RMLGym: a Formal Reward Machine Framework for Reinforcement Learning

Hisham Unniyankal[1], Francesco Belardinelli[2], Angelo Ferrando[1,*] and Vadim Malvone[3]

[1]*University of Genoa, Italy*
[2]*Imperial College London, United Kingdom*
[3]*Télécom Paris, France*

Abstract

Reinforcement learning (RL) is a powerful technique for learning optimal policies from trial and error. However, designing a reward function that captures the desired behavior of an agent is often a challenging and tedious task, especially when the agent has to deal with complex and multi-objective problems. To address this issue, researchers have proposed to use higher-level languages, such as Signal Temporal Logic (STL), to specify reward functions in a declarative and expressive way, and then automatically compile them into lower-level functions that can be used by standard RL algorithms. In this paper, we present RMLGym, a tool that integrates RML, a runtime verification tool, with OpenAI Gym, a popular framework for developing and comparing RL algorithms. RMLGym allows users to define reward functions using RML specifications and then generates reward monitors that evaluate the agent's performance and provide feedback at each step. We demonstrate the usefulness and flexibility of RMLGym by applying it to a famous benchmark problem from OpenAI Gym, and we analyze the strengths and limitations of our approach.

Keywords

Reinforcement Learning, Runtime Verification, OpenAI Gym

1. Introduction

Reinforcement learning (RL) is a subfield of machine learning that enables an agent to learn optimal behavior by interacting with an environment and receiving rewards as feedback. RL draws inspiration from the natural learning process observed in humans and animals, where learning occurs through personal experience and trial-and-error. In RL, an agent learns to navigate an environment by exploring various actions and observing their consequences. The agent receives rewards for actions that lead to desired outcomes and penalties for actions resulting in undesired consequences. The overarching goal of the agent is to maximize its cumulative reward over time. RL has found applications across diverse domains, including

WOA 2023: 24th Workshop From Objects to Agents, November 6–8, Rome, Italy
*Corresponding author.
s5049651@studenti.unige.it (H. Unniyankal); francesco.belardinelli@imperial.ac.uk (F. Belardinelli); angelo.ferrando@unige.it (A. Ferrando); vadim.malvone@telecom-paris.fr (V. Malvone)
https://www.doc.ic.ac.uk/~fbelard/ (F. Belardinelli); https://angeloferrando.github.io (A. Ferrando); https://vadimmalvone.github.io/ (V. Malvone)
0000-0002-7768-1794 (F. Belardinelli); 0000-0002-8711-4670 (A. Ferrando); 0000-0001-6138-4229 (V. Malvone)
© 2023 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).
CEUR Workshop Proceedings (CEUR-WS.org)

CEUR-WS.org/Vol-3579/paper1.pdf

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

games, robotics, control systems, and optimization, achieving remarkable results in challenging tasks.

Despite its successes, RL encounters several challenges and limitations. These include the complexity of specifying precise behaviors and the alignment problem, which relates to discrepancies between training and evaluation objectives. To address these challenges, researchers have proposed the integration of Runtime Verification (RV)-based approaches as complementary techniques to RL [1, 2].

Software correctness has long been a fundamental objective in the fields of programming language theory and theoretical computer science. Runtime verification [3], a relatively recent software verification approach grounded in model-checking principles, has gained prominence. RV involves monitoring a system's behavior during execution to ascertain whether it adheres to or deviates from predefined properties or specifications. RV serves various purposes, such as error detection and correction, enforcement of security policies, or providing feedback to adaptive systems. Furthermore, runtime verification techniques can be seamlessly integrated with reinforcement learning methods, with possibilities ranging from using runtime verification tools as reward machines [4, 5].

Reward machines employ finite-state machines to structure reinforcement learning reward functions. These machines feature states corresponding to sub-goals and transitions representing high-level events. The adoption of reward machines simplifies problem-solving, enhances efficiency through off-policy learning, and accommodates the specification of complex and precise behaviors, including temporal logic and non-Markovian rewards. Consequently, reward machines contribute to enhancing the safety and effectiveness of reinforcement learning agents.

In this paper, we introduce RMLGym, a novel framework designed to specify and learn complex behaviors within reinforcement learning environments. RMLGym builds upon OpenAI's gym framework and aims to bridge the gap between RL and RV. The framework leverages the RML [6], a runtime verification framework enabling the definition and verification of formal properties. Unlike conventional reinforcement learning setups with predefined reward functions, RMLGym employs RML specifications as reward constructors, generating rewards based on the satisfaction or violation of these specifications.

In our work, we define various RML properties and employ a range of reinforcement learning algorithms, including PPO [7], TD3 [8], SAC [9], and TRPO [10], to evaluate the RMLGym framework from diverse perspectives. Additionally, we conduct a comparative analysis between RMLGym and STLGym [5].

RMLGym provides a powerful tool for specifying and learning intricate behaviors within reinforcement learning environments. By utilizing RML specifications, we can express various properties and constraints governing the agent's behavior and generate rewards based on their compliance or violation. With RML rewards, we can evaluate the agent's performance in alignment with the specifications, rather than relying solely on the optimization of a predefined reward function.

```
env = gym.make('CartPole-v0')
```

Listing 1: The Python code for loading CartPole-v0 environment.

2. Preliminaries

In this section, we provide a concise introduction to reinforcement learning (RL), Runtime Monitoring Language (RML), and OpenAI's gym environment, a widely used toolkit for developing and comparing RL algorithms.

2.1. Reinforcement Learning

Reinforcement learning [11] represents a machine learning paradigm that empowers agents to learn how to navigate an environment through trial and error. Unlike other machine learning branches, such as supervised and unsupervised learning, where models rely on pre-annotated or pre-collected datasets, RL agents embark on a self-guided learning journey. This characteristic makes RL a more challenging yet potent approach.

RL agents interact with the environment in discrete time steps, observing the current state of the environment and the rewards received after each action. The fundamental objective of an RL agent is to maximize its cumulative rewards over time. One key modeling tool for RL problems is the Markov Decision Process (MDP), which encapsulates a set of states, actions, a transition function, and a discount factor.

2.2. OpenAI Gym

OpenAI Gym [12] serves as a toolkit meticulously crafted to facilitate exploration, experimentation, and innovation within the realm of reinforcement learning. This platform offers a rich collection of environments designed for training and evaluating RL agents. These environments encompass a wide spectrum of activities, including playing Atari games, manipulating robotic systems, and engaging in stock trading. OpenAI Gym provides a standardized API that facilitates seamless communication between RL algorithms and the designated environments.

To initiate the utilization of the OpenAI Gym Python library, the first step entails creating an environment. This process involves invoking the make() method, as demonstrated in Listing 1.

Once the environment is established, interactions with the environment can commence using the step () method (as shown in Listing 2). This method allows agents to interact with the environment, accepting an action as input and yielding the subsequent state, reward, and an indicator signaling the end of an episode.

The provided code exemplifies the execution of a single episode comprising 100 time steps. It showcases the interaction between an agent and the environment, where the agent selects actions based on its current state, and the environment provides feedback in the form of rewards and state transitions.

```python
t = 0
state = env.reset()
while not env.done and t < 100:
    action = agent.act(state)
    next_state, reward, done, _ = env.step(action)
    print(next_state, reward)
```

Listing 2: The Python code for single episode.

2.3. Runtime Monitoring Language (RML)

The Runtime Monitoring Language (RML)[1] [6] is a Domain-Specific Language (DSL) tailored for specifying highly expressive properties in runtime verification (RV), including non-context-free properties. We have chosen to employ RML in this work due to its support for parametric specifications and its inherent ability to define interaction protocols. The underlying low-level language of RML was originally developed for specifying communication protocols [13, 14].

In the context of our work, we provide a simplified and condensed overview of RML's syntax and semantics. However, for a complete presentation, you can refer to [6].

In RML, a property is expressed as a tuple $\langle t, ETs \rangle$, where t represents a term, and $ETs = \{\ ET_1, \ldots, ET_n\ \}$ is a set of event types. An event type ET is defined as a set of pairs $\{\ k_1 : v_1, \ldots, k_n : v_n\ \}$, where each pair identifies a specific piece of information (k_i) and its corresponding value (v_i). An event Ev is represented as a set of pairs $\{\ k'_1 : v'_1, \ldots, k'_m : v'_m\ \}$. An event Ev matches an event type ET if $ET \subseteq Ev$, meaning that for all $(k_i : v_i) \in ET$, there exists $(k_j : v_j) \in Ev$ such that $k_i = k_j$ and $v_i = v_j$. In simpler terms, an event type ET specifies the requirements that an event Ev must meet to be considered valid.

An RML term t, with t_1 and t_2 as other RML terms, can take various forms:

- ET, representing a set of individual traces containing events Ev where $ET \subseteq Ev$.
- $t_1\ \ t_2$, indicating the sequential composition of two sets of traces.
- $t_1 \mid t_2$, denoting the unordered composition of two sets of traces (also known as shuffle or interleaving).
- $t_1 \wedge t_2$, representing the intersection of two sets of traces.
- $t_1 \vee t_2$, signifying the union of two sets of traces.
- $\{$ let $x;\ t\ \}$, denoting the set of traces t where the variable x can be utilized, allowing the variable x to appear in event types in t and be unified with values.
- t^*, indicating the set of chains resulting from concatenating traces in t.

[1] https://rmlatdibris.github.io/ [visited on September 2023]

Event types in RML can also include variables. For instance, $ET(ag1, ag2) = \{\ sender : ag1, receiver : ag2, content : "hello"\ \}$ allows for flexibility in the sender and receiver fields while requiring the content to be the string "$hello$". When an event matches an event type with variables, the variables take on values from the event. For example, if the observed event is $Ev = \{\ sender : "Alice", receiver : "Bob", content : "hello"\ \}$, it matches ET with variable assignments: $ag1 =$ "$Alice$" and $ag2 =$ "Bob". This feature is essential for enforcing specific message order constraints using variables in RML terms.

3. RMLGym Architecture

This section provides an overview of the RMLGym framework, an extension of the OpenAI Gym environment that incorporates verdicts generated by the RML to create rewards. We begin by examining the architectural design and flow of the RMLGym framework, followed by a comprehensive exploration of its core components: the reinforcement learning (RL) agent, the environment, and the RML monitor.

Our methodology focuses on enhancing the environmental aspect of the RL process by integrating a reward machine that operates in conjunction with the RML monitor. Consequently, our approach is algorithm-agnostic and does not require modifications to the RL algorithm. Additionally, leveraging existing frameworks, we can retrain acquired policies to optimize compliance with specifications.

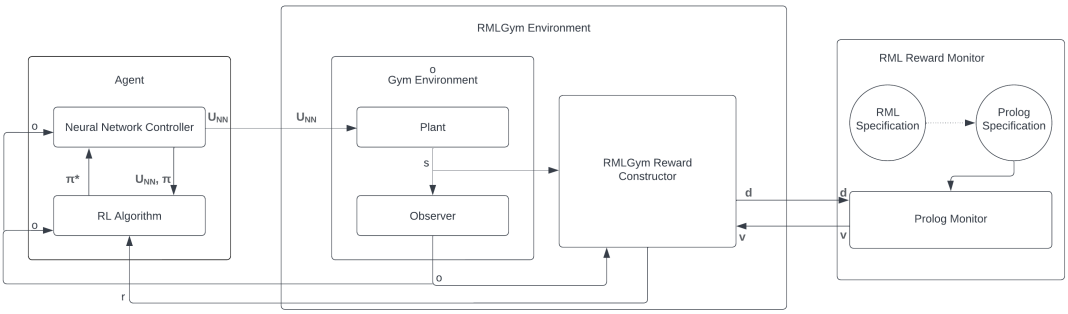


Figure 1: Representation of RMLGym with the RML monitor as a reward machine. The diagram uses U_{NN} to represent action probabilities, s and o for state information, and d and v to denote an event and verdict respectively.

As depicted in Figure 1, the RL training process using the RMLGym tool comprises three primary components: the Agent, the RMLGym environment, and the RML reward monitor. The agent component in RMLGym is a pre-existing RL agent (implemented using various RL algorithms). The RMLGym environment extends the capabilities of OpenAI's gym environment (as shown in Figure 1). In the following, we focus on the most relevant introduced component, that is the RML reward constructor.

3.1. RML Reward Constructor

The RML reward constructor plays a pivotal role within RMLGym. It is responsible for generating the reward, which acts as the reward machine. The reward constructor serves as an intermediary between the gym environment and the RML reward monitor. The integration of the Gym environment and the RML reward constructor results in the creation of the augmented gym environment known as RMLGym.

The reward constructor collects updated state information from both the plant and the observer OpenAI components. It then generates data intended for transmission to the RML reward monitor. Upon receiving a decision from the reward monitor, the constructor produces a reward based on the assessment received. For example, when dealing with Boolean values, a $False$ verdict may result in a negative numerical reward.

3.2. RML Reward Monitor

This paper's central focus is on integrating runtime verification into the reinforcement learning framework to validate the environment's state. The RML reward monitor is responsible for this runtime verification task. The monitor operates concurrently with the RL training process and reacts to incoming signals. It establishes a WebSocket connection and awaits incoming signals. Upon receiving a signal from the RMLGym environment's reward constructor, the monitor verifies the signal against the specified criteria (the formal property of interest). Subsequently, it generates a verdict based on the verification process and transmits it back to the reward constructor. All of this is accomplished by utilizing the existing RML engine [6].

4. Implementation

In this section, we introduce RMLGym, which extends OpenAI's Gym with RML integration. We discuss its key components: RMLGym environment, RL algorithms, RML specifications, and RML monitor implementation.

First, we cover RMLGym's development, showcasing how it enhances Gym with RML monitors for real-time RL training verification. Next, we explain the RML ecosystem setup, including specification generation, installation, integration, event structure, and provide a sample specification. Finally, we detail RL training using RMLGym, including algorithm configuration and utilization.

4.1. Gym Environment extension

In RMLGym, we use a YAML file to take configuration from the user. The YAML config file contains information about the environments, reward monitors, states, and rewards. Secondly, we override the functions such as $step\,()$, $make\,()$, and $close\,()$. We also added a new function called $monitor_reward\,()$ to perform the duty of the Reward Constructor.

```yaml
env_name: CartPole-v0
host: 127.0.0.1
port: 8081
```

Listing 3: Initial information to be provided through YAML config.

```yaml
variables:
    - name: pos
      type: float
      location: obs
      identifier: 0
    - name: angle
      type: float
      location: obs
      identifier: 2
```

Listing 4: The sample of observation details provided in the configuration file.

4.1.1. YAML configuration file

Listing 3 shows the initial configurations added to the YAML file. The first variable, env_name, is a parameter for specifying the name of the environment. The environment name should correspond to an environment available in OpenAI's gym. This example uses the CartPole environment from the gym. The second and third parameters in the configuration provide details about the monitor. The RML reward monitor is available on a WebSocket. The $host$ parameter specifies where the monitor is located, and the $port$ parameter specifies which port the monitor is available on. As an example, in Listing 3, RMLGym looks for the monitor at host 127.0.0.1:8081.

To generate data that is acceptable to RML monitors, an event should be created and sent to the monitor. The event can be generated in JSON format using the Reward Constructor, based on the specification. The Reward Constructor requires the details of the observations, as shown in Listing 4. The sample in Listing 4 shows a set of details that represent the observations in the environment. The pos is the position of the cart, and the angle is the angle of the pole. The type is the data type of the observation, and the location is the details of where the observation can be found. The identifier is a variable that specifies the index of the observation in the specified location.

After the event is sent to the monitor, the monitor verifies it and returns a verdict. Listing 5 shows a sample of reward details. RML monitors produce a four-valued logic for verdicts (as in standard runtime verification of LTL properties [15]). Therefore, Listing 5 represents the reward associated with each logic value.

```yaml
reward:
      name: task
      true: 1
      currently_true: 0
      currently_false: 0
      false: -1
```

Listing 5: Sample reward details provided in the configuration file.

4.1.2. Overriding gym functions

In order to incorporate Reinforcement Learning (RL) in the gym environment, we made modifications to existing functions within the gym framework. The `gym.Env` class serves as the fundamental class within the OpenAI gym framework. The class encapsulates an environment characterized by concealed dynamics that exhibit variability contingent upon the specific problem at hand. The system provides a standardized interface for engaging with the environment, encompassing functions such as resetting the state, executing actions, receiving rewards and observations, and determining whether the episode has concluded. The aforementioned functions establish the structure and scope of the observations and actions within the given environment.

We started with the `init()` method of the RMLGym class to create an instance of the RMLGym environment. It takes two parameters: `config_path` and `env`. The `config_path` is a string that specifies the path to a YAML file that contains the configuration settings for the environment. The `env` is an optional parameter that can be used to pass an existing gym environment object. If `env` is None, the method will create a new gym environment using the `env_name` from the config file. In this part, we initialize the WebSocket, the variables, the socket, and the reward details.

Then we override the `step()` function in `gym.Env` class. This function moves the environment forward by one step. The function accepts an action as its input and outputs four variables: `observation`, `reward`, `done`, and `info`. The return variable 'observation' is a component of the observation space within the environment. For instance, it could be an array containing the positional and velocity data of various objects. The reward is the outcome that the agent receives as a consequence of its action. The variable 'info' contains supplementary details that can be advantageous for the purposes of debugging, acquiring knowledge, and maintaining a log. For instance, potential components of the system may encompass metrics evaluating the agent's efficacy, latent factors influencing outcomes, or personalized incentives. The system is also capable of distinguishing between truncation and termination; however, this functionality is expected to be discontinued in the near future. The variable "done" is a Boolean value that indicates the conclusion of the episode. If the statement is accurate, subsequent invocations of this function will yield outcomes that are not defined.

In the `step()` function, we extract the information from `observation` and `info` to send it to the `monitor_reward()` function which is responsible for reward construction. We also update the reward info received from the reward constructor to `info` for debugging purposes.

At the end of each episode, it is necessary to invoke the `reset()` method to reset this

environment's state. The `reset()` function initiates a fresh episode by establishing the initial state and providing the initial observation as output. The random number generator can also be reset using the `seed` parameter, which can take on a numerical value or None. It is considered optimal to employ a numerical value only once subsequent to the establishment of the given setting. The `options` parameter provides supplementary data for the purpose of resetting, contingent upon the prevailing environment. In addition to its primary functionality of resetting the observation, the `reset()` method also provides the user with access to the `info` dictionary, which contains additional details pertaining to the observation. Both observation and information are comparable to the outputs obtained from the `step()` function. We also override the `reset()` function to reset the event details. Apart from the above functions, we override the function `close()` to close the web socket initialized for sending events to the monitor.

4.1.3. The function monitor_reward()

In addition, we have implemented a function named `monitor_reward()` which serves as the constructor for rewards. The function serves as an intermediary between the RML monitor and the environment. The function utilizes the parameter "done" to determine the completion status of the episode. The function utilizes the data extracted within the `step()` function and produces a novel event that is intended to be transmitted to the RML monitor. Once the event has been created, the function transmits it to the monitor via the initialized web socket and enters a state of waiting for the response. Upon receiving the response, the function generates a novel reward and subsequently modifies it to reflect the changes made to the agent. The monitor reward function is invoked at each timestep.

4.2. RMLGym integration

In this section, we explain the steps for integrating RML with reinforcement learning algorithms in OpenAI gym.

First, we need to install the RMLGym tool in the system. This can be achieved by cloning the GitHub repository[2] and install it through pip.

Next, we need to create and configure a YAML configuration file. The path of the YAML file should be passed to the RMLGym environment. Listing 6 shows how we loaded the RMLGym environment. The configuration file contains the environment name, web socket details, data details, and reward details, as we discussed previously. In the first line of Listing 6 we can see how the RMLGym library can be easily imported in the Python environment. In the third line instead the actual RMLGym environment is created (that is, the environment extended with all the functionalities to interact with the external RML reward machine).

Listing 7 shows the code we used to import and train PPO on the RMLGym environment. To train the RMLGym environment, we first load two environments using partial functions: one for training and one for testing. Both environments are created using the rmlgym.make function, which takes a configuration path as an argument. The configuration path points to a YAML file that contains the settings for the RML monitor, the reward constructor, and the gym

[2] https://github.com/hishamu7776/rml-gym

```
import rmlgym
config_path = './examples/environment.yaml'
rml_env = rmlgym.make(config_path)
```

Listing 6: Code to create a gym environment using RMLGym.

```
from spinup import ppo_pytorch as ppo

rml_env = partial(rmlgym.make, config_path)
test_rml_env = partial(rmlgym.make,  config_path_test)

ppo(rml_env,
    test_env_fn=test_rml_env,
    ac_kwargs=dict(hidden_sizes=(64, 64,)),
    seed=1630, steps_per_epoch=4000, epochs=100,
    gamma=0.99, pi_lr = 3e-4, vf_lr = 1e-3,
    train_pi_iters = 80, train_v_iters = 80, lam = 0.97,
    num_test_episodes = 10, max_ep_len = 200,
    target_kl = 0.01, save_freq = 1, clip_ratio=0.2)
```

Listing 7: Code to train PPO using rmlgym.

environment. However, both environments need different monitors to run on two different web sockets in parallel, so we use different configuration paths for training and testing. The configuration files also specify the specifications and rewards for each environment, which can be different depending on our objectives. For example, we can use a stricter specification for testing than for training, or we can use a sparse reward for training and a dense reward for testing.

After loading the environments, we call the RL algorithm of choice to be guided by the RML reward machine previously created and deployed. We pass the `rml_env` function as the first argument, which creates the training environment for the agent. We pass the `test_rml_env` function as the `test_env_fn` argument, which creates the testing environment for evaluation. We also pass a dictionary as `ac_kwargs` argument, which contains the arguments for the actor-critic network architecture. The rest of the arguments are related to different aspects of PPO's algorithm, such as learning rates, gradient steps, discount factor, GAE parameter, clipping parameter, target KL divergence, etc. We can tune these arguments to optimize PPO's performance on our RMLGym environment.

The PPO function runs PPO on our RMLGym environment for a specified number of epochs. In each epoch, it collects a batch of data from interacting with the environment using its current policy. It then updates its policy and value networks using gradient descent on PPO's objective function. It also evaluates its policy on the testing environment using a different monitor and specification. It logs and saves various metrics and statistics about its performance, such as

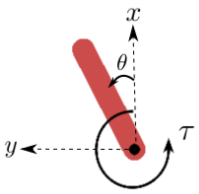


Figure 2: A screenshot of annotated pendulum environment.

episode length, episode return, entropy, KL divergence, etc. It also saves its model periodically so that we can load it later for analysis or further training.

5. Experimental results

In this section, we describe the experiments we conducted and their results to evaluate the effectiveness of different RL algorithms in various environments using the RMLGym tool and RML monitors. The main objective of these experiments is to compare the performance of RL algorithms with and without RML-based verification and to analyze the impact of RML on the learning process.

We begin by employing RL algorithms in the pendulum environment provided by the OpenAI Gym. In this environment, the goal is to find a solution to keep the pendulum in the upright position.

Since our tool is inspired by STLGym [5], which uses STL for specification and RTAMT tool[3] for generating rewards, we first employ the PPO algorithm to compare the results observed for the conventional rewards, STL rewards, and RML rewards. We then explore multiple algorithms such as PPO, TD3, and SAC using RML monitor-based rewards.

5.1. Pendulum Environment

In this experiment, we used the pendulum environment provided by the OpenAI Gym. This environment is a classic control environment in which the goal is to swing an inverted pendulum to an upright position and maintain it there. The pendulum is attached to a fixed point on one end, and the other end is free to move. The agent can control the pendulum by applying torque to the free end. The pendulum environment is a challenging environment, as the pendulum is unstable and can easily fall over. The agent must learn to apply the correct amount of torque at the right time in order to swing the pendulum up and keep it upright.

5.1.1. Variables of pendulum environment

We used the Pendulum-v1 environment for the experiment, which is available in OpenAI Gym version 0.15.17. The diagram depicted in Figure 2 illustrates the coordinate system that is employed for the implementation of the dynamic equations governing the behavior of the

[3] https://github.com/nickovic/rtamt

pendulum. There are 3 variables that can be seen in Figure 2. One variable that can be observed is the angle of the pendulum. The angle θ is measured in radians and represents the deviation of the pendulum from the vertical position. A positive theta value indicates a right tilt of the pendulum, while a negative theta value indicates a left tilt. Another variable that can be observed by the agent is the angular velocity ω of the pendulum, which represents the rate of change of change of theta. The angular velocity of the pendulum denotes the magnitude and direction of its swinging motion.

The agent has the ability to control τ, which represents the applied torque on the pendulum joint. The torque exerted on the pendulum, denoted as τ, is quantified in units of Newton meters (N m). A positive value of tau indicates that the agent exerts torque in the counter-clockwise direction, resulting in a right push on the pendulum. Conversely, a negative value of tau signifies that the agent applies a torque in the clockwise direction, causing a left push on the pendulum. In order to maintain stability, the agent must acquire the ability to adapt τ to θ and ω. The last variable in the pendulum environment is the pendulum end's Cartesian coordinates ($x - y$). The pendulum tip's horizontal and vertical positions relative to the joint origin are measured in meters. The agent cannot directly observe x-y coordinates, but they help visualize and analyze the pendulum behavior. From theta, trigonometric functions calculate x-y coordinates. L is the pendulum length, so $x = L \cdot sin(\theta)$ and $y = -L \cdot cos(\theta)$.

5.1.2. Rewards of pendulum environment

In this experiment, we evaluate four distinct rewards: baseline, STL dense, STL sparse, and RML rewards. The baseline reward, provided by the gym environment, encourages the agent to maintain the pendulum in an upright position with minimal energy consumption. It is calculated using the formula:

$$r = -(theta^2 + 0.1 * \theta_dt^2 + 0.001 * \tau^2)$$

Here, θ represents the pendulum's angle normalized between $[-\pi, \pi]$, θ_dt is its angular velocity, and τ is the applied torque.

The STL dense and sparse rewards from STLGym are based on Signal Temporal Logic specifications. The specification used in the Pendulum-V1 environment is:

$$eventually(always((\text{abs}(\theta) \leq 0.5)))$$

It expresses the goal of swinging the pendulum to an upright position and maintaining it there. The dense reward is calculated at every timestep, while the sparse reward is computed only at the episode's end. RTAMT computes these rewards based on the STL specification.

The RML reward, generated by our RML monitor, utilizes RML to check if the pendulum's angle (θ) remains within 0.5 radians from the vertical position at each timestep. It provides a positive reward of 1 when the condition is met (currently true) and a negative reward of -1 when it's not (currently false).

RML specification to verify the pendulum-v1 environment's observation.

$$Main = (good_theta_1 \wedge good_theta_2)^*$$

$$ETs = \{\ good_theta_1, good_theta_2\ \}$$
$$good_theta_1 = \{\ theta:\ x\ \}\ with\ x \leq 0.5$$
$$good_theta_2 = \{\ theta:\ x\ \}\ with\ x \geq -0.5$$

5.1.3. Result of pendulum

In this section, we evaluate the performance of three deep reinforcement learning algorithms (PPO, TD3, SAC) in the pendulum environment using various rewards: OpenAI Gym's baseline reward, STLGym's STL dense and sparse rewards, and RMLGym's RML reward. We aim to assess the RML reward's learning efficiency, robustness, and behavior quality compared to other rewards. For the PPO algorithm, we conducted 100 training epochs, each with 4,000 timesteps and a maximum episode length of 200. The four reward types were used for training. We evaluate the trained models using the baseline reward, STL sparse reward, and RML reward, with 10 test episodes for each assessment.

We plotted average returns over time for three types of rewards in Figure 4: baseline reward based on pendulum angle, velocity, and torque (Figure 4(a)); STL sparse reward based on STL specifications (Figure 4(b)); and RML reward based on the presented RML specification (Figure 4(c)). All rewards show similar learning efficiency and robustness, outperforming STL sparse reward. RML reward and STL dense reward slightly outperform baseline, reaching around 130 average return before 250,000 timesteps compared to baseline's 300,000 timesteps.

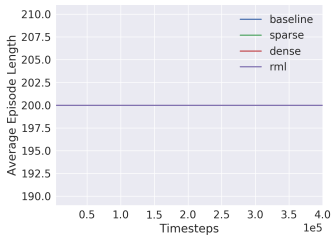
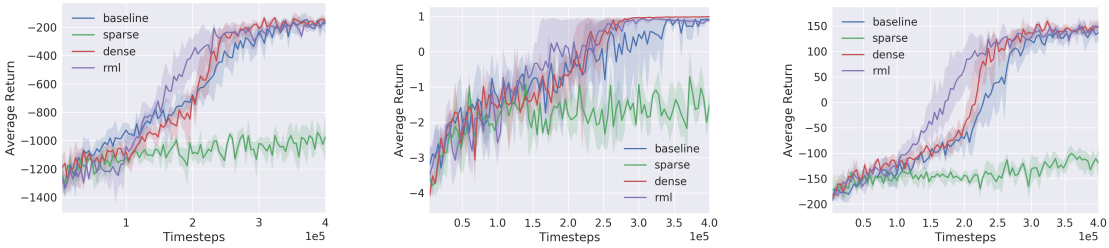


Figure 3: Average episode length observed after training PPO algorithm on pendulum environment with 4 different reward systems.



(a) Baseline reward. (b) STL sparse reward (c) RML reward

Figure 4: Comparative performance analysis of PPO algorithm with different reward methods. Each subfigure (4a, 4b, 4c) evaluates the models using a different reward metric: Baseline, STL Sparse, and RML respectively.

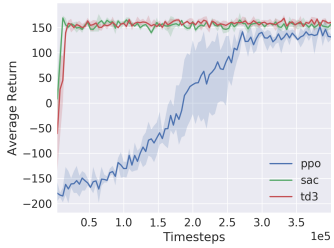


Figure 5: Average returns using RML rewards for PPO, TD3, and SAC algorithms.

We also plotted episode lengths over time in Figure 3, which remained constant at 200 steps for all rewards throughout training. The pendulum environment lacks a termination condition based on the pendulum state.

Lastly, we evaluated three deep RL algorithms (PPO, TD3, SAC) using RML reward, training each for 100 epochs with 4,000 timesteps per epoch, and a maximum episode length of 200 in the pendulum environment.

Figure 5 shows the average return for each algorithm over each timestep. The average return indicates how effectively the agent can maximize its cumulative reward in each episode by swinging and balancing the pendulum in the upright position.

Figure 5 shows that SAC and TD3 outperform PPO in terms of learning efficiency and robustness. The graph shows that SAC and TD3 learn faster and more stably than PPO. The high average return achieved by SAC and TD3 corresponds to an RML reward of 150, and it takes less than 20000 timesteps to converge to this reward. This is much faster than PPO, which takes more than 250000 timesteps to reach its peak. However, PPO only manages to attain a high return between 120 and 140 approximately.

6. Conclusions and Future Work

In summary, RMLGym is a novel RL framework that utilizes RML for constructing rewards, significantly enhancing the training of agents to meet complex safety requirements. Through extensive experimentation with various RL algorithms and environments, RMLGym has consistently demonstrated its effectiveness, often outperforming baseline rewards. Moreover, it has shown an improved alignment between training and evaluation objectives, promoting the development of robust RL agents. Despite its success, RMLGym does have limitations, such as the lack of quantitative rewards and its current reliance on OpenAI Gym. However, it represents a promising step forward in specifying and learning intricate behaviors in RL environments, ultimately contributing to scalability and stability in RL.

Looking ahead, future work may explore methods to enhance reward expressiveness with quantitative rewards, extend RMLGym's compatibility beyond OpenAI Gym, and apply RML to non-Markovian problems. These directions hold the potential to further advance the state of the art in reinforcement learning.

References

[1] C. Lazarus, J. G. Lopez, M. J. Kochenderfer, Runtime safety assurance using reinforcement learning, CoRR abs/2010.10618 (2020). URL: https://arxiv.org/abs/2010.10618. arXiv:2010.10618.

[2] B. Könighofer, F. Lorber, N. Jansen, R. Bloem, Shield synthesis for reinforcement learning, in: T. Margaria, B. Steffen (Eds.), Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part I, volume 12476 of *Lecture Notes in Computer Science*, Springer, 2020, pp. 290–306. URL: https://doi.org/10.1007/978-3-030-61362-4_16. doi:10.1007/978-3-030-61362-4_16.

[3] M. Leucker, C. Schallhart, A brief account of runtime verification, J. Log. Algebraic Methods Program. 78 (2009) 293–303. URL: https://doi.org/10.1016/j.jlap.2008.08.004. doi:10.1016/j.jlap.2008.08.004.

[4] A. Shukla, K. N. Hudemann, A. Hecker, S. Schmid, Runtime verification of P4 switches with reinforcement learning, in: Proceedings of the 2019 Workshop on Network Meets AI & ML, NetAI@SIGCOMM 2019, Beijing, China, August 23, 2019, ACM, 2019, pp. 1–7. URL: https://doi.org/10.1145/3341216.3342206. doi:10.1145/3341216.3342206.

[5] N. Hamilton, P. Robinette, T. T. Johnson, Training agents to satisfy timed and untimed signal temporal logic specifications with reinforcement learning, in: B. Schlingloff, M. Chai (Eds.), Software Engineering and Formal Methods - 20th International Conference, SEFM 2022, Berlin, Germany, September 26-30, 2022, Proceedings, volume 13550 of *Lecture Notes in Computer Science*, Springer, 2022, pp. 190–206. URL: https://doi.org/10.1007/978-3-031-17108-6_12. doi:10.1007/978-3-031-17108-6_12.

[6] D. Ancona, L. Franceschini, A. Ferrando, V. Mascardi, RML: theory and practice of a domain specific language for runtime verification, Sci. Comput. Program. 205 (2021) 102610. URL: https://doi.org/10.1016/j.scico.2021.102610. doi:10.1016/j.scico.2021.102610.

[7] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, O. Klimov, Proximal policy optimization algorithms, CoRR abs/1707.06347 (2017). URL: http://arxiv.org/abs/1707.06347. arXiv:1707.06347.

[8] S. Fujimoto, H. van Hoof, D. Meger, Addressing function approximation error in actor-critic methods, in: J. G. Dy, A. Krause (Eds.), Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018, volume 80 of *Proceedings of Machine Learning Research*, PMLR, 2018, pp. 1582–1591. URL: http://proceedings.mlr.press/v80/fujimoto18a.html.

[9] T. Haarnoja, A. Zhou, P. Abbeel, S. Levine, Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor, in: J. G. Dy, A. Krause (Eds.), Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018, volume 80 of *Proceedings of Machine Learning Research*, PMLR, 2018, pp. 1856–1865. URL: http://proceedings.mlr.press/v80/haarnoja18b.html.

[10] J. Schulman, S. Levine, P. Abbeel, M. I. Jordan, P. Moritz, Trust region policy optimization, in: F. R. Bach, D. M. Blei (Eds.), Proceedings of the 32nd International Conference on

Machine Learning, ICML 2015, Lille, France, 6-11 July 2015, volume 37 of *JMLR Workshop and Conference Proceedings*, JMLR.org, 2015, pp. 1889–1897. URL: http://proceedings.mlr.press/v37/schulman15.html.

[11] R. S. Sutton, A. G. Barto, Reinforcement learning: An introduction, MIT press, 2018.

[12] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, W. Zaremba, Openai gym, arXiv preprint arXiv:1606.01540 (2016).

[13] D. Ancona, A. Ferrando, V. Mascardi, Comparing trace expressions and linear temporal logic for runtime verification, in: E. Ábrahám, M. M. Bonsangue, E. B. Johnsen (Eds.), Theory and Practice of Formal Methods - Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday, volume 9660 of *Lecture Notes in Computer Science*, Springer, 2016, pp. 47–64. URL: https://doi.org/10.1007/978-3-319-30734-3_6. doi:10.1007/978-3-319-30734-3_6.

[14] D. Ancona, A. Ferrando, V. Mascardi, Parametric runtime verification of multiagent systems, in: K. Larson, M. Winikoff, S. Das, E. H. Durfee (Eds.), Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2017, São Paulo, Brazil, May 8-12, 2017, ACM, Richland, SC, 2017, pp. 1457–1459. URL: http://dl.acm.org/citation.cfm?id=3091328.

[15] A. Bauer, M. Leucker, C. Schallhart, The good, the bad, and the ugly, but how ugly is ugly?, in: O. Sokolsky, S. Tasiran (Eds.), Runtime Verification, 7th International Workshop, RV 2007, Vancouver, Canada, March 13, 2007, Revised Selected Papers, volume 4839 of *Lecture Notes in Computer Science*, Springer, 2007, pp. 126–138. URL: https://doi.org/10.1007/978-3-540-77395-5_11. doi:10.1007/978-3-540-77395-5_11.