Jadx Environment Documentation

Ildeniz Sekban

Institute for Program Structures and Data Organization (IPD)
Advisor: Dipl. Ing. Daniel Zimmermann

1 Introduction

Jadx Environment is a gym environment, which mimics the GUI of the java decompiler software jadx. It looks and behaves like a normal window application; however, it provides no additional functionality apart from displaying its GUI. A user can interact with the environment only by using mouse clicks. After every mouse click action, the environment returns a user-specified reward and an observation, which is an RGB image displaying the state of the GUI. The goal is to maximize rewards by clicking all buttons at least once.

1.1 Prerequisites

- Jadx Envrionment is completely written in Python and requires a Python version of 3.8.0 or higher.
- The observations, RGB images containing the GUI-State, are handled as matrices, and represented internally by Numpy arrays.
- Pygame module is used for visualizing the observations by drawing the observation Numpy arrays on the screen.
- Gym is also required as Jadx Environment implements the Env interface of the Gym to act as a Gym Environment.
- OpenCV library is used when taking a snapshot of the GUI-State with the render() method.

The version requirements of these packages can be found in requirements.txt.

2 Usage

2.1 Operation Modes

There are two ways of using Jadx Environment:

1. Interactive Mode

   In this mode the environment is displayed on the user's screen and the user can interact with the environment by using a mouse. Pygame is used for capturing the mouse clicks of the user as well as displaying the observations coming from the environment, so this mode requires that Pygame is installed. Using this mode, a user can inspect how the environment looks and how it behaves.

   Interactive mode can be started by running main.py script. Before starting the environment, the display size can be adjusted by changing DISPLAY_SIZE value in main.py script. It accepts a tuple with two elements, first element being the width of the display while the second one is the height.

   The size of the display should not be confused with the size of the environment which is fixed and equal to 400x268. Changing the DISPLAY_SIZE merely changes the size of the display and has no effect on the environment size, that is why at display sizes greater than 400x268 the images become distorted.

   While in interactive mode, the user can perform a mouse click with the left mouse button. After each mouse click the reward is displayed on the command line interface. If the user clicks on the close button at the top right corner of the environment (not the close button of the Pygame window) the environment resets and an array showing which buttons were clicked is displayed on the command line. Figure 1 shows the sequence diagram of the main Pygame loop.

2. AI Mode

   Alternatively, Jadx Environment can be integrated into an AI algorithm by instantiating it and using the interface methods which are described in the section below. The main use case of this model would be to training of an AI algorithm on this environment. Your algorithm should accept a 400x268x3 matrix as input as the observations are outputted in this format. The script main.py includes a code in which the environment is initialized, and mouse click actions are performed.

2.2 Jadx Environment Interface

Below are the methods through which the environment is used. These methods are also used by Pygame in Interactive Mode, and they should be used by the AI algorithm in AI Mode as well.

- step()

  This method accepts a 2-dimensional Numpy array as click coordinates and performs a single click action. It returns:

  - an observation, which corresponds to the RGB image of the next GUI-State as a 400x268x3 Numpy array
  - an integer reward
  - done status, which is equal to 1 if the training episode is over, 0 else

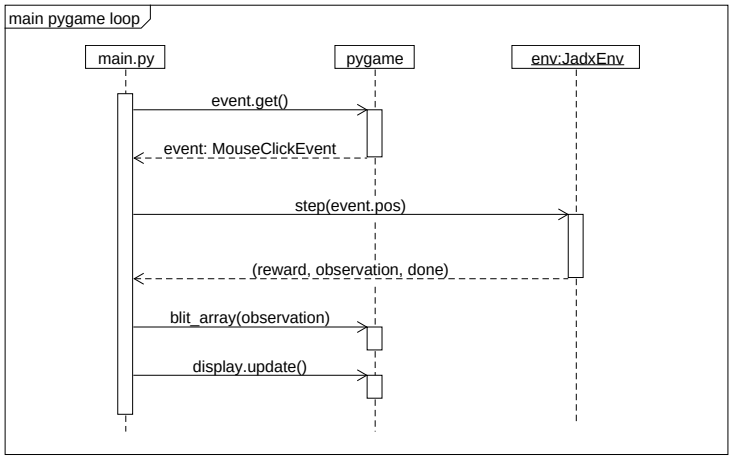

Figure 1: Pygame Loop Sequence Diagram

- reset()

  Resets the environment to an initial state and returns an initial observation

- render()

  Draws the current observation on the screen using OpenCV

- get_progress()

  Returns a Numpy array with a length equal to the number of all buttons in the environment. It shows which buttons are clicked and which buttons are not.

3 Implementation

3.1 GUI Components

Under the hood JadxEnv uses its own component system while creating its GUI structure. Figure 2 shows a simplified UML class diagram of this component system. Abstract class Drawable is the parent class of all components. Any future component should also inherit this class.

The abstract class CompositeDrawable (which is itself a Drawable) on the other hand describes a Drawable which can contain other Drawables as its children. Any such component to be implemented in the future (such as Panels), should inherit the CompositeDrawable class. CompositeDrawable stores its children in a List and the order of the List also dictates the order in which the children are drawn; children with lower indices being at the bottom.

Each concrete component class should also have the following attributes in order to be drawn by the environment:

1. Coordinates relative to the parent CompositeDrawable

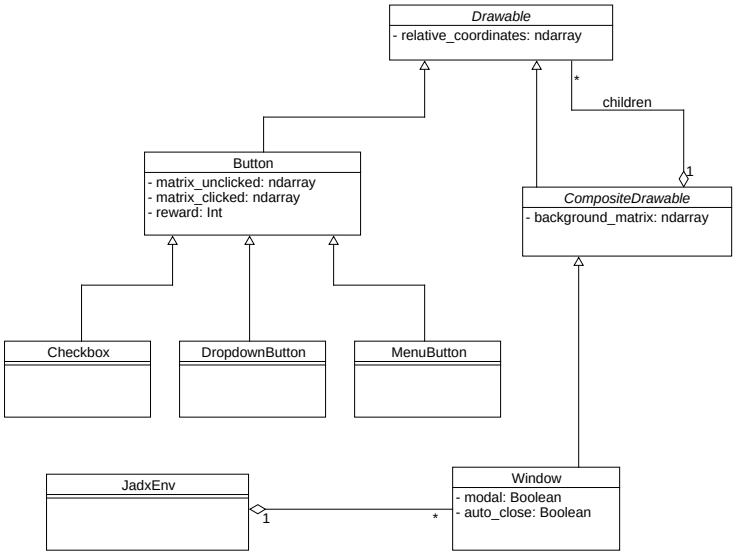

Figure 2: UML Class Diagram of the Component System

2. An RGB image (Numpy array), which depicts how the Drawable should look when drawn

Some components (e.g., Button) may have multiple images for each of its states. These components should decide which of their images should be used at any given time. Two components are important in understanding the UI logic of the environment:

1. Button

   Buttons are the only component which can increase the reward when clicked on. They can have two RGB images, one for when the Button is in "clicked" state and the other one for the "unclicked" state. They can also be resettable. A resettable Button's clicked state is reset, every time the Window containing the Button is closed.

2. Window

   At the moment Window is the sole CompositeDrawable implemented. A window can be modal, which means while the modal window is active, no other Drawable behind that Window can be interacted with. A non-modal window can also be an auto-close Window. An auto-close Window closes when the user clicks on anywhere outside the Window. Dropdown Menus for instance, are implemented as non-modal auto-close Windows.

3.2 Reward System

As it was described above, right now Button is the only component which can increase the reward when clicked on. Each Button has a reward attribute (see Figure 2), which holds an integer value. This value corresponds to the potential reward that can be earned from that specific Button. The rule for acquiring rewards is as follows:

- Given Button B and the corresponding reward value X_B of Button B, the reward is equal to
  - X_B, if Button B is clicked for the first time
  - 0, else

Thus, following a click action, one can only receive a reward, when a Button is clicked for the first time. Subsequent clicks or clicks which did not land on any Buttons receive a reward of 0.

The aim of the agent acting in this environment is therefore, maximizing the rewards by clicking on each and every Button at least once.

3.3 JadxEnv Class

The class JadxEnv is where the environment is implemented. It implements the interface Env of the gym Python module. This way it acts as a Gym environment.

JadxEnv knows its own components through storing a List of Windows. In the initial state there is only a single Window in the list, the main Window. Whenever a new window is opened or closed a new Window is appended to the list or removed from the list accordingly. The order of this list dictates the ordering of the Windows as well. A Window with a low index is located under Windows with a higher index.

JadxEnv comes with a predefined GUI structure out-of-the-box, however the user can create his/her own GUI structure by using the components from the component system. The GUI structure is defined in JadxEnv.__init_components() and it can be changed according to the needs of the user. There are two things to consider when changing this method:

1. Each Button created should also be added to the JadxEnv.__all_buttons List, so that the status of the Button can be shown in the the JadxEnv.get_progress() method
2. At the end, the main Window should be returned so that JadxEnv can draw its initial GUI

While creating the GUI the user should also give each component a relative position and an RGB image as a Numpy array. The images in the out-of-the-box GUI were created by taking screenshots of the real Jadx software. That is why the out-of-the-box GUI looks similar to the GUI of the Jadx.

3.4 Blit Operation

Blit is an operation which is frequently used while drawing the GUI and has a significant effect on the overall performance. It accepts two matrices, called source and destination and a 2-dimensional coordinate as input and outputs a matrix with the same size as the destination matrix. The operation “blits” the source image on the destination matrix at the point P by changing the values of the destination matrix with the values of the source matrix starting from the point P. An example is shown below. When blitting an RGB image on another RGB image, the operation is done on each channel separately. Blit operation is implemented in the MatrixUtils class.

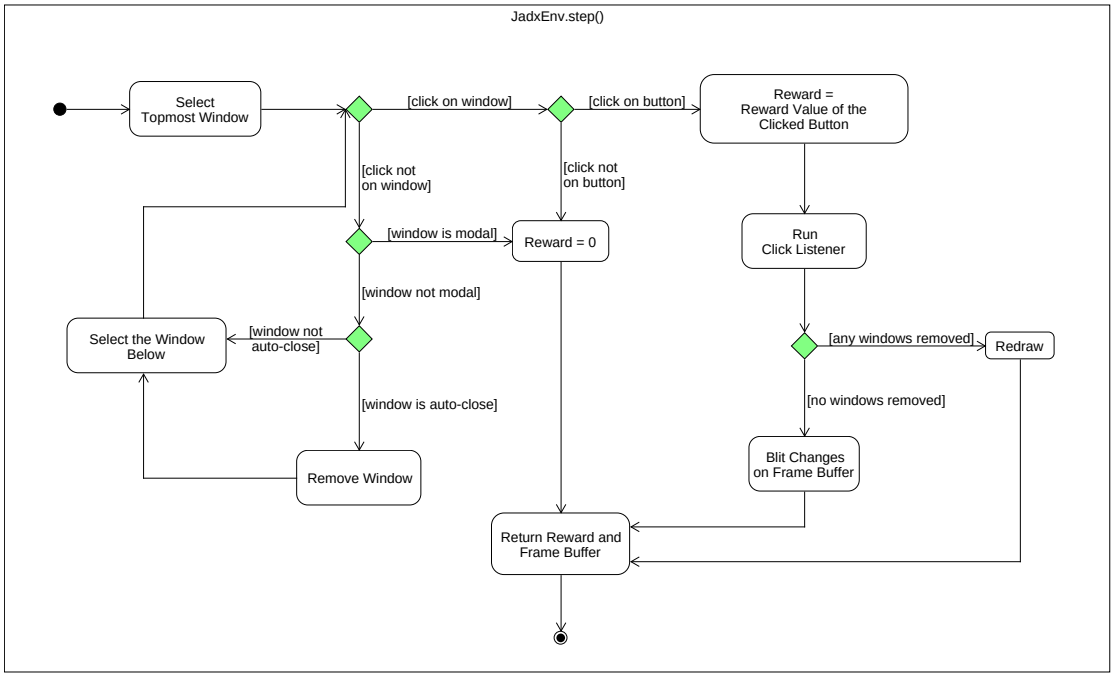

Figure 3: UML Activity Diagram of GUI Logic

3.5 Drawing

The components are responsible for drawing themselves with the draw_self method. However, how a component draws itself changes from component to component.

A Window draws itself in its draw_self method by iterating over its children and calling their draw_self method. Each Button in turn blits its image on the background image of the Window. The resulting image is then stored in the Window as its current image. That means each Window keeps a copy of its current image (apart from its background image) and is responsible for maintaining it during its life-cycle. Thus, as the environment has only a single Window at the beginning (main Window), the initial GUI image of the whole environment is the same as the current image of the main Window.

3.6 GUI Logic

In the context of Jadx Environment, GUI logic defines how the next GUI-State is calculated, given a current GUI-State and a mouse click action. JadxEnv uses a frame buffer to store its current GUI image. This frame buffer can be updated with blit operation when needed. The calculation of the next GUI-State happens in the JadxEnv.step() method. Figure 3 shows the internal workings of this method.

Starting with the topmost Window (last element of the Window list) the environment checks if the click has landed on the window by calling the click method of the Window, if not then it continues with the Window below.

When the appearance of a Button changes (e.g., unclicked -> clicked) or a new Window is added, the changes are blitted directly onto the frame buffer. The worst case happens

when a Window is closed. When that happens, the environment stacks the remaining Windows on top of each other from scratch by blitting them. Since all Windows store their most current image, these current images are directly used when stacking Windows, which means the Windows do not have to redraw themselves during stacking.

4 Evaluation

4.1 Testing

Unit testing was done by using the unittest framework of Python. Overall, there are 17 unit tests, testing the utility functions as well as the JadxEnv class. When combined, these tests have a line coverage of 98% over the whole project. However, there may still be edge cases which were not covered by these tests.

4.2 Performance

The main performance metric used when evaluating the environment was frame time; the time it takes to compute a single frame, which is roughly equal to the runtime of the step function in JadxEnv. Having a short frame time is crucial because it directly influences the training time of an agent. Frame time was measured with the Profiler of PyCharm IDE, the results are as follows:

- 0,096 ms average frame time at 1 million random clicks
- 0,21 ms average frame time at 1 million worst case clicks (opening and closing a window repeatedly)

4.3 Possible Improvements

There are possible improvements which can be implemented with a relatively small effort:

- Panels can be implemented in order to group Buttons. That way deciding if the click landed on a Button would take less time.
- GUI can be improved by taking more screenshots and creating more components, resulting with a more realistic GUI.
- Changing the visibility and activation/deactivation of Drawables can be implemented