pyRDDLGym: From RDDL to Gym Environments

Ayal Taitler[1], Michael Gimelfarb[1], Jihwan Jeong[1], Sriram Gopalakrishnan[2], Martin Mladenov[3], Xiaotian Liu[1], Scott Sanner[1]

[1] University of Toronto
[2] J.P. Morgan AI Research
[3] Google, BR

ataitler@gmail.com, mike.gimelfarb@mail.utoronto.ca, jhjeong@mie.utoronto.ca, sriram.gopalakrishnan@jpmchase.com, mmladenov@google.com xiaotian.liu@mail.utoronto.ca, ssanner@mie.utoronto.ca

Abstract

We present pyRDDLGym, a Python framework for the auto-generation of OpenAI Gym environments from RDDL declarative description. The discrete time step evolution of variables in RDDL is described by conditional probability functions, which fit naturally into the Gym step scheme. Furthermore, since RDDL is a lifted description, the modification and scaling up of environments to support multiple entities and different configurations becomes trivial rather than a tedious process prone to errors. We hope that pyRDDLGym will serve as a new wind in the reinforcement learning community by enabling easy and rapid development of benchmarks due to the unique expressive power of RDDL. By providing explicit access to the model in the RDDL description, pyRDDLGym can also facilitate research on hybrid approaches to learning from interaction while leveraging model knowledge. We present the design and built-in examples of pyRDDLGym, and the additions made to the RDDL language that were incorporated into the framework.

1 Introduction

Reinforcement Learning (RL) (Sutton and Barto 2018) and Probabilistic planning (Puterman 2014) are two research branches that address stochastic problems, often under the Markov assumption for state dynamics. The planning approach requires a given model, while the learning approach improves through repeated interaction with an environment, which can be viewed as a black box. Thus, the tools and the benchmarks for these two branches have grown apart. Learning agents do not require to be able to simulate model-based transitions, and thus frameworks such as OpenAI Gym (Brockman et al. 2016) have become a standard, serving also as an interface for third-party benchmarks such as Todorov, Erez, and Tassa (2012), Bellemare et al. (2013) and more.

As the model is not necessary for solving the learning problem, the environments are hard-coded in a programming language. This has several downsides; if one does wish to see the model describing the environment, it has to be reverse-engineered from the environment framework, complex problems can result in a significant development period, code bugs may make their way into the environment and finally, there is no clean way to verify the model or reuse it directly. Thus, the creation of a verified acceptable benchmark is a challenging task.

Planning agents on the other hand can interact with an environment (Sanner 2010a), but in many cases simulate the model within the planning agent in order to solve the problem (Keller and Eyerich 2012). The planning community has also come up with formal description languages for various types of problems; these include the Planning Domain Definition Language (PDDL) (Aeronautiques et al. 1998) for classical planning problems, PDDL2.1 (Fox and Long 2003) for problems involving time and continuous variables, PPDDL (Bryce and Buet 2008) for classical planning problems with action probabilistic effects and rewards, and Relational Dynamic Influence Diagram Language (RDDL) (Sanner 2010b) for describing MDPs and POMDPs. While agents can use the models described by these languages to simulate transitions and compute plans, it is also natural to leverage these description languages to auto-generate environments by decoupling the mathematical problem description from the environment generation.

In recent years auto-generation tools have emerged for specific description languages, allowing for both auto-generation of environments, and access to the problem formal model, serving as a bridge between planning and learning methods to interact with a single framework. *RDDLSim* (Sanner 2010a) is a long-standing independent framework that translates RDDL problems into an interactable environment. RDDLSim is a Java simulator, with a unique API, that requires interacting agents to manage TCP/IP connections. Although this is suitable for International Planning Competitions (IPC), the entry bar for rapid RL research is high. A more recent Python version of RDDLSim was developed to support mainly MDPs with continuous variables (Bueno 2020). As it had implemented the OpenAI Gym interface this tool was named *rddlgym*. Following that approach, *PDDLGym* (Silver and Chitnis 2020) was introduced. A Python tool that generates Gym environments from PDDL domain and problem files. Since PDDLGym works on PDDL files it can generate classical planning problems, i.e., deterministic problems. PDDLGym has some support for PPDDL which allows it to simulate action probabilistic effects, but state noise, concurrency, observations, and more

Copyright © 2023, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

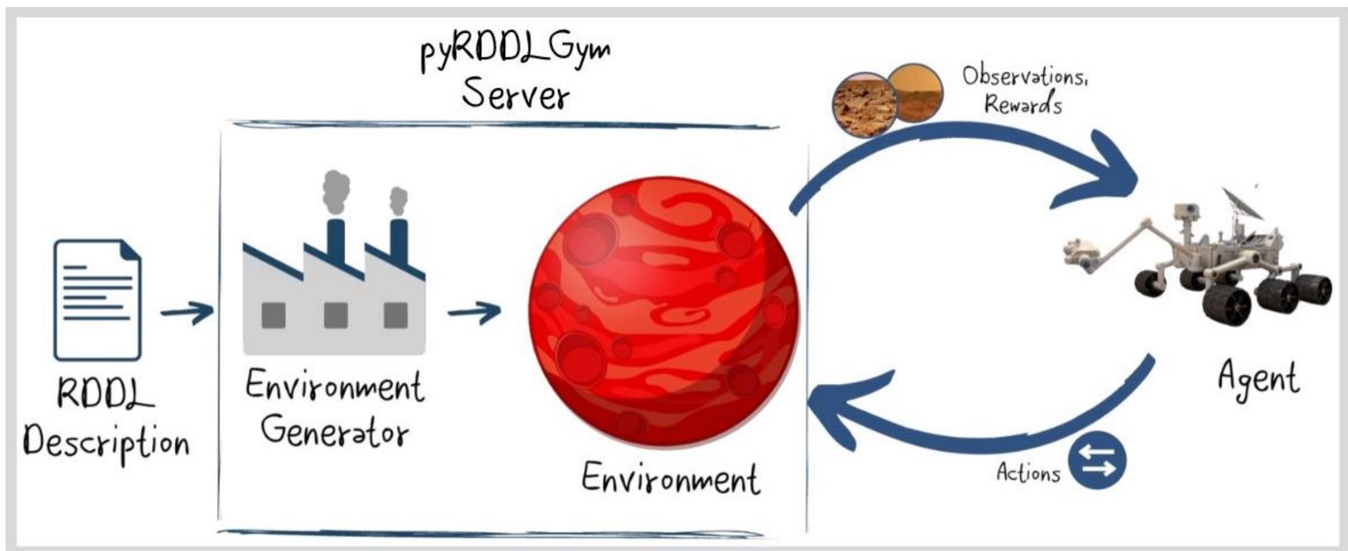


Figure 1: pyRDDLGym design concept.

components of the Markov model are not supported.

In order to be able to describe MDPs and POMDPs in a general way, allowing for factored descriptions, we present in this paper *pyRDDLGym*, a Python framework for auto-generation of Gym environments from RDDL description. The library is available at https://github.com/ataitler/pyRDDLGym. pyRDDLGym supports a Major subset of the RDDL language and in addition, has extended it to support for terminal states in MDPs. pyRDDLGym differs from rddlgym as it allows for derived-fluents, observations, discrete fluents, and more. It is also the only framework currently that supports state exogenous and endogenous noise, action concurrency, observations, and other blocks required for fully describing MDPs. We hope that pyRDDLGym will encourage more collaboration between the RL and Planning communities. This will allow hybrid methods leveraging both model description and interactions to emerge. Moreover, we aim to build a verified benchmark of stochastic domains for the probabilistic planning and RL communities.

2 RDDL Support and Extensions

RDDL (Sanner 2010b) is a *lifted* declarative language to describe MDPs, where states, actions, and observations (whether discrete or continuous) are parameterized variables. RDDL leverages parameterized variables, which helps with scaling up domains. These are simple templates for ground variables that can be obtained when given a particular problem instance defining possible domain objects. The evolution of a fully or partially observed stochastic process is specified via conditional probability functions (CPFs) over the next state variables conditioned on the current state and action variables, with allowed concurrency. The objective function in RDDL is defined by the immediate rewards and the discount factor one specifies. For a grounded model (instance), RDDL is just a factored MDP, or POMDP, if partially observed.

Thus, a *grounded* RDDL problem can be fitted into the Gym scheme of interaction where an agent acts and receives observation and reward from the environment. In this black box scenario, the explicit structure of the problem is lost and left for the agent to reason about. However, agents that can leverage the information in the model have the potential to boost their performance significantly.

Like all languages, RDDL also evolves in order to avoid ambiguity and increase expressibility. Also, in order to fit a Gym scheme additional features have been introduced. The deviation from the original language description (Sanner 2010b) is listed in the next section.

2.1 Language Variant

pyRDDLGym supports the majority of the original RDDL. The following components are omitted (or marked as deprecated) from the language variant implemented in pyRDDLGym:

- derived-fluent are supported by the framework as described in the language description. However, they are considered deprecated and will be removed from future versions.
- fluent levels are deprecated and are reasoned automatically by the framework as described in the next section.
- state-action-constraints are not implemented and considered deprecated in the language to avoid ambiguity. Only the newer syntax of specifying state-invariants and action-preconditions is supported.

Additional components and structures have been added to the language to increase expressivity, and to accommodate learning interaction type. These are listed here:

- Terminal state description has been added, described in details in the following sections.

```
types {
    x-pos, y-pos : object;
};
pvariables {
    NEIGHBOR(x-pos, y-pos, x-pos, y-pos) : { non-fluent, bool, default = false };
    alive(x-pos, y-pos) : { state-fluent, bool, default = false };
    set(x-pos, y-pos) : { action-fluent, bool, default = false };
};
cpfs {
    alive'(?x,?y) = if ([alive(?x,?y) ^ ([sum_{?x2 : x_pos, ?y2 : y_pos} NEIGHBOR(?x,?y,?x2,?y2)
                          ^ alive(?x2,?y2)] >= 2) ^ ([sum_{?x2 : x_pos, ?y2 : y_pos} NEIGHBOR(?x,?y,?x2,?y2)
                          ^ alive(?x2,?y2)] <= 3)] | [~alive(?x,?y) ^ ([sum_{?x2 : x_pos, ?y2 : y_pos}
                          NEIGHBOR(?x,?y,?x2,?y2) ^ alive(?x2,?y2)] == 3)] | set(?x,?y))
                    then Bernoulli(0.9)
                    else Bernoulli(0.1);
};
reward = sum_{?x : x-pos, ?y : y-pos} [alive(?x, ?y) - set(?x, ?y)];
```
(a)

```
non-fluents nf_GOL_inst {
    domain = game_of_life;
    objects {
        x-pos : {x1, x2};
        y-pos : {y1};
    };
    non-fluents {
        NEIGHBOR(x1,y1,x2,y1);
        NEIGHBOR(x2,y1,x1,y1);
    };
}
```
(b)

```
instance GOL_inst {
    domain = game_of_life;
    non-fluents = nf_GOL_inst;
    init-state {
        alive(x1, y1);
    };
    max-nondef-actions = 1;
    horizon = 40;
    discount = 1.0;
}
```
(c)

```
from pyRDDLGym import RDDLEnv

env = RDDLEnv.RDDLEnv(domain=
        'GOT.rddl', instance='GOT_inst.rddl')
state = env.reset()
for step in range(env.horizon):
    env.render()
    action = select(env.action_space,
            env.NumConcurrentActions)
    obs, reward, done, info =
            env.step(action)
```
(d)

Figure 2: pyRDDLGym code examples. A pyRDDLGym environment is characterized by an RDDL domain file and an instance & non-fluents file. (a) Fluents, CPFs, and reward for the game of life problem (b) A non-fluents block, defining a two neighboring cell topology. (c) An instance block defining the parameters and init-state of the problem. (d) Using the domain and instance files, an RDDLEnv can be initialized. Interaction is similar to that of any OpenAI Gym environment.

- action-preconditions are implemented according to Sanner (2010b). However, they are subject to user preference. By default the framework *does not* enforce the expressions in the action-preconditions block. Thus, upon violation, a warning will be printed to the user and the simulation will push the actions inside the legal space by using the default value and the simulation will continue. To ensure correct behavior it is expected from the domain designer to include the appropriate logic of how to handle an invalid action within the *CPFs* block. In the case where the user does choose to enforce action-preconditions, the simulation will be interrupted and an appropriate exception will be thrown.

- Direct inquiry of variable (states/action) domains is supported through the standard action_space and state_space properties of the environment. For this functionality to work correctly, the domain designer is required to specify each (lifted-)variable bound within the action-preconditions block in the format "*fluent* OP BOUND" where OP $\in \{<, >, >=, <=\}$, and BOUND is a deterministic function of the problem parameter to be evaluated at instantiation.
- Parameter inequality is supported for lifted types. I.e., the following expression $?p == ?r$ can be evaluated to True or False.
- Nested indexing is now supported, e.g., $fluent'(?p, ?q) = NEXT(fluent(?p, ?q))$.
- Vectorized distributions such as Multivariate normal, Student, Dirichlet, and Multinomial are now supported.
- Basic matrix algebra such as determinant and inverse operation are supported for two appropriate fluents.
- $argmax$ and $argmin$ are supported over enumerated types (enums).

2.2 Level Reasoning

Formally in RDDL when a derived or an interm fluent is declared, its level should also be stated to define its position in the Dynamic Bayes Net (DBN) of the problem, which dictates the fluents' order of evaluation. While the level hierarchy declaration is not overly complicated from the domain designer's side, it is completely unnecessary. In pyRDDLGym, following the implementation of Sanner (2010a), this declaration is omitted, and ignored if supplied. The order of fluent evaluation at any given time step is always as follows

$$s_t \rightarrow d_t \rightarrow i_t \rightarrow s_{t+1}, \quad (1)$$

where s_t denotes the current state, d_t denotes the derived fluents, i_t denotes the interm fluents, and s_{t+1} denotes the next state. A requirement when designing a domain is that the fluent order must form a directed acyclic graph (DAG). Thus, we first generate a call graph from the CPFs, for all fluent types. Then we use topological sorting to sort the fluents by the order of evaluation. This has two merits, the first is the reasoning over the over of fluent evaluation, and second, validation that there are no cycles in the evaluation order, and that the correct order of evaluation in (1) is conserved.

2.3 Terminal States

Another extension to RDDL is the addition of terminal states to the language. An MDP may have a terminal state, which can be in the form of a goal state or a state where there are no available actions anymore. The key is that in both of these cases, it is desired to end the simulation, e.g., an agent hits a wall, a pendulum falls beyond a threshold, etc. RDDL currently does not support these cases, only fixed horizon problems. On the other hand, Gym can terminate an episode anytime with the *done* flag.

Thus, an additional block has been introduced into RDDL to support terminal states. The keyword denoting the terminal states block is *termination*, and it supports a list of conditions:

$$termination \; \{cond_1; \; cond_1; \; ... \; cond_N\}; \quad (2)$$

where the $cond_i, \; i \in \{1, .., N\}$ are Boolean conditions, and the full termination condition is a disjunction over all the inner conditions, i.e.,

$$termination = cond_1 \vee cond_2 \vee ... \vee cond_N. \quad (3)$$

3 Design and Implementation

In order to be compatible with the Gym API, five methods need to be implemented. First, *__init__ ()*, which initializes the environment, and in that method, all the parsing of the RDDL, grounding, and fitting into the environment scheme is done. *reset()* in which the environment is returned to the initial state, and the initial state is returned or *None* in case of a POMDP. *step()* in which the transition function is calculated according to the CPFs in the RDDL file. *render()* in which the visualization is implemented, and finally the *close()* method in which resources are being freed so the simulation can be terminated. Two more constructs to be implemented are the properties *action_space* and *observation_space*, which inform the agent about the type and domain of the action and observation spaces respectively.

3.1 Instantiation of Gym Environments

The RDDL description contains three components. The first is the *domain* block, in which information about the lifted domain is provided, e.g., types, CPFs, fluent definitions, etc. The second component is the *instance* block which specifies a specific problem, i.e., all that is needed in order to ground the lifted abstract domain. The third block is the non-fluents block, which is being pointed at from the instance block as it is also part of how to instantiate a specific problem. This block is the constant of the instance, or more precisely it defines the topology of the problem. To create an environment all three are required. Thus, the domain containing the global problem definitions should be provided in a ".rddl" file, and the other two blocks, as they together define an instance, should be placed in a second separate ".rddl" file. The *__init__()* method requires both these files in order to generate a Gym environment.

The call to the *__init__()* method, invokes first the parsing of the RDDL description provided in the domain, instance, and non-fluents, then the grounder is invoked on the parsing tree in order to generate a specific instance as required. At this point, the transformation from RDDL types to Gym spaces is done and the action and observation properties are populated. The CPF sampler object is also instantiated here for the grounded problem. In pyRDDLGym three additional properties are provided. The first is *horizon*, which informs the agent about the horizon of the problem, assuming no terminal state is encountered. The value returned by this property is the number of horizon steps as defined in the RDDL horizon field in the instance block. The second property is *NumConcurrentActions* which denotes the maximum number of concurrent actions the agent can send to the framework in a single time step. In the case where the field *max-nondef-actions* in the RDDL instance specify *pos-inf*, the number of all available grounded actions in the problem is returned, indicating that there is no limit on the number of concurrent actions. The third and last property is *discount*, which simply informs the discount factor the environment will use to calculate the total reward.

3.2 Observation and Action Spaces

RDDL and Gym have different representations of the state and action spaces. While RDDL always expects to receive the full vector of actions even if *max-nondef-actions* indicates a lower number than the available number of actions, Gym interacting agents are requested to supply only the desired actions without explicitly keeping track of all the actions in the problem and their default values. In order to mitigate these issues pyRDDLGym implements actions and states (and observations) in a *Gym.Spaces.Dict* object where the keys are the action/state/observation grounded name, and the value is the intended value to pass to the simulation in case of actions, or the state/observation value in the current time step. When an agent is requested to pass actions to the environment, it is required to pass only the desired action, and not the full list of actions, the environment will augment the agent's specified action with the remainder of the actions with their default values before evaluation of the CPFs.

The conversion of RDDL types to Gym spaces is intuitive. Real valued fluents are represented as *Gym.Spaces.Box*, integers are converted to *Gym.Spaces.Discrete*, and Booleans are converted to *Gym.Spaces.Discrete(2)*. The bounds are according to the constraints specified in the *action-preconditions* block, or *Numpy.inf* to denote that there is no bounding value. The *action_space* and *observation_space* will return the actions and observations respectively, where the dictionary key is the action/state/observation grounded name, and the value is the appropriate *Gym.Spaces*. Sampling from the environment *action_space* will always return a valid value as the type is a two-way transformation between Gym.Spaces and RDDL types, and the bounds informed by the environment property are also in accordance with the *action-preconditions* specification. In any case, the CPFs also serve as a last line of defense against values that are out of bounds.

From Lifted to Grounded RDDL is a lifted description of a problem, i.e., the domain block defines operator schemes with parameters in contrast to explicit variables. Thus the expression

$$fluent_cpf'(?type) = expr; \tag{4}$$

is a template for all objects of type $?type$. The objects are a concrete realization of a problem and so defined in the instance. E.g., for $o_1,\ o_2$ of type $type$ the grounded realization for the objects will be

$$\begin{aligned} fluent_cpf'(o_1) = expr; \\ fluent_cpf'(o_2) = expr; \end{aligned} \tag{5}$$

For both state/observation and action fluents, a name conversion takes place when grounding a lifted fluent with arguments, to a grounded fluent. The conversion of lifted fluent names to grounded fluent names is done with underscores. As underscores are a valid character for naming in RDDL, fluents are separated from parameters by triple underscore (___), and parameters are separated from each other by double underscore (__). I.e., a lifted fluent with two arguments will be grounded as follows

$$\begin{aligned} &fluent(type_1, type_2) \\ &\quad \to fluent(o_1, o_2) \to fluent___o_1__o_2, \end{aligned} \tag{6}$$

where o_1 and o_2 are objects of types $type_1$ and $type_2$ respectively. $fluent(type_1, type_2)$ is the lifted template, $fluent(o_1, o_2)$ is the grounded variable, and $fluent___o_1__o_2$ is the exposed variable name in pyRDDLGym.

MDPs and POMDPs pyRDDLGym supports both MDPs and POMDPs. If no observation fluent is declared in the RDDL files, the framework will return the full state. In the case where observation fluents are present in the RDDL, only the observation fluents are returned by the framework. An observation fluent should be defined for a state fluent if it is observed; it is not possible to flag a state fluent as observed. Although this might seem like an unnecessary duplication of fluents, it allows for increased flexibility in specifying observations, e.g., deterministic observation, stochastic observation, etc. Due to the conversion of defining observations over the transitioned state, there is no observation at time zero, only the initial state in the case of MDPs.

3.3 Single Time-Step Evolution

In a standard MDP, variable values are computed at discrete time steps, leading to the evolution of the process through repeated execution of the single-step transition function. Consequently, the *step()* method in pyRDDLGym evaluates the CPFs defined in the RDDL problem description for all fluents in the instance. The *step()* method in pyRDDLGym undergoes four main steps. It first verifies if the actions are within the allowed range, raising a warning or exception accordingly. Then, it sorts the CPFs based on their level determined by dependency analysis. Next, it evaluates the CPF expressions using the current states and actions. Finally, it checks the state against the state invariants to ensure a legal transition and identify terminal states. If any state invariants are violated, indicating a design error, an exception is raised, and the episode is terminated, indicating it should not be used for learning.

3.4 Resetting the Environment

When *reset()* is called the simulation is simply reverted to the initial state as specified in the instance block of the RDDL. Note that there is no randomness in the *reset()* method, if one desires to reset to a different initial state, a new environment must be instantiated with a new instance. Naturally, only the initial state will be identical between episodes of the same environment, the rest of the steps will be stochastic as per the dynamics specified in that RDDL domain's CPFs. In the case of MDP the *reset()* method will return the initial values of the states, and in the case of POMDP, a dictionary with the observations will be returned, where all the values are set to the Python value *None*.

3.5 Visualization

pyRDDLGym also supports visualization through the Gym's *render()* method. By invoking the *render()* method, a visualization of the current state is displayed on the screen, and an image object is returned to the user. The default built-in visualizer that is implemented in pyRDDLGym for every environment is called *TextViz*, which generates an image with a textual description of the observations and their current values. In order to create a user-defined visualizer, all is required is to implement the *pyRDDLGym.Visualizer.StateViz* interface, and specify the visualization object to the environment with the method *set_visualizer(<VizObject>)*. Some samples of the built-in visualizations provided with pyRDDLGym are shown in figure 3.

4 pyRDDLGym Beyond the Engine

4.1 Built-in Environments

pyRDDLGym offers advanced visualizations for all example environments. Except for the Fire Fighting domain,

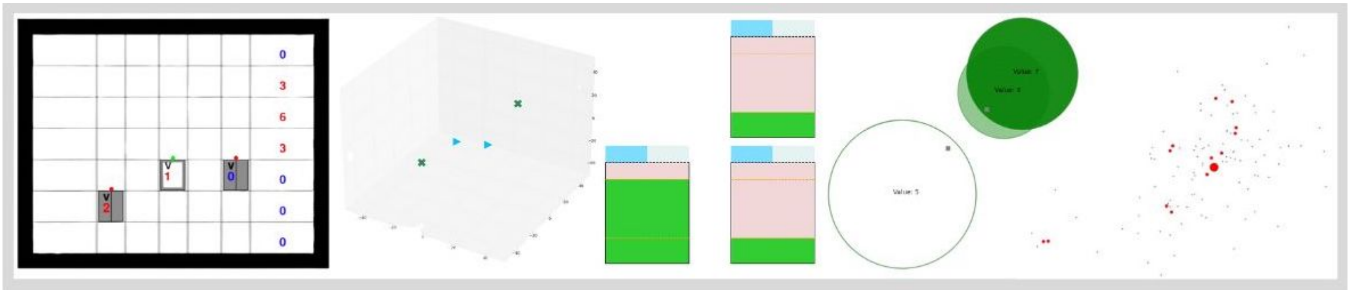


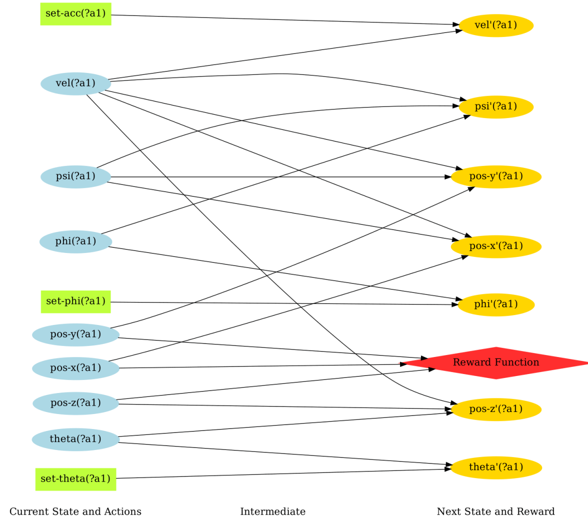
Figure 3: Examples of environments implemented in pyRDDLGym. From left to right: elevators, UAV, power unit commitment, Mars rover and recommender systems.

Domain Name	Action Space	State Space	Source
Cart-pole Continuous	Continuous	Continuous	Barto, Sutton, and Anderson (1983)
Cart-pole Discrete	Discrete	Continuous	Barto, Sutton, and Anderson (1983)
Elevators	Discrete	Discrete	Ours
Mars Rover	Mixed	Mixed	Taitler et al. (2019)
Mountain-car	Continuous	Continuous	Moore (1990)
Power Unit Commitment Discrete	Discrete	Mixed	Abdou and Tkiouat (2018)
Power Unit Commitment Continuous	Continuous	Continuous	Abdou and Tkiouat (2018)
Racing Car	Continuous	Continuous	Ours
Recommender Systems	Discrete	Continuous	Mladenov et al. (2020)
Fire Fighting	Discrete	Discrete	Karafyllidis and Thanailakis (1997)
UAV Continuous	Continuous	Continuous	Ours
UAV Discrete	Discrete	Continuous	Ours
UAV Mixed	Mixed	Continuous	Ours
Supply Chain	Discrete	Discrete	Kemmer et al. (2018)
Traffic	Discrete	Continuous	Lin et al. (2009)

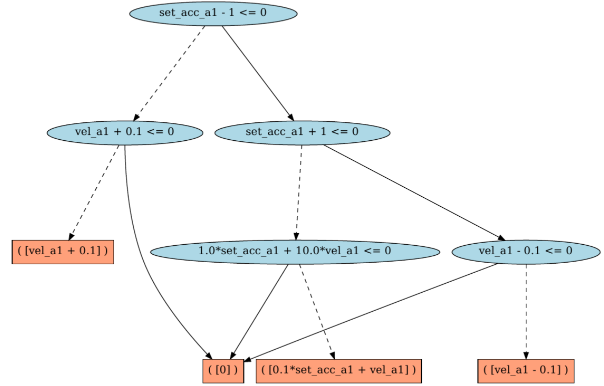
Table 1: List of domains currently included in pyRDDLGym. For each environment, we report the original source behind the RDDL files and the type of action and state spaces, whether they are fully discrete, continuous, or mixed discrete-continuous.

which is a classical RDDL domain used in the pyRDDLGym tutorial, all other examples involve continuous or hybrid spaces. All the domains have well-defined internal structures (factored spaces) that are explicitly specified in the RDDL files. The initial goal of the framework was to promote hybrid methods that combine reinforcement learning and model-based planning. Thus, we believe it's time to establish a benchmark for problems that require more than just model-free learning or combinatorial search. As pyRDDLGym is a versatile framework, all RDDL domains from previous IPCs can be easily imported by providing the relevant domain and instance files (Sanner 2010a). It's worth noting that all the past IPC problems can be accessed and imported into pyRDDLGym through the "rddlrepository" repository, which can be found at: https://github.com/ataitler/rddlrepository.

Currently, there are 15 environments implemented in pyRDDLGym. Adapted from the previous simulator (Sanner 2010a), the Fire Fighting domain is a discrete domain.



(a) DBN generated for the UAV domain



(b) xADD generated for the velocity state in the UAV domain

Figure 4: pyRDDLGym auxiliary tools

It was added to pyRDDLGym as an introduction domain for learning pyRDDLGym and RDDL. Mars Rover domain is a dynamic version of the standard MAPF problem (Standley 2010), with dynamic properties and inspiration from Taitler et al. (2019) and Fernandez-Gonzalez, Williams, and Karpas (2018). The Power Unit Commitment example has two versions, a discrete action version which was taken as it is from the previous 2014 IPC, and a continuous action version. There are three original domains; elevators, which is a discrete domain, Racing car is a continuous control domain, in which completely different racing tracks can be constructed by playing with the non-fluents. The UAV domain is the third, and it has three versions as well. The three versions differ by the type of action space. The first is fully continuous, the second is fully discrete, and the third has a mixed discrete-continuous action space. The dynamics in the UAV are a slightly simplified version of the full model described in Hull (2007). Two domains were adapted from OpenAI Gym classic control domains, in order to have some familiar domains and to show how simple the conversion process into RDDL is. Mountain Car, and Cart-pole. Cart-pole comes in two versions: one for continuous actions and one for discrete actions. Another domain is a Recommander System (Mladenov et al. 2020), which is not a classical control or Operation Research problem but has gained focus in the last years, and in real life scales up to millions of objects, which by itself is a challenge to decision-making algorithms. The last domain is a traffic domain, modeled after the QT-M/BLX models (Guilliard et al. 2016; Lin et al. 2009). It is a macroscopic flow and traffic model that can be scaled from a single isolated intersection to a large network of signalized intersections. The list of environments with the details is presented in Table 1. Additionally, example of how to go from problem definition to a working pyRDDLGym environment is given in appendix A.

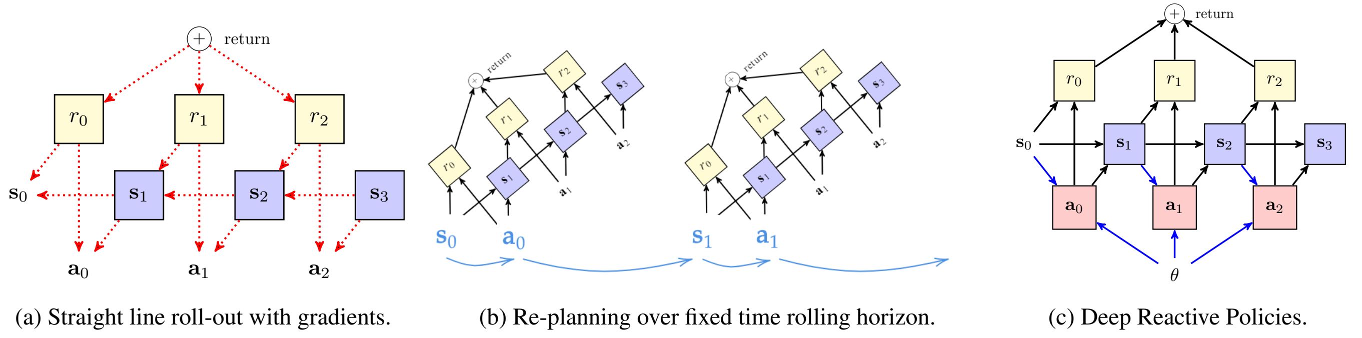
4.2 The Example Manager

In order to access the built-in examples in pyRDDLGym, the unified interface of supplying domain and instance files, and a visualizer object is used. The ExampleManager object is where the domains, instances, and visualizers for all examples are registered. The ExampleManager is documented in the pyRDDLGym documentation, but it has several key methods. The first is a *ListExamples()*, a static method that lists all the examples with a short description. The names of the domains as listed in that method can be used to instantiate the ExampleManager object and access the example details. Then the methods *get_domain()*, *get_instance(#)* can be used to get the path to the domain and instance files. Finally *get_visualizer()* returns the dedicated visualization object for the example.

4.3 Auxiliary Tools

XADDs XADD (eXtended Algebraic Decision Diagram) (Sanner, Delgado, and de Barros 2011) enables compact representation and operations with symbolic variables and functions. In fact, this data structure can be used to represent CPFs defined in a RDDL domain once it is grounded for a specific RDDL instance. pyRDDLGym can generate XADDs for all states/observations in the grounded domain for the use of planning agents.

Dynamic Bayes Nets Visualization With the XADD compilation of a given domain instance, one can easily visualize the dependencies between different fluents. For this purpose, a Dynamic Bayes Nets (DBNs) visualization tool



(a) Straight line roll-out with gradients. (b) Re-planning over fixed time rolling horizon. (c) Deep Reactive Policies.

Figure 5: JAXPlanner modes of operations.

is also provided in the framework. This tool provides a way to produce diagrams similar to an influence diagram.

4.4 Model-Based Planner

While NumPy (Harris et al. 2020) serves as the default backend for pyRDDLGym, it also offers support for JAX (Bradbury et al. 2018) as an alternative that can handle gradients. This capability enables planning using back-propagation approaches (Wu, Say, and Sanner 2017; Bueno et al. 2019). Moreover, pyRDDLGym features an implemented JAXPlanner, which encompasses the entire range of the RDDL language, facilitates GPU execution, and addresses stochastic problems. In practical terms, given model, a roll-out described by an influence graph can be generated, and the total reward or other performance metrics, such as risk (Patton et al. 2022), can be calculated and differentiated with respect to the inputs.

One notable feature of JAXPlanner is its ability to handle hybrid continuous-discrete state and action spaces. To facilitate derivatives of CPFs with respect to action-fluents, JAXPlanner replaces functional dependencies F_i with differentiable relaxations, formalized as families of functions $\{f_{i,\tau} : \tau > 0\}$ indexed by some hyper-parameter τ. The variables $\tilde{X}_i = f_{i,\tau}(\mathrm{Pa}(\tilde{X}_i))$ with $\tilde{X}_1 = X_1$ then define an equivalent DAG $\tilde{\mathcal{G}} = (\tilde{X}, \mathrm{Pa})$ with the same edges as $\mathcal{G}$ but where nodes X_i are replaced by $\tilde{X}_i$. Thus, a differentiable model approximation is achieved.

Given that Boolean logic in RDDL lacks inherent differentiability, it becomes essential to discover functions $f_{i,\tau}$ that effectively approximate Boolean logic. One approach is to substitute Boolean operations with t-norms (Hájek 2013). JAXPlanner incorporates t-norm approximations, but it also provides support for operation overloading, allowing users to define their own operators. This flexibility enables users to tailor the implementation according to their specific requirements. Specifically, a t-norm is a function $T : [0, 1]^2 \to [0, 1]$ which satisfy 4 properties: commutativity, monotonicity, associativity, and inclusion of the identity element. For Boolean-valued quantities a, b, JAXPlanner uses the following approximations:

- $a \wedge b \approx T(a, b)$
- $\neg a \approx 1 - a$,

from which the other RDDL operations can be derived, e.g.:

- $a \vee b \equiv \neg(\neg a \wedge \neg b) \approx 1 - T(1 - a, 1 - b)$
- $a \implies b \equiv \neg a \vee b \approx 1 - T(a, 1 - b)$
- $\forall\{x_1, x_2, \ldots x_m\} \equiv x_1 \wedge x_2 \cdots \wedge x_m \approx T(x_1, T(x_2, T(\ldots)))$
- $\exists\{x_1, x_2, \ldots x_m\} \equiv \neg\forall\{\neg x_1, \neg x_2, \ldots \neg x_m\} \approx 1 - T(1 - x_1, T(1 - x_2, T(\ldots)))$.

The conditional branching statement such as

$$\text{if } (c) \text{ then } a \text{ else } b$$

is rewritten in our framework as

$$f(a, b, c) = c * a + (1 - c) * b,$$

which is a continuous function. A popular choice for approximating relational operations such as $a > b$, $a < b$ and $a = b$ as suggested in prior literature is to use sigmoid functions. JAXPlanner opt to use the logistic sigmoid:

$$a > b \approx f_{>}(a, b, \tau) = \mathrm{sigmoid}((a - b)/\tau)$$
$$a = b \approx f_{=}(a, b, \tau) = \mathrm{sech}^2((b - a)/\tau),$$

where τ refers to the temperature parameter.

JAXPlanner offers two operational modes, depicted in Figure 5. The first mode involves generating a straight line plan (SLP) by performing a complete horizon rollout of the problem. It is worth noting that utilizing this method with a fixed horizon at each step produces a solution resembling model predictive control. The second mode involves employing a neural network to train a deep reactive policy (DRP), which is similar to the approach taken in Bueno et al. (2019).

5 Conclusion and Future Work

We have presented pyRDDLGym, an open-source Python framework that automatically creates OpenAI Gym environments from the RDDL domain and instance files. We hope that the availability of such a framework will help foster collaboration between researchers in the learning and planning communities. We also believe that by separating the process of problem design and the no longer needed programming task, a verified benchmark for RL and planning can be established independently of a specific platform. Finally from our own experience the generation of a brand-new environment has been accelerated significantly and the logic can be formally verified, which so far has not been possible.

References

Abdou, I.; and Tkiouat, M. 2018. Unit Commitment Problem in Electrical Power System: A Literature Review. *International Journal of Electrical & Computer Engineering (2088-8708)*, 8(3).

Aeronautiques, C.; Howe, A.; Knoblock, C.; McDermott, I. D.; Ram, A.; Veloso, M.; Weld, D.; SRI, D. W.; Barrett, A.; Christianson, D.; et al. 1998. PDDL— The Planning Domain Definition Language. *Technical Report, Tech. Rep.*

Barto, A. G.; Sutton, R. S.; and Anderson, C. W. 1983. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE transactions on systems, man, and cybernetics*, (5): 834–846.

Bellemare, M. G.; Naddaf, Y.; Veness, J.; and Bowling, M. 2013. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47: 253–279.

Bradbury, J.; Frostig, R.; Hawkins, P.; Johnson, M. J.; Leary, C.; Maclaurin, D.; Necula, G.; Paszke, A.; VanderPlas, J.; Wanderman-Milne, S.; and Zhang, Q. 2018. JAX: composable transformations of Python+NumPy programs.

Brockman, G.; Cheung, V.; Pettersson, L.; Schneider, J.; Schulman, J.; Tang, J.; and Zaremba, W. 2016. Openai gym. *arXiv preprint arXiv:1606.01540*.

Bryce, D.; and Buet, O. 2008. International Planning Competition Uncertainty Part: Benchmarks and Results. In *The Sixth International Planning Competition, ICAPS*.

Bueno, T. P. 2020. rddlgym. https://github.com/thiagopbueno/rddlgym.

Bueno, T. P.; de Barros, L. N.; Mauá, D. D.; and Sanner, S. 2019. Deep reactive policies for planning in stochastic nonlinear domains. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, 7530–7537.

Fernandez-Gonzalez, E.; Williams, B.; and Karpas, E. 2018. Scottyactivity: Mixed discrete-continuous planning with convex optimization. *Journal of Artificial Intelligence Research*, 62: 579–664.

Fox, M.; and Long, D. 2003. PDDL2. 1: An extension to PDDL for expressing temporal planning domains. *Journal of artificial intelligence research*, 20: 61–124.

Guilliard, I.; Sanner, S.; Trevizan, F. W.; and Williams, B. C. 2016. Nonhomogeneous time mixed integer linear programming formulation for traffic signal control. *Transportation Research Record*, 2595(1): 128–138.

Hájek, P. 2013. *Metamathematics of fuzzy logic*, volume 4. Springer Science & Business Media.

Harris, C. R.; Millman, K. J.; van der Walt, S. J.; Gommers, R.; Virtanen, P.; Cournapeau, D.; Wieser, E.; Taylor, J.; Berg, S.; Smith, N. J.; Kern, R.; Picus, M.; Hoyer, S.; van Kerkwijk, M. H.; Brett, M.; Haldane, A.; del Río, J. F.; Wiebe, M.; Peterson, P.; Gérard-Marchant, P.; Sheppard, K.; Reddy, T.; Weckesser, W.; Abbasi, H.; Gohlke, C.; and Oliphant, T. E. 2020. Array programming with NumPy. *Nature*, 585(7825): 357–362.

Hull, D. G. 2007. *Fundamentals of airplane flight mechanics*, volume 19. Springer.

Karafyllidis, I.; and Thanailakis, A. 1997. A model for predicting forest fire spreading using cellular automata. *Ecological Modelling*, 99(1): 87–97.

Keller, T.; and Eyerich, P. 2012. PROST: Probabilistic Planning Based on UCT. In *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling (ICAPS 2012)*, 119–127. AAAI Press.

Kemmer, L.; von Kleist, H.; de Rochebouët, D.; Tziortziotis, N.; and Read, J. 2018. Reinforcement learning for supply chain optimization. In *European Workshop on Reinforcement Learning*, volume 14.

Lin, S.; De Schutter, B.; Xi, Y.; and Hellendoorn, J. 2009. A simplified macroscopic urban traffic network model for model-based predictive control. *IFAC Proceedings Volumes*, 42(15): 286–291.

Mladenov, M.; Creager, E.; Ben-Porat, O.; Swersky, K.; Zemel, R.; and Boutilier, C. 2020. Optimizing Long-term Social Welfare in Recommender Systems: A Constrained Matching Approach. In III, H. D.; and Singh, A., eds., *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, 6987–6998. PMLR.

Moore, A. W. 1990. Efficient Memory-based Learning for Robot Control. Technical report, University of Cambridge.

Patton, N.; Jeong, J.; Gimelfarb, M.; and Sanner, S. 2022. A Distributional Framework for Risk-Sensitive End-to-End Planning in Continuous MDPs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, 9894–9901.

Puterman, M. L. 2014. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons.

Sanner, S. 2010a. RDDLSim. https://github.com/ssanner/rddlsim.

Sanner, S. 2010b. Relational Dynamic Influence Diagram Language (RDDL): Language Description. Http://users.cecs.anu.edu.au/ ssanner/IPPC_2011/RDDL.pdf.

Sanner, S.; Delgado, K. V.; and de Barros, L. N. 2011. Symbolic dynamic programming for discrete and continuous state MDPs. In *Proceedings of the Twenty-Seventh Conference on Uncertainty in Artificial Intelligence*, 643–652.

Silver, T.; and Chitnis, R. 2020. PDDLGym: Gym Environments from PDDL Problems. In *International Conference on Automated Planning and Scheduling (ICAPS) PRL Workshop*.

Standley, T. 2010. Finding optimal solutions to cooperative pathfinding problems. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 24, 173–178.

Sutton, R. S.; and Barto, A. G. 2018. *Reinforcement learning: An introduction*. MIT press.

Taitler, A.; Ioslovich, I.; Gutman, P.-O.; and Karpas, E. 2019. Combined time and energy optimal trajectory planning with quadratic drag for mixed discrete-continuous task planning. *Optimization*, 68(1): 125–143.

Todorov, E.; Erez, T.; and Tassa, Y. 2012. MuJoCo: A physics engine for model-based control. In *2012 IEEE/RSJ*

International Conference on Intelligent Robots and Systems, 5026–5033. IEEE.

Wu, G.; Say, B.; and Sanner, S. 2017. Scalable planning with tensorflow for hybrid nonlinear domains. *Advances in Neural Information Processing Systems*, 30.

A From Math to RDDL Domain Case Study

We take as a case study the classical Gym environment Cartpole, the physical realization of the system is given in figure 6.

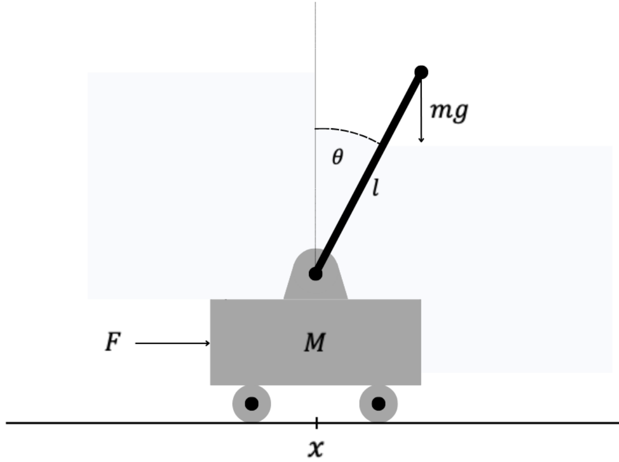


Figure 6: The cart-pole balancing model

The horizontal position of the cart is represented as x, while the angular position of the pole is denoted as θ. The velocities or rates of change are appropriately indicated by the dot notation, $\dot{x}$ and $\dot{\theta}$. The force propelling the cart along the horizontal axis is denoted as F, and all of these variables are time-dependent functions. The system parameters include the cart's mass (M), the pole's mass (m), the length of the pole (l), and the gravitational constant (g). Additional constants that can be incorporated are the allowable bounds for the pole angle, track length, and force limits. The Gym environment implements in python directly the system dynamics, thus, all the parameters are hard-coded within the *step()* function. To implement the same environment with the same behaviour in RDDL we should start with the dynamic equations. The state vector naturally is $[x, v_x, \theta, v_\theta]^T$. In continuous time the equations are given by (Barto, Sutton, and Anderson 1983):

$$\begin{aligned}\dot{v}_\theta &= \frac{g\sin\theta + \cos\theta\left(\frac{-F - mlv_\theta^2\sin\theta}{m+M}\right)}{l\left(\frac{4}{3} - \frac{m\cos^2\theta}{m+M}\right)} \\ \dot{v}_x &= \frac{F + ml(v_\theta^2\sin\theta - \dot{v}_\theta\cos\theta)}{m+M}\end{aligned} \tag{7}$$

Using a first order discrete approximation with sampling time T, and writing (7) as a system of first order linear system yields:

$$\begin{aligned}x(k+1) &= x(k) + Tv(k) \\ v(k+1) &= v(k) + Tu_x(k) \\ \theta(k+1) &= \theta(k) + Tv_\theta(k) \\ v_\theta(k+1) &= v_\theta(k) + Tu_\theta(k)\end{aligned} \tag{8}$$

where u_x, u_θ are virtual inputs to the cart and pole motion appropriately, defined as:

$$\begin{aligned}u_\theta(k) &= \frac{g\sin\theta(k) - \cos\theta(k)t(k)}{l\left(\frac{4}{3} - \frac{m\cos^2\theta(k)}{m+M}\right)} \\ u_x(k) &= t(k) - \frac{lmu_\theta(k)\cos\theta(k)}{m+M}\end{aligned} \tag{9}$$

and $t(k)$ is an auxiliary definition to keep the math a little cleaner:

$$t(k) = \frac{F(k) + lmv_\theta^2(k)\sin\theta(k)}{m+M} \tag{10}$$

Now, we are prepared to convert the mathematical representation into RDDL. We commence by declaring the variables involved in the problem. The gravity constant is a non-fluent with a default value of 9.8. Since the Gym environment features a discrete action space, where 1 represents a constant force to the right and 0 represents a constant force to the left, we define a non-fluent that represents the force. All other changing constants, such as the masses of the cart and pole, pole length, time step, and the limits on the cart position and pole angle, are also non-fluents. The auxiliary functions described in equations (9) and (10) are considered interm-fluents. The four variables that define the state are referred to as state-fluents, while the input force applied to the cart's side is an action-fluent. The functional definition is presented within the "cpfs" block, where the next time step's state variables are denoted with a prime symbol, e.g., x' instead of $x(k+1)$. The "cpfs" block directly implements equations (8)-(10). An additional interm-fluent is defined for the purpose of translating the action index to force value, implemented in the first CPF at the "cpfs" block.

The simulation terminates when the "termination" block evaluates to "true," indicating that either the cart has moved outside the track or the pole angle has exceeded the angle limit. The state-invariants serve as safeguards to ensure that no instance places the cart or pole outside the feasible ranges at the initial state, and that the problem's constants are non-negative. It is permissible to define an instance on the Moon, for example, with a gravity coefficient of $1.64[m/s^2]$. The action-preconditions ensure that the only acceptable actions are the integers 1 and 2. Lastly, the reward function grants the agent a positive value of 1.0 for every time step in which the cart remains on the track and the pole does not fall outside the angle limits. Each aspect of the domain is defined mathematically, without the need to implement complex functionality or termination decisions. Typically, an instance file would solely set the initial state of the Cart-pole system, although one has the option to adjust the constants (input force, cart mass, pole length, etc.) according to their requirements. The complete domain description can be found in Figure 7.

```
pvariables {

    // forces
    GRAVITY   : { non-fluent, real, default = 9.8 };
    FORCE-MAG : { non-fluent, real, default = 10.0 };

    // cart and pole properties
    CART-MASS : { non-fluent, real, default = 1.0 };
    POLE-MASS : { non-fluent, real, default = 0.1 };
    POLE-LEN  : { non-fluent, real, default = 0.5 };

    // other constants
    TIME-STEP : { non-fluent, real, default = 0.02 };
    POS-LIMIT : { non-fluent, real, default = 2.4 };
    ANG-LIMIT : { non-fluent, real, default = 0.2094395 };

    // interm & derived fluents
    force : { interm-fluent, real };
    temp : { interm-fluent, real };
    acc : { interm-fluent, real };
    ang-acc : { interm-fluent, real };

    // states
    pos     : { state-fluent, real, default = 0 };
    ang-pos : { state-fluent, real, default = 0 };
    vel     : { state-fluent, real, default = 0 };
    ang-vel : { state-fluent, real, default = 0 };

    // actions
    force-side : { action-fluent, int, default = 0 };
};
```

```
termination {
    pos < -POS-LIMIT | pos > POS-LIMIT;
    ang-pos < -ANG-LIMIT | ang-pos > ANG-LIMIT;
};›

state-invariants {

    // state is within limits
    pos >= -POS-LIMIT;
    pos <= POS-LIMIT;
    ang-pos >= -ANG-LIMIT;
    ang-pos <= ANG-LIMIT;

    // system is physically plausible
    GRAVITY >= 0 ^ FORCE-MAG >= 0;
    CART-MASS >= 0 ^ POLE-MASS >= 0 ^ POLE-LEN >= 0;
    TIME-STEP >= 0 ^ POS-LIMIT >= 0 ^ ANG-LIMIT >= 0;
};

action-preconditions {
    force-side >= 0;
    force-side <= 1;
};
```

```
cpfs {
    // the actual signed force on the cart
    force = if(force-side == 1)
              then FORCE-MAG
              else -FORCE-MAG;

    // compute the pole angular acceleration
    temp = (force + POLE-LEN * POLE-MASS * pow[ang-vel, 2] * sin[ang-pos]) / (CART-MASS + POLE-MASS);
    ang-acc = (GRAVITY * sin[ang-pos] - cos[ang-pos] * temp) / (
        POLE-LEN * ((4.0 / 3.0) - (POLE-MASS * pow[cos[ang-pos], 2] / (CART-MASS + POLE-MASS))));

    // compute the cart acceleration
    acc = temp - (POLE-LEN * POLE-MASS * ang-acc * cos[ang-pos] / (CART-MASS + POLE-MASS));

    // Euler integration formula
    pos' = pos + TIME-STEP * vel;
    ang-pos' = ang-pos + TIME-STEP * ang-vel;
    vel' = vel + TIME-STEP * acc;
    ang-vel' = ang-vel + TIME-STEP * ang-acc;
};
```

```
reward = if (pos < -POS-LIMIT | pos > POS-LIMIT | ang-pos < -ANG-LIMIT | ang-pos > ANG-LIMIT)
            then 0 else 1;
```

Figure 7: The Cart-pole problem depicted in RDDL format, featuring variable definitions (top left), mathematical functions and state evolution (middle section), termination conditions and constraints (top right), and reward function definition (bottom).