MASPY: Towards the Creation of BDI Multi-Agent Systems

Alexandre L. L. Mellado[1][*], **Igor Guilherme Fidler**[1],
André Pinz Borges[1], **Gleifer Vaz Alves**[1]

[1]Departamento Acadêmico de Informática
Universidade Tecnológica Federal do Paraná (UTFPR)
Ponta Grossa – PR – Brasil

{mellado, fidler}@alunos.utfpr.edu.br

{apborges, gleifer}@utfpr.edu.br

Abstract. *Integrating intelligent agents is essential to the design and functionality of numerous modern computing solutions. Several industries and research domains, from health to finance, manufacturing to customer service, are influenced by advances in the area of Intelligent Agents. Therefore, this work presents a Python library for creating systems composed of intelligent agents following the Belief, Desire and Intention paradigm. To develop systems following these characteristics, four base classes were designed. These classes, agent, environment, communication and handler, create and manage the structural part, leaving the design and specific functions to the programmer. This paper shows that, to the best of our knowledge, there is no other library in Python with the same features and functionalities as the one described here.*

1. Introduction

Multi-Agent Systems (MAS) have gained prominence in various domains [Dorri et al. 2018], including Artificial Intelligence, Robotics and Social Sciences. With the increase in the complexity and scale of such systems, there is always a need for dedicated tools to streamline the development process. By providing a set of pre-built components, methods, and communication protocols, a library enables developers to focus on higher-level system design and behaviour aspects.

Incorporation of the concept of Belief-Desire-Intention (BDI) [Bratman 1987], further adds to the benefits of one such library. This widely recognized [Georgeff et al. 1999, Abar et al. 2017] paradigm is a classic theoretical framework used mainly to represent knowledge when developing agents and autonomous systems. The BDI architecture allows agents to represent and model their decision-making and behaviour using their beliefs, desires, and intentions [Bratman 1987].

This paper presents a Python library designed to simplify the development of MAS with BDI agents, our library is called **MASPY** (Multi-Agent System for PYthon). The motivation to create the MASPY is the need for a programmable agent system to add a reinforcement learning implementation for negotiations between agents. As Python is a mainstream language with several packages designed to quickly implement reinforcement learning techniques. Our idea was to find another library with the same characteristics.

[*]O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001

Many tools, frameworks and libraries already offer the ability to program agents. This can be seen in [Kravari and Bassiliades 2015], [Pal et al. 2020] and [Cardoso and Ferrando 2021], where, combined, over thirty platforms for agent development were compared. Even though their list was not exhaustive, it shows a significant number of platforms with *Java* as the target language, while only a few utilizing *Python*, and none using the latter implementing the BDI paradigm with multi-agents.

MAPSY includes new specific functionalities and heavy focus on the versatility of the design and capabilities of the BDI-MAS. The other main objective for creating this library, specifically in Python, is the use of learning methods in developing agents using old and new libraries for machine learning made for this language. It is inspired by functionalities defined in the JaCaMo Framework [Boissier et al. 2013], composed of three languages. Jason, to develop agents, Cartago, to implement environment artifacts, and Moise, to define the system organization. Some names of variables and structures are directly referenced to some in the Jason language. The main similarities are in managing beliefs, objectives and plans following the BDI paradigm and the directives for agent communication. This type of communication is referenced as KQML (Knowledge Query and Manipulation Language), which uses performatives and directives to support information sharing between agents. Another concept JaCaMo (Moise) inspired is using roles in the environment. While they are not close to the level of organization found in Moise, the role functions as a way to define what an agent can see and do in an environment. Section 3 provides more detail on these implementations.

2. Comparison Between Programmable Agents in Python

As described in [Kravari and Bassiliades 2015], [Pal et al. 2020] and more recently in [Cardoso and Ferrando 2021] there are few frameworks developed in *Python* for the development of agents. The existent tools are *PADE* [Melo et al. 2019] and *SPADE* [Palanca et al. 2020] for behaviour agents, *MESA* [Masad and Kazil 2015] for simulation, the ethical robot from [Bremner et al. 2019] and *PROFETA* from [Fichera et al. 2017] where they implement BDI agents. As follows, we describe each one of the agent tools designed with Python.

2.1. *PADE* framework

The PADE framework presented in [Melo et al. 2019] is used to develop behavioural agents distributed over nodes in a network. This framework contains base classes to create an agent and its behaviour. To manage the MAS, PADE uses an *Agent Management System* (AMS), the first agent initiated in the system. This agent manages a table with all identifiers of active agents. Agent communication is straightforward since each agent possesses knowledge of all the participants in the system using this table. Each message follows a FIPA protocol, and its contents can contain a serializable *Python* object.

The MASPY library has similarities with PADE framework: both support the creation of MAS, provide abstractions for managing and modelling agents. Their main difference is which paradigm is used: PADE uses behavioural agents while MASPY uses BDI agents. Also, PADE does not implement the environment in its code and only considers the sensors in a physical embedding as its environment.

2.2. SPADE

SPADE [Palanca et al. 2020] aims to be a general agent development middleware. Each agent has to register to SPADE using a unique identifier consisting of the agent name, the server where it is running and a password. After that, the agent can create one or more behaviour to run. To allow communication between agents, SPADE provides a mechanism to dispatch messages to each registered agent which redirects an incoming message to the agent which can be waiting for it or relaying outgoing messages to the SPADE communication system.

Their communication system uses the eXtensible Messaging and Presence Protocol (XMPP) which is open and allows instant messaging and presence notification. It is always possible to know who is in the system and how to exchange messages with them. Using the XMPP as its communication protocol allows a SPADE agent to communicate with every XMPP server, making possible communication with other services or agents.

Although SPADE allows the execution of BDI agents, this is done using a plugin. While in MASPY, the agents are natively implemented as BDI. Besides, SPADE is ideal for systems focusing on the communication aspect or the ability to exchange information with external services. However, it does not contain an environment abstraction layer.

2.3. MESA

MESA [Masad and Kazil 2015] is a *Python* framework that allows simulation, visualization, and analysis of agent models inspired by NetLogo. It offers built-in core components such as spatial grids and agent schedulers. The different schedulers allow the control of the activation regime for each agent. The spatial components are used to model the environment where the agent is located. The agent model can act and change its current state based on its position, environment, and interaction with other agents. The communication between agents, while possible to be implemented, is not straightforward, as MESA does not offer communication utilities.

While it offers free control of created agents actions and ways to interact with a defined environment, this framework is strictly for simulation and data visualization purposes. Moreover, it does not contain an explicit communication layer necessary for a Multi-Agent System.

2.4. BDI Python

In [Bremner et al. 2019], the authors present how to embed ethical considerations into a BDI agent reasoning. This work is a proof of concept. Therefore, it does not provide a library or framework for managing MAS. Instead, it shows how a BDI agent could be implemented using *Python*. An agent's objective is to give a robot ethical reasoning in its actions. As it was only made to support a single agent, it can not be extended to a MAS and does not have a communication layer. MASPY has different goals and supports the creation of different agents while allowing them to communicate between themselves.

2.5. PROFETA

PROFETA (Python RObotic Framework for dEsigning sTrAtegies) [Fichera et al. 2017] is a programming framework designed for autonomous robots based on the BDI paradigm.

It uses metaprogramming capabilities to incorporate the operational semantics of AgentSpeak into *Python*. This allows the implementation of object-oriented and declarative constructs and, therefore, the definition of an agent's behaviour more straightforwardly.

Comparing it to MASPY, it does support the integration of BDI reasoning, followed even by the introduction of new types of beliefs, but falls short of the need for a communication layer. PROFETA is a framework for the development of a Robot. Doing so involves only one agent. Also, the real environment perceived by this robot's sensors does not need to be, and was not, implemented in a separate programmable layer.

2.6. Comparative Agent Development in Python

For the MASPY library, the generic development of MAS with BDI Agents was the main characteristic when being designed. Table 3 shows the base components used for a MAS and compares all the presented tools. In MASPY, agents manage beliefs and objectives to execute plans, following the BDI paradigm. These agents can be situated in generically created environments and communicate using performatives closely following the KQML protocol used in JaCaMo.

As shown in this section, the goal of MASPY is to be able to build all of this in a single library. While other tools offer MAS creation capabilities, none contains all layers wanted. PADE and SPADE fall short on the environment and the BDI agent. MESA is focused too much on simulation. At the same time, BDIPython and PROFETA are made with only one agent in mind and not a multi-agent system.

Table 1. Tools for agent development in Python

Name	Description	Inspiration	Agent	Environment	Communication
MASPY	**BDI Multi-Agent System**	**JaCaMo Framework**	**BDI Agent**	**General Class Abstraction**	**Inspired by Speech-Based KQML**
PADE	**Distributed Multi-Agent Network**	**JADE Framework**	**Agent Behaviours**	**Physical Embedding**	**FIPA-ACL Messages**
SPADE	**Multi-Agent Plataform**	**XMPP Instant Messaging**	**Agent Behaviours**	**External Simulation Tool**	**XMPP Server**
MESA	**Agent-Based Simulation**	**NetLogo, MASON and Repast**	**Agent Model**	**Spatial Model Interaction**	**External Messaging Tool**
BDIPython	**Ethical BDI Agent**	**Autonomous Ethical Robot**	**BDI Agent**	**Physical Robot Sensors**	**Does Not Support**
PROFETA	**Autonomous BDI Agent**	**AgentSpeak Language**	**BDI Agent**	**Physical Robot Sensors**	**Does Not Support**

3. The MASPY Library

MASPY library aims to facilitate the development of a BDI-MAS. This section presents the library classes and how they work to allow this implementation. Agents represent entities with beliefs about their circumstances, desires or goals implemented as objectives they wish to achieve, and intentions in the form of plans that guides their action towards those goals. The environment class models its namesake, simulating an agent acting with its surroundings through actions and changing facts of a non-autonomous entity. And the communication class is used to open a route for information to travel between agents and form an interlocked system. While still necessary, the handler class that exists to help the programmer configure the order and connections for agents is entirely optional. This library was created by providing an abstraction of base classes and methods to enable programmers to design and implement a MAS quickly.

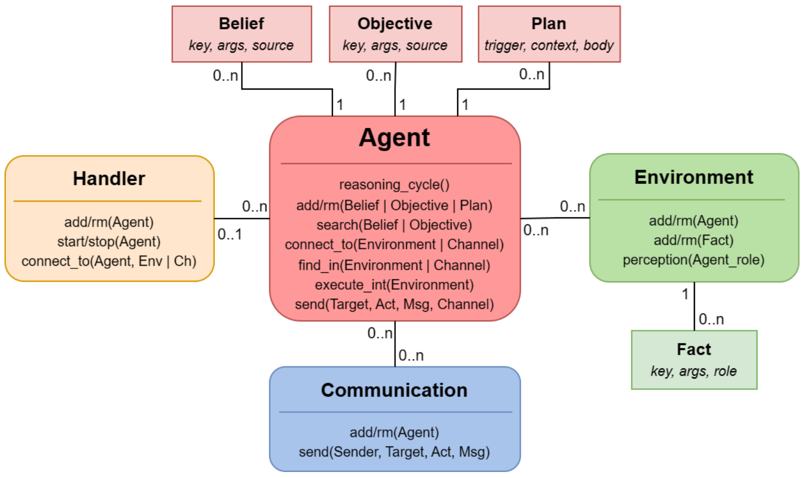

Figure 1. Diagram for the MASPY library

In practice, the system developer can use and extend the library components presented in Fig. 1. It allows the definition and initialization of agent objects, which can contain any number of beliefs, objectives and plans. These can interact with several environments and communicate through different instances of channels. Such agents and their connections to environments and channels can be configured by a single class that gives and saves distinct names for everyone, as described next.

3.1. Agent Class

The agent class is the fundamental building block of this library and the only strictly needed class to run a program. It contains the abstractions for managing an individual agent's beliefs, objectives and plans. It also has the methods to execute actions with any environment, as shown in subsection 3.2, and knows the protocol for sending and receiving messages (see subsection 3.3). The agent methods are used by being extended in classes created by the programmer. In code 1, it is presented the creation of an agent instance and the structure of methods to add a belief, objective and plan.

```python
from maspy.agent import Agent
class Sample(Agent):
    def __init__(self, agent_name):
        super().__init__(agent_name)
# For removing, 'add' is just changed to 'rm'
        self.add(belief, key, arguments, source)
        self.add(objective, key, arguments, source)
        self.add_plan([(trigger, [context], Sample.body)])
# Every plan body must contain at least self and src arguments
    def body(self, src, *args, **kwargs):
```

Code Listing 1. Instance of an Agent

Beliefs, Objectives and Plans: Each agent may have any number of beliefs, objectives or plans. The structure of beliefs and objectives are similar: both consist of a key, a variable number of arguments to store any data and the source from which this belief or objective was created. A plan, however, is formed by its trigger, the context for its activation and a body, which can be seen as an ordinary function or method in *Python*. This context can consist of any number of beliefs or objectives that the agent must have to execute the plan. The agent class provides methods for the creation and removal of each.

Reasoning Cycle: Information from the environment and communication classes can be used to update the agent's beliefs and objective bases during the reasoning cycle. These objectives depict the agent's desires, and when combined with their beliefs, create intentions that activate plans. The environment affects this through the agent's perception and communication by exchanging messages between agents. This process is presented in Fig. 2 and in Code 2. In the current implementation, each agent only considers one intention per cycle. This intention, being achieved by a plan, is executed until their end without reconsideration to new beliefs or objectives gained during it.

```python
def reasoning_cycle(self, stop_flag):
    while not stop_flag.is_set():
        self._perception()
        self._mail()
        chosen_plan, trigger = self._deliberation()
        if chosen_plan is not None:
            self._execution(chosen_plan, trigger)
```

Code Listing 2. Reasoning Cycle

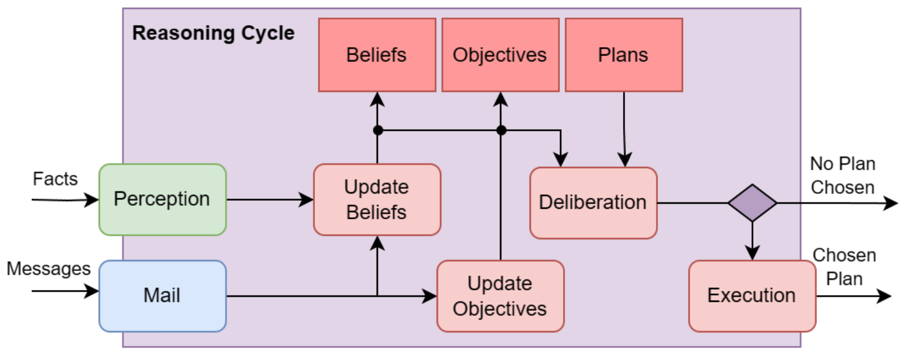

Figure 2. Reasoning Cycle

Each cycle starts by the agent perceiving all of its environments and polling all messages received from other agents, this will update its beliefs and objectives as necessary. Next, the agent deliberates which plan needs to be executed based on the available objectives and new beliefs acquired in this last cycle. In code, the reasoning cycle of each agent is started as a separate thread and is running until stopped or closing the system. This cycle can be stopped after every iteration if the *stop_flag* is set in another method.

3.2. Environment Class

The environment class represents a non-autonomous entity which agents can act upon. It can be used to model where the agent is situated, such as a street or a room. This class provides the mechanisms for agents to perceive and act upon the environment, which uses facts to model the current state, updated by available actions in this environment. A fact consists of its key, a variable number of arguments, and the role field.

Facts and Roles: For the environment, facts work like beliefs do for agents. The main difference is how and which agents can change these facts. The role field, in fact, states that only connected agents with that designated role have access to that fact. The agent's role in the environment is defined when they are first connected, but can be changed. When perceiving the environment, the agent-assigned role is compared to the one in facts to determine which ones can be transmitted to the agent, who converts these facts to beliefs after being perceived, with its source being the environment it came from.

```python
from maspy.environment import Environment
# Sample of an Environment with fact methods an action
class Sample_Env(Environment):
    def __init__(self, env_name="env"):
        super().__init__(env_name)
        self.create_fact(key, args, role)
        self.update_fact(key, new_args, new_role)
        self.extend_fact(key, added_args, added_role)
        self.shorten_fact(key, removed_args, removed_role)
# An environment action checking the agent role for execution
    def action(self, src):
        if not self.check_role(src, action_role):
            return None
```

Code Listing 3. Instance of an Environment

When trying to execute one of the environment actions, a role can be used to check if the agent can continue. Otherwise, agents without permission may still do other actions exposed by the environment that can indirectly affect those unavailable to perceive facts. The Code 4 show an example containing a sample agent connected to two different environments. In one, it enters as an "Observer", while in the other, it is connected as a "Manager". Without these roles, the presented facts, "Scoreboard" and "Quantity", would not be visible during the agent perception phase of its reasoning cycle.

```python
ag = Sample("Ag")
env1 = Sample_Env()
env2 = Sample_Env("Warehouse")
env1.create_fact("Scoreboard",(10,50), "Observer")
env2.create_fact("Quantity",{"A": 3, "B": 7}, "Manager")
ag.connect_to(Sample_Env(),"Observer")
ag.connect_to(Sample_Env("Warehouse"), "Manager")
```

Code Listing 4. Examples of fact creation with roles

3.3. Communication Class

Our library enables the exchange of messages between agents by implementing a communication class. This class contains methods for connecting the message from the sender to the receiver. By default, an agent does not wait for a response after sending a message. An answer is only expected after asking for information from another agent.

Channels: In a classical agents' communication system, only one route for messages exists. Differently in this library, each instance of a communication class with a different name is a distinct channel in which connected agents can communicate with other connected agents. There is still an implicit way of using the classical method. Agents can connect to an unnamed channel that all other agents known by default.

The creation of channels came because of two reasons: first, the ease of implementation. In python, when defining a class for communication, different instances can be independent and still function, making a clear way to abstract different routes for messages; second, just as a simulation can contain multiple environments for agents to act upon, we considered it important for communication groups to be creatable.

```python
from maspy.communication import Channel
ch1 = Channel() # Default Channel
ch2 = Channel("Crossroads") # Specific Channel
ch3 = Channel("Private") # Another Distinct Channel
```

Code Listing 5. Instances of Channels

Message Protocol: Each message has five parameters, of which four are required: the sender agent; the target agent; the type (or act) of the message; the content of the message; and an optional parameter, the channel used to send the message. Both sender and receiver have methods accounting for the different types of messages. These types always involve a transference of beliefs or objectives.

Directives: The type of message is defined as the directive of this message. It is mainly divided by way of exchanging beliefs or objectives. Agents can inform, request or ask for the contents of other agents. In code 6, three examples are presented. On line 1, a belief is sent to agent Ag1 through the "Private" channel. On line 2, a request is sent to Ag2 through the "Crossroads" channel. On line 3, an agent asks Ag3 for a belief using the default channel. On line 4, a broadcast is made, sending the belief "begin" to every agent in each channel contained in the channels list. And line 5 shows how a plan can be sent to multiple agents, by using a list, using the default channel.

```python
1 self.send(Ag1,"tell",("price_interval",[10, 20]), Channel("Private"))
2 self.send(Ag2,"achieve",("cross","right"), Channel("Crossroads"))
3 self.send(Ag3, "ask", ("channel_name",))
4 self.send("Broadcast", "tell", ("begin",), Channel_List)
5 self.send(Ag_List, "tellHow", self.plan)
```

Code Listing 6. Examples of sent messages

3.4. Handler Class

The handler is responsible for assigning unique identifiers to each agent. It can be used to determine the timing and order of initialization for the agent's reasoning cycle. It can also more intuitively connect multiple agents to multiple communication channels and environments. Only one instance of this class will be active throughout the execution of the system; however, it is not a centralizing point. All of its methods exist only for the ease of configuring the created system.

Multi-Agent System Configuration: The agent class contains the necessary methods for beginning and stopping their reasoning cycle and ways for connecting with the environment and communication channels. The job of this handler class is for a more straightforward configuration step. When multiple agents must communicate, distinct names are necessary to avoid ambiguities. The programmer can, and sometimes should, create agents with different names for better distinction, but all instances of agents in this library are given an identifier used to differentiate between agents with the same name.

The codes 7 and 8 show the difference when using the handler. In this example, 50 Sample agents are created and started after being connected to the Communication Channel "Private". While listing 7 gives a number to the instance of Sample in line 3. The handler already gives a different ID to each agent in line 1 of Code 8.

```
1 agent_list = []
2 for i in range(50):
3     ag = Sample(f"Ag{i}")
4     ag.connect_to(Channel("Private"))
5     agent_list.append(ag)
6 for ag in agent_list:
7     ag.reasoning_cycle()
```

Code Listing 7. Explicit Configuration

```
1 agent_list = Handler().create_agents(50, Sample("Ag"))
2 Handler().connect_to(agent_list, [Channel("Private")])
3 Handler().start_all_agents()
```

Code Listing 8. Handler Configuration

4. MASPY in Practice

This section shows two practical implementations as examples of how this library works. The first one highlights a simple message exchange, using context to decide between sending and receiving the message. The second example shows an interaction by an agent executing an action in an environment.

4.1. Message Exchange Example

In code 9, two instances of the same Sample Agent are created to present a message being sent between instances. The first instance is given the name "*Sender*" and adds the belief "*Sender*" along with the objective "*send_info*". The second instance is the "*Receiver*" being given the belief "*Receiver*". After creating both instances of the Sample Agent, they are connected to the default Channel and their reasoning cycle is started by the Handler.

The execution goes as follows: The Agent "*Sender*" has the "*send_info*" objective and so triggers the available plan during deliberation. For this plan to execute, it checks if the agent contains the belief "*Sender*" that it has. In this plan, first, the "*Receiver*" is located in the channel's list of agents. Then all are sent the objective to "*receive_info*". The "*Receiver*" chooses to execute the plan "*recv_info*" triggered by this new objective. This plan also only is chosen if the agent has the belief "*Receiver*" defined in its context. In this plan, the message is displayed along with the sender.

```python
from maspy.agent import Agent
from maspy.communication import Channel
from maspy.handler import Handler

class Sample(Agent):
    def __init__(self, agent_name):
        super().__init__(agent_name)
        self.add_plan([
            ("send_info",[("belief","Sender")],Sample.send_info),
            ("receive_info",[("blf","Receiver")],Sample.recv_info)
        ])
    def send_info(self, src, msg):
        agents_list = self.find_in("Sample","Channel")["Receiver"]
        for agent in agents_list:
            self.send(agent,"achieve",("receive_info",msg))
    def recv_info(self, src, msg):
        self.print(f"Information [{msg}] - Received from {src}")

if __name__ == "__main__":
    sender = Sample("Sender")
    sender.add("blf","Sender")
    sender.add("obj","send_info",("Hello",))
    receiver = Sample("Receiver")
    receiver.add("belief","Receiver")
    Handler().connect_to([sender,receiver],[Channel()])
    Handler().start_all_agents()
```

Code Listing 9. Sending and Receiving a Message

4.2. Environment Interaction Example

Code 10 shows an example with most of the available functions in the MASPY library. It consists of a crossroads environment managed by an agent that chooses when another, the vehicle, can cross by communication in a private channel. Three classes were created. The environment Crossing has a fact for the traffic light indicating that it is “*Green*” and only agents with the role “*Manager*” can perceive it. This Crossing also has an action to cross it, which shows the agent that executed the action. This example has two agents, the Cross Manager who checks the traffic light and the Vehicle that crosses the junction.

During the configuration phase beginning in line 32, each of the wanted components is initiated before the agents’ reasoning cycle starts. First, the channel “*Crossing*” and the environment “*Cross_Junction*” are instantiated. Second, both agents are given the exact name of their class. And finally, both are connected to the channel and environment, followed by starting the reasoning cycle using the handler.

After all connections and the handler starting the agents, the *Cross_Manager* is the first to act. It begins its plan *traffic_light* after perceiving the fact with the same name in the environment “Crossing” because it has the role of “*Manager*”. In this plan, the *Cross_Manager* sends an objective for *crossing_over* to all connected vehicles in the “*Crossing*” channel. The Vehicle connected starts its plan to cross after receiving this objective from the *Cross_Manager* and executes an action in the environment to cross it. The *Cross_Junction* then shows the agent executing its action.

```python
from maspy.agent import Agent
from maspy.environment import Environment
from maspy.communication import Channel
from maspy.handler import Handler

class Crossing(Environment):
    def __init__(self, env_name):
        super().__init__(env_name)
        self.create_fact("traffic_light","Green","Manager")
    def cross(self, src):
        self.print(f"Agent {src.my_name} is now crossing")

class Cross_Manager(Agent):
    def __init__(self, mg_name):
        super().__init__(mg_name)
        self.add_plan([("traffic_light",[],Cross_Manager.trf_light)])
    def trf_light(self, src, color):
        vehicles = self.find_in("Vehicle","Env","Cross_Junction")
        for vehicle in vehicles["Vehicle"]:
            self.print(f"Detected traffic light: {color} in env {src}")
            self.print(f"Sending signal to {vehicle}")
            self.send(vehicle,"achieve",("crossing_over",),"Crossing")

class Vehicle(Agent):
    def __init__(self, vh_name):
        super().__init__(vh_name)
        self.add_plan([("crossing_over",[],Vehicle.crossing)])
    def crossing(self, src):
        self.print(f"Confirmation for crossing by {src}")
        self.execute_in("Cross_Junction").cross(self)

if __name__ == "__main__":
    cross_channel = Channel("Crossing")
    cross_env = Crossing("Cross_Junction")
    cross_manager = Cross_Manager("Cross_Manager")
    vehicle = Vehicle("Vehicle")
    Handler().connect_to([(cross_manager,"Manager"),vehicle],
                         [cross_channel,cross_env])
    Handler().start_all_agents()
```

Code Listing 10. Executing Action in Environment

5. Conclusion

This paper presented the MASPY library as a way to develop BDI MAS with Python. It describes its functionalities made possible by four base classes. The agent class manages their BDI components, Beliefs, Objectives and Plans, and the logic to perceive and communicate through environments and communication channels. The environment class models surrounding spaces for the agent to interact. The communication class contains functions to connect sent messages between agents. And the handler class can be used as a configuration tool to build and start your MAS more easily.

Other tools in Python were described and compared to MASPY. While they are more complete than our library, none offer all the wanted layers of abstraction as the

MASPY library. To present this, multiple implementation samples were shown to provide practical examples of this library's work. These varied between code snippets to the complete example with a real working system, albeit very simple. In these examples, the objective was to present a functional library with a very general disposition to multi-agent system development, with more room for such design.

In future work, we plan to define some expansions for the library. The implementation of more descriptive components to methods for context in plans and checking roles in actions. A warning system for better depuration of implemented code. Making the reasoning cycle more robust and adding the direct option for working with continuous and abstract execution. Finally, the introduction to machine learning, specifically reinforcement learning, in agent reasoning.

References

Abar, S., Theodoropoulos, G. K., Lemarinier, P., and O'Hare, G. M. (2017). Agent based modelling and simulation tools: A review of the state-of-art software. *Computer Science Review*, 24:13–33.

Boissier, O., Bordini, R. H., Hübner, J. F., Ricci, A., and Santi, A. (2013). Multi-agent oriented programming with jacamo. *Science of Computer Programming*, 78(6):747–761.

Bratman, M. (1987). *Intention, Plans, and Practical Reason.* Cambridge: Cambridge, MA: Harvard University Press.

Bremner, P., Dennis, L. A., Fisher, M., and Winfield, A. F. (2019). On proactive, transparent, and verifiable ethical reasoning for robots. *Proceedings of the IEEE*, 107(3):541–561.

Cardoso, R. C. and Ferrando, A. (2021). A review of agent-based programming for multi-agent systems. *Computers*, 10(2):16.

Dorri, A., Kanhere, S. S., and Jurdak, R. (2018). Multi-agent systems: A survey. *IEEE Access*, 6:28573–28593.

Fichera, L., Messina, F., Pappalardo, G., and Santoro, C. (2017). A python framework for programming autonomous robots using a declarative approach. *Science of Computer Programming*, 139:36–55.

Georgeff, M., Pell, B., Pollack, M., Tambe, M., and Wooldridge, M. (1999). The belief-desire-intention model of agency. In *Intelligent Agents V: Agents Theories, Architectures, and Languages: 5th International Workshop, ATAL'98 Paris, France, July 4–7, 1998 Proceedings 5*, pages 1–10. Springer.

Kravari, K. and Bassiliades, N. (2015). A survey of agent platforms. *Journal of Artificial Societies and Social Simulation*, 18(1):11.

Masad, D. and Kazil, J. (2015). Mesa: an agent-based modeling framework. In *14th PYTHON in Science Conference*, volume 2015, pages 53–60. Citeseer.

Melo, L. S., Sampaio, R. F., Leão, R. P. S., Barroso, G. C., and Bezerra, J. R. (2019). Python-based multi-agent platform for application on power grids. *International transactions on electrical energy systems*, 29(6):e12012.

Pal, C.-V., Leon, F., Paprzycki, M., and Ganzha, M. (2020). A review of platforms for the development of agent systems. *arXiv preprint arXiv:2007.08961.*

Palanca, J., Terrasa, A., Julian, V., and Carrascosa, C. (2020). Spade 3: Supporting the new generation of multi-agent systems. *IEEE Access*, 8:182537–182549.