MultiAgent Systems Documentation

Release 2020-2021

Emmanuel Hermellin, Wassila Ouerdane, Nicolas Sabouret

Mar 02, 2021

CONTENTS

Multiagent systems (MAS) are currently widely used, especially for complex applications requiring interaction between several entities. More specifically, they are used in applications where it is necessary to solve problems in a distributed manner (data processing) or in the design of distributed systems in which each component has some degree of autonomy (control of processes). Some examples of applications are trading agents, drones, smart grids.

This course will begin with an introduction to the notion of agent and multi-agent systems. It will present the concepts that will lead to an understanding of what an agent is and how it can be built. Then, we will address a classical problem of MAS, namely how to model and simulate a situation through the concept of agents. The idea is to give a basis for understanding how agent-based simulations can be used as a tool for understanding human societies or complex problems and situations. Besides, we will discuss how agents can communicate and interact to solve problems, and more specifically, to make non-centralized decisions. For this, we will rely on negotiation protocols based on argumentation theory, a process of constructing and evaluating arguments (positive and negative reasons/evidence) to resolve conflicts.

CHAPTER

ONE

GENERAL PRESENTATION OF THE COURSE

1.1 The team

Emmanuel Hermellin

Wassila Ouerdane

Nicolas Sabouret

1.2 Agenda

1. Friday, March the 5th, 2021 : Introduction to MAS : definitions and implementation of a platform
2. Friday, March the 12th, 2021 : Multiagent simulation : preys and predators
3. Friday, March the 19th, 2021 : Interaction mechanisms : models and implementation
4. Friday, March the 26th, 2021 : Argumentation-based negotiation I: Practical session
5. Friday, April the 2nd, 2021 : Argumentation-based negotiation II: what is argumentation?
6. Friday, April the 9th, 2021 : Argumentation-based negotiation III: what is a negotiation protocol?
7. Friday, April the 16th, 2021 : Argumentation-based negotiation IV: Practical session

1.3 Content & Evaluation

Our objective in this course is to introduce you to some concepts and notions in Multi-Agent systems. It alternates lecture parts (in COVID context, we chose to provide you with as much information as possible to allow you to understand the course by yourself) with practical work (which means that you'll have to implement things to understand the underlying concepts).

The first three sessions will give you the basics of MAS architectures and platforms. Based on this knowledge, you will implement a simple multi-agent based simulation (session 2). After that, sessions 3 to 7 will be dedicated to a more complex project that combines argumentation and negotiation models with the MAS architecture. This will give rise to a second multi-agent based model.

The evaluation during this course will take into account: the multi-agent based simulation (session 2) and the argumentation-based negoitiation model (session 7). This evaluation will be based on your source code and a very short report of your work (see the evaluation rules on EDUNAO).

1.4 Projects: Python and implementation

All implementation in this course is done using the Python programming language (Mesa libreary). The reason for this is that students at CentraleSupélec are familiar with this language. However, it is important to note that existing multiagent platforms mostly use different high-level programming languages such as Java, C++ or C#. Indeed, these languages are more time-efficient, they better support parallel computing, network-based architectures or object-oriented models. The most popular multiagent platforms are Jade (especially for system programming) and Repast Symphony (especially for social simulation). Both use the Java programming language.

- Make sure your environment is set up with Python 3.
- Please use your favorite IDE for programming in Python. We do not care if you prefer to use Pycharm, Visual Studio, pyzo or a Jupyter notebook…, as long as you structure your work in separate files and folders (as it is suggested along the course).
- Install the mesa library:
  - in a terminal : `pip install mesa` (some of you may use pip3)
  - in a notebook: `!pip install mesa`
  - test in a python environment: `import mesa` (it should not rise an error)

Warning: The content of this website has been converted to pdf and the document can be downloaded `here`.

CHAPTER

TWO

1. INTRODUCTION TO MAS : DEFINITIONS AND IMPLEMENTATION OF A PLATFORM

2.1 Agenda – Recall

This course aims at presenting basic notions and concepts of multi-agent systems. It is organised as follows:

1. Friday, March the 5th, 2021 : Introduction to MAS : definitions and implementation of a platform
2. Friday, March the 12th, 2021 : Multiagent simulation : preys and predators
3. Friday, March the 19th, 2021 : Interaction mechanisms : models and implementation
4. Friday, March the 26th, 2021 : Argumentation-based negotiation I: Practical session
5. Friday, April the 2nd, 2021 : Argumentation-based negotiation II: what is argumentation?
6. Friday, April the 9th, 2021 : Argumentation-based negotiation III: what is a negotiation protocol?
7. Friday, April the 16th, 2021 : Argumentation-based negotiation IV: Practical session

2.2 Session 1.

This first session aims at defining and guiding you through the comprehension of the notions of *autonomous agent* and *multi-agent system* (a.k.a. MAS).We shall begin with a brief history of MAS (first part). We then present the computational definition of an agent and we question its implementation (second part). The third part of the course will introduce to the *Mesa library* for agent-based modelling.

The understanding of concepts through their implementation in Python is an important part of this session. Take the time to achieve all practical exercises to understand the difficulty of MAS programming before you switch to the Mesa library (which does all the job for you).

2.3 Some references

- Ferber, J. (1995), *Les Systèmes Multi-Agents*, InterEditions. **(French version)**
- Ferber, J. (1999), *Multi-agent systems: An introduction to distributed artificial intelligence*, Addison Wesley. **(English version)**
- Michael Wooldridge (2002), *An Introduction to MultiAgent Systems*, John Wiley & Sons Ltd.
- The AgentLink roadmap

CHAPTER

THREE

1.1. A BRIEF HISTORY OF MULTIAGENT SYSTEMS

Multiagent Systems are a part of Computer Science, at the frontier between *software engineering*, *distributed computing* and *artificial intelligence*. They appeared in the 1990s when the research and engineering questions of these three subfields came to similar solutions to deal with different problems. Understanding multiagent programming and multiagent simulation requires to have all three aspects of MAS in mind.

For each field, we give a brief history of the domain to show how MAS apperead as a breakthrough in this research area. We identify the key characteristics of MAS that issued from this field. The fourth section concludes on the definition of agents. The last section gives an overview of MAS concrete applications.

3.1 1. MAS as seen by a software engineer

Writing software using different programming languages, design principles or systematic validation models has been a research question in Computer Science since the first digital computers appeared in the 50s. Questions such as how to maximise the speed, the reusability, the readability, the security of code is at the core of software engineering. It has a long history of breakthroughs such as the abstraction of functions (reusable and parameterised pieces of code), the modularity (separation and encapsulation of pieces of code) or object-oriented programming. All these inventions aim at helping programmers deal with the ever-increasing complexity of software systems.

In the 90s, software engineering research moved from the notion of objects to the notion of *service*. The core idea is that a piece of code, possibly a thread, should be provided with some machine-understandable interface that could allow other code to use it without a priori knowledge of its operational content. Such interface defines which method to call first, what are the expected properties of the outputs given the inputs for each method, *etc*. Software engineers rely on **interaction protocols** to define how the different services should be assembled, how the information has to be exchanged between them, so as to produce some expected result. Multiagent systems were proposed in this context as a series of platforms to support the development of such service-oriented software, with coordination entities capable to interpreting protocols.

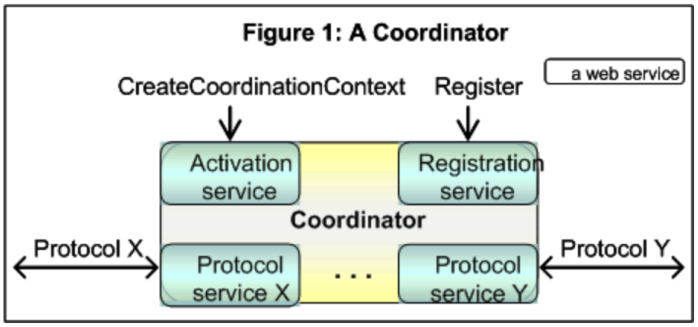


Fig. 1: The OASIS Web Services Coordination standard (version 1.1, 2007)

As a consequence, for a software engineer's point of view, what matters in multiagent systems is the possibility to **assemble separate components or threads by designing well-formed interaction protocols**. These protocols ensure that the right methods are called, that the answer are of correct type wrt the expectation of the calling method, etc. It is also important for software engineers to have a specific programming language to define the behaviour of the agents. As we shall see in sessions 4 to 7 of this course, logic-based programming and its relation to AI played a key role in this regards.

3.2 2. MAS as seen by a distributed systems engineer

Distributed computing and more generally distributed systems originally developed in the 60s but they really became popular in the 70s with the development of Arpanet (the predecessor of Internet) and distributed architectures such as client-server models in the 80s. The core idea behind distributed systems is that different processes are located on different physical systems (all connected by some network infrastructure) and interact with each other using some message passing mechanism. Actually, the *agents* (or processes) can run on one single computer. What matters is that they have *independent memory spaces* and *runtimes*. As a consequence, they cannot rely on *synchronous* interactions as is usually done with classical function calls. Multiagent systems were designed in this context to support such asynchronous runtimes.

Let us illustrate this notion with a concrete example. In the following code, the calling function waits for the `compute` function to end before the code continues. When reaching the "end of code" line, the value of `res` is guaranteed by the fact that the function was properly terminated. The whole code is synchronous.

```
def compute(x,y,z):
    ... computation here ...
    return result

def calling_function():
    ... beginning of code ...
    res = compute(1,3,'hello')
    ... end of code ...
```

On the contrary, distributed system consider situation in which several problems can occur:

- The called process can not be running, or even not be accessible in the system;
- The called process can not answer, whatever the reason;
- If an answer arrives, there is no guarantee on the delay before it arrives;

All these problems assume a perfectly functioning infrastructure: it is not about having a network failure or whatever. It is about programming a piece of software by taking into account the fact that the other half of the code is run in a completely asynchronous manner. This means two things:

1. Agents send message but don't *wait* for the answer: they must carry on their activity while a possible answer is computed by another agent;
2. Agents must use timeouts to deal with the absence of answers.

For a distributed system engineer, what matters in multiagent systems is the capacity of agent to **deal with asynchronous interactions** and possible errors. Of course, it is also important to support the distribution of the system among different physical supports but one can imagine a multiagent system that runs on a single computer (this is the case of most existing MAS architectures).

3.3 3. MAS as seen by an Artificial Intelligence engineer

Artificial Intelligence is about having machine solve problems (through computation) that human beings solve with their intelligence. To this goal, several models have been developed, from symbolic models to machine learning. CentraleSupélec students following the AI course series are familiar with many of them.

In the 80s, the mainstream approach to AI was logic-based reasoning. System experts such a MYCIN were abandoned in the early 90s, due to misplaced expectations from investors and stakeholders in the Industry. However, AI researchers were still very active in the design of action and changes. In particular, the sub-field call **planning** was very active. This domain tries to solve problems that consist in finding a sequence of actions to go from a given state to another.

The STanford Research Institute Problem Solver (STRIPS) is one of the first (and most famous) planner. One of its most important contribution is to represent actions as a pair of *preconditions* and *effects*, wich are set of first-order logic propositions. Here is a simple example (that can also be found on Wikipedia):

```
Initial state: At(A), Level(low), BoxAt(C), BananasAt(B)
Goal state:    Have(bananas)

Actions:
           // move from X to Y
           _Move(X, Y)_
           Preconditions:  At(X), Level(low)
           Postconditions: not At(X), At(Y)

           // climb up on the box
           _ClimbUp(Location)_
           Preconditions:  At(Location), BoxAt(Location), Level(low)
           Postconditions: Level(high), not Level(low)

           // climb down from the box
           _ClimbDown(Location)_
           Preconditions:  At(Location), BoxAt(Location), Level(high)
           Postconditions: Level(low), not Level(high)

           // move monkey and box from X to Y
           _MoveBox(X, Y)_
           Preconditions:  At(X), BoxAt(X), Level(low)
           Postconditions: BoxAt(Y), not BoxAt(X), At(Y), not At(X)

           // take the bananas
           _TakeBananas(Location)_
           Preconditions:  At(Location), BananasAt(Location), Level(high)
           Postconditions: Have(bananas)
```

While it might seem obvious to you that the following sequence of actions is a solution to the problem :

```
Move(A,C), MoveBox(C,B), ClimbUp(B), TakeBananas(B)
```

computing this sequence automatically is an NP-hard problem problem in the general case (actually, it is P-SPACE-complete).

3.3.1 Collective Intelligence

While AI researchers worked on action representation in the 80s, two new approaches emerged in the domain:

1. Taking inspiration in the research by philosopher Michael Bratman, some researchers such as Cohen and Levesque proposed a new logic for modelling actions and changes and simulating human-like behaviour. This logic is called **BDI** for Beliefs, Desires and Intentions. The core idea is that the agent selects one of its desires (or goals) and, using its Beliefs about the situation, turns this desire into an intention that can be achieved by a plan (or sequence of actions).

   This research is about programming systems that simulate human decision making for action selection based on its perception of the environment.

2. Taking inspiration in social animals such that can build complex structures such as ant nests or bird flocking, other researchers in AI took an interest in **collective intelligence**. In this domain, the goal is not to find solutions to a problem by combining several basic behaviours. The solution does not arise from the behaviour of one single individual, but it *emerges* from the interactions between the individuals.

   The most famous example of such a system is the ant colony optimisation algorithm, proposed in 1992 by Marco Dorigo for his PhD thesis. This algorithm can be used, for example, to compute a solution to the Travelling Salesman Problem (TSP). Hundreds of agents walk the graph more or less randomly, and exchange with each other on the quality of their respective solutions so as to influence the other agent's decision for the next iteration.

   This research is about programming systems that interact with each other to compute a solution.

All these research contributions (reasoning about action and changes, modelling human decision about action, distributed AI), coupled with the results in knowledge engineering and decision aiding, gave birth to "multiagent based modelling" as a sub-field of AI.

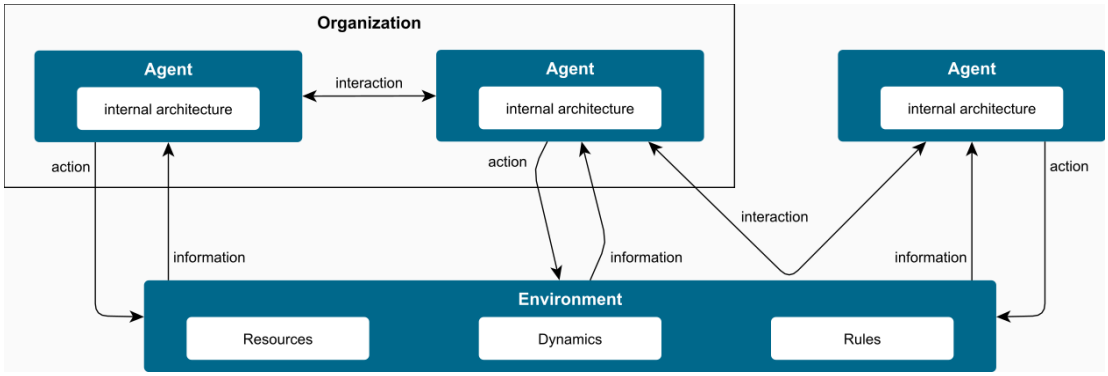
For an Artificial Intelligence engineer, what matters in multiagent systems is the capacity of agents to **solve problems in a distributed manner**, using interactions and automated reasoning.

3.4 4. To be remembered

Due to the variety of origins of multiagent research, there exists several definition of agents and multiagent systems. However, after 25 years of existence, the MAS research community came to a relatively good level of agreement on the key notions.

- **An agent is :**
  - a *software process* (or part of a process, or possibly attached to some physical components *e.g.* in robotics...)
  - with some *encapsulated data* (*i.e.* other agents/processes can't modify directly this data),
  - capable of providing a predefined set of *services* (its actions)
  - and capable of *reasoning* about its actions and the other agents' actions and to *coordinate* with the other agents so as to participate in a **collective problem solving**.
- A **multiagent system** (MAS) is a group of agents that share a *common environment* and that act in a *distributed* manner (the agents runtime are supposed to be *asynchronous*) to solve a problem for a *user*.

  This last element (the user) is important: a MAS always has a predefined purpose, and it is important to define the expected outcomes, should it be a solution to a concrete problem (*e.g.* a path in a TSP problem) or a set of observable properties in a multiagent based simulation.



In the next section, we shall see very concretely how an agent and a multiagent system work. But before we do so, here is a brief presentation of some concrete MAS applications.

3.5 5. As a conclusion: practical applications of MAS

MAS is a research domain that has spread software engineering methods and a certain view of distributed AI over the years. Many of these methods are now widely used in industry and commerce without people explicitely referring to this as Multi-Agent Systems. Indeed, as we will see in the next section, from the conceptual level to software or even physical systems, there are many views of MAS.

However, here is a list of some concrete applications that use MAS as such. It is far from exhaustive : there exists hundreds of MAS applications in the industry!

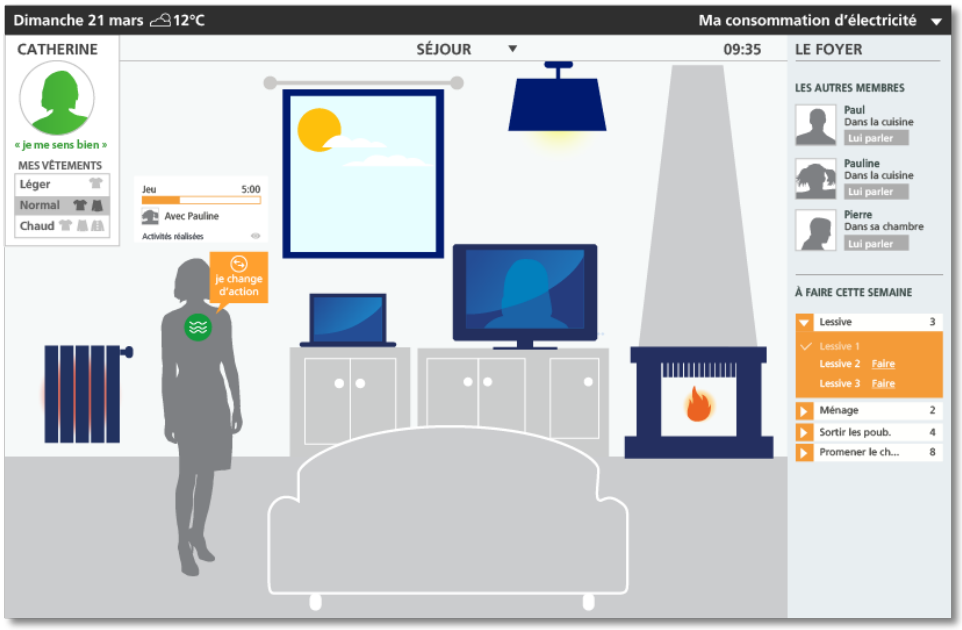
3.5.1 MAS for Crowd Simulation in Movies

Massive Software was originally developed for use in Peter Jackson's *The Lord Of The Rings* film trilogy. Subsequently the company Massive Software was created to bring this technology to film and television productions around the world. Since then Massive has become the leading software for crowd related visual effects and autonomous character animation. It combines MultiAgent Systems with Graphical Design.

3.5.2 MAS Applied to Taxi Fleet Management

MAS are widely used in logistics. The company Magenta Technology has developped in 2008 a MAS-based technology for the management of taxi fleets. This software solution is now being used by many taxi companies, such as Green Tomato Cars or Blackberry Cars, two major UK companies in London, UK.

3.5.3 MAS Applied to the Study of Electrical Consumption



The R&D branch of EDF, one of the French electricity companies, has been working on a MAS for the simulation of electrical consumption in households. This applied research project is called SMACH. The MAS platform that was developped in this project is now being used by all services at EDF R&D for the generation of realistic load

curves in answer to prospective studies: market research & pricing, new production means (*e.g.* renewable energy self-consumption), new consumption habits (*e.g.* electrical vehicules), *etc*.

3.5.4 MAS Applied to Fault Detection

SurferLab is a joint research lab formed by *Bombardier Transport* and the university of Valencienne, France. They use MAS for diagnosis of trains with the Jade platform. The multiagent systems is capable to detect faulty cars so as to correct them faster. This time saver for the diagnosis has huge economical impacts for it presents heavy breakdowns.

CHAPTER

FOUR

1.2. WHAT IS AN AGENT?

The core element in a multiagent system is the agent. An agent is always situated in a *environment*. The environment can be understood as "everything outside the agent", although this definition is largely inaccurate. People say that the agent *interacts* with the environment, which also can have many different meanings. Let us try to clarify all this.

4.1 1. Procedural Reasoning System

The basic model of an agent is the Procedural Reasoning System (PRS) proposed by Georgeff in 1989 and illustrated on the figure below:

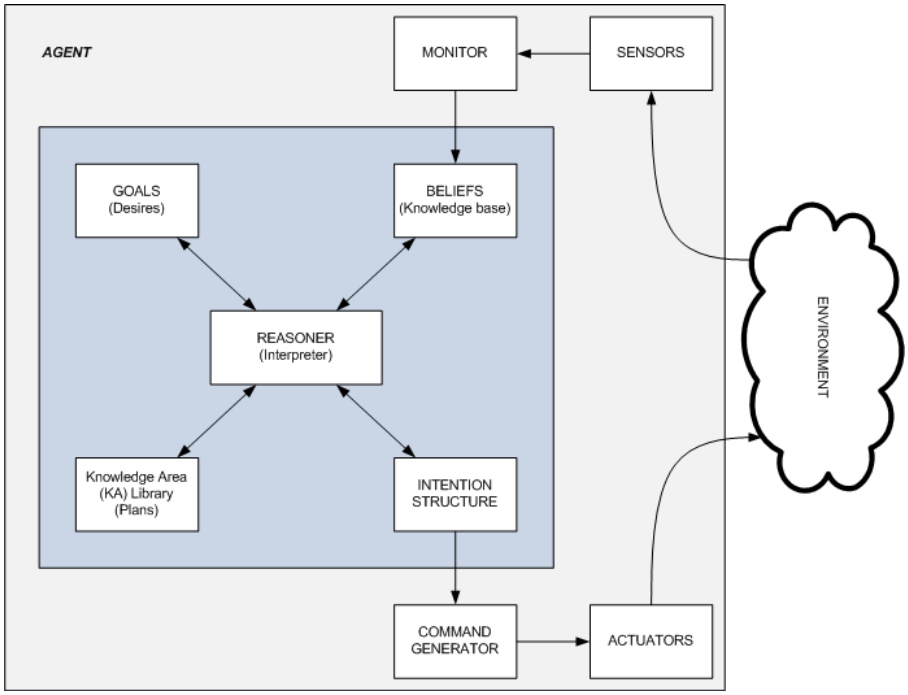


Fig. 1: The Procedural Reasoning System (Georgeff and Ingrand, 1989).

In this model, the first object to consider is the **environment**. The environment is characterised by a (partially) observable state, which means that 1) it can be modified and 2) some of the variable values can be observed by the agents. As a consequence, an environment is always provided with a set of observation functions and a set of modification functions.

In the PRS model, the agent accesses the observation function through its *sensors*, and the modification functions through its *actuators*. One can easily imagine that different agents in the system can have different sensors and

actuators and, thus, different capabilities. Concretely, this means that the agent has some code that calls the observation and modification functions. It is not clear now what exactly this code should be but you could say that the following piece of code implements the beginning of an agent-environment pair.

```
environment_variable = 0

def operation_increase_variable():
    global environment_variable
    environment_variable += 1

def perception_get_variable():
    global environment_variable
    return environment_variable

def agent(preferred_value):
    v = perception_get_variable() # this is a sensor
    while (v<preferred_value):
        operation_increase_variable() # this is an actuator
        v = perception_get_variable()
```

However, the most important aspect of the PRS model is the **procedural loop**. An agent runs continuously in three steps:

1. Perception of the environment
2. Deliberation = selection of the action
3. Action on the environment

This structure is at the core of a multiagent system. It makes it possible to have asynchronous runtimes for the agents, that coordinate by observing the environment (including the other agents) and deciding for their next move while the other agents continue to execute. **This is what makes a MAS!**

In the PRS model, the deliberation part of the *procedural loop* is given some details: it first transforms the result of the perception into internal variables, called beliefs. It then combines these with a set of goals (remember the BDI architecture by Bratman, Cohen and Levesque) and a set of plans so as to decide for some intention that will be turn into a concrete action in the environment. **Not all MAS adopt such a complex structure**. Actually, BDI agents are a specific case of so-called *cognitive agents* that will be defined later.

What you must keep in mind is that an agent:

1. Is **situated** in an environment: it can perceive and act;
2. Continuously performs a **procedural loop**: perception, deliberation, action.

Two or more agents running in the same environment make a **multiagent system**. That's it. The rest depends on the problem you want to solve. Situatedness and procedural loop are what makes an agent.

4.2 2. Key Concepts

There exists a wide range of agent models in the literature. What makes the difference is mostly the level of complexity of the deliberation phase in the procedural loop.

At one extremity of the spectrum are purely **reactive agents**. Such agent do not make use of any internal variable: they directly wire all perception to a specific action (or set of actions), hence their name: they perceive and react.

As you add internal variables and data, you slightly move the cursor toward so-called **cognitive agents**. This name encompasses a very broad set of agent types.

The least cognitive agent will simply use internal variables to store its perception. This means it has a memory of what it perceived, and it can make decisions based on this memory. Its actions are no longer pure reactions: this is the first type of cognitive agents.

The internal variables to store perceptions and, possibly, other values that the agents might compute from the perceptions, are called **beliefs**. This reflects the fact that their value might differ from the value in the environment. Indeed, due to the asynchronous nature of MAS, an agent should never assume that its beliefs corresponds to reality. This has one important consequence on the actions: remember the STRIPS example we saw previously and the plan to catch the bananas:

```
Move(A,C), MoveBox(C,B), ClimbUp(B), TakeBananas(B)
```

Nothing prevents, in a MAS, that the box will actually be at position B when the agent tries to climb on it. Indeed, after it moved the box from C to B, another agent might decide to move the box back to C (or to jumb on the box and steal the bananas). For this reason, *cognitive agents must be capable of handling with action failures*. Several options are to be considered:

- The environment returns a success value for the action, which allows the agent to update its beliefs about the action failure for the next procedural loop;
- The environment says nothing but the agent will check the result during the next procedural loop;
- In all cases, the agent must check its next action's precondition to determine if the plan is still valid, and possibly select a different action.

Note that *explicit* planning is not mandatory at this level of cognitive agents. You can imagine that the agent simply checks its current state of beliefs and search for a possible action. The "plan" is then hard-wired in the code. Before we consider explicit planning, there exists other levels of "reasoning" about the beliefs that can be considered. Simple deduction from the beliefs to add some elements in the beliefs base, more complex belief revision when a new perception conflicts with the current belief base, or even diagnosis to identify the source of a failure in the plan, etc. When it comes to explicit planning, *i.e.* agents that have explicit goals and that perform any sort of computation (from simple plan selection in a predefined list to complex planing algorithms such as SATPLAN), we call these **rational agents**.

There is a spectrum that goes from **reactive** agents to **cognitive** ones and then to **rational** agents.

Remark: for now, we do not question the interaction mechanism. This will be done in the third session of the course.

4.3 3. To be remembered

According to the **PRS model** that is at the core of agent modelling, an agent:

- is **situated** in the environment (it can perceive and act);
- might have some internal data called **beliefs** and these are encapsulated within the agent (they are not accessible to other agents or to the environment);
- runs with a **procedural loop** (perceive, decide for an action, act) in an **asynchronous** w.r.t other agents;
- in the decision phase, it can do any sort of reasoning, from simple reaction (**reactive agents**) to more complex reasoning on the beliefs (**cognitive agents**);
- cognitive agents consider actions with **preconditions and effects** (*note:* this can be way more complex but we shall skip the details for this MAS introductory course);
- agents that have *explicit goals* and *planning procedures* in the deliberation phase are called **rational agents**.

This said, there exists as many different manners of implementing a MAS as there are MAS in the literature. Implementing an agent model generally consists in answering questions for both modelling of the problem and construction of the runtime. The next subsection guides you through some of these questions.

4.4 4. Exercises

Let us begin with two very simple agents in an environment. Please copy-paste the following Python code in your IDE:

```
from time import sleep

class Environment:
    def act(self,message):
        print(message)
    def perceive(self):
        pass

class Agent:
    def __init__(self,name,env):
        self.name = name
        self.env = env
    def procedural_loop(self):
        while True:
            self.env.act("Agent "+self.name+" says hello!")
            sleep(0.1)

class Runtime:
    def __init__(self):
        e = Environment()
        (Agent("Alice",e)).procedural_loop()
        (Agent("Bob",e)).procedural_loop()

Runtime()
```

This code uses Python object-oriented programming: the environment and the agents are defined in two different classes:

- The environment provides perception and action methods that are, in this case, reduced to the display of a value;
- The agents have a simple procedural loop that consists in acting with their own name on the environment;
- The runtime instantiates two agents in the same environment.

4.4.1 Question

1. Why does this not behave as a multiagent system?
2. What is the problem?

4.4.2 Answer

Try to answer the question by yourself before you continue. Write down your answers as comments in the code. Discuss with your colleagues and with the teacher.

The answer is available here: *1. Two simple agents*.

4.4.3 Question

To overcome the above limitation, two solutions can be considered. The first one is to write a **scheduler**, *i.e.* a piece of code that calls the procedural loops of all agents, one after the other.

Modify the previous code so that Runtime creates two agents and calls a single-step procedural loop for all agents.

4.4.4 Answer (1)

Try to write the code before you go further. Do not simply read the answer.

Here is a first solution: *2. Two simple agents (continued)*.

In the above solution, you'll notice two things:

1. This MAS verifies a specific property, which is that the procedural loop of all agents is performed at each time step: all agents run at the same "speed".

   This is called a *synchronous* MAS. While the agent has its own runtime, interleaved with the one of the other agents (which corresponds to the specification of a MAS) these runtimes are synchronised.

2. The agents procedural loops are always invoked in the same order, which is not a valid hypothesis in a MAS. Agents should **never** use that property. If you want to avoid this, you can simply modify the `Runtime` class as follows:

```
    from random import shuffle

    class Runtime:
        def __init__(self):
            e = Environment()
            agents = [Agent("Alice", e), Agent("Bob", e)]
            while True:
                shuffle(agents)
                for a in agents:
                    a.procedural_loop()


Note that this new version still is a synchronous MAS.
```

4.4.5 Answer (2)

Here is a now a second solution, with an asynchronous MAS that relies on the Operating System multitasking mechanisms: *Version 2 : with threads*.

Note that since both agents have the exact same sleeping time and that the operations performed are quite reduced, they will behave as if in a synchronous MAS. This can be easily changed (although it is of no importance for our situation):

```
from random import uniform
...
class Agent(Thread):

    def procedural_loop(self):
```

(continues on next page)

(continued from previous page)

```
        ...
        sleep(uniform(0.1,0.5))
...
```

This will make agents have random sleep time. It can simulate a changing time complexity in the procedural loop.

4.5 5. Do we have a multiagent system yet?

The agents in the preceding example are not really situated since the perception phase does nothing. They use encapsulated data (the agent's name), but we can't really call these a set of beliefs since they are not connected to the perception. Although they have asynchronous runtimes, with a procedural loop, we cannot really call them *agents*, even not *reactive agents*. In order to write reactive agents, we want the deliberation phase, in the procedural loop, to depend on the perceptions.

4.5.1 Question

Write a new multiagent system, either in the synchronous or in the asynchronous version, in which the environment has some variable that the agents can perceive. Write two reactive agents: the first one increases the variable by a random value when it is even, the other one when it odd.

Test your program.

4.5.2 Answer

Here is a possible solution: *3. Two agents and an environment's variable*. You might have a different one depending on how you implemented your agents. Just check that all properties in the PRS model are satisfied: situatedness, procedural loop, *etc*.

4.6 6. Concurrent modifications

Let us consider the code given in *3. Two agents and an environment's variable*. In this code, agents' runtimes are completely asynchronous. Each agent is a thread. Both agents perceive and act on the value `v` in the environment.

Writing asynchronous MAS with threads requires to consider possible concurrent modifications. In our case, you should not see any problem because the agents spend most of their lifetime sleeping. It is very unlikely that two agents run their procedural loops at the exact same time and that the Operating System's scheduler has to interrupt this piece of code.

However, in the general case, you must consider a possible interruption of the following sequence of code:

```
1: self.b = self.env.perceive()     # in Agent.procedural_loop()
2: if condition_on_b:                # in AgentX.act()
3:     x = randint(1,4)              # in Environment.increase() -- 4 lines
4:     print("Agent " + name + " increase the value by "+ str(x))
5:     self.v = self.v+x
6:     print("  --> " + str(self.v))
```

4.6.1 Questions

1. What do you expect to be a problem? Explain why.
2. What is a possible solution?

4.6.2 Answers

Take the time to think about it. Make propositions to the teacher and to your peers in the (virtual) classroom. Some of the possible interruptions are not a problem in a MAS.

The answer is available here: *4. Concurrent modifications*. It also proposes a solution to handle such concurrent modifications.

CHAPTER

FIVE

1.3. PROPERTIES & DEFINITIONS

Before we move to the next important aspects of the course (*i.e.* multiagent interactions), we must consider some important definitions in the domain. All these definitions are related to some crucial questions to be answered before implementing a MAS.

5.1 1. Properties of the Environment

5.1.1 Fully or partially observable?

Fully observable environments mean that all variables in the environment can be accessed by any agent. This is generally done with some generic mechanism such as:

```
class Environment:
    variables = { 'name1':'value1',
                  'name2':'value2',
                  ... }

    def perceive(self, name):
        return variables[name]

    def act(self, name, value):
        variables[name] = value
```

However, most MAS use *partially observable* environments, in which each agent only accesses a specific set of variables (as was the case in our simple example with Alice and Bob). This is generally done by a set of ad-hoc perception functions that are called by the agents.

5.1.2 Deterministic or stochastic?

From the agent's point of view, it is always possible that some action has unexpected effects. Indeed, other agents might have changed the outcomes of the action due to their own actions before the first one starts its next perception phase. Let us consider a simple variation on the Alice-Bob example:

```
from threading import Thread

class Environment:
    v = 0
    def act(self):
        self.v = 1 - self.v
        print(self.v)
```

(continues on next page)

(continued from previous page)

```
    def perceive(self):
        return self.v

class Agent(Thread):
    def __init__(self, name, preferred_value, env):
        Thread.__init__(self)
        self.name = name
        self.env = env
        self.pv = preferred_value
    def run(self):
        while True:
            self.procedural_loop()
    def procedural_loop(self):
        if self.env.perceive() != self.pv:
            self.env.act()

class Runtime:
    def __init__(self):
        e = Environment()
        Agent("Alice", 0, e).start()
        Agent("Bob", 1, e).start()

Runtime()
```

In this example, Alice moves v to 0 every time its value is 1 and Bob does the contrary. From Alice's point of view, and if Bob is quick enough, she perceives that v is set to 1, changes it to 0 and, during next turn, notices that v is still at the value 1. She might consider that her action did not have the expected effect.

However, another viewpoint on this implementation is that both Alice and Bob are aware that her actions did work, but that other agents can change the value of v. We can say that their actions are *deterministic* because, from the implementation's point of view, both agents can be assured that their action has the expected effect (and that other agents might counter this effect).

In a *stochastic* environment, some actions have effects that are no guaranteed or not unique. This was partially the case in the initial Alice and Bob example, with an increase in v that could be any value between 1 and 4. But one can imagine more complex environments in which some action's effects depend on hidden variables so that the result of the action, even when no other agent acts in the environment, is not predictable for the agent. In such MAS, machine learning algorithms can be used to build Markov Decision Processes representations of the MAS behaviour.

5.1.3 Static or dynamic?

In the Alice and Bob example, the environment does not have any internal process. All changes are performed by the agents. We can say that the environment is *static*. However, in some MAS, it is better to use a *dynamic* environment. One typical example is ant colony simulation in which pheromons dropped by the agents in the environment need to spread and evaporate with time. The Netlogo platform supports this mechanism natively.

5.1.4 Discrete or continuous?

As in most modelling problem, computer science systems deal with the discrete *vs* continuous question. However, in multiagent systems, this question comes to a particular level, as far as actions in the environment are considered. In the Alice and Bob example, the number of possible actions and perceptions is fixed and finite. This is the case in most MAS and we can say that we have a *discrete* environments. However, some problems require the environment to support an infinite set of actions, possibly with continuous values as parameters.

5.2 2. Properties of the Agents in the MAS

5.2.1 Autonomous agents

Autonomy in a Multi-Agent Systems does not mean that the agents can do whatever they want to. To begin with, all agents behave according to a well-defined program. Autonomy must be understood as the independence of one agent's behaviour (*i.e.* action selection) with respect to some part of the system. Understanding this concept requires to consider different levels.

- At the agent level, pro-action is the first degree of autonomy. Cognitive agents can decide to perform an action due to some internal process changing their belief base. In that case, the agent can trigger some action in the environment in a pro-active manner, *i.e.* independently from any specific signal from the environment. Imagine an agent that plays hide and seek and that begins seeking after ending a countdown. This is a (very simple) example of pro-action.
- At the interaction level, although we have not discussed interaction mechanisms yet, autonomy consists in having agents that do not necessarily accept other agent's requests. Like in human communication, several situations can occur: the agent deliberately ignores the message because it is considered irrelevant; the agent does not answer immediately while it could have; the agent answers but refuses the request; *etc*.
- At the MAS level, there is no autonomy: agents are not autonomous *w.r.t.* the system. On the contrary, they perform what they are designed for. They follow well-defined protocols so that the global task is achieved. Otherwise, there is some misconception in the system.

5.2.2 Loosely vs tightly coupled?

From the software engineering point of view, several approaches can be considered when designing the agents behaviour. In tightly coupled MAS, the developer has a complete view of the MAS, of possible actions performed by other agents. In other words, it can use some of this information to design action mechanism in a more efficient manner. On the contrary, in a loosely coupled MAS, you must make no assumption on the design of other agents in the MAS. This means that you must define very robust behaviours to avoid deadlocks, infinite loops or unnecessary waiting times in communication. We will see several examples in the third session of this course.

5.2.3 Open MAS?

Open multi-agent systems are a specific kind of system in which you assume that agents can enter and leave the system at any time during the execution. This hypothesis requires programmers to design specific mechanism to ensure that messages are not left unanswered, that some feature in the system will not disappear when not expected, *etc*. While many MAS are closed, the most platforms support the addition and deletion of agents at runtime. The difficulty lies in the programming of the agents themselves to avoid inconsistent behaviours in the system.

Concrete examples of *open* MAS are ant colonies or prey-predator simulations, such as what we will do in the next session of this course.

5.2.4 Distributed systems?

Distribution in a MAS is a very difficult question, because it can be addressed at different levels. The first, and most important level, is the *conceptual* one. One can perfectly design a MAS that runs in a single thread. Here is the Alice-Bob example from above in its mono-thread version:

```
from random import shuffle

class Environment:
    v = 0
    def act(self):
        self.v = 1 - self.v
        print(self.v)
    def perceive(self):
        return self.v

class Agent:
    def __init__(self, name, preferred_value, env):
        self.name = name
        self.env = env
        self.pv = preferred_value
    def procedural_loop(self):
        if self.env.perceive() != self.pv:
            self.env.act()

class Runtime:
    agents_list = []
    def __init__(self):
        e = Environment()
        self.agents_list.append(Agent("Alice", 0, e))
        self.agents_list.append(Agent("Bob", 1, e))
    def run(self):
        while(True):
            shuffle(self.agents_list)
            for a in self.agents_list:
                a.procedural_loop()

r = Runtime()
r.run()
```

Not all MAS are multi-threaded. What matters is that it implements distribution principles that were presented in the first section of this course: encapsulated data, accessed via perception and action methods only, using a *procedural loop* that interleaves the perceptions and actions of agents. This is distribution at a conceptual level. Most MAS implemented in Netlogo or Gamma stay at this level and it is perfectly right!

This said, distribution can also be considered at the *software level*. This is what we achieve with threads: agents get separated in different threads or processes (depending on the language and implementation). These threads generally run on a single computer. They share its computation time, thanks to the OS scheduler, so as to perform their operations. Most MAS implemented in Repast Symphony are distributed at the software level (and, of course, at the conceptual level).

A third level of distribution can also be considered. It is more difficult to achieve and not all platforms support it. It is the *physical distribution*. Imagine a group of robots that participate in a collective task, a network of smart sensors that exchange information with each other or even living creatures such as social insects (bees, ants...) or human beings. They all share a common physical environment and interact with each other. Such systems are distributed at the physical level.

Implementing such systems (robots, sensors, distributed services or simulation that requires physical distribution) is

often difficult because you will need specific code to connect the abstract actions and perceptions to physical operations, message passing, *etc*. The most famous platforms you can use to make this task easier are:

- The Java Agent Development Environment (Jade) which is a Java API to run multi-threaded, multi-systems applications with brokers for message passing. It is widely used for implementing networked applications (sensor networks, distributed simulation or services).
- The Robot OS (ROS) which provides a set of tools to ease the implementation and simulation of robotic systems and robots in a physical environment. The resulting code can be easily transferred to physical robots.

5.3 3. Concluding remarks

At this point in the course, you figured out that MAS programming is largely a question of design patterns to implement a distributed system with some interesting properties. You can implement a MAS in plain C if you want. Thread, classes, or even more complex features are all compiled to assembler language in the end. However, computer scientists generally rely on high-level models and multi-agent platforms to make agent based programming easier. A multi-agent platform implements the scheduler, the procedural loop and the interaction mechanisms that we will see in the third session of this course. Some of them rely on existing languages (*e.g.* Repast Symphony or Jade for Java, Mesa or Spade for Python). Some other define dedicated languages to get rid of the initial language's hassle: this is the case for Netlogo or Gamma (both are initially programmed in Java).

The idea with platforms is to avoid programmers to rewrite the basic mechanisms in MAS. This principle is at the basis of C/C++/C# libraries, or Java APIs, or Python modules. When such libraries get complex enough and take control over the runtime, we call them platforms. In the next section, we will introduce the Mesa platform for Python.

CHAPTER

SIX

1.4. THE MESA PLATFORM FOR PYTHON

Mesa is a Python framework for agent-based modeling. Getting started with Mesa is easy. In this section, we will walk through creating a simple model and progressively add functionality which will illustrate Mesa's core features. The running example will be the implementation of a simple Agent-based Computational Economics (ACE) model that simulates distributions of money, income, and wealth in society using statistical physics. A complete description of the model can be found in this paper.

6.1 1. Installation

Mesa requires Python 3 and does not work in Python 2 environments. **Make sure your environment is set up with Python 3**.

If your IDE support extensions (*e.g.* VisualStudio, PyCharm), you can add Mesa directly in the IDE (this is probably the easiest solution). If not, you will need to install Mesa manually. If you are using a Windows/Conda environment, you can do this by opening a conda console and typing the following installation instruction:

```
pip install mesa
```

If you are using a Mac or Linux system, simply type this instruction in the OS terminal. It will install Mesa system-wide.

You may check that Mesa has been properly installed with the following Python instruction (either in a Python console or in your IDE):

```
import mesa
```

This should not produce any error!

Matplotlib and numpy will also be used during the tutorial session. Install them if they are not already in your Python environment.

6.1.1 Setting up the project

Create a folder named `Money_Model` (or create a project with this name in your IDE) and create a new python file `money_model.py`

In the following, we will assume that your project is implemented in a `Money_Model` folder and that your first (and main) Python file in the project is name `money_model.py`. You will need to know where these files are located on your computer.

6.1.2 Documentation

To help you during this tutorial, we recommend that you bookmark the Mesa API documentation.

6.2 2. Mesa Basics

Implementing a multi-agent system in Mesa requires to define two core classes:

- One for multiagent platform: it will inherit the Mesa `Model` class;
- The other for the agents: it will inherit the Mesa `Agent` class.

The `Model` class holds the model-level attributes, manages the agents, and generally handles the global level of our agent-based model (hence the same). Each instance of the `Model` class will make a specific run of the multi-agent system.

The `Agent` class is characterized by a `step` method that implements the *procedural loop*. It will be called by the model's scheduler at runtime for the agents to do whatever you have decided they should do. Agents can access to their model with the `self.model` attribute.

The *scheduler* is a specific component of the model that can be used to control the agent's runtime. The Mesa platform runs in a *synchronous manner* (see the second section of today's course) and several classes are proposed to ease the scheduling process. They all have in common two properties:

- Their constructor takes an instance of `Model` as parameter;
- They all implement a `step` method that calls the `step` method of each agent once.

In this session, we shall use the Mesa `RandomActivation` scheduler from the `mesa.time` package. Its `step` method invokes all agents in a random order.

With these three components in mind, take a close look to the skeleton of a Mesa multiagent system:

```
from mesa import Agent, Model
from mesa.time import RandomActivation

# let's create a class for our agents
class MyAgent(Agent):
    def __init__(self, unique_id, model):
        super().__init__(unique_id, model)
        #... define local variables here ...
    def step(self):
        #... define the perception-deliberation-action loop here ...

# and now let's create a MAS
class MyModel(Model):
    def __init__(self):
```

(continues on next page)

(continued from previous page)

```
        # we need a scheduler
        self.scheduler = RandomActivation(self)

        # we need to create agent
        a = MyAgent(id, self) # id is generally a number; it must be a different␣
↪value for each agents
        self.scheduler.add(a) # don't forget to add the agent to the scheduler

        #... do the same for all agents ...

    # and now, we can define methods to control the runtime of the platform
    # here are three examples

    # 1. this method performs only 1 step for all agents
    def one_step(self):
        self.scheduler.step()
    # 2. this method performs only n steps for all agents
    def run_n_steps(self,n):
        for i in range(n):
            self.one_step()
    # 3. this method runs the agent until you stop the code manually
    def run_continuously(self):
        while(True):
            self.one_step()

# This is how you start the MAS:
m = MyModel()
m.run_continuously()
```

The `one_step`, `run_n_steps` and `run_continuously` methods are not mandatory: they illustrate the fact that running the agents is done by the scheduler's `step` method, which must be invoked from the model. In this skeleton, our model class creates a random activation scheduler and adds agents to this scheduler. The agent class must define the content of the `step` method that will be called by the scheduler.

6.2.1 Practice yourself!

Question

Based on the above example, implement a simple Mesa multi-agent system with `N` agents that all have some internal variable with an initial value of `0`. The agents' ids will range from `0` to `N-1`. The procedural loop of the agents performs two actions:

1. Increase the value of the internal value by 1;
2. Print the agent's number and its current internal value.

Running three steps of the system with N=4 should produce the following output (with the agents in random a order):

```
Agent 2 has value 1
Agent 3 has value 1
Agent 1 has value 1
Agent 0 has value 1
Agent 1 has value 2
Agent 2 has value 2
Agent 3 has value 2
Agent 0 has value 2
```

(continues on next page)

(continued from previous page)

```
Agent 0 has value 3
Agent 1 has value 3
Agent 3 has value 3
Agent 2 has value 3
```

Check your code with your neighbour of with the teachers.

6.3 3. The Money example

Our goal is to implement a simple agent model in which each agent has only one variable that represents **how much wealth it currently has** (each agent will also have a unique identifier as in the example above). We want the agent to adopt the following behaviour at each step of their execution (*i.e.* in their *procedural loop*):

1. Check their wealth;
2. If they have the money, give one unit of it away to another random agent.

To allow the agent to choose another agent at random, we shall use the `model.random` random-number generator. This works just like Python's `random` module, but with a fixed seed set when the model is instantiated, that can be used to replicate a specific model run later. To access an agent's wealth, we can use the scheduler's internal list of all the agents it is scheduled to activate.

Note that agents also have a `self.random` attribute that refers to `self.model.random`.

```
other_agent = self.random.choice(self.model.schedule.agents)
other_agent.any_method()
```

Question

1. Write a class `MoneyAgent` that implements an agent with the above behaviour.
2. This code violates one of the principles of MAS we have discussed in the first and second sections of this course. Which one?

6.3.1 The Money MAS model

Question

Implement a `MoneyModel` class that creates `N` money-agents. Run some steps of the MAS and store the resulting value of `wealth` for each agent into a Python list. You can then display the result using the `matplotlib` library:

```
import matplotlib.pyplot as plt

# insert your code for MoneyAgent and MoneyModel

model = MoneyModel(10) # 10 agents in our example
model.run_n_steps(50)  # 50 steps in our model
all_wealth = model.get_all_agents_wealth()
print(all_wealth)
plt.hist(all_wealth,bins=range(max(all_wealth)+1))
plt.show()
```

6.4 4. Adding a spatial environment

Many agent-based models have a spatial element, with agents moving around and interacting with nearby neighbors. Mesa supports two overall kinds of spaces:

- *Grid spaces*: they are divided into cells, and agents can only be on a particular cell, like pieces on a chess board;
- *Continuous spaces*: they allow agents to have any arbitrary position.

Both grids and continuous spaces are frequently toroidal, meaning that the edges wrap around, with cells on the right edge connected to those on the left edge, and the top to the bottom. This prevents some cells having fewer neighbors than others, or agents being able to go off the edge of the environment.

In our simple model, we will use a grid space in which agents walk at random. Instead of giving their unit of money to any random agent, they will give it to an agent on the same cell. We shall use the Mesa `MultiGrid` class that allows multiple agents to be in the same cell (as opposed to `SimpleGrid` which only allows one agent at a time in each cell). Its constructor receives `width` and `height` parameters, and a boolean as to whether the grid is toroidal. We can place agents on a grid with the grid's `place_agent` method, which takes an agent and an `(x, y)` tuple of the coordinates to place the agent.

```
from mesa.space import MultiGrid

grid = MultiGrid(width, height, toroidal)

grid.place_agent(a, (x, y))
```

All these method should be invoked in the model, at the agents' creation level.

Each agent is now provided with a `self.pos` variable that indicates its position in the grid. Moving an agent in the space (*i.e.* changing its position's value) is done with the `move_agent` method, which is typically invoked in the agent's `step` method. It takes as parameter the agent and its new position `(x,y)`.

```
# This code is invoked from within the Agent class :
# - self is the agent
# - self.model is the agent model
# - self.model.grid is the model's grid space

self.model.grid.move_agent( self, (new_x,new_y) )
```

Instead of computing the possible positions manually, you can use the `get_neighborhood` method which returns all the neighbors of a given cell. This method can get two types of cell neighborhoods: Moore (including diagonals), and Von Neumann (only up/down/left/right). It also needs an argument as to whether to include the center cell itself as one of the neighbors.

```
list_of_positions = grid.get_neighborhood( (x,y), moore=True, include_center=False )

# moving an agent to a random position in this list is then easy:
new_position = model.random.choice(possible_steps)
grid.move_agent(agent, new_position)
```

Last but not least, one can get the list of agents that occupy a given cell with:

```
cellmates = self.model.grid.get_cell_list_contents([self.pos])
```

In this example, `cellmates` is a list of agents.

Question

Modify the previous code so that :

- The model receives a `width` and `height` as parameters to initiate a toroidal grid space;
- All agents are positioned randomly in the space;
- At each step, all agents move randomly and then exchange wealth with the agents that share their current cell.

To make your code easier to read, we recommend that you write two methods in your `MoneyAgent` class:

- The `move` method moves the agent to one of its neighbouring cells;
- The `share` method modifies the wealth.

6.4.1 Viewing the spatial environment

Viewing the position of agents might be interesting in such a multi-agent system. We can do it quite easily with the `matplotlib` and `numpy` libraries. We create a `numpy` array of the same size as the grid, filled with zeros. Then we use the `coord_iter` method in the `grid` object, which lets us loop over every cell in the grid, giving us each cell's coordinates and contents in turn.

```
model = MoneyModel(20,10,10)  # 10 agents in our example
model.run_n_steps(50)  # 50 steps in our model

import numpy as np

agent_counts = np.zeros((model.grid.width, model.grid.height))
for cell in model.grid.coord_iter():
    cell_content, x, y = cell
    agent_count = len(cell_content)
    agent_counts[x][y] = agent_count
plt.imshow(agent_counts, interpolation='nearest')
plt.colorbar()

# If running from a text editor or IDE, remember you'll need the following:
plt.show()
```

6.5 Concluding Remarks

You now have all elements you need to run MAS and obseve its behaviour. Two additional feature of Mesa (that you might want to use) are proposed in the following section: - Data collection, which simplifies monitoring the agent's internal data and the environment. - Interactive visualization, which allows to run multiagent simulations that can be observed at runtime.

The next section of today's session is **optional**. It concerns only students that are already at ease with the rest of the MAS models in Mesa. You can skip it.

- Next week (second session), we will use what we have learned today on MAS and the Mesa library to implement a simple simulation.
- The third session will present multiagent interaction mechanisms, both in theory and in practice with the Mesa library.

CHAPTER

SEVEN

1.5. EXERCISE SOLUTIONS

7.1 1. Two simple agents

Running this code produces a continuous output of « `Agent Alice says hello!` ». It does not behave as a MAS because the agents do not run asynchronously. The reason is that the runtime never leaves the procedural loop of agent Alice. Run this code in debug mode, with a break point in the first line of the `procedural_loop` method if you have any doubt.

Return to previous page: *Question*.

7.2 2. Two simple agents (continued)

7.2.1 Version 1 : with home-made scheduler

```
from time import sleep

class Environment:
    def act(self,message):
        print(message)
    def perceive(self):
        pass

class Agent:
    def __init__(self,name,env):
        self.name = name
        self.env = env
    def procedural_loop(self):
        self.env.act("Agent "+self.name+" says hello!")
        sleep(0.1)

class Runtime:
    def __init__(self):
        e = Environment()
        a = Agent("Alice",e)
        b = Agent("Bob",e)
        while True:
            a.procedural_loop()
            b.procedural_loop()

Runtime()
```

Return to previous page: *Question*.

7.2.2 Version 2 : with threads

```
from time import sleep

class Environment:
    def act(self, message):
        print(message)

    def perceive(self):
        pass

from threading import Thread

class Agent(Thread):
    def __init__(self, name, env):
        Thread.__init__(self)
        self.name = name
        self.env = env

    def run(self):
        while True:
            self.procedural_loop()

    def procedural_loop(self):
        self.env.act("Agent " + self.name + " says hello!")
        sleep(0.1)

class Runtime:
    def __init__(self):
        e = Environment()
        a = Agent("Alice", e)
        b = Agent("Bob", e)
        a.start()
        b.start()

Runtime()
```

Return to previous page: *Question*.

7.3 3. Two agents and an environment's variable

```
from time import sleep
from random import *
from threading import Thread

class Environment:
    v = 0

    def increase(self, name):
        x = randint(1,4)
        print("Agent " + name + " increase the value by "+ str(x))
        self.v = self.v+x
        print("  --> " + str(self.v))

    def perceive(self):
        return self.v

class Agent(Thread):
    def __init__(self, name, env):
        Thread.__init__(self)
        self.name = name
        self.env = env

    def run(self):
        while True:
            self.procedural_loop()

    def procedural_loop(self):
        self.b = self.env.perceive()
        self.act()
        sleep(uniform(0.1,0.5))

class AgentOdd(Agent):
    def act(self):
        if self.b%2!=0:
            self.env.increase(self.name)

class AgentEven(Agent):
    def act(self):
        if self.b % 2 == 0:
            self.env.increase(self.name)
```

(continues on next page)

(continued from previous page)

```
class Runtime:
    def __init__(self):
        e = Environment()
        a = AgentOdd("Alice", e)
        b = AgentEven("Bob", e)
        a.start()
        b.start()

Runtime()
```

Return to previous page: here: *Question*.

7.4 4. Concurrent modifications

If the scheduler interrupts the code between line 1 and 2, or even between line 2 and 3, and passes the "handle" to the other agent, the first agent will perform an increase operation on a variable that might not be of the right parity. **This is perfectly correct according to a MAS specification!** Agents perform actions based on their beliefs, and some of these actions might be erroneous from an external observer's point of view. From the viewpoint of the agent, it is not.

Now let us consider the case where the scheduler stops in the middle of the Environment.increase() method. This is a problem because the calling agent could perfectly assume this method to be atomic, uninterruptible. This is what you expect when you perform operations in an environment. And, indeed, this is an important property of a MAS: **actions in the environment must be atomic** because agents have no control over these actions. While this is not a big deal in our "increase the value" example, it might cause disastrous failures in more complex multiagent systems.

To overcome this problem, you must use thread locks. In python, this can be done easily with the following code:

```
from threading import Lock

l = Lock()
with l:
    ... concurrent code ...
```

We will not enter further in the details of this mechanism. However, you should keep in mind this rule: when writing an asynchronous MAS with threads, the environment operations should be protected by some lock. In our case:

```
from threading import Lock
from random import randint

class Environment:
    v = 0
    l = Lock()

    def increase(self, name):
        with self.l:
            x = randint(1,4)
            print("Agent " + name + " increase the value by "+ str(x))
            self.v = self.v+x
            print("  --> " + str(self.v))

    def perceive(self):
        with self.l:
            return self.v
            # The lock is not required here because the operation was already atomic..
↪.
```

Return to previous page: here: *Questions*.

CHAPTER

EIGHT

2. MULTIAGENT SIMULATION : PREYS AND PREDATORS

8.1 Agenda – Recall

1. Friday, March the 5th, 2021 : Introduction to MAS : definitions and implementation of a platform
2. Friday, March the 12th, 2021 : Multiagent simulation : preys and predators
3. Friday, March the 19th, 2021 : Interaction mechanisms : models and implementation
4. Friday, March the 26th, 2021 : Argumentation-based negotiation I: Practical session
5. Friday, April the 2nd, 2021 : Argumentation-based negotiation II: what is argumentation?
6. Friday, April the 9th, 2021 : Argumentation-based negotiation III: what is a negotiation protocol?
7. Friday, April the 16th, 2021 : Argumentation-based negotiation IV: Practical session

8.2 Session 2.

The second session aims at presenting Multi-Agent Based Simulation. We will first introduce what a simulation is, the different stages that compose it and the time's representation used in simulation. Then, we will present more precisely Multi-Agent Based Simulation and its context of use. To finish, we will talk about how to implement and manage the problems of time and concurrency.

As for the previous session, all implementation is done using the Python programming language and the Mesa library.

8.3 Some references

- Ferber, J. (1995), *Les Systèmes Multi-Agents*, InterEditions. **(French version)**
- Ferber, J. (1999), *Multi-agent systems: An introduction to distributed artificial intelligence*, Addison Wesley. **(English version)**
- Michael Wooldridge (2002), *An Introduction to MultiAgent Systems*, John Wiley & Sons Ltd.
- Robert E. Shannon (1977), *Simulation modeling and methodology*, SIGSIM Simulation Digital.
- Robert E. Shannon (1998), *Introduction to the art and science of simulation*, IEEE Computer Society Press.
- Bernard P. Zeigler (2000), *Theory of Modeling and Simulation*, Academic Press, Inc.

CHAPTER

NINE

2.1. MULTI-AGENT BASED SIMULATION: INTRODUCTION

Between Multi-Agent System, that you saw previously, and computer simulation, Multi-Agent Based Simulation is a relevant solution for the engineering and the study of complex systems in many fields (artificial life, biology, economics, etc.) thanks to its individual-centered representation of the considered systems. Indeed, Multi-Agent Based Simulation implements models in which individual entities (autonomous agents), their environment and their interactions are directly represented. Thus, the description of the dynamics of the system is no longer done by declaring equations relating to the macroscopic properties of the simulation, but to the individual behavior of each entity.

Multi-agent simulation is more and more widespread because it has become, like computer simulation, an essential tool for understanding phenomena competing with experimentation and prototyping but at a much lower cost. It is thus used by a large part of the scientific community but also by many industrial players.

In this course, we will first present the concept of simulation and computer simulation. Then we will focus on Multi-Agent Based Simulation.

CHAPTER

TEN

2.2. SIMULATION IN GENERAL

10.1 1. Simulation definition

Simulation is one of the most effective decision support tools available to scientists, designers and managers of complex systems. It consists in building a model of a real system and conducting experiments on this model in order to understand the behavior of this system and improve its performance. Here is one of the definitions of a simulation:

> "The process of designing a model of a real system and conducting experiments with this model for the purpose either of understanding the behaviour of the system or of evaluating various strategies (within the limits imposed by a criterion or a set of criteria) for the operation of the system." *Shannon*, 1998.

The issue behind a simulation is therefore to study a real system in order to understand its internal functioning and / or to predict its evolution under certain conditions. Moreover, to achieve these objectives, this study is necessarily done through a **model** of the **real system** which is used to carry out the experiments. The terms **real system** and **model** are the two keywords in this definition.

It is important to understand that the real system does not necessarily refer to a phenomenon that exists in nature. It can also be an intellectual construction of a virtual phenomenon. This denomination is in fact used to clearly distinguish the phenomenon to be studied from its model which is also considered as a system.

Shannon adds that a simulation process is constituted by the construction of the model and by the analytical use which is made of the model to study the system.

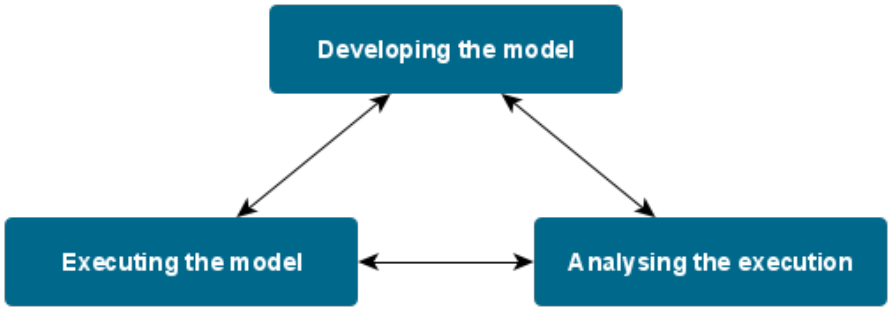
10.2 2. Computeur simulation definition

Computer simulation is in fact inseparable from the *experimental process* which is linked to its objective.

> "Computer simulation is the discipline of designing a model of an actual or theoretical physical system, executing the model on a digital computer, and analyzing the execution output." *Fishwick*, 1997.

This definition gives a global overview of a computer simulation process. *Fishwick* defines this discipline as an iterative and nonlinear process composed of three interdependent fundamental tasks:

1. The development of the model.
2. Running the model on a computer.
3. Analysis the execution of the model and the obtained results.

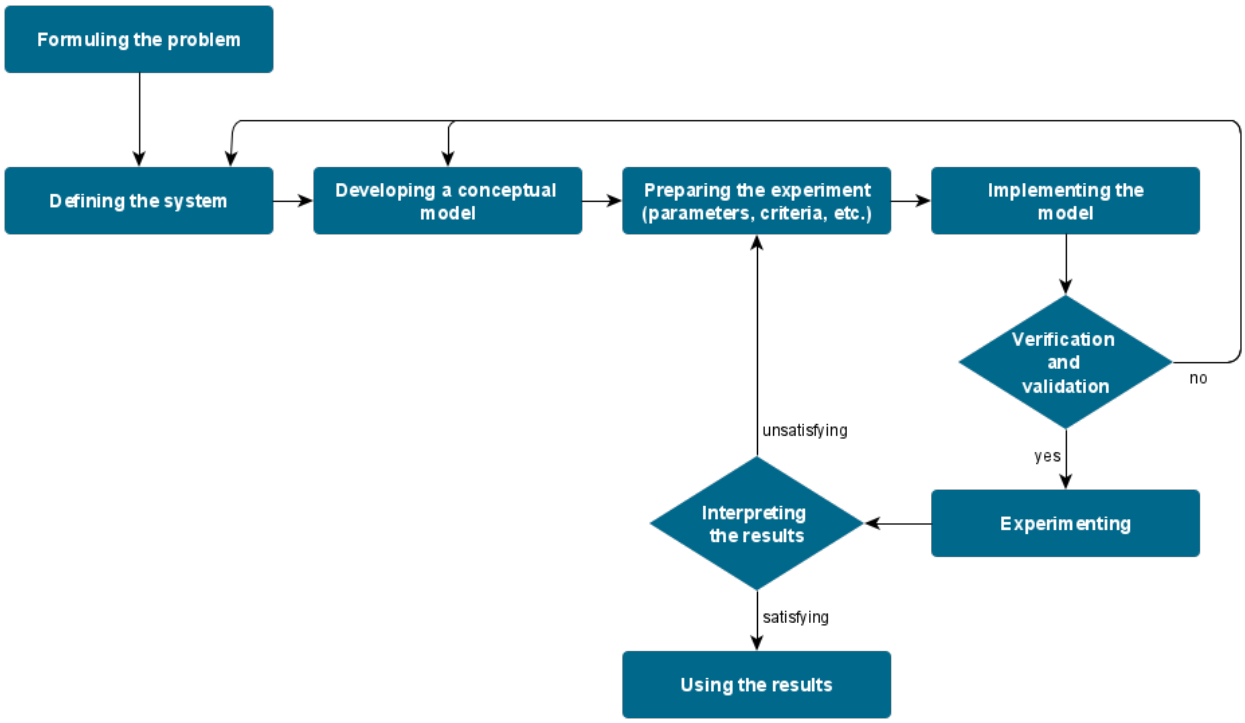


This definition clearly shows the importance of the model and its development for the design of a simulation.

10.3 3. Simulation as an experimental process

Most of the time, *Fishwick*'s definition is sufficient to illustrate clearly and simply the different stages that constitute the development of a computer simulation. However, it may be interesting to detail these steps:

1. Definition of the problem: clearly define the objectives of the study. What are the questions we want to answer?
2. Definition of the system: determine which parts of the system to study. The model will then be developed according to the objectives set.
3. Definition of the model: develop a first model graphically or in pseudo code. This involves defining the different entities that belong to the system: components, variables, interaction between components, etc.
4. Preliminary analysis of the experiment: determine what the criteria are which will make it possible to assess the quality of the experiment: what are the parameters to vary, with what amplitude and over how many runs? How experience will be needed for the whole experiment?
5. Definition of initial parameters: determine and collect the data that are necessary for the development of the initial values that will be used for the configuration of the model.
6. Implementation of the model: convert the developed model into a simulation language to allow its implementation on a computer.
7. Verification and validation of the model: check that the simulator correctly execute the model to validate the results obtained by it. Are they acceptable and representative of the system being want to study?
8. Experimentation: run the actual simulation to recover desired results and perform a sensitivity analysis of the model taccording to the initial parameters.
9. Interpretation of the results: conclude on the model from the results obtained.



This more complete description provides a better idea of what a computer simulation is and the different challenges to be overcome. It also introduces the validation and verification steps in a simulation: these two steps are fundamental and cannot be dissociated from the experimental process (even more in the context of a computer simulation).

10.4 4. Dynamic systems

Now that we have an idea of what a computer simulation is, it is worth presenting the different concepts that are manipulated to model real systems. In this context, the notion of **dynamic system** is essential. By **dynamic system** we mean:

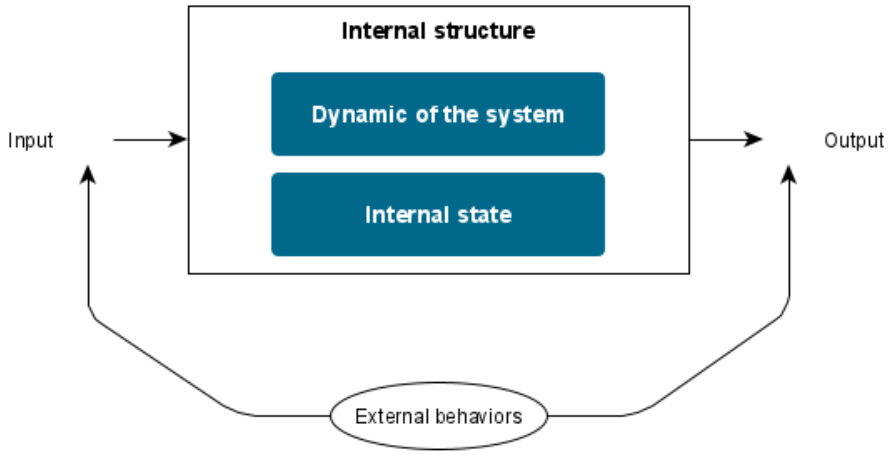
> "Any formal construct which provides general modeling concepts for various kind of disciplines." Rozenblit & Zeigler, 1993.

Most models used to represent a **dynamic system** are based on the abstractions offered by the **systems theory**.

The basic principle of this theory is to consider that a system can be specified according to two fundamental aspects:

- The *behavior of the system* at its limits (external behavior): the observable reactions of the system from outside it;
- The *internal structure* of the system: its internal state and its intrinsic functioning (its dynamics).

Thus, at the highest level of abstraction, a dynamic system is seen as a black box which has an input, an output and an internal structure as shown in the following figure:



The external behavior of the system is defined by the relationship between the history of inputs and the history of observed output results. In other words, it characterizes how the system reacts from the observer's point of view at the input / output level (I/O).

The internal structure of the system is defined according to three parameters:

- The *system state* which is generally represented by one or more variables called **state variables**.
- The *system state change mechanism*, which refers to how *state variables* change based on inputs or on their own. This aspect of *system dynamics* is modeled by what is generally referred to as **the state transition function**.
- The *production mechanism* of the system which refers to how the system produces an output result based on its internal state. This is called the **system output function**.

Generally, the last two points are implicitly grouped together when talking about the dynamics of the system which concerns the mechanisms of evolution of the system over time, as opposed to the state of the system which refers to the situation in which the system is at a specific time step.

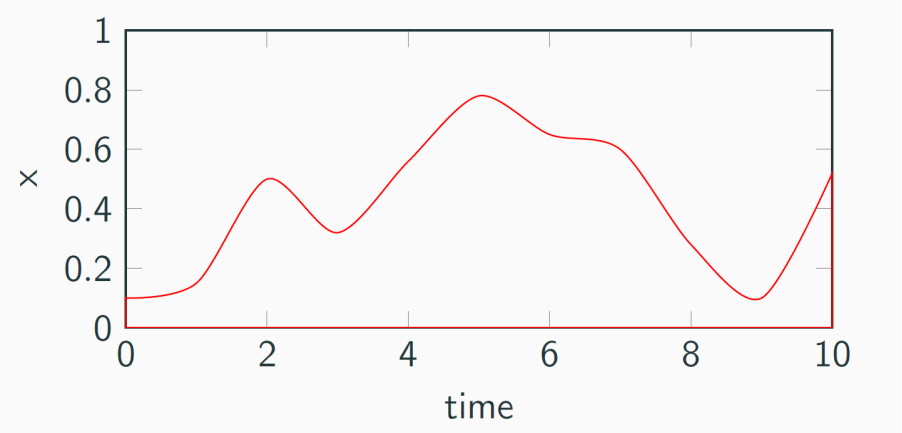
The important point to remember here is the distinction that systems theory makes between the different aspects of a *dynamic system*: behavior and internal structure.

10.5 5. The different temporal models used for the representation of dynamic systems

A dynamic system is defined by how it evolves over time. So, one of the most important features of a model which represent a dynamic system is how time is represented.

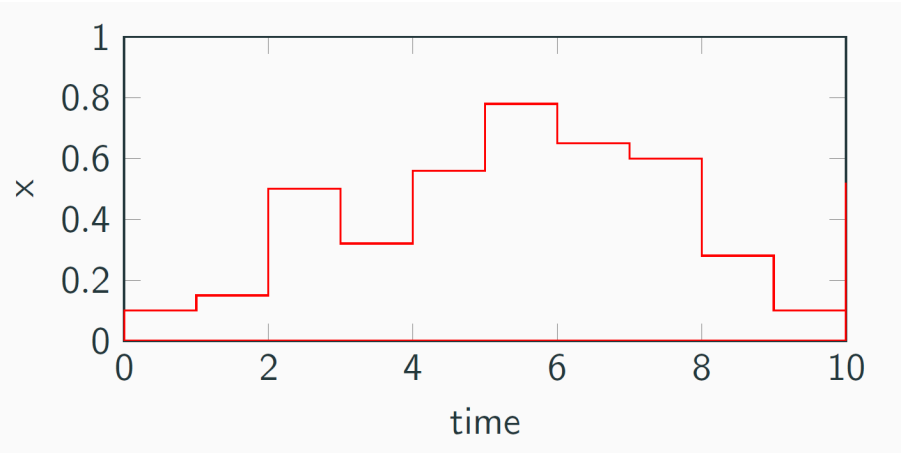
There are three types of time representations:

- *Continuous* time models: in a finite time interval, the system state variables change in value infinitely often, ie continuously.

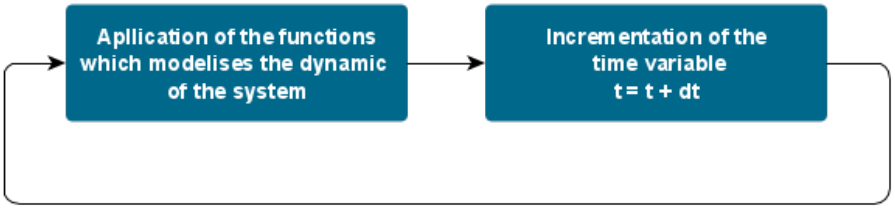


Warning: The simulation of continuous models raises many problems due to the nature of the computer: it is simply impossible to reproduce the continuity of the dynamics of a system because it evolves infinitely often while the computer simulation needs punctual computations.

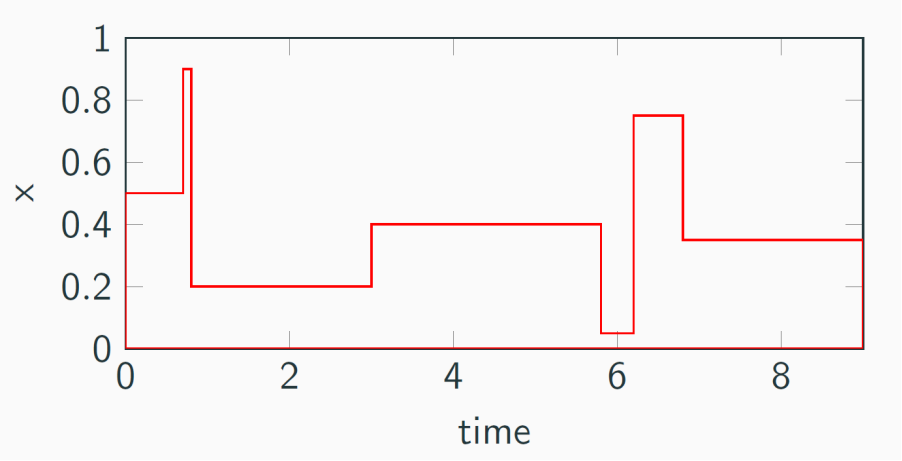
- *Discrete* time models: the time axis is discretized according to a constant period of time called time step (dt). The evolution of the system state variables is done in a discrete way, ie instantaneous, from t to t + dt.



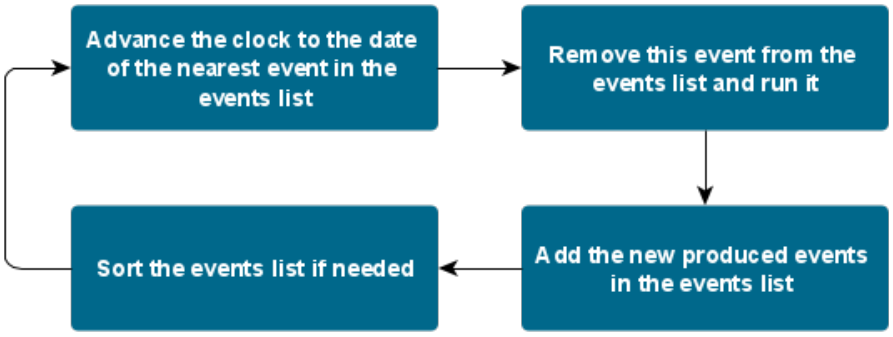
Note: The simulation of discrete models is easy. When the functions which implement the dynamics of the system are clearly defined, it is only necessary to set up an algorithm that applies these functions and then increments time.



- *Discrete event* time models: the time axis is generally continuous, ie represented by a real number. However, unlike continuous models, system state variables change discretely to specific times that are called *events*.



Note: There are three ways to simulate discrete event time models: (1) activity scanning, (2) interaction process and (3) event sceduling. The last one is the most common way and is described by the figure below.



10.6 6. Modeling & Simulation theory

Because to what it allows, computer simulation should not be only considered as a tool but as a discipline and should therefore be theorized. This is the goal of Zeigler's *Modeling and Simulation theory*.

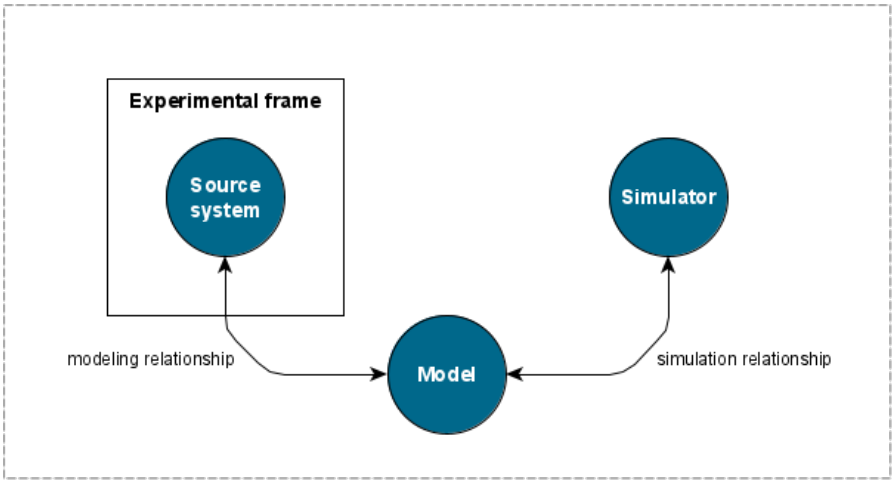
Thereby, the Modeling and Simulation theory helps to identify the different entities that constitute a simulation and study the relations that exist between these different entities. This is to give precise definitions to the different concepts that are manipulated in the field of computer simulation.

There are 6 entities clearly defined in the Modeling and Simulation theory:

- The *source system* and its behavioral database. The source system corresponds to the environment to be modeled. It must be seen as a source of observable data that constitutes what is called the behavioral database.
- The *experimental scope*. It is a specification of (1) the observing conditions of the system and of (2) the objectives of the simulation project.
- The *model*. It refers to the specification of all the instructions used to generate the behavior of the system.
- The *simulator*. It refers to any computing system capable of executing the model and generate its behavior. By separating a model from its simulator, a model can be run by different simulators which increases its portability.
- The *simulation relationship*. It defines the notion of validity of the model. *Does the modeling which is made of the system is an acceptable simplification of this one according to the qualitative criteria chosen and the*

objectives of the experimentation ?

- The *modeling relationship*. It defines the notion of validity of the simulator. *Does the simulator correctly generates the behavior of the model ? Does the simulator reproduces the mechanisms defined in the model (without introducing errors) ?*



The Modeling and Simulation theory makes it possible to highlight, through the modeling and simulation relationships, important issues related to simulation such as the simplicity of the model, the behavior of the simulated model, the validation of the simulation, the reproducibility of the simulation, etc. We will discuss some of these issues in more detail later.

CHAPTER

ELEVEN

2.3. MULTI-AGENT BASED SIMULATION

11.1 1. Difficulties of classical modeling

To model dynamic systems, a mathematical approach has long been used. Called Equation Based Model (EBM), it is build on an interrelation of a set of equations that captures the variability of a system over time. EBM represents the whole system and does not support an explicit representation of components (top-down). The whole system is globally represented by equations defined at the macroscopic level which does not make it possible to take into account the individual characteristics of the entities that compose the system. Moreover, this level of representation raises the question of the realism and the complexity of the parameters used to formulate the equations which model the system. So, EBM is most naturally used to model central systems.

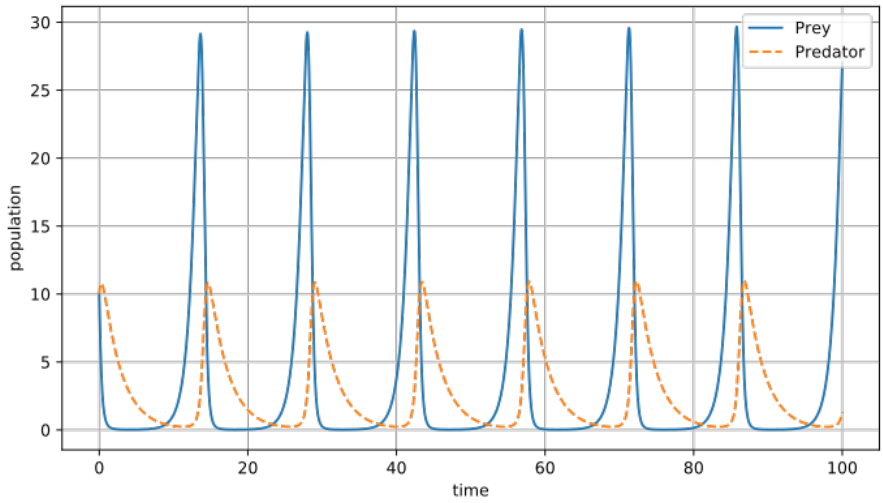
Note: Take the example of the Prey - Predatory model: it is used to describe the dynamics of biological systems in which two species interact, one as a predator and the other as prey.

$$\frac{dx(t)}{dt} = a * x(t) - b * x(t) * y(t)$$

$$\frac{dy(t)}{dt} = c * x(t) * y(t) - d * y(t)$$

$x(t)$	=	number of preys over time
$y(t)$	=	number of predators over time
a	=	preys reproduction rate
b	=	preys death rate
c	=	predators reproduction rate
d	=	predators death rate

It focus on the global variation of the prey and predator populations. Below, you can find the output of a Prey - Predator EBM simulation:



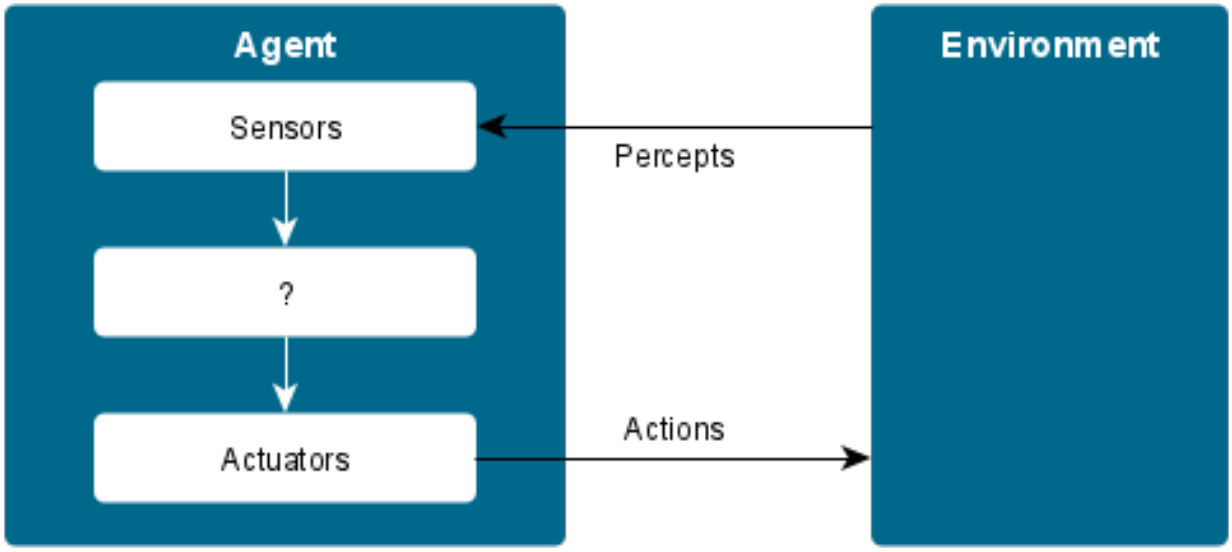
Warning: Limits of the Equation Based Model (EBM):

- Large number of parameters sometimes hard to understand.
- Diffculty to move from macro to micro level.
- Does not represent behaviors but behaviors results.
- Difficulty to represent behaviors.
- Does not represent interactions and organizations.

To overcome EBM limitations, it is possible to use models that focus on entities and their interactions (bottom-up) and also considers that the dynamics of the system come from the interactions between the entities.

11.2 2. An agent is a dynamic system

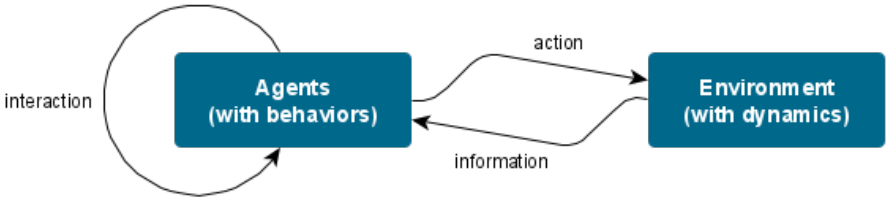
Modeling a dynamic system using agents is simple because it is obvious that an agent can be considered as a dynamic system within the meaning of systems theory. Indeed, an agent has an internal state that changes according to his perceptions (inputs) and it has behaviors to react in its environment.



As shown in the figure above, an agent is defined by a set of perceptions (inputs), a set of actions (outputs) and we speak of the internal architecture of the agent to denote the mechanisms that define its intrinsic dynamics.

11.3 3. Multi-agent modeling

The multi-agent approach is based on an individual-centered approach: it considers that it is possible to model not only individuals and their behaviors, but also the interactions that take place between these individuals. It considers that the overall dynamics of a system, at the macroscopic level, result directly from the interactions between the individuals who compose this system at the microscopic level. Thus, while EBM model the relationships that exist between the different identified entities of a system using mathematical equations, the multi-agent approach directly models the interactions generated by individual behaviors.



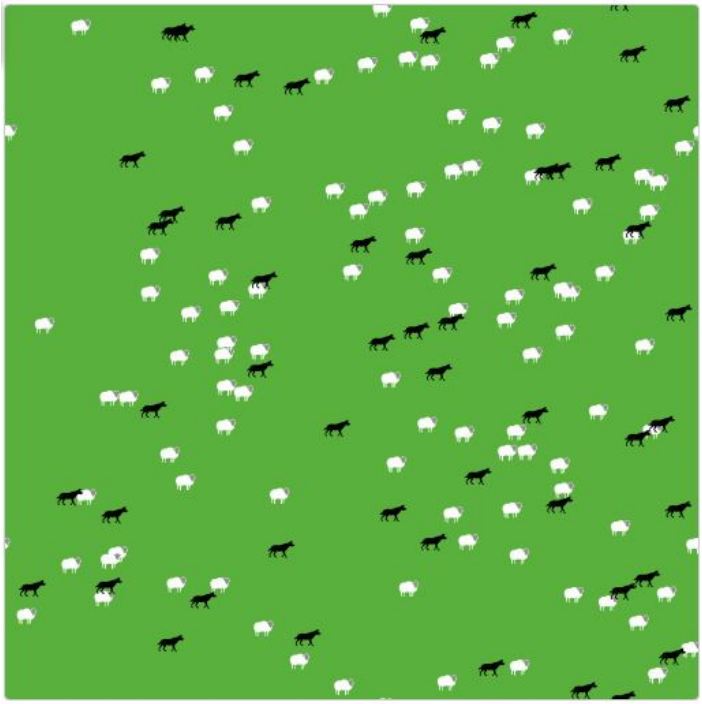
To create a Multi-Agent Based Simulation, you must follow the principles of the agent paradigm presented in the previous course:

- Creating an artificial world made up of interacting agents (Agent-Based Models, ABM) living in an environment.
- Each agent is described as an autonomous entity.
- The behaviour of agents is the consequence of their observations, internal trends, beliefs and interactions with the environment and other agents.
- Agents act and change the state of the environment through their actions.

Note: Take again the example of the Prey - Predator model. Each entity (prey and predator) is represented by an agent and has behaviors:

- Prey: move, eat, reproduce, flee and die.
- Predator: move, eat, reproduce, hunt and die.

Each agent activate one of its behavior according to its perception of the environment. This simulation focus on the local behaviors of the prey and predator populations. Below, you can find the output of a Prey - Predator EBM simulation:



Due to its ability to model the interactions between entities and the variations of a system at the micro level, Multi-Agent Based Simulation is used in many fields of application.

- Special effects in movies:

Fig. 1: *Massive software for crowd simulation*

- Video games:
- Simulation of complex systems:

Fig. 2: *Glassbox engine for SimCity*

Fig. 3: *Flock of birds simulation*

- Collective robotics:

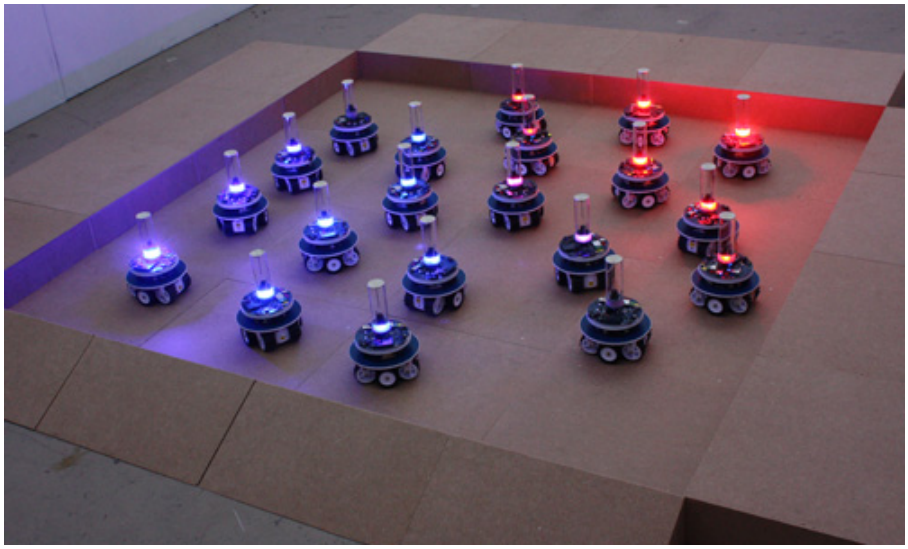


Fig. 4: *Collective behaviors for cooperative actions*

CHAPTER

TWELVE

2.4. VISUALIZING MULTI-AGENT BASED SIMULATION

As we have seen, simulation creates a digital environment that looks like an artificial laboratory where we can test hypotheses, prospective scenarios or simply the evolution of the system in the future.

As with any laboratory experiment, it is necessary to be able to follow the evolution of the experiments: with Multi-Agent based Simulation, it is necessary to visualize the evolution of the simulation and to collect data.

The visualization of the simulation is very important in the context of Multi-Agent based Simulation because the simple and localized rules that allow to reproduce complex and global behaviors of the system are also responsible of the emergence of many phenomena. You can learn more about the emergence capabilities of multi-agent systems at this address: https://medium.com/scalian/dynamiques-locales-cons%C3%A9quences-globales-97a2ef44ba58

In this section, we will present the two features of Mesa that make it possible to make data collection and visualization.

12.1 1. Collecting data

Mesa provides a class which can handle data collection and storage at runtime, which avoids to manipulates tables or dictionaries manually in the model. All this is done via the `DataCollector` class:

```
from mesa.datacollection import DataCollector
```

The data collector stores three categories of data: model-level variables, agent-level variables, and tables (which are a catch-all for everything else). Let's start from the previous example: the *Money Model*.

12.1.1 Agent-level

First, let us consider an agent-level variable. Imagine we want to store the value of each agent's wealth at each time step. We must add in the `MoneyModel` constructor a `DataCollector` with the parameter `agent_reporters` set to a dictionary that associates the label of the data and the name of the collected variable:

```
self.datacollector = DataCollector(agent_reporters={"Wealth": "wealth"})
```

We now simply tell the model to collect data with:

```
self.datacollector.collect(self)
```

each time we want to store values (*e.g.* right before or after we call `self.schedule.step()`).

Once the execution is finished, we can retrieve the data with:

```
model.datacollector.get_agent_vars_dataframe()
```

The result is a Pandas `DataFrame`. See the Pandas documentation for more information. You might also want to refer to the user guide for information about Pandas' data visualization.

Question

Add this data collector to your model and visualize the health value of one or several agents.

12.1.2 Model-level

It is also possible to use functions in the collector. For example, assume we want to collect the value of the Gini Coefficient, a measure of wealth inequality, at each time step on the model. Let's define the `compute_gini` method:

```
def compute_gini(model):
    agent_wealths = [agent.wealth for agent in model.schedule.agents]
    x = sorted(agent_wealths)
    N = model.num_agents
    B = sum( xi * (N-i) for i,xi in enumerate(x) ) / (N*sum(x))
    return (1 + (1/N) - 2*B)
```

Now, we can collect this data at runtime using the `DataCollector` with the parameter `model_reporters` set to a dictionary that associates the label of the data and the above function:

```
self.datacollector = DataCollector(
    model_reporters={"Gini": compute_gini},
    agent_reporters={"Wealth": "wealth"})
```

Note that we still collect the agents' wealth values.

Question

Visualize the evolution of the Gini Coefficient in the simulation. What do you observe ?

12.2 2. Dynamic visualization

So far, we've built a model, run it, and analyzed some output afterwards. However, one of the advantages of agent-based models is that we can often watch them run step by step, potentially spotting unexpected patterns, behaviors or bugs, or developing new intuitions, hypotheses, or insights. Other times, watching a model run can explain it to an unfamiliar audience better than static explanations. Like many ABM frameworks, Mesa allows you to create an interactive visualization of the model. In this section we'll walk through creating a visualization using built-in components.

First, a quick explanation of how Mesa's interactive visualization works. Visualization is done in a browser window, using JavaScript to draw the different things being visualized at each step of the model. To do this, Mesa launches a small web server, which runs the model, turns each step into a JSON object (essentially, structured plain text) and sends those steps to the browser.

A visualization is built up of a few different modules: for example, a module for drawing agents on a grid, and another one for drawing a chart of some variable. Each module has a Python part, which runs on the server and turns a model state into JSON data; and a JavaScript side, which takes that JSON data and draws it in the browser window. Mesa comes with a few modules built in, and let you add your own as well.

12.2.1 Grid Visualization

To start with, let's have a visualization where we can watch the agents moving around the grid. For this, you will need to put your model code in a separate Python source file; for example, `MoneyModel.py`. Next, either in the same file or in a new one (*e.g.* `MoneyModel_Viz.py`) import the server class and the `CanvasGrid` class (so-called because it uses HTML5 canvas to draw a grid). If you're in a new file, you'll also need to import the actual model object.

```
from mesa.visualization.modules import CanvasGrid
from mesa.visualization.ModularVisualization import ModularServer

# If MoneyModel.py is where your code is:
# from MoneyModel import MoneyModel
```

`CanvasGrid` works by looping over every cell in a grid, and generating a portrayal for every agent it finds. A portrayal is a dictionary (which can easily be turned into a JSON object) which tells the JavaScript side how to draw it. The only thing we need to provide is a function which takes an agent, and returns a portrayal object. Here's the simplest one: it'll draw each agent as a red, filled circle which fills half of each cell.

```
def agent_portrayal(agent):
    portrayal = {"Shape": "circle",
                 "Color": "red",
                 "Filled": "true",
                 "Layer": 0,
                 "r": 0.5}
    return portrayal
```

In addition to the portrayal method, we instantiate a canvas grid with its width and height in cells, and in pixels. In this case, let's create a 10 x 10 grid, drawn in 500 x 500 pixels.

```
grid = CanvasGrid(agent_portrayal, 10, 10, 500, 500)
```

Now we create and launch the actual server. We do this with the following arguments:

- The model class we're running and visualizing; in this case, `MoneyModel`.
- A list of module objects to include in the visualization; here, just `[grid]`.
- The title of the model: `"Money Model"`.
- Any inputs or arguments for the model itself. In this case, 100 agents, and height and width of 10.

Once we create the server, we set the port for it to listen on (you can treat this as just a piece of the URL you'll open in the browser). Finally, when you're ready to run the visualization, use the server's `launch()` method.

```
server = ModularServer(MoneyModel,
                       [grid],
                       "Money Model",
                       {"N":100, "width":10, "height":10})
server.port = 8521 # The default
server.launch()
```

The full code should now look like:

```
from MoneyModel import *
from mesa.visualization.modules import CanvasGrid
from mesa.visualization.ModularVisualization import ModularServer

def agent_portrayal(agent):
    portrayal = {"Shape": "circle",
```

(continues on next page)

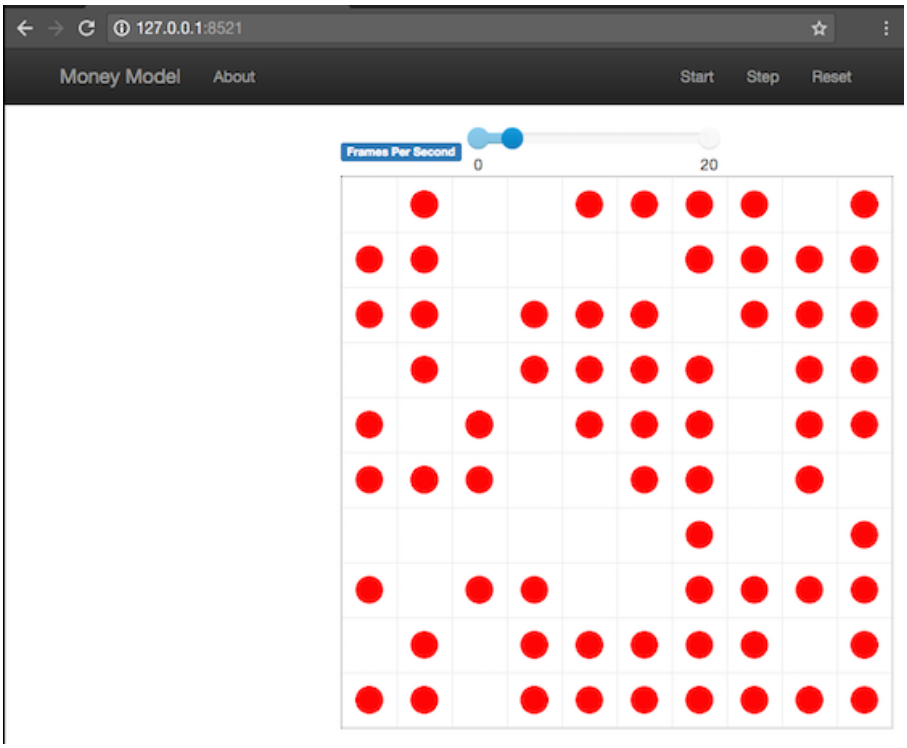
(continued from previous page)

```
                 "Filled": "true",
                 "Layer": 0,
                 "Color": "red",
                 "r": 0.5}
    return portrayal

grid = CanvasGrid(agent_portrayal, 10, 10, 500, 500)
server = ModularServer(MoneyModel,
                       [grid],
                       "Money Model",
                       {"N":100, "width":10, "height":10})
server.port = 8521 # The default
server.launch()
```

Now run this file; this should launch the interactive visualization server and open your web browser automatically. (If the browser doesn't open automatically, try pointing it at `http://127.0.0.1:8521` manually. If this doesn't show you the visualization, something may have gone wrong with the server launch.)

You should see something like the figure below: the model title, a grid filled with red circles representing agents, and a set of buttons to the right for running and resetting the model.



- Click `step` to advance the model by one step, and the agents will move around.
- Click `run` and the agents will keep moving around, at the rate set by the `fps` (frames per second) slider at the top. Try moving it around and see how the speed of the model changes.
- Pressing `pause` will (as you'd expect) pause the model; pressing `run` again will restart it.
- Finally, `reset` will start a new instantiation of the model.

To stop the visualization server, go back to the terminal where you launched it, and press `Control+c`.

12.2.2 Changing the agents

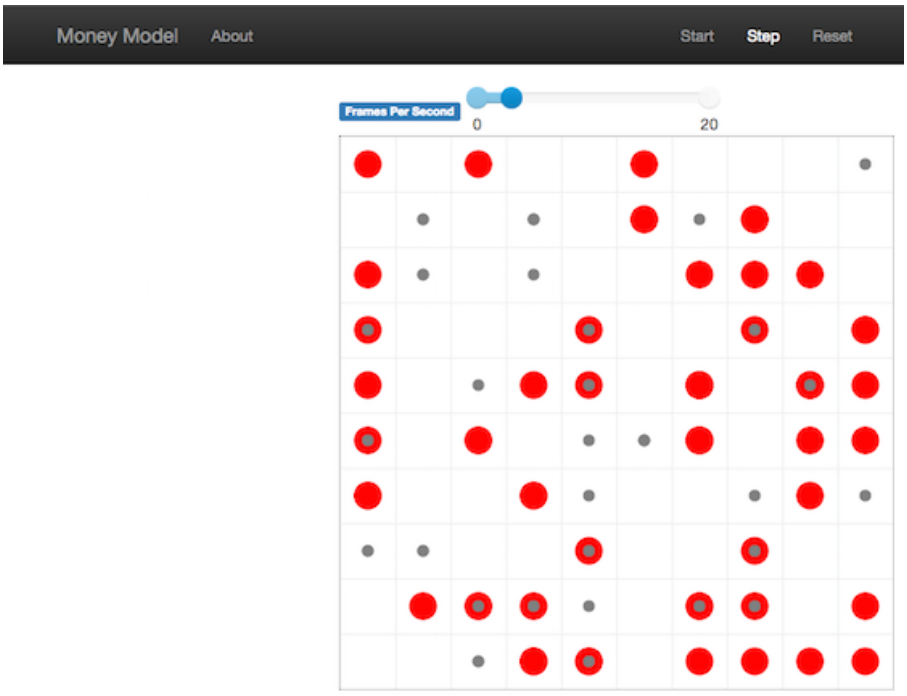
In the visualization above, all we could see is the agents moving around – but not how much money they had, or anything else of interest. Let's change it so that agents who are broke (wealth 0) are drawn in grey, smaller, and above agents who still have money.

To do this, we go back to our agent_portrayal code and add some code to change the portrayal based on the agent properties.

```
def agent_portrayal(agent):
    portrayal = {"Shape": "circle",
                 "Filled": "true",
                 "r": 0.5}

    if agent.wealth > 0:
        portrayal["Color"] = "red"
        portrayal["Layer"] = 0
    else:
        portrayal["Color"] = "grey"
        portrayal["Layer"] = 1
        portrayal["r"] = 0.2
    return portrayal
```

Now launch the server again - this will open a new browser window pointed at the updated visualization. Initially it looks the same, but advance the model and smaller grey circles start to appear. Note that since the zero-wealth agents have a higher layer number, they are drawn on top of the red agents.



12.2.3 Adding a chart

Next, let's add another element to the visualization: a chart, tracking the model's Gini Coefficient. This is another built-in element that Mesa provides.

```
from mesa.visualization.modules import ChartModule
```

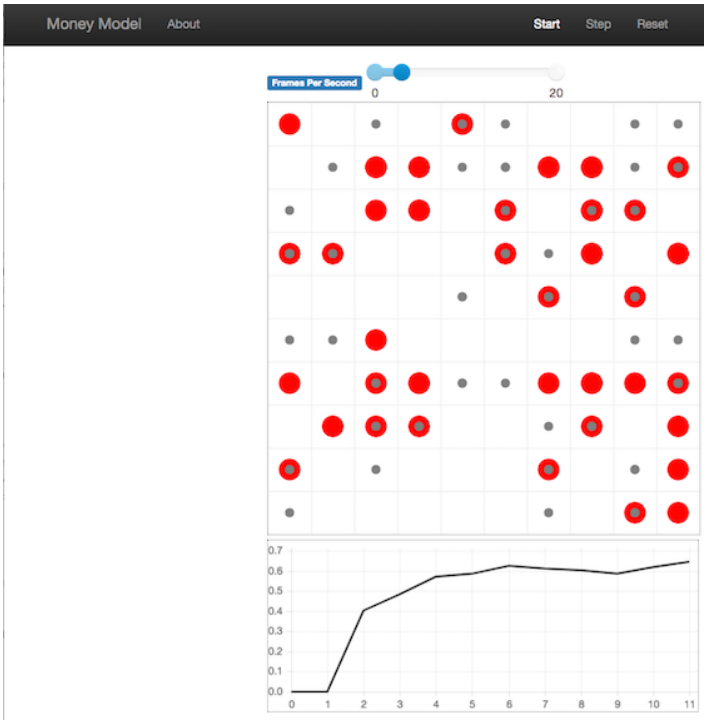
The basic chart pulls data from the model's `DataCollector`, and draws it as a line graph using the `Charts.js` JavaScript libraries. We instantiate a chart element with a list of series for the chart to track. Each series is defined in a dictionary, and has a Label (which must match the name of a model-level variable collected by the `DataCollector`) and a `Color` name. We can also give the chart the name of the `DataCollector` object in the model.

Finally, we add the chart to the list of elements in the server. The elements are added to the visualization in the order they appear, so the chart will appear underneath the grid.

```
chart = ChartModule([{"Label": "Gini",
                      "Color": "Black"}],
                    data_collector_name='datacollector')

server = ModularServer(MoneyModel,
                       [grid, chart],
                       "Money Model",
                       {"N":100, "width":10, "height":10})
```

Launch the visualization and start a model run, and you'll see a line chart underneath the grid. Every step of the model, the line chart updates along with the grid. Reset the model, and the chart resets too.



12.2.4 User settable parameter

The last part of the tutorial helps you to create a visualization interface in order to see the evolution of the ABM in the simulation. To allow faster modification of model parameters, it is possible to use `UserSettableParameter`. `UserSettableParameter` means that the user can modify this parameter in the web page. It takes 6 parameters (type, name, initial value, min value, max value, value per step).

Let's start by importing the right mesa packages:

```
from mesa.visualization.modules import CanvasGrid, ChartModule, TextElement
from mesa.visualization.UserParam import UserSettableParameter
```

Then, add the following line in the definition of the `ModularServer`:

```
"density": UserSettableParameter("slider", "Agent density", 0.8, 0.1, 1.0, 0.1),
```

The call of the `ModularServer` became:

```
server = ModularServer(MoneyModel,
                       [grid, chart],
                       "Money Model",
                       {"width":10, "height":10, "density": UserSettableParameter("slider
↪", "Agent density", 0.8, 0.1, 1.0, 0.1)})
```

Note: to be used in the visualization interface, the density should be define in the `MoneyModel`.

CHAPTER

THIRTEEN

2.5. IMPLEMENTING MULTI-AGENT BASED SIMULATION

From a technical point of view, the characteristics of a Multi-Agent Based Simulaton are related to the software architecture used to implement the agents and the environment. In this context, the needs and constraints relating to the software architecture can vary due to the heterogeneity of the fields on which Multi-Agent Based Simulation can be applied. The structures of the models representing the agents and the environment are not the same depending on the experimental framework (biology, ecology, ethology, etc.).

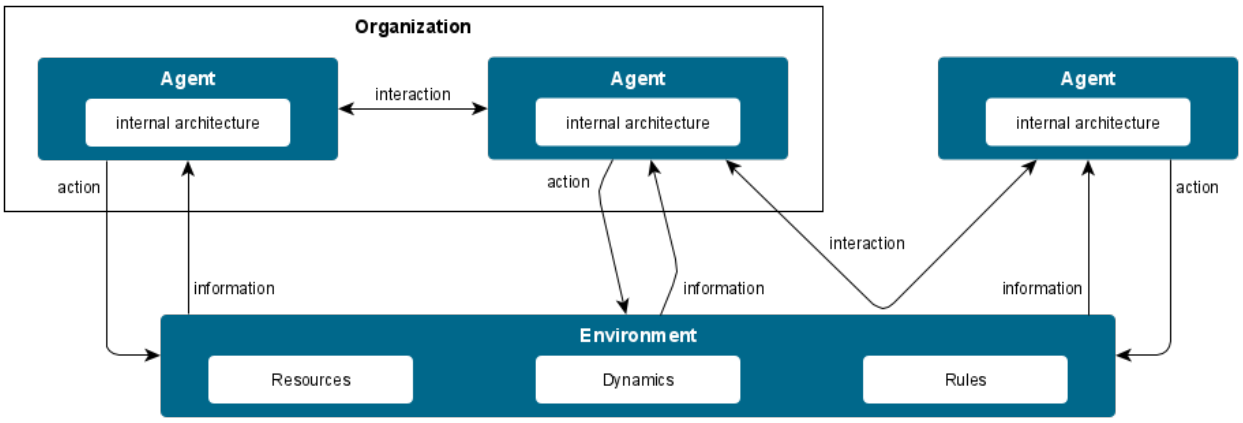
13.1 1. Architecture of a Multi-Agent Base Simulation

The software architecture of a Multi-Agent Base Simulation is strongly influenced by the nature of the source system that we want to model. This is one of the reasons why there is currently no generic methodology linked to the modeling and the implementation of Multi-Agent Base Simulation.

Consequently, the nature of the environment (**continuous** or **discrete**, **static** or **dynamic**, etc.), the granularity of actions / perceptions (**fine** or **coarse**) and the **complexity of the internal architecture** of an agent are all points on which the modeler has to make personal choices which are based on his needs. The large number of existing platforms is proof of this (see the *note* below).

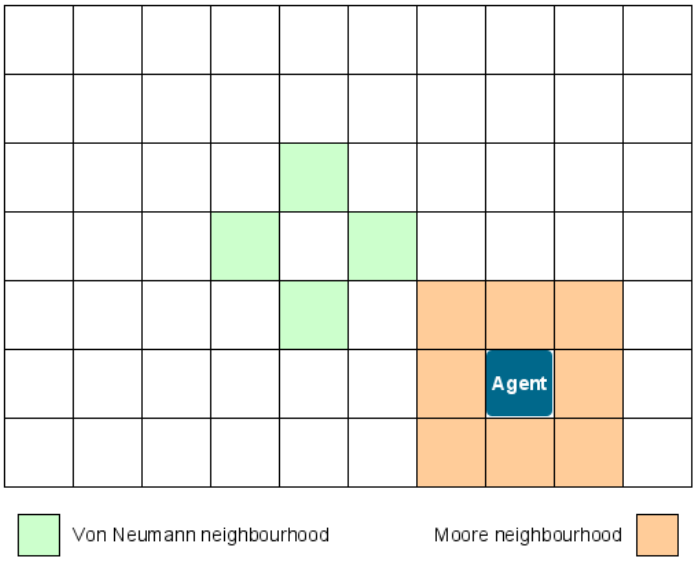
Thus, when we want to implement a simulation, we must clearly define :

- The architecture of autonomous entities that act in the system (Agent);
- The interactions between entities (Interaction);
- The shared resources and processes between agents (Environment);
- The coordination and cooperation between agents (Organization).

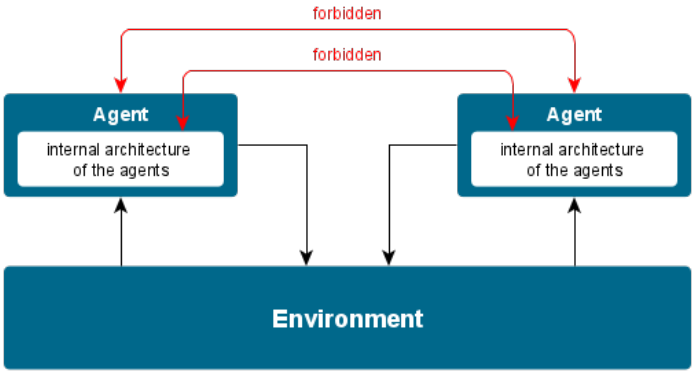


The implementation of multi-agent models also requires to take into account three constraints:

1. Genericity and modularity constraint Individual and autonomous entities are modeled and implemented. We want to be able to vary their implementation without questioning the entire source code.
2. Locality constraint The environment is not fully accessible to an agent. An agent has only a local perception of the world and his actions/perceptions are limited in their scope.



3. Environmental integrity constraint There is no direct relationship between the internal architectures of two autonomous agents. An agent must not be able to directly change environment state variables.



Note: Here is a small list of existing platforms with advantages and disadvantages:

- Specialized tools for Multi-Agent Based Simulation: ATOM, ARCHISIM, MATSim, MITSIMlab
  - (+) Allow to reuse some parts of models.
  - (-) Models need adaptation to be used in a new platform.
- Generic tools for Multi-Agent Based Simulation: AnyLogic, GAMA, MASON, SWARM, CORMAS, TurtleKit, Repast, Netlogo, JASMIN
  - (+) Possible to use the same tool for different models.
  - (-) Need to create a single operational model for each simulation.
- Generic tools for multi-agent softwares: Jason, JADE/TAPAS, JADE/PlaSMA, MADKIT
  - (+) General knowledge of the tool (easy to use).

- (-) Need adaptation to implement simulation models.

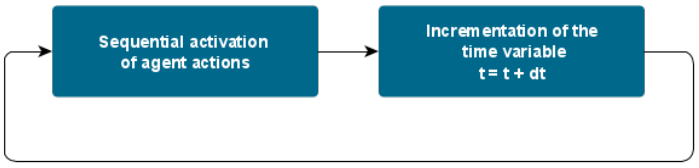
13.2 2. Time management in MABS

Multi-Agent Based Simulation is based on the idea that it is possible to represent a set of autonomous entities operating in a common environment. In this context, all agents must be subject to the same temporal law in order to respect the principle of causality (see [Edem Fianyo and all, The Representation of Time, 1998] for more details).

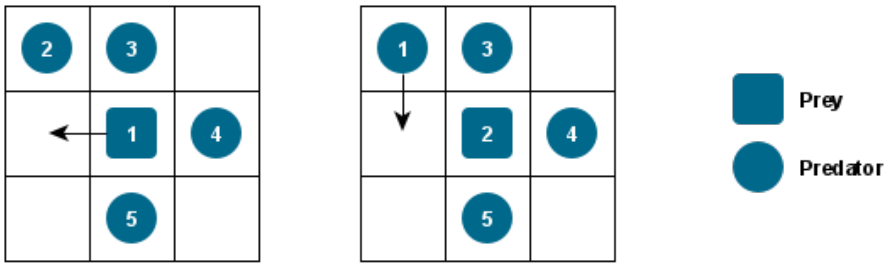
Note: It must be noticed that all parameters that shape an agent's internal architecture and model its behaviour (desires, beliefs, world representation, etc.) change instantaneously.

The behaviour of an agent is inherently discrete. The two main implementation principles for handling discrete events are:

- Synchronous approach: regular discretization of time. Very simple to use because it only need to program a very simple function that activates all the agents behaviors.

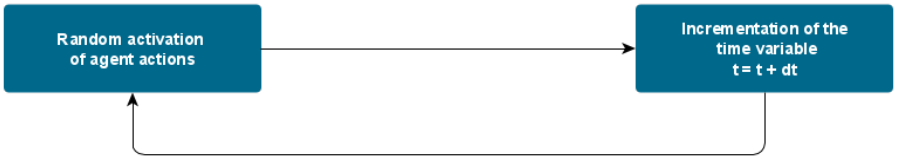


With a synchronous apporach, the activation order of the agents has a direct impact on the evolution of the world. In the Prey - Predator example, if a prey is captured when it is surrounded by four predators following the vicinity of Von Neumann, the simple fact of changing the order of activation of the agents can have a strong impact on the final result. In the figure below, the right activation make the prey died, not the left activation.



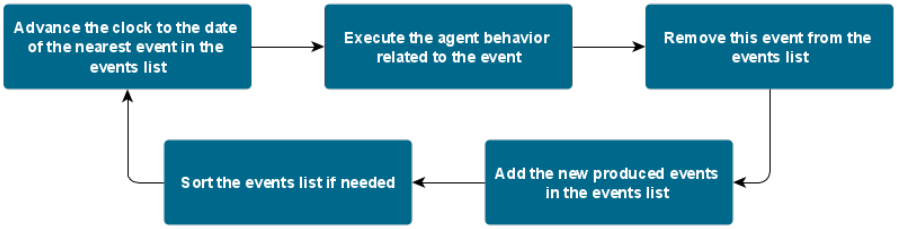
From the same state of the world, the same behaviors can produce different states of the world according to the order of activation of the agents. To overcome this issue, it is possible to use:

- a *random activation process*



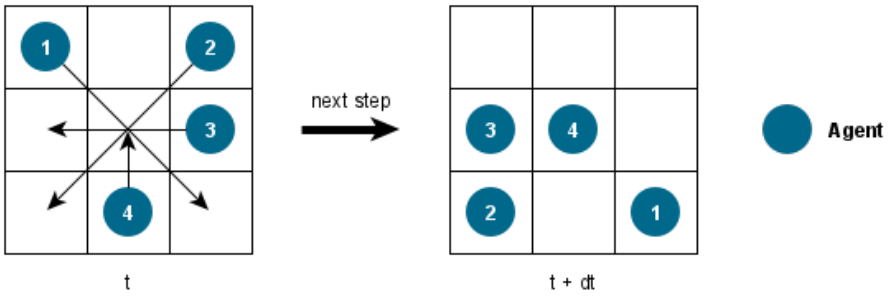
- *buffer variables*

- Asynchronous approach: event simulation principle. It is hard to picture out a reality where a set of entities would be updated simultaneously by a global clock. It is more natural that each autonomous entity evolves at its own pace and generates events whose date of realization is personal to it. So, the agent activation is asynchronous. There are two ways to implement asynchronous approach:
  - Events are determined at the beginning and the simulation consists in executing the list of events.
  - Events are determined during the simulation to take into account the effects of previous events.



As in synchronous approach, an asynchronous approach could produce some conflicts: for example, two events can occur at the same simulation time step. To resolve these conflits, it is possible to introduce competition between agents or to make a random choice on the action to be prioritized.

Warning: To conclude this part, it must be noticed that a consistency relationship between the time granularity of the actions and the time step must exist.



If an agent can move on many cells in one time step. Several agents may have paths that intersect without ever colliding (speed not directly related to time). So, the temporal granularity of actions must be taken into account.

Note: To go further, you may be interested in the problem of simultaneity of actions in Multi-Agent Based Simulation.

CHAPTER

FOURTEEN

2.6. IMPLEMENTING A PREY - PREDATOR SIMULATION WITH MESA

The objective of this practical work is to implement a simulation of a Prey - Predator model. The Prey - Predator model is a simple ecological model, consisting of three agent types: wolves, sheep, and grass. The wolves and the sheep wander around the grid at random. Wolves and sheep both expend energy moving around, and replenish it by eating. Sheep eat grass, and wolves eat sheep if they end up on the same grid cell.

If wolves and sheep have enough energy, they reproduce, creating a new wolf or sheep (in this simplified model, only one parent is needed for reproduction). The grass on each cell regrows at a constant rate. If any wolves and sheep run out of energy, they die.

The implementation of this model will make you use several Mesa concepts and features:

- MultiGrid.
- Multiple agent types (wolves, sheep, grass).
- Overlay arbitrary text (wolf's energy) on agent's shapes while drawing on CanvasGrid.
- Agents inheriting a behavior (random movement) from an abstract parent
- Writing a model composed of multiple files.
- Dynamically adding and removing agents from the schedule.

The goal is to implement a Prey - Predator model closely based on the NetLogo Wolf-Sheep Predation Model: *Wilensky, U. (1997). NetLogo Wolf Sheep Predation model*. http://ccl.northwestern.edu/netlogo/models/WolfSheepPredation

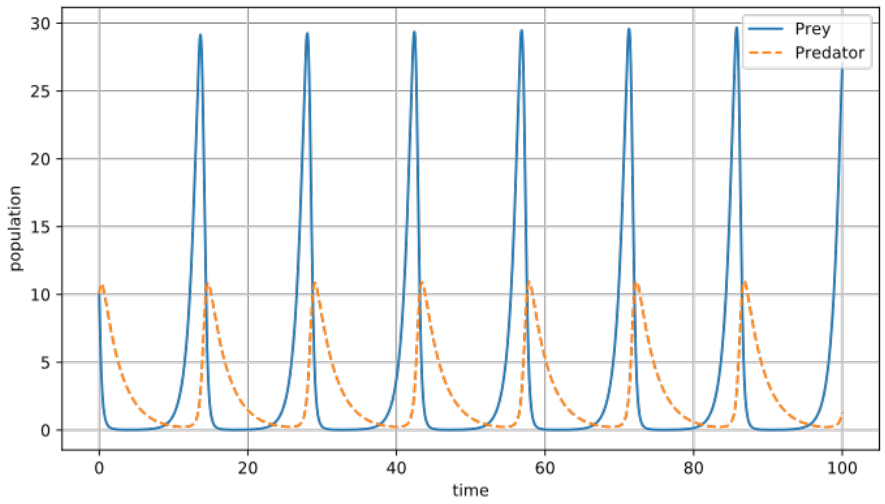
14.1 Practice yourself!

Download the following archive which contains the project from which to start and extract it. The extracted folder contains 6 python files:

- *prey_predator/random_walker.py*: this defines the *RandomWalker* agent, which implements the behavior of moving randomly across a grid, one cell at a time. Both the Wolf and Sheep agents will inherit from it.
- *prey_predator/agents.py*: defines the Wolf, Sheep, and GrassPatch agent classes.
- *prey_predator/schedule.py*: defines a custom variant on the RandomActivation scheduler, where all agents of one class are activated (in random order) before the next class goes – e.g. all the wolves go, then all the sheep, then all the grass.
- *prey_predator/model.py*: defines the Prey - Predator model itself.
- *prey_predator/server.py*: sets up the interactive visualization server
- *run.py*: launches a model visualization server.

So, now it's up to you to work and implement the Prey - Predator model:

1. Defines the Wolf, Sheep, and GrassPatch agent classes in the *prey_predator/agents.py* file.
   - A sheep that walks around, reproduces (asexually) and gets eaten.
   - A wolf that walks around, reproduces (asexually) and eats sheep.
   - A patch of grass that grows at a fixed rate and it is eaten by sheep.
2. Defines the Prey - Predator model by completing the *prey_predator/model.py* file.
3. Sets up the interactive visualization server by completing *prey_predator/server.py* file.
   - Display the different agents on the grid.
   - Added buttons to control the initial settings.
4. Tune the initial parameters to find a balanced state in the model: none of the species disappears during the simulation. The output graph should be like that:



CHAPTER

FIFTEEN

3. INTERACTION MECHANISMS : MODELS AND IMPLEMENTATION

15.1 Agenda – Recall

1. Friday, March the 5th, 2021 : Introduction to MAS : definitions and implementation of a platform
2. Friday, March the 12th, 2021 : Multiagent simulation : preys and predators
3. Friday, March the 19th, 2021 : Interaction mechanisms : models and implementation
4. Friday, March the 26th, 2021 : Argumentation-based negotiation I: Practical session
5. Friday, April the 2nd, 2021 : Argumentation-based negotiation II: what is argumentation?
6. Friday, April the 9th, 2021 : Argumentation-based negotiation III: what is a negotiation protocol?
7. Friday, April the 16th, 2021 : Argumentation-based negotiation IV: Practical session

15.2 Session 3.

Multiagent systems is not just about having different processes (or piece of code) that run in parallel, as we did in the previous sessions. Indeed, agents need to share some information to work together.

In the Alice and Bob examples (session 1, section 2) as well as in the Multi-Agent Based Simulation you implemented in the second session, agents worked separately and acted, in a concurrent manner, on a shared variable in the environment. This very primitive type of interactions is called *indirect interactions*. We will give a brief presentation of indirect interaction mechanisms in the first section of today's course session.

In the Money example (session 1, section 4), agents act directly on their peers variables. This is not supposed to happen in a MAS: agents can only interact with their environment. However, it is often required that agents request modifications to other agents. Such mechanisms are called *direct interactions* and will be presented in the second section of this course session.

The third section will give you an overview of software engineering practices and tools that you should adopt as MAS designers.

The fourth section will present a Mesa-based library for direct interactions that we shall use in the remaining of this course.

As for the previous sessions, all implementation is done using the Python programming language and the Mesa library.

15.3 Some References

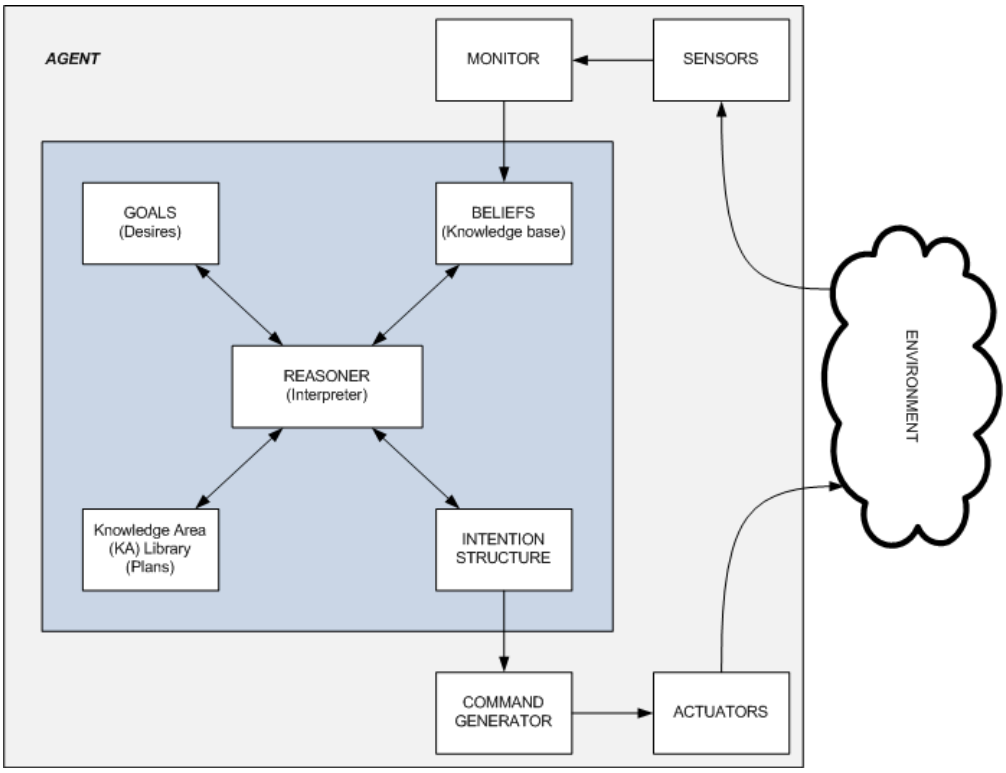
- Ferber, J. (1995), *Les Systèmes Multi-Agents*, InterEditions. **(French version)**
- Ferber, J. (1999), *Multi-agent systems: An introduction to distributed artificial intelligence*, Addison Wesley. **(English version)**
- Michael Wooldridge (2002), *An Introduction to MultiAgent Systems*, John Wiley & Sons Ltd.
- The AgentLink roadmap

CHAPTER

SIXTEEN

3.1. INDIRECT INTERACTIONS

Indirect interactions is the main interaction mechanism in MAS. Remember the PRS architecture:



Agents can only act (and perceive) from their environment. The idea behind indirect interactions is that agents that act on shared variables in the environment interact through this variable, even if they don't have the intention to do so.

16.1 1. Blackboards

The first multiagent platforms in the 90s implement *blackboards* in the environment. A blackboard is an array (or a python list) in which agents can write information that can be used by other agents.

Let us consider the Alice-Bob example again:

```
from time import sleep
from random import *
from threading import Thread
```

(continues on next page)

(continued from previous page)

```
class Environment:
    v = 0
    def act(self):
        self.v = 1 - self.v
        print(self.v)
    def perceive(self):
        return self.v

class Agent(Thread):
    def __init__(self, name, preferred_value, env):
        Thread.__init__(self)
        self.name = name
        self.env = env
        self.pv = preferred_value
    def run(self):
        while True:
            self.procedural_loop()
    def procedural_loop(self):
        if self.env.perceive() != self.pv:
            self.env.act()

class Runtime:
    def __init__(self):
        e = Environment()
        Agent("Alice", 0, e).start()
        Agent("Bob", 1, e).start()

Runtime()
```

Both agents act on a variable named "v" in the environment, which they try to maintain to their preferred value (note that we used the asynchronous version, but that was for a change; it would be the same in a synchronous implementation or in a Mesa model). The list of variables has to be defined in the environment, which does not make our implementation very *generic* or reusable. Providing the environment with a blackboard can achieve this feature.

In the following example, we implemented a **blackboard as a Python dictionary** in which agent can set variable values.

```
class Environment:
    blackboard = {}
    def act(self,name,value):
        self.blackboard[name] = value
    def perceive(self,name):
        if name in self.blackboard:
            return self.blackboard[name]
        return None
```

Based on this, the code for Alice and Bob agents could be:

```
class Agent(Thread):
    def __init__(self, name, preferred_value, env):
        Thread.__init__(self)
        self.name = name
        self.env = env
        self.pv = preferred_value
    def run(self):
        while True:
            self.procedural_loop()
```

(continues on next page)

(continued from previous page)

```
    def procedural_loop(self):
        if self.env.perceive("v") != self.pv:
            self.env.act("v",self.pv)
```

The idea is that all agents can now share variable or exchange any sort of information on the blackboard.

16.2 2. From Blackboards to Stigmergy

Blackboards architectures however were not very popular. There are two main reasons. The first one is that they assume a centralised mechanism (the blackboard) that is a possible failure point in the MAS. Some distributed architectures proposed distributed blackboards but they were soon abandoned in the profit of direct interactions (see next section). The other limit of blackboards is that all agents perceive the whole environment. Distributed AI designers tried to propose other models in which each agent has a local view on the environment, as we did in the Money example with agents exchanging wealth on their own cell only.

Such model gave rise to *spacially-situated* agent systems in which agents move on a spatial model (generally, a two dimensions grid) that they can observe. The content of each cell can be a mini-blackboard or any other data structure. Here is a possible implementation of such an environment :

```
class Environment:
    def __init__(self,width,height):
        self.grid = [ [{} for i in range(width)] for j in range(height) ]
    def act(self,agent,name,value):
        (x,y) = agent.get_current_position()
        self.grid[x][y][name] = value
    def perceive(self,agent,name):
        (x,y) = agent.get_current_position()
        bb = self.grid[x][y]
        if name in bb:
            return bb[name]
        return None

class Agent():
    def __init__(self, name, env, initial_position):
        self.name = name
        self.env = env
        self.pos = initial_position
    def get_current_position(self):
        return self.pos
    def move(self,new_position):
        self.pos = position
    ...
```

The name *stigmergy* comes from the behaviour of some social insects such as ants or termites. Such animals drop pheromones in the environment as they move around. These pheromones give information to other members of the colony about their movements (where they were, how long ago they were here...) but also about what they were doing at that time (carrying food, hunting...). **Spatially situated agent model with indirect interaction** took their inspiration in this biological model.

One important characteristics of *stigmergy* is that the information contained in the pheromones fades away with time. Multiagent models that implement stigmergy generally have a similar mechanism to alter the value of some variables. This means that **the environment also acts on its variable values**. This can be used by agent to follow another agent's track (using gradient descent) or to improve space coverage by searching for spaces with lower values of pheromones.

Note that agents in stygmergic environments generally are *reactive agents* (although nothing forbids them from being cognitive, see session 1, section 2).

16.2.1 Practice yourself (optional)

If you have time, try to implement a stigmergic model of a simple multiagent systems in Python (without using the Mesa library):

- The environment is a 10x10 grid space. It offers methods to move randomly in any of the four possible directions.
- The system runs in a synchronous manner.
- Each cell has a value, initially set to 100 that decreases by 1 at each turn.
- Agents can either move or remain on a cell at each turn. When they choose to remain, the value of the cell is increased by 10.
- Agents try to maintain all cells values above 50.

If you are good with graphical interfaces (*e.g.* tkinter), you can draw the grid's values and embed your runtime in the graphical mainloop to observe the simulation.

16.3 3. From Stigmergy to Artefacts

In the early 2000s, researchers in MAS began to question the environment: How passive should it be? Since stigmergic environments change their values with time, what really makes them different from an agent on which other agents would act? What is the difference between an object (in the environment) and an agent?

All these questions gave rise to a model called **Agents and Artifacts**, proposed by Alessandro Ricci and his colleagues from the University of Bologna. The A&A model became very popular because it unified different theories and proposed a clear distinction between objects, services and agents. Understanding all subtleties of A&A is out of the scope of this course and only few platforms use this model. However, its theoretical value is of importance.

To make a long story short, the environment in A&A is a set of *artifacts* which behave like reactive services. Each artifact has a set of variables that can be modified by agents (inputs), and a set of variables that can be observed by agents (outputs). Agents can create artifacts in the environment and tag them for other agents to notice them. Whenever an agent sets input values to the artifact, it begins its computation in an asynchronous manner *w.r.t.* other agents or artifacts. When this computation is over, it informs the initiator agent. Moreover, artifacts have some documentation (*e.g.* using the Web Service Description Language) for agents to interact with them without a priori knowledge. However artifacts are not **autonomous**, contrary to agents: they simply compute whatever is requested, they have no internal decision mechanism.

16.3.1 Conclusion on Indirect Interactions

Indirect interactions is still largely used, especially in Multi-Agent Based Simulation (as you did in the Prey-Predator simulation during the previous session). However, the most widespread interaction model in multiagent platforms (thanks to the FIPA stantard) is based on *direct interactions* that we present in the next section.

CHAPTER

SEVENTEEN

3.2. DIRECT INTERACTIONS

17.1 1. The Theory

Direct interactions in multiagent systems relies on a very strong assumption: the **intention** to communicate. Whereas agents in a stigmergic environment do not explicitely communicate, direct interactions means that agents know they are exchanging information with another agent. They control the communication.

The first consequence is that communication is done by **sending messages** from an agent to another (or to a group or others). There are a couple of theories that you need to understand before we actually implement direction communication in a MAS.

17.1.1 Communication levels

Communication has been widely studied in mathematics and computer science in the second half of the XXst Century, after the publication of Claude Shannon's A Mathematical Theory of Communication in 1948. One main contribution in the 70s is the Open Systems Interconnection (OSI) model that is now an ISO standard for computer networked systems, with 7 layers to describe the communication. However other models were proposed to study all sorts of communications.

Let us first take a look at some of these communication layers:

- The **physical** layer which is responsible for the transport of the message from one agent to another. Human beings cannot talk without ambiant air. Bees require visual space to perform their "dance" that indicates flowers positions. Physical layers are always required, and MAS won't be an exception;
- The **syntactic** layer which defines how elements of the message must be arranged. In human communication, we share grammatical rules that decide how sounds must be arranged to form words. Similarly, computer agents will require some message structure to ensure that they can communicate with each other. To this goal, we shall use a well-defined *Agent Communication Language* (ACL);
- The **knowledge** layer which defines the *meaning* of the message content. Most of us hardly understand the meaning of bird songs, but ethnologists proved that birds can exchange different kinds of information with their songs. Same thing for the ants with their pheromones. And, of course, words in our language do have specific meanings that need to be shared among a group of people for them to understand each others. This is also true for agents: the content of the messages must be attached to so-called *ontologies* that define how these concepts relate to the agent's activity.
- The **protocol** layer which defines some rules for the communication. When you call a friend on the phone, when you write an email, people expect you to exchange some messages that have no other purpose than establishing the communication (*e.g.* saying hello, saying goodbye, *etc*). Similarly, MAS engineering requires to define communication protocols for the agents to be able to perform the task the system was designed for.

In the following subsections, we will see how these layers are achieved in agent communication models.

17.1.2 Speech Acts Theory

The most important model for agent communication was not designed by a computer scientist. It is the *Speech Acts Theory* that was invented in 1955 by the philosopher John L. Austin and described in his famous book How to do things with words. Austin's work was pursued in the 60s by John R. Searle, another philosopher that largely contributed to computer science and linguistics.

Austin's proposal is that communication is a way to change one' interlocutor state of mind. When I write (or say) "Elephants are pink", I change your belief base. If you have no knowledge in Biology, I will have added the belief that elephants are pink in your mind. I **made you believe that elephants are pink**. If you have a more critical view on my knowledge about elephants, I will have at least **made you believe that I think that elephants are pink**. And even if none of this happen, I'll have **made you believe that I said that elephants are pink**. To make a long story short, when I communicate, I change your beliefs. Therefore, this is an **action** on your beliefs base. Hence the name: speech **acts** theory, which says that communication is a form of action.

Austin first identified three aspects of communication, *i.e.* three speech acts or *locutions*. The first one is the **locutionary act**. It is the action of producing a meaningful sentence. In my elephant example, saying that elephants are pink has some meaning, regardless from the actual truth of the assertion. The second (and most important for us) aspect is the **illocutionary act**. This is what is intended by the sender of the message. In my elephant example, it could be to inform you about some Biological knowledge, or to illustrate an example, or to amuse you. When you say "How are you doing?", the illocutionary act is either a traditional politeness formula or to obtain some information from your interlocutor about its current state. This is what agents will do in the MAS for direct communication. The last aspect of communication is the **perlocutionary act**, which is what is actually obtained from the communication. In MAS, we will generally assume that the communication is completely successful, *i.e.* that the perlocutionary act is equal to the illocutionary act.

John Searle's main contribution to this theory was to identify different categories of illocutionary acts. Searle propose to distinguish between:

- **Assertive** acts to send information: *Elephants are pink*
- **Directive** acts to request for some action: *Could you open the window?*, *What time is it?* (in this second example, the requested action is that you tell me the time)
- **Commissive** acts to commit to some action's achievement: *I'll do it!*
- **Declaration** acts to control the protocol: *I declare that today's session is over!*, *Hello!*
- **Expressive** acts: *I feel sad today.*

17.1.3 Performative and contents

Based on this, the speech acts theory propose to consider an illocutionary locution as a couple **(Performative,Content)** in which the performative defines the interpretation of the content.

Let us consider a concrete example with a computer-science perspective: if you consider the content `raining=True`, you can use it in different manner:

- `Assert(raining=true)`: the agent will inform the message's recipient that it is raining;
- `Question(raining=true)`: the agent wants to know whether it is raining or not;
- `Order(raining=true)`: the agent wants that its interlocutor starts the rain (the good thing with computer science is that it does not need to be realistic... :-));
- `Commit(raining=true)`: the agent commits to the value of `raining` at some point in the future;
- *etc.*

In multiagent systems, we will define a series of *performatives* that all must have a very precise effect on the receiver's beliefs base. The definition of these performatives is part of the software engineering of the MAS.

17.1.4 The FIPA Agent Communication Language (ACL)

The Fundation for Intelligent Physical Agents (FIPA) is an IEEE organisation that defined some standards for MAS design. In this course, we shall consider three of their contributions:

- The FIPA-ACL model, which defines the structure of messages;
- The FIPA performatives, which defines 22 performatives with their semantics;
- The AUML protocol description model, which specifies how message exchanges should be presented from a software engineering point of view.

The FIPA-ACL model defines the possible fields of a message in direct communication models. Four fields are of outmost importance:

- **ID of the message sender**: each message has a unique sender agent ID. This can be used by receivers to answer to this message.

  Note that you do not send a pointer to the agent, only its idea. The idea is that an agent never accesses its interlocutor's variable or methods directly. As we will see in a few moments, direct communication mechanisms relies only on agents ids.

- **List of IDs of the message recipients**.

  Some platform support message broadcasting (using special ids with wildcard characters) but this is not the common rule. In most platforms, the system is provided with a *Directory Facilitator* that serves as a yellow-pages server. Each agent registers to the DF with the name of the services it can offer (as we will see in the next subsection, these services correspond to *roles* in the MAS *protocols*) with its ID, so that other agents can obtain the list of agents that can offer a given service by requesting the DF.

- **Message performative** selected in the list of performatives that this MAS supports.
- **Message content** using a syntax that is common to all agents in the MAS (for example, RDF triples, logical formulaes, `(name,value)` pairs, *etc*).

ACL messages have other optional fields such as:

- A reference to the language used for the content (this allow loosely-coupled MAS to communicate with heterogeneous syntactic languages, as long as there is some conversion rule from one language to another);
- A reference to the ontology used for the content (this allows semantic heterogeneity between agents);
- The conversation id, to allow agents to participate in several protocols at a time without loosing track of the exchanges;

17.2 2. Direct interactions implementation

That was quite a long theoretical part for today. Let us now discuss how all this can be implemented in a MAS platform.

Here is what we **cannot** do:

```
class Environment:
    ... environment definition ...

class Agent():
    ... agent definition ...
    ... procedural loop ...
```

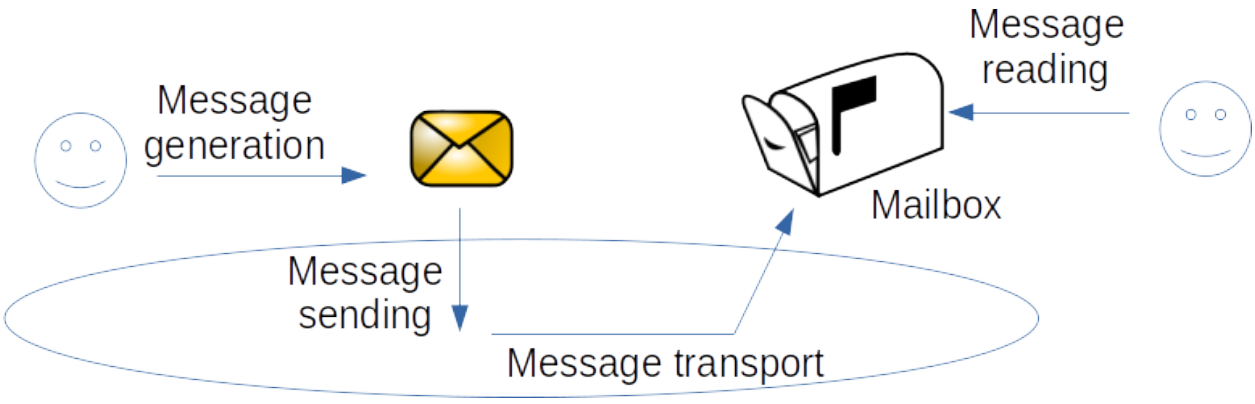
(continues on next page)

(continued from previous page)

```
    def receive(message):
        ... what to do with the message ...
```

By doing so, we would have agents that work in a synchronous manner: the sender's execution would have to end the termination of the `receive` method on the recipient's side before it can continue its own execution.

To overcome this limitation, computer scientist use a four step mechanism illustrated on the figure below:



1. The sender agent builds the message;
2. The sender agents invokes a `send` method in the environment to send the message, as one of its actions;

   This method returns immediately. The agent can now continue its procedural loop. It is assumed, from his point of view, that the recipient agent has received the message.
3. The environment drops the message in the mailbox of the agent. This can be done either:
   - In a synchronous manner as part of the `send` method: when method returns, the message is actually in the recipient's mailbox; or
   - In an asynchronous manner: the `send` method simply stores the message in the environment. The code that drops the message in the mailboxes will be invoked later (either as part of an environment's thread or, if the MAS is synchronous, at each time step).

   In both cases, the `send` method does not block the sender.
4. The receiver agent reads its mailbox, either in a systematic manner as part of the perception mechanism in the procedural loop (*passive perception*) or on purpose, by calling a specific method (*active perception*).

   Active perception is much more common than passive one, because it is much easier to use for the MAS programmer.

You shall not be surprised by this mechanism, since it is very close to a blackboard model... except that agents drop messages on the blackboard and only receive their messages (it is generally assumed that agents cannot overhear the other agent's message, although this is not of importance in most MAS models).

17.2.1 Concrete implementation

Let us implement this mechanism in a Python MAS. By doing so, we begin to write a MAS platform. The reason why most people use existing libraries, APIs and platforms is to avoid to implement this again and again.

At this point of the course, we need to consider two aspects:

1. In a MAS with direct interactions, there is no need to distinguish between the environment and the runtime (as we did for indirect interactions). Indeed, there is no such thing as "shared" variables that would be accessible to all agents. On the contrary, each agent only has access to its own internal variables... and messages for other agents. The perception and action phases in the procedural loop are reduced to message sending.

(*This is not entirely true since you will certainly use* `print` *statements for debugging issues, and a* `print` *statement in the procedural loop is a form of action in the environment*)

2. We are no longer implementing one single MAS example, but we are considering generic mechanisms for agent communications. To this goal, we need to separate the agent platform from the agent model. Concretely, we wand to implement generic `Agent` and `Environement` classes. As was done in session 1 (section 4) with the Mesa library, the classes in your MAS will inherit these generic classes.

17.2.2 Question 1: some degree of genericity

Using either synchronous or asynchronous mechanisms, write generic `Agent` and `Environment` classes that could be used to implement a simple Alice-Bob example. The `Environment` class should include the scheduler and `start`/`stop` methods, but it should not propose any perception and action mechanism for now.

Here is the code's skeleton:

```
import time

class Environment:

    # ... to be continued ...

class Agent:

    # ... to be continued ...

class AliceBob(Agent):
    def __init__(self, env, pv):
        super().__init__(env)
        self.preferred_value = pv
    def procedural_loop(self):
        # nothing to do for now

e = Environment()
a = AliceBob(e,1)
b = AliceBob(e,2)
e.start()
```

17.2.3 Question 2: Messages

Let us create a generic `Message` class:

```
class Message:
    def __init__(self,performative,sender_id,content=None):
        self.perf = performative
        self.sender = sender_id
        self.content = content
        self.dest = []
    def set_content(self,content):
        self.content = content
    def add_receiver(self,agent_id):
        self.dest.append(agent_id)
    def get_content(self):
        return self.content
    def get_performative(self):
        return self.perf
    def get_sender(self):
        return self.sender
    def get_receivers(self):
        return self.dest
```

Note that the content is optional in the constructor (some messages have an empty content).

1. Based on this new class, implement a `send` method in the `Environement` class (which will now be responsible for the agent execution **and** for the transport of messages).
2. Also write a `send` method in the generic `Agent` class to call the environment's one (that is syntactic sugar).
3. Implement a `receive` method in the generic `Agent` class that allows to retrieve the message in the mailbox.

The correction is given below, so that you can continue the session. But you really should try to do it yourself!

```
class Environment:
    agents = {}
    id = 0
    l = Lock()
    def add_agent(self,agent):
        with self.l:
            s = "agent"+str(self.id)
            self.id+=1
        self.agents[s]=agent
        return s
    def start(self):
        for id in self.agents:
            self.agents[id].start()
    def stop(self):
        for id in self.agents:
            self.agents[id].stop()
    def send(self,message):
        print(str(message.get_sender())+" -> "+str(message.get_receivers())+" : "+ \
              str(message.get_performative())+"("+str(message.get_content())+")")
        for id in message.get_receivers():
            self.agents[id].append_message(message)

class Agent(Thread):
    def __init__(self,env):
        Thread.__init__(self)
```

(continues on next page)

(continued from previous page)

```
        self.env = env
        self.name = env.add_agent(self)
        self.running = True
        self.mailbox = []
    def get_name(self):
        return self.name
    def run(self):
        while self.running:
            self.procedural_loop()
    def procedural_loop(self): # will be overridden
        pass
    def stop(self):
        self.running = False
    def append_message(self,message):
        self.mailbox.append(message)
    def send(self,message):
        self.env.send(message)
    def receive(self):
        l = self.mailbox.copy()
        self.mailbox.clear()
        return l
```

17.2.4 Question 3: Concrete example

Based on this, we can create a third agent name `Charles` whose role is hold a variable `v` and to process messages from Alice and Bob:

- On their turn, Alice and Bob ask Charles for the value of `v`, using a message;
- If the value is different from their preferred value, they send a message to Charles to change the value of `v`;
- On its turn, Charles reads its mailbox and processes all messages:
  - Messages that request information about `v` produce an anwer;
  - Messages that request a change to `v` are applied.

Before jumping into the implementation, answer the following questions: 1. What will be the performative for asking Charles the value of `v`? What will be the content (note that empty contents are always allowed)? 2. What will be the performative for telling Alice or Bob the value of `v`? What will be the content? 3. What will be the performative for asking Charles to change the value of `v`? What will be the content? 4. Implement the Alice-Bob-Charles example.

CHAPTER

EIGHTEEN

3.3. ENGINEERING INTERACTIONS

Before we move to the next section and the implementation of direct communication in Mesa, you need some information about direction interaction engineering.

As explained above, MAS engineering requires to define two things when it comes to direction communication:

- The list of *performatives* with their precise semantics;
- The *communication protocols* that will be used by your agents.

18.1 1. Performatives

The first thing to consider when you design a MAS with direct interactions is the list of *performatives* and associated *message content types* that will be used in your communication model. In the Alice, Bob and Charles example, we have three types of messages: questions about the value of Charle's internal variable, answers about the value of this variable, orders to change the value. There are at least four different solutions for engineering this interaction. What matters is the **semantics** we give to each performative.

The semantics of the performatives define the belief base for both the sender and the receiver of a message. Let us consider the Alice, Bob and Charles example and the case *Alice sends a message to Charles to ask the value of variable ``v``*. We can achieve this with a performative `question-variable` with the following semantics:

- The content of the message must be a `String` that is the name of a variable;
- From the sender agent's point of view, before sending the message (these are the *preconditions* of the performative):
  - The agent is unsure about the value of the variable;
  - The agent beliefs that the recipient has the value of the variable in its belief base;
  - The agent intends to tell the recipient that it wants to be told about this value.
- And from the receiver agent's point of view, upon reception of the message (these are the *postconditions* of the performative):
  - The agent beliefs that the sender is unsure about the value of the variable;
  - The agent beliefs that the sender agent expects it to send the value of the variable.

This make a lot of assumptions and they have strong consequences on the code. If you decide that `question-value` means that the sender agent is unsure about the value of the variable, you need some mechanism to change the beliefs of agents Alice and Bob so that they ask for the value again after they sent a `set-value` message. Otherwise, there is no reason for them sending a `question-value` message once they ordered a change. You can use a **timeout**, for instance, to remove the value of `v` from the Alice and Bob's belief bases after some time.

It is very unlikely that you implemented such a complex mechanism in your model. But that's ok: it all depends on the semantics of your performatives. For instance, we could have defined a much more simple interaction model:

- We still use 3 performatives: `question-v`, `assert-v` and `set-value`
- `question-v` requires no content. It makes no assumption on the sender's belief base (no precondition), but its postcondition is that the receiver intends to send the value of `v`;
- `assert-v` has the value as content. It assumes that the sender agent has a belief about the value of `v` as a precondition (and, of course, that the value is the content of the message), and that the receiver agent believes the new value of `v` (as a postcondition).
- `set-v` has the value as content. It assumes that the sender agent believes the value of `v` not to be the one sent as content (precondition) and it has two postconditions: the receiver agent changes the value of `v` and the sender agent believes that the value is now what was requested.

In this model, although there are some assumptions about a coherence between the sender and receiver in the case of `set-v` messages, there is no precondition for `question-v` messages: the agent can send it whenever it wants to. The advantage is that it is much simpler to implement. The drawbacks are twofold:

- The model lacks genericity (it can only be used in a MAS with agents that share information about a variable `v` they all know of: this is a very specific case of tightly coupled MAS);
- The agents will certainly spam each other with `question-v` messages since there is no precondition to this message.

As a conclusion, engineering MAS interactions is not an easy task. You need to find the correct balance between very generic models that will require a lot of code to work and too specific ones that miss the basic notions of MAS. Some solid theoretical tools exist in the MAS literature to help programmers in this task. One of them is the BDI logic by Cohen and Levesque that we already referred to in the first session (section 1) of this course. By representing agents' beliefs and intentions, it allows to describe precisely the effects of the performatives. This is the solution that was used by the FIPA to define the semantics of their 22 standard performatives. However, we will not detail these in this course (more information can be found on this page).

18.2 2. Communication protocols

In the *performative specification* task, we only define the message content, preconditions and postconditions in terms of agents beliefs: what is the structure of the message and how it relates to the belief base of the two agents. However, we did not define the **structure of the interaction**, *i.e.* how the message relate with each other. This must be define as **communication protocols**.

A *protocol* is a possible series of message exchanges (or interactions) between two or more agents to achieve a subtask of the MAS. Engineers try to have these protocols as short as possible to as to keep control of what is actually done by the system. When engaging in a protocol, agents commit to provide the expected answers. They adopt a given **role** that defines what they can receive as message and what they can answer.

In our Alice-Bob-Charles example, we can consider two protocols:

- In the first protocol (let's call it "Tell me the value of V"-protocol), one agent (taking the role of *initiator*) asks another agent (taking the role of *informator*) about the value of a variable. The initiator must send a `question-value` message with the name of the variable and the informator must answer to this message with a `assert-value` message.

  Note that Charles is not the only agent that could serve as an informator. One could imagine that Alice and Bob adopt this role too. However, upon the MAS creation, we shall register Charles as the only agent that fulfils this role in the Directory Facilitator. Based on this, when other agents (Alice and Bob) ask the DF about which agent can achieve which role, they will only have Charles as a proposal. All their `question-value` will then be directed toward Charles.

Also note that the agents do not need to send message to each other to commit to the execution of a protocol. It is commonly admitted that **all protocols start with a different message** (ideally, a different performative, but a different *performative(content)* pair might do the trick) and that the receiver agent implicitly accepts to enter the protocol by answering the message accordingly (in our case, using an `assert-value` message).

- The second protocol (let's call it "Change your value"-protocol) is a very specific kind of protocol, that you rarely encounter in open, loosely coupled MAS. In this protocol, the *requester* agent simply sends a `set-value` message to the *participant* agent, and the protocols ends. There is not return message.

  Note that, contrary to the first protocol, Charles is the only agent that could serve as a participant agent in this protocol. Since it has registered on the DF for this task, Alice and Bob can send him `set-value` message.

It is important to distinguish between agents and roles in MAS. Several agents can play the same role (Alice, Bob and Charles can all play the role of informator) and several roles can be played by the same agent in different protocols (Charles can be an *informator* in the "Tell me the value of V"-protocol and a participant in the "Change your value"-protocol) or even in different instances of the same protocol (while there is no example in the ABC example, one can easily imagine a protocol with three roles for which one agent could hold two roles). **You must define protocols with roles**, not with agents.

Also keep in mind that protocols are small. There is no need to specify all the possible interactions of your MAS in one single protocol. On the contrary, keep each interaction separated so that you can control it. Protocol specifications only defines **what the agents can answer**, not the agent's internal reasoning or conditions that decide for this answer! You must have these three layers in mind:

- Performative specification: what is a correct message content, under which conditions in the beliefs base can this message be sent, with which effects;
- Protocol specification: what is expected as an answer to a given message, and what is not correct, based on the above specification;
- MAS specification: what the agent actually answers, when and why, based on the above two specifications.

Now that this is all clear in your mind, let's see how we can write down such protocols.

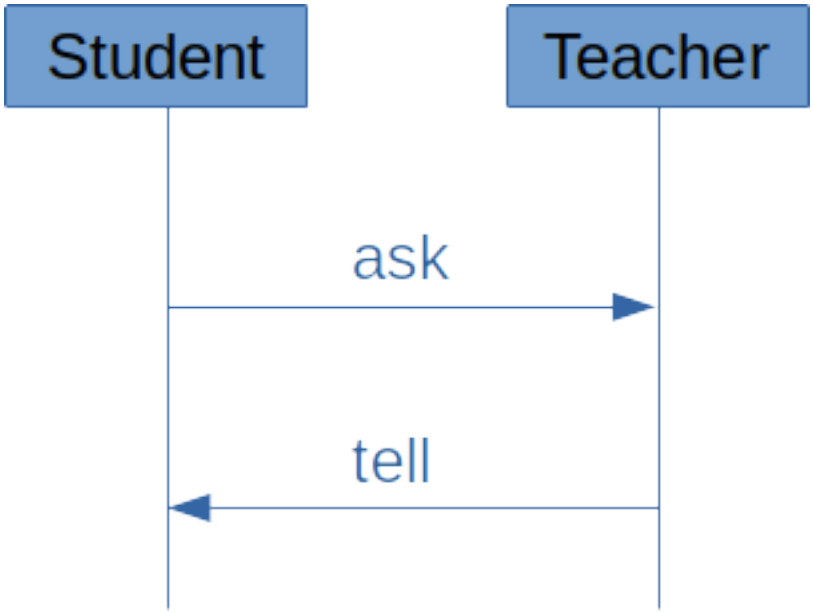
18.2.1 The AUML model

The language used by multiagent software engineers for describing their protocols is called AUML, for Agent-UML. Actually, it is not a standard (contrary to UML, for which the specifications are defined and maintained by the OMG) and most of its proposed features are now part of the UML standard, which means that an AUML is very similar to a UML sequence diagram, with some exceptions:

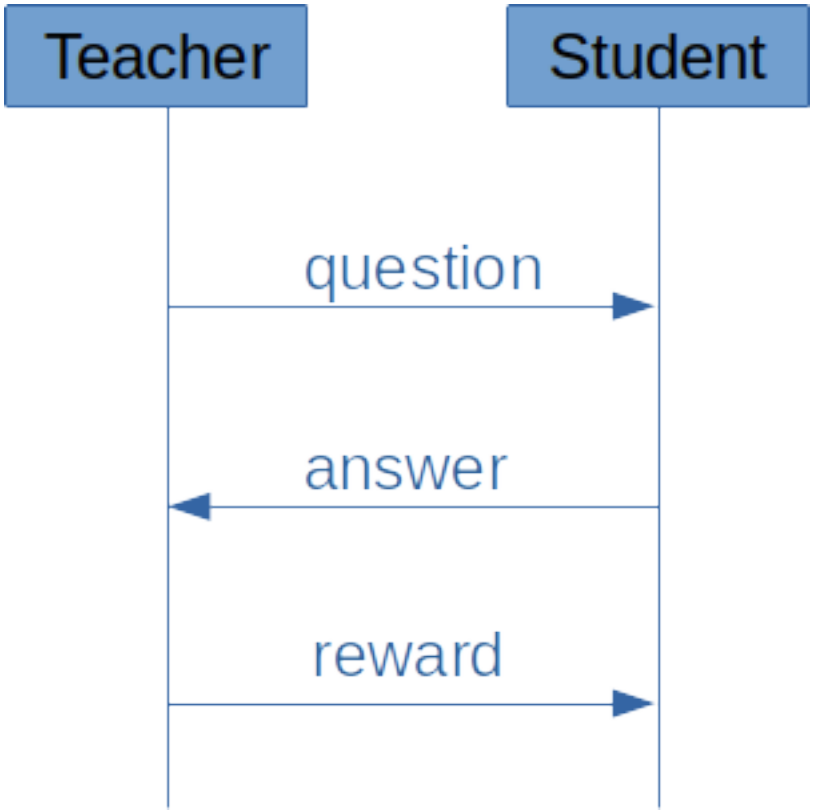
1. Instead of classes as labels for each thread, you must use **roles** (and *not* agents);
2. There is not "execution block" that represents time: agents are *asychronous*;
3. Message passing are labeled with **performatives** (and, if necessary, contents);
4. Conditions for message passing will only be about the number of recipients and the presence of timeouts;

   Indeed, the goal of an AUML diagram is not to define the *conditions* of each message selection, but only their agency : how many answers are expected, is there a timeout that conditions the execution, *etc*;
5. We shall use ALT, OPT and LOOP blocks to define the protocols.

Here is a simple example of protocol:

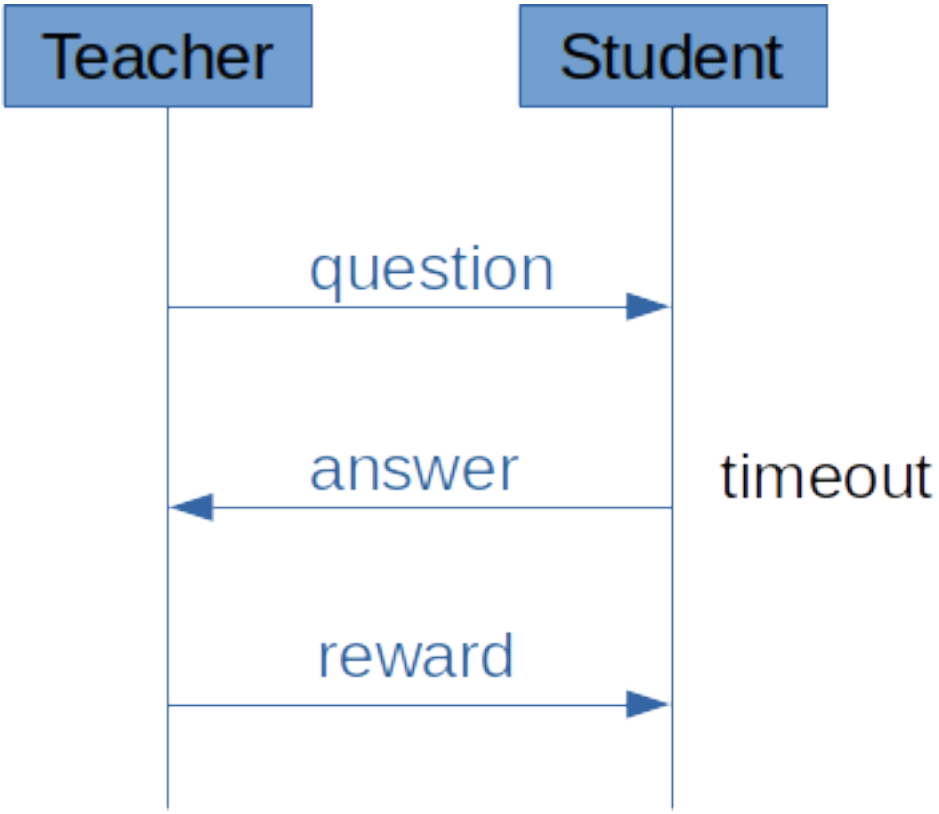


This protocol defines that teacher agents must answer with a “tell” performative when students send them an “ask” message. Now, let us consider a new version of the protocol:



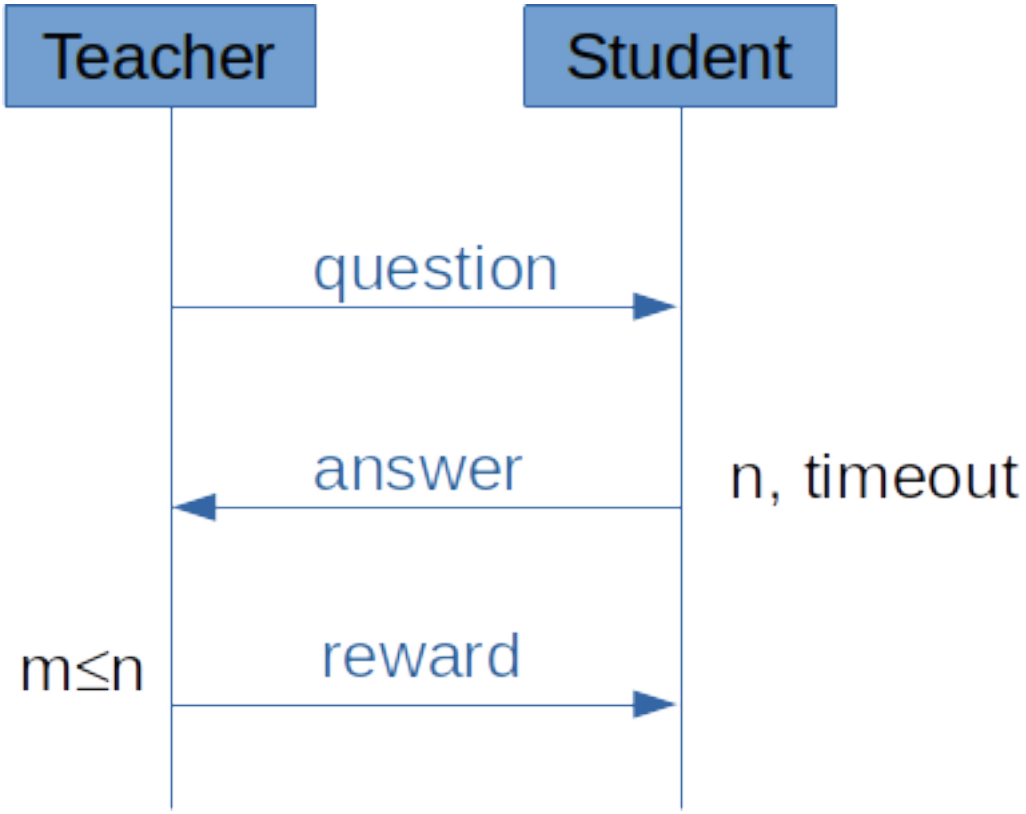
This protocol defines that students must answer with a “answer” performative to a “question” message sent by a teacher, and that they will receive a “reward” message in return. Now, we can add some constraints:

1. Only some students answer before a timeout:



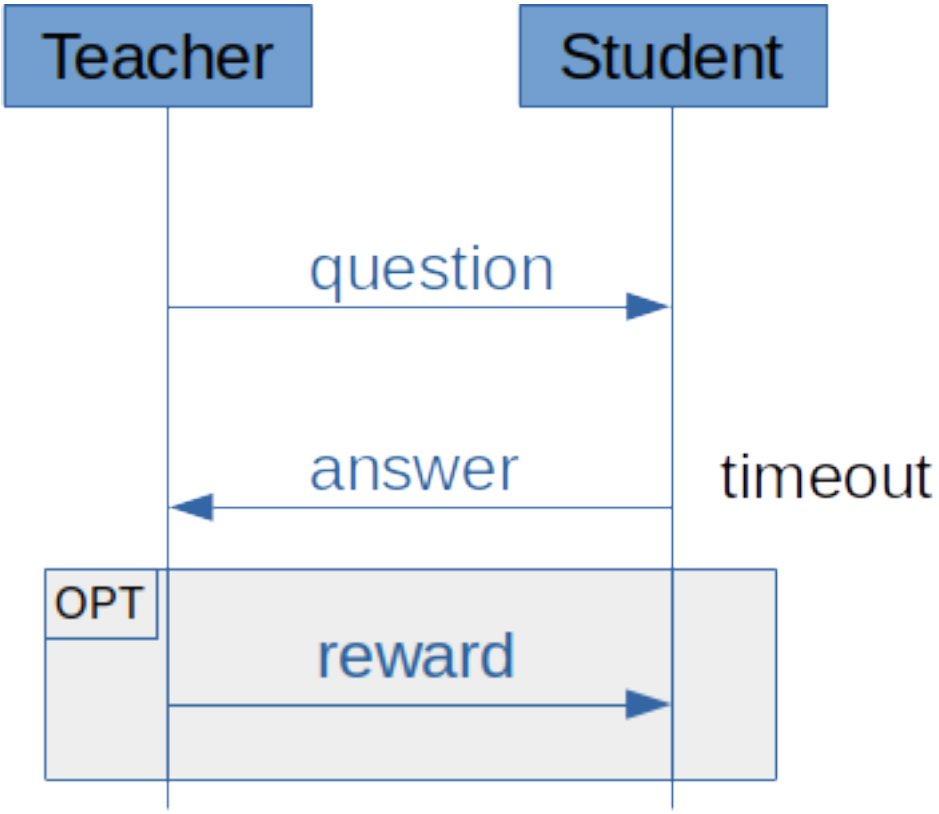
Note that only agents that did answer before the timeout remain in the protocol. All other agents are ignored. They won't receive the "reward" message.

2. Only some of the students (who answered before the timeout) will receive the reward:

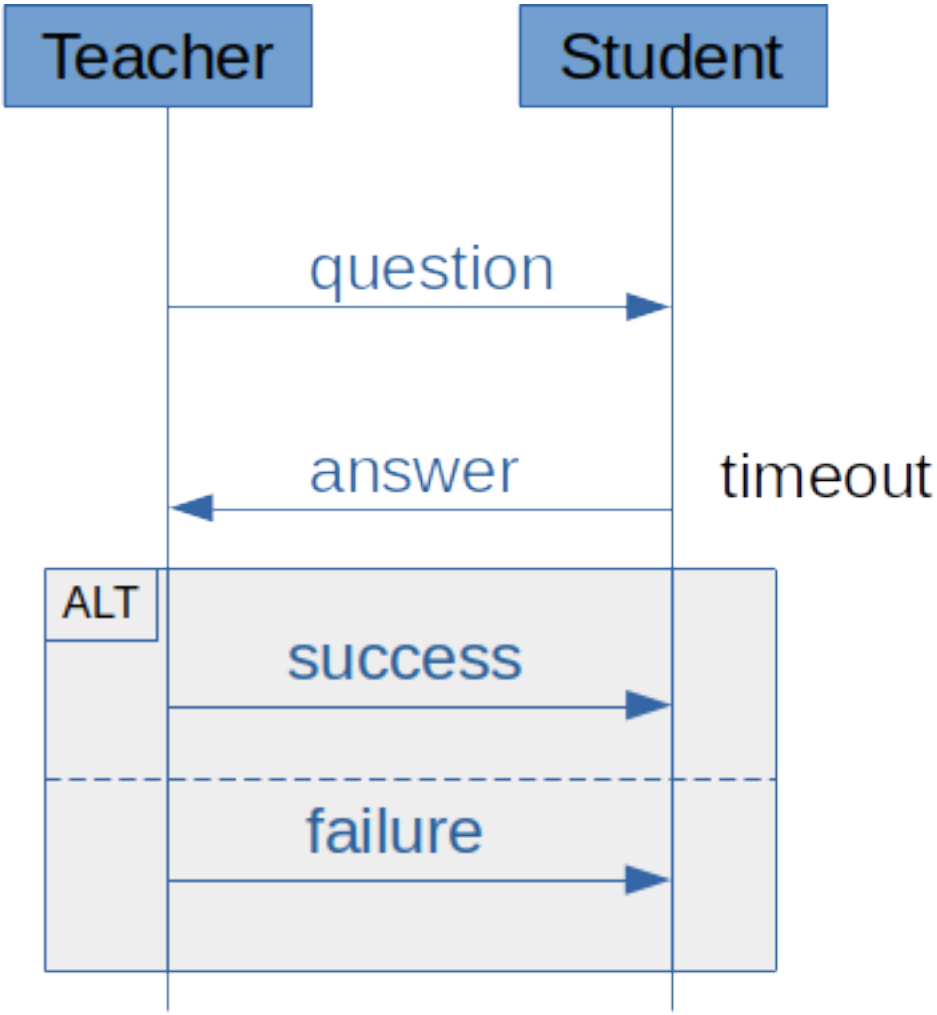


Note that *m* and *n* are not values: the conditions always remain at an abstract level. It is up to the agent to decide how many students it wants to reward, and for which reason.

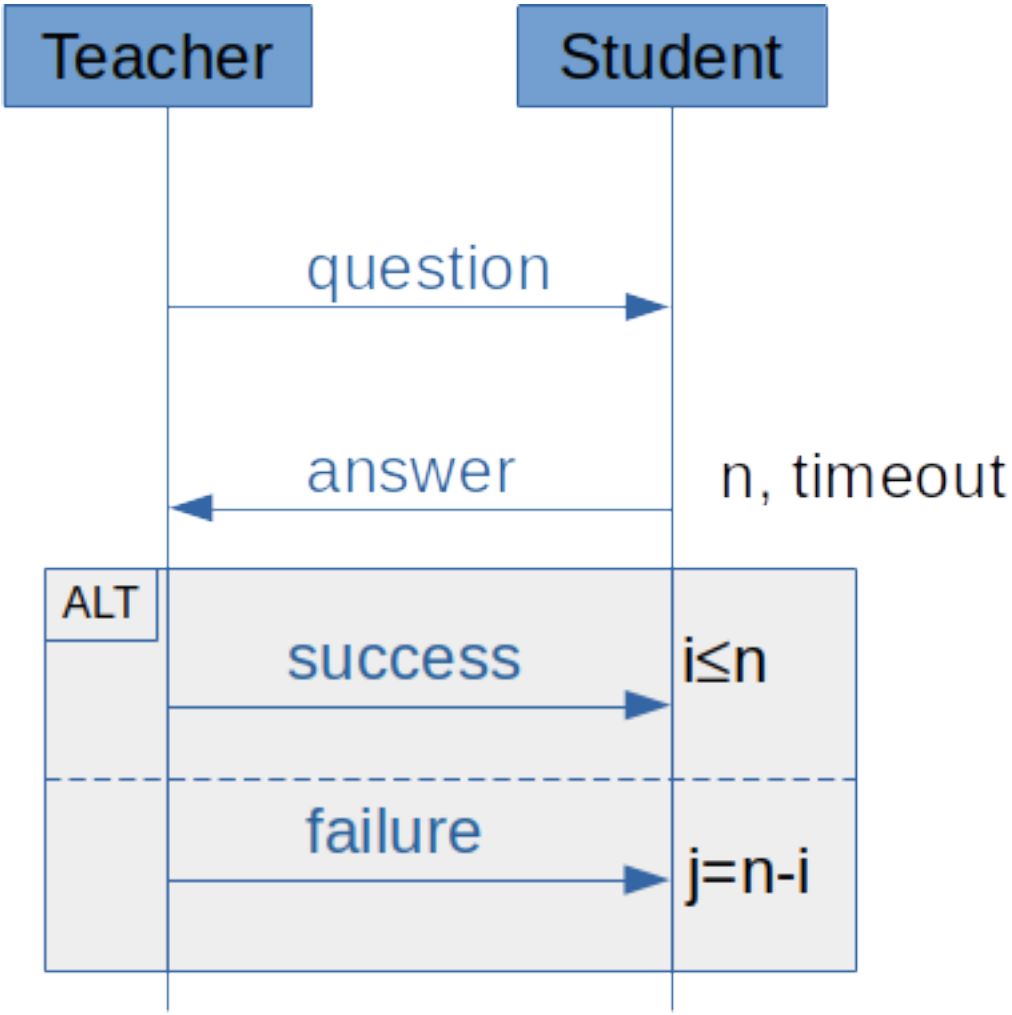
3. Another manner of writing the same protocol with the "optional" answer is to use an "OPT" block:



4. The most commonly used block is however the ALT block with shows different options with conditions. For instance:



5. And in that case, people often want to add conditions to the blocks:



Here, the condition expresses that *all* agents that answered receive either an "success" or a "failure" message. No agent remains unanswered.

6. It is also possible to use the classical "choice" UML symbol to represent ALT blocks. This makes the protocol easier to read when you have blocks within blocks. Here is the famous "Contract Net" protocol proposed by FIPA, with 3 ALT blocks:

As a MAS engineer, you must always define the protocols that your agents will participate to, in addition to the list of performatives.

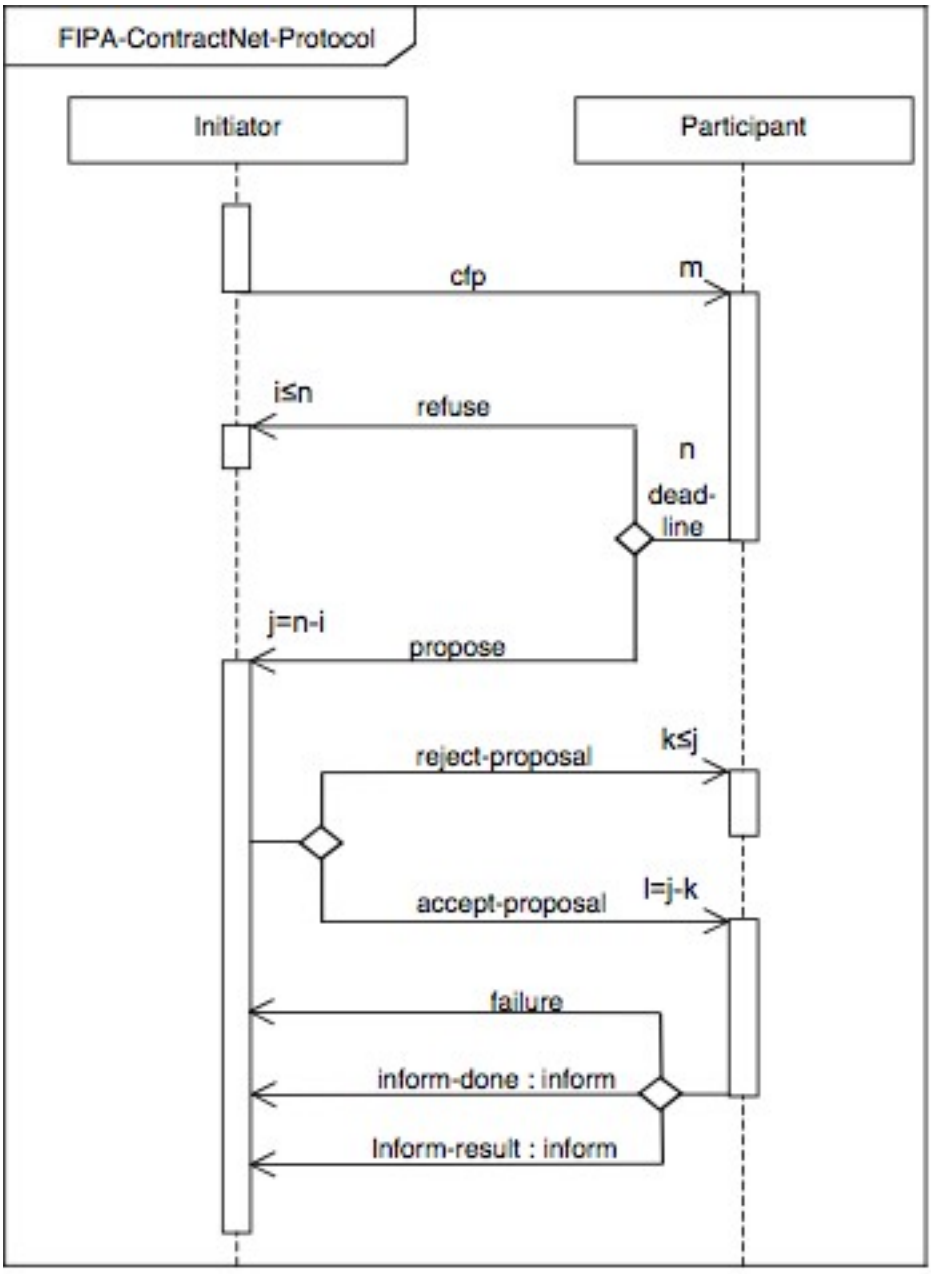


Fig. 1: The *Contract Net Protocol* <*http://www.fipa.org/specs/fipa00029/SC00029H.html*> defined by FIPA

18.2.2 Question

Write down the AUML protocols that we implemented in the Alice-Bob-Charles examples.

18.3 3. Communication in Mesa

In the next section, you will see how to implement a message passing system in the Mesa Python library.

CHAPTER

NINETEEN

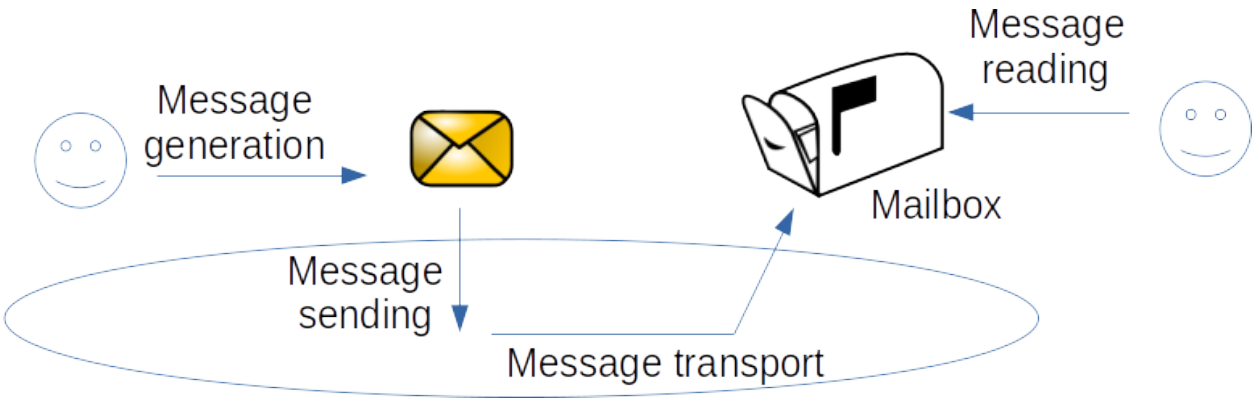
3.4. INTERACTIONS IN THE MESA LIBRARY

Mesa is a Python framework for agent-based modeling. You have already created a simple model and progressively added functionality which have illustrated Mesa's core features (first course session, section 4). Now, we will implement in **Mesa** a direct interactions mechanism: messaging communication.

19.1 1. Implementing messaging communication in Mesa

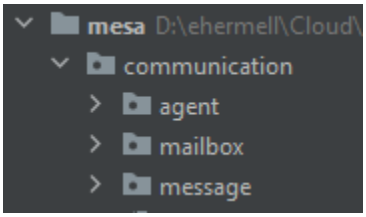
With the **Mesa** library, it is possible to set up indirect interaction mechanisms through the environment (like a *blackboard* or *stigmergy* mechanism, see today's session, section 1). However, it does not handle direct interactions such as message communication. So, first, we will implement our own communication layer in **Mesa**. This communication layer will be necessary for the realization of the final project.

As seen previously, the messaging communication in MAS is a four step mechanism.



Before implementing the communicating agents and according to the picture, we must create: (1) a `Message` object, (2) a `Mailbox` object and (3) a list of allowed performative for messages.

To do so, create a new folder hierarchy as shown in the following picture:



Here is the description of what each folder will contains:

- **mesa**: root folder which will contain your python codes using the communication layer;

- **communication**: the root folder of the communication layer;
- **agent**: the folder which will contain the implementation of the communicating agent class;
- **mailbox**: the folder which will contain the implementation of the mailbox class;
- **message**: the folder which will contain the implementation of the message and performative class.

Create an *__init__.py* file in the **communication**, **agent**, **mailbox** and **message** folders to initiate python packages.

As everything is setting up, we can now move on to the implementation of the communication layer.

19.1.1 Messages

Enter the **message** folder and create a new *Message.py* file. Let's open this file and implement the `Message` class. The purpose of this `Message` class is to create a python message object containing the receiver and sender identifiers but also the performative of the message sent as well as a content. Agents will exchange information using these items during their communication phases. The `Message` class is therefore composed of four attributes, four accessor methods (used to access the state of the object i.e, the data hidden in the object can be accessed from this method) and a string method (which returns a string, which is considered an informal or nicely printable representation of the message object).

The four attributes of the `Message` class are:

- *from_agent*: the sender of the message identified by its id;
- *to_agent*: the receiver of the message identified by its id;
- *message_performative*: the performative of the message;
- *content*: the content of the message.

The four accessor methods of the `Message` class are:

- *get_exp()*: return the sender of the message;
- *get_dest()*: return the receiver of the message;
- *get_performative()*: return the performative of the message;
- *get_content()*: return the content of the message.

Practice yourself!: you can try to implement this class by your own!

Note: a possible implementation of the `Message` class is *1. Message class*.

Now that we have a usable `Message` object, we are going to create the set of allowed message performatives from a python enumeration. We will define seven message performatives for the moment. It will be very easy to add more later. The seven performatives are as follows:

- propose;
- accept;
- commit;
- ask why;
- argue;
- query;
- inform.

In the **message** folder, create a new file called *MessagePerformative.py*. Open it and implement the `MessagePerformative` enumeration class.

Note: a possible implementation of the `MessagePerformative` class is *2. Message Performative Class*

19.1.2 Mailbox

To manage messages, each communicating agent will have his own mailbox. The purpose of this class is to provide to agents some mechanisms for handling sent and received messages. So, go to the **mailbox** folder and create a new *Mailbox.py* file. Let's open this file and implement the `Mailbox` class which is composed of two attributes and five methods.

The two attributes of the `Mailbox` class are:

- *unread_messages*: the list of unread messages;
- *read_messages*: the list of read messages.

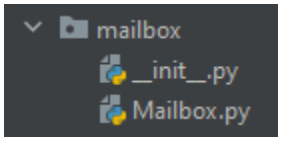
The five methods of the `Mailbox` class are:

- *receive_messages(message)*: receive a message and add it in the unread messages list;
- *get_new_messages()*: return all the messages from unread messages list;
- *get_messages()*: return all the messages from both unread and read messages list;
- *get_messages_from_performative(performative)*: return a list of messages which have the same performative;
- *get_messages_from_exp(exp)*: return a list of messages which have the same sender.

Practice yourself!: you can try to implement this class by your own!

Note: A possible implementation of the `Mailbox` class is *3. Mailbox Class*

The `Message` and `Mailbox` classes being created, we will test them. Go to the **communication** folder and create a new *runtests.py* file. In this file, we will incrementally add tests to verify that our implementations are working well.

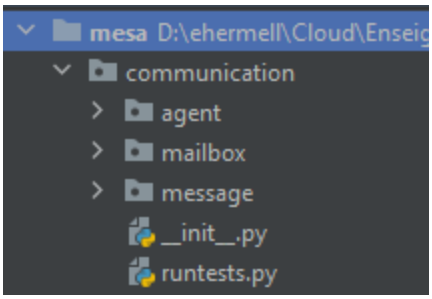


Practice yourself!

Let's start testing the `Mailbox` class: using the `assert()` function. Create three messages with various performatives, one mailbox and test the different methods of the `Mailbox` class (`receive_messages(message)`, `get_new_messages()`, `get_messages()`, `get_messages_from_exp()` and `get_messages_from_performative()`). Before looking at the solution, try to implement the tests by your own.

Note: A possible implementation of the *runtests.py* file is *4. Runtest*

If you see all the *OK* messages appear, it means that your classes are well implemented and will behave correctly. This is called doing unit tests. It is very important in making sure your code is robust and bugs free.



19.1.3 Message Service

At this point, each agent will have their own mailbox instance and will be able to exchange messages. However, there are still no mechanisms to ensure that sent messages reach the right agents. As agents must not directly drop messages in the mailboxes of the other agents, we need to create a service (a message transport mechanism) which will be managed by the environment and which will take care of the management of message shipments and deliveries.

Go to the **message** folder and create a new *MessageService.py* file. Let's open this file and paste the `MessageService` class implementation that you can find below:

```
#!/usr/bin/env python3

class MessageService:
        """MessageService class.
        Class implementing the message service used to dispatch messages between␣
↪communicating agents.

        Not intended to be created more than once: it's a singleton.

        attr:
                scheduler: the scheduler of the SMA (Scheduler)
                instant_delivery: the instant delivery status of the MessageService
                messages_to_proceed: the list of message to proceed mailbox of the␣
↪agent (list)
        """

        __instance = None

        @staticmethod
        def get_instance():
                """ Static access method.
                """
                return MessageService.__instance

        def __init__(self, scheduler, instant_delivery=True):
                """ Create a new MessageService object.
                """
                if MessageService.__instance is not None:
                        raise Exception("This class is a singleton!")
                else:
                        MessageService.__instance = self
                        self.__scheduler = scheduler
                        self.__instant_delivery = instant_delivery
                        self.__messages_to_proceed = []

        def set_instant_delivery(self, instant_delivery):
```

(continues on next page)

(continued from previous page)

```
                """ Set the instant delivery parameter.
                """
                self.__instant_delivery = instant_delivery

        def send_message(self, message):
                """ Dispatch message if instant delivery active, otherwise add the␣
↪message to proceed list.
                """
                if self.__instant_delivery:
                        self.dispatch_message(message)
                else:
                        self.__messages_to_proceed.append(message)

        def dispatch_message(self, message):
                """ Dispatch the message to the right agent.
                """
                self.find_agent_from_name(message.get_dest()).receive_message(message)

        def dispatch_messages(self):
                """ Proceed each message received by the message service.
                """
                if len(self.__messages_to_proceed) > 0:
                        for message in self.__messages_to_proceed:
                                self.dispatch_message(message)

                self.__messages_to_proceed.clear()

        def find_agent_from_name(self, agent_name):
                """ Return the agent according to the agent name given.
                """
                for agent in self.__scheduler.agents:
                        if agent.get_name() == agent_name:
                                return agent
```

As you can see, this class implement a `MessageService` object which is a *singleton*: it can only be instantiated once. Thus, there can only be one postal service in the MAS. This service has three attributes, one mutator method and four methoods.

The three attributes of the `MessageService` class are:

- *scheduler*: the scheduler of the SMA initialized in the **Mesa** model;
- *instant_delivery*: the instant delivery status of the MessageService. If `True`, the message will be delivered instantly in the mailbox of the agent;
- *messages_to_proceed*: the list of message to proceed.

The mutator method of the `MessageService` class is:

- *set_instant_delivery(instant_delivery)*: change the instant delivery status of the MessageService.

The four methods of the `MessageService` class are:

- *send_message(message)*: dispatch a given message if instant delivery is actived, otherwise add the message in the MessageService message list to be proceeded after;
- *dispatch_message(message)*: dispatch the given message to the right agent;
- *dispatch_messages()*: proceed and dispatch each message received by the message service;
- *find_agent_from_name(agent_name)*: return the agent according to the agent name given.

How to use ``MessageService`` class

As described previously, the `MessageService` will be in charge of distributing the sent messages to the right agents. To use `MessageService`, it is necessary to instantiate it in the constructor of the **Mesa** `Model` and give it as parameter the reference of the `Scheduler`:

```
def __init__(self):
self.schedule = RandomActivation(self)
self.__messages_service = MessageService(self.schedule)
        ...
```

Then, just call the method `dispatch_messages()` of the `MessageService` in the `step()` method of the **Mesa** `Model`.

```
def step(self):
            ...
            self.__messages_service.dispatch_messages()
            ...
```

The `MessageService` can dispatch the messages instantly or at each time step. You can change this behavior by calling the `set_instant_delivery(isntantly)` method and give as parameter a `Boolean` which represents by its value the activation or not of the instantaneous mode.

```
MessageService.get_instance().set_instant_delivery(False)
```

19.1.4 Communicating Agent

We now have all the tools implemented: (1) `Message`, (2) `MessagePerformative`, (3) `Mailbox` and (4) `MessageService`. We are going to create a communicating agent that inherits from **Mesa** `Agent` class. This `CommunicatingAgent` class is not intended to be used on its own, but must be inherit to create other agent classes.

If we refer to the four step mechanism presented on the previous section of the course, a communicating agent must be able to:

1. A communicating agent (sender) builds a message;
2. A communicating agent (sender) invokes a `send` method in the environment (through the `MessageService`) to send the message, as one of its actions;
3. The environment (through the `MessageService`) drops the message in the mailbox of the communicating agent (receiver). This can be done either instantly or not.
4. A communicating agent (receiver) reads its mailbox, either in a systematic manner as part of the perception mechanism in the procedural loop (*passive perception*) or on purpose, by calling a specific method (*active perception*).

The `CommunicatingAgent` must therefore have three attributes, one accessor method and seven methods. Go to the **agent** folder and create a new *CommunicatingAgent.py* file. Let's open this file and implement the `CommunicatingAgent` class

The three attributes of the `CommunicatingAgent` class are:

- *name*: the name of the communicating agent which replaces the unique id of **Mesa** `Agent` class;
- *mailbox*: the agent's unique and private mailbox;
- *message_service*: the reference to the message service instantiated in the environment.

The acceessor method of the `CommunicatingAgent` class is:

- *get_name()*: return the unique name of the communicating agent.

The seven methods of the `CommunicatingAgent` class are:

- *step()*: the `step()` methods of the communicating agent called by the **Mesa** `Scheduler` at each time tick;
- *receive_message(message)*: receive a message (called by the `MessageService`) and store it in the mailbox;
- *send_message(message)*: send message through the `MessageService` (`messages_service.send_message(message)`);
- *get_new_messages()*: return all the unread messages;
- *get_messages()*: return all the received messages;
- *get_messages_from_performative(performative)*: return a list of messages which have the same performative;
- *get_messages_from_exp(exp)*: return a list of messages which have the same sender.

As you can see, many methods of the `CommunicatingAgent` class are the same as the methods of the `Mailbox` class: we have chosen to have mailbox accessible only through dedicated methods which makes it private.

Practice yourself!: you can try to implement the `CommunicatingAgent` class by your own!

Note: A possible implementation of the `CommunicatingAgent` class is *5. CommunicatingAgent Class*

Practice yourself!

All the features of the **Mesa** communication layer being implemented, we will complete the *runtests.py* file to test the `MessageService` and `CommunicatingAgent` classes. To do this, we will implement a **Mesa** `Model` and communicating agents:

1. Open the *runtests.py* file;
2. Implement a `TestAgent` class which inherits from `CommunicatingAgent` class;
3. Implement a `TestModel` class which inherits from **Mesa** `Model`. This model isntantiate the `MessageService` and two communicating agents;
4. In the `if __name__ == "__main__":`, instantiate the `TestModel` and make the two communicating agents communicate through messages. Use the `assert()` function to ensure that the behaviors (number of messages sent, received, etc.) are those expected.

Note: A possible implementation of these four steps is *6. Tests*

If you reuse the provided code in your `if __name__ == "__main__":` and see all the *OK* messages appear, it means that your classes are well implemented and will behave correctly. The implementation of the communication layer in **Mesa** is now complete.

19.2 2. Concrete using of messaging communication in Mesa

In the previous section of the course, you have implemented an interaction mechanism in a simple Alice-Bob MAS. The objective here is to reimplement this very simple example by using the communication layer that we have just integrated in **Mesa**. Reimplementing this example will allow you to better understand how to use the **Mesa** communication layer and thus save time in the creation of the final project which will consist in setting up a communication and argumentation protocol between several communicating agents.

19.2.1 Alice-Bob-Charles: a concrete example

Create two communicating agents named *Alice* and *Bob*. Create a third agent named *Charles* whose role is hold a variable `v` and process messages from *Alice* and *Bob*:

- On their turn, Alice and Bob ask Charles for the value of `v`, using a message;
- If the value is different from their preferred value, they send a message to Charles to change the value of `v`;
- On its turn, Charles reads its mailbox and processes all messages:
  - Messages that request information about `v` produce an anwer;
  - Messages that request a change to `v` are applied.

Implement the Alice-Bob-Charles example.

19.2.2 Alice-Bob-Charles: a solution

To be integrated after !

CHAPTER

TWENTY

3.5. CORRECTIONS MESA COMMUNICATION

20.1 1. Message class

```
#!/usr/bin/env python3


class Message:
        """Message class.
        Class implementing the message object which is exchanged between agents␣
↪through a message service during communication.

        attr:
                from_agent: the sender of the message (id)
                to_agent: the receiver of the message (id)
                message_performative: the performative of the message
                content: the content of the message
         """
        def __init__(self, from_agent, to_agent, message_performative, content):
                """ Create a new message.
                """
                self.__from_agent = from_agent
                self.__to_agent = to_agent
                self.__message_performative = message_performative
                self.__content = content

        def __str__(self):
                """ Return Message as a String.
                """
                return "From " + str(self.__from_agent) + " to " + str(self.__to_
↪agent) \
                                + " (" + str(self.__message_performative) + ") " +␣
↪str(self.__content)

        def get_exp(self):
                """ Return the sender of the message.
                """
                return self.__from_agent

        def get_dest(self):
                """ Return the receiver of the message.
                """
                return self.__to_agent
```

(continues on next page)

(continued from previous page)

```
        def get_performative(self):
                """ Return the performative of the message.
                """
                return self.__message_performative

        def get_content(self):
                """ Return the content of the message.
                """
                return self.__content
```

Return to previous page: *Messages*.

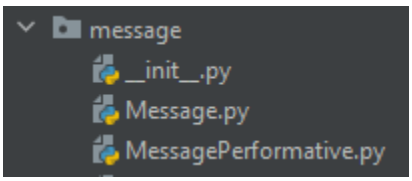
20.2 2. Message Performative Class

```
#!/usr/bin/env python3

from enum import Enum


class MessagePerformative(Enum):
        """MessagePerformative enum class.
        Enumeration containing the possible message performative.
        """
        PROPOSE = 101
        ACCEPT = 102
        COMMIT = 103
        ASK_WHY = 104
        ARGUE = 105
        QUERY_REF = 106
        INFORM_REF = 107

        def __str__(self):
                """Returns the name of the enum item.
                """
                return '{0}'.format(self.name)
```



Return to previous page: *Messages*.

20.3 3. Mailbox Class

```
#!/usr/bin/env python3


class Mailbox:
        """Mailbox class.
        Class implementing the mailbox object which manages messages in communicating␣
↪agents.

        attr:
                unread_messages: The list of unread messages
                read_messages: The list of read messages
         """

        def __init__(self):
                """ Create a new Mailbox.
                """
                self.__unread_messages = []
                self.__read_messages = []

        def receive_messages(self, message):
                """ Receive a message and add it in the unread messages list.
                """
                self.__unread_messages.append(message)

        def get_new_messages(self):
                """ Return all the messages from unread messages list.
                """
                unread_messages = self.__unread_messages.copy()
                if len(unread_messages) > 0:
                        for messages in unread_messages:
                                self.__read_messages.append(messages)

                self.__unread_messages.clear()
                return unread_messages

        def get_messages(self):
                """ Return all the messages from both unread and read messages list.
                """
                if len(self.__unread_messages) > 0:
                        self.get_new_messages()
                return self.__read_messages

        def get_messages_from_performative(self, performative):
                """ Return a list of messages which have the same performative.
                """
                messages_from_performative = []
                for message in self.__unread_messages + self.__read_messages:
                        if message.get_performative() == performative:
                                messages_from_performative.append(message)
                return messages_from_performative

        def get_messages_from_exp(self, exp):
                """ Return a list of messages which have the same sender.
                """
                messages_from_exp = []
```

(continues on next page)

(continued from previous page)

```
                for message in self.__unread_messages + self.__read_messages:
                        if message.get_exp() == exp:
                                messages_from_exp.append(message)
                return messages_from_exp
```

Return to previous page: *Mailbox*.

20.4 4. Runtest

```
#!/usr/bin/env python3
"""
Testing all the functionalities of the communication package.
"""

from communication.mailbox.Mailbox import Mailbox
from communication.message.Message import Message
from communication.message.MessagePerformative import MessagePerformative


if __name__ == "__main__":
        print("*---- Testing communication package ----")
        print("*")
        print("* 1) Testing Mailbox receive & get methods")

        mailbox = Mailbox()
        m1 = Message("Agent1", "Agent2", MessagePerformative.PROPOSE, "Bonjour")
        m2 = Message("Agent1", "Agent2", MessagePerformative.ACCEPT, "Hello")
        m3 = Message("Agent2", "Agent1", MessagePerformative.ARGUE, "Buenos Dias")

        mailbox.receive_messages(m1)
        mailbox.receive_messages(m2)

        assert(len(mailbox.get_new_messages()) == 2)
        print("*     get_new_messages() => OK")
        assert(len(mailbox.get_messages()) == 2)
        print("*     get_messages() => OK")

        mailbox.receive_messages(m3)
        assert(len(mailbox.get_messages()) == 3)
        assert(len(mailbox.get_messages_from_exp("Agent1")) == 2)
        print("*     get_messages_from_exp() => OK")
        assert(len(mailbox.get_messages_from_performative(MessagePerformative.
↪ACCEPT)) == 1)
        assert(len(mailbox.get_messages_from_performative(MessagePerformative.
↪PROPOSE)) == 1)
        assert(len(mailbox.get_messages_from_performative(MessagePerformative.ARGUE))␣
↪== 1)
        print("*     get_messages_from_performative() => OK")
```

Return to previous page: *Mailbox*.

20.5 5. CommunicatingAgent Class

```
#!/usr/bin/env python3

from mesa import Agent

from communication.mailbox.Mailbox import Mailbox
from communication.message.MessageService import MessageService


class CommunicatingAgent(Agent):
        """CommunicatingAgent class.
        Class implementing communicating agent in a generalized manner.

        Not intended to be used on its own, but to inherit its methods to multiple
        other agents.

        attr:
                name: The name of the agent (str)
                mailbox: The mailbox of the agent (Mailbox)
                message_service: The message service used to send and receive message␣
↪(MessageService)
        """

        def __init__(self, unique_id, model, name):
                """ Create a new communicating agent.
                """
                super().__init__(unique_id, model)
                self.__name = name
                self.__mailbox = Mailbox()
                self.__messages_service = MessageService.get_instance()

        def step(self):
                """ The step methods of the agent called by the scheduler at each␣
↪time tick.
                """
                super().step()

        def get_name(self):
                """ Return the name of the communicating agent."""
                return self.__name

        def receive_message(self, message):
                """ Receive a message (called by the MessageService object) and store␣
↪it in the mailbox.
                """
                self.__mailbox.receive_messages(message)

        def send_message(self, message):
                """ Send message through the MessageService object.
                """
                self.__messages_service.send_message(message)

        def get_new_messages(self):
                """ Return all the unread messages.
                """
                return self.__mailbox.get_new_messages()
```

(continues on next page)

(continued from previous page)

```
        def get_messages(self):
                """ Return all the received messages.
                """
                return self.__mailbox.get_messages()

        def get_messages_from_performative(self, performative):
                """ Return a list of messages which have the same performative.
                """
                return self.__mailbox.get_messages_from_performative(performative)

        def get_messages_from_exp(self, exp):
                """ Return a list of messages which have the same sender.
                """
                return self.__mailbox.get_messages_from_exp(exp)
```

Return to previous page: *Communicating Agent*.

20.6 6. Tests

1. Open the *runtests.py* file;
2. The implementation of the `TestAgent` class is very simple and only consists of calling the constructor of the inherited class and the `step()` method.

```
class TestAgent(CommunicatingAgent):
        """ TestAgent which inherit from CommunicatingAgent to test these␣
↪functionalities.
        """
        def __init__(self, unique_id, model, name):
                super().__init__(unique_id, model, name)

        def step(self):
                super().step()
```

3. The implementation of the `TestModel` class is very similar to what you did earlier with others **Mesa** `Model`. The only difference is the instantiation of the `MessageService` and its call in the `step()` method.

```
class TestModel(Model):
        """ TestModel which inherit from Model to test CommunicatingAgent and␣
↪MessageService.
        """
        def __init__(self):
                self.schedule = RandomActivation(self)
                self.__messages_service = MessageService(self.schedule)
                for i in range(2):
                        a = TestAgent(i, self, "Agent" + str(i))
                        self.schedule.add(a)
                self.running = True

        def step(self):
                self.__messages_service.dispatch_messages()
                self.schedule.step()
```

4. In the `if __name__ == "__main__":`, we test the sending of messages instantly at first and then at each

simulation step using `MessageService.get_instance().set_instant_delivery(False)`. To test the communication between each agent, we retrieve the two agents created by the `TestModel` via the scheduler (`agent0 = communicating_model.schedule.agents[0]`) and we make send them messages to each other (`agent0.send_message(Message("Agent0", "Agent1", MessagePerformative.COMMIT, "Bonjour"))`). We use the `assert()` function to verify that the exchanges are carried out as expected: `assert(len(agent1.get_new_messages()) == 1)` (the number of new messages of agent 1 is equal to 1). We try to cover all implemented functionality in the same way.

```
    print("* 2) Testing CommunicatingAgent & MessageService")

communicating_model = TestModel()

assert(len(communicating_model.schedule.agents) == 2)
print("*     get the number of CommunicatingAgent => OK")

agent0 = communicating_model.schedule.agents[0]
agent1 = communicating_model.schedule.agents[1]

assert(agent0.get_name() == "Agent0")
assert(agent1.get_name() == "Agent1")
print("*     get_name() => OK")

agent0.send_message(Message("Agent0", "Agent1", MessagePerformative.COMMIT, "Bonjour
↪"))
agent1.send_message(Message("Agent1", "Agent0", MessagePerformative.COMMIT, "Bonjour
↪"))
agent0.send_message(Message("Agent0", "Agent1", MessagePerformative.COMMIT, "Comment␣
↪ça va ?"))

assert(len(agent0.get_new_messages()) == 1)
assert(len(agent1.get_new_messages()) == 2)
assert(len(agent0.get_messages()) == 1)
assert(len(agent1.get_messages()) == 2)
print("*     send_message() & dispatch_message (instant delivery) => OK")

MessageService.get_instance().set_instant_delivery(False)

agent0.send_message(Message("Agent0", "Agent1", MessagePerformative.COMMIT, "Bonjour
↪"))
agent1.send_message(Message("Agent1", "Agent0", MessagePerformative.COMMIT, "Bonjour
↪"))
agent0.send_message(Message("Agent0", "Agent1", MessagePerformative.COMMIT, "Comment␣
↪ça va ?"))

assert(len(agent0.get_messages()) == 1)
assert(len(agent1.get_messages()) == 2)

communicating_model.step()

assert(len(agent0.get_new_messages()) == 1)
assert(len(agent1.get_new_messages()) == 2)
assert(len(agent0.get_messages()) == 2)
assert(len(agent1.get_messages()) == 4)
print("*     send_message() & dispatch_messages => OK")
```

Return to previous page: *Communicating Agent*.

CHAPTER

TWENTYONE

4. ARGUMENTATION-BASED NEGOTIATION

21.1 Agenda – Recall

1. Friday, March the 5th, 2021 : Introduction to MAS : definitions and implementation of a platform
2. Friday, March the 12th, 2021 : Multiagent simulation : preys and predators
3. Friday, March the 19th, 2021 : Interaction mechanisms : models and implementation
4. Friday, March the 26th, 2021 : Argumentation-based negotiation I: Practical session
5. Friday, April the 2nd, 2021 : Argumentation-based negotiation II: what is argumentation?
6. Friday, April the 9th, 2021 : Argumentation-based negotiation III: what is a negotiation protocol?
7. Friday, April the 16th, 2021 : Argumentation-based negotiation IV: Practical session

21.2 Session 4.

Argumentation-based negotiation has been proposed as an alternative to proposal-based approaches such as game theory and heuristic. The main advantage is that it allows agents to exchange additional information rather than just simple proposals. This property of argumentation protocols can lead to beneficial agreements when used for complex multi-agent negotiation.

This session is the starting point towards understanding what is an argumentation-based negotiation protocol. Moreover, in this session we will start the second practical work aiming to implement your second mulit-agent based model.

21.3 Some references

- Ferber, J. (1995), *Les Systèmes Multi-Agents*, InterEditions. **(French version)**
- Ferber, J. (1999), *Multi-agent systems: An introduction to distributed artificial intelligence*, Addison Wesley. **(English version)**
- Michael Wooldridge (2002), *An Introduction to MultiAgent Systems*, John Wiley & Sons Ltd.
- The AgentLink roadmap
- Rahwan, Iyad (2009), *Argumentation in Artificial Intelligence*,Springer.
- Lopes, Fernando & Coelho, Helder. (2014). *Negotiation and Argumentation in Multi-Agent Systems: Fundamentals, Theories, Systems and Applications*. Bentham Science Publishers.
- Argumentation in Multi-Agent Systems (ArgMAS) Workshop Series

CHAPTER

TWENTYTWO

4.1. ARGUMENTATION-BASED NEGOTIATION: INTRODUCTION

As it was discussed in the previsous sessions, Multi-Agent Systems (MAS) are systems composed of software agents that interact to solve problems that are beyond the individual capabilities of each agent. Software agents are elements situated in some environment and capable of flexible autonomous action in order to meet their design objectives. In almost cases, such agents need to interact to fulfill their objctives or improve their performance. Diffent types of interactions/dialogues exit (as it is illustrated in the following table), it depends on the application. In this course, we are interested by negotiation in multi-agent systems.

Type of dialogue	Initial Situation	Participant' Goal	Goal of Dialogue
Persuasion	Conflict of opinions	Persuade other party	Resolve or clarify issue
Inquiry	ignorance	find and verify or falsify evidence	proof or disproof
Negotiation	Conflict of interest	get what you most want	reasonable settlement that both can live with
Information seeking	unequal spread of information	Acquire or give information	spreading information
Deliberation	dilemma or practical choice	influence and contribute to outcome	Decide best course of action
Eristic	Personal conflict	Verbally hit out at opponent	Reveal deeper basis of conflict and reach some accommodation

Fig. 1: Types of dialogue (Norman et al., 2003)

Negotiation is a common, everyday activity that most people use to resolve opposing interests. Labor and management negotiate the terms of contracts. Businesses negotiate to purchase raw materials and to sell final products. Lawyers negotiate to settle legal claims before they go to court. Friends negotiate to decide which television programs to watch, etc. The negotiation process is a complex dynamic process. The core of such a process is reciprocal offer and counter-offer, argument and counter-argument in an attempt to agree upon outcomes mutually perceived as beneficial.

In MAS and according to [Rahwan et al., 2004]

> "Negotiation is a form of interaction in which a group of agents, with conflicting interests and a desire to cooperate, try to come a mutually acceptable agreement on the division of scarce resources".

The use of the word "ressources" here is to be taken in the broadest possible sens. In short, anything that is needed to achieve something.

Before discussing different frameworks for studying automated negotiation, we discuss in what follows some fundamental concepts in the automated negotiation literature [Rahwan, 2004].

1- **The negotiation agreement space**. Negotiation aims at reaching some allocation of resources that is acceptable to all parties. Since there are usually many different possible allocations (or possible agreements, or deals, or outcomes), negotiation can be seen as a "distributed search through a space of potential agreements" (Jennings et al., 2001). Hence, we first require some way of characterising such a space. Abstractly, the space can be seen as a set of deals $\psi = \{\Omega_1, \ldots, \Omega_n\}$ where n is the size of the search space.

Another way of characterising the search space is in terms of vectors in a multidimensional Euclidean space. In this characterisation, a deal is defined in terms of a set of attributes $A_1, \ldots, A_n$ (dimensions in an n dimensional vector space), where each attribute A_i can take a set of values $a_1^i, \ldots, a_m^i$. In this course we will adopt this representation

2- **Evaluating deals**. To evaluate a deal (outcome) at the agent level, we need to capture agent preference over ψ. As you have seen in the course 'Systèmes de Décision', preferences of agent i can be captured using a *preference relation* $\succsim_i$ over ψ, and we denote by $\Omega_1 \succsim_i \Omega_2$ that for agent i, Ω_1 is at least as good as Ω_2. Moreover, the preference relation of agent i is often described in termes of a *utility function* that captures the level of satisfaction of an agent with a particular outcome (for a recall see course 'Systèmes de Décision'). Thus, a *rational* agent seeks to reach a deal that maximises the utility it receives. Another evaluation concerns the *global evaluation* of the deals (outcomes). At this level we may use notions such as domiance and pareto optimality (again please refer to the decision course).

3- **Negotiation mechanisms**. Given a set of agents, a set of resources (decision, actions,...), the main goal of negotiation is to find an allocation that is better in some sense, if such allocation exists. In order to achieve this goal, agents need some mechanism. Abstractly, a mechanism specifies a set of rules, such as: what agents are allowed to say and when, how allocations are calculated, whether calculations are done using a centralised algorithm or in a distributed fashion, and so on. Different mechanisms may have different properties.

For designing an "efficient" negotiation mechanisms, following is a list of desirable features adapted from (Rosenschein and Zlotkin, 1994):

- *Simplicity*: A mechanism that requires less computational processing and communication overhead is preferable to one that does not.
- *Efficiency*: A mechanism is efficient if it produces a good outcome. What is meant by 'good,' however, may differ from one domain to another.
- *Distribution*: It is preferable to have a negotiation mechanism that does not involve a central decision-maker.
- *Symmetry*: The mechanism should not be biased for or against some agent based on inappropriate criteria. Again, what constitutes an 'inappropriate' criterion depends on the domain.
- *Stability*: A mechanism is stable if no agent has incentive to deviate from some agreed-upon strategy or set of strategies (e.g. to bid truthfully in an auction).
- *Flexibility*: By this property, its means that the mechanism should lead to agreement even if agents did not have complete and correct private information in relation to their own decisions and preferences. This property requires a mechanism for enabling agents to refine their decisions, in light of new information, during the negotiation process.

CHAPTER

TWENTYTHREE

4.2. AUTOMATED NEGOTIATION MECHANISMS

A variety of automated negotiation mechanisms have been studied in the literature. Different mechanisms have different advantages and disadvantages. These approaches are divedeed maintly into three groups. The more traditional ones, such as:

- **Game Theoretic analysis** (Rosenschein & Zlotkin, 1994; Sandholm, 2002) , where the aim is to determine the optimal strategy by analyzing the interaction as a game between identical participant and seeking its equilibrium; and
- **Heuristic-based approaches** (Faratin, 2000; Kowalczyk & Bui, 2001 ; Fatima et al., 2002), where the aim is to use heuristics rules that produce good enough, rather than optimal, outcomes/decisions.

However, these approches suffers from some limitations. For instance, the game-abased approaches assume that agent have unbounded computational resources and that the space of outcomes is completely known. For the second approaches, outcomes are sub-optimal (do not examine the full space of possible outcomes), and it is difficult to predict precisely how the system and the constituent agents will behave, thus the models need extensive evaluation through simulation and empirical analysis. Moreover, both approaches assume that agent's utilities or preferences are usually assumed to be completely characterized prior to interaction (what about situation with incomplete information), and fixed (no possibility to influence another agent's preference model or internal mental attitudes: beliefs, desires goals, etc.)

To overcome these limitations, new kind of approaches appear, called **Argumentation-based approaches** (Kraus et al., 1998; Parsons et al., 1998; Sierra et al., 1998). They allow agents to exchange different kinds of information, to "argue" about their beliefs and other mental attitudes during the negotiation process. Indeed, the two first types of settings do not allow for the addition of information or for exchanging opinions about offers. Thus, an argument can be viewed as a piece of information to i) *justify* its negotiation stance, or ii) *influence* another agent's negotiation stance. In addition to accepting or rejecting a porposal, an agent can offer a critique of it. This can help negotiation more efficient. The justification of a proposal, stating why an agent made such a proposal or why the counterpart should accept it, is an important issue which is currently at heart of the design of AI systems.

In this course, we are interested by the arguementation-based approaches.

CHAPTER

TWENTYFOUR

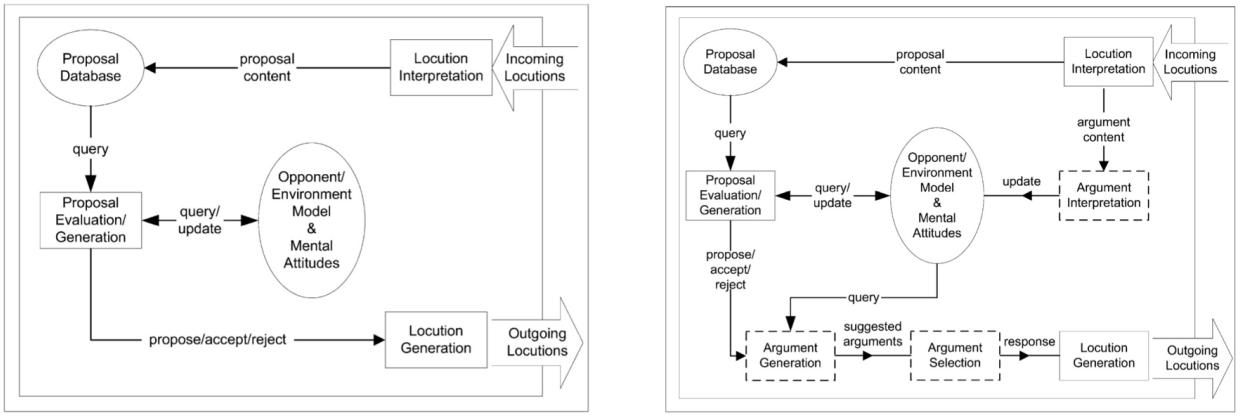
4.3. ARGUMENTATION-BASED NEGOTIATION (ABN)

A negotiation framework can be viewed in terms of its **negotiating agents** (with their internal motivations, decision mechanisms, knowledge bases, etc.) and the **environment** in which these agents interact. Thus, we can distinguish:

1- External elements of an ABN frameworks: they represent the environment in which the agents evolve. It includes:

- **Communication language** which are usually referred to as *locutions*, utterances or speach acts (see session 3). Traditional automated negotiation include the basic locutions such as PROPOSE for making proposal, ACCEPT for accepting a proposal and REJECT for rejecting proposals.
- **Negotiation protocol**. A protocol can be viewed as a fromal *set of conventions* governing the interaction among participants. This includes, as we have seen in session 3 , the interaction protocol as well as other rules of the dialogue. These rules will be discussed in the next session.

2- Internal elements of an ABN frameworks. If we take a look to a basic, non-ABN negotiating agent (see figure-left), a first component is *a locution interpretation*, which parses incoming messages. These locutions (see Course 3) usually contain a proposal, or an acceptance or rejection message of a previous proposal. Other information can be added such as the identity of the sender. The proposal is then stored in a *proposal database* for future reference. Then, proposals feed into a *proposal evaluation and generation* component, which makes a decision about whether to accept, reject or generate a counter-proposal. This finally feeds into the *locution generation* component which sends the response.



Conceptual elements of a classical negotiating agent

Conceptual elements of an argumentation-based negotiating agent (the dashed lined boxes represent the additional components)

Fig. 1: Classical negotiating agents vs Argument-based negotiating agent [Rahwan et al., 2004]

In contrast with this classical negotiating agent, more sophisticated meta-level information can be exchanged between ABN agents (see Figure-right). Indeed, in addition to evaluating and generating proposals, an agent capable of particiapting in ABN must be equipped with mechanisms for *evaluating* arguments and for *generating* and *selecting* arguments. The argument *generation* is responsible for deciding what response to actually send to the counterpart

and what (if any) arguments should accompagny the response. Deciding on which argument to actually send is the responsability of the *selection* mechanism.

More precisely, to design and build an agent capable of effective argumentation-based negotiation, it requires the following:

a- Mechanisms for passing proposals and their supporting arguments in a way that other agents understand.

b- Techniques for generating proposals (counter-proposals or critiques) and for providing the supporting arguments;

Proposal generation involves two main activities: (i) instantiating the negotiation object in accordance with the agent's acceptability region and its rating function; (ii) determining which argument(s) should accompany the agreement (if any) in order to maximise the likelihood of it being accepted. In terms of the latter point, in the majority of cases there will be many types of argument which can be made in support of a proposal (varying from explanations to threats). In determining which ones to send, the agent needs to pick those arguments which are most likely to be effective, but within the constraints of the agent's negotiation objectives. Thus, for example, continually issuing threats may provoke short-term gains, but may not be a good long-term strategy if the agent has to interact frequently with the same group.

c- Techniques for assessing proposals (counter-proposals or critiques) and their associated supporting arguments;

Received proposals need to be evaluated to determine how the agent should respond. This evaluation involves two main facets: (i) assessing the desirability of the proposal contained in the negotiation object; (ii) assessing the likely impact of the supporting arguments. From this, a number of potential outcomes are possible: the negotiation object is acceptable as it stands, the negotiation object alone is unacceptable but the supporting arguments overcome this and make the proposal acceptable, or the negotiation object is unacceptable and the supporting arguments are insufficient to warrant proposal acceptance. Having assessed the proposal, the agent may decide to update its acceptability region or rating function to reflect the incoming proposal's arguments.

d- Techniques for responding to proposals (counter-proposals or critiques) and their associated supporting arguments;

Having assessed a proposal, the agent can respond by accepting it, by rejecting it, by generating a critique, or by returning a counter-proposal. So the first functional requirement is to determine which of these courses of action should be taken. In the case of a critique, the agent has to determine what components it wants to accept and which it wants to reject, which issues it intends to provide constraints on, and what such constraints should be. It must then decide what arguments (if any) it will offer in support of this stance, and how it should respond to any arguments which accompanied the incoming proposal (varying from ignoring them to trying to undermine them). Counter-proposals are handled in a broadly similar manner, except that rather than giving feedback and constraints the agent has to instantiate the negotiation object with particular values.

This session is dedicated for implementing the first components of an ABN agents (generating and evaluating proposals) and its decision environment. We will tackle the negotiation rules and protocol during the two next sessions.

CHAPTER

TWENTYFIVE

4.4. PRACTICAL WORK: THE STORY...

Imagine that a car manufacturer wants to launch on the market a new car. For this, a crucial choice is the one of the engine that should meet some technical requirements but at the same time be attractive for the customers (economic, robust, ecological, etc.). Several types of engines exist and thus provide a large offers of cars models: essence or diesel Internal Combustion Engine (ICE), compressed natural GAS (CNG), Electric Battery (EB), Fuel Cell (FC), to cite a few. The company decides to take into account different criteria to evaluate them: Consumption, environmental impact (CO2, clean fuel, NOX,...), , cost, durability, weight, targeted maximum speed, etc. To establish the best offer/choice among a considerable set of options, she decides to simulate a negotiation process where agents, with different opinions and preferences (even different knowledge and expertise), discuss the issue to ends-up with the best offer. Agents may correspond to ..The simulation will offre the compagny the possibility to simulate different behaviors, typology of agents (expertise, role, preferences, ...) at a lower cost within reasonable time.

The practical sessions in this Multi-Agent System Course will be devoted to the programming of the negotiation and argumentation simulation. Agents will need to negotiate with each other to make a common decision regarding the choice of the best engine. The negotiation comes when the agents have different preferences on the criteria and the argumentation will be used to help them to decide which item to select. Moreover, the arguments supporting the best choice will help to build the justification supporting such a choice, an important element for the company to build its marketing campaign.

25.1 1. Some assumptions–only to ease the programming

As we are limited in the time and the idea is not to built at the end a software, but to understand the different concepts described in the course, we will take the following assumptions to ease the programming:

- the example illustrates only three agents for the moment !
- The agents share the same set of options (items) and the same set of criteria.
- The negotiation protocol is run only between each pair of agents.
- We will not update or modify the knowledge base of an agent.

25.2 2. An illustrative example

Let consider three agents: Agent1, Agent 2 and Agent3. They have to select only one item between the ICE Diesel (ICED) engine and the Electric (E) one: there is only room left for one of them. The agents consider five different criteria: C_1 : Cost (of production), C_2: Consumption, C_3: durability, C_4: Environment impact, C_5: Degree of Noise. Moreover, each agent has it own evaluation table for the items. For instance, the following performance table corresponds to Agent1.

	$C_1 \downarrow$	$C_2 \downarrow$	$C_3 \uparrow$	$C_4 \downarrow$	$C_5 \downarrow$
ICED	Very Good	Good	Very Good	Very Bad	Very Bad
E	Bad	Very Bad	Good	Very Good	Very Good

The scale for each criterion ranges from Very Bad to Very Good. The majority of criteria are to be minimized $\downarrow$ (the lower the better) except C_3 which is to be maximized $\uparrow$. Morever, agents have different order of preferences among the criteria themselves (total order in our example):

Agent1 : Cost $\succ$ Environment Impact $\succ$ Consumption $\succ$ durability $\succ$ Noise

Agent2 : Environment Impact $\succ$ Noise $\succ$ Cost $\succ$ Consumption $\succ$ durability

Agent3 : Durability $\succ$ Environment Impact $\succ$ Noise $\succ$ Consumption $\succ$ cost

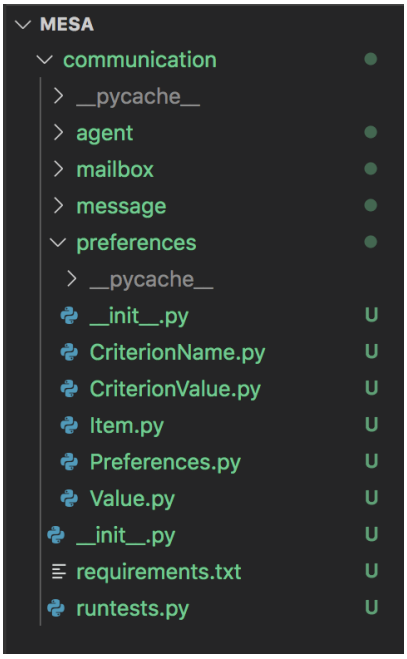
---

Note: Will not discuss the rating and how the values are obtained, it can be computed by taking into account different sources, different statical analysis, etc. Moreover, we can use another scale (you may decide to change it).

25.3 3. The Python project

In the third session of the course, in section 3.2. "Interaction with mesa libreary", we have implemented a communication layer in Mesa to handle the direct interactions. In this practical work we will use this layer and add argumentation and negotiation features. To do so, you can either download the following `archive` (preference package) or start from the following `archive` in which the preference package is already integrated in mesa.

Thus, you should have the following:



For reminder,

- **communication**: the root folder of the communication layer;
- **agent**: the folder which will contain the implementation of the communicating agent class;
- **mailbox**: the folder which will contain the implementation of the mailbox class;

- **message**: the folder which will contain the implementation of the message and performative class.

25.4 4. What is our goal?

To give you an idea of what is the target of this practical work, in what follows is the algoriothm of the argumentation-based negotiation that we will try to implement. You can examine the algorithm to get a first idea of the structure and content. A complete understanding of the algorithm is not necessary at the moment, as the different steps in the next sessions will allow you to build it.

1 Intialization

- create n agents: $\{agent_i, \ldots, agent_n\}$
- set-up a list of objects: $\{o_1 \ldots, o_m\}$
- set-up a list of criteria: $\{c_1, \ldots, c_k\}$

foreach $i \in n$ **do**
  define preferences of i (criteria order, a performance table)
end
foreach $(X, Y) \in agents$ **do**
  X: **propose**(o_i) # here the item is selected randomly
  Y **receives-message(propose**(o_i)**):**
  if *($o_i \in$ 10% favorite items of Y)* **then**
    if *($o_i = o_j$ / o_j top item of Y)* **then**
      Y: accept(o_i)
      X: commit(o_i)
      Y: commit(o_i)
    end
    else
      Y: propose(o_j)
    end
  end
  else
    Y: ask_why(o_i)
  end
  X **receives-message(ask_why(**o_i**)):**
  if *(X has at least one argument pro o_i)* **then**
    X: argue(o_i, reasons)
  end
  else
    X propose(o_j) another item.
  end
  Y **receives-message(argue(**o_i**, reasons)):**
  if *(Y has counter-argument(o_i))* **then**
    if *reasons=($c_i = x$)* **then**
      switch *reply* **do**
        case *1* **do**
          Y: argue(not p, $c_j = y$ and $c_j \succ c_i$)
        end
        case *2* **do**
          Y: argue(not p, $c_i = y$ and y is worst than x)
        end
        otherwise do
          Y: argue(p', $c_i = y$ and y is better than x)
        end
      end
    end
    if *reasons=($c_i = x$ and $c_i \succ c_j$)* **then**
      switch *reply* **do**
        case *1* **do**
          Y: argue(not p, $c_j = y$ and $c_j \succ c_i$)
        end
        otherwise do
          Y: argue(p', $c_i = y$ and y is better than x)
        end
      end
    end
  end
  else
    Y: accept(o_i)
    X: commit(o_i)
    Y: commit(o_i)
  end
end

Warning: From now do not forgot to make unit tests for each function implemented to check the behavior of your code.

So let start !

CHAPTER

TWENTYSIX

4.5. AGENTS' PREFERENCES

26.1 1. The preference package

Thus, the the preferences package includes:

- `Item` class: encodes the items (engines). Each item is described by a name represented by a String value and a descrcription represented by a String value;
- `CriterionName` class: implements the possible criterion name (e.g. environment impact, Cost, etc.)
- `CriterionValue` class : associates an Item with a CriterionName and a Value.
- `Value` class: implements the Value class Enumeration containing the possible values (e.g. Bad, Good, etc.)
- `Preferences` class: whose instances represent the preferences of agents. One agent will be associated with a single instance of this class. The preferences consists of: a list of criteria ordered (from most important to least important) and a list of value about each item on each criterion.

Note: The provided package is only to help you to start, you are of course allowed to change the structure of the code and to add your own classes and functions.

26.2 2. Testing your `Preferences` class

1- Using the methods present in the `Preferences` class and the get_score(self, preferences) function that computes the value of each item (see `Item` class), add to your class a method that allows an agent to select its most preferred item in a list. In case of equality, select one randomly. If you think that it is hlepful to add another function of your choice, please feel free to do it.

```
def most_preferred(self, item_list):
     """Returns the most preferred item from a list.
     """
    # To be completed
     return best_item
```

2- Add to your class a method that checks whether an item belongs to the 10% most preferred one of the agent.

```
def is_item_among_top_10_percent(self, item):
    """
    Return whether a given item is among the top 10 percent of the preferred items.

    :return: a boolean, True means that the item is among the favourite ones
```

(continues on next page)

(continued from previous page)

```
    """
    # To be completed
```

3- Run the unit tests at the end of the `Preferences` class (Prefereces.py) to check wether you code is correct or not. You should get the following:

```
Diesel Engine (A super cool diesel engine)
Electric Engine (A very quiet engine)
Value.VERY_GOOD
True
Electric Engine > Diesel Engine : False
Diesel Engine > Electric Engine : True
Electric Engine (for agent 1) = 362.5
Diesel Engine (for agent 1) = 525.0
Most preferred item is : Diesel Engine
```

You can add other units tests not included in the file.

CHAPTER

TWENTYSEVEN

4.6. AGENTS AND MESSAGES

27.1 1. Performatives and Messages content

During the dialogue (negotiation), we will need some locutions/performatives. we will mainly consider the following:

- PROPOSE: it is used to propose to select an item. Its content is the name of the item.
- ACCEPT: it is used to accept to select an item. Its content is the item's name. It should always appear after a PROPOSE of the same item, but several other message can exist between the proposal and the acceptance.
- COMMIT: it is used to confirm that an object has to be selected. Its content is an item's name. Both agents must send (and receive) this message at the end of the interaction.
- ASK_WHY: it is used to ask another agent to propose arguments for a given item. Its content is an item's name. It should be used after a propose. This message expects an ARGUE message as answer. This latter is decribed in the next session.

You should check that these performatives are implemented in MessagePerformative.py, if not make sure to add the missing ones. Again, you can add other perofmatives (for instance, REJECT. You need then to update the protocol to integrate this new locution).

27.2 2. Agents and communication

1- Create a new Python file named *pw_argumentation.py* (as it is illustrated in the following picture) at the root of the mesa folder (or download it `here`).

This file will contain our class of agents and our model for the argumentation simulation (will be completed as we go through our sessions).

Translated with www.DeepL.com/Translator (free version)).

```
from mesa import Model
from mesa.time import RandomActivation

from communication.agent.CommunicatingAgent import CommunicatingAgent
from communication.message.MessageService import MessageService


 class ArgumentAgent(CommunicatingAgent):
  """ ArgumentAgent which inherit from CommunicatingAgent.
  """
  def __init__(self, unique_id, model, name):
      super().__init__(unique_id, model, name)
      self.preference = None

  def step(self):
      super().step()

  def get_preference(self):
      return self.preference

  def generate_preferences(self, preferences):
      # see question 3
      # To be completed

 class ArgumentModel(Model):
  """ ArgumentModel which inherit from Model.
  """
  def __init__(self):
      self.schedule = RandomActivation(self)
      self.__messages_service = MessageService(self.schedule)

      # To be completed
      #
      # a = ArgumentAgent(id, "agent_name")
      # a.generate_preferences(preferences)
      # self.schedule.add(a)
      # ...

      self.running = True

  def step(self):
      self.__messages_service.dispatch_messages()
      self.schedule.step()


 if __name__ == "__main__":
  argument_model = ArgumentModel()

  # To be completed
```

2- As it was done at the end of the course 3 (Alice/Bob example), create three communicating agents named Agent1, Agent2 and Agent3 (you may give them names if you want). These agents have an instance of `Preferences`. The list of items is the same for the different agents (the list of criteria may be not, you choose).

3- Write a method that allows to generate the preferences of the agent (see the example for Agent1 and its corresponding ranking on criteria). You can also try to have a function that read the data from a csv file (not mandatory).

```
def generate_preferences(self, preferences):
    """
     Set the preferences of the agent

     """
     # To be completed
```

4- Implement the following situation between the three agents:

- On their turn, Agent1 and Agent2 **ask** Agent3 its most preferred item.
- If the item is different from their most preferred one, they send a message to Agent3 to remove the item from its list.
- On its turn, Agent3 reads its mailbox and processes all messages: Messages that request information about the preferred item produce an anwer; and Messages that request a change to the list are applied.

5- Now implement a simple propose/accept interaction between Agent1 and Agent2, such that:

- Agent1 to Agent2 : PROPOSE (item) (no matter which one for the moment)
- Agent2 to Agent1 : ACCEPT (item).

It means that Agent1 sends a message and Agent2 reads it mailbox and replies.

6- Update the propose/accept interaction between Agent1 and Agent2, with the following:

- Agent1 to Agent2: PROPOSE (item)
- Agent2 to Agent1: ACCEPT (item) if the item belongs to **its 10% most preferred item**, otherwise ASK_WHY(item).

7- In case of acceptance, update the protocol to take into account the double-COMMIT interaction (see below). Both agents (Agent1 and Agent2) simply send a COMMIT message to each other as soon as they have the three following information:

- One agent made a proposal on the item;
- The other agent accept the proposal on the item;
- The item is available in the agent's list.

  Agent1 to Agent2: PROPOSE (item)

  Agent2 to Agent1: ACCEPT (item)

  Agent1 to Agent2: COMMIT (item)

  Agent2 to Agent1: COMMIT (item)

After receiving the COMMIT each agent remove the item from its list of items.

Ortherwise Agent1 to Agent2: ARGUE()

For the moment the content of this message is empty, more in the next session.

CHAPTER

TWENTYEIGHT

5. ARGUMENTATION-BASED NEGOTIATION (CONT.)

28.1 Agenda – Recall

1. Friday, March the 5th, 2021 : Introduction to MAS : definitions and implementation of a platform
2. Friday, March the 12th, 2021 : Multiagent simulation : preys and predators
3. Friday, March the 19th, 2021 : Interaction mechanisms : models and implementation
4. Friday, March the 26th, 2021 : Argumentation-based negotiation I: Practical session
5. Friday, April the 2nd, 2021 : Argumentation-based negotiation II: what is argumentation?
6. Friday, April the 9th, 2021 : Argumentation-based negotiation III: what is a negotiation protocol?
7. Friday, April the 16th, 2021 : Argumentation-based negotiation IV: Practical session

28.2 Session 5.

Argumenation Theory is a growing field of Artificial Intelligence. In short, it is the process of constructing and evaluating arguments in order to justify conclusions. The aim of this session is to give an overview of different concepts and notions of an argumentation process. We introduce the general idea of non monotonic reasoning and discuss the different steps of an argumentation process. Indeed, different levels can be associated to such process: the logical level which provides the logical structure of a single arguments; the dialectical level which address the notion of conflict between arguments and the procedural level which introduces how we can use arguments within a dialogue.

28.3 some references

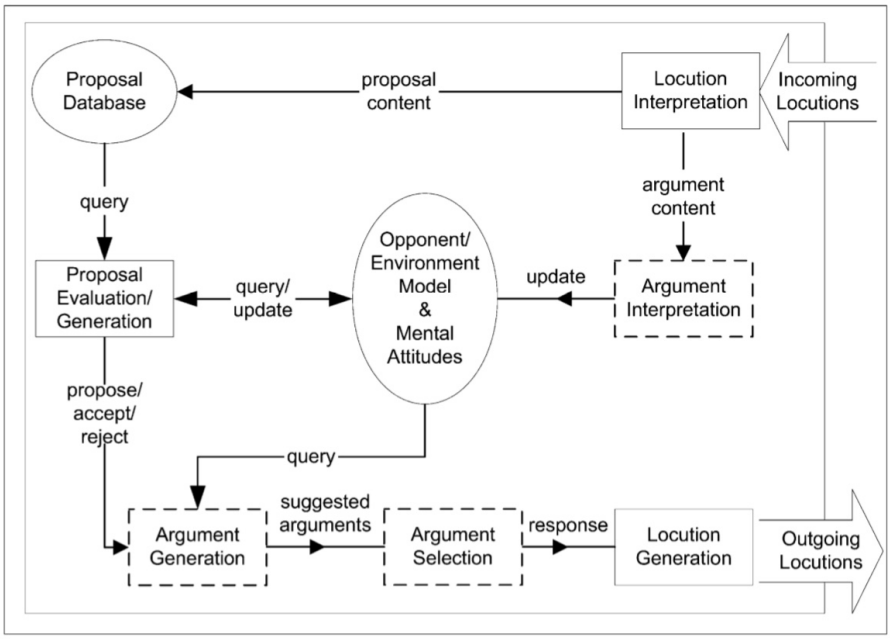
- Ferber, J. (1995), *Les Systèmes Multi-Agents*, InterEditions. **(French version)**
- Ferber, J. (1999), *Multi-agent systems: An introduction to distributed artificial intelligence*, Addison Wesley. **(English version)**
- Michael Wooldridge (2002), *An Introduction to MultiAgent Systems*, John Wiley & Sons Ltd.
- The AgentLink roadmap
- Rahwan, Iyad (2009), *Argumentation in Artificial Intelligence*,Springer.
- Lopes, Fernando & Coelho, Helder. (2014). *Negotiation and Argumentation in Multi-Agent Systems: Fundamentals, Theories, Systems and Applications*. Bentham Science Publishers.
- Argumentation in Multi-Agent Systems (ArgMAS) Workshop Series

CHAPTER

TWENTYNINE

5.1. PREVISOUSLY . . .

In the last session, we started the discussion about an argumentation-based negotiation agent (see figure below for recall).



In this session we will pursue the discussion and address more precisely what do we mean by arguing and argumentation. We are concerned with argument genration, evaluation and selection.

CHAPTER

THIRTY

5.2. INTRODUCTION TO ARGUMENTATION THEORY

Reasoning is generally defeasible, i.e. based on assumptions, exceptions, uncertainty, etc. AI formalises such reasoning with **non-monotonic reasoning logics**. The idea is that often available knowledge is incomplete, and to model commonsense reasoning it is necessary to be able to jump to plausible conclusions from the given knowledge. Thus, to do that it is necessary to make assumptions. The choice of assumptions is not blind: most of the knowledge on the world is given by means of general rules which specify typical properties of objects.

Non-monotonic reasoning deals with problem of deriving plausible conclusion, but not infallible, from a knowledge base (a set of formulas). Since the conclusion are not certain, it must be possible to **retract** some of them if **new information** shows that they are wrong. Classical logic is inadequate since it is monotonic, which means that if a formula B is derivable from a set of formulas S, then B is also derivable from any superset of S

$$S \vdash B \; implies \; S \cup \{A\} \vdash B, \; for \; any \; formula \; A$$

Example 1 (Reiter, 1987)

- Birds fly,
- Tweety is a Birds,
- Therefore, Tweety flies.

But what if *tweety is a penguin*, type of bird that does not fly, is added to the knowledge base (KB)?

A knowledge base is a set of propositions represented in some formal logics (classical or non-monotonic).By adding this information to the KB, we have good reasons to retract the previous inference and instead, the new conclusion that Tweety does not fly will hold. However, this is not possible with the classical logic (the three sentences in the premises are true so the derived conclusion is true).

One way to do this is to say, ''Normally, birds fly" or '' if x is a typical bird, then we can assume by default that x flies ''. This can be read as: "in the absence of information to the contrary, assume that x flies"

Argumentation logics formalises such **defeasible reasoning** as construction and comparison of arguments. In other terms, Argumentation is the process of attempting to agree about what to believe or **what to do**[1]. We use argumentation only when information, beliefs, actions, ... are **contradictory**. It means if eveyrhting is consistent, just merge information from multiple agents. In other terms, the process offers sensible rules for deciding what to believe or what to do in the face of inconsistency. argumentation-based approaches are suited for applications in which agents have incomplete or inconsistent information about each other and the environment in which they act. For instance, in : consumer websites (Amazon for instance), law (policy making, supreme transcript, case based reasoning,...), mediacal diagnosis, social media, etc.

Several works were interested by using argumentation in different fields, such as legal reasoning, multi-agent systems, decision making, etc. (Prakken and Sartor, 2002; Amgoud and Prade, 2006 ; Parsons and McBurney, 2003, ...). In most of such approaches, arguments are constructed from an *inconsistent knowledge base*. In this case, argumentation can be viewed as a method for **deducing** justified conclusion from an inconsistent knowledge base. Which conclusion

[1] In this course we are mainly interessted by decision making. Thus, we will discuss the structure of arguments withing this setting.

are justified depends on the attack and the defeat relations among the arguments which can be constructed from the knowledge base (see relations among arguments).

The classical steps in an argumentation process are depticted in the following figure.

It includes:

1- Constructing arguments in favor/against statements (pertaining to beliefs or decisions) (see section *Modelling arguments*),

2- Defining relations among arguments: it may include differents tasks. Such as: evaluating the strength of each argument, determining the different conflicts among arguments, etc (see Sections *selecting arguments* and *evaluating arguments*).

3- Evaluating the acceptability of arguments (see section *status of arguments*), and

4- Comparing decisions on the basis of relevant "accepted" arguments (see next course/session).

In the following, we will detail what do we mean by an argument, an attack relation and how the arguments are used within a negotiation process.

30.1 1. Modelling arguments

In this section we are interested by the the first step of an argumentation process, i.e. the construction of arguments in favor or against a statement (we recall that for our purpose the statements correspond to a decision/option). This step corresponds also to the box "agument generation" of our ABN agent (see Figure above).

The main challenge in this step is to determine the *form or the structure of the arguments*. Indeed, there are many ways to address the form of an argument: *as trees of inferences* (Karacapilidis and Papadias, 1998a), as a *sequences of inferences* (deductions) (Verheij, 1996), or as simple *premiss-conclusion pairs*. The different forms of arguments depend on the language and on the rules for constructing them. The choice between the different options depends on the context and the objective sought through the use of argumentation.

In this course, we will adopt the *premiss-conclusion pair* as a structure of an argument, which is defined as follows:

Definition 1 (An argument)

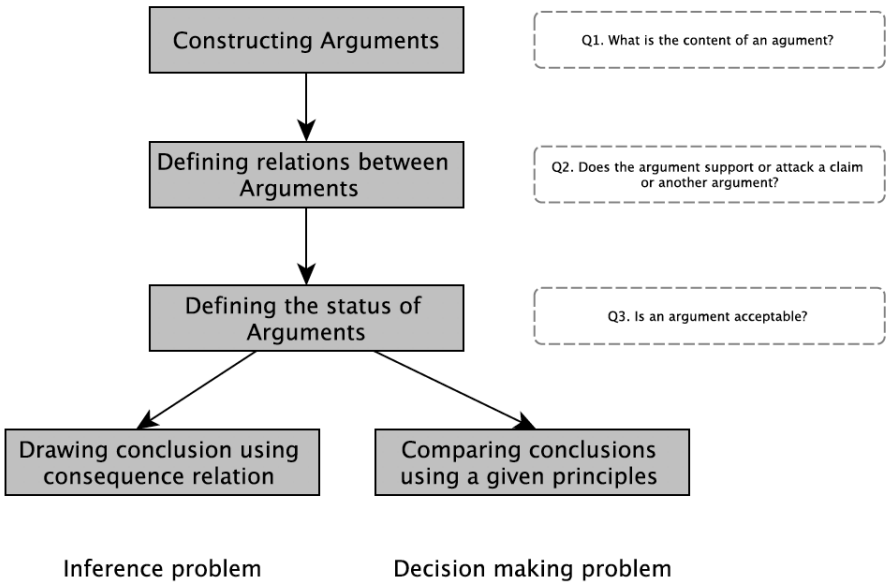


Fig. 1: Generic Steps of an Argumentation Process

Let Σ be a propositional knowledge base. An argument A is a pair (Φ, α) such that: (1) $\Phi \subset \Sigma$, (2) Φ is consistent (i.e. if there is no formula ϕ such that $\Phi \vdash \phi$ and $\Phi \vdash \neg\phi$), (3) $\Phi \vdash \alpha$ and (4) Φ is minimal (for set inclusion) satisfying 1, 2 and 3. We call Φ: premises (supports) and α : the conclusion (consequence).

Example 2

A murder has been performed and the suspects are Liz, Mary and Peter. The following pieces of information have been gathered:

- The type of murder suggests us that the killer is a female;
- The killer is certainly small;
- Liz is tall and Mary and Peter are small.
- The killer has long hair and uses a lipstick;
- A witness claims that he saw the killer who was tall;
- We are told that the witness is short-sighted, so he is no more reliable.

We can encode the previous statements as the following:

- s (the killer is small);
- f (the killer is a female);
- m (the killer is Mary);
- l (the killer has long hair and uses a lipstick);
- w (the witness is reliable);
- b (the witness is short-sighted).

Thus we can construct the following arguments:

- $A_1 : < \{s, f, s \wedge f \rightarrow m\}, m >$ (in favour of m);
- $A_2 : < \{w, w \rightarrow \neg s\}, \neg s >$ (in favour of $\neg s$);

- $A_3 : < \{b, b \rightarrow \neg w\}, \neg w >$ (in favour of $\neg w$);
- $A_4 : < \{l, l \rightarrow f\}, f >$ (in favour of f).

$\rightarrow$ *Who is the killer?* ... Wait, the answer is soon :-)

30.1.1 Let's go back to our Engine choice: Generating arguments.

At the core of the dialogue between our three agents, the performative **ARGUE** (you need to check that this performative is present in our MessagePerformative.py). It is used to explain to the other party in the dialogue the reasons in favor or against a given item. As it was mentionned in the course the idea is to communicate a logical inference that defends the agent's position. As a consequence, the content of such a message is of the form: $B \leftarrow A$, such that the terms of A are the **premises** and B a positive or negative literal, which is the conclusion.

Example 3

> An agent explains that it is in favor of Fuell Cell (FC) because it respects the environment. It can be encoded as follows:
>
> $$FC \leftarrow ecology\_impact = Very\,Good.$$

Thus, the two components of the argument are as follows:

1- **Premises**: we will limit ourselves to two types of propositions:

- The value of a criterion C_i is x. This is a know fact for both agents, but using it in an ARGUE message simply asserts that this information can be used by the sender to infer a position in favor or against the item. In our communciation model we shall write $C_i = x$.
- The Criterion C_i is prefered to the criterion C_j . This Information is local to the sender agent. It does not necessary hold for the receiver. However, using it in an ARGUE message informs the receiver about the sender's preference. In our communciation model, we shall write $C_i \succ C_j$.

Those two possible types of propositions can be combined in the same message. Thus, when several premises are used, they are separated by a comma.

Example 4

> ARGUE (not FC because Cost = Very Good, Cost $\succ$ CO2 emission)

2- **Conclusion**: it must be the name of an item, preceded by the operator "not" when the inference is against the item.

Warning: To generate arguments or to make a logical inference, each agent needs to have a knowledge base (KB), which contains different information, facts, domain knowledge... Due to time limitation, we restrict this base to the agent's preferences represented by the performance table (see previous session). This is clearly not sufficient to build diffrent kinds of arguments. Moreover, each time an agent receives an argument, he is able to update this knowledge base to enrich his information and to update his behavior and reasoning according to the new information. Again, this is not taken into account in this practical work in order to ease the programming and this is also due to time limitation.

30.1.2 Recall of the example

Let consider three agents: Agent1, Agent 2 and Agent3. They have to select only one item between the ICE Diesel (ICED) engine and the Electric (E) one: there is only room left for one of them. The agents consider five different criteria: C_1 : Cost (of production), C_2: Consumption, C_3: durability, C_4: Environment impact, C_5: Degree of Noise. Moreover, each agent has it own evaluation table for the items. For instance, the following performance table corresponds to Agent1.

	$C_1 \downarrow$	$C_2 \downarrow$	$C_3 \uparrow$	$C_4 \downarrow$	$C_5 \downarrow$
ICED	Very Good	Good	Very Good	Very Bad	Very Bad
E	Bad	Very Bad	Good	Very Good	Very Good

The scale for each criterion ranges from Very Bad to Very Good. The majority of criteria are to be minimized $\downarrow$ (the lower the better) except C_3 which is to be maximized $\uparrow$. Morever, agents have different order of preferences among the criteria themselves (total order in our example):

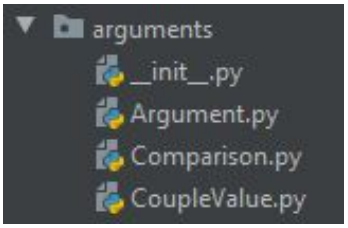
Agent1 : Cost $\succ$ Environment Impact $\succ$ Consumption $\succ$ durability $\succ$ Noise

Agent2 : Environment Impact $\succ$ Noise $\succ$ Cost $\succ$ Consumption $\succ$ durability

Agent3 : Durability $\succ$ Environment Impact $\succ$ Noise $\succ$ Consumption $\succ$ cost

30.1.3 Questions

1- In your folder you should have the package *arguments* as it is depicted in the following picture (you can download it `here` or download all the package of mesa `here`). This package contains three classes.



2- In the class `Comparison` implement the comparison object used in an argument object ($C_i \succ C_j$).

```
#!/usr/bin/env python3


class Comparison:
 """Comparison class.
 This class implements a comparison object used in argument object.

 attr:
     best_criterion_name:
     worst_criterion_name:
 """

 def __init__(self, best_criterion_name, worst_criterion_name):
     """Creates a new comparison.
     """
     # To be completed
```

3- In the class `CoupleValue` implement the couple value used in an argument object ($C_i = x$).

```python
#!/usr/bin/env python3


class CoupleValue:
 """CoupleValue class.
 This class implements a couple value used in argument object.

 attr:
     criterion_name:
     value:
 """

 def __init__(self, criterion_name, value):
     """Creates a new couple value.
     """
     # To be completed
```

4- In the class `Argument` implement an argument that consists of a tuple with:

- The item name;
- A boolean value (for positive or nagtive arguments)
- A list of couples taht can be either *(crtierion, value)* and/or *(criterion, criterion)*

```python
#!/usr/bin/env python3

from arguments.Comparison import Comparison
from arguments.CoupleValue import CoupleValue


class Argument:
   """Argument class.
   This class implements an argument used in the negotiation.

   attr:
      decision:
      item:
      comparison_list:
      couple_values_list:
   """

  def __init__(self, boolean_decision, item):
      """Creates a new Argument.
      """
      #To be completed

  def add_premiss_comparison(self, criterion_name_1, criterion_name_2):
      """Adds a premiss comparison in the comparison list.
      """
      # To be completed

  def add_premiss_couple_values(self, criterion_name, value):
      """Add a premiss couple values in the couple values list.
      """
      # To be completed
```

5- Write a method that constructs the list of positive arguments for a proposal (an item). In our context, positive arguments will correspond to the ones for which criteria have either GOOD or VERY GOOD values for the item (see

Value class in preferences implemented during the previsous session).

```
def List_supporting_proposal(self, item):
    """Generate a list of arguments which can be used to support an item
    :param item: Item - name of the item
    :return: list of all arguments PRO an item (sorted by order of importance based␣
↪on agent's preferences)
    """
    # To be completed
```

6- Write a method that constructs the list of negative arguments against an item. They will correspond to the ones for which criteria'a values are either Bad or Very Bad.

```
def List_attacking_proposal(self, item):
    """Generate a list of arguments which can be used to attack an item
    :param item: Item - name of the item
    :return: list of all arguments CON an item (sorted by order of importance based␣
↪on preferences)
    """
    # To be completed
```

Warning: Recall an argument is composed of a conclusion and a set of premisses, and we have two types of premisses.

30.2 2. Selecting Arguments

As it is depicted in the figure above antoher component in our ABN agent is the *argument selection*. The question is: given a number of candidate arguments that an agent may utter to its counterpart, which is the "best" argument from the point of view of the speaker?

Different stategies are possible. Just for illustration, in the work of (Kraus et al., 1998), arguments are selected according to the following argument strength order, with threats being the strongest argument. (1) Appeal to prevailing practice, (2) A counter example, (3) An appeal to past promise, (4) A promise to self-interested, (5) A promise of future reward, and (6) A threat.

Suitable argument selection in a negotiation context must take into account information about the negotiation counterpart. To this end, for instance is Bayesian game theory a counterpart is modelled by a probability distribution representing the uncertainty of the first party regarding the counterparts' initial information and the payoffs they receive from the different outcomes (i.e. action profiles). This raises the opportunity to use **learning** techniques in order to find patterns in the counterpart's behaviour an use these findings in future encounters with the same (or similar) counterpart(s) (see for e.g. Sandholm and Crites, 1995).

This is a particularly challenging task for ABN since agent may not only model the observed "behaviour" of one another, but also the "mental attitudes" motivating that behaviour.

Note: Instead what we are doing here, argument selection may take place in conjunction with argument generation. An agent needs not to generate all possible arguments before it makes a selection of the most suitable one.

30.2.1 Questions

For our context, to select which argument to use, we choose a simple way which is: *an agent takes the argument with the criterion with the highest position in its own preference order* (you may choose another stratgy if you want!). For instance, Agent1 (see its perfomance table above) has four positive arguments for the item ICED. To make a propsal, according to our rule, he will choose the argument involving *Cost* as it is its perefered criterion.

Consider now the following interaction:

Agent1 to Agent2: PROPOSE(item)

Agent2 to Agent1: ASK_WHY(item)

Agent1 to Agent2: ARGUE(item, premisses)

For the last turn the agent will use its "best" argument from its list of positive arguments (see previous question 5).

1- Write a method that allows an agent to select among a list of arguments its "best" one (according to our rule or your own rule) to support a given proposal.

```
def support_proposal(self, item):
        """
        Used when the agent recieves "ASK_WHY" after having proposed an item
        :param item: str - name of the item which was proposed
        :return: string - the strongest supportive argument
        """

        # To be completed
```

2- Update and test the interaction protocol to implement the situation above.

30.3 3. Relations among arguments.

Once the arguments constructed, they cannot be considered independently. Indeed, most of the arguments are in interaction: arguments may be conflicting or on the contrary, arguments may support other arguments. (Pollock, 1995) drew an important distinction between two kinds of arguments that can attack and defeat another argument, calling them **rebutting** defeaters and **undercutting** defeaters.

A rebutting attack concerns arguments that have contradictory conclusions. Let $\equiv$ represents the equivalence relation in classical logic, and $\neg$ the negation. Then formally,

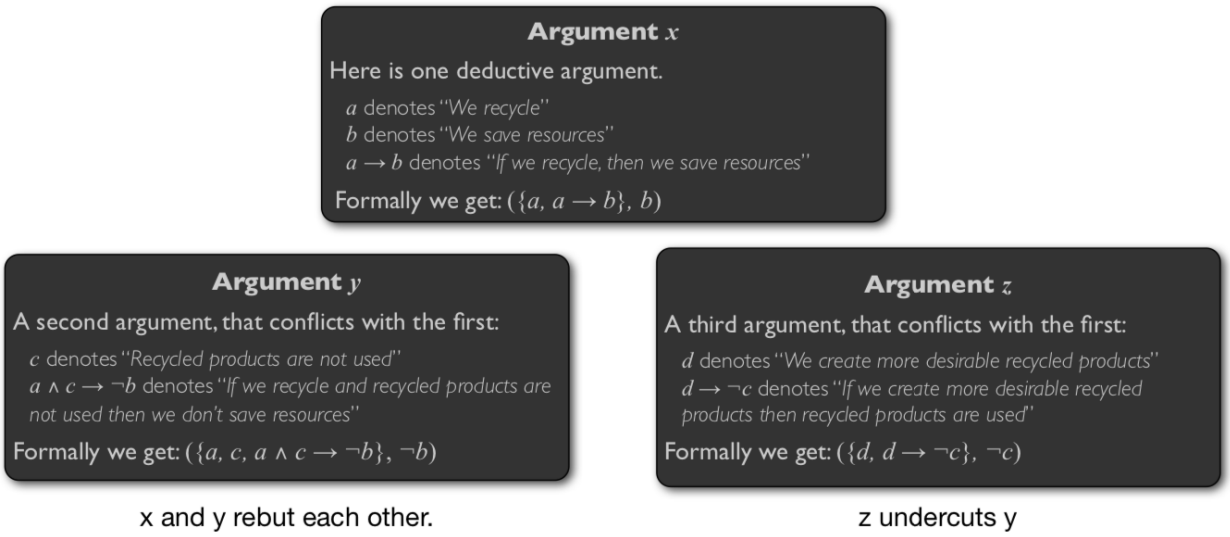
Defintion 2 (Rebut attack) Given two arguments[2] (ϕ_1, α_1) and (ϕ_2, α_2) , (ϕ_1, α_1) rebuts (ϕ_2, α_2) if and and only if $\alpha_1 \equiv \neg\alpha_2$

An undercutting defeater has a different claim. It attacks the inferential link between the conclusion and the premise rather than attacking the conclusion. In other words, we are in presence of undercutting when one argument challenges a rule inference of another argument. Formally,

Defintion 3 (Undercut attack) Given two arguments (ϕ_1, α_1) and (ϕ_2, α_2) , (ϕ_1, α_1) undercuts (ϕ_2, α_2) if and and only if $\exists\alpha \in \phi_2$ such that $\alpha_1 \equiv \neg\alpha$

Example 4

[2] In the sens of defintion 2.



Moreover, let us remark that both types of defeaters, discussed above, represent negative relations between arguments. However, recent studies have proposed another kind of relation between argument, namely a *positive relation*, called *support* relation [Amgoud et al., 2004; Karacapilidis and Papadias, 1998]. Indeed, an argument can defeat another argument, but it can also support another one. This new relation is completely independent of the attack relation (i.e., the support relation is not defined in terms of the attack relation, and vice-versa). As it was pointed by [Cayrol and Schiex, 2005]:

> "This suggests a notion of bipolarity, i.e. the existence of two independent types of information which have a diametrically opposed nature and which represent contrasting forces"

Example 5

Consider again the arguments of the murder example.

- $A_1 :< \{s, f, s \wedge f \rightarrow m\}, m >$ (in favour of m);
- $A_2 :< \{w, w \rightarrow \neg s\}, \neg s >$ (in favour of $\neg s$);
- $A_3 :< \{b, b \rightarrow \neg w\}, \neg w >$ (in favour of $\neg w$);
- $A_4 :< \{l, l \rightarrow f\}, f >$ (in favour of f).

Relations:

- A_3 defends A_1 against A_2
- A_4 confirms the premiss of A_1 thus, A_4 *supports* A_1

Thus, as it is represented in the argumentation process (see Figure above), once you have identified your arguments, you need to define the relations among those arguments. Moreover, at the end, given the set of arguments and their attack relations, we can construt a graph such that nodes represent the arguments and the edges the relations between the arguments.

Example 6

The following figure represents a graph where we have 5 arguments (each color represent an argument) and 4 attack relations (represented by the red dashed lines). Note that the black lines inside a color represent the logical inference of an argument from the premises to the conclusion.

More generally, the arguments and their relations can be represented by what we call "abstract argumentation system". For our previsous example we may have the following:

Example 7

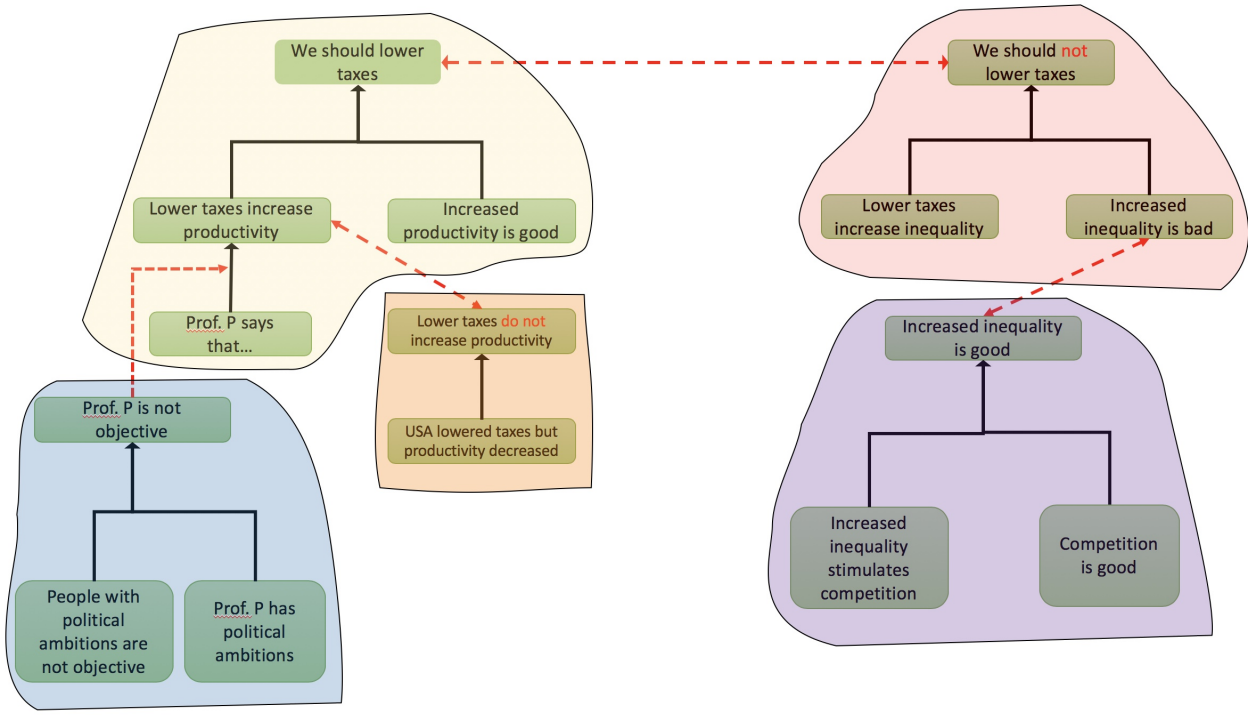
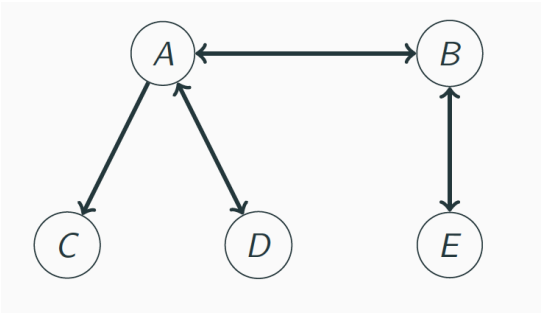


Fig. 2: Example of an argument graph (Prakken, 2014)



More formally, we define an argumentation framework or system as follows:

Definition 4 (Argumentation System)

an argumentation system is a pair $\langle \mathcal{A}, \mathcal{R} \rangle$, where: $\mathcal{A}$ is a set of arguments and $\mathcal{R} \subset \mathcal{A} \times \mathcal{A}$: an attack relation among arguments.

Now, the key question to ask about such an argumentation framework is whether a given argument, $x \in \mathcal{A}$, should be *accepted*? (see the next section)

Before to go further, let us consider again our deicison problem about engine choice and let's build a system for generating arguments for our negotiation.

30.3.1 Questions

1- Write a method that take as input an argument and recovers its premises and its conclusion.

```
def argument_parsing(self, argument):
    """ returns ....
    :param argument:
    :return:
    """
    # To be completed
```

2- Write a method that decides whether an argument can be attacked or not. In our context the agent can attack or contradict an argument provided by another agent if:

- The criterion is not important for him (regarding his order)
- Its local value for the item is lower than the one of the other agent on the considered criteria
- He prefers another item and he can defend it by an argument with a better value on the same criterion.
- etc.

Warning: Of course, once it is decided that the agument can be contradicted, you need to construct its counter-arguement (or to select it in your list of counter-arguments–see question 6 in section "Modelling arguments").

Example 8

Agent1 to Agent2: argue(ICED, Consumption = Good)

Agent 2 may reply by:

- argue (not ICED, environment impact $\succ$ Consumption)
- argue (item, Consumption= Very Good)
- argue (not ICED, Consumption = Bad)
- argue (note ICED, environment impact = Bad and environment impact $\succ$ Consumption)

3- Update the interaction protocol to take into account the attack function and to implement the following situation:

Agent1 to Agent2: PROPOSE(item)

Agent2 to Agent1: ASK_WHY(item)

Agent1 to Agent2: ARGUE(item, premisses)

Agent2 to Agent1: reply with one of the previous counter-arguement –> ARGUE(not item, prmisses) or ARGUE (new item, premisses)

4- In addition, at each ARGUE message, **store in a data structure the set of exchanged arguments**, such that it is possible to extract for each argument its set of counter-arguments.

30.4 4. Status of arguments?

Note: You can skip for the moment this section and go ahead for the next course.

In an argumentation process, it is important to define the status of arguments (or to evaluate them) on the basis of all the ways in which they interact. Thus, the best or acceptable arguments must be identified at the end of the argumentation process.

Most of argumentation systems are based on the notion of acceptability as it was identified by [Dung, 1995]. Dung has proposed an abstract framework for argumentation in which he focuses only on the definition of the status of arguments. In such framework, the acceptability of an argument depends on its membership of some sets, called *acceptable sets or extensions*. In other terms, the acceptability of arguments is defined without considering the internal structure of the arguments.

Different extensions exists, namely:

- *Admissible set*: it is a minimal notion of a reasonable position (internally consistent and defends itslef, and it is coherent, defendable position).
- *Stable set*: un argument is not accepted because it is attacked by at least one accepted argument;
- *Preferred extension*: it represents maximal coherent positions, able to defend themselves against all attackers;
- *Grounded extension*: accept only the argument that one cannot avoid to accept.

Fore more formal defintions of the different extensions, and examples, you can have a look at this `summary`.

30.4.1 *Notes*

CHAPTER

THIRTYONE

6. ARGUMENTATION-BASED NEGOTIATION (CONT.)

31.1 Agenda – Recall

1. Friday, March the 5th, 2021 : Introduction to MAS : definitions and implementation of a platform
2. Friday, March the 12th, 2021 : Multiagent simulation : preys and predators
3. Friday, March the 19th, 2021 : Interaction mechanisms : models and implementation
4. Friday, March the 26th, 2021 : Argumentation-based negotiation I: Practical session
5. Friday, April the 2nd, 2021 : Argumentation-based negotiation II: what is argumentation?
6. Friday, April the 9th, 2021 : Argumentation-based negotiation III: what is a negotiation protocol?
7. Friday, April the 16th, 2021 : Argumentation-based negotiation IV: Practical session

31.2 Session 6.

This session is devoted to discuss the notion of dialogue system in general and negotiation protocol in particular. We will focus on negotiation procotol that relies in the exchanged messages on arguments to support the proposal. More precisely, in this session will tackle the last part of the the practical work, i.e. the implementation of the interaction algorithm between our agents (which corresponds of course to an argumentation-based negotiation dialogue).

31.3 Some references

- Ferber, J. (1995), *Les Systèmes Multi-Agents*, InterEditions. **(French version)**
- Ferber, J. (1999), *Multi-agent systems: An introduction to distributed artificial intelligence*, Addison Wesley. **(English version)**
- Michael Wooldridge (2002), *An Introduction to MultiAgent Systems*, John Wiley & Sons Ltd.
- The AgentLink roadmap
- Rahwan, Iyad (2009), *Argumentation in Artificial Intelligence*,Springer.
- Lopes, Fernando & Coelho, Helder. (2014). *Negotiation and Argumentation in Multi-Agent Systems: Fundamentals, Theories, Systems and Applications*. Bentham Science Publishers.
- Argumentation in Multi-Agent Systems (ArgMAS) Workshop Series

CHAPTER

THIRTYTWO

6.1. ARGUMENTS IN DIALOGUE

In the previous session, we have seen what is an argumentation system without any reference to interaction. However, arguments can be seen as embedded in a procedural context, in that they are put forward on one side or the other of an issue during a dialogue between human and/or artificial agents. In other terms, one way to define argumentation logics is in the dialectical form of dialogue games (or dialogue systems). Such games model interaction between two or more players, where arguments in favour and against a proposition are exchanged according to certain rules and conditions [Carlson, 1983].

According to [Gordon et al., 2007]

"The information provided by a dialogue for constructing and evaluating argument is richer that just a set of sentences. Indeed, the context can tell us whether some party has questioned or conceded a statements, or whether a decision has been taken to accept or reject a claim."

CHAPTER

THIRTYTHREE

6.2. DIALOGUE SYSTEM

Dialogue systems essentially define the principle of coherent dialogue and the conditions under which a statement made by an individual is appropriate. Different formal dialogues exist, taking into account various information, such as: participants, communication language, roles of participants, the dialogue goal, etc.

In what follows we present the main rules to take into account when designing a dialogue system. Of course, we can have other rules depending of the situation and the nature of the problem. For more details on dialogue systems, we refer the reader to, for instance, (Prakken, 2005 ; McBurney and Parsons, 2009).

- **Admission rules**, which specify when an agent can participate in a dialogue and under what conditions;
- **Locutions rules (speech acts, moves)**. Rules which indicate what utterances are permitted. Typically, legal locutions permit participants to assert propositions, permit others to question or contest prior assertions, and permit those asserting propositions which are subsequently questioned or contested to justify their assertions. Justifications may involve the presentation of a proof of the proposition or an argument for it.
- **Commitments rules**. Rules defining the effect of the moves in the "commitment stores". Indeed, associated with each player is a commitment store, which holds the statements players have made and the challenges they have issued. There are then rules which define how the commitment stores are updated. For example, a question posed by one agent to another may impose a commitment on the second to provide a response; until provided, this commitment remains undischarged.
- **Dialogue rules (protocol)**, for regulating the moves. It specifies for instance the set of speech acts allowed in a dialogue and their allowed types of replies. Various dialogue protocols can be found in the literature, especially for persuasion [Prakken, 2001] and negotiation [Parsons et al., 1998]. For example, the set of rules given in the following table correspond to a persuasion dialogue.

Acts	Attacks	Surrenders
claim ϕ	why ϕ	concede ϕ
why ϕ	argue A (conclusion(A) = ϕ)	retract ϕ
argue A	why ϕ ($\phi \in$ premise(A)) argue B (B attacks A)	concede ϕ ($\phi \in$ premise(A) or ϕ= conclusion(A))
concede ϕ		
retract ϕ		

Fig. 1: Examples of speech acts

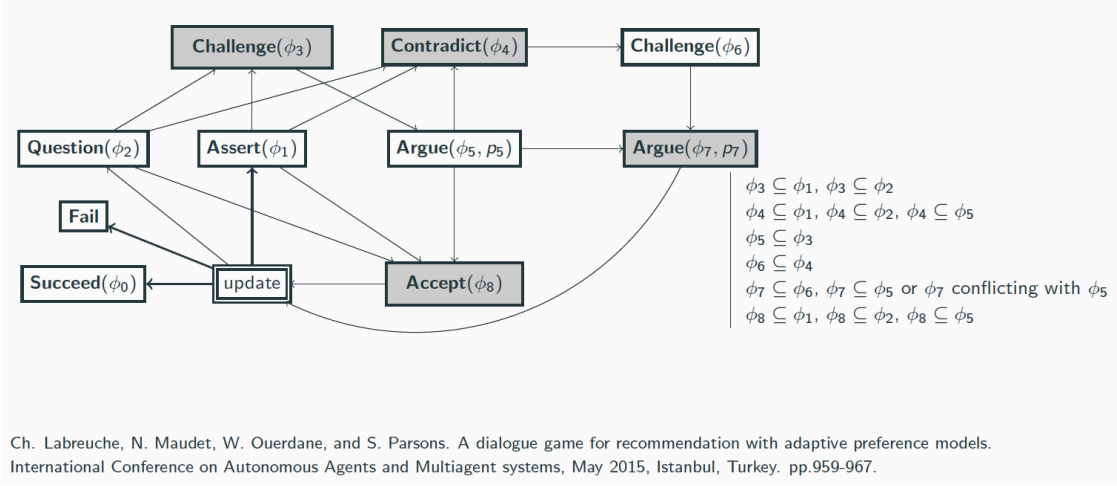
- **Rules for proposal validity**, which specify when a proposal is compliant with some conditions (e.g., an agent may not be allowed to make a proposal that has already been rejected);
- **Outcome determination rules**, which specify the outcome of the interaction; in argumentation-based frameworks, these rules might enforce some outcome based on the underlying theory of argumentation (e.g., if an agent cannot construct an argument against a request, it accepts it [Parsons et al., 1998]);

- **Termination rules**, that define the circumstances under which the dialogue ends.

Finaly, mainly two different ways to specify the dialogue (interaction) protocol:

1- *Finite state machines*: it represent an explicit specification of interaction. It is useful when the interaction involves a limited number of locutions.

Example



Ch. Labreuche, N. Maudet, W. Ouerdane, and S. Parsons. A dialogue game for recommendation with adaptive preference models. International Conference on Autonomous Agents and Multiagent systems, May 2015, Istanbul, Turkey. pp.959-967.

2- *Dialogue games*: by stating clearly the pre and post conditions of each locution as well as its e cts on agent's commitment. Its advantage is that it provids a clear and precise semantics of the dialogue

Example [McBurney et al. ,2003]

This locution allows a seller (or advisor) agent to announce that it (or another seller) is willing to sell a particular option .

- *Locution*: `willing_to_sell` (P_1, T, P_2, V), where P_1 is either an advisor or a seller, T is the set of participants, P_2 is a seller and V is a set of sales options.
- *Preconditions*: some participant P_3 must have previously uttered a locution `seek_info`(P_3, S, p) where $P \in S$ (the set of sellers), and the options in V satisfy constraint p.
- *Meaning*: the speaker P_1 indicates to audience T that agent P_2 is willing to supply the finite set $V = \{a, b, \dots\}$ of purchase options to any buyer in set T. Each of these options satisfy constraint p uttered as part of the prior math:*texttt{seek(.)}* locution.
- *Response*: none required.
- *Information store updates*: for each $\neg a \in V$, the 3-tuple $(T, P_2, \neg a)$ is inserted into $IS(P_1)$, the information store for agent P_1.
- *Commitment store updates*: no effects.

CHAPTER

THIRTYFOUR

6.3. OUR ARGUMENTATION-BASED NEGOTIATION PROTOCOL !

We are at the end of this practical work. The different questions and functions implemented till now will help you to run (very soon :-)) a negotiation between at least two agents. Before to push the run button, you need to check that you have implemented all the necessary components.

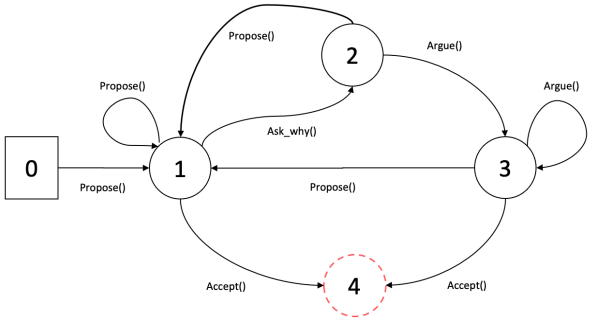
For this, the following algorithm synthesizes the conditions and the rules for the different moves during an interaction.

1 Intialization

- create n agents: $\{agent_i, \ldots, agent_n\}$
- set-up a list of objects: $\{o_1 \ldots, o_m\}$
- set-up a list of criteria: $\{c_1, \ldots, c_k\}$

foreach $i \in n$ **do**
  define preferences of i (criteria order, a performance table)
end
foreach $(X, Y) \in agents$ **do**
  X: **propose**(o_i) # here the item is selected randomly
  Y **receives-message(propose**(o_i)**):**
  if *($o_i \in$ 10% favorite items of Y)* **then**
    if *($o_i = o_j$ / o_j top item of Y)* **then**
      Y: accept(o_i)
      X: commit(o_i)
      Y: commit(o_i)
    end
    else
      Y: propose(o_j)
    end
  end
  else
    Y: ask_why(o_i)
  end
  X **receives-message(ask_why(**o_i**)):**
  if *(X has at least one argument pro o_i)* **then**
    X: argue(o_i, reasons)
  end
  else
    X propose(o_j) another item.
  end
  Y **receives-message(argue(**o_i**, reasons)):**
  if *(Y has counter-argument(o_i))* **then**
    if *reasons=($c_i = x$)* **then**
      switch *reply* **do**
        case *1* **do**
          Y: argue(not p, $c_j = y$ and $c_j \succ c_i$)
        end
        case *2* **do**
          Y: argue(not p, $c_i = y$ and y is worst than x)
        end
        otherwise do
          Y: argue(p', $c_i = y$ and y is better than x)
        end
      end
    end
    if *reasons=($c_i = x$ and $c_i \succ c_j$)* **then**
      switch *reply* **do**
        case *1* **do**
          Y: argue(not p, $c_j = y$ and $c_j \succ c_i$)
        end
        otherwise do
          Y: argue(p', $c_i = y$ and y is better than x)
        end
      end
    end
  end
  else
    Y: accept(o_i)
    X: commit(o_i)
    Y: commit(o_i)
  end
end

This algorithm is represented by the following transition state diagram and a possible output (interaction) is depicted in the figure below.




```
01: Agent1 - PROPOSE(ICED)
02: Agent2 - ASK_WHY(ICED)
03: Agent1 - ARGUE(ICED <= Cost=Very Good)
04: Agent2 - PROPOSE(E)
05: Agent1 - ASK_WHY(E)
06: Agent2 - ARGUE(E <= Environment Impact=Very Good)
07: Agent1 - ARGUE(not E <= Cost=Very Bad, Cost>Environment)
08: Agent2 - ARGUE(E <= Noise=Very Good, Noise > Cost)
09: Agent1 - ARGUE(not E <= Consumption=Very Bad)
10: Agent2 - ARGUE(notICED <= Noise=Very Bad, Noise> Cost)
11: Agent1 - ARGUE(ICED <= Durability=Very Good)
12: Agent2 - ARGUE(not ICED <= Environment =Very Bad, Environment >
Durability)
13: Agent2 - ARGUE(E <= Durability=Good)
13: Agent1 - ACCEPT(E)
14: Agent1 - COMMIT(E)
15: Agent2 - COMMIT(E)
```

The different perfomatives used in the algorithm are described in the previous sessions.

34.1 Questions

1- Read carefully the algorithm and update your implementation (when it is necessary) that it corresponds to the described functioning. Again this is just an example and you can choose different strategies/conditions. Feel free to make your own choices.

2- set-up a number of agents and lunch the negotiation processes between each pair of agents.

3- At the end of each negotiation:

- retrieve the winning agent, the one who spoke last before an ACCEPT or use the data structure storing the arguments to find this wining agent.
- retrieve the item(s) defended by this agent and the supporting arguments
- Make a small analysis for each wining agent: how much negotiation he has won, which item is the most defended, which criterion is the most put forward, and so on (you may choose your own statistics).

CHAPTER

THIRTYFIVE

6.4. TO GO FURTHER...

This section is for those everything runs well and the previous algorithm is completely implemented. It is based on the last section of the previous session, namely "Status of arguments".

1- From $n > 4$ (the number of agents), build the argument graphs such that: the nodes correspond to the winning argument(s) at the end of each negotiation and the edges are the attack relations between these arguments.

2- Calculate the set of **acceptable arguments** according to a given semantic of your choice. What conclusion can be deduced?

CHAPTER

THIRTYSIX

7. LAST MILES...

36.1 Agenda – Recall

1. Friday, March the 5th, 2021 : Introduction to MAS : definitions and implementation of a platform
2. Friday, March the 12th, 2021 : Multiagent simulation : preys and predators
3. Friday, March the 19th, 2021 : Interaction mechanisms : models and implementation
4. Friday, March the 26th, 2021 : Argumentation-based negotiation I: Practical session
5. Friday, April the 2nd, 2021 : Argumentation-based negotiation II: what is argumentation?
6. Friday, April the 9th, 2021 : Argumentation-based negotiation III: what is a negotiation protocol?
7. Friday, April the 16th, 2021 : Argumentation-based negotiation IV: Practical session

36.2 Session 7.

This session is entirely dedicated to finish the implementation and prepare the evaluation (see below).

36.3 For the evaluation !

- We expect that you write a small report (up to 5 pages) about your negotiation model. Give some printouts of negotiations. Compute some statistics on the negotiation outputs depending on the simulation parameters. You should clearly identify the parameters and the observed values.
- Prepare a zip of your code,
- The report and the code should be sumbitted on EDUNAO. **Deadline April 23rd, 2021 - midnight**.

CHAPTER

THIRTYSEVEN

INDICES AND TABLES

- genindex
- modindex
- search