RoboCupRescue 2024
TDP Infrastructure AIT-Rescue (Japan)

Haruki Uehara[1], Yuki Shimada[1], Keisuke Ando[1], Takeshi Uchitane[1], Kazunori Iwata[2], and Nobuhiro Ito[1]

[1]Department of Information Science, Aichi Institute of Technology, Japan
[2]Department of Business Administration, Aichi University, Japan

Abstract. In recent years, development and research using Python has been increasing. This may be due to Python's low learning curve and its rich library of algorithms and machine learning tools. The Agent Development Framework (ADF) used for agent development in RoboCupRescue Simulation (RRS) currently supports only Java, which poses a challenge for newcomers unfamiliar with the Java language. Additionally, Python boasts more extensive libraries in the fields of data science and machine learning than Java, and its development community is more active, making it possible to develop more efficient rescue agents. Therefore, we have designed and developed a prototype of the ADF for Python to lower the entry barrier for new developers in the RRS agent field and to facilitate the development of more efficient rescue agents.

1 Introduction

In RoboCup, development and research using Python have been increasing in recent years. For example, in RoboCupSoccer Simulation 2D (RSS2D), Zare et al. developed a Python-based framework called Pyrus [6]. The base code of RSS2D has been developed mostly in C++, and its complex syntax makes it difficult to use, especially for beginners. Therefore, Zare et al. developed Pyrus to abstract complex functionality and enable the use of Python's machine learning libraries, allowing Python developers to focus on high-level strategy development.

In RoboCupRescue Simulation (RRS), Goyal et al. developed an agent that aims to optimize the cooperative behavior of multiple agents using deep reinforcement learning [1]. The RRS agent environment, written in Java, and the deep learning module, developed in Python, are designed to communicate with each other. Specifically, using data format sharing via Protocol Buffers and communication via gRPC, Java-implemented agents and Python-implemented deep reinforcement learning modules work together to learn optimal behavior. This optimal behavior is learned through the collaboration between the Java-implemented agent and the Python-implemented deep reinforcement learning module.

The following features of Python are considered to be the reasons for the aforementioned needs.

- Easy to learn
  Python has a straightforward syntax, making it accessible for beginners. This lowers the barrier for newcomers to engage in research and development.
- The rich ecosystem of libraries for machine learning and data science
  Python boasts a comprehensive collection of libraries and frameworks essential for machine learning, such as Numpy, Pandas, and scikit-learn. Additionally, user-friendly libraries such as scikit-learn and Keras facilitate the application of algorithms and machine learning concepts, even for those with limited expertise in algorithms and machine learning. This fosters broader development and research in algorithms and machine learning.
- Well-documented
  Python and its libraries are extensively documented through numerous resources and books. This reduces the learning curve for new entrants, enabling efficient program development.

Currently, the Agent Development Framework (ADF) is used for agent development in RRS. However, ADF only supports Java, complicates entry for developers interested in using other languages within the RRS community. Moreover, Java offers a smaller selection of algorithms and machine learning libraries compared to Python. Integrating Python libraries presents a challenge as it requires developers to establish a data-sharing mechanism between Java and Python. Therefore, there is a current necessity to set up a development environment that accommodates Python in RRS agent development. In response, this study aims to design an agent development framework compatible with Python and developed a prototype.

2 Overview of ADF

First, the ADF used in the current agent development is described.

Initially introduced as Agent Development Framework Version1 (ADFv1) by Takayanagi et al. in 2015 [2], it aimed to address issues stemming from researchers using disparate code designs. Prior to ADFv1, each researcher developed agents independently, using their own code design, which resulted in difficulties in sharing source code and algorithms. Moreover, documentation was often lacking, requiring time-consuming efforts to decipher complex code. This high threshold hindered new entrants and complicated research in RRS. ADFv1 attempted to mitigate these challenges by providing a manual, template package, and build tool, thereby easing the burden of agent program development and code sharing. However, there is a practical issue with the document-sharing method. Additionally, there were challenges such as not supporting precomputation, inadequate algorithm organization, and lack of unified communication modules.

Then, Takami et al. published ADF Version 2 (ADFv2) in 2016 [5]. ADFv2 addresses the shortcomings of ADFv1. Firstly, regarding document-sharing methods, ADFv1 used pdf while ADFv2 employs MediaWiki. This shift allows easier collaboration among developers, enhancing documentation quality. Secondly, regarding the issue of precomputation, ADFv2 introduces the following modes: No precomputation mode, precomputation execution mode, and precomputation complete mode. Thirdly, to address insufficient algorithm organization, ADFv2 divides algorithms into modules, facilitating easier inheritance and portability.. This modular approach also supports targeted algorithm development. Finally, ADFv2 modularizes the Communication Library (CommLib) to unify communication modules, making it easier to change communication modules.

3 Purpose and design

3.1 Purpose

As outlined in Chapter 1, there is an increasing demand for Python. The current ADF is supported only by Java; hence, there is a necessity for an ADF for Python (ADF-Python). The specific objectives of developing ADF-Python are:

- Lowering the barrier for new entrants
  By leveraging Python's ease of learning, newcomers can develop agents using only Python knowledge, thereby reducing the threshold for entry into agent development.
- Utilization of Python libraries
  Harness Python's extensive libraries for algorithms and machine learning to develop agents capable of more efficient rescue operations.
- Integration with existing Java code assets
  Facilitate the reuse of existing Java source code, leveraging previous assets and eliminating the need to rewrite code from Java to Python.

In addition to the aforementioned objectives, to ensure the framework' s continuity and practicality, we aim to meet the following criteria.

- Ease of maintenance
  High maintenance costs require significant manpower and time, potentially hindering continuous provision of ADF-Python to new entrants By reducing the maintenance costs, we alleviate operational burdens and enable sustainable long-term maintenance.
- Comparable performance to current ADF
  If ADF-Python' s performance is reduced compared to the current ADF, existing developers may face compatibility issues and increased hardware requirements, raising entry barriers for new users. Therefore, eliminating performance differences between the current ADF and ADF-Python, ensures compatibility with existing hardware specifications.

3.2 Design of the framework

We designed ADF-Python based on the objectives and conditions described in Section 3.1. The design diagram of ADF-Python is illustrated in Fig. 1.

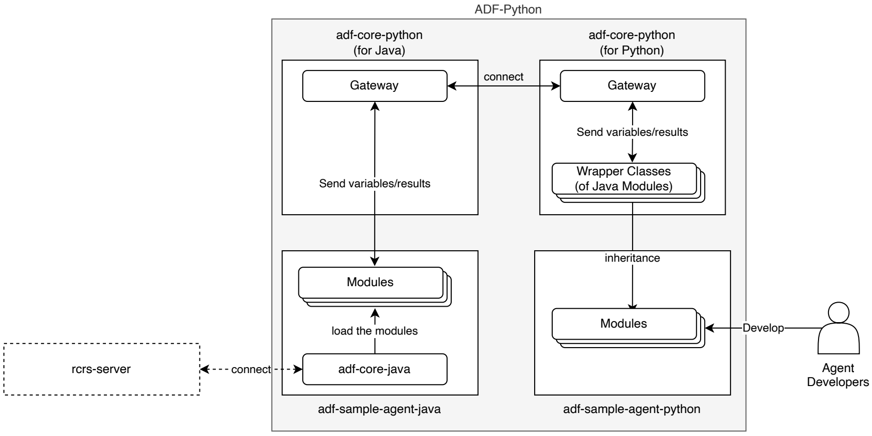

Fig. 1: Design diagram of ADF-Python

We have designed ADF-Python to function as a wrapper library for the current ADF. In addition to enabling agent development using Python, as outlined in Section 3.1, there is also a requirement to leverage existing Java source code assets. To meet this requirement, the adf-core-java [4] or Java modules developed by individual developers are accessible to Python-using agent developers through the wrapper class. This approach allows for the utilization of existing Java source code and maximizes the effectiveness of past assets. Moreover, by using existing Java source code such as adf-sample-agent-java [3] and adf-core-java, there is no need to rewrite all the programs required for agent execution in Python, thereby reducing maintenance costs.

Adf-core-python serves as a data-sharing and wrapper library for adf-core-java, featuring separate libraries for Java and Python. Each library includes a Gateway class responsible for facilitating bi-directional data communication between Java and Python modules. This Gateway class interfaces with the Python wrapper class, enabling seamless data exchange between the two environments.

The processing procedure of ADF-Python is outlined below.

1. Adf-core-java in adf-sample-agent-java calls `Modules` specified in the config file.
2. Each `Module` invokes the `Gateway` class from adf-core-python (for Java), the Java library for adf-core-python, and sends necessary information, such as values and function names to the `Gateway` class from adf-core-python (for Python), the Python library for ADF-Python.
3. The `Gateway` class from adf-core-python (for Python) receives values, functions, etc. and forwards them to respective wrapper classes.
4. Each `Module` in adf-sample-agent-python inherits the wrapper class from adf-core-python and performs operations.
5. Optionally, each `Module` in adf-sample-agent-python returns a result value to corresponding `Module` in adf-core-java through a wrapper class.

6. Repeat from 2 until the simulation concludes.

In 4, the Python-side processing is separated into concrete classes within the wrapper class and adf-sample-agent-python. This separation ensures that changes to the `Gateway` class or the adf-core-java class only require modification to the wrapper class code, minimizing the impact on the concrete classes within adf-sample-agent-python. Additionally, by providing an abstract interface in wrapper class, modules in adf-sample-agent-java and adf-core-java can be easily managed, effectively lowering the threshold for agent development.

4 Implementation of the prototype

Based on Section 3.2, we developed a prototype of ADF-Python to verify its feasibility as a wrapper library for ADF-Java. A diagram of the ADF-Python prototype is depicted in Fig. 2.

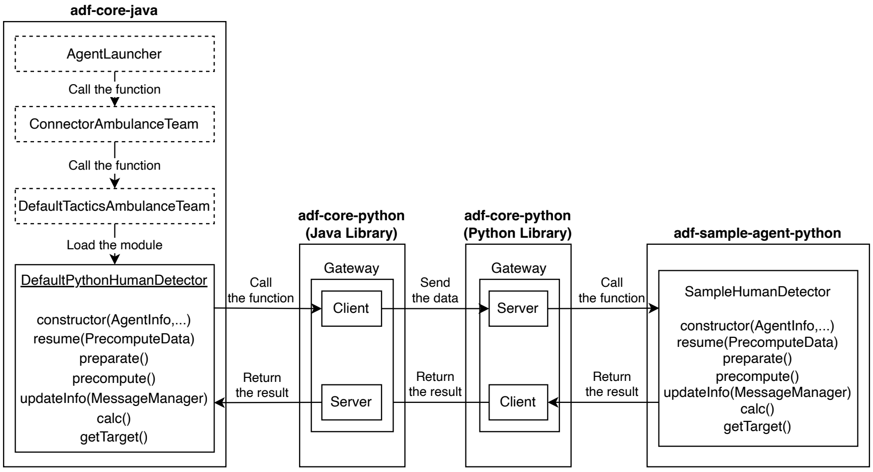

Fig. 2: ADF-Python prototype configuration diagram

In this prototype we have implemented only the `HumanDetector` for AmbulanceTeam. This decision was made because the `HumanDetector` program directly influences the agent's behavior, allowing for straightforward evaluation of its intended functionality. The communication protocol chosen is gRPC, and the data format utilized is Protocol Buffers. These were selected for their ability to facilitate communication across different programing languages and to handle complex data types, which are essential for the representing a wide variety of RRS data.

Data exchange between Java and Python follows a client-server model. Within adf-core-python, the `Gateway` package includes the `Client` and `Server` classes. The `Client` class sends necessary information to the `Server` class, enabling data reception and function execution.

The procedure for initializing `HumanDetector` in the prototype ADF-Python is shown below:

1. When the agent program starts, an instance of the `Server` class from the `Gateway` package is created and started.
2. `DefaultTacticsAmbulanceTeam` creates instances of `DefaultPythonHumanDetector`.
3. During the execution of the constructor of `DefaultPythonHumanDetector`, an instance of the `Client` class from the `Gateway` package is created.
4. Once the `Client` instance is initialized, it retrieves the constructor arguments from `DefaultPythonHumanDetector`.
5. The obtained argument data is serialized using Protocol Buffers.
6. The serialized data is sent to an instance of the `Server` class in the Python library adf-core-python.
7. The `Server` instance deserializes the submitted data.
8. The `Server` instance instantiates the `SampleHumanDetector` class from adf-sample-agent-python and assigns the deserialized data from step 7 as arguments to the constructor.

Next, the processing during the simulation of `HumanDetector` in the prototype ADF-Python is described. In `DefaultPythonHumanDetecotr`, the following functions are called by `DefaultTacticsAmbulanceTeam` at any steps during the simulation.

- `precompute` function
  Functions performed when simulation is in precompute mode
- `preparate` function
  Functions performed when the simulation is not in precompute mode
- `resume` function
  Function to get the result data of the `precompute` function executed in the precompute mode
- `updateInfo` function
  Function that retrieves information such as the message from the previous step
- `calc` function
  Function to select citizens to transport
- `getTarget` function
  Function that returns the result of the calc function

In this prototype of ADF-Python, when each function is executed, the corresponding function within the Python instance is invoked, and the process defined within each function is executed.

The Java and Python instances are assigned the same ID, as illustrated in Fig. 3. These IDs are generated during the initialization of the Java-side HumanDetector and are shared with the corresponding Python-side HumanDetector. When a Java instance calls a Python instance, such as the precompute or calc function, only the Python instance assigned the same ID as the Java instance will execute. Therefore, even if multiple Java instances are generated, only the Python instance with the matching ID will execute.

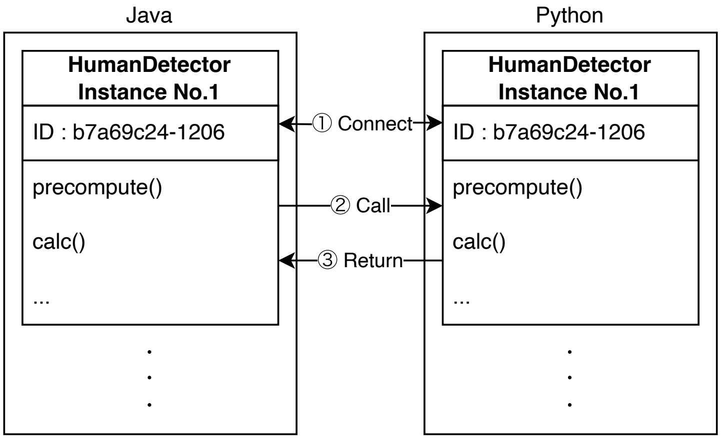

Fig. 3: Synchronizing Java and Python Instances

5 Evaluation and consideration

In this chapter, we verify that the agent implemented based on the design described in Chapter 4 is operational. Additionally, we compare its performance and implementation with ADF-Java. ADF-Python only implemented the AmbulanceTeam's `HumanDetector` by Python, and its processing procedure mirrored that of ADF-Java. The Cluster module, called as an external module within `HumanDetector`, remains implemented in Java. For comparison with ADF-Python, we used adf-sample-agent-java for ADF-Java. The simulation was performed using the computer configuration listed in Table 1

Table 1: Computer configuration used in this experiment

Operating System	Ubuntu 22.04.4 LTS
Processor	Intel(R) Core(TM) i9-14900K (24 Cores and 32 Threads)
Memory	128 GB

In addition, the scenarios in Table 2 were used in the experiments.

Table 2: Scenarios used in the experiment

Map	Area(km^2)	Number of buildings	Number of roads	Existence of blockades	Number of AT	Number of FB	Number of PF
Algiers	1.7	430	1867	Yes	25	40	25
Berlin	3.3	1426	3385	Yes	50	35	35
Eindhoven	3.0	1308	5172	Yes	30	40	40
Istanbul	1.5	1244	3337	Yes	15	30	30
Joao	1.0	879	3467	Yes	30	35	15
Paris	1.0	1618	3025	Yes	28	46	10
Sakae	1.7	626	1182	No	25	0	0
SF	1.0	815	2720	Yes	17	30	37
VC	0.3	1263	1954	No	20	30	0
Vernon	1.5	760	1281	Yes	30	40	25

Initially, we used the scenarios listed in Table 2 to validate the consistency of agent behavior between ADF-Java and ADF-Python. Across all maps, the actions of each agent remained consistent. For example, Fig. 4 displays the travel paths of a specific AmbulanceTeam in Vernon. It is evident from Fig. 4 that the travel paths generated by both ADF-Java and ADF-Python exhibit consistency.

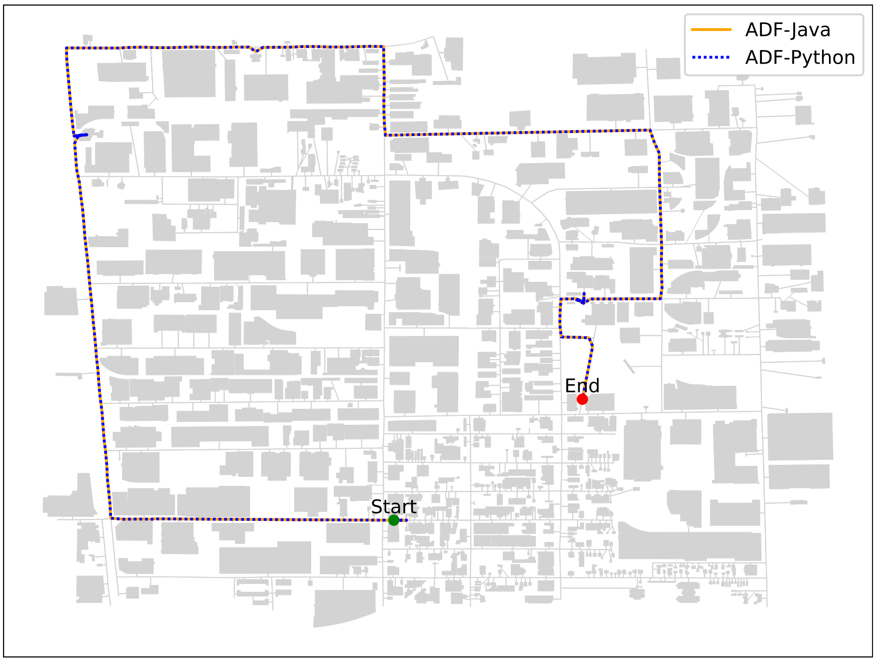

Fig. 4: The travel paths of a certain AmbulanceTeam in Vernon

We also compared the score evolution between ADF-Java and ADF-Python using the scenarios in Table 2. The results indicate consistent scores across all maps and each step. For instance, Fig.5 illustrates the score progression of ADF-

Java and ADF-Python in Paris. As shown in Fig.5, the score transitions between ADF-Java and ADF-Python are identical.

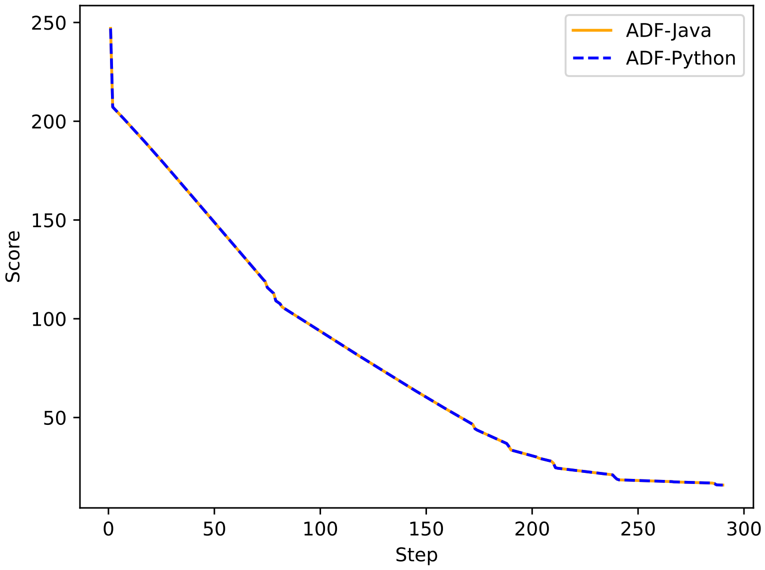

Fig. 5: The score transition between ADF-Java and ADF-Python in Paris

From this, it was confirmed that ADF-Python can reproduce the behavior of agents by writing the same algorithm as ADF-Java in Python.

Next, we compared the CPU and memory usage during the simulation of ADF-Java and ADF-Python. Table 3 and Table 4 show the CPU and memory usage of ADF-Java and ADF-Python in each scenario, respectively.

Table 3: Average CPU usage for ADF-Java and ADF-Python

Scenario	**Agent**	
	ADF-Java (%)	**ADF-Python (%)**
Algiers	8.04	10.97
Berlin	20.83	26.13
Eindhoven	16.32	19.94
Istanbul	11.96	17.43
Joao	16.01	22.41
Paris	15.69	21.88
Sakae	3.04	6.78
SF	13.96	17.88
VC	6.27	17.33
Vernon	8.62	11.56

Table 4: Average Memory Usage for ADF-Java and ADF-Python

Scenario	**Agent**	
	ADF-Java (MB)	**ADF-Python (MB)**
Algiers	9304.33	4021.31
Berlin	11077.70	9511.84
Eindhoven	11631.79	10220.95
Istanbul	9784.32	6246.05
Joao	10130.72	5912.14
Paris	9139.03	7521.18
Sakae	3188.70	1841.93
SF	9527.85	5227.68
VC	6457.22	3853.04
Vernon	9881.41	4695.84

First, comparing the CPU usage of ADF-Java and ADF-Python in each scenario, we observe that ADF-Python's CPU usage is higher than that of ADF-Java. As an example, Fig. 6 shows the CPU usage of ADF-Java and ADF-Python in Vernon. The orange line represents ADF-Java CPU usage, and the blue line represents ADF-Python CPU usage. Note that we have omitted the first 50 s after the agent’ s launch, as the values are high. As shown in Fig. 6, ADF-Python exhibits higher CPU usage compared to ADF-Java.

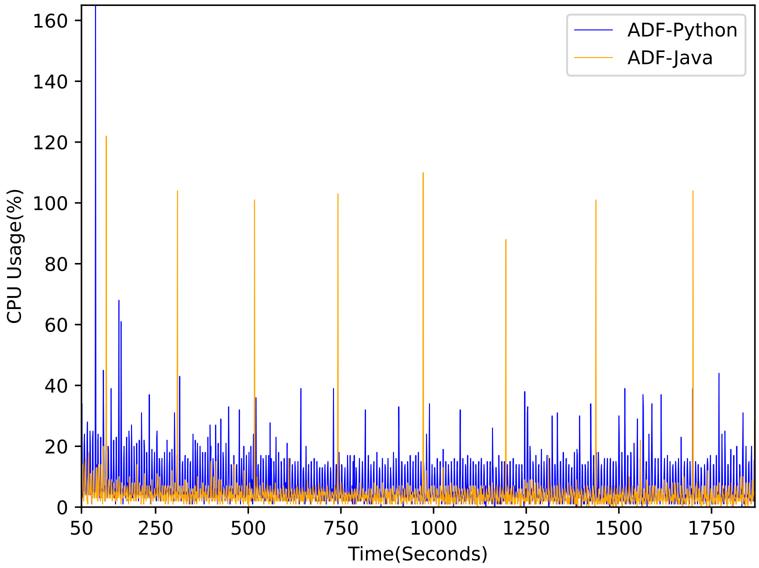

Fig. 6: ADF-Java and ADF-Python CPU usage in Vernon

This increased CPU usage in ADF-Python is attributed to the simultaneous execution of Java and Python programs.

Next, we compare the memory usage of ADF-Java and ADF-Python in each scenario and find that ADF-Python uses less memory than ADF-Java. For example, Fig. 7 shows the memory usage of ADF-Java and ADF-Python in Vernon. The orange line represents the memory usage of ADF-Java, and the blue line represents the memory usage of ADF-Python. Note that we omitted the 50 s after the agent' s launch, as the values are high. As shown in Fig. 7, ADF-Python demonstrates lower memory usage compared to ADF-Java.

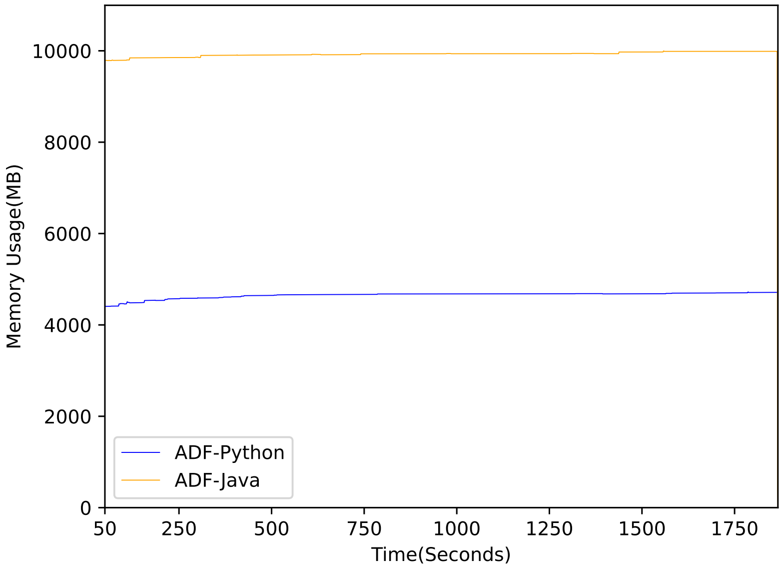

Fig. 7: ADF-Java and ADF-Python memory usage on Vernon

We believe that Java's memory management is the cause of these differences. When an object is created in Java, the memory heap size that the object will use is predicted and reserved. Consequently, more memory heap size might be reserved than the object actually requires. In contrast, Python dynamically reserves memory heap size as needed, resulting in lower memory usage than Java.

Fig. 8 and Fig. 9 illustrate these differences. Fig. 8 shows the memory heap size and actual heap size used by ADF-Java in Vernon. Fig. 9 presents the memory heap size and actual heap size used by Java running on ADF-Python in Vernon.

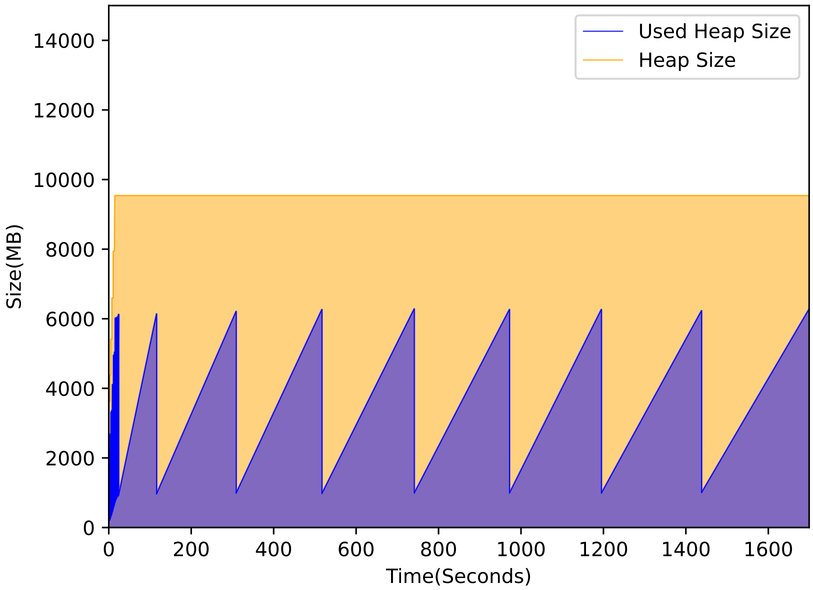

Fig. 8: Heap size for ADF-Java in Vernon

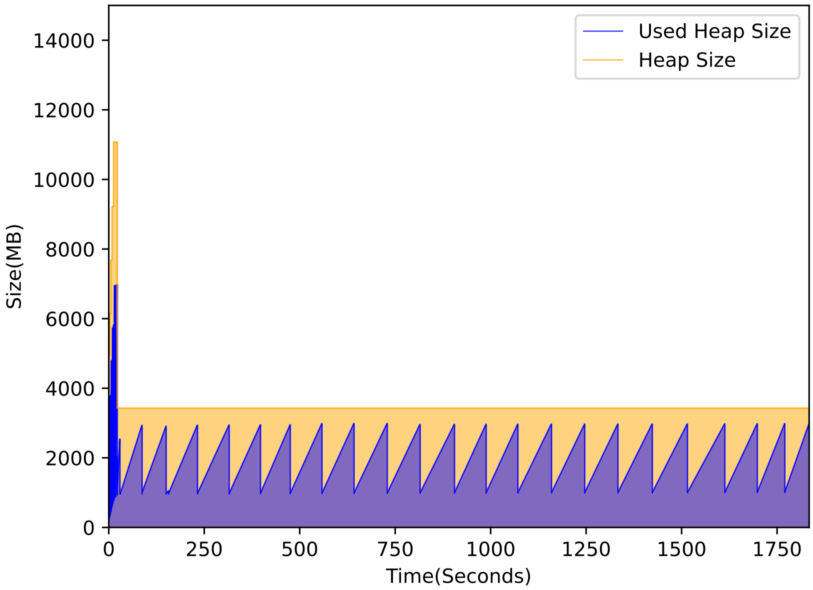

Fig. 9: Heap size used for Java running on ADF-Python in Vernon

In the figures, the orange line shows the maximum memory heap size, and the blue line shows the actual used heap size. We can see that the maximum memory heap size of ADF-Java's adf-sample-agent-java is approximately 10 GB, and the actual heap size used is approximately 6 GB. On the other hand, the maximum heap size of adf-sample-agent-java running on ADF-Python is approximately 4 GB, and the actual heap size used is approximately 3 GB.

This demonstrates that agents written partly in Python use less memory than agents written entirely in Java.

6 Conclusions

In this paper, we identified the problems with the current RRS agents and designed and developed the Agent Development Framework for Python, successfully achieving the integration of Java and Python as outlined in our objectives. The prototype demonstrated that ADF-Python can effectively run agents with components written in both languages, as evidenced by the functional HumanDetector for the AmbulanceTeam. This successful integration implies that ADF-Python can leverage Python's ease of use and dynamic memory management, potentially lowering the barrier for new developers and enhancing performance. However, the current state, where only the HumanDetector is written in Python while other agents remain in Java, highlights the transitional nature of our work. Full implementation of all agents in Python is a necessary next step. Future research will focus on developing comprehensive Python implementations for all agent algorithms, further enhancing the framework's functionality, and conducting thorough performance evaluations. By addressing these aspects, we aim to provide a robust and efficient tool for agent developers, ultimately facilitating the broader adoption and utility of ADF-Python in RRS agent development.

Acknowledgements

This work was supported by JSPS KAKENHI Grant Number JP17K00317, JP21K12039, and JP23K11225; and the Kayamori Foundation of Information Science Advancement.

References

1. Goyal, A.: Multi-Agent Deep Reinforcement Learning for RoboCup Rescue Simulator. Master's thesis, The University of Texas at Austin (2020), `https://repositories.lib.utexas.edu/items/55eb34d2-5029-4798-98fe-d9aae83249c3`
2. Kazuo, T., Shunki, T., Yoshiyuki, K., Nobuhiro, I.: Robocup rescue simulation 新規参入者のためのエージェントフレームワーク について. 人工知能学会全国大会論文集 **JSAI2015**(0), 2B5NFC02c2–2B5NFC02c2 (2015), `https://cir.nii.ac.jp/crid/1390564238001456000`
3. roborescue: adf-sample-agent-java. `https://github.com/roborescue/adf-sample-agent-java` (2016), accessed at 05/10/2024
4. roborescue: adf-core-java. `https://github.com/roborescue/adf-core-java` (2017), accessed at 05/10/2024

5. Shunki, T., Kazuo, T., Yoshiyuki, K., Nobuhiro, I.: モジュール構造を用いた災害救助研究プラットフォームの提案. 人工知能学会全国大会論文集 **JSAI2016**(0), 1I2NFC013–1I2NFC013 (2016), `https://cir.nii.ac.jp/crid/1390001288047294080`
6. Zare, N., Sayareh, A., Amini, O., Sarvmaili, M., Firouzkouhi, A., Matwin, S., Soares, A.: Pyrus base: An open source python framework for the robocup 2d soccer simulation (2023)