Towards a Domain-Specific Modelling Environment for Reinforcement Learning

Natalie Sinani, Sahil Salma, Paul Boutot and Sadaf Mustafiz
Department of Computer Science, Toronto Metropolitan University, Toronto, ON, Canada
{*natalie.sinani, ssalma, pboutot, sadaf.mustafiz*}*@torontomu.ca*

Keywords: Reinforcement Learning, Machine Learning, Domain-Specific Modelling Environments, Modelling Languages, Low-Code Environment.

Abstract: In recent years, machine learning technologies have gained immense popularity and are being used in a wide range of domains. However, due to the complexity associated with machine learning algorithms, it is a challenge to make it user-friendly, easy to understand and apply. In particular, reinforcement learning (RL) applications are especially challenging for users who do not have proficiency in this area. In this paper, we use model-driven engineering (MDE) methods and tools for developing a framework for abstracting RL technologies to improve the learning curve for RL users. Our domain-specific modelling environment for reinforcement learning supports syntax-directed editing, constraint checking, code synthesis, and enables comparative analysis of results generated with multiple RL algorithms. We demonstrate our framework with the use of several reinforcement learning applications.

1 INTRODUCTION

The advent of artificial intelligence and machine learning technologies marks a significant transformation in the software and systems landscape, leading to groundbreaking developments across various fields. Among these, reinforcement learning (RL) (Sutton and Barto, 2018) (Ding et al., 2020), a fundamental paradigm of machine learning, is gaining considerable attention. Initially recognized for its success in gaming, where machines were able to outperform expert human players, RL is now increasingly relevant in dynamic and adaptive environments, with applications spanning from healthcare and finance to autonomous vehicles. However, the complexity of RL algorithms presents a significant barrier, often requiring domain expertise and technical skills for effective implementation and utilization. Despite the growing demand for RL solutions, current professional profiles lack the comprehensive skills sets necessary to fully leverage its potential, posing a challenge for widespread adoption across industries (Bucchiarone et al., 2020).

Just as the need for intelligence in various application areas has led to the integration of machine learning algorithms with more user-friendly interfaces, there is a similar demand for RL (Naveed et al., 2024). Data scientists typically rely on specific libraries such as OpenAI Gym, TensorFlow Agents (TF-Agents), and Stable Baselines to implement RL algorithms. These require a deep understanding of the intricate interfaces. However, majority of the research work in the RL domain focuses on enhancing algorithms and approaches to achieve better accuracy and results in prediction and learning. There is very limited work on simplifying RL concepts and tools to enable non-technical users, such as business analysts, project managers, domain experts, and students to engage with RL technologies. A user-friendly RL framework reduces the technical barriers to entry, allowing individuals without extensive programming or data science backgrounds to experiment with RL and incorporate it into their work. Such a framework can also lead to enhanced collaboration between technical teams and domain experts, since non-technical users often have valuable domain expertise that can considerably enhance RL projects, ultimately ensuring that solutions align more closely with real-world needs.

Model-driven engineering (MDE) can contribute to this challenge by providing enablers to directly express and manipulate domain-specific problems (Bucchiarone et al., 2020). Domain-specific languages (DSL) in MDE aim to reduce complexity with the use of abstraction. We propose a DSL tailored for RL, Reinforcement Learning Modelling Language (RLML), that serves as an intuitive and accessible front-end for

Sinani, N., Salma, S., Boutot, P. and Mustafiz, S.
Towards a Domain-Specific Modelling Environment for Reinforcement Learning.
DOI: 10.5220/0013123800003896

In *Proceedings of the 13th International Conference on Model-Based Software and Systems Engineering (MODELSWARD 2025)*, pages 40-51
ISBN: 978-989-758-729-0; ISSN: 2184-4348

RL users. RLML focuses on model-free algorithms, which are more widely used and extensively tested, to ensure broad applicability. The RLML framework, built on top of the JetBrains MPS[1] platform (Voelter et al., 2013), provides an integrated textual modelling environment that streamlines the creation, execution, and analysis of RL models. The RLML concrete syntax makes it easier for users to run RL algorithms with limited RL knowledge. Depending on the level of technical expertise, the users can choose to change the algorithm parameters or use the default values. Furthermore, to bridge the gap between Python, the predominant language in ML, and Java, we have implemented model-to-code transformations for both languages, enhancing the accessibility of RL algorithms.

This paper is organized as follows: Section 2 provides a brief background on reinforcement learning and discusses related work. Section 3 presents our domain-specific modelling language, RLML. Section 4 covers the RLML environment built using the language workbench, JetBrains MPS. Section 5 demonstrates the use of our framework with several reinforcement learning applications. Section 6 concludes the paper.

2 BACKGROUND AND RELATED WORK

This section provides some necessary background on reinforcement learning and discusses related work.

2.1 Reinforcement Learning

Reinforcement learning (RL) is an area of machine learning concerned with how intelligent agents ought to take actions in an environment in order to maximize the notion of rewards. It is a self-teaching system trying to find an appropriate action model that would maximize an agent's total cumulative reward, by following the trial and error method. In general, the RL algorithms reward the agent for taking desired actions in the environment, and punishes i.e., grants negative or zero rewards, for the undesired ones (Sutton and Barto, 2018). The following are the key components that describe RL problems.

- Environment: The RL environment (Graesser and Keng, 2019) represents all the existing states that the agent can enter. It produces information that describe the states of the system. The agent interacts with the environment by observing the state space and taking an action based on the observation. Each action receives a positive or negative reward, which informs the agent on selecting the next state.
- Agent: This is represented by an intelligent RL algorithm that learns and makes decisions to maximize the future rewards while moving within the environment.
- State: The state represents the current situation of the agent.
- Action: The mechanism by which the agent transitions between states of the environment.
- Reward: The environment feeds the agent with rewards, which are numerical values that the agent tries to maximize over time. They are received on each action and may be positive or negative.

Reinforcement learning algorithms estimate how *good* it is for the agent to be in a certain state. This estimation is the calculation of what is known as a value function. The value function gets measured based on the expected future rewards that the agent will receive starting from a given state s, and according to the actions that the agent will make, and this is referred to as the *expected return*. The goal of an RL algorithm is to find the optimal policy for an agent to follow that maximizes the expected return. An optimal policy will have the highest possible value in every state. The optimal policy is implicit and can be derived directly from the optimal value function. There are many different approaches to find the optimal policy. They are mainly categorized as model-based or model-free learning algorithms, in addition to deep reinforcement learning. It is worth mentioning that this categorization is not comprehensive and it is often blurry (Brunton and Kutz, 2019).

When the model of the environment is available, which is the case with model-based algorithms, the RL problem is simpler and we can utilize policy iteration or value iteration algorithms. To learn the optimal policy or value function we either need access to the model (environment) and its probability distribution over states, or we try to build a model. When the agent knows this information, it can use it to plan its next moves. However, it is more challenging when we are dealing with model-free algorithms, and it is often the case in real life scenarios, where the agent does not know the environment and needs to discover it. As stated by Sutton and Barto (Sutton and Barto, 2018): *model-based methods rely on planning as their primary component, while model-free methods primarily rely on learning*. Finally, deep reinforcement learning incorporates deep learning techniques and algorithms in order to learn the model (François-Lavet

[1] https://www.jetbrains.com/mps/

et al., 2018).

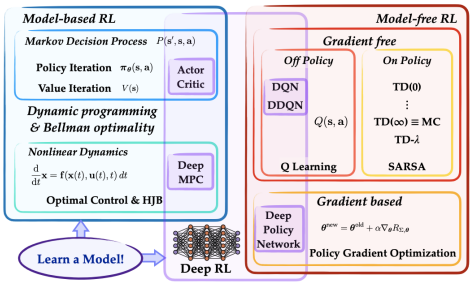

Figure 1: Reinforcement Learning (RL) Classification (Adapted from (Brunton and Kutz, 2019)).

In model-free approaches, the agent learns and evaluates how good actions are by trial and error method. The agent relies on the past experiences to derive the optimal policy (described earlier). Various available algorithms in this approach include (see Fig. 1), Monte Carlo learning, Model-Free Actor Critic, SARSA: State–action–reward–state–action learning, DQN and Q-Learning.

In this work, we have focused on model-free gradient-free algorithms. The objective of such algorithms is to maximize an arbitrary score, which is the value function, hence also referred to as value-based algorithms. In value-based methods, the algorithm does not store any explicit policy, only a value function. Some algorithms, such as actor critic, are both value and policy-based. Q-Learning, perhaps, is one of the most dominant model-free algorithm which learns the Q-function directly from experience, without requiring access to a model.

Since we are using both model-driven engineering and machine learning technologies in our research, we would like to clarify that the term *models* have different meaning in these two fields. Models in MDE refer to software models and are an abstract representation of the elements that define the software and system domain (Schmidt, 2006). On the other hand, models in machine learning are algorithms that contain defined instruction and mathematical formulations (Jiang, 2021). Models in ML can be trained to recognize certain patterns in provided data.

2.2 Related Work

While machine learning is widely applied in the MDE area, there is limited work available on the application of MDE in the ML area (Bucchiarone et al., 2020). Moreover, as stated in (Naveed et al., 2024), only four studies have proposed MDE solutions for RL. Domain-specific languages for the artificial intelligence domain and more specifically, the machine learning domain, are recent contributions, driven by the need to make ML algorithms more accessible and to reduce the learning curve. To the best of our knowledge, this is the first work on developing an MDE-based framework for reinforcement learning.

In this section, we discuss some relevant ML/RL work that use some form of MDE. We also present an overview of relevant existing RL libraries and toolkits.

2.2.1 Application of MDE in ML

Our work takes inspiration from the Classification Algorithm Framework (CAF) (Meacham et al., 2020). While CAF was developed for machine learning classification algorithms, RLML is developed for reinforcement learning algorithms. They have similar configuration-like interface for non-technical users. CAF supports code generation in the Java language, while RLML supports both Java and Python. Unlike CAF, RLML offers a comparator feature.

Liaskos et al. (Liaskos et al., 2022) present a modelling and design process for generating simulation environments for RL based on goal models defined using iStar. This allows model-based reasoning to be carried out, and for agents to be trained prior to deployment in the target environment. High-level RL models can be automatically mapped to these simulation components. While the scope of our work is different, this tool can be integrated into our framework to allow RLML models (i.e. alogrithms used) to be mapped to high-level RL models.

DeepDSL (Zhao et al., 2017) is a DSL embedded in Scala, for developing deep learning applications. It provides compiler-based optimizations for deep learning models to run with less memory usage and/or in shorter time. DeepDSL allows users to define deep learning networks as tensor functions and has its own compiler that produces DeepDSL Java program. Just as DeepDSL aims to bridge the gap for non-technical uses in deep learning, we developed RLML to fill a similar gap in reinforcement learning.

Our proposed approach is unique and different than the rest of the reviewed work, because we use model-driven engineering to create abstractions in the RL domain. This not only provides simplification for RL but also provides other benefits of model-driven engineering, such as improved maintainability, visualization and scalability.

2.2.2 RL Libraries and Toolkits

As the field of reinforcement learning evolves, various platforms and libraries have been established to facilitate the development of RL applications, each with

its distinct features and focus areas. For instance, the Reinforcement Learning Toolkit developed with Unreal Engine (Sapio and Ratini, 2022) provides immersive simulation environments. Python libraries such as RL-coach[2], Tensorforce[3],TRFL[4], and TF Agents[5] offer robust support for RL algorithms but require developers to engage deeply with algorithmic details and understand the intricate concepts of RL. These existing tools, written in Python, demand substantial technical understanding of RL processes, from algorithm implementation to the handling of complex calculations. This requirement can be a barrier for those who are not specialists in RL.

Another recent addition to these set of tools is Scikit-decide[6], an AI framework for RL, which supports automated planning and scheduling. It offers an intuitive interface, but, similar to other tools understanding of RL fundamentals is still essential.

The development of RLML represents a novel approach within the RL landscape. By focusing on a higher level of abstraction, RLML seeks to simplify the use of RL with a user-friendly modelling environment. This contribution could be particularly valuable in making advanced RL technologies more approachable for everyone including non-data scientists or for users without extensive background in the field, thereby expanding the reach and application of RL in various domains.

3 MODELLING LANGUAGE DESIGN

In this section, we describe the proposed DSL for reinforcement learning, RLML. The core language concepts were designed based on the main elements representing the RL problem and solution algorithms.

3.1 RLML Abstract Syntax

Reflecting the RL domain concepts, RLML mainly consists of an *environment* element, an *agent* element, and the *result* element. Successively these elements contain all the other details involved in solving an RL problem. Similarly, the *RLMLComparator* consists of the same elements as RLML, except it can have multiple agent elements as well as corresponding number of result elements. Figure 2 presents the metamodel for RLML.

- RLML: The RLML element is the root element of all the other elements in the language, and contains the environment element, RL agent and the result. It includes properties that lets the user decide an input method and a run language method.
- RLML Comparator: This is another root element which is almost a replica of RLML and contains all the other elements in the language except RLML. It contains the environment element, multiple RL agent elements, and multiple result elements.
- Environment: This represents the RL problem environment for describing the RL problem and the goal that the agent needs to reach. It is broken down into states, actions, terminal states (also referred to as Done states) and rewards elements. Each one of these elements contains a value property, which expects a string value and have associated constraints.
- Reinforcement Learning Agent: The RL agent in the domain is represented by the RL algorithm, which will be used to to solve the RL problem, given by the RL environment.
- Reinforcement Learning Algorithm: It is specialized into the many different RL algorithms which can be chosen and implemented to solve the RL problem. It holds the settings property with reference to the required settings and parameters to tune an RL algorithm. All the child RL algorithms will inherit the settings property. The settings element carries the common RL algorithm parameters. A specific type of RL algorithm can have its own specific properties.

  The algorithms currently covered in the language include Q-Learning, SARSA, Monte Carlo, Actor Critic and DQN all of which fall under model-free RL. The metamodel is easily extensible and can support addition of more RL algorithms as the language matures, to cover more tests cases and broader RL problems.
- Settings and Hyperparameters: The settings element contains hyperparameters, which include all the common properties for the selected RL algorithms. The hyperparameters element contains alpha (the learning rate), gamma (the discount factor), epsilon (specifies the exploration rate), and total episodes (total number of episodes to train the agent). More parameters can be added for specific algorithms in their individual concepts.

[2]https://intellabs.github.io/coach/

[3]https://tensorforce.readthedocs.io/

[4]https://www.deepmind.com/open-source/trfl

[5]https://www.tensorflow.org/agents

[6]https://airbus.github.io/scikit-decide/guide/introduction

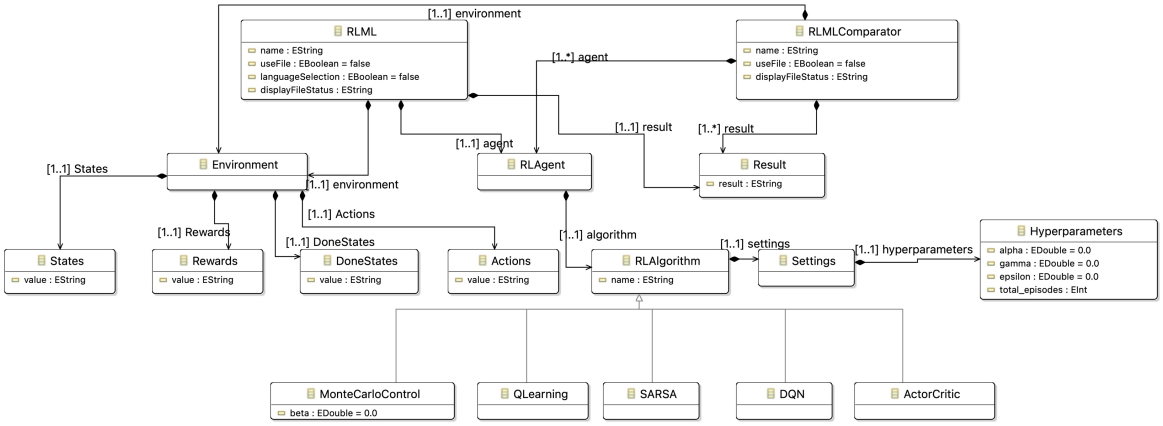

Figure 2: RLML Metamodel.

- Result: The result element contains a result string property, and is used to display the results of running the chosen algorithm.

3.2 RLML Concrete Syntax

One of the goals of RLML is to reduce the complexity involved in implementing RL applications. RLML uses textual concrete syntax and can be modelled as a simple configuration-like properties file, as shown in the sample model in Fig. 4.

The inspiration of the RLML concrete syntax comes from YAML representation, which is a human-readable data format used for data serialization. It is used for reading and writing data independent of a specific programming language. Another significant aspect for this concrete syntax is its relevance in model-free algorithms, where a dynamic state-action space is required for the agent's actions. This interaction-focused approach is key in model-free reinforcement learning, allowing the agent to effectively learn and refine its strategy through direct experience, even in complex and variable scenarios. As per the abstract syntax, the model needs to specify the project's name, the environment element properties (the states, actions, rewards and terminal states), the agent's RL algorithm type and the settings for that algorithm.

3.3 RLML Constraints

The property values are considered valid when they are in a format that RLML can use to implement the chosen RL algorithm. To ensure that the user is entering valid properties, we defined the following DSL validation constraints.

- States Constraint: States property value is expecting a string representation of all the possible states an agent can move within the current world or the environment of the current task. The value of the states property must be a comma-separated list of state strings, within square brackets. The individual state names cannot have comma or spaces.
  Valid example: [A, B, C, D, E, F]
- Actions Constraint: The possible actions that the agent can take for each state of the states array. This value is also in string format and expects a two-dimensional array of indexes. The array of indexes contain the index values of the states that the agent can go to, starting from the given state. Each array is a comma-separated list within square brackets. The constraint validates the format of the provided string value and checks that the length of actions array element is equivalent to the length of the states array. In the valid example below, we can see that there are six arrays of indexes to match the length of the example array for states.
  States example: [A, B, C, D, E, F]
  Valid Example: [[1,3], [0,2,4], [2], [0,4], [1,3,5], [2,4]]
- Rewards Constraint: The rewards property value is similar to the actions property value. In this case, the two-dimensional array contains an array of rewards that the agent will receive when moving from the given state to other states in the environment. The RL algorithm will eventually learn to move towards the states that give maximum future rewards and ignore the ones that do not give rewards. Each array is a comma-separated list within square brackets. Similar to actions value validation, the rewards constraint validates the format of the string and checks that the length of the rewards array is equal to the length of states array and the length of individual rewards elements, is equivalent to

the length of the states array. In the valid example below, there are also six arrays of six reward values to match the length of states example array.
States example: [A, B, C, D, E, F]
Valid Example: [[0,0,0,0,0,0], [0,0,100,0,0,0], [0,0,0,0,0,0], [0,0,0,0,0,0], [0,0,0,0,0,0], [0,0,100,0,0,0]]

- Terminal States Constraint: In the RL domain, the terminal states is a subgroup of all the states that can end a training episode, either because it is the goal state or because it is a terminating state. Therefore, the terminal states array should provide a smaller string array than the states array. The terminal states constraint ensures the format of the string value provided is a comma-separated list within square brackets and checks that this array is a sub-array of the states example array.
States example: [A, B, C, D, E, F]
Valid Example: [C]

4 DOMAIN-SPECIFIC MODELLING ENVIRONMENT

This section describes the proposed RL framework.

4.1 RLML Features

A modelling environment has been designed and developed to create RLML models. Translational semantics have been implemented to support execution of the models through the environment. The modelling environment supports use of different agents as well as displays the output of the RL training.

Our framework provides support for the same algorithms in both Java and Python programming languages, thus maintaining algorithmic uniformity. Concurrently, efforts were made to enhance Java's RL capabilities, ensuring it remains a viable option for those preferring or requiring it. Our balanced approach enhances the project's overall utility, catering to the diverse needs of the RL community and maintaining inclusivity across programming preferences.

Our environment provides support for saving trained RL models, thus facilitating the retention and subsequent utilization of these models and offering researchers a valuable resource. Additionally, we developed support for running multiple algorithms simultaneously, presenting data for each distinct variation. This functionality not only allows users to compare and analyze different algorithmic approaches side by side but also facilitates a deeper understanding of how variations in parameters affect outcomes (see Fig. 5). It provides a robust platform for experimentation, enabling users to efficiently identify the most effective algorithms and parameter settings for their specific use cases. This multi-algorithm capability greatly enhances the tool's utility in complex scenarios, making it an invaluable asset for both research and practical applications in diverse fields where nuanced algorithmic comparisons are essential.

Recognizing the complexity of RL inputs and the impracticality of manual entry in some cases, we have enhanced RLML with the capability to import values through a text file. This feature allows users to select a file (see Fig. 6), which is then processed to ensure it contains valid data. Upon confirmation of valid input, the system automatically populates the *States*, *Actions*, *Rewards*, and *Done States*. This addition significantly enhances the versatility of the tool, making it suitable for use cases that involve large input sizes.

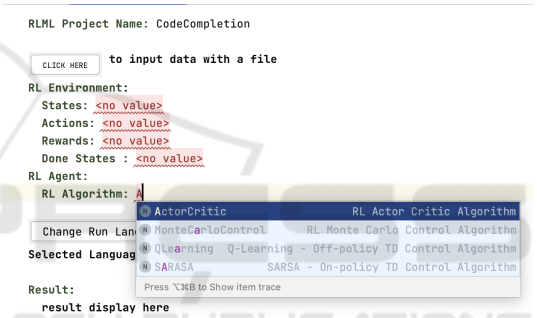

Figure 3: RLML Environment: Code Completion.

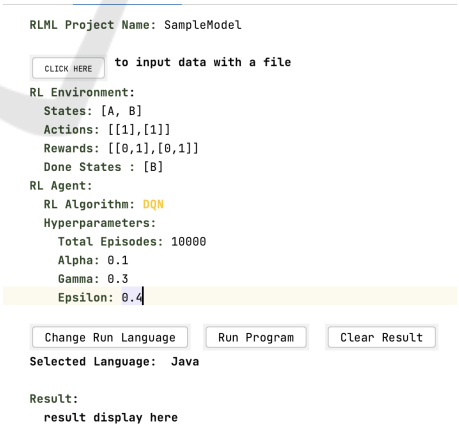

Figure 4: Sample RLML Model.

4.2 RLML Editor

MPS is a language workbench which provides a tool or set of tools to support language definition, and it implements language-oriented programming. MPS

is an integrated development environment (IDE) for DSL development, which promotes re-usability and extensibility. The language definition in MPS consists of several aspects: structure, editor, actions, constraints, behaviour, type system, intentions, plugin and data flow. Only the *structure* aspect is essential for language definition and the rest are for additional features. These aspects describe the different facets of a language.

We have employed the structure, editor, constraints, and behaviour aspects in the RLML definition. The structure aspect defines the nodes of the Abstract Syntax Tree (AST), known as concepts in MPS. The editor aspect describes how a language is presented and edited in the editor. It enables the language designer to create a user interface for editing their concepts. Constraints describes the restrictions on the AST. Finally, the behaviour aspect enables creation of constructors for the node.

- *RLML Structure:* The structure aspect contains the concepts that represent the RLML metamodel. Each concept consist of properties and children, reflecting the properties and the relationships in the RLML metamodel, shown in Fig. 2.
- *RLML Editor:* The RLML editor aspect is configured to define RLML's concrete syntax, as described and illustrated earlier in Fig. 4. The concept editor for the RLML root element contains the "Click Here", "Browse File", "Change Run Language", "Run Program" and "Clear Result" buttons (see Fig. 4). The "Click Here" button toggles the visibility of "Browse File" option which opens the file selection dialog. The "Change Run Language" option allows users to switch between Python and Java code. The "Run Program" executes the generated code in MPS. When the code is run in MPS, the results are displayed in the editor. Finally, the "Clear Result" button resets the displayed output to blank. The setup provides a user-friendly environment and enables code execution and results display right in MPS.

  With the support for automatic code completion in MPS, the environment shows suggestions as the user creates the RLML model. The code completion feature helps the RLML user to see the list of available RL algorithms (refer to Fig. 3) and choose the one which can solve the targeted RL problem.
- *RLML Constraints:* The validation constraints are implemented using MPS's concept constraints. For each defined structure concept, we can develop a concept constraint to validate it. RLML constraints aspect reflect RLML's constraints (explained in Sec. 3.3), which are the actions, rewards, states and terminal states constraints.
- *RLML Behaviour:* The behaviour aspect lets us set the default values for each algorithm concept, as well as the default *run* language (currently Java).

  The sandbox solution in MPS facilitates implementing the developed language and holds the end user code. Figure 4 shows an example of an RLML model in MPS.

4.3 RLML Code Generation

This work aims to provide abstractions to reduce the complexity associated with RL problems and algorithms by generating runnable code from the RLML models. Generators define possible transformations between a source modelling language and a target language, typically a general purpose language, like Java or Python.

For our proposed language, we implemented the model to code transformation to generate code from RLML models. We have used a root mapping rule and reduction rules for our code generation. While Java is directly supported by MPS, it is limited in generating code for Python. Since support for the generation of Python code is highly desirable in the RL domain, we utilize an open-source MPS module[7] for this purpose. This module allows us to extend MPS's capabilities to generate Python code, applying similar model-to-text transformations as with Java. This integration enhances the versatility of our tool, supporting a wider range of programming languages and accommodating a broader user base.

- Root Mapping Rule: RLML's generator module contains two root mapping rules, one for the RLML element and other for RLMLComparator element, which are the root elements of RLML. The rule specifies the template to transform the RLML element or RLML concept in MPS, into a valid general purpose language class with fields and methods corresponding to those in RLML element's properties and children.
- Reduction Rule: The generator module contains reduction rules for all supported algorithms. Supporting more RL algorithms simply means extending the language with additional RL algorithm concepts and their reduction rules/transformation rules. However, it is important to note that since Python does not have native support in MPS, the reduction rules cannot be used. To extend an algorithm in Python,

[7]https://github.com/juliuscanute/python-mps

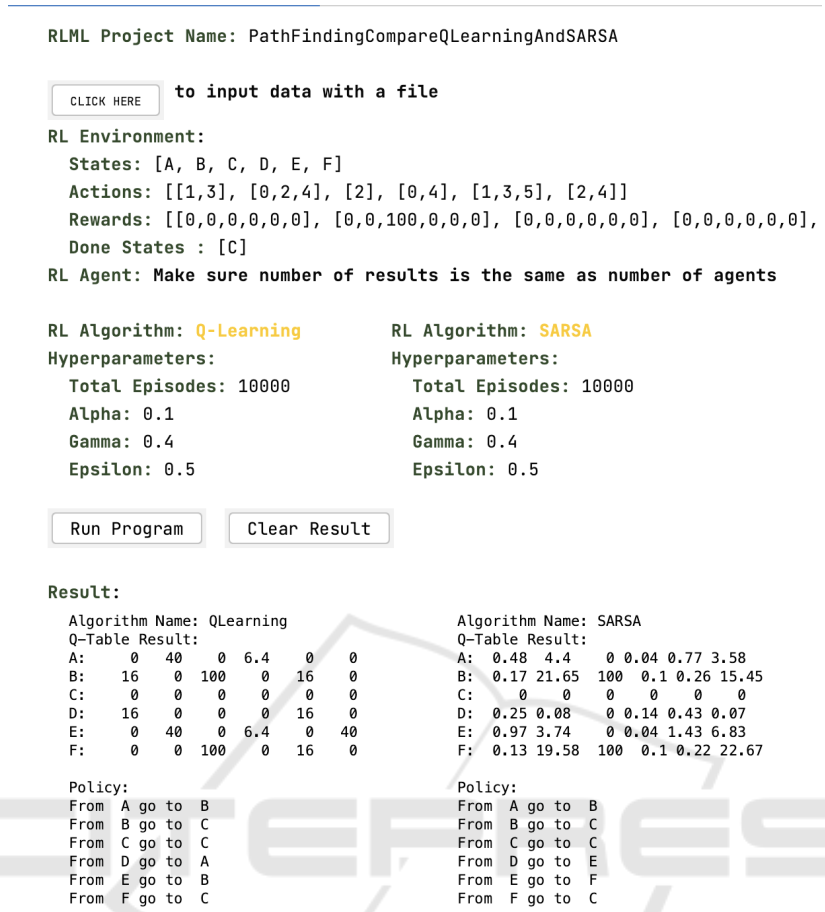

Figure 5: RLML Environment: Comparator.

we need to add a new function definition to the `mapRLMLmain.py` file. As part of our contributions, we are building a Java library of RL algorithms, to be used with MPS. While Python implementations are typical for RL, developers and students with Java expertise will find such a library quite beneficial.

Using these transformation rules, MPS can transform an RLML model to runnable code. The generated file contains more than 1500 lines of code for Java and about 300 lines for Python. This emphasizes the simplicity offered by RLML. The name of the file will be mapped to RLML element's project name property, and it contains a method called `run` which implements the RL algorithm calculations based on the reduction rules for Java or the function definition for Python defined earlier.

With regards to the user experience, we have integrated UI elements like buttons in the RLML editor and enabled file importation with data validation. Addressing MPS's inability to support Python, we devised innovative solutions for Python code generation and execution within MPS by running the python code as Java process. These enhancements not only streamline the RLML user experience but also broadly benefit MPS's community, particularly in modelling language engineering.

A video demonstrating the RLML modelling environment is available at https://cs.torontomu.ca/~sml/demos/rlml.html.

4.4 Discussion

Most machine learning libraries are widely available as Python libraries and not as Java libraries, hence it was challenging to find Java libraries to support RL algorithms. For the few available libraries, they were not fully supported by MPS. We were able to overcome this challenge by implementing algorithms in Java ourselves and using an open source MPS module to support Python code generation. Apart from these algorithms, we did not implement any RL algorithms, but instead relied of tried and tested implementations of RL algorithms and focused on enabling non-technical users to leverage existing implementa-

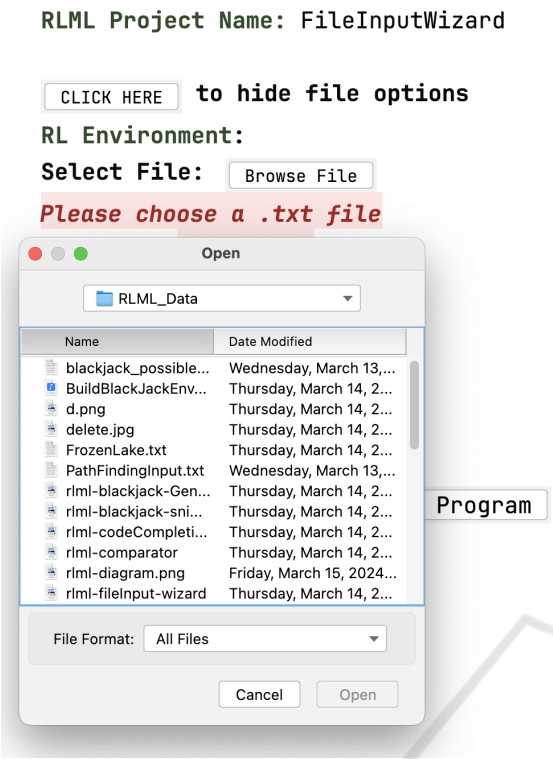

Figure 6: RLML Environment: Text File Input Wizard.

tions of RL algorithms easily through our framework.

Another issue was, RL problems do not have fixed input format from the perspective of actions, rewards and states. Therefore, it was not straightforward to come up with a format for those inputs. More validation and mapping is needed for broader problem coverage. This could be fixed by using more model-based algorithms where the agent is given an environment it can interact with to solve the problem.

Addressing the challenge of handling large data set inputs, our tool offers a feature for data input through a file. This method is particularly useful for adding large inputs efficiently. However, it is important to note that this feature requires the text files to adhere to a specific format (as defined in the concrete syntax of the language). Consequently, users need to either manually create these files or generate them specifically for use with this tool, considering the format requirements. In our tests, we were able to use a large language model (LLM) (Radford et al., 2019) along with precise prompts to generate these files.

We added support to save the RL model which can be useful for researchers to reuse the trained model. However, the limitation of the *save* feature is that it cannot be reused if the original parameters (States, Actions, Rewards, or Done States) are changed. Effectively, reusing the saved model would be the same as increasing the number of episodes to train.

In practical RL applications, selecting appropriate states and actions can be quite complex, especially in real-world scenarios. Future extensions will focus on enhancing the language syntax to allow users more flexibility. This includes the ability to define custom actions that better reflect the complexities encountered in real-world situations. We also plan to incorporate features that let users specify probability distributions for state transitions based on different actions. By allowing for more detailed and realistic modelling of state transitions and actions, our framework will be better suited to tackle the nuanced and often unpredictable nature of practical RL applications.

5 RL APPLICATIONS

We validated our framework with four well-known applications from the RL domain: path finding, blackjack, simple game, and frozen lake(Ravichandiran, 2018). Due to space constraints, we only present two of the applications here. For details on the use of RLML on the simple game and frozen lake applications, please refer to (Sinani, 2022). The artifacts are available at https://github.com/mde-tmu/RLML. Implementation of the Monte Carlo and DQN algorithms is currently work in progress, hence the validation does not cover these algorithms.

5.1 Path Finding Application

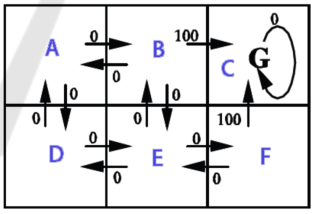

Figure 7: Path Finding Environment.

The path finding problem (Verma et al., 2020) is a common application in the machine learning domain that can be solved with different algorithms, including RL. In the path finding environment, the agent's goal is to learn the path to a target state, starting from a randomly selected state (see Fig. 7). There are in total six states in this application, represented by the alphabetical letters A to F. On each episode, the agent starts in a random state and takes actions to reach the goal state, C. Once the agent reaches the goal state, the episode will be considered complete. The agent will repeat the training episodes spcific number of times, as configured in the RL algorithm. In an RLML model, this is set as the total episodes in the RL algorithm entity.

At the end of the training, the agent will learn the best path to the goal state C, starting from the random initial state. The agent learns the path to the goal state by updating what is referred to as *Q-Table* and aims to calculate the optimal action value function that can be used to derive the optimal policy.

The RL environment needs to be modelled in a format that conforms to the RLML abstract and concrete syntax (see Sec. 3). We model the path finding environment as states, actions, rewards and terminal states arrays, as shown in Fig. 5. Next, the path finding application environment variables and RL algorithm option needs to be selected in the RLML model. Sample RLML model instances for the path finding algorithm are shown in Fig. 4.

The source code is automatically generated from the RLML model for each selected algorithm. At a high level, it is a Java/Python file named according to the RLML root element name, e.g., `PathFindingQLearning` (similar to the code in Fig. 10). It contains a methods to implement, run and print the results of the chosen algorithm.

The RLML environment contains the `Run Program` button (see Sec. 4.2). Once we click on the `Run Program` button, the environment is dynamically updated with the calculated results and we can see the result of running the program within the environment. The *Q-Table* and policy are dynamically calculated and displayed. This can be viewed in the modelling environment in the *Results* section (see Fig. 8). The policy, derived from *Q-Table* values, shows the preferred action the agent will make at each state. As seen in the results, the agent learned to go to state B from state A, to state C from state B, and so on. Overall, the agent learned the shortest path to go to the target state, which is C.

So far the application was implementing the Q-Learning algorithm, however we can easily substitute the algorithm with another algorithm, such as, SARSA or Actor Critic algorithm. The difference between Q-Learning, SARSA and Actor Critic implementations is minor at the RLML level. RLML only shows the algorithm type and hyperparameters necessary for each algorithm to run. However, it will handle the details of the algorithm calculations during code generation and based on that, it will produce the valid results. As can be seen in Fig. 9, in all cases, the agent successfully learns the path to the goal state.

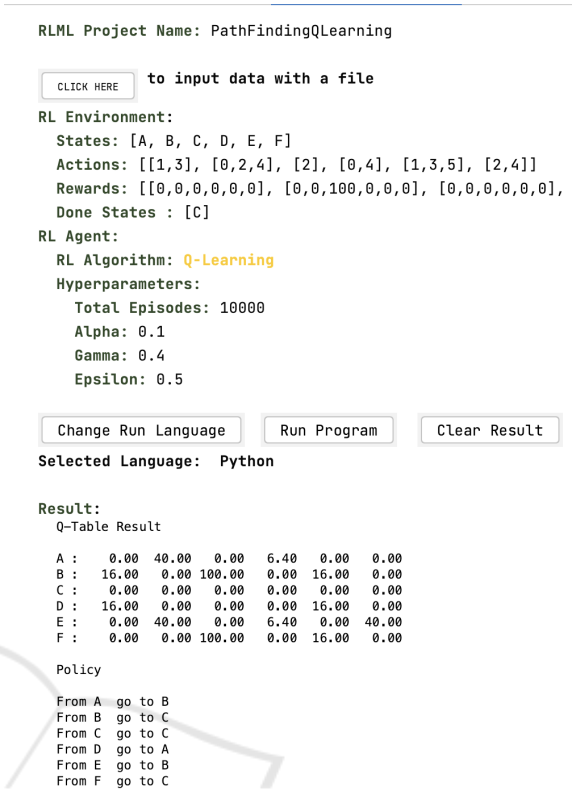

Figure 8: Path Finding Results in RLML with Q-Learning.

5.2 Blackjack Game

The Blackjack game is a prominent application in the machine learning domain, solvable through various algorithms, including those from reinforcement learning. In the Blackjack environment, the objective of the agent is to master decision-making strategies to maximize winnings, beginning from an initial hand. This involves understanding when to hit or stand, based on the current hand and the dealer's visible card, aiming to attain a hand value as close to 21 as possible without exceeding it. Given the concrete syntax of the language, going to any state but the current state counts as a *Hit* action, whereas staying at current state counts as a *Stand*.

Since, the game represents a real world application the state space is quite complex consisting of around 460 states. In each episode, the agent starts with an initial hand in Blackjack, taking actions based on the hand's value and the dealer's visible card, with the aim of optimizing its strategy for winning. The goal in this context is to make decisions that maximize the agent's chances of beating the dealer without exceeding a hand value of 21. Each completed hand is an episode, and the agent undergoes numerous episodes as defined in the RL algorithm's settings. Through training, the agent learns the best decision-making strategy for Blackjack, starting from any given hand. It achieves this by updating the Q-Table, with the ultimate aim of determining the optimal action-value function to derive the optimal policy. Similar to the pathfinding application, the RL

```
RLML Project Name: PathFindingCompareQLearning_SARSA_ActorCritic

CLICK HERE to input data with a file
RL Environment:
  States: [A, B, C, D, E, F]
  Actions: [[1,3], [0,2,4], [2], [0,4], [1,3,5], [2,4]]
  Rewards: [[0,0,0,0,0,0], [0,0,100,0,0,0], [0,0,0,0,0,0], [0,0,0,0,0,0], [0,0,0,0,0,0],[0,0,100,0,0,0]]
  Done States : [C]
RL Agent: Make sure number of results is the same as number of agents

RL Algorithm: Q-Learning          RL Algorithm: SARSA           RL Algorithm: ActorCritic
Hyperparameters:                  Hyperparameters:              Hyperparameters:
  Total Episodes: 10000             Total Episodes: 10000         Total Episodes: 10000
  Alpha: 0.1                        Alpha: 0.1                    Alpha: 0.1
  Gamma: 0.4                        Gamma: 0.4                    Gamma: 0.4
  Epsilon: 0.5                      Epsilon: 0.5                  Epsilon: 0.5

Run Program   Clear Result

Result:
  Algorithm Name: QLearning
  Q-Table Result:
A:     0   40    0  6.4    0    0
B:    16    0  100    0   16    0
C:     0    0    0    0    0    0
D:    16    0    0    0   16    0
E:     0   40    0  6.4    0   40
F:     0    0  100    0   16    0

  Policy:
  From  A go to  B
  From  B go to  C
  From  C go to  C
  From  D go to  A
  From  E go to  B
  From  F go to  C

  Algorithm Name: SARSA
  Q-Table Result:
A:  0.82 3.46    0 0.03 0.92 1.48
B:  0.13 14.84  100  0.1 0.21 11.23
C:     0    0    0    0    0    0
D:  0.21 0.06    0  0.1 0.41 0.08
E:  0.72 4.58    0 0.03 1.36 5.21
F:  0.09 19.7  100 0.04 0.19 17.08

  Policy:
  From  A go to  B
  From  B go to  C
  From  C go to  C
  From  D go to  E
  From  E go to  F
  From  F go to  C

  Algorithm Name: ActorCritic
  Q-Table Result:
A:  0.82 0.01    0 0.03 0.92 1.48
B:  0.13 14.84 10.01  0.1 0.21 11.23
C:     0    0 0.01    0    0    0
D:  0.01 0.06    0  0.1 0.41 0.08
E:  0.72 4.58    0 0.01 1.36 0.01
F:  0.09 19.7  100 0.04 0.01 17.08

  Policy:
  From  A go to  D
  From  B go to  C
  From  C go to  C
  From  D go to  E
  From  E go to  B
  From  F go to  C
```

Figure 9: Path Finding Results in RLML: Q-Learning VS SARSA VS ActorCritic.

environment needs to be modelled in a format that conforms to the RLML abstract and concrete syntax. In line with that, we can generate and run code for a Blackjack game (e.g., named *BlackjackQLearning*) using the RLML editor's "Run Program" button. Examining the calculated Q-Table and derived policy reveals how the player has learned optimal decision-making strategies in the Blackjack environment. In the Q-Table, specific actions in certain states may have negative values, reflecting decisions that typically lead to losing hands.

The algorithm learns to avoid actions that historically result in losses, even if they seem initially appealing. The player, or agent, has been trained to prioritize decisions that maximize overall gains over time, rather than immediate, riskier gains, as indicated by the positive and negative rewards in the Q-Table.

```
116
117     public void run() {
118       {
119         // Q-learning: When we update the Q(St, At), we will choose the A(t+
120         // biggest. But when we get to state S(t+1), we have the probability
121         // For example, if its policy is Epsilon-Greedy algorithm, then in s
122         // with the probability = (1 — epsilon) + (epsilon / k), in contrast
123
124         final double alpha = 0.4;
125         final double gamma = 0.2;
126         boolean done = false;
127         Random rand = new Random();
128
129         // Train episodes
130         for (int i = 0; i < 10000; i++) {
131
132           // For each episode: select random initial state
133           int state = rand.nextInt(statesCount);
134
135           done = false;
136           // Do while not reach goal state
137           while (!(done)) {
138
139             // Select one among all possible actions for the current state
140             // Selection strategy is random in this example
141             // Action outcome is set to deterministic in this example
142             // Transition probability is 1
143             int index = rand.nextInt(actions[state].length);
144             int action = actions[state][index];
145
146             int nextState = action;
147             int r = rewards[state][action];
148
```

Figure 10: Blackjack with Q-Learning: Generated Code with RLML.

6 CONCLUSION

In our work, we applied MDE in the machine learning area to develop an RL environment for non-technical

users, making it easier for a wider audience to leverage RL's potential. Our no-code solution allows users to quickly build and test RL models without extensive programming skills, hence enabling faster prototyping and experimentation. RLML is developed to be easily extensible to support a wide range of RL algorithms. To the best of our knowledge, this work is a first step in this direction for reinforcement learning.

With the use of the language workbench MPS, we built a domain-specific modelling environment supporting model editing, syntax checking, constraints checking and validation, as well as code generation. RLML achieves the abstraction needed in RL applications, by providing a configuration-like model to provide input values of the RL problem environment and a choice of the RL algorithm. From that point, our framework can generate executable code, run it and display the results. The environment also provides a comparator to compare results obtained with different RL algorithms. It supports both Java and Python implementations.

We demonstrated the use of our proposed framework with the path finding and blackjack RL applications. It can also be used for business applications as well as to get feedback from RL users at different levels of expertise. Moreover, RLML can be helpful in academia for making reinforcement learning accessible for non-technical students.

This work is a starting point towards developing a framework for supporting various types of RL technologies, both model-free and model-based, with the ultimate goal of democratizing access to advanced AI capabilities. We are currently working on incorporating probability distributions and custom actions into RLML, which will allow it to model real world use cases more effectively.

ACKNOWLEDGEMENTS

This work has been partially supported by Natural Sciences and Engineering Research Council of Canada (NSERC) and Toronto Metropolitan University. The authors would like to extend their thanks to Prof. Nariman Farsad for his feedback on this work.

REFERENCES

Brunton, S. and Kutz, J. (2019). *Data-Driven Science and Engineering*. Cambridge University Press.

Bucchiarone, A., Cabot, J., Paige, R., and Pierantonio, A. (2020). Grand challenges in model-driven engineering: an analysis of the state of the research. *Software and Systems Modeling*, 19:1–9.

Ding, Z., Huang, Y., Yuan, H., and Dong, H. (2020). Introduction to reinforcement learning. In *Deep Reinforcement Learning*, pages 47–123. Springer Singapore.

François-Lavet, V., Henderson, P., Islam, R., Bellemare, M. G., Pineau, J., et al. (2018). An introduction to deep reinforcement learning. *Foundations and Trends® in Machine Learning*, 11(3-4):219–354.

Graesser, L. and Keng, W. L. (2019). *Foundations of Deep Reinforcement Learning: Theory and Practice in Python*. Addison-Wesley Professional, Boston.

Jiang, H. (2021). *Machine learning fundamentals: a concise introduction*. Cambridge University Press.

Liaskos, S., Khan, S. M., Golipour, R., and Mylopoulos, J. (2022). Towards goal-based generation of reinforcement learning domain simulations. In *iStar*, pages 22–28.

Meacham, S., Pech, V., and Nauck, D. (2020). Classification algorithms framework (CAF) to enable intelligent systems using jetbrains MPS domain-specific languages environment. *IEEE access*, 8:14832–14840.

Naveed, H., Arora, C., Khalajzadeh, H., Grundy, J., and Haggag, O. (2024). Model driven engineering for machine learning components: A systematic literature review. *Information and Software Technology*, 169:107423.

Radford, A., Wu, J., Amodei, D., Amodei, D., Clark, J., Brundage, M., and Sutskever, I. (2019). Better language models and their implications. *OpenAI blog*, 1(2).

Ravichandiran, S. (2018). *Hands-On Reinforcement Learning with Python: Master Reinforcement and Deep Reinforcement Learning Using OpenAI Gym and TensorFlow*. Packt Publishing, Limited, Birmingham.

Sapio, F. and Ratini, R. (2022). Developing and testing a new reinforcement learning toolkit with unreal engine. In *Artificial Intelligence in HCI*, pages 317–334, Cham. Springer.

Schmidt, D. C. (2006). Model-driven engineering. *Computer-IEEE Computer Society*, 39(2):25.

Sinani, N. (2022). RLML: A domain-specific modelling language for reinforcement learning. MRP Report.

Sutton, R. S. and Barto, A. (2018). Reinforcement learning: An introduction. *A Bradford Book*.

Verma, P., Dhanre, U., Khekare, S., Sheikh, S., and Khekare, G. (2020). The optimal path finding algorithm based on reinforcement learning. *International Journal of Software Science and Computational Intelligence*, 12:1–18.

Voelter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L. C. L., Visser, E., and Wachsmuth, G. (2013). *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. CreateSpace Independent Publishing Platform, US.

Zhao, T., Huang, X., and Cao, Y. (2017). DeepDSL: A compilation-based domain-specific language for deep learning. In *Proceedings of the 5th International Conference on Learning Representations,*.