2635920>
- Richard McElreath. 2020. *Statistical Rethinking: A Bayesian Course with Examples in R and Stan* (2nd ed.). CRC Press.
- Andrew W. Moore. 1990. Efficient memory-based learning for robot control. Ph.D. Thesis, University of Cambridge.
- Tuomas Oikarinen, Wang Zhang, Alexandre Megretski, Luca Daniel, and Tsui-Wei Weng. 2021. Robust Deep Reinforcement Learning through Adversarial Loss. In *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan (Eds.), Vol. 34. Curran Associates, 26156–26167.
- Qi Pang, Yuanyuan Yuan, and Shu Wang. 2022. MDPFuzz: Testing Models Solving Markov Decision Processes. In *Proc. 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2022)*. ACM Press, 378–390. <https://doi.org/10.1145/3533767.3534388>
- Mike Papadakis, Marinou Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Chapter Six - Mutation Testing Advances: An Analysis and Survey. *Advances in Computers*, Vol. 112. Elsevier, 275–378. <https://doi.org/10.1016/bs.adcom.2018.03.015>
- Norman Ramsey and Avi Pfeffer. 2002. Stochastic lambda calculus and monads of probability distributions. In *Proc. 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2002)*, John Launchbury and John C. Mitchell (Eds.). ACM Press, 154–165. <https://doi.org/10.1145/503272.503288>
- Vincenzo Riccio, Nargiz Humbatova, Gunel Jahangirova, and Paolo Tonella. 2021. DeepMetis: Augmenting a Deep Learning Test Set to Increase its Mutation Score. In *36th IEEE/ACM International Conference on Automated Software Engineering (ASE 2021)*. 355–367. <https://doi.org/10.1109/ASE51524.2021.9678764>
- Andrea Romdhana, Mariano Ceccato, Alessio Merlo, and Paolo Tonella. 2022. IFRIT: Focused Testing through Deep Reinforcement Learning. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. 24–34. <https://doi.org/10.1109/ICST53961.2022.00013>
- Avraham Ruderman, Richard Everett, Bristy Sikder, Hubert Soyer, Jonathan Uesato, Ananya Kumar, Charlie Beattie, and Pushmeet Kohli. 2019. Uncovering Surprising Behaviors in Reinforcement Learning via Worst-case Analysis. In *Safe Machine Learning workshop at ICLR 2019*.
- G. A. Rummery and M. Niranjan. 1994. On-line Q-learning Using Connectionist Systems. *Technical Report CUED/F-INFEN/TR (1994)*. <https://cir.nii.ac.jp/crid/157366892427769344>
- Stuart J Russell and Peter Norvig. 2016. *Artificial intelligence: A modern approach*. Pearson Education Limited.

- John Schulman, Filip Wolski, Pra fulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. *ArXiv abs/1707.06347* (2017). <https://doi.org/10.48550/arXiv.1707.06347>
- Jianzhong Su, Hong-Ning Dai, Lingjun Zhao, Zibin Zheng, and Xiapu Luo. 2022. Effectively Generating Vulnerable Transaction Sequences in Smart Contracts with Reinforcement Learning-guided Fuzzing. In *Proc. 37th IEEE/ACM International Conference on Automated Software Engineering (ASE 2022)*. ACM Press, 36:1–36:12. <https://doi.org/10.1145/3551349.3560429>
- Youcheng Sun, Min Wu, Wenjie Ruan, Xiaowei Huang, Marta Kwiatkowska, and Daniel Kroening. 2018. Concolic testing for deep neural networks. In *Proc. 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018)*. 109–119. <https://doi.org/10.1145/3238147.3238172>
- Richard S. Sutton. 1988. Learning to Predict by the Methods of Temporal Differences. *Mach. Learn.* 3, 1 (1988), 9–44. <https://doi.org/10.1023/A:1022633531479>
- Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction* (2nd ed.). The MIT Press.
- Martin Tappler, Filip Cano Córdoba, Bernhard K. Aichernig, and Bettina Könighofer. 2022a. Search-Based Testing of Reinforcement Learning. In *Proc. Thirty-First International Joint Conference on Artificial Intelligence (IJCAI-22)*, Lud De Raedt (Ed.). International Joint Conferences on Artificial Intelligence Organization, 503–510. <https://doi.org/10.24963/ijcai.2022/72>
- Martin Tappler, Stefan Pranger, Bettina Könighofer, Edi Muskardin, Roderick Bloem, and Kim G. Larsen. 2022b. Automata Learning Meets Shielding. In *Proc. 11th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation. Verification Principles (ISoLA 2022) (Lecture Notes in Computer Science, Vol. 13701)*, Tiziana Margaria and Bernhard Steffen (Eds.). Springer, 335–359. https://doi.org/10.1007/978-3-031-19849-6_20
- Rosalía Tufano, Simone Scalabrino, Luca Pascarella, Emad Aghajani, Rocco Oliveto, and Gabriele Bavota. 2022. Using Reinforcement Learning for Load Testing of Video Games. In *Proc. IEEE/ACM 44th International Conference on Software Engineering (ICSE 2022)*. ACM Press. <https://doi.org/10.1145/3510003.3510625>
- Uraz Cengiz Türker, Robert M. Hierons, Mohammad Reza Mousavi, and Ivan Y. Tyukin. 2021. Efficient state synchronisation in model-based testing through reinforcement learning. In *Proc. 36th IEEE/ACM International Conference on Automated Software Engineering (ASE 2021)*. 368–380. <https://doi.org/10.1109/ASE51524.2021.9678566>
- Harm Van Seijen, Hado Van Hasselt, Shimon Whiteson, and Marco Wiering. 2009. A theoretical and empirical analysis of Expected SARSA. In *Proc. Symposium on Adaptive Dynamic Programming and Reinforcement Learning*. IEEE, 177–184. <https://doi.org/10.1109/ADPRL.2009.4927542>
- Harsh Vardhan and Janos Sztipanovits. 2021. Rare Event Failure Test Case Generation in Learning-Enabled-Controllers. In *2021 6th International Conference on Machine Learning Technologies (Jeju Island, Republic of Korea) (ICMLT 2021)*. ACM Press, 34–40. <https://doi.org/10.1145/3468891.3468897>
- Mahsa Varshosaz, Mohsen Ghaffari, Einar Broch Johnsen, and Andrzej Wasowski. 2023. *Formal Specification and Testing for Reinforcement Learning (Supplementary Material)*. <https://doi.org/10.5281/zenodo.8083298>
- Christopher John Cornish Hellaby Watkins. 1989. Learning from delayed rewards. (1989).
- Min Wu, Matthew Wicker, Wenjie Ruan, Xiaowei Huang, and Marta Kwiatkowska. 2020. A game-based approximate verification of deep neural networks with provable guarantees. *Theor. Comput. Sci.* 807 (2020), 298–329. <https://doi.org/10.1016/j.tcs.2019.05.046>
- Shaohua Zhang, Shuang Liu, Jun Sun, Yuqi Chen, Wenzhi Huang, Jinyi Liu, Jian Liu, and Jianye Hao. 2021. FIGCPS: Effective Failure-inducing Input Generation for Cyber-Physical Systems with Deep Reinforcement Learning. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 555–567. <https://doi.org/10.1109/ASE51524.2021.9678832>
- Yan Zheng, Yi Liu, Xiaofei Xie, Yepang Liu, Lei Ma, Jianye Hao, and Yang Liu. 2021. Automatic Web Testing Using Curiosity-Driven Reinforcement Learning. In *Proc. IEEE/ACM 43rd International Conference on Software Engineering (ICSE 2021)*. ACM Press, 423–435. <https://doi.org/10.1109/ICSE43902.2021.00048>
- Amirhossein Zolfagharian, Manel Abdellatif, Lionel C. Briand, Mojtaba Bagherzadeh, and Ramesh S. 2023. A Search-Based Testing Approach for Deep Reinforcement Learning Agents. *IEEE Transactions on Software Engineering* (2023), 1–22. <https://doi.org/10.1109/TSE.2023.3269804> To appear.

Received 2023-03-01; accepted 2023-06-27