UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

Integration of social aspects in a multi-agent platform running in a supercomputer

David Marín Gutiérrez

Bachelor Thesis
Department of Computer Science

Director: Javier Vázquez Salceda
Co-director: Dmitry Gnatyshak

Barcelona, 21 June 2022

Abstract

Agent-based modeling is one of the most suitable ways to simulate and analyze complex problems and simulations, such as the simulation of societal environments and scenarios. The kind of platform most commonly used in these endeavors is that of a multi-agent system.

Multi-agent systems are comprised of various actors (agents) in a concrete simulation environment, each of them possessing an individual knowledge and an individual behavior. These systems can be used to analyze collective emergent behavior in contexts such as sociology, economics, policy making, etc.

Current Multi-agent platforms either scale in computation quite well but implement very simple reasoning mechanisms, or employ complex reasoning systems at the expense of scalability. In recent work done at UPC, a platform enabling complex agents with HTN planning to scale and run parallelly was proposed, theorized, and implemented. This project extends said platform to enable a better analysis of the social relationships between agents by means of preferences over their objectives, preferences over their plans, actions, and moral values, while making sure our additions are scalable, to maintain the spirit and purpose of the platform.

In this work, we start from the previous work done by Dmitry Gnatyshak on implementing said platform, and we expand it, both formally and implementation-wise. We formalize the additions to the model of the system, as well as its modifications, and we do the same for the implementation. In the end, we provide a complex example scenario to showcase all the additions we have created.

Acknowledgements

In the first place, I would like to express my most sincere gratitude to the advisors of this work, Javier Vázquez Salceda, and Dmitry Gnatyshak. Their constant help, feedback, and guidance over the whole research, development, and writing process, has been priceless. Likewise, I would like to thank Sergio Álvarez Napagao for acting as an extra advisor to this project throughout all of its life cycle. His insights, recommendations, and help have been of enormous importance. Without these three people and their invaluable support, this project would not have succeeded.

Additionally, I would also like to thank Javier for helping me find and assigning me this project as my Bachelor's thesis, since I found myself a bit lost when it came to finding a suitable project.

Julian Padget has also my sincere gratitude for being the first person from outside the project to whom we presented it in depth, and gave us very positive and valuable feedback.

My thanks also go to the people at BSC who created it and maintain COMPSs, for this is perhaps the most useful and crucial component that this project employs.

Lastly, but not least, I would like to thank my family, my friends, and my partner, for their constant backing and support of everything I do in my life.

Contents

List of Figures

List of Tables

List of Acronyms

AI	Artificial Intelligence	15
ABM	agent-based modeling	1
BDI	Beliefs-Desires-Intentions	2
DAG	directed acyclic graph	35
GOAP	Goal-Oriented Agent Planning	22
HPAI BSC	High Performance Artificial Intelligence research group at Barcelona Supercomputing Center	3
HPC	high-performance computing	2
HTN	hierarchical task network	7
KEMLG	Knowledge Engineering and Machine Learning Group	3
MAS	multi-agent system	1
WLOG	without loss of generality	40
OOP	object-oriented programming	16

Chapter 1

Introduction

Agent-based modeling (ABM) is a computational approach for simulating the activities and interactions of autonomous agents (individual or collective entities such as organizations or groups) in order to better understand how a system behaves and how its results are determined. Furthermore, they allow for the simulation of complex environments where its perception, the decision-making processes, and the actions carried out are dispersed among several stakeholders or agents. The purpose of ABM is therefore to obtain explanatory insight into the behavior of a group of agents which share a common environment.

The fields in which ABM can be applied are numerous: biology, social sciences, ecology, economics, policy-making, etc. A specific application of ABM would be to analyze the social relationships between agents by means of norms, moral values, and social conventions, their adherence to those norms and values, how they affect and limit their actions, and how they may change over time as the agents interact with each other and the environment they are in.

One concept that is closely related to ABM is that of multi-agent system (MAS). Despite their similarities, a multi-agent system and an agent-based model are not always the same. An ABM's purpose is to find explanatory insight into the collective behavior of agents who follow basic rules, usually in natural systems, whereas the concept of MAS refers to the implementation of systems used to carry out simulations of environments featuring multiple agents. In summary, ABM is more related to studying the phenomenon of numerous agents co-existing in a shared environment, and MAS is more related to modeling these systems in a computational manner.

Many ABMs have been built using basic reactive agents[1]. Simulators like

[1]As we explain in section §2.1.1, **reactive agents** are the simplest type of agents: they simply react to changes in the environment.

this are crucial for understanding complex environments and exploring the fundamental principles of emergent social behavior. These models may be elevated to and examined at genuinely large scales thanks to recent advances in high-performance computing (HPC), assisting to find answers to critical scientific or social issues. Some well-known ABM systems worth mentioning are Repast[31], NETLOGO[27], and MASON[6].

One of the main downsides that these models present, is that many of them feature agents with very little reasoning capabilities, sometimes even reducing agents to mere rule-based or functional input-to-output transformers. There are some simulation scenarios in which these simplifications of the agents' reasoning suffices (e.g., if we want to simulate the general flow of traffic in a big city, it may be enough to have agents with simple behaviors who simply react to changes in the environment arround them), but there are other scenarios which require agents capable of more complex reasoning and behaviors.

One of the most widely used designs to model complex reasoning capabilities in agents is the Beliefs-Desires-Intentions (BDI) model[2]. In a typical BDI model, agents, which are goal-oriented, perceive their environment and gather some knowledge from it (represented as their **beliefs**) and then, based on it (and other factors as well), they decide what they want to achieve (in relation to their **desires** and **intentions**), and how to achieve it (typically through some means-ends reasoner). This is called the perceive-reason-actuate cycle, and there exist many MAS implementations whose agents feature this cycle, or some variation of it (for instance, Jadex[14], 2APL[8], Soar[15], and GOAL[12]).

The BDI model is more in line with the complex reasoning we described before. On the downside, however, these systems typically lack scalability. While they are great when it comes to simulating agents that exhibit complex behavior and reasoning, they fail at enabling simulations with larger numbers of agents, due to the need to explore multiple potential instantiations of abstract goals (which of all my goals are feasible/reachable now?) and plans (which plans are applicable now) in a given state of the system. Upon closer inspection of the BDI, one reaches the conclusion that it features a very prohibitive time complexity, and that any system designed to run sequentially will only be able to sport a very limited number of agents before displaying unacceptable execution times, as is the case with the four MASs cited in the above paragraph.

In 2019, one of my advisors, Dmitry Gnatyshak, took on a project to address this issue as his Master's thesis[10]. The problem he focused on solv-

[2]We provide a more detailed description of the BDI model in §2.1.4.

ing was that of the poor scalability in MASs that feature BDI-like agents. In collaboration with the Knowledge Engineering and Machine Learning Group (KEMLG) at UPC, and the High Performance Artificial Intelligence research group at Barcelona Supercomputing Center (HPAI BSC), he developed a custom Python-based BDI-agent simulation framework capable of both hosting agents imbued with complex reasoning capacity *and* running simulations with large numbers of these agents. The theoretical grounds for this framework were laid out in 2019 as well, and can be checked here [11]

The framework accomplishes this by using the COMPSs[1] to allow efficient and effective scalability on clusters. Put very briefly, he tackled the scalability problem by creating a framework capable of parallelizing the reasoning cycle of its agents, allowing them to run concurrently whenever possible.

In this work, we address the issue of further enhancing the platform implemented by Gnatyshak in 2019, by enhancing the agents' reasoning capabilities, as well as dotting them with social aspects such as preferences over their objectives, preferences over the actions they take in order to accomplish those objectives, values, making them aware of social norms.

Another equally important objective is carrying out an extensive exploration and research on the state of the art regarding MASs and microsimulation using BDI-like agents. This second goal will feed directly into the first described goal, as we will use the results of our research in order to implement and come up with concrete enhancements.

This chapter is structured as follows: first, we lay out the problem at hand in §1.1, then in §1.2 we give a description of how the platform worked before we enhanced it; in §1.3 we list and illustrate the objectives of this work, and finally in §1.4 we summarize the structure of the thesis.

1.1 Problem statement

In this work, we address the problem of enhancing the scalable BDI-based microsimulation platform created by Dmitry Gnatyshak[10] to provide its agents with more complex reasoning capabilities and social aspects such as norms, values, and preferences, as well as exploring the state of the art on MASs and microsimulation using BDI-like agents.

At the beginning of this project, we also set ourselves another objective, independent from that of bettering the agents. We also wanted to expand into the monitoring capabilities of the system. The platform has a controller that handles the running of the simulation: it manages the agents' messages, runs and updates the environment, and implements some basic monitoring functionalities as well. At the end, this goal was not finally addressed. This happened because, when we assiduously examined the state of the agents and their reasoning capabilities in the current implementation, we determined that enhancing them was much higher priority than increasing the monitoring capabilities, and we also envisaged that doing so would require a higher amount of work than we previously had thought and, therefore, we dropped the objective of improving the monitoring in order to divert all our resources and time into improving the overall state of the platform by enhancing the agents' reasoning capabilities, as well as their capacity to express desires and intentions.

The other main direction we thought of in order to improve the platform was to enhance its agents' reasoning capabilities. As stated before in the introduction to §1, the platform supports simulations sporting a high number of agents capable of somewhat complex reasoning running in parallel. We also wanted to improve the platform by enhancing the agents' reasoning capabilities, thus making display of even more complex reasoning in the ways they pursue goals, make choices and, in general, the ways they behave.

In §1.2, we will give a fully detailed description of every feature that was present in the system before we started adding improvements, and in §1.3 we will elaborate a complete list of the specific objectives that we have established in order to tackle the problems described in this section.

1.2 Description of the current implementation

In order to understand the state of the platform before this project took over it, several key concepts must be introduced. Firstly, we will shortly describe the COMPSs framework. We will not go as deep as our co-advisor did, since this project does not focus on parallelism. Then, we will explain the theoretical specification of the platform, and we will finish by describing how it was implemented.

Regarding COMPSs, aside from the official documentation, a more in-depth explanation of its inner workings is provided in [10] (pages 23–30). In the same document, detailed descriptions of the theoretical model (pages 30–39), and its previous implementation (pages 41–59), can also be found.

1.2.1 COMPSs

COMPSs[1] is a framework developed by BSC aimed at making parallel application creation and execution for distributed infrastructures easier. Its fundamental goal is to make it possible to create distributed cloud or grid-based apps without having to worry about the underlying systems that will run them. As a result, it provides a layer of abstraction, allowing the development of hardware-agnostic apps (which will then be automatically disseminated), saving developers the time and effort required to learn the low-level characteristics of the hardware in use. In simpler terms, it eases the task of programming code that will later be distributed and executed in parallel.

Its main runtime is written in Java, but COMPSs also has versions dedicated to dealing with programs written in C/C++ and Python. Since all the code produced in this thesis is Python code, in the next section we will briefly introduce the version of PyCOMPSs that works with Python programs.

1.2.1.1 PyCOMPSs

PyCOMPSs[26] is the version of the COMPSs framework created to work with programs written in Python, and the version of COMPSs that we will be making extensive use of in this project. Very briefly, its runtime features some extra layers that transform the Python data types into Java data types, as this is required in order to conduct the dependency analysis that are characteristic or parallel solutions.

1.2.2 Theoretical model of the platform

The formal model of our agent-based HPC simulation system will be presented in this section. We will mainly focus on showcasing its most relevant features and important definitions.

This formal model features a collection of elements that together build it, as well as sets of transition rules, for both agents and the model itself, that collectively define its operational semantics. Additionally, it also exhibits a concrete instantiation of its reasoning model, which we will also explain.

1.2.2.1 Elements and formal definitions

The **multi-agent system** $\mathcal{M}$ is defined as the following tuple:

$$\mathcal{M} = \{E, \mathcal{A}^+, \mathcal{C}\} \tag{1.1}$$

where:

- E is an **environment**, in which the agents reside, that they can perceive, gather information from, and act on
- $\mathcal{A}^+$ is a non-empty set of agents

- $\mathcal{C}$ is a **controller**, a structure that maintains the environment, handles agent-to-agent communication, and regulates how agents access and act on the environment

And individual **agent** $\mathcal{A}_i$ is defined as the following tuple:

$$\mathcal{A}_i = \{ID, msgQs, Bh, \mathbb{B}, \mathbb{G}, \mathcal{P}, outAcs\} \tag{1.2}$$

where:

- $ID = \{AgID, AgDesc\}$ is $\mathcal{A}_i$'s identity data:
  - $AgID$ is the unique identifier of $\mathcal{A}_i$
  - $AgDesc$ is an arbitrary description of $\mathcal{A}_i$
- $msgQs = \{\mathcal{I}, \mathcal{O}\}$ is the set of $\mathcal{A}_i$'s message queues
  - $\mathcal{I} = \{\dots, msg_i, \dots\}$ is the Inbox, the set of messages sent *to* $\mathcal{A}_i$
  - $\mathcal{O} = \{\dots, msg_i, \dots\}$ is the Outbox, the set of messages sent *by* $\mathcal{A}_i$
  - $msg_i = \{AgID_s, AgID_r, performative, content, priority\}$ is a **message** sent agent with $ID = AgID_s$ to the agent with $ID = AgID_r$, with the corresponding performative type, content, and priority. $performative$ are FIPA-compliant
- $Bh = \{MendR, \mathbb{RG}\}$ is 's **role behavior**
  - $MendR$ is the **means-ends reasoner** that $\mathcal{A}_i$ uses to generate plans
  - $\mathbb{RG}$ is the set of goals associated with the Bh which $\mathcal{A}_i$ is enacting
- $\mathbb{B}$ is the set of $\mathcal{A}_i$'s **beliefs**. It has the same internal structure than the environment E has
- $\mathbb{G}$ is the set of $\mathcal{A}_i$'s **goals**
- $\mathcal{P} = \{\dots, ab_i, \dots\}$ is $\mathcal{A}_i$'s current **plan**
  - $ab_i = \{\dots, a_{ij}, \dots\}$ is an **action block**, i.e., an ordered set of actions (each a_{ij} is an action). The system features three mutually exclusive types of actions:
    - ∗ **Internal actions**: these are the actions that are executed by the agent in order to change their beliefs

  * **External actions**: these are the actions that are sent by the agent to the controller in order to be executed on the environment, i.e., that are executed to alter the environment
  * **Message actions**: these are the actions that are used to generate messages intended to other agents

- $outAcs$ is the set of external actions to be executed on the environment. It is composed of tuples of the form:
  - $\{senderID, a^e\}$, where ID is the sender's ID, and a^e is the action that is being sent

Finally, the **controller** $\mathcal{C}$ is defined as the following tuple:

$$\mathcal{C} = \{\mathcal{I}, inAcs\} \tag{1.3}$$

where:

- $\mathcal{I}$ is the inbox for all the agents' outgoing messages
- $inAcs$ is the set of all the actions to be exercised on the environment

1.2.2.2 Means-ends reasoner

The implementation of the means-ends reasoner for the platform was originally a hierarchical task network (HTN) planner[13]. The reason to choose an HTN as the way for agents to reason was because it was shown in the document that HTN planners could fit well in BDI reasoning processes.

The HTN is a tree composed of three types of nodes: (i) Primitive Tasks, (ii) Methods, and (iii) Compound Tasks. Primitive Tasks are always leaves, while methods and compound tasks are always nodes with at least one child.

Primitive Tasks represent the concrete actions to be added to a plan and a way to decompose abstract tasks into concrete actions. They have an associated action block, which is simply a collection of actions that make up the Primitive Task, and possess a set of preconditions that help determine if the primitive task is available to be executed or not.

Compound Tasks are one way to represent abstract tasks. They are, put simply, an ordered set of Methods, and cannot have preconditions. On the other hand, Methods are the alternative way to represent abstract tasks. They retain a set of preconditions, and are an ordered set of either Primitive or Compound Tasks. The main difference between Compound Tasks and Methods is that the former are used to establish an '*or*' relationship between

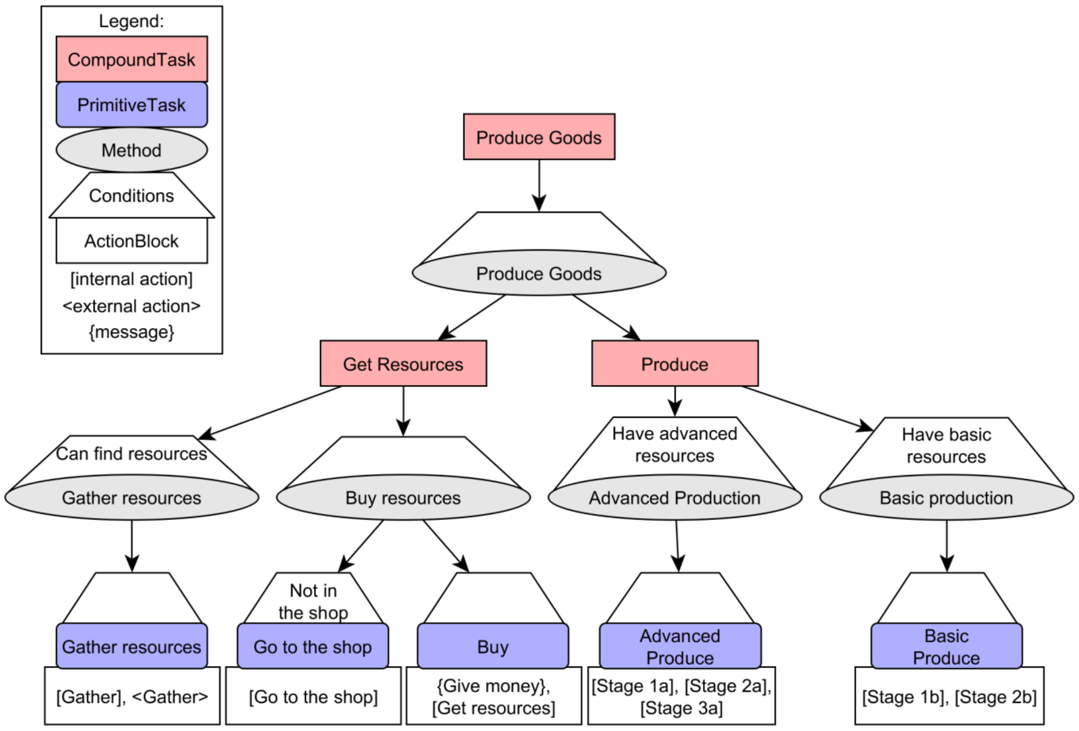


Figure 1.1: HNT example [10]

their children, while the latter are used to enact an '*and*' (more specifically, '*seq*') relationship between their children nodes.

An example of an HTN planner can be seen in Figure 1.1. The root is a Compound Task, which represents the goal (in this specific case, to 'produce goods'). As its only child, sits the Method 'produce goods', which has two children, 'get resources' and 'produce'. As mentioned before, the children of Method nodes must be completed following a '*seq*' relationship; therefore, this is interpreted in the HTN planner as: "first, get resources; then, produce goods". Now let us take a look at the children of the 'get resources' Compound Task. They are two Methods, one to 'gather resources', and another to 'buy resources'. As their names suggest, they are *alternative* ways of achieving the same subgoal and, as said before, we can see how the children nodes of Compound Tasks are governed by an '*or*' relationship, as only one must be completed. It is interpreted in the HTN planner as "in order to get resources, we can either gather them or buy them". Finally, two more details to cover: the first one is to notice that only Primitive Tasks and Methods have preconditions, and the second one is to look at how, as we previously stated, all the leaves are Primitive Tasks, and it is at them where the HTN 'ends'.

One may have noticed that the only situation where there are choices is when dealing with the children of Compound Tasks. The children of Compound Tasks are also always Methods. Therefore, the only part of the HTN where we can introduce alternative choices or preferences is when dealing with Methods (children of Compound Tasks).

Briefly, let us introduce how the HTN currently deals with this scenario. What it does is it picks the first available[3] method that it finds by iterating through the children of the Compound Task from left to right. Finally, let us advance that there is where we will modify the HTN in order to introduce preferences over plans, subplans, and actions.

1.2.3 Implementation of the platform

In this section we will briefly introduce how the previously described formal model of the platform has actually been implemented. We will provide a small and succinct description of every formal concept's implementation, as well as highlight any simplifications done to the model or any liberties taken that make it diverge from its theoretical declaration. Lastly, let us remember that all the code produced is Python code.

The totality of the MAS is organized as a package with numerous submodules and a list of main classes that support redefinition of functions by the user, and some methods are even left explicitly as trivial functions left for the user to implement their actual inner workings. The most important modules are:

- **Agent**: module which implements the concept of Agent through the class `Agent`. This class acts as a simple container for the description of an agent

- **Behavior**: contains the `Behavior` class, which is what governs all aspects of an agent's behavior: its deliberative cycle, its reasoning, what it does with the messages it receives, how it perceives the environment, when (and how) it replans, etc. Users might need to inherit from this class and reimplement some of its methods in order to allow for more complex behaviors.

- **Controller**: this module is perhaps one of the most important ones, along with Behavior. It contains the `Controller` class, which is solely responsible for running the simulation, handling the messages, updating the environment, hosting the agents, running PyCOMPSs tasks, etc.

[3]'Available' here means that its preconditions are satisfied

- **HTN**: contains the `HTNPlanner` class, which is the implementation of the HTN, as well as all the other related classes, such as `PrimitiveTask`, `Method`, and other auxiliary classes such as `Conditions` or `BeliefSet`
- **State**: this module contains the `State` class, which is a very basic structure that contains all the relevant information and status of an agent at a given moment in time: its current goal, its current plan, etc. It is separated from the agent and used to transfer and work with persistent states

Regarding goals, notice how they have not been mentioned so far. This is because one of the biggest simplifications that the implementation does with regards to its formal definition, is that agents do not have a set of goals. Instead, all they have is a plan with a root task acting as the agent's only goal. That is, not only does the system not support the declaration of multiple goals, it also has its only goal coupled to a plan (while, ideally, goals should be independent from plans in a BDI approach).

Another thing one might notice whilst inspecting the source code is that the `Conditions` class does not support disjunctions of statements, only conjunctions. That is, the preconditions of a task can only be of the form $\alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_n$, where each individual α_n is a logical statement that can be either true or false. This structure will be expanded in order for it to allow expressions such as $(\alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_n) \vee \ldots \vee (\beta_1 \wedge \beta_2 \wedge \ldots \wedge \beta_m)$.

Finally, one more thing we will mention is that, among all the available Python classes, users wishing to build their own simulations using our framework should only have direct access to, actively modify, and/or expand on the following: `ActionBlock`, `Behavior`, `BeliefSet`, `CompoundTask`, `Conditions`, `Controller`, `Effect`, `Method`, and `PrimitiveTask`. We, however, since not only do we intend to design and run simulations, but also to expand on the platform and its capabilities, will need to modify classes beyond the ones mentioned in this list, such as the `HNTPlanner` class, as well as creating our own classes.

This has just been a quick overview of how the model was implemented, where we have highlighted its main classes, as well as pointed at its bigger simplifications and details that we will attempt to polish. For a holistic, thorough, and comprehensive analysis and examination of all the platform's code and classes, please refer to [10], pages 41–59.

1.3 Objectives

The objectives of this work are the following:

1. Regarding **goals**:
   (a) Adding an explicit structure that encodes goals in agents
   (b) Totally decoupling goals from plans, i.e., making plans and goals totally independent
   (c) Adding the capacity for agents to have many goals, not just one

2. Adding **preferences over goals**. That is, adding a data structure that allows the designer to specify in which order the agent will pursue its goals. This structure should be based on widely used techniques, and most preferably, it should be in line with the state-of-the-art approaches of the field

3. Adding **preferences over plans and subplans**. That is, adding a data structure that allows the designer to specify preferences over multiple valid plans or subplans that achieve the same goal or subgoal. If goal g_i can be accomplished through any plan in the set $\{\mathcal{P}_1, \dots, \mathcal{P}_n\}$, then this structure (which should encode the agents' preferences over plans) should pick $\mathcal{P}^*$, that is, the plan that adheres the most to the agent's preferences

4. Adding **preferences over actions**. We should also devise another data structure that allows the designed to specify preferences over multiple applicable actions that help achieve the same plan or subplan. The same as in objective 2 is true for actions: if plan $\mathcal{P}_i$ or any subplan of it can be completed through any action block in the set $\{ab_1, \dots, ab_n\}$, then this structure should pick ab^*, that is, the action block that adheres the most to the agent's preferences over actions

5. Adding **values**. That is, adding a data structure that allows the designer to specify (moral) values in the agent. When the agent plans in order to achieve a goal, these values should be taken into account, and different agents with different values might come up with different plans for the same goal, even if all of the other contextual circumstances are the same

6. Make sure our additions are **scalable**. There is no doubt that, with all the new features we will be adding to the system, time-efficiency is bound to decrease. However, we should take every action and precaution at hand to ensure that the loss of time-efficiency is as small as possible, and does not affect the runtime of the system very much. Therefore, when programming and designing, we should always have

this goal in mind to ensure the scalability of the code. In other words, all the extra additions should not have a great impact on the scalability of the system and its overall performance.

7. Related to the objectives regarding preferences (objectives 2–4), one extra objective is to try to see **how 'far' we can go without using numbers** to encode preferences, since we have limited our design to not using them. The main reason why this goal was added is because we humans do not reason using hard numbers, at least explicitly. We do not think "today I prefer to go to the beach with a weight of 86, but to go to the cinema with a weight of 91; therefore, I will go to the cinema"; we, simply, think "today I would rather go to the cinema than to the beach; therefore, I will go to the cinema", that is, we think in qualitative terms, at least when we deliberate explicitly. However, all state-of-the-art approaches we have consulted, sooner or later, end up adding hard numbers to their implementations. For this reason, we have set this goal: we wish to see how *not* using numbers limits the expressiveness of our system, to what degree (how severe) this limitation is, and finally draw some conclusions as to whether it is acceptable to not use numbers or, on the contrary, we should use them if we wish to attain the desired level of complex reasoning for agents.

Initially, we also had the goal of enhancing the platform's monitoring capabilities. However, as stated in §1.1, early in the development of this project it was decided to drop this goal and to fully focus on making the agents' reasoning more complex, as the current state of the framework would require a lot of effort to achieve that objective alone.

In summary we have objectives to add goals to the platform (objective 1), to add social aspects (objectives 2–6), to test the quality of our implementation (objective 7), and finally, to see how not using numbers to encode preferences in a BDI approach can limit the expressiveness of the system.

1.4 Structure

This document is divided into 6 chapters and an appendix.

Chapter 1 is the introduction, which we are currently in. Here, we have established the purpose of this project, both in terms of the problem it attempts to solve and the 'gap" it tries to fill in the current state of the art. We have also provided a list and description of the goals we have set our minds out to accomplish, and finally, here we are providing a description of the thesis's structure.

Right after this introduction, in Chapter 2 we provide a study of numerous works that are related to our own. We analyze different MASs, both their formal definitions and their practical implementations. Additionally, we also explore the state of the art on all the features we wish to materialize into the platform. All the information, knowledge, and insights gained here will be used in further chapters when it comes to either formally designing features or implementing them.

Following that, in Chapter 3 we present all the proposed improvements we wish to add to the framework. For every feature we add, we specify whether it is a brand new addition to the model, or an enhancement of an already existing one. This chapter is limited to the formalization aspect, that is, here we will only describe our extensions in terms of formal definitions, transition rules, etc.

Right after laying down the theoretical grounds of our work, in Chapter 4 we provide all the technical aspects of the implementations of the features described and presented in Chapter 3. We introduce all the relevant changes done to the Python code, the newly created classes, the modifications done to the reasoning cycle, to the means-ends reasoner, etc. Additionally, and insomuch as one of the main focuses of this framework is to allow for time-efficient simulations of a high number of agents, we will also provide a complexity analysis of all the relevant code we have produced, and reason about how the newly added code alters the overall complexity of the whole system.

Posterior to the description of the implementation of the model and all the technical aspects related to it, in Chapter 5 we uncover the experiments we have devised in order to test our additions. For every experiment, we delineate its structure, its purpose, the expected results, the actual results, and we draw conclusions from those results.

Finally, in Chapter 6 we synopsize all the results and the knowledge we have obtained throughout the development of this project. We provide a thorough revision of our initial objectives, commenting on how each goal has been brought about and to what degree; thereafter, we summarize the main contributions of this work to the field of AI and MASs, and we conclude with an analysis of possible lines of future work and provide a list of several improvements that could be added to the framework.

In the appendix, we will find all the information related to project planning (budget, methodology, tasks, etc.), as well as some extra examples from the tests we perform. Appendix A will cover project planning, Appendix B will contain the budget and sustainability analysis of this work, and Appendix C will contain extra examples for our tests.

Chapter 2

Related work

In the previous chapter, we provided context for this work in the field of Artificial Intelligence (AI), and stated the main problem we will be attempting to solve. We also presented the framework we will be expanding, its main elements, and its current state. Additionally, we stated the main objectives of this work, and briefly explained the general structure to which this document will adhere.

In this chapter, we provide a thorough analysis of the state of the art in all the important areas related to this project, as well as provide justifications for which works influenced our design decisions later on, and for the ones that did not. This chapter is of the utmost importance, since it is from here we will be drawing inspiration and knowledge to later be used in the accomplishment of the objectives laid out in Chapter 1.

This chapter is structured as follows: we begin in §2.1 with an introduction to the main concepts in the field of multi-agent systems (such as agents, the BDI model and so on). Then, in §2.2, we cover our research into goals, their representation, their implementation, and how plans are traced to achieve goals in various real-world platforms; we continue with our research into the state of the art of social aspects in §2.3, where we will cover different modern approaches on representing and simulating agents that possess preferences over their objectives, their plans, and the actions that make up their plans, as well as the representation of values. Finally, in §2.4 we provide a summary of the main findings we have obtained from the research into the state of the art.

2.1 Main Multi-agent Systems' concepts

One of the most essential works in the field of multi-agent systems is Wooldridge's "An Introduction to MultiAgent Systems"[30]. In his work, Wooldridge concentrates on the definition of what an agent is, the properties it should have, different types of agents, how they should interact, etc. Other aspects of agents and MAS are also extensively covered in another crucial work on the field: Russell and Norvig's "Artificial Intelligence: A Modern Approach"[23].

Additional, it has been indispensable to the enrichment of my knowledge on MASs that Javier, this thesis's advisor, allowed me to attend his classes on Multiagent Systems Design during the development of this project. A great amount of the knowledge used to develop this project comes from me attending his classes, and they have been a general influence on this thesis, since I have had to understand, design, and implement many concepts taught in the classes.

In this section, we will define the most relevant and key concepts in the field of MAS. We will provide thorough and precise definitions according to the cited bibliography, as well as illustrate with the aid of examples wherever it may be necessary or helpful. The key concepts we will cover are: agents, multi-agent systems, the BDI-model, the concept of environment, and planning.

2.1.1 Agents

An agent is a computational system that is capable of independent action, usually on behalf of its user or owner. This is one of the most common definitions of an agent, and is the one given by Wooldridge in his aforementioned work.

Agents differ from other systems such as objects from object-oriented programming (OOP) in that their actions are autonomous: they do not need to be told what to do at every given moment. Instead, they are given goals or directives, a set of actions they can perform, and a knowledge base they can use to reason, and they independently make use of all these tools in order to figure out what actions to carry out to satisfy the objectives they have been given. Agents can be embodied (i.e., they have a physical body that exists in the real world with sensors to perceive the environment and actuators to perform actions) or totally virtual (all perceptions refer to digilat elements, and actions have effects in a igital environment). An agent can be both embodied (connected to the real world) and virtual (connected to a digital environment).

An example of a very simple embodied agent is smart thermostats: the user specifies an ideal temperature and the thermostat will try to keep the room around said temperature. Notice how the user does not need to tell the thermostat to lower or raise the temperature, it will instead have sensors and a set of internal rules that it will use to decide when to turn on the heating and when to turn it off, and at what level of power it should be working at. All the user does is give the thermostat a goal (e.g., "keep the room at around 22 ºC"), and the agent decides on their own the best means to fulfill that goal (always within the frame of the actions they know how to perform and the rules they have in their knowledge base). More instances of embodied agents could be a 4.0 industry production plant, or the more classic example: the robot. The clearest example of a virtual (non-embodied) agent would be a chat-bot or an investment fund management bot (because they do not exist in the physical world, but they can perform very real actions nonetheless). Finally, an example of an agent both embodied and virtual would be Amazon's Alexa, or any other smart home assistant, as is capable of both acting in the physical world (e.g., opening windows, turning lights on and off) and in the digital one (e.g., buying a movie to watch at home tonight).

Over the years, many properties of agents have been proposed and discussed by experts on the field. Some have proven to be fundamental over time (e.g., reactivity or autonomy), while others have been deemed impractical for complex, real-life applications. The above-cited source names a set of properties that agents should have:

- **Autonomy**: Agents need to be able to act on their own without users instructing every step of their actions. Autonomy is thus the ability of acting independently, exhibiting a certain level of control over their internal state
- **Reactivity**: Agents need to respond to changes in their environment. Reactive agents therefore maintain an ongoing monitoring of the environment they are in, and react to (meaningful) changes that may occur in it
- **Proactiveness**: Agents need to strive towards meeting their objective(s). Agents therefore are not limited to reacting to stimuli from the environment (reactivity), they also should have goals and need to attempt to achieve them and find ways to succeed. The scope of proactiveness then covers the acts of generating intermediate goals to meet final ones, recognising opportunities, etc.
- **Social ability**: Agents need to interact with each other, or humans. This includes exchanging information, cooperating, etc

- **Flexibility**: An agent is said to possess flexibility when it is reactive, proactive, and social (see above)

Other properties worth mentioning that agents also need to have are, for instance, **rationality** (all the actions of an agent should have the purpose of bringing them closer to achieving their goals) and **reasoning capabilities** (an agent should be able to reason about their environment, their goals, and actions, and plan possible courses of action).

Some extra minor properties worth looking into are: veracity, benevolence, and learning/adaptation. These are by no means mandatory, and should only be added to agents when they are necessary. For instance, there are many situations in which we may not need an agent to be able to learn from past experiences, and thus granting them the ability to do so would be ultimately useless.

Finally, the cited bibliography proposes 3 main architectures for agents: reactive, deliberative, and hybrid. **Reactive agents** are the simplest type of agents: they simply react to changes in the environment (i.e., they respond to stimuli in a functional manner), as they only possess the autonomy, reactivity, and social ability properties. Contrariwise, **deliberative agents**, rather than reacting to meaningful changes in their environment, they model it and perform comprehensive planning. And then, there are **hybrid agents**, which are a middle-ground approach: they strive to maintain a balance between reactivity and deliberation, two parameters which are very often in opposition to one another.

2.1.2 Multi-agent system

Briefly, a multi-agent system is a system in which a number of agents coexist and interact with each other and the environment. Typically, they all exist in the same environment and have the capabilities to interact, socialize, and even cooperate with one another, as they all perform actions that affect their shared scenario and each other.

2.1.3 Environment

This far, throughout this document, we have been using the word 'environment' without providing a formal definition for it. In the topic of intelligent agents, their environment is the physical or virtual context in which the agents exist. Normally, the most important properties of an environment are that it is **partially accessible** (the agents can obtain some, but not all, information about it through their sensors) and **dynamic** (it can be altered through the agents' actions, its own internal processes, time, etc.), although there are other properties as well.

Most implementations of environments describe the world as a collection of relevant[1] variables (normally referred to as 'names'), and their current values. Thus, in most cases, an environment is formally a set of the form:

$$E = \{(n_1, v_1), (n_2, v_2), \ldots, (n_n, v_n)\}$$

where each n_i is a *unique* name for a variable that is relevant to the description of the world, and each v_i is the value that n_i has at that moment.

As an example, a concrete instance of an environment could be:

$$E = \{(is\_raining, True), (day, Monday), (temperature, 12°C)\}$$

and a possible interpretation of it would be: "the current day is a Monday, it is raining, and it is 12 degrees Celsius in this environment".

In most of the MAS implementations (that we will see in §2.2), environments are represented through the structure commonly referred to as *dictionary* or hashmap in most programming languages. It makes sense to use a dictionary to represent the concept of environment introduced in the previous paragraph, as it provides an easy way to store values that are paired with unique keys (the names), and an efficient way to access those values by indexing them through the keys.

2.1.4 The BDI model

As we mentioned in Chapter 1, in a BDI approach, agents perceive their environment and gather some knowledge from it (represented as their **beliefs**) and then, based on it (and other factors as well), they decide what they want to achieve (in relation to their **desires** and **intentions**), and how to achieve it (typically through some means-ends reasoner). This concept is called the BDI cycle, and every agent that is part of a simulation performs it in a loop-like manner at every iteration or step.

We provide a both more formal and more complete version of the concept defined above in Algorithm 1. Its most important parts are:

- B are the agent's beliefs. They represent the knowledge the agent has about the environment. In most implementations, an agent's beliefs are usually myopic, and not necessary fully in tune with the actual state of the environment.

[1]What is 'relevant' to describe the world is most of the times determined by hand by the designer of the system.

- D are the agent's desires. They represent the motivational state of the agent: objectives or situations that the agent would like to bring about, but not ones that it has necessarily committed to achieve.

- I are the agent's intentions. They represent the deliberative state of the agent: objectives or situations that the agent has committed to bring about. Summed up, intentions are desires that the agent has committed to achieve.

- π is the agent's current plan. Typically, plans are an ordered set of actions that an agent can perform. If performed in the correct order, the intention that the agent was trying to accomplish, should be attained, assuming no extra changes in the environment occur due to other reasons or other actors.

- $brf(\dots)$ is the agent's belief revision function. It takes the agent's current beliefs (B) and the latest perception of the world (ρ), and it updates the beliefs accordingly. It is, in most scenarios, user-defined. In general, all the following functions are typically user-defined, as they need to adapt to their current context and the specific situation that is being simulated.

- $options(\dots)$ is the agent's function to obtain and renew its desires (options). It takes the current beliefs (B) and intentions (I) as input, and it determines a set of desirable states of the world that the agent may or may not immediately commit to.

- $filter(\dots)$ is the agent's function to obtain and renew its intentions. It takes its current beliefs (B), desires (D) and intentions (I) as input, and it returns a new set of intentions that have been revised according to the input. Typically, what this function does is choose from, among the set of desires, one desire to commit to achieve in the short term or to actively focus on.

- $plan(\dots)$ is the agent's function to draw plans. It takes its current beliefs (B) and intentions (I) as input, and it returns an ordered set of action that, according to the agent's knowledge (beliefs), should achieve its goals (intentions).

- $reconsider(\dots)$ is a useful function that allows the agent to reconsider its current desires and intentions according to its updated beliefs. This function is run right after the agent has executed an action. The execution of said action may have drastically altered the agent's knowledge of the world, and this new knowledge might (or might not) trigger a need in the agent to reconsider its desires and intentions. This is akin to when in real life we carry out an action the consequences of which makes us realize that we were taking the wrong approach and/or that we had wrong knowledge or assumptions, and then we are forced to take some step backs and re-think our approach.

- *sound*(...) is a function that determines whether the current plan π is sound or not. We say that a plan π to achieve intentions I is *sound* according to some beliefs B, if after fully executing π, we would have achieved I. To draw some real-life parallels, this is akin to when we try to determine whether a plan we have in our heads will succeed (or not) at accomplishing our goals and, obviously, we infer from our current knowledge of the world (beliefs). What this function does is, taking those elements as inputs, try to determine if the current plan is sound.

Algorithm 1 BDI cycle, from [5]

```
$B := B_0$;                                            ▷ Initial beliefs
$I := I_0$;                                            ▷ Initial intentions
while true do
    get next percept $\rho$;
    $B := brf(B, \rho)$;                               ▷ Belief revision function
    $D := options(B, I)$;                              ▷ Option generation function
    $I := filter(B, D, I)$;                            ▷ Alternative filtering beliefs
    $\pi := plan(B, I)$;                               ▷ Means-ends reasoner
    while not(empty($\pi$) or $succeeded(I, B)$ or $impossible(I, B)$) do
        $\alpha := hd(\pi)$;                           ▷ Next action in plan $\pi$
        $execute(\alpha)$;                             ▷ Executing the next action
        $\pi := tail(\pi)$; ▷ The current plan is updated (last executed action is popped)
        get next percept $\rho$;
        $B := brf(B, \rho)$;     ▷ Update beliefs to account for the action just executed
        if $reconsider(I, B)$ then                     ▷ Reconsideration function
            $D := options(B, I)$;
            $I := filter(B, D, I)$;
        if not $sound(\pi, I, B)$ then                 ▷ Plan soundness function
            $\pi := plan(B, I)$;
```

Most MASs that are modeled after the BDI model dot adhere to it fully.

Instead, they use it as the main inspiration source, drawing many of its elements from it, but ultimately end up differing in some place or another. This is because the BDI approach is a theoretical model, and some implementations may have extra requirements that cannot be met by fully adhering to the BDI (e.g., the BDI cycle as defined in Algorithm 1 is very computationally expensive, and some implementations may simplify it in order to strive for balance between BDI-likeness and efficiency).

The most typical ways in which the majority of MASs stray away from the theoretical BDI model are simplifying one or more of its elements (e.g., not distinguishing desires from intentions, limiting the expressiveness of desires, etc.) or simplifying the BDI cycle (e.g., skipping some functions like $reconsider(\dots)$ or $sound(\dots)$, or making their structure and return values very trivial and easy to compute). Even the MASs designed with the intention of being as faithful to the theoretical BDI model as possible are not totally compliant with it, due to one more of the reasons given in the previous paragraph.

2.2 Related research on agents' goals and plans

When it came to research the concept of goals, and reasoning using goals, the model of goals and priorities that we have developed has been inspired by two agent frameworks with working implementations: GOAP and BDI4JADE.

2.2.1 Goals and Plans in GOAP

The work of Jeff Orkin on the Goal-Oriented Agent Planning (GOAP) model [18] has proven fundamental to this thesis's general progress, and has also been the main inspiration source for our model of goals[2].

GOAP is the AI created for the enemies of the video game F.E.A.R, which was mainly formalized and created by Orkin. Its agents follow a BDI approach, and behave according to the BDI cycle. In this section, we provide a summary of GOAP's main elements, as well as dive into some independent users' adaptations of Orkin's GOAP.

Summarized, the way Orkin represents goals in GOAP is by specifying a **desired state of the world** that agents strive to achieve. Agents can have many independent goals, but they can only pursue one at the same time. The main paper on GOAP can be found here [21], although Orkin did write on the topic of agent planning[20] and representation of the world for

[2]Our proposed Goal model and all other modifications introduced to the platform's model are described in detail in Chapter 3.

agents[19] in the past, and he uses this previous work in the implementation of the aforementioned AI of F.E.A.R[21].

Agents can sense the world through sensors that can be either polling or event-based. They have a belief base that follows the same structure used to describe the world, separate from it. Beliefs (called attributes) have a confidence value [0, 1] with different meanings, depending on the variable, and they can be inaccurate (not consistent with the true state of the world).

The state of the world and beliefs are both represented by a conjunction of literals, and an assignment of values to these literals, which are basically names of variables that, put all together, represent a current state of the world.

To build plans, agents use Actions. Actions have a name, a list of preconditions, and a list of effects. They can only be executed if all its preconditions are true, and change the world (as well as the beliefs of agents who perceive the action being executed) by making its effects become true. They have a cost associated to them, which is there to basically guide plan creation, which we will cover later in this chapter. The preconditions and effects of Actions can have procedural conditions and effects, meaning that they are not limited to some hard-coded assignment of values, but rather they can have a function that, depending on the current context, computes the necessary value of the effects or the preconditions.

As said before in this section, goals describe a state of the world we want to achieve. This state of the world is described using the same syntax to describe the current state of the world, an agent's beliefs, actions' effects, etc. Agents can have many goals, and each goal has a *dynamic* priority, that can be altered due to external events or internal actions, changes in the world, stimuli, etc. Finally, only one goal is pursued at a time (this would be akin to the Intentions concept of the theoretical BDI model, while non-active goals would be Desires).

In order to plan, an agent must have: a set of available actions, a set of beliefs about the world and sensors to periodically update those beliefs, and a set of goals. Each goal has a current priority, and the agent will choose to plan for the goal with the highest current priority. These are numeric priorities (i.e., a quantitative relation, not a qualitative one). The generated plan will be the cheapest cost plan (in terms of cost of actions). Finally, agents use A* to plan. The heuristics used is trying to minimize the weighted number of actions used to reach the desired state, i.e., minimize the sum of costs of the actions in the plan.

The selected plan for a goal will always be the cheapest available plan for that goal. Replanning can happen because of:

- Another goal has become more prioritary than the current goal (replanning for a new goal)
- The current goal has been achieved (and the next goal is selected)
- The current plan has failed but there exists another more costly plan for the same goal (replanning for the same goal)
- The current plan has failed and the agent can't find a plan for the current goal (replans for a new, different goal instead)

A plan is only to satisfy one goal purposefully (i.e., it does not support plans to satisfy multiple goals). However, it could be the case that, by pure chance, an agent's plan achieves two or more goals than once. This would be purely coincidental and it should not be taken as the agent planning more efficiently. Finally, note that replanning does not take into account the cost of plans, only the priorities of goals, that is, if we have two goals, g_i and g_j, with priorities 51 and 52, respectively, and the cheapest plan for g_i has a cost of 1 (cheapest possible plan), while the cheapest plan for g_j has a cost of, say, 10^6 (or any arbitrarily large cost), the agent would still select g_j over g_i, only because 52>51.

We also reviewed some independent users' implementations of Orkin's GOAP: GPGOAP[25], and ReGOAP[9].

In the case of GPGOAP the main differences are that GPGOAP is a very stripped down and simplified version of GOAP. It has no concept of beliefs, agents simply perceive the environment all the time. Additionally, the world can only be described by boolean atoms, that is, variables that are either True or False, and there is a maximum of 64 variables that can represent the world at the same time. These atoms are also used to specify pre- and postconditions (effects) of actions, which can also feature a cost, just as in the original GOAP. Goals are directly specified to the agent in the code, defined as a desired worldstate, using a subset of atoms that describe the world, along with their desired values. They are decoupled from plans, but they do not have a priority or importance field, as they did in the original GOAP. Only one goal can be defined for an agent, and it is the only goal that the agent will try to achieve. Therefore, this implies that there is no reevaluation of goals and no replanning due to a sudden change of goals, and no goal selection either, because there is only one. Finally, it uses A* to plan, with the same heuristic and general strategy as GOAP.

In the case of ReGOAP, it is a more fleshed-out implementation of GOAP, but still has some limitations. Represents the memory of the agent (beliefs) as a conjunction of literals and their values, stored in a dictionary. As opposed to GPGOAP, agents can have inaccurate or myopic views of the world, being both wrong and incomplete about its variables and their values, which they can partially sense through some sensor function. The state of the world is represented using the same structure as the beliefs of the agents. Agents also have actions. Each action is defined by its name, and a list of preconditions and effects. Both lists are also described using the same structure used for beliefs, and they can be compared against an agent's beliefs or the current state of the world, etc. Agents can have many goals (represented as a desired state of the world), and goals have different priorities, but only one may be active at a time. Always picks the most prioritary goal, and it replans whenever a goal becomes more prioritary or when it achieves the committed goal. It uses A* to plan, just as GOAP.

2.2.2 Goals and Plans in BDI4JADE

Another source of inspiration for our model of goals was Ingrid Nunes's BDI platform called BDI4JADE[17][16]. BDI4JADE, as it name suggests, is a BDI layer on top of JADE[2], an agent development framework whose agents lack the typical BDI abstractions and the reasoning cycle featured by other similar models. It was the purpose of BDI4JADE's authors to, among other things, enhance JADE's agents with BDI abstractions and the theoretical BDI cycle.

After a careful analysis of BDI4JADE, we concluded that it uses the same structure as Orkin's GOAP to represent goals (desired state of the world), which made us commit harder to that design decision, as we saw it was a very popular design decision, since it was present in almost any MAS we checked. However, we should also highlight that BDI4JADE supports the declaration of different types of goals, among which there are 'belief goals' (goals that deal with states of the world described through boolean variables), 'beliefset value goals' (same as before, but now the variables are continuous or have more than two possible values), 'composite goals' (used to represent goals composed of subgoals which have to be achieved sequentially or in parallel), etc. It also differentiates between desires (non-committed goals) and intentions (committed goals).

Plans are an ordered set of actions and are executed with the aim of achieving a specific goal. The reasoning cycle functions this way:

1. Beliefs are updated

2. Finished goals are removed. So far, this is new. Before, in all the other systems that we have checked, goals were either achieved or not. Here,

they can also be removed. This allows BDI4JADE to specify goals other than maintaining a state of the world always being true.

3. Options are generated. In this step, the goals available to the agent are determined (its desires). It can generate new desired goals, determine that existing goals are no longer desired, or keep existing goals that are still desired.

4. Removal of dropped goals: when a goal, or set of goals, is determined as no longer desired in the previous step, it is removed from the set of goals of the agent in this next step

5. Deliberating goals: in this step, the agent's current goals are partitioned into two subsets: goals to be tried to be achieved (intentions), and goals to not be tried to be achieved (desires)

6. Updating goals' status: based on the partition performed in previous step, the status of the goals are updated. Selected goals are updated to the status of trying to achieve (committed), and not selected goals are updated to the status of waiting. When a goal has the status of trying to achieve, the agent will select plans for achieving that goal, and the goal has become an intention

The main inspiration we took from BDI4JADE, however, was not its model of goals, but its approach on how agents build and/or choose their plans. Up until now, all the models we had looked into used a similar strategy for their agents to build plans: the agents had a set of actions with an associated set of preconditions and effects, and they use these actions, along with a heuristic search algorithm, in order to build a plan (an ordered sequence of actions) that can achieve their desired goal (desired state of the world). For instance, Orkin's GOAP uses A* for this purposes. What sets BDI4JADE apart from these approaches, is that its agents do not posses a set of actions that they can use to build plans, but rather, they have a library of plans that the agents can choose from. Each plan in the library has some applicability conditions (equivalent to actions' preconditions), and agents choose directly from a set of available plans that the designer added to the library.

There are three main reasons to why we choose this approach instead of building plans using a search algorithm. Firstly, it is more in line with the state of the art on agent design: the majority of papers we would go on to read after researching BDI4JADE, which happen to be more recent than GOAP, all feature examples of agents having a static library of plans, instead of having an amorphous set of actions they can use to build concrete plans. These papers are the ones cited in §2.3. Secondly, it is more in line with the spirit of our original platform, as we use an HTN planner as a means-ends reasoner, and choosing to use a heuristic search algorithm would mean having to discard the already-implemented HTN. And finally, it gives the designer more control over the way in which the agents behave, as he/she can directly trace the shape of one plan the agents may end up using, instead of giving them an unordered set of actions that they have to figure out how to sort in order to achieve their goals.

2.3 Related research on social aspects

After the research on goals, plans, and plan-formation, we moved on to one of the main aspects and purposes of this project: the integration of social aspects in the MAS. After careful consideration, we determined that, first of all, we should start by providing agents with preferences. So far, our agents now have goals, and plans decoupled from those goals, but still lack any integrated way to specify preferences over them, and that is what we intend to implement next: a preference-based system allowing the designer to specify preferences over both goals and plans (and sub-plans and actions).

2.3.1 Research: preferences over goals

Our main inspiration for the implementation of preferences over goals comes from CP-nets[4]. Although our actual implementation is definitely not an implementation of a CP-net, the main inspirations we have drawn from them is to establish one default and many conditional preorder relationships over goals, and building a graph to both visualize them and interpret them. We also had looked into the approach that Dignum et al. take in [7] to model values and adapting it to model preferences over goals, but upon closer inspection, we decided against modeling our implementation based on it, since it uses a lot of numerical values under the hood, and we decided to favor more qualitative approaches over quantitative ones.

Other papers which we also consulted and that either make use of CP-nets or that employ the idea of having conditional preferences based on some trigger conditions are [22] and [28].

The theoretical details of our implementations of preferences over goals will be discussed in Chapter 3, and the practical aspects in Chapter 4.

2.3.2 Research: preferences over plans and actions

Moving on to preferences over plans, we drew a great deal of inspiration from the paper "Preference-based reasoning in BDI agent systems" by Visser et al.[29]. Here, they introduce the concepts of goals' properties, which we use extensively in our modeling of priorities over plans. We also make use of their mechanism for the propagation of properties in our implementation. We should note that our implementation is simpler than theirs, since we decided to apply only a subset of the concepts defined in the paper. For instance, the paper defines both properties of goals (discrete values that a property can take) and resources of goals (numerical values and intervals that represent how much of a resource (e.g., money, food) is being consumed by a goal or a sub-goal), but we have chosen to simplify the approach and add only properties of goals, as we have determined that, for the expressiveness purposes we had in mind, those are enough.

Let us briefly introduce the concepts of **properties of goals** and their **propagation**. These concepts will be explained and linked in detail in Chapter 3, as we will be using them in our formal model, but seeing as they are the core of the paper cited in this section, we feel it would be natural to first introduce them here.

A property of a goal (or plan, or action) is the representation of one relevant[3] aspect of reality which is of particular interest in describing *how* a goal is achieved (or *how* a plan is achieved, or *what* implications executing an action has). For example, a plan to go to school using a car could have the property 'transport' equal to 'car', while an alternative plan to go there on foot could have its property 'transport' equal to 'walk'. These properties can be used to express preferences on how to achieve goals and can help us set up a system where an agent can reason with preferences and choose among equally valid plans using preferences. For example, if an agent prefers going to school by car, then this preference could be easily encoded in any appropriate data structure and, whenever possible, the agent will use the car over walking to go to school.

And then, the concept of propagation of properties is very straightforward. In short, if a goal can be achieved through *alternative* plans, then the properties of said plans are propagated 'upwards' to the goal in an '*or*' manner. Following our example from before, the goal to go to school would have the property 'transport' equal to a structure that implies that it can take either the value 'car' or the value 'walking', for example, a set. Likewise, if a plan can be achieved through *sequential* subgoals or actions, then

[3]Again, what is 'relevant' to describe reality is most of the times determined by hand by the designer of the system.

it will have the union of the properties of the children as its own properties. For instance, if we have plan π_i with subgoals $g_{i,1}$ and $g_{i,2}$ as children, with $g_{i,1}$ having the single property $p_1 = \{v_1\}$ and $g_{i,2}$ having the single property $p_2 = \{v_1, v_2\}$, then plan π_i will have *both* p_1 and p_2 as its properties, each with their corresponding possible values.

On top of that, the paper ultimately gives preferences some numerical weights as well, and we have decided to keep the system qualitative and use the order of declaration of the preferences as some form of implicit hierarchy. Nonetheless, our implementation (that will be discussed in Chapters 3 and 4), is still very faithful to the one devised in the cited paper.

2.3.3 Research: values

The research we have conducted on values also mainly came from the paper "No Pizza for You: Value-based Plan Selection in BDI Agents" [7]. Here, they implement something akin to the concept of properties of goals we mentioned before, but using moral values instead. Therefore, values can be modeled by taking the same approach as 'hard' preferences over plans and actions. Consider the previously defined example of the goal to go school by either car or on foot. Just as we defined 'hard', objective properties over each of the goal's plans, we can also ingrain moral values into each plan. For example, we could give the plan to walk the property 'environmentalism' with the value 'high', and give the plan to use the car the property 'environmentalism' with the value 'poor' or 'low'. Just to give an extra example, we could give the action of stealing (supposing it formed part of a plan) the property 'evil' with the value 'very', and so on.

2.4 State of the art summary

Throughout this chapter, we have discussed numerous research papers and books on the related topics. We started giving an introduction to multi-agent systems and defining some core concepts to our project, and we continued with our research into how goals are represented in state-of-the-art approaches, as well as how goals and plans are related in those approaches; then, we moved on to our analysis of social aspects, be it preferences over goals, plans, or values.We have also highlighted how most BDI-like MASs implementations simplify the theoretical BDI model and the reasons as to why it happens.

The main contribution of the analysis of the related research to our project have been some relevant ideas that have inspired the improvements on the model (that we will discuss in Chapter 4). On one hand, from GOAP we get the conception of goals as desired world states, an agent possibly having many goals at the same time but only pursuing one at a time. On the other hand, from BDI4JADE we get inspiration on its plan selection strategy. CP-nets are the main inspiration for our goal preference strategy. And Visser's propagation of goal properties has shaped our way to introduce preferences on plans.

When it comes to which 'gap' in the state of the art we are 'filling' with this work, the answer is the same for this project than it is for its predecessor[10]: we are creating (in our case, improving) a BDI-like MAS populated with complex agents that is able to scale and perform simulations with a high number of those agents, through the use of parallelism and HPC.

Chapter 3

Adding Goals and Preferences to the Multi-agent Platform

In the previous chapter, we reviewed some relevant works and papers related to our research into the state of the art. As we mention in \$2.4, the analysis of this works has provided inspiration on how we can add goals and preferences to the agent platform.

In this chapter, we explore, one by one, all the improvements and additions we will introduce in the system, from a theoretical point of view. To do so, we will revisit the contents of §1.2.2, and modify the formal definitions of the applicable elements to represent and cover all the additions we have inserted in the model. Throughout all sections of this chapter, we will be making extensive use of the research we conducted and explain in Chapter 2 and, in general, we will always talk in terms of things we *add*, *subtract*, or *modify* from the elements described in §1.2.2.

This chapter is structured as follows: first, in §3.1, we describe everything related to the addition of goals: definition of the concept of goal, set of goals, their properties, decoupling goals from plans, how plans will be traced in order to achieve goals, etc. Then, we follow with the addition of preferences over goals in §3.2, where we will explain its theoretical definition, its elements, how it works, etc. The same explanation, but for preferences over plans and actions, will be introduced in §3.3. Moving on, in §3.4 we will briefly explain how our additions also support the expression of moral values. Lastly, this chapter will be concluded by providing a list of limitations stemming from our theoretical model in §3.5.

3.1 Addition of goals and a library of plans

Let us revisit the appropriate element regarding goals from the original formal model. Recall that an individual agent $\mathcal{A}_i$ was defined as the tuple:

$$\mathcal{A}_i = \{ID, msgQs, Bh, \mathbb{B}, \mathbb{G}, \mathcal{P}, outAcs\} \tag{3.1}$$

The only element we are concerning ourselves now, however, is $\mathbb{G}$, the set of goals. Formally, in the original formal model it was declared that an agent has a set of goals, but goals were left undefined. Therefore there were no notions of *what* a goal is, *how* it is defined, or how they are *related with plans*. In this section, we will cover all these aspects and explain how they have been integrated into the formal model.

We have chosen to model goals as desired states of the world, that agents strive to achieve. It is equivalent to the concept of **desires** in BDI. A goal is therefore defined by a collection of subsets of the variables that describe a state of the world (its **conditions**), and an assertion of their desired value(s). These conditions are expressions such as 'cash==10' or 'speed>=50' to mean that having exactly 10 units of cash and that maintaining a speed of 50 or above are part of the desired state of the world, respectively. Each subset describes a conjunction of variables that describe a desired state of the world, and in order for a goal to be considered achieved it is required that all the variables of at least *one* of these subset have the desired values in the eyes of the agent (their **beliefs**).

Let us formally define the structure of $\mathbb{G}$, the set of goals just described. It is an unordered set of goals of the form:

$$\mathbb{G} = \{g_1, g_2, \ldots, g_n\} \tag{3.2}$$

where each g_i is an individual goal among the many goals an agent has. Simultaneously, a goal is defined by:

$$g_i = \{name, descr, \mathbb{C}, status\} \tag{3.3}$$

where:

- *name* is a **unique** identifier of the goal
- *descr* is an optional text describing the goal
- $\mathbb{C}$ is the set of conditions for the goal to be considered achieved
- *status* is a boolean value that is True $\iff$ the conditions $\mathbb{C}$ are satisfied according to the agent's current beliefs $\mathbb{B}$

The set of conditions of a goal deserves a more thorough definition. It is therefore formally defined as unordered collections of assertions over the state of the world (the **environment**) of the form:

$$\mathbb{C} = \{a_1, a_2, \ldots, a_n\} \tag{3.4}$$

where:

- $a_i = \{n_1 \star v_1, n_2 \star v_2, \ldots, n_m \star v_m\}$ is a conjunction of statements over the values of variables of the agent's beliefs, defined by:
  - n_i, which is the *unique* name of a variable of the agent's beliefs
  - $\star$, which is any of the following binary operators:

    $$\{=, \neq, >, \geq, <, \leq\}$$

  - v_i, which is the value of interest that is being asserted to n_i

Let us explicitly note that the agent possesses the capabilities to check whether or not an individual goal has been achieved according to its beliefs: $check\_goal(g_i, \mathbb{B})$ outputs True if, according to the agent's beliefs, the conditions of the goal have been met, and false otherwise.

Other important aspect of our implementation is that our agents are allowed to have multiple goals, but are restricted to pursuing only one at a time (this *commitment* to a goal that is intended to be pursued is equivalent to the concept of **intention** in BDI). They have the capability to re-consider which goal they want to pursue, and may change the goal they are committed to even if they have not achieved their current one, depending on their current beliefs and the state of the world they perceive. This single, committed goal is expressed by adding an extra element to the definition of the agent, which now looks like:

$$\mathcal{A}_i = \{ID, msgQs, Bh, \mathbb{B}, \mathbb{G}, \mathcal{P}, outAcs, g_c\} \tag{3.5}$$

where $g_c \in \mathbb{G}$ (the current *committed* goal) is simply a goal selected among the non-achieved goals contained in $\mathbb{G}$.

Now that we have carefully defined what goals are and their properties, let us move on to doing the same thing for the **library of plans** and how plans for goals are traced.

Before we took on this project, as we mentioned in §1.2.3, the definition of agents included only a single plan in them, and no explicit way to specify different plans for the same goal or to pick a totally different plan from the current one. Also, since we could only have a single plan, it meant that agents, in practice, only had a single goal as well. We have addressed and fixed that part of the issue with our definition of goals.

Now, we will define a new element that will act as a library of plans, and a means for the agent to pick from those plans and relate them with goals. As stated in §2.2, we decided to take the approach of having a library of already defined plans that the agent can pick for, and these plans will be related to goals by means of the structure of the **metaplanner**. Let us formally introduce it in the definition of the agent:

$$\mathcal{A}_i = \{ID, msgQs, Bh, \mathbb{B}, \mathbb{G}, \mathcal{P}, outAcs, g_c, \mathcal{MP}\} \tag{3.6}$$

The metaplanner $\mathcal{MP}$ is, a library of plans for each goal. Formally, it can be viewed as a matching relationship from goals towards plans. It is a structure that, given a single goal as input, returns all the plans that can be used to achieve said goal:

$$\mathcal{MP} : \mathbb{G} \longrightarrow \mathbb{P}^* \tag{3.7}$$

where $\mathbb{P}$ is the set of plans $\mathcal{P}$ associated with goal g_i and $\mathbb{P}^*$ is used to indicate that it can output tuples of plans of arbitrary cardinality (meaning one specific goal may have, for instance, three plans associated to it, while a different goal might have five, or two). So we can better understand the concept, one can see that, programmatically, the object that resembles a metaplanner the most is a dictionary where the keys are goals and the values associated to those keys are sets of plans.

Before moving on to explaining more about the metaplanner, we need first to make a small addition to plans. Now, all plans have applicability conditions, as opposed to before, where we only had a single plan (therefore, it always applied). As it now stands, every plan has a some applicability conditions that must be true before it can be picked.

Other noteworthy aspects of the metaplanner are that it will incorporate appropriate functions for plan selection. Therefore, it will not simply act as a library/collection of plans, but it will also perform part of the reasoning. This reasoning includes both checking which of the associated plans are available for application, as well as ordering them based on the preferences[1].

For the first functionality, the metaplanner will feature a function of the form $get\_available\_plans(g_i, \mathbb{B})$ which, taking into account the current beliefs of the agent, will output a subset of the set of plans associated with the goal, containing only all plans that are applicable. For the second functionality, the metaplanner will have a function of the form $pick\_plans(g_i, \mathbb{B}, prefs_{\mathcal{P}})$, where $prefs_{\mathcal{P}}$ are the agent's preferences over plans, that will pick the plan that is

[1]We describe how we model preferences over plans in §3.3.

more adequate to the current situation according to the agent's preferences and beliefs, from among all the applicable plans.

One alternative approach that we considered but did not follow was to, instead of employing a library of plans, building plans directly from actions using heuristic search. We, however, made the decision to keep the HTN and to implement the concept of having a library of plans instead of using heuristic search to build plans as an ordered sequence of actions, and in section §2.2 we justified our choice.

So far, in this section we have formally defined goals, the set of goals, and their most useful properties and features that can be used. Additionally, we have expanded the definition of plans, and introduced the concept of metaplanner as a library of plans, as well as its most relevant features and properties. And throughout all, we have been updating the formal definition of the concept of agent, carefully adding its new elements at every step of the way.

3.2 Addition of preferences over goals

Now that we have defined a specific concept of goals, set of goals, and how to associate plans with goals, we move on to specifying preferences over goals. As we cover in §2.3.1, we drew inspiration from CP-nets and conditional preference formulas, to some extent, but we simplified the approach in order to be able to work without scalars, that is, having a fully qualitative approach to specifying preferences over goals.

In order to encode preferences over goals in our agents, we have added the following element, $\mathbb{P}_g$ (which stands for "$\underline{\mathbb{P}}$references over $\underline{\text{g}}$oals") to their definition, making it now be of the form:

$$\mathcal{A}_i = \{ID, msgQs, Bh, \mathbb{B}, \mathbb{G}, \mathcal{P}, outAcs, g_c, \mathcal{MP}, \mathbb{P}_g\} \tag{3.8}$$

where $\mathbb{P}_g = \{dGP, cGP_1, cGP_2, \ldots, cGP_n\}$:

- dGP are the *default* preferences over goals. That is, under 'normal' circumstances, these are the preferences that apply.

- cGP_i are *conditional* preferences over goals. That is, they have some trigger set of conditions $\mathbb{C}_i$ of the same form of the conditions defined in Definition 3.4 (§3.1).

The dGP and each cGP_i are all the same type of element, a directed acyclic graph (DAG) that corresponds directly to a **strict partial order** relationship between goals, and the only difference between them is that the

dGP is the one active by default (does not need any conditions to be met), while the various cGP_i become active and replace dGP if some associated conditions are true. The way in which the conditional preferences replace the default preferences will be explained later in this section. Now, let us first focus on the DAGs and the relationships they characterize.

A binary relation R over a set X is called a strict partial order if it is *irreflexive*, *transitive* and *asymmetric*. In other words, it has to satisfy:

1. **Irreflexivity**: $a \not R a$, for all $a \in X$
2. **Transitivity**: $(a \; R \; b) \wedge (b \; R \; c) \Rightarrow a \; R \; c$ for all $a, b, c \in X$
3. **Asymmetry**: $(a \; R \; b) \Rightarrow (b \not R a)$ for all $a, b \in X$

Although asymmetry is implied by the conjunction of irreflexivity and transitivity, it is nice to enumerate it in order to better understand the nature of the strict partial order relation (sometimes called a *strict preorder* as well). A nice property of strict preorders is that they have always a unique DAG associated to them.

Please take note of the word 'partial', which implies that not every pair of elements is necessarily comparable. That is to say, there may be pairs of elements where neither element comes first.

To define preferences over a set of goals, the approach we have taken is to establish a strict partial order relation between them to indicate which goals must be pursued before trying to achieve other goals. To model the context-dependent nature of preferences, we allow the declaration of conditional preferences, which are also a strict preorder relation over goals, but they only apply when their trigger conditions are met.

Once all the strict preorder relations have been established, we deduce their associated DAGs. From those DAGs, we compute a valid topological ordering of each, and this orders are the ones in which goals will be pursued by the agents (the first non-achieved goal in the topological ordering will be the agent's current goal). Let us visualize it better with an example.

Consider the following context:

- We have one agent: $\mathcal{A}_\prime$
- Agent $\mathcal{A}_\prime$ has the following goals $\mathbb{G}_0 = \{g_0, g_1, g_2\}$. For clarity's sake, let us consider that g_0 is a goal to tidy the agent's bedroom, g_1 is a goal to tidy the agent's kitchen, and g_2 is a goal to store clothes that are hanging out to dry in the open.
- Let us denote "goal i must be achieved before goal j" by $g_i \rightarrow g_j$.

- With this, we define the default preferences over goals of the agent: $\{g_0 \rightarrow g_2, g_1 \rightarrow g_2\}$, that is, before going to store the clothes that are outside, the agent must have cleaned both his bedroom and his kitchen. Notice how both g_0 and g_1 must be accomplished before focusing on g_2, but there is no established order between g_0 and g_1, because it is a strict *partial* order.

- Let us also define one set of conditional preferences over goals: $\{g_2 \rightarrow g_0, g_1 \rightarrow g_0\}$ with the associated trigger conditions that the variable 'raining' must be True. That is, if it is raining, the agent's most priority goal will be to collect the clothes, and then either cleaning his kitchen or his bedroom, again, in no specific order.

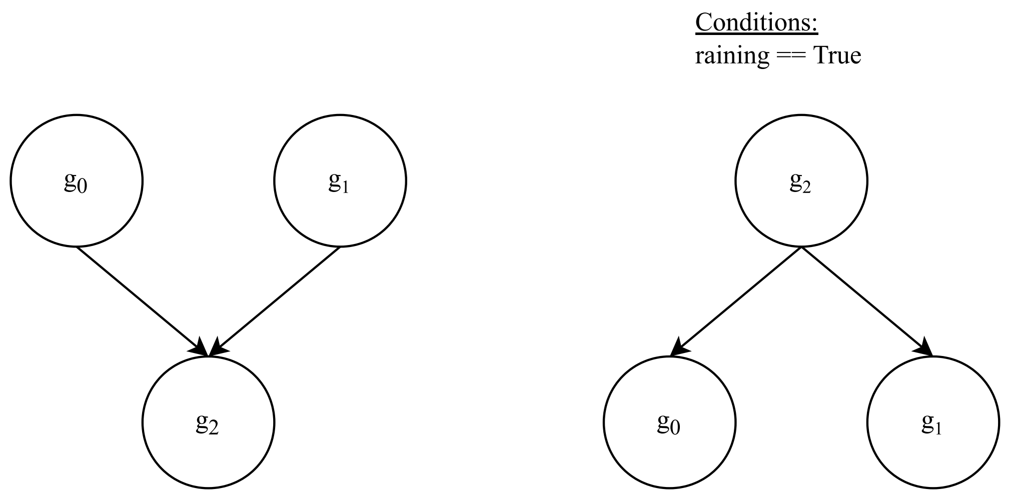


Figure 3.1: Example of default and conditional preferences over goals [Own making]

From this example's context, we draw the illustration of the example that can be checked in Figure 3.1. The left graph is the one deduced from the relation that defined the default preferences over goals, while the graph on the right-hand side is the one defined by the trigger conditions described in the example's context. A valid topological ordering of the left graph might be: g_0, g_1, g_2, but also g_1, g_0, g_2. By default, the agent will pursue his goals in either of those orders, but the moment it starts to rain, he will switch to any of the topological orderings that can be given to the right graph, for instance, g_2, g_1, g_0. The concepts of this example are extendable to having many conditional preferences.

What to do when two (or more) conditional preferences can be picked at the same time will be explained in §4.2.2, but let us advance here that it is a choice that will be left for the designer ultimately, and we provide a default approach (that can be overridden and replaced by the designer at any time).

3.3 Addition of preferences over plans and actions

We now understand how preferences over goals can be specified, that is, we know how to provide agents with the capacity to choose *what* to pursue. But we also need to provide them with means to have preferences over *how* to achieve what they are pursuing. At the end of the day, even if your goal is to eat, it is not the same to achieve that goal by eating a delicious pizza or to achieve it by eating a boring, plain white rice, even if both actions achieve the goal all the same.

We humans have preferences not only over *what* goals we want to achieve, but also over *how* we want to achieve them. Some people might prefer to walk to their workplace, while some others would rather drive there; a specific individual might prefer to have fun by going to the beach, but will resort to reading a nice book if the weather turns bad, etc. These examples provide us with further, key information: the preferences we have over how we achieve things are also context-dependent; we may wish achieve a specific goal by means of some actions under some circumstances, but under different circumstances we might prefer to achieve the same goal through different actions. Since the purpose of this work is to imbue agents with human-like social aspects for simulation purposes, we will need to take all these considerations into account when modeling preferences over plans and actions.

In order to encode preferences over plans and actions in our agents, we have added the following element, $\mathbb{P}_p$ (which stands for "$\underline{\mathbb{P}}$references over $\underline{\textbf{p}}$lans") to their definition, making it now be of the form:

$$\mathcal{A}_i = \{ID, msgQs, Bh, \mathbb{B}, \mathbb{G}, \mathcal{P}, outAcs, g_c, \mathcal{MP}, \mathbb{P}_g, \mathbb{P}_p\} \tag{3.9}$$

where $\mathbb{P}_p = \{gP_1, gP_2, \ldots, gP_n\}$ will have preferences defined over the plans of each goal. We denote the preferences over plans for goal g_i by $gP_i = \{dPP, cPP_1, cPP_2, \ldots, cPP_n\}$, where:

- dPP are the *default* preferences over plans for goal g_i. That is, under 'normal' circumstances, these are the preferences that apply to all plans for that goal.

- cPP_i are *conditional* preferences over plans for goal g_i. That is, they have some trigger set of condi¡tions $\mathbb{C}_i$ of the same form of the conditions defined in Definition 3.4 (§3.1).

The dPP and each cPP_i are all the same type of element, the only difference being that dPP is applied by default and a cPP_i is applied if their trigger conditions are true. Notice that the structure of each gP_i is the same as the preference over goals ($\mathbb{P}_g$). However, as we need to specify a whole set of preferences for each goal (e.g., one might prefer to go to work on foot, but go to their second residence by car), we need to encapsulate them all in a single set, $\mathbb{P}_p$.

Let us explain the element dPP, which will also explain the structure behind each cPP_i, given that they are the same. Recall the concept of **properties** of a goal, and the concept of **propagation** of these properties, both introduced in §2.3.2. In order to give each agent preferences over plans, the basic actions of these plans must be populated with properties. Also recall that when we defined the structure of the means-ends reasoner in §1.2.2.2, we mentioned that it had Compound Tasks, Methods, and Primitive Tasks.

Here, we make use of the Primitive Tasks and the concept of properties and, for each plan, we attach an arbitrary number of properties to every Primitive Task of the Plan. These properties are then propagated 'upwards' to the non-leaf nodes, and finally, agents sort the available plans and actions according to how much they are compatible with their preferences. Please note that these preferences are 'soft' restrictions, that is, they are used to give priority to one plan over another, never to *discard* plans or actions. Therefore, we might encounter one situation where an agent carries out an action that goes against all of its preferences and values because no other actions were available. In human behavior, for example, we might one day have to eat our least favorite food in an scenario where it is the *only* food available and it is our current goal to eat something.

In the following subsections (§3.3.1 to §3.3.4), we compliment the definitions of properties and propagation made in Chapter 2, formalize their structure, how they are used, and give an example to better explain the concepts.

3.3.1 Properties of goals

A property of a goal is the name of a variable of interest that a goal has the capacity to alter. Said variable does not necessarily have to be the name of a variable in the set of beliefs of an agent. It is simply something noteworthy that achieving a goal has the capacity to give a specific set of values.

For example, if a goal is to 'cook dinner', some of the properties might be 'vegetarian' and 'cuisine', and their possibles values might be {True, False} and {'French', 'Italian', 'Spanish', 'Turkish'}, respectively. These properties are always set on the *actions* of plans, and are propagated to subplans and subgoals until they reach the root goal, using the process that we will define in the §3.3.2.

Each goal, plan, subplan, and action will have a set of properties PS, of the form:

$$PS = \{prop_1, prop_2, \dots, prop_n\} \tag{3.10}$$

And each property $prop_i$ is of the form:

$$prop_i = \{v_1, v_2, \dots, v_n\} \tag{3.11}$$

where:

- $prop_i$ is the *unique* name/identifier of the property
- v_i is one of the possible values that the property can take. These values can be boolean, numeric, etc., depending on the nature of the property itself

The set of values that make up each property are used to indicate possible values the property can take. All properties can have the special *None* value inside the set of their possible values. The presence of this value in a property of a plan or subplan indicates that said plan or subplan can be achieved through one or more actions that do not use or alter the property in question at all.

3.3.2 Propagation of properties

Briefly explained, propagation of properties consists in sending the properties 'upwards' from the most concrete actions, up to the root goal, passing through every subplan and subgoal in the way. This happens in a very intuitive way: without loss of generality (WLOG), let us consider two *sequential* actions that have the same parent. The parent's set of properties will be the result of computing the union between the two children's properties. Notice that each child will not have different possible values for the same properties, since they are sequential actions, and it would not make sense to design a plan in which child action no. 1 sets 'cuisine'='Spanish' only for the child action no. 2 to set the cuisine to be 'French'. Therefore, the properties of the two (sequential) children will always be different, and the resulting properties of the parent node will simply be the joining of the children's sets of properties, and it is trivial to see that this process applies to n sequential children actions.

Now, WLOG, consider two *alternative* actions that have the same parent. The parent's set of properties will be the result of merging the properties of the children in the following manner: if both children set different values for the same property then, for the father, the values of the property will be the union of the values that the children had (e.g., if child no. 1 had 'cuisine'='Spanish' and child no. 2 had 'cuisine'='French', the parent task will have 'cuisine'={'Spanish', 'French'} to indicate that if that node is chosen, we will limit the possible values of 'cuisine' to those two values). And if either child has a property that the other does not, the parent will simply take the same properties of the child that has it, and will add the special value $None$, to indicate that if that node is chosen, there is a path of the plan that accomplishes the goal without ever giving a value to that property.

3.3.3 Selection of plans and actions using properties

Now that we know what properties of goals are, how they are set, and how they are propagated, let us explain how an agent uses them and specifies preferences over his plans for a goal. Given a concrete goal g_i an agent has a set of preferences over the plans to achieve g_i. We defined this set earlier as $gP_i = \{dPP, cPP_1, cPP_2, \dots, cPP_n\}$, where every element is either the default set of preferences, or a conditional set of preferences. The dPP and each cPP_i are all an *ordered* instantiation of the values of different properties of the goal's plans. Besides being ordered, it has the exact same structure as the set of properties described in Definition 3.10. In fact, mathematically speaking, the dPP and each cPP_i are all subsets of their goal's property set.

As an example, let us consider the goal of ordering food delivery. A possible instance of dPP can be $dPP = \{cuisine = \{Chinese, Italian\}, veggie = \{True\}, local = \{True\}\}$. We will use this example to explain how an agent would choose plans and actions to order food, assuming that the current preferences are dPP. When the agent is planning, and it finds itself having to choose among multiple, *alternative* subplans or actions, it will always put first the ones who have their property 'cuisine' equal to either 'Chinese' or 'Italian'. If there are no subplans or actions satisfying these conditions, then it will do the same for the second property, and so on. If, however, there were, it will choose, among those, from the ones that have their 'veggie' property equal to True, and so on with the third property, etc. If the special value $None$ is found, it will be treated as satisfying the property.

In summary, it tries to find all plans which satisfy the first property, and among those, the ones that satisfy the second property, and so on, essentially making the first property the most important and, in general terms, making the i^{th} property more important than all its following properties put together.

The way in which conditional preferences over plans and actions are activated and replace the default one is identical to the case preferences over goals. Once again, we have favored a qualitative system over a quantitative one, for the same reasons than we did when implementing preferences over goals, but just as it was the case with preferences over goals, the designer can redefine this process.

3.3.4 Example of properties, propagation, and application of properties

Let us provide an example that is an expansion of the previously described situation. Our goal is to order food from a restaurant in town. Let us assume that we have three plans for that: a plan to order burgers, a plan to order falafel, and a plan to order pizza. Let us also assume that there is only a local burger, a local falafel, and both a local pizza restaurant and a big company that makes pizza. Other assumptions that we take are that all burgers and pizzas are non-vegan, and that all falafels are vegetarian. Let us first see what the designer would have produced, in Figure 3.2.

Notice how *only* the actions have properties. This exemplifies how properties are declared and encoded in plans, beginning from actions. Now, let us show the result of property propagation in Figure 3.3. It is the same collection of plans as in Figure 3.2, but now all vertices have their own set of properties that have propagated upwards, from the leaves (actions). Notice how, in general, all properties have propagated towards the upward nodes. However, most of these propagations have been very simple ones: from single child to parent, although there are two cases worth looking at.

The first one is the propagation from the subplans to order local pizza and order from big pizza company. Notice how their properties are the same in all fields except for the ‘local’ field, with one holding it as True, and the other as False. However, these two *alternative* subplans share a common parent, and when their properties are propagated to it, they are merged in the way we described earlier. Now, the parent has its property ‘local’ with *all* of its children values, to represent that, if that subgoal (or its parent subplan) is picked, then we can still order from either a local restaurant or a big chain. The other note-worthy example is the propagation of properties to the root node, where all options have been compiled in its properties.

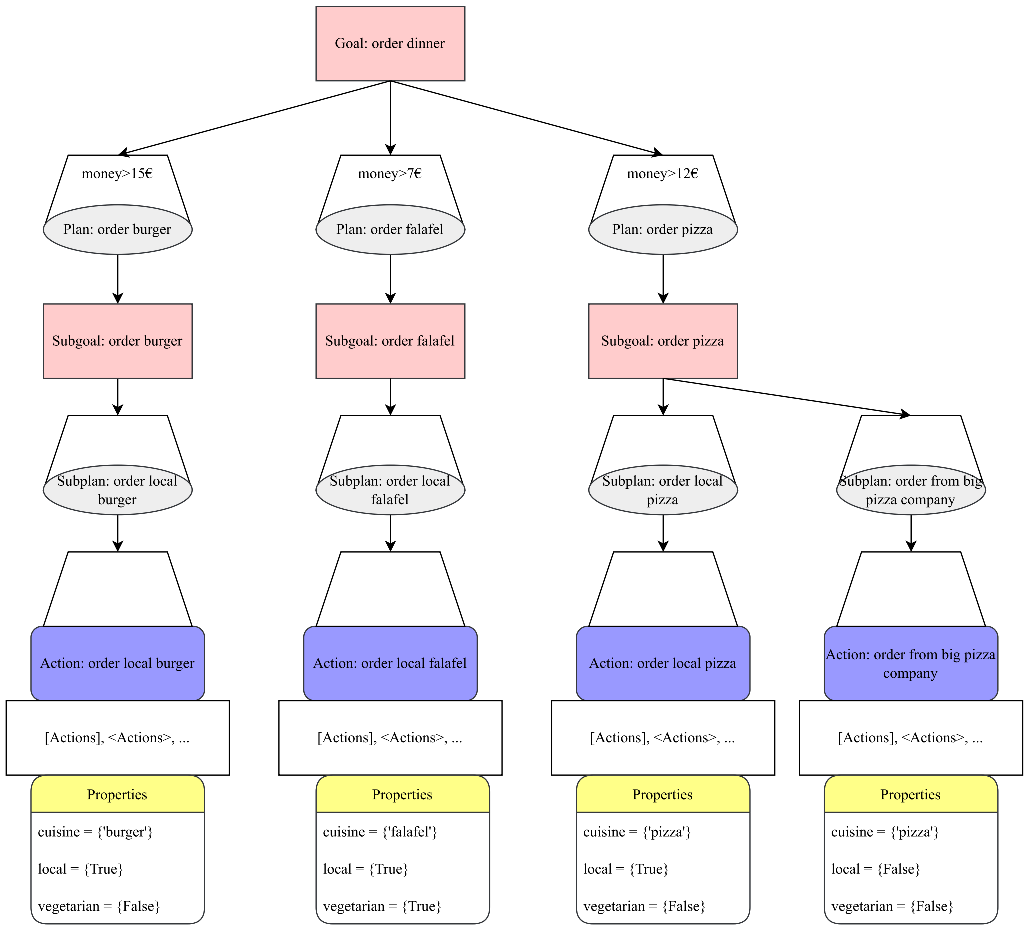


Figure 3.2: Example of the initial status of a set of plans to order food [Own making]

This is, however, a very small example provided to understand the basics of propagation of properties. A much more complex example is provided here [29] (page 9, Figure 2).

Now, let us complete this example by showing the process of choosing a plan taking preferences into account. An assumption we make throughout this whole example is that all plans are available, that is, our choices are not restricted by the environment in any way, shape, or form.

Let us assume that we have the following preferences over how to achieve the goal to order dinner:

1. $\{cuisine = \{falafel\}\}$

2. $\{cuisine = \{falafel, pizza\}\}$

3. $\{cuisine = \{falafel, pizza\}, local = \{True\}\}$

4. $\{local = \{False\}, vegetarian = \{True\}\}$

5. $\{vegetarian = \{True\}, local = \{False\}\}$

6. $\{local = \{False\}, vegetarian = \{False\}, cuisine = \{burger\}\}$

In case no. 1, the agent would simply order from the falafel restaurant. In case no. 2, the agent would order either falafel or from any pizza restaurant. In case no. 3, it would either order from the falafel, or from the local pizza place. Cases no. 4 and 5 help us exemplify the importance of the order in preferences over plans. In case no. 4, the agent would order from the big pizza company, despite it being not vegetarian. This is because the preference over the value of property 'local' goes before the preference over the value of property 'vegetarian', and therefore trumps it. In order to specify that we want to order from a non-local, vegetarian place (if applicable), with emphasis on the food being vegetarian, we should do it like case no. 5. Finally, case no 6 helps us further exemplify this phenomenon: even if we wanted to order non-vegetarian burgers, we would still get pizza from the big company, due to the first property being that 'local' has to be false.

In general, one can think of it in the following way: the agent picks from all the plans that satisfy the leftmost property, then, from those plans, it picks from those that satisfy the next leftmost property, etc. Again, as said before, the process for picking between subplans, subgoals, and actions using properties and preferences over those properties is the same. Finally, remark that this process is how it works for both default and conditionally triggered preferences, as they have the same structure, the only difference being that the latter need to be activated in order to take over and replace the default properties.

3.4 Addition of values

As mentioned in §2.3.3, we note that moral values can be simulated using the above described system of preferences over plans and actions. Consider the previous example of ordering food. Just as we defined 'hard', objective properties over each of the goal's plans (the cuisine, whether it is vegetarian or not etc.), we can also ingrain moral values into each plan. For instance, we could associate an action to steal with the property 'evil' equal to 'very', and so on.

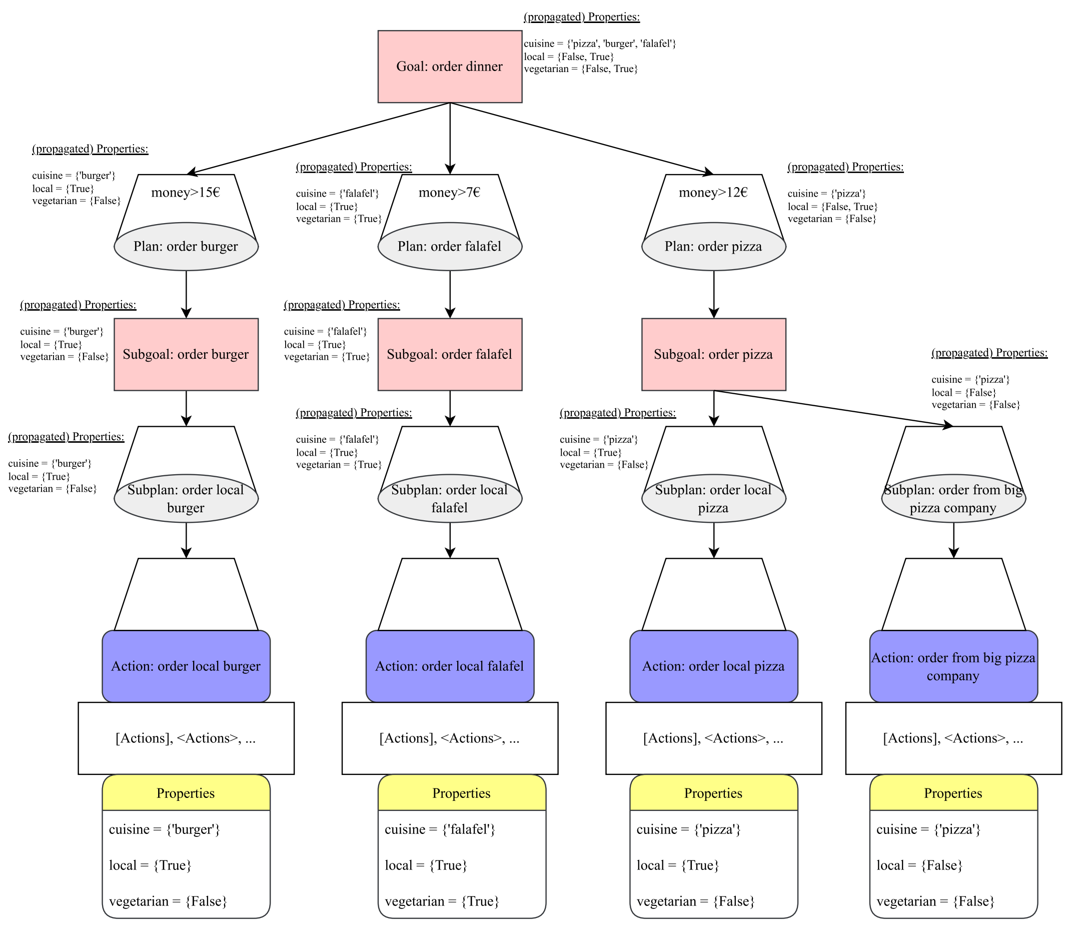


Figure 3.3: Example of property propagation [Own making]

At first glance, this looks like we are presupposing moral absolutism[2], but in actuality, that is not true. This is because properties are defined for each plan of each agent. That is, we can create an agent who thinks that lying is morally wrong, and an agent that thinks that it is morally right, or morally gray, etc. Also, since the same action can be part of different subplans, we can also encode the fact that the morality of actions depends on their context. For example, if an agent kills an animal as part of a subplan to have fun, we can label that action as morally evil, but if the same agent puts down an animal in his job as a veterinarian, then that action was not morally evil, even though they were the same action.

[2]Moral absolutism is the position that there are universal ethical standards that apply to actions, and according to these principles, these actions are intrinsically right or wrong, regardless of what any person thinks, or context.

3.5 Limitations of the model and possible improvements

Along the previous sections, we have explained every addition we have made to the agent model from a formal point of view. Now, in this section we will discuss about the limitations that our modifications carry, and possible ways in which those limitations can be fixed or palliated.

3.5.1 Limitations from goals

One of the biggest limitations stemming from our declaration of goals is that, at any given moment, our agent can only pursue one goal at a time. We cannot have two or more goals active at the same time, and we also cannot 'merge' goals together, either. But it is important to note here that this limitation is present in the theoretical BDI model, and it is also common in many BDI-inspired implementations. Only few agent platforms (such as Jason[3] or 2APL[8]) allow to pursue several goals at the same time.

Another limitation is that we only encode goals as a desired state of the world. Therefore, this does not allow goals such as 'maximize X variable' or 'perform Y action'. This limitation is also present in many BDI-inspired implementations. However, in our case these expressions can be simulated by, for instance, encoding them in goals such as '$X > \infty$', or having a variable named 'Y_has_been_performed' that is only set to True after Y has been performed, respectively. Nonetheless, it would be good to explicitly support these and other kinds of goals.

Finally, our agents do not support the addition (or subtraction) of goals mid-simulation. That is, and agent is created and dies with the same goals. These goals can be either achieved or not achieved at any given moment, but they cannot be eliminated (nor new goals can be added). This is a limitation not present in other BDI-inspired implementations that we have introduced for performance reasons[3], however, the ability for an agent to create new-goals or to drop existing goals mid-simulation could be interesting to have in a future extension of the platform.

3.5.2 Limitations from preferences over goals

Perhaps the biggest limitation in our declaration of preferences over goals is that they are absolute, and this stems from the fact that we do not use numbers anywhere here. Therefore, we cannot express things like 'I prefer

[3]We will better discuss this limitation in §4.1

this **a little** more than that', or 'I prefer that **a lot** more than this': it is all absolute.

A bit of an issue regarding the trigger conditions of non-default preferences over goals is what to do when those conditions overlap. As of right now, our approach is to pick whichever, and to allow the designer to implement an ad-hoc, more complex solution, if their scenario so requires, but it would be better to modify the structure to allow for a native way to handle this issue that is more complex than simply picking one among many.

3.5.3 Limitations from preferences over plans, and moral values

The main limitation of our preferences over plans is that it is a simplification of Visser's approach to doing the same, where he provides a more complex structure that allows his agents to have more complex preferences. For instance, his agents can reason about quantities, prefer to maximize or minimize a quantity, prefer that a quantity never exceeds a certain value, etc. Also, his agents are able to automatically extract properties of goals by looking at the actions, and then derive the relevant properties of the goals, while our model relies on the designer carefully listing the properties.

The limitation regarding overlapping trigger conditions also exists here, as both are handled in the same way.

Finally, our encoding of moral values also totally relies on the designer carefully listing which actions have what moral implications and, while this is good from an expressiveness point of view (it allows us to declare moral relativism (different agents having different moral convictions) and context-dependent morality (the same action carried out under different circumstances having different moral implications), it is a very exhaustive and daunting task, and it would be good to have the system partly automated, perhaps employing some matching between the purpose of an action and some pre-existing model of values such as Schwartz's[24], which is employed by Dignum et al. in [7].

Chapter 4

Implementation

In the previous chapter, we explored, one by one, all the improvements we introduced into the multi-agent model, from a theoretical/formal point of view. We revisited the contents of the original model (in §1.2.2), and modified the formal definitions of all the applicable elements to represent and cover all the additions we have inserted.

In this chapter, we will describe the actual implementation of all the additions, the object classes used, the ones that have been expanded, etc., in order to understand both how the implementation works and also how simulations must be designed and specified. Throughout every section of this chapter, all modified classes and modified members of each class will be listed, and their modifications will be highlighted and explained, along with providing the complexity for the most relevant algorithms and parts.

This chapter is structured as follows: first, in §4.1, we will detail every element of our implementation of goals and the library of plans, closely followed by the explanation of our implementation of preferences preferences over plans in §4.2. Then, in §4.3, we provide an explanation of the same degree for our implementation of preferences of plans, actions, and moral values. Finally, in §4.4, we describe other relevant modifications that may have been done to any of the classes that may not have been covered in the earlier sections, or that may be the consequence of any of our main implementations, but it is something secondary (like, for instance, updating the code of a parameter-passing function to accommodate for extra parameters).

4.1 Implementations of goals and the Metaplanner

As said in section §3.1, the first improvement we added to the model was the concept of goals. With them, now agents have a data structure independent from the HTN, that they can use to specify their objectives (the desires of the BDI model). Furthermore, they can have many independent goals, as opposed to ultimately having a single goal like they did before.

In this section, we will go over of our implementation of goals and the library of goals, formally described in §3.1. If we recall, we defined goals as a desired state of the world that the agent strives to achieve, and we defined the library of plans (Metaplanner), as a collection, for each goal, of all the plans that can be used to achieve that goal in particular.

4.1.1 Goal class

For the purposes stated above, we created the Goal class from scratch. This class contains all the functions necessary to declare goals, check their status, update them, etc. In it, the basic structure and syntax of a Goal is specified, as well as the function `check_goal` is defined, which is what determines if a goal has been achieved or not, according to some set of beliefs.

The structure of a `Goal` is defined as follows:

```python
def __init__(self, name=None, conditions=None, description=None):
    self.name = name
    self.description = description
    self.conditions = conditions  # conditions = Set of Dictionary<belief name: str, (operation: str, value)>
    if self.conditions is None:
        self.conditions = []
    self.achieved = False
```

Source Code 4.1: Constructor of Goal

As we can see, a `Goal` is defined by a name (which should be used as a *unique* identified), a description, and the most important part, its conditions. The conditions of the `Goal` are directly specified as a list of dictionaries. Each dictionary works as a way to express conjunctions (logic '*and*') of different assertions, and the list that holds the dictionaries implicitly establishes a disjunctive (logic '*or*') relationship between them. A `Goal` is achieved if, and only if, all the conditions at least one of its dictionaries are true. Each assertion is done by matching the name of the variable to a tuple, where the first element is a binary operator and the second element is what we are comparing the variable to. An example of the conditions of a `Goal` could be:

```
conditions = [{'money': (>=, 10)}, {'friend_can_pay': (==,
    True), 'friend_is_here': (==, True)}]
```

which encodes, what in natural language could be expressed at “either we have at least 10€, or we have a friend that can pay for us and that friend is here with us”.

A `Goal` has other attributes, such as the attribute ‘achieved’, which tells us whether its conditions have been met or not, according to some belief base.

Finally, we have the `check_goal` method, which takes a `BeliefSet` as input and returns `True` if, and only if, the `Goal`’s conditions are met under said belief base. It is defined by the following code:

```
def check_goal(self, beliefs):  # beliefs: BeliefSet
    if self.conditions == []:
        return True
    ret = False
    for dic in self.conditions:  # in this loop we iterate
   through every dictionary 'dic' of the set 'self.
   conditions'
        curr_set = True
        for name in dic:
            op, desired_val = dic[name]
            if name in beliefs:
                real_val = beliefs[name]
            else:
                curr_set = False
                continue
            if op == "==":
                if real_val != desired_val: curr_set = False
            if op == "!=":
                if real_val == desired_val: curr_set = False
            if op == ">=":
                if real_val < desired_val: curr_set = False
            if op == ">":
                if real_val <= desired_val: curr_set = False
            if op == "<=":
                if real_val > desired_val: curr_set = False
            if op == "<":
                if real_val >= desired_val: curr_set = False
        ret = ret or curr_set
        if ret == True:
            self.achieved = True
            return True
    self.achieved = ret
```

```python
    return ret
```

Source Code 4.2: Function check_goal

Very briefly, what this function does is return `True` if the set of conditions is empty. Otherwise, it iterates through every dictionary of it (there is at least one), and checks whether all the assertions of at least one dictionary are met. Its syntax and inner workings are a bit rudimentary, but it allows to express any logic formula (by using conjunctions, disjunctions, and negations). Finally, it also sets to either `True` or `False` the value of the 'achieved' attribute, which will be useful later on. If we let n be the total number of assertions made in the conditions, then its cost would be $\mathcal{O}(n)$, seeing as we perform, in the worst case, n iterations and, at each iteration, we check six conditions (which has cost $\mathcal{O}(1)$), and we access a dictionary, which is a hashmap, and the average cost of getting an item from a hashmap is also $\mathcal{O}(1)$.

4.1.2 GoalSet class

Now that we have fully implemented the `Goal` class, we implement a collection of `Goals`. The concept and the inner workings of this class are very straightforward: the `GoalSet` class serves as an unordered set of `Goals`, serving to keep the `Goals` of each agent stored together. It is, basically, a Python list, but with extra functionalities. It is solely defined by a set of `Goals`, it is iterable, and specific `Goals` can be extracted from it by unique name:

```python
def __init__(self, goals=None):
    if goals is None:
        self.goals = []
    else:
        self.goals = goals

def __iter__(self):
    for each in self.goals:
        yield each

def __len__(self):
    return len(self.goals)

def __getitem__(self, item):
    for g in self.goals:
        if g.name == item:
            return g
```

Source Code 4.3: Constructor of GoalSet

It has other functions, such as `remove_goal`, `add_goal`, `get_status` (which returns a dictionary with the status of every `Goal`, and `all_achieved` (which returns `True` if, and only if, all `Goals` have been achieved the last time they were checked).

4.1.3 Metaplanner class

The next step we took to finally decouple goals from plans was to implement the `Metaplanner` class. It is a very simple class, which inherits from the `HTNPlanner` class, and modifies some of its elements to its advantage. It is defined by:

```python
class MetaPlanner(HTNPlanner):
    def __init__(self, goal, methods):
        root_task = CompoundTask(methods=methods, name="Achieve " + goal.name)
        self.goal = goal
        HTNPlanner.__init__(self, root_task=root_task, preconditions=None, postconditions=None, verbose=True)
```

Source Code 4.4: Declaration and constructor of Metaplanner

As can be seen in Source Code 4.4, each alternative plan for a goal is described by a concrete instance of the `Method` class, and the constructor of the `Metaplanner` automatically puts them all together as alternative children of a `CompondTask` to achieve the appropriate goal. It has a single method, `pick_plan`, which is responsible for picking among the different plans that are defined for a concrete goal. Its code is the following:

```python
def pick_plan(self, beliefs, preferences):
    methods = self._root_task.methods
    avail_methods = []
    for m in methods:
        if m.conditions.check_conditions(beliefs):
            avail_methods.append(m)
    chosen_method = choose_method(avail_methods, beliefs, preferences)
    if chosen_method is None:
        chosen_method = Method(name="Empty method")
    root_task = CompoundTask(methods=[chosen_method], name="CompTask: Plan to: " + chosen_method._name)
    return HTNPlanner(root_task=root_task, verbose=True)
```

Source Code 4.5: Function pick_plan

Very briefly, this function takes as input a belief base and preferences over plans (which we will not analise here, as we will discuss them in the next section). It uses the belief base to filter out all the plans are *not* available for execution, and all we are left is the plans that we can actually follow given the current circumstances. Out of all these plans, it uses the `choose_method` function (which picks the available plan that adjusts the most to the agent's preferences), and it automatically creates an instance of an `HTNPlanner` object that has a single plan, the plan that has been chosen. The content of the `choose_method` will be covered in §4.3.

The cost of this function will be $\mathcal{O}(m\alpha+\beta)$, where m is the total number of alternative plans, α is the cost of checking a plan's preconditions, and β is the cost of the `choose_method` function. Likewise, $\mathcal{O}(\alpha)$ is actually $\mathcal{O}(|C|)$, where $|C|$ is the total number of assertions that make up a plan's preconditions, and $\mathcal{O}(\beta)$ will be expanded in its corresponding section (§4.3).

So far, we have already introduced remarkable improvements to the system. Before we added these, agents could only have a single goal, and said goal was given to the agent in the form of a plan (or plan with alternative subplans). Now, the agent has a whole collection of goals to pick from, a structure to encode goals that is independent from plans, and a library of plans for goals.

4.2 Implementation of preferences over goals

Thus far, have given our agents a set of goals, and ways to know which goal has been achieved and which is yet to be achieved. Now, we will explain how the agents' capabilities to reason about which goal to pursue first or, in general, in which order to pursue goals, have been implemented.

In this section, we will go over of our implementation of preferences over goals, formally described in §3.2. If we recall, preferences over goals would be stated by defining *one* default strict partial order relationship between goals, and *potentially many* alternative strict partial orders between the same goals, with trigger conditions associated to them. From whichever strict partial order is currently active, we deduce its associated DAG (because every strict partial order has a unique DAG associated to it), and from its DAG we compute a valid topological ordering, which is the order in which the goals will be pursued, as long as the current active preferences' conditions hold.

For these purposes, we defined two classes: `GoalPreference`, which implements a single strict preorder relation over a `GoalSet`, and its associated conditions, and the `GoalPreferenceSet` class, which basically acts as a collection of `GoalPreference`s, keeps the currently active preferences updated, and picks the current or next `Goal` to pursue according to them.

4.2.1 GoalPreference class

The `GoalPreference` class which, as said before, implements one strict partial order relation over goals, is instantiated as follows:

```
def __init__(self, conditions=None, goals=None, **kwargs):
    # conditions: same as in Goal class, goals: GoalSet,
    kwargs: g_i=[g_j,g_k], g_j=...
    self.conditions = conditions
    self.goals = goals
    self.adj_list = kwargs
    self.topo = None
    for g in self.goals:
        if g.name not in self.adj_list:
            self.adj_list[g.name] = []
```

Source Code 4.6: Constructor of GoalPreference

It has a set of goals, and the relationship is specified in the keyword arguments, and it acts directly as the graph as an adjacency list. If one `Goal` was not specified preferences over, it is added to the graph without children. Its attribute ‘conditions’ is of the form and it works exactly the same as `Goal` class’s conditions. It possesses a `check_conditions` function that is the same as `Goal`’s, a `compute_topo_sort` which computes the topological ordering, with a cost of $\mathcal{O}(|V| + |E|)$, where $|V|$ is the number of `Goals`, and $|E|$ is the number of edges in the DAG. This cost, however, is not per iteration, it is *per simulation*, because the topological ordering is computed only once, before the beginning of the simulation, and simply consulted every time that it needed. This is one of the cases where we are interested in trading off memory space to gain time efficiency.

Finally, it also comes with other utility functions, such as `pick_goal` and `goal_available`, which pick the next goal according to the agent’s preferences beliefs, and check if a goal is available or not according to the agent’s beliefs, respectively. In order to do so, the former iterates through the topological ordering and returns the first goal that has not been achieved, with a cost of $\mathcal{O}(|V|)$; and the latter uses the former to obtain the next goal and compare it to a given goal, and therefore has the same cost.

4.2.2 GoalPreferenceSet class

Finally, we have the `GoalPreferenceSet class`, which acts as a collection of `GoalPreferences` and helps the agent decide which conditional preferences apply, etc. Its whole body looks is implemented by this code:

```
class GoalPreferenceSet(object):
```

```python
    def __init__(self, default=None, alternatives=None, beliefs=None):  # default: GoalPreference, preferences = set of GoalPreference
        self.default = default
        if alternatives is not None:
            self.alternatives = sorted(alternatives, reverse=True, key=lambda x: len(x))
        else:
            self.alternatives = []
        self.current = self.select_preference_set(beliefs)

        self.default.compute_topo_sort()
        for gp in self.alternatives:
            gp.compute_topo_sort()

    def select_preference_set(self, beliefs):  # we return the first alternative that is available due to its trigger conditions being true. Remember that alternatives are sorted based on the # of trigger conditions from more to less
        for p in self.alternatives:
            if p.check_conditions(beliefs):
                self.current = p
                return p
        self.current = self.default
        return self.default

    def pick_goal(self, beliefs):
        return self.current.pick_goal(beliefs)
```

Source Code 4.7: GoalPreferenceSet class

As we can see from its constructor method in Source Code 4.7, it computes the topological ordering of every graph at declaration time (and therefore we can save that part during the simulation), and orders the alternatives in the manner described in Chapter 3: from most conditions to less (as the length of a `GoalPreference` is defined as the sum of all its assertions). Finally, it has a `select_preference_set` method which, just as its name indicates, traverses the alternatives in order (that order is from most number of assertions in their conditions to least) and returns the first one that applies (and, if none apply, it returns the default preferences, which is always applicable), and a very naive `pick_goal` method, which simply calls the `pick_goal` method of the `GoalPreference` that applies right now.

With all these classes and methods, our agents can now choose their next goals based on their preferences, and these preferences are not static, they are context-dependent and might change as the environment changes and evolves.

4.3 Implementation of preferences over plans and moral values

Finally, here we will describe the implementation aspects of our last major addition: preferences over plans, actions, and moral values. After this is implemented, our agents will be able to reason using preferences over both *what* they want to achieve and *how* they want to achieve it.

In this section, we will go over of our implementation of preferences over plans, actions and moral values, formally described in §3.3 and §3.4. If we recall, we defined the concept of Properties of a goal, which were a collection of relevant variables that describe the different ways in which a goal can be achieved and how they would affect the world. We defined preferences over plans and actions as an instantiation of a subset of the goal's properties, assigning them a subset of their values, which represent the values the agents prefer (and assumes that the agents does *not* prefer values and properties not mentioned). Then, we specify many trigger-based preferences over plans for the same goal, to make these preferences also context-dependent.

4.3.1 Properties class

First of all, let us start with the `Properties` class. Very briefly, the class works mostly as an ordered dictionary, with some additional methods and features, such as being iterable, searchable, etc. Its most important code is its constructor function:

```python
def __init__(self, dict=None, **kwargs):  # kwargs = Dict[
    Str, List]
    if not dict and not kwargs:
        self.props = OrderedDict()
    elif not dict:
        self.props = OrderedDict(kwargs)
    else:
        self.props = OrderedDict(dict)
```

Source Code 4.8: Constructor of the Properties class

As we can see from Source Code 4.8, the class mostly works as a ordered dictionary. It has some nice methods, such as `add_property` or `remove_property`, which do as their names suggest, but overall it just exists to store Properties (with *unique* names) and a list of their possible values. When used to specify preferences over plans, the list of possible values acquires a new meaning: it becomes the list of *preferred* values for the Property.

4.3.2 PlanPreferences class

This class basically works as a collection of `Properties` objects for each `Goal` of an agent. Its body and concept is the exact same as the `GoalPreference` class: it has *one* default `Properties`, and *potentially many* alternative, trigger-based `Properties`. They are selected and replace the default ones in the exact same way as it happens in the `GoalPreference` class.

Finally, it also possesses a method to update the current/active preferences based on the most recent beliefs of the agent. In general, all these methods that update the preferences (or that check if the preferences should be updated) are constantly run at each iteration, and where exactly they are run and under which conditions will be explained in detail in §4.4.1.

Its most relevant source code is the following:

```python
def __init__(self, goalname=None, default_prefs=None, beliefs=None, *args):  # goalname: Str, default_prefs: Properties, args: [(Properties obj, conditions as in goals)]
    self.goalname = goalname
    self.default = default_prefs
    self.alternatives = sorted(args, reverse=True, key=lambda x: len(x[1]))
    self.current = self.select_preference_set(beliefs)
```

Source Code 4.9: Constructor of the PlanPreferences class

As we can see in Source Code 4.9, it mainly works as a container for conditional preferences over plans. It differs a bit from the way in which its sibling class `GoalPreference` it is initialized because here the alternatives are defined in the optional arguments, but this is mostly cosmetic, as this way, if there are no alternatives, the code looks smoother.

Finally, let us mention that due to PyCOMPSs utilizing Python 2.*, we need to make the declaration of `Properties` for preferences a bit annoying. Because we need them to maintain an order, we need to create them as an empty `Properties`, and then fill them one by one. This is because every other way of creating them implies using a (non-ordered) dict, which in our current version of Python we are *not* guaranteed that it will preserve order, and in many instances it does not. We are only guaranteed to maintain order if we declare them empty and fill them one by one using the `add_property` method.

4.3.3 Adaptation of the HTN nodes

In order for preferences over plans to work, we need to actually embed the `Properties` to the HTN nodes. If we recall, we only specified them by hand to the leaves (i.e., the Primitive Tasks), and they were propagated automatically to the nodes. All HTN node classes (`PrimitiveTask`, `CompoundTask`, and `Method`) have been given 'properties' attribute, and we have created a `merge_properties` method, whose code is Source Code 4.10. As we can see, it takes two `Properties` object as input, along with a boolean to signify whether we are merging alternative or sequential tasks. The process it carries out is the implementation of the merging process described in Chapter 3. If we let $n = max(|p_1|, |p_2|)$, then its cost would be $\mathcal{O}(n)$.

```python
def merge_properties(p1, p2, alt=True):  # alt means if we're merging properties of alternative tasks (True) or of sequential tasks (False)
    if p1.props == {} or p2.props == {}:
        return p1 if p2.props == {} else p2
    res = {}
    dic1 = p1.props
    dic2 = p2.props
    for v in dic1:
        if v in dic2:
            res[v] = list(set(dic1[v] + dic2[v]))  # dic1[v] and dic2[v] contain a list, we merge them, make them a set, then a list again to remove repetitions
        else:
            if "__none__" not in dic1[v] and alt:
                res[v] = dic1[v] + ["__none__"]
            else:
                res[v] = dic1[v]
    for v in dic2:
        if v not in dic1:
            if "__none__" not in dic2[v] and alt:
                res[v] = dic2[v] + ["__none__"]
            else:
                res[v] = dic2[v]
    return Properties(res)
```

Source Code 4.10: Code of method merge_properties

This function is later used to generalize the process of propagating properties in the following way:

```python
def propagate_properties(self):
    res = Properties(dict={})
    for prim_t in self._subtasks:
        res = merge_properties(res, prim_t.properties, alt=False)
    self.properties = res
```

Source Code 4.11: Function to propagate the properties of all sequential children, from the Method class

As we can see, it iterates through every child of the root `Method` and it starts by merging the first child's `Properties` with an empty one, then the result is merged with the second child's, and so on. All these propagation processes are executed only once, before the simulation is run, and they could even potentially be stored to reduce even further the number of times they have to be executed.

4.3.4 Adaptation of the means-ends reasoner

Finally, we adapted the means-ends reasoner to actually take preferences over plans into account. If we recall from earlier, the *only* place where there are alternative choices in the HTN is when having to pick one `Method` among the many that a `CompoundTask` has as children. What the HTN did before was picking the first `Method` whose preconditions were satisfied. Now what it does is filter the non-applicable `Method`s, and pick the one that satisfies the currently active preferences the most. Here is the whole code in charge of planning:

```python
def _plan(self, beliefs, planprefs):  # beliefs: BeliefSet
    current_beliefs = beliefs.copy()
    tasks_to_process = [self._root_task]
    decomp_history = []
    while tasks_to_process:
        current_task = tasks_to_process.pop(0)
        if isinstance(current_task, CompoundTask):  # treatment of Compound Tasks
            current_method = None
            # here is where preferences come into action, as methods are the ONLY place where there are alternative choices in the HTN
            avail_methods = []
            for m in current_task.methods:
                if m.conditions.check_conditions(current_beliefs): avail_methods.append(m)
            current_method = choose_method(avail_methods, current_beliefs, planprefs)
            if current_method is None:
                if decomp_history:
                    current_task, self._current_plan = decomp_history.pop()
                else:
                    return
            else:
                decomp_history.append((current_task, self._current_plan[:]))
```

```python
21                 tasks_to_process[0:0] = current_method.subtasks[:]
22         else:  # treament of Primitive Tasks
23             if current_task.preconditions.check_conditions(current_beliefs):
24                 for effect in current_task.effects:
25                     current_beliefs = effect.apply(current_beliefs)
26                 if self._current_plan and self._current_plan[-1].append == current_task.append:
27                     for action in current_task.action_block:
28                         self._current_plan[-1].add_raw_action(action)
29                 else:
30                     self._current_plan.append(current_task.action_block.copy())
```

Source Code 4.12: Code of method _plan, from the HTN class

We only need to pay attention to lines 7–13, as these are the only lines we have introduced. As we can see, these lines do exactly what we just described. However, if we look closely, we see that the `choose_method` function appears again. This is because the same process is used to pick alternative plans than to reason about preferences at every level of the HTN. Let us take a closer look upon that function's code:

```python
# this function will sort all available plans according to how much they fit our priorities, and return the one that fits them the most. Used both by MetaPlanner (in pick_plan) and by HTNPlanner (in _plan)
def choose_method(avail_methods, beliefs, preferences):  # prefs: PlanPreferences
    preferences.update_preference_set(beliefs)  # we check to see if new conditions apply
    current = preferences.current  # we extract the current preferences

    if not current:
        return avail_methods[0]

    classes = [[] for _ in range(len(current))]  # classes[i] contains all the methods (plans) that satisfy the i-th preference
    i = 0
    for p in current:
        for m in avail_methods:
            if p in m.properties:
                if check_preferred(current[p], m.properties[p]):
                    classes[i].append(m)
        i += 1

    first = True
    candidates = []
    for i in range(len(classes)):
        if first:  # at this point we still have not found the first non-empty class
            if classes[i]:
                first = False
                candidates = classes[i]
            else:
                continue
        else:  # we have already found the first non-empty class
            if not classes[i]:
                continue
            if classes[i]:
                inter = intersect(classes[i], candidates)
                if inter:
                    candidates = list(inter)
                else:
                    continue
        if len(candidates) == 1:
            break
    if first:
        return avail_methods[0] if avail_methods else None
    return candidates[0]
```

Source Code 4.13: Code of function choose_method

This code implements the functionality used to pick plans according to preferences over them described in §3.3.3. For that purpose, it creates a list it calls ‘classes’ and a list it calls ‘candidates’. The classes list holds one position for each property our current preferences have some assertions over. Thus, `classes[i]` stores all the applicable `Methods` which can satisfy our desired preferences. This is done through the `check_preferred` function, which basically checks if some preferences are a subset of a method’s preferences. Its code can be checked in Source Code 4.14.

Once the ‘classes’ list is fully defined, it tries to find the fittest `Method` candidates. Again, it replicates the functionality described in §3.3.3: it tries to find the left-most satisfied preference, and then it tries to see, from all the Methods that satisfied it, if they satisfy the next ones, and so on. It achieves this by intersecting the Methods that satisfy i with the Methods that satisfy $i+1$, as long as the intersection is not empty. If it is empty, it skips to $i+2$, assuming i is the left-most first preference satisfied by a Method or more. At the end, it has a list of potential candidate `Methods`, and it returns the first one (all would be equally valid) for more determinism. If there were no candidates at all (i.e., not even a single preference was satisfied), it attempts to return the first applicable Method. Again, we could return any Method, but for repetition of simulations and determinism purposes, we choose to return always the same.

Finally, let us reason about the function's cost. There are two main sequential loops that dominate the function's complexity. The first one is where we instantiate the 'classes' list, and the second one is where we actually pick the candidate `Methods`. The first loop has a cost of $\mathcal{O}(pm\bar{v})$, where p is the number of properties we have preferences over, m the number of candidate methods, and $\bar{v}$ the average number of preferred values per property we have preferences over. The second loop's cost is, in the worst case (i.e., assuming we always have to perform an intersection), $\mathcal{O}(p^2)$, because it iterates through the 'classes' list (p elements) and it calculates the intersection (and, for that, it iterates through every element again). Therefore, the total cost of the dominant parts of algorithm would be $\mathcal{O}(p^2) + \mathcal{O}(pm\bar{v}) = \mathcal{O}(p * (p + m\bar{v}))$.

```
def check_preferred(own, method):
    for v in own:
        if v in method or v == "__none__":
            return True
    return False
```

Source Code 4.14: Code of function check_preferred

At this point, we have covered every substantial modification done to the code to implement our additions. However, we have had to alter more parts of the code as a side effect, to accommodate for our modifications, and these changes will be addressed in the next section.

Finally, it should be noted that, while some of the costs we have deduced from our code might seem a bit prohibitive, mostly because they have products of many variables in them (and those products are not accounting that some functions are run for each iteration, for each agent), but reality could not be further away from the truth. Since, in most instances, these variables usually do not exceed one digit-length, let alone two. That is, in most cases, the number of alternative methods will range between 3 and 7, the number of preferences will be between 5 and 10, the number of preconditions will be bounded between 0 and 10, and so on, just to give some numbers as an example.

4.4 Alterations to the reasoning cycle and other modifications

In this section, we will briefly cover all the collateral changes of classes, code, etc., made necessary to accommodate all the newly added elements. These changes have happened mostly as a side effect of our main implementations, that is, as a consequence of it instead of being a main feature or addition, and therefore are not explained or covered in the same depth as our actual, main implementation objectives.

4.4.1 The new reasoning cycle

Here, we will cover how all our implementations have been totally integrated into each agent's reasoning cycle. The Source Code 4.15 shows the function `step`, which each agent executes for each iteration. There, the agents update their beliefs, pick their next goal using their knowledge of the world *and* their preferences over goals, and plan according to their preferences over plans and over moral values.

The first new thing we see is in line no. 5, where the agent updates the status of its goals at the very beginning, to see if other agents' actions or the environment have altered the status of their goals. Then, it goes on to check if the preferences over goals have changed in line no. 7. Seeing as *checking* if the preferences over goals have changed and *computing* them has the same cost, we compute them directly, but they may retain their value from last iteration. Then, we do the same for goals in lines no. 11–18, where we check if the agent should continue to pursue the goal it committed to in the last iteration or if it should change. Here, it is also checked if all goals have been achieved and should therefore transition intro 'idle' state.

Then, it goes on to do the equivalent process for the preferences over plans. In line no. 21 we check if we have defined preferences for a specific goal, and then in line no. 26 we update them. This function not only update the preferences over plans, it also returns `True` if, and only if, they changed with respect to the last iteration. This is done to check if we need to update the root of the planner or not, as seen in lines no. 29–34.

The rest of the lines have not been modified and remain the same as they were before we took over this project. They mostly deal with the processing of messages, reasoning once a plan has been picked, acting upon the plan, etc. We have included two extra functions we use at the end of the Source Code. We do not comment on them because their behavior is trivial.

```
1 def step(self, inbox, agent, environment, directory, state):
```

```python
    # perceive the environment:
    state.beliefs = self.perceive(environment, state.beliefs)
    # update status of goals:
    self.update_goals_status(state.beliefs, state.goals)

    # (possibly) update preferences over goals:
    state.goalprefs.select_preference_set(state.beliefs)

    # (possibly) update the current goal:
    new_goal = state.goalprefs.pick_goal(state.beliefs)
    goal_changed = state.current_goal != new_goal
    if goal_changed:
        state.current_goal = new_goal

    # check if the agent needs to go to the idle state
    if state.current_goal is None:
        state.current_goal = Goal(name="Idle", description="Idle")

    # update preferences over plans:
    if state.current_goal.name in state.planprefset:
        planprefs = state.planprefset[state.current_goal.name]
    else:
        planprefs = PlanPreferences()

    planprefs_changed = planprefs.update_preference_set(state.beliefs)

    # choose a NEW plan if necessary:
    if goal_changed or planprefs_changed:  # if the goal changed, we need to pick a plan for the new goal
        if state.current_goal.name != "Idle":
            metaplanner = state.metaplanners[state.current_goal]
            state.planner = metaplanner.pick_plan(state.beliefs, planprefs)
        else:
            state.planner = state.idle_planner

    # process messages:
    for message in inbox:
        state.beliefs, state.planner, reply = self.process(message, state.beliefs, state.planner)
        if reply is not None:
            self.outbox.put(reply)

    # goal check
    state.planner = self.goal_check(state.beliefs, state.planner)

    # reason:
    if state.planner is None:
        block = state.default_block.copy()
    else:
        state.planner = self.reason(state.beliefs, state.planner, planprefs)
        print("CURRENT PLAN: {0}\n".format(state.planner._current_plan))
        block, state.planner = state.planner.next_block()

    # act upon a plan:
    state.beliefs = self.execute(agent, block, state.beliefs, directory)

    # check role:
    role = self.role_check(state.beliefs)
    if role is not None:
        self.outbox.put(agent.compose_command("behavior", {"behavior": role}))

    self.outbox.put(Message(agent.id, agent.id, "state", state, special="s"))



def choose_goal(self, beliefs, goalprefs):
    return goalprefs.pick_goal(beliefs)



def update_goals_status(self, beliefs, goals):
    for g in goals:
        g.check_goal(beliefs)
```

Source Code 4.15: Code of the reasoning cycle and other auxiliary functions

4.4.2 Changes to State class and other inner changes

The State class was also modified as a side effect. It was added all the newly introduced elements to actually represent the state of an agent. Its constructor method now looks like this:

```
def __init__(self):
    self.beliefs = BeliefSet()
    self.planner = None
    self.default_block = ActionBlock()

    # elements we have added in this work:
    self.goals = GoalSet(goals=None)
    self.current_goal = None
    self.metaplanners = None
    self.goalprefs = None
    self.planprefset = None
    self.idle_planner = None
    # ignore all the 'None's, since they are all set in the
   function set_state, not in the constructor. The
   constructor is only used to first instantiate an agent's
   state, but it is actually filled when the aforementioned
   function is called
```

Source Code 4.16: New attributes of the State class

As we can see, we have separated the elements that were already there from the elements we have needed to add.

Along the way, all functions that are used to pass parameters to generate agents, mostly located in the `Controller` class, have been adapted to be able to create new agents with all the necessary components listed below. The code is not included since it is a trivial passing of parameters.

It is also worth noting that another secondary feature we have bestowed upon agents is the possibility of exhibiting an **idle** behavior. In general, when an `Agent` is created, it will be given a set of goals. When these goals have been achieved, what should the agent do? We have added a functionality to address that specific question. By default, if not specified otherwise, all agents will have a default 'idle_planner' plan, that they will follow if, and only if, all their goals have been achieved. This default idle plan will have the agent do nothing for the whole iteration. However, we have given the designer to give each agent a custom idle plan. This is useful and it helps produce more realistic simulation, as we now can have agents with different 'idle' behavior. For instance, if we are simulating a house cleaning agent, its most likely idle plan will be to shut down and wait to be activated again, but if, however, we were simulating a taxi-driver agent, perhaps it would be more useful to have it wander around the city looking for potential customers as its idle behavior.

4.4.3 Changes to preconditions of the HTN nodes

Finally, the class used to set the preconditions for HTN nodes and actions was *upgraded* to allow to express '*or*'s with them. Before, it could only express conjunctions of assertions. Now, it can also express disjunctions of conjunctions of assertions. For that purpose, we have reused its old code and embedded it in a class that acts as an implicit disjuncturator of all the conjunctions. It is called `CondSet` implemented as, basically, a list of `Conditions`. Its `check_conditions` returns `True` if, and only if, at least one of the `Conditions` in the set return `True` when executing its homologous method.

Chapter 5

Testing and examples

In the previous chapter, we outlined the details and specifics of our concrete implementation of the elements introduced in Chapter 3. We highlighted every modification done to every class affected, explained the inner workings of every newly added function and method, and provided a cost analysis of every major function that is run.

In this chapter, we aim to exhibit how our system works, and we will do so through the execution of a number of experiments. Each of these experiments has its own purpose, and putting them all together, we achieve the purpose of showcasing every new feature of our modified model. First, we will describe the experiments that are meant to display the performance of the system. Then, we will present a complex scenario that is supposed to show how our agents fare with the new additions: agents having many goals, goals decoupled from plans, preferences over goals, plans, and over moral values, etc.

This chapter is structured as follows: first, we will provide a general structure for all the tests and better present them in §5.1; then, in §5.2, we will introduce all our functionality and performance tests, along with their results and comparisons. Lastly, in section §5.2, we will thoroughly explain the complex scenario which will be used to showcase all the new additions to the system, along with the expectations and results of that test too.

5.1 Structure of the tests

All tests will follow the same structure: the scenario will be described, and we will briefly cover how they have been implemented using the classes defined in Chapter 4. We have two big families of tests based on their complexity, basic and complex. Basic tests are about showing or proving a small functionality or subset of functionalities that we have in our system, and complex tests are about showcasing most, if not all, the features of the system working at the same time. In the former, the scenario is easily described in one or two sentences, while in the latter, the scenario is normally complex to describe and modeled after a real-life scenario.

Based on their purpose, tests can be divided into either functionality or performance. The purpose of functionality tests is to demonstrate and showcase the system at work, while the sole purpose of performance tests is to test the platform's performance, time to run based on the size of the input, and scalability.

For these purposes, we have defined three tests. Two of them are basic, and one of them is complex. The basic tests are actually the adaptation of some of the tests carried out by Gnatyshak when he was designing the original system, while the complex test is the simulation of some citizens' days in a made-up town, where they have to complete their daily chores, and have preferences over in what order to achieve them, and over how to achieve them.

When it comes to the functionality/performance dichotomy, the aforementioned tests will act as both functionality and performance tests. When they are acting as functionality tests, they will be verbose (showing the state of each agent at every iteration), we will make some predictions about what should happen, and we will contrast these predictions with the results we get. If, however, they are acting as performance tests, then they will not be verbose (because verbose implies the creation of multiple log elements that directly alter the execution time), and we will run them with an increasing number of agents to measure their runtime and derive other interesting, performance-related metrics.

For the performance tests, it is imperative that we compare them with their original version. Otherwise, we would just see some metrics (e.g., time to run the whole simulation, time to run each step, etc.) which, on their own, have no much value, besides from knowing if the overall runtime is appropriate or not. However, when compared to their predecessor, we can get a much richer insight into how scalable our implementations are, and what amount of efficiency we have traded for extra features such as preferences, or moral values.

Finally, let us give a detailed description of the environment all the performance tests will be run in:

- Asus Desktop Computer (7 years of age)
- 16 GB RAM DDR3

- Intel(R) Core(TM) i7-4790 CPU @ 3.6 GHz (8 CPUs)
- OS: Windows 10 (64-bit)
- Average room temperature: $27^{\circ C}$

This will be the host computer, as the simulations will be run in a VM that has installed all the necessary components to run COMPSs and PyCOMPSs, with the following settings:

- OS: Ubuntu 18.04 LTS (64-bit)
- 10 GB RAM from host
- 8 (virtual) processors from host

5.2 Basic scenarios

We have implemented two basic scenarios for our tests, and both are adaptations of the original tests from [10], adapted to run with the new `Goal` class and other elements.

The first one is the incrementation scenario, where we will have some agents with a very simple HTN to increment an internal counter in their beliefs. For the functionality part, we will see that this very basic examples can be encoded with `Goals` and other classes. For the performance scenario, we will run this test many times with different input sizes, and we will derive some performance-related metrics; then, we will run the *old* version of this test under the same conditions, extract the same metrics, and compare results to try to quantify any drops in performance.

The second test is the ping-pong scenario, where agents randomly 'ping' other agents by default. When an agent is 'pinged', then it tries to 'pong' whoever agent 'pinged' him. When an agent has been 'pinged' three times, then it will finish the execution and be removed from the simulation, until no agents remain. We will reimplement this scenario with goals, preferences over goals, etc., and then compare it with the original one. For both functionality and performance objectives, we will perform the same type of tests described in the above paragraph.

5.2.1 Incrementation test

This first test is also the simplest one. The default scenario is, basically, three agents with a default HTN. Said default HTN has only one internal action which increments the value of a `counter` field in the agent. It is run for 10 steps.

5.2.1.1 Outline and implementation

The implementation of this test is very simple as well. We create 3 agents (for the performance part, the number of agents is variable). Each of the agents has the same set of beliefs ({`counter=0`}), a single goal (`counter==numiter`), and a very simple HTN associated to it, that only increments the variable by one. This goal is inserted into the agents' set of goals, this HTN is associated with the goal through a MetaPlanner, and the agent is given empty preferences over goals (because there is only one) and actions (for the same reason). The scenario is summarized in Figure 5.1.

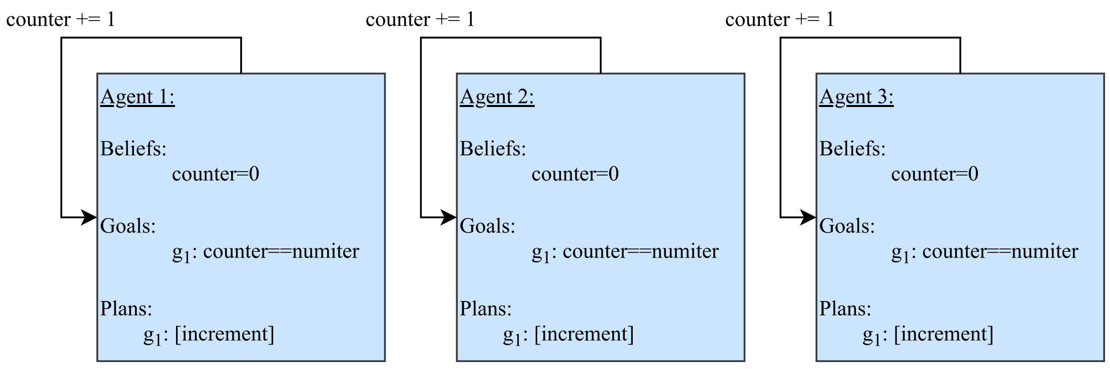


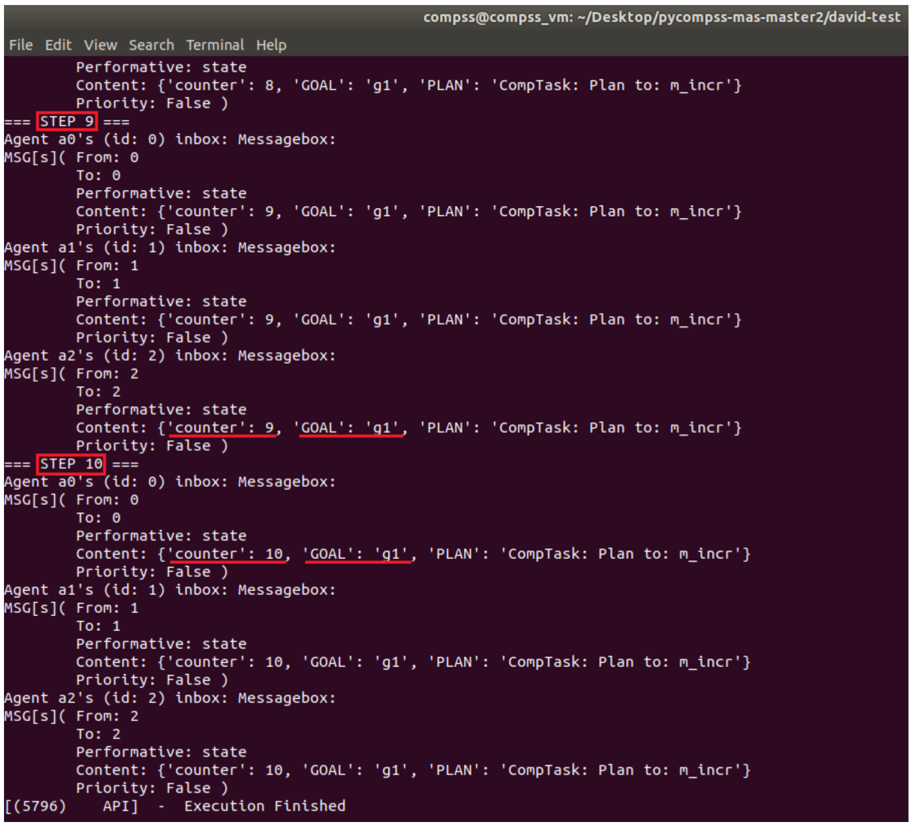
Figure 5.1: Increment scenario summary [Own making]

We expect the following:

1. The counter to increase equally for every agent and to be the same as the current number of steps
2. Since there are no messages, the only message should be the state of each agent at each step
3. In general, from a functional point of view, we expect it to work exactly as it worked in [10]

For the functionality version of this test, we will attempt to see that the scenario works as intended, while for the performance-related results we will run both our version and the original version of the test, and compute the appropriate performance-related metrics. For this scenario, and for all scenarios in this thesis, when displaying a time measurement for a runtime, that value displayed is always the average result of ten consecutive runs.

5.2.1.2 Functionality-related results


compss@compss_vm: ~/Desktop/pycompss-mas-master2/david-test
File Edit View Search Terminal Help

```
        Performative: state
        Content: {'counter': 8, 'GOAL': 'g1', 'PLAN': 'CompTask: Plan to: m_incr'}
        Priority: False )
=== STEP 9 ===
Agent a0's (id: 0) inbox: Messagebox:
MSG[s]( From: 0
        To: 0
        Performative: state
        Content: {'counter': 9, 'GOAL': 'g1', 'PLAN': 'CompTask: Plan to: m_incr'}
        Priority: False )
Agent a1's (id: 1) inbox: Messagebox:
MSG[s]( From: 1
        To: 1
        Performative: state
        Content: {'counter': 9, 'GOAL': 'g1', 'PLAN': 'CompTask: Plan to: m_incr'}
        Priority: False )
Agent a2's (id: 2) inbox: Messagebox:
MSG[s]( From: 2
        To: 2
        Performative: state
        Content: {'counter': 9, 'GOAL': 'g1', 'PLAN': 'CompTask: Plan to: m_incr'}
        Priority: False )
=== STEP 10 ===
Agent a0's (id: 0) inbox: Messagebox:
MSG[s]( From: 0
        To: 0
        Performative: state
        Content: {'counter': 10, 'GOAL': 'g1', 'PLAN': 'CompTask: Plan to: m_incr'}
        Priority: False )
Agent a1's (id: 1) inbox: Messagebox:
MSG[s]( From: 1
        To: 1
        Performative: state
        Content: {'counter': 10, 'GOAL': 'g1', 'PLAN': 'CompTask: Plan to: m_incr'}
        Priority: False )
Agent a2's (id: 2) inbox: Messagebox:
MSG[s]( From: 2
        To: 2
        Performative: state
        Content: {'counter': 10, 'GOAL': 'g1', 'PLAN': 'CompTask: Plan to: m_incr'}
        Priority: False )
[(5796)    API]  -  Execution Finished
```

Figure 5.2: Fragment of verbose output of the incrementation scenario [Own making]

First of all, let us start with the functionality version of this test. A fragment of the output can be seen in Figure 5.2. As we can see, the agent picks the goal correctly, and from the goal it picks the appropriate plan, through the MetaPlanner, and the general correct functioning of the scenario.

All of our expectations have been met, and we mark this functionality test as successfully passed.

5.2.1.3 Performance-related results

When it comes to the performance test, we expect the increase in runtime to be negligible when compared to the old test. This is because in this very simple test, our agents have one single goal, one single plan for the goal, no preferences whatsoever, and a very simple set of beliefs and environment.

No. of agents	Old runtime (s)	New runtime (s)	Difference (s)	Old runtime per ag. (s)	New runtime per ag. (s)
Default (3)	4.43	4.45	0.02	0.6772009029	0.6741573034
4	4.52	4.49	-0.03	1.13	1.1225
8	5.12	5.16	0.04	0.64	0.645
16	5.99	6.02	0.03	0.374375	0.37625
32	7.71	7.75	0.04	0.2409375	0.2421875
64	10.26	10.40	0.14	0.1603125	0.1625
128	15.24	15.45	0.21	0.1190625	0.120703125
256	25.19	25.88	0.69	0.0983984375	0.10109375

Table 5.1: Performance results of increment scenario [Own making]

Therefore, they do not have to check for any applicability conditions of preferences over goals or over plans, only one goal has to be checked at each iteration, etc.

We have computed the runtime for the scenario with the default number of agents (3), 4, 8, 16, 32, 64, 128, and 256, for both the old and new implementations of this test. Then, we have computed the differences in runtime between the old and the new versions of the scenario, for each number of agents. Finally, we also provide the average time that an agent was running in the simulation. All this information is displayed in Table 5.1.

As we can see, we have successfully predicted what would happen. There is close to no increment in runtime, and this is likely due to the fact that in this specific scenario, our additions are almost never used, and therefore they do not have the ability to delay the program's execution.

5.2.2 Ping-pong test

This next test is also somewhat basic, but here we can put our implementation of preferences over goals to the test. This scenario also has three agents in it, all having the same goals, the same library of plans, and the same preferences over goals. They will, by default, message other agents. If they have been messaged, they will switch to a different goal of responding; all that, until they have been messaged for a set number of times.

5.2.2.1 Outline and implementation

The environment has a single field: `counter`. The beliefs of every agent are a bit more complex, and they amount to:

- list `message`: the messages the agent has received
- boolean `got_msgs`: `True` if, and only if, the agent has received a message
- integer `to_env`: the value to be sent to the environment to be added to its `counter` field

- integer `reply_count`: the number of replies the agent has to send
- integer `counter`: stores the number of 'pongs' received by the agent

Their behavior is described as follows. At the beginning of every simulation step, if an agent has received a 'ping' message from any other agent, its goal will be to respond to that agent with a 'pong' message. If, however, it was a 'pong' message, then the agent will simply increase both its `counter` belief and the environment's by one. If no messages were received, then the agent will send a 'ping' message to another agent at random. When an agent's `counter` belief reaches 3 or more, they finish the execution.

Finally, the goals, and the preferences over goals, are defined by the following code:

```python
# goals:
g1 = Goal(name='g1', conditions=[{'counter': (">=", 10)}],
    description="Goal to treat incomming Messages")
g2 = Goal(name='g2', conditions=[{'counter': (">=", 10)}],
    description="Goal to send Messages")
goals = GoalSet(goals=[g1, g2])

# preferences over goals:
gpref1 = GoalPreference(goals=goals, g2=['g1'])  # by
    default, we want to send messages
gpref2 = GoalPreference(goals=goals, g1=['g2'], conditions
    =[{'got_msgs': ("==", True)}])  # if we have received
    messages, we want to treat them
goalprefs = GoalPreferenceSet(default=gpref1, alternatives=[
    gpref2], beliefs=beliefs)
```

Source Code 5.1: Instancing of goals and preferences over goals for the ping-pong scenario breaklines

As we can see in Source Code **??**, we have two goals, g_1 and g_2. Both have the same fulfillment conditions, which are only there to keep the goals active until the agents die, but they have different plans associated to them in the library of plans. If we look at the preferences over goals, we will see that, by default, g_2 goes before g_1; that is, by default, we want agents to randomly message other agents. However, if the belief `got_msgs` becomes true, then we switch preferences over goals, and now g_1 comes before g_2, thus switching the agent from randomly sending messages to treating the received messages. The plans for each goal can be seen in Figure 5.3.

From this test, we expect the following:

1. Agents to have goal g_1 by default

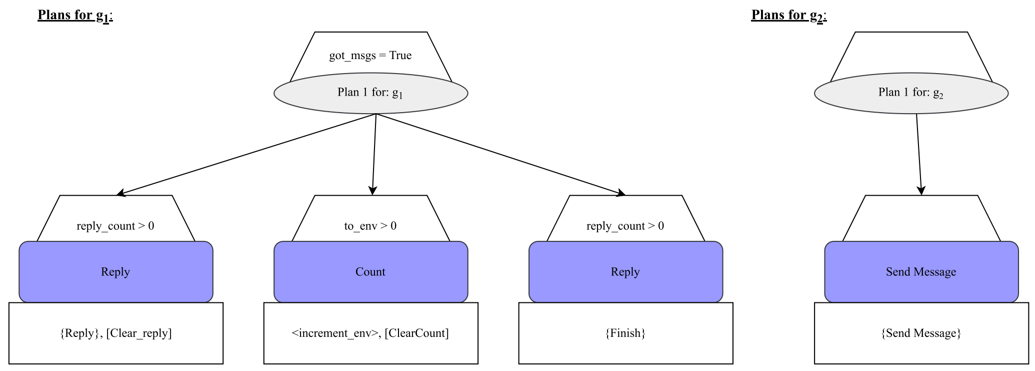


Figure 5.3: Plans for ping-pong scenario [Own making]

2. When an agent has been sent a message, we expect for it to switch to goal g_2
3. In general, we expect it to work exactly as its original implementation worked, from a functional point of view

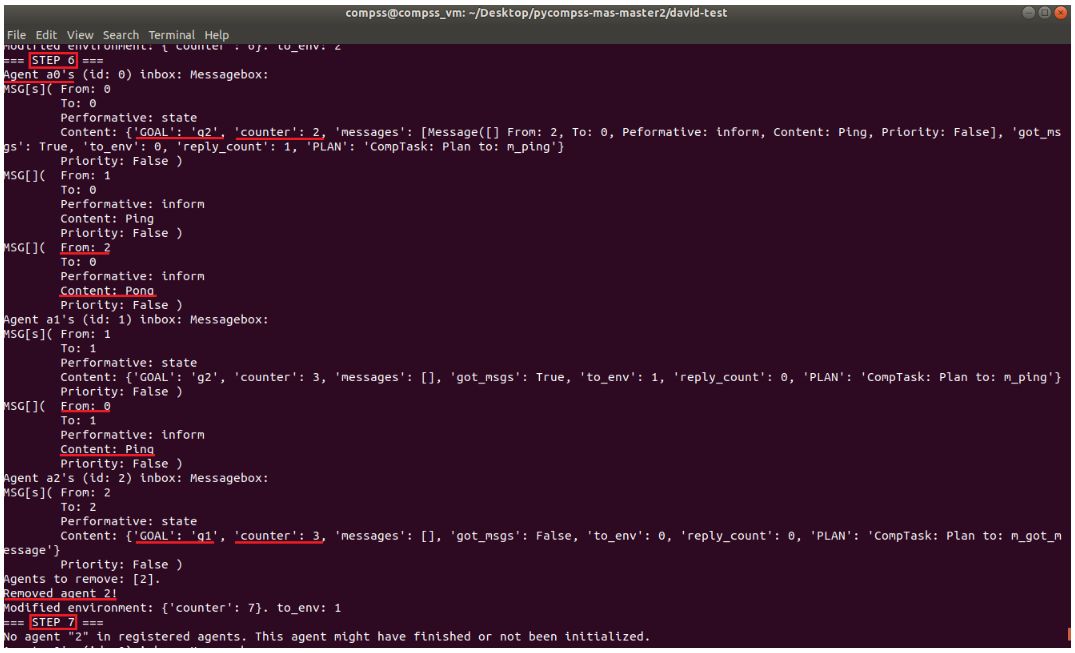
Just like with the increment test, here, for the functionality version of this test, we will attempt to see that the scenario works as intended, while for the performance-related results we will run both our version and the original version of the test, and compute the appropriate performance-related metrics.

5.2.2.2 Functionality-related results

Let us now see the results of the functionality versions of this scenario. Figures 5.4 and 5.5 are two fragments of the output. As we can see in them, the agents that are following goal g_2 are sending 'ping' messages, while the agents following g_2 are sending 'pong' replies. Whenever an agent's `counter` reaches 3, the agent is eliminated. Agents switch goals depending on weather they have been 'pinged' or not, and function properly. In general, our expectations have also been satisfied.

5.2.2.3 Performance-related results

Now, for this performance test, we actually expect the runtime to slightly increase for our new version. This is because now the scenario is a bit more complex. We have more than one goal to choose from, and the same goes for the number of libraries of plans and preferences over goals. Therefore, at each iteration, every agent will have to check the status of every goal, check



```
compss@compss_vm: ~/Desktop/pycompss-mas-master2/david-test
File  Edit  View  Search  Terminal  Help
Modified environment: {'counter': 6}. to_env: 2
=== STEP 6 ===
Agent a0's (id: 0) inbox: Messagebox:
MSG[s]( From: 0
        To: 0
        Performative: state
        Content: {'GOAL': 'g2', 'counter': 2, 'messages': [Message([] From: 2, To: 0, Peformative: inform, Content: Ping, Priority: False], 'got_ms
gs': True, 'to_env': 0, 'reply_count': 1, 'PLAN': 'CompTask: Plan to: m_ping'}
        Priority: False )
MSG[](  From: 1
        To: 0
        Performative: inform
        Content: Ping
        Priority: False )
MSG[](  From: 2
        To: 0
        Performative: inform
        Content: Pong
        Priority: False )
Agent a1's (id: 1) inbox: Messagebox:
MSG[s]( From: 1
        To: 1
        Performative: state
        Content: {'GOAL': 'g2', 'counter': 3, 'messages': [], 'got_msgs': True, 'to_env': 1, 'reply_count': 0, 'PLAN': 'CompTask: Plan to: m_ping'}
        Priority: False )
MSG[](  From: 0
        To: 1
        Performative: inform
        Content: Ping
        Priority: False )
Agent a2's (id: 2) inbox: Messagebox:
MSG[s]( From: 2
        To: 2
        Performative: state
        Content: {'GOAL': 'g1', 'counter': 3, 'messages': [], 'got_msgs': False, 'to_env': 0, 'reply_count': 0, 'PLAN': 'CompTask: Plan to: m_got_m
essage'}
        Priority: False )
Agents to remove: [2].
Removed agent 2!
Modified environment: {'counter': 7}. to_env: 1
=== STEP 7 ===
No agent "2" in registered agents. This agent might have finished or not been initialized.
```

Figure 5.4: Fragment of verbose output of the ping-pong scenario (1/2) [Own making]

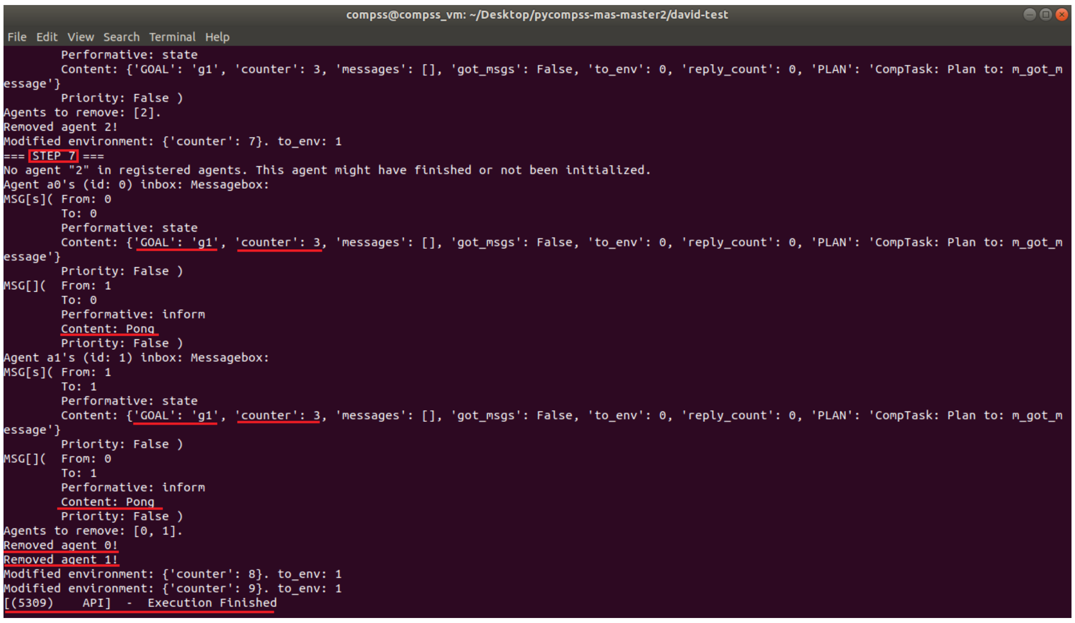
for the applicability of conditions, choosing a plan from the library, etc., thus making the agents take longer to run on average.

We have, again, computed the runtime for the scenario with the default number of agents (3), 4, 8, 16, 32, 64, 128, and 256, for both the old and new implementations of this test, and all the same metrics from the increment scenario. All this information is displayed in Table 5.2.

As we can see, we have again successfully predicted what would happen. At the beginning, with a small number of agents, the difference appears negligible. However, as we increase the number of agents, we can appreciate how the difference in runtime slightly increases, just as we said it would.

No. of agents	Old runtime (s)	New runtime (s)	Difference (s)	Old runtime per ag. (s)	New runtime per ag. (s)
Default (3)	4.25	4.44	0.19	0.7058823529	0.6756756757
4	4.51	4.63	0.12	1.1275	1.1575
8	4.85	5.1	0.25	0.60625	0.6375
16	5.86	6.08	0.22	0.36625	0.38
32	6.93	7.64	0.71	0.2165625	0.23875
64	9.62	11.03	1.41	0.1503125	0.17234375
128	13.38	15.59	2.21	0.10453125	0.121796875
264	22.83	28.73	5.9	0.08647727273	0.1088257576

Table 5.2: Performance results of ping-pong scenario [Own making]


compss@compss_vm: ~/Desktop/pycompss-mas-master2/david-test
File Edit View Search Terminal Help

```
        Performative: state
        Content: {'GOAL': 'g1', 'counter': 3, 'messages': [], 'got_msgs': False, 'to_env': 0, 'reply_count': 0, 'PLAN': 'CompTask: Plan to: m_got_m
essage'}
        Priority: False )
Agents to remove: [2].
Removed agent 2!
Modified environment: {'counter': 7}. to_env: 1
=== STEP 7 ===
No agent "2" in registered agents. This agent might have finished or not been initialized.
Agent a0's (id: 0) inbox: Messagebox:
MSG[s]( From: 0
        To: 0
        Performative: state
        Content: {'GOAL': 'g1', 'counter': 3, 'messages': [], 'got_msgs': False, 'to_env': 0, 'reply_count': 0, 'PLAN': 'CompTask: Plan to: m_got_m
essage'}
        Priority: False )
MSG[](  From: 1
        To: 0
        Performative: inform
        Content: Pong
        Priority: False )
Agent a1's (id: 1) inbox: Messagebox:
MSG[s]( From: 1
        To: 1
        Performative: state
        Content: {'GOAL': 'g1', 'counter': 3, 'messages': [], 'got_msgs': False, 'to_env': 0, 'reply_count': 0, 'PLAN': 'CompTask: Plan to: m_got_m
essage'}
        Priority: False )
MSG[](  From: 0
        To: 1
        Performative: inform
        Content: Pong
        Priority: False )
Agents to remove: [0, 1].
Removed agent 0!
Removed agent 1!
Modified environment: {'counter': 8}. to_env: 1
Modified environment: {'counter': 9}. to_env: 1
[(5309)    API]  -  Execution Finished
```

Figure 5.5: Fragment of verbose output of the ping-pong scenario (2/2) [Own making]

5.3 Complex scenario: a day in Goodsprings

In the previous section, we introduced a couple of toy examples to both showcase our implementation's features in a simple manner, under a very limited environment, and also to compare how these recreated tests scaled in their previous version (before to made any additions to this project) compared to our current version (with all our additions).

Now, however, we move on to the next scenario, which is a complex one. It features a very rich environment with several variables and resources for the agents to interact with. The environment will be randomly generated using a *seed* (so that the same random simulation can be repeated), it will be changing based on some user-defined probabilities, and the agents will react and plan accordingly to these changes of the environment. Agents have several goals, and different preferences over goals, as well as having different plans for every goal, preferences over these plans, and even moral values. The default value of the seed is '2022' (as a `string`), but it can be given any other string value.

5.3.1 Outline and implementation

Welcome to the proud town of Goodsprings! Goodsprings is a small town with some citizens living in it. These citizens are people, just like you and me, and they have their own set of daily goals (e.g., go to their workplace, have fun, eat dinner, etc.). Like the real people they are, they have preferences over *in which order* to pursue their goals, as well as preferences over `how` to achieve them: at the end of the day, it is not the same to work and then have dinner, or vice versa; and it is not the same either to go somewhere on foot, by car, etc. Additionally, for instance, we might prefer to go somewhere on foot if the weather is clear, but as soon as it starts raining, we might prefer to go by car instead. Finally, we might have some moral inquiries into the actions we perform (e.g., are we environmentalists and we think the unnecessary usage of cars is immoral?, etc.). We will be able to simulate all these characteristics in this complex scenario that we now present.

Goodsprings is a small town with the following features:

- It has a *town center* through all the other locations of the town are connected. People can live in the city center, too
- There are places where people go to have fun:
  - A beach
  - A park
  - A cinema
- There are also workplaces:
  - A factory
  - Corporate offices
- A residential neighborhood away from the city center, where some citizens live
- Places to go shopping:
  - A local market and farm. In the local market there are:
    - A local Italian restaurant
    - A local Chinese restaurant
    - A local falafel restaurant
  - A shopping center and supermarket. Inside the shopping center there are:

        * A big chain pizza company
        * A fast food, burger company
        * A big chain, wholefood/vegetarian meals company

- It has other public facilities:
    - A school for kids to attend
    - A hospital to treat the ailments of citizens. Citizens of Goodsprings might, by chance, experience a medical emergency, in which case, if they go to the hospital, they will be tended to and cured for free, so they can carry on with their day
- The town can experience the following weather conditions:
    - Clear weather (sunny)
    - Cloudy weather
    - Rainy weather
    - Snowy weather (schools are closed under this weather conditions)
- It also sports a public bike rental system. There are bike stations scattered among the many locations of the town, which any citizen can approach and take a bike out of, in order to use it to move in the city. When they get off the bike, they must leave it in the new location, and other citizens can potentially take that bike for their commutes. it is possible that some locations might not have any bikes at any given moment.
- There are three main ways to go around the city: by car, by bike, or on foot. In order to drive a car, an agent needs to own one. In order to drive a bike, an agent needs to be at a location where a bike is available, and pick it.

For a more visual overview of the town of Goodsprings, one can take a look at Figure 5.6. Other aspects of the town not present in the map are: the current weather, the current time, and the number of bikes at every location.

5.3.1.1 Environment

The `environment` implements the map from Figure 5.6, as well as other variables such as the current weather, the time, and extra internal variables for purposes of running the simulation. It is initialized by a random seed, and ht has the following fields:

- `location_name`: a dictionary with information about every location, for every location (i.e., one for 'school', one for 'factory', etc.). In actuality, it keeps track of how many bikes are at each location
- `weather`: the current weather
- `time`: a `list` of two elements. The 0th element is the hour, while the 1st element is the minute
- `__control__`: a dictionary of control variables for the Controller, not visible for any of the agents. It contains numbers for internal and random calculations, all derived from the seed, for purposes such as randomly changing the weather, etc.
- `__rands__`: a list of random numbers derived from the random seed that will be used throughout the whole simulation, for random calculations as well

The simulation consists of 64 steps. It starts at 08:00, and ends at 00:00 of the next day. Each simulation step corresponds to 15 minutes in the town. By default, the town starts with clear weather. Every iteration, there is a 10% chance of the weather changing. If that chance happens, there is a 60% chance of the weather becoming clear, 30% chance of becoming cloudy, 9% chance of raining, and 1% chance of snowing. At every iteration, there is also a 0.2% chance, for every agent, to experience a medical emergency. All these numbers can be set by the user at will at the start of every simulation, as well as the initial weather.

When an agent perceives the environment, they will only perceive the current time, the current weather, and the information of the location that they are currently in. For instance, if an agent is at the city center, it will not update its information about the state of the school, only about the state of the city center, the weather, and the time.

For performance test purposes, the user can also specify a number of extra agents, that will be generated along the main actors. These agents are copies of the main actors, with the same goals and plans, but will stress the system all the same in big numbers.

Finally, there is always a 'hidden', observer agent, who does nothing but perceive the environment, to show it at every step of the simulation.

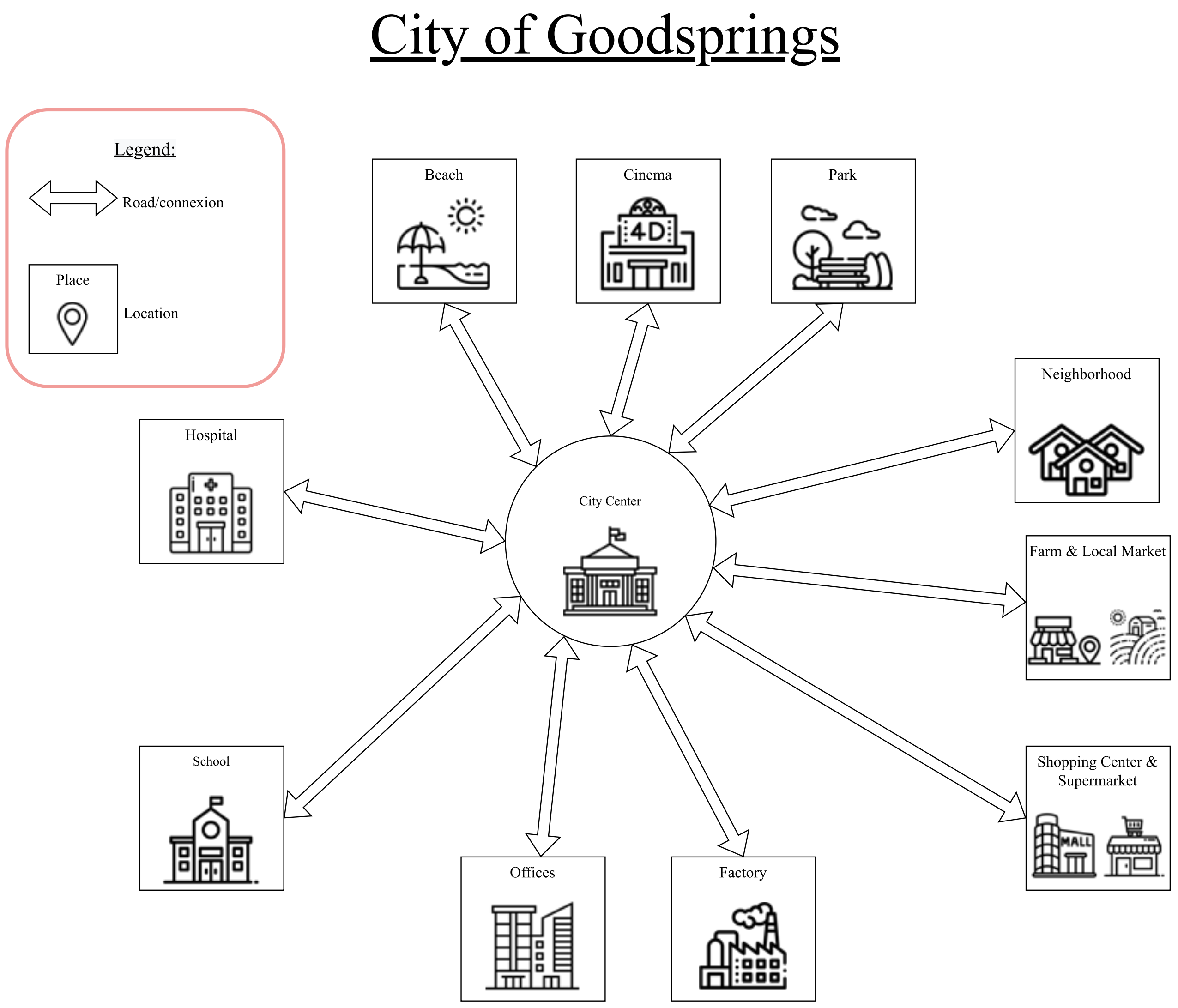


Figure 5.6: Map of the town of Goodsprings [Map: Own making; Icons: images free of use from www.flaticon.es]

5.3.1.2 Agents

There are two main actors in our environment. They both are complex agents with numerous goals, conditional preferences over these goals, a rich library of plans, and preferences over those plans, along with moral values. These two agents are Alice and Bob.

5.3.1.2.1 Alice

Citizen Alice is described by the following:

- Alice is the CEO of a big TMT company. She, therefore, works at the offices every day until 16:45
- Alice lives in the neighborhood
- She is the single mother of two children. Thus, she has to take the children to school every morning, collect them from school at 17:00, and go have fun with them in the afternoons (until 19:45). Then, they order food at 21:00
- Her goals are the following:
  1. Take children to school
  2. Go to work
  3. Work
  4. Go collect her kids to school
  5. Have fun with her kids
  6. Go back home
  7. Eat dinner
  8. Attend any medical emergency that might happen during the day

  – Goals no. 1, 2, 4, 5, and 6, include commuting. We will be able to see the agent's preferences on means of transportation, as well as its moral values, when it tries to achieve these

  – Goal no. 5 also has many options to be achieved. We will be able to see the agent's preferences on how to have fun when it tries to achieve this goal

  – Goal no. 7 will display the agent's preferences over food, as well as its moral values, when it tries to achieve this goal

- Her preferences over goals are the following:
  - **Default**: $g_1 \to g_2 \to g_3 \to g_4 \to g_5 \to g_6 \to g_7$
  - Conditional preferences (if **medical emergency**): $g_8 \to g_1 \to g_2 \to g_3 \to g_4 \to g_5 \to g_6 \to g_7$
  - Conditional preferences (if it is **snowing**): $g_2 \to g_3 \to g_4 \to g_5 \to g_6 \to g_7$
- Her preferences over plans and moral values are the following:
  - For **transport goals**:
    - Above all, Alice prefers to travel by car
    - On top of that, she is not an environmentalist
  - For **fun-related goals**:
    - Above all, and by default, Alice prefers to take her kids to the beach
    - If it gets cloudy, Alice prefers to go to the park
    - If it rains or snows, Alice then prefers to go to the cinema
  - For **food-related goals**:
    - Above all, Alice likes pizza from the big pizza company
    - On rainy days, however, Alice prefers to eat Chinese food
- Her initial beliefs are her current location, the current weather and time, the current location of her children, whether she owns a car, whether she has worked, if her children have gone to school, if she is at the center of the city, and whether there is a medical emergency. However, extra beliefs might be added during the simulation, for instance, if she has eaten dinner, her current goal and plan, what food she has eaten, etc. The purpose of many of these goals is to understand Alice's reasoning process, and not for her to use.

Now, we will provide Alice's library of plans for each of her goals in the form of figures. In Figure 5.7 can be seen all her plans for transport goals. In Figure 5.8, we can see all her plans for the fun-related goals. Finally, in Figure 5.9, we can see all her plans for ordering food. The root Methods are each individual plan. When stored in their MetaPlanner, the data structure that selects the plan automatically adds them a Compound Task on top as the true root task.

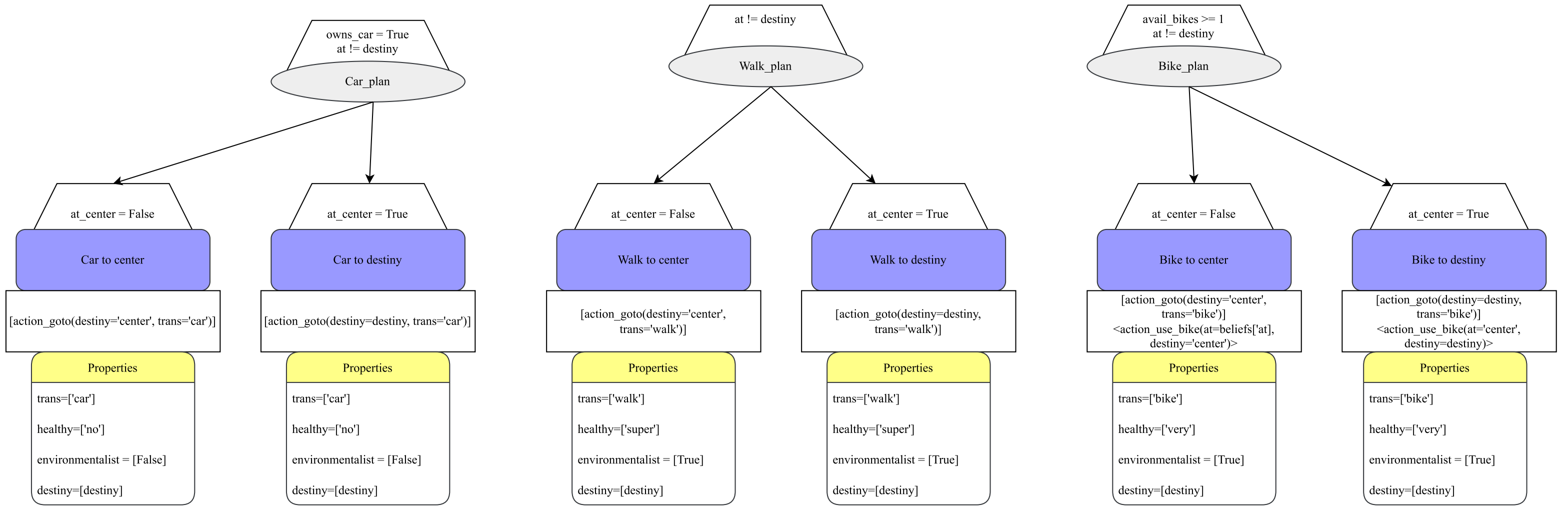


Figure 5.7: Library of plans for transport goals [Own making]

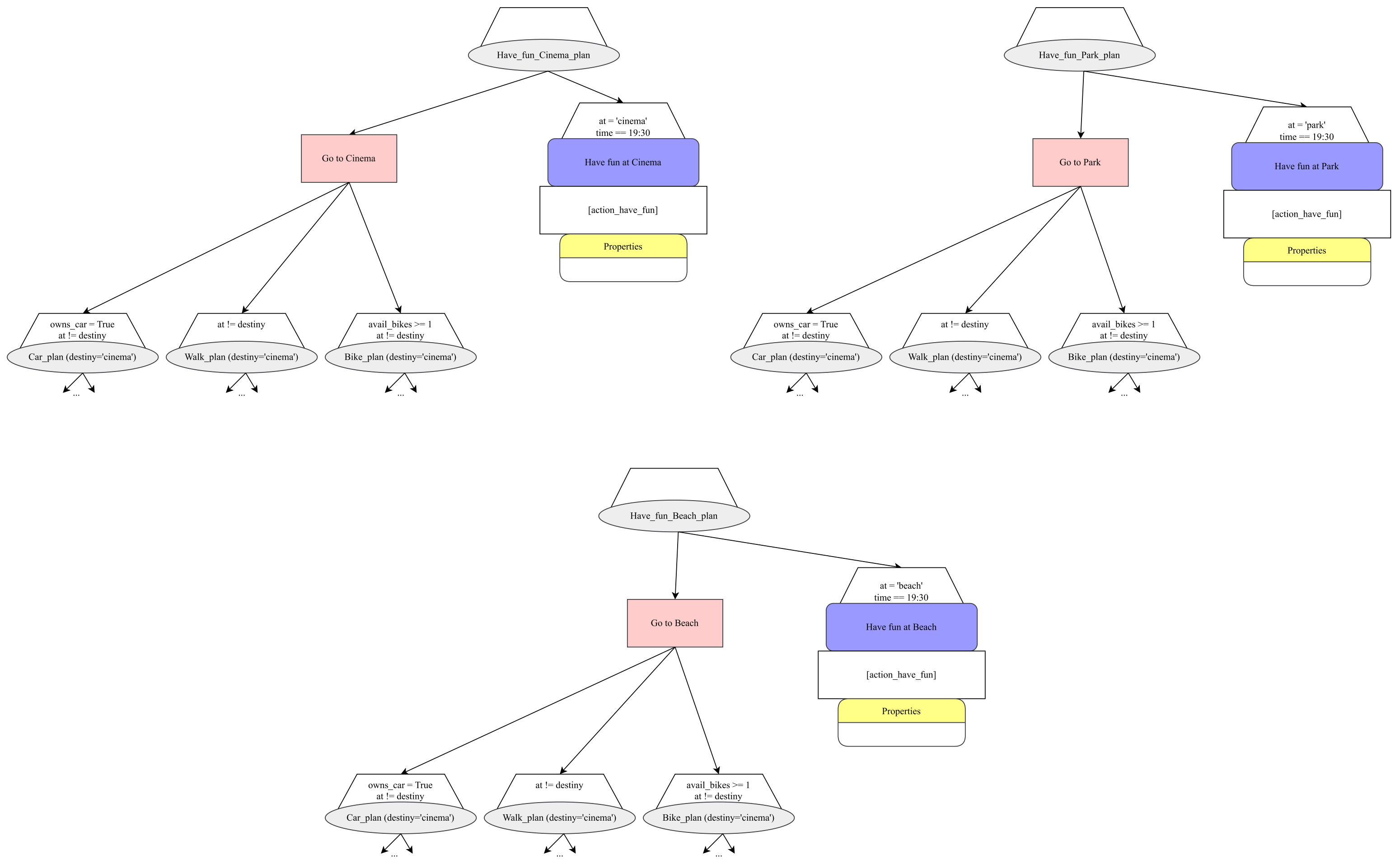


Figure 5.8: Library of plans for fun goals [Own making]

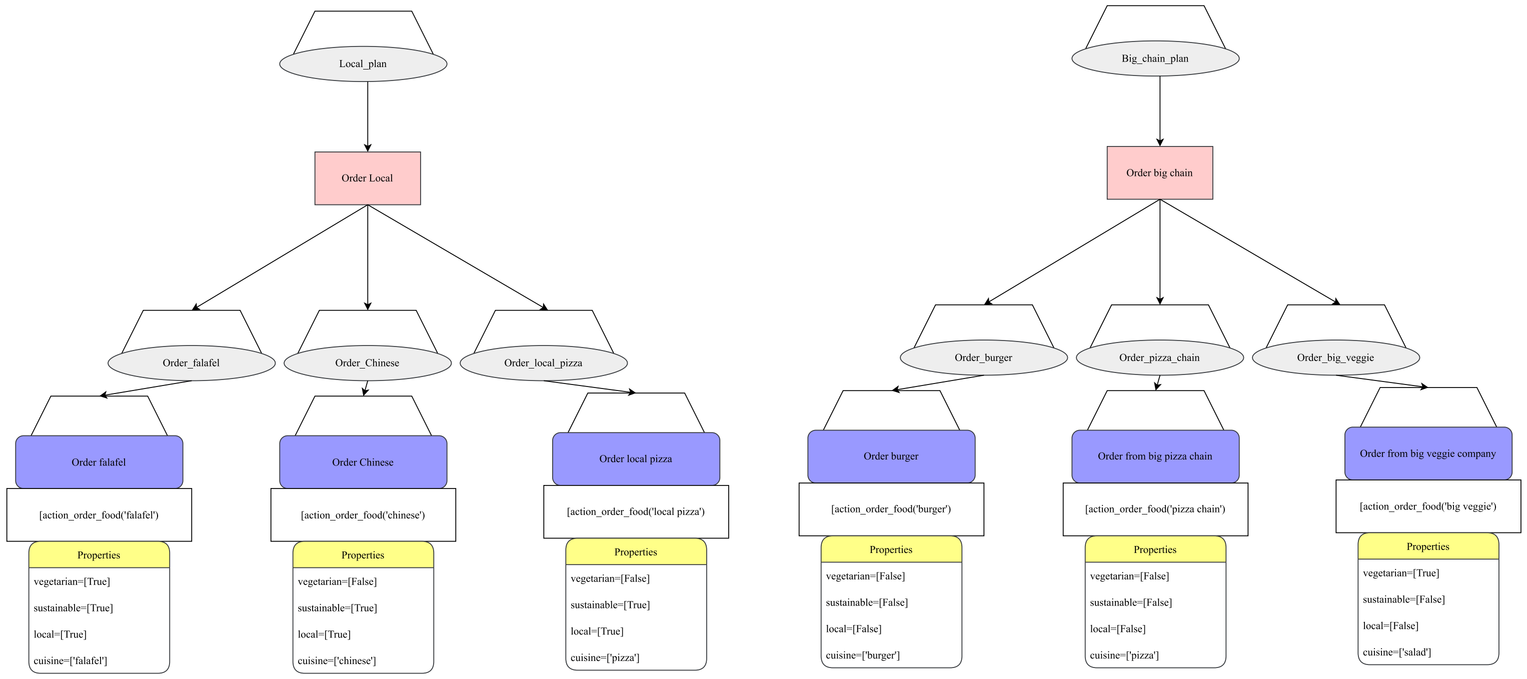


Figure 5.9: Library of plans for food goals [Own making]

The other plans for other goals are not included because they are trivial: they have a single plan, with a single action (e.g., in the case of the plan to work, there is only one method, with a single action). Finally, we make not that, for Alice, some of the internal actions have extra arguments dedicated to dealing with baeliefs about her kids, but they have not been included in the diagram because they have no effects in any of her preferences over goals, plans, or values.

From Alice, we expect her to, among other things:

- Attend any medical emergency, at any given moment
- Follow her goals in the right order and applying the right preferences
- Order pizza from the big pizza chain, unless it is raining
- Go by car anywhere
- Have fun in the beach by default, in the cinema if there is bad weather, and at the park if it is cloudy

5.3.1.2.2 Bob

Bob is the second agent we have created for this demonstration. Like Alice, he has his own set of beliefs, a place where he lives, a place where he goes to work, preferences over how to have fun, etc. Citizen Bob is described by the following:

- Bob is a worker in the local factory. He, therefore, works at the factory every day until 16:45
- Bob lives in the city center
- Bob has no children. Thus, he goes to work directly every morning. Then, once he is done, he goes to have fun in whatever way he prefers. Then, goes back home, and orders food at 21:00
- Her goals are the following:
  1. Go to work
  2. Work
  3. Have fun
  4. Go back home
  5. Eat dinner

6. Attend any medical emergency that might happen during the day

   - Goals no. 1, 3, and 4 include commuting
   - Goal no. 3 is Bob's fun goal
   - Goal no. 5 is Bob's food goal

- His preferences over goals are the following:
  - **Default**: $g_1 \rightarrow g_2 \rightarrow g_3 \rightarrow g_4 \rightarrow g_5$
  - Conditional preferences (if **medical emergency**): $g_6 \rightarrow g_1 \rightarrow g_2 \rightarrow g_3 \rightarrow g_4 \rightarrow g_5$
- His preferences over plans and moral values are the following:
  - For **transport goals**:
    - By default, he to take the bike, if possible
    - On top of that, he is an environmentalist
    - If it is cloudy, he likes to walk instead
    - And if the weather is bad, he takes the car
  - For **fun-related goals**:
    - Above all, and by default, if the weather is sunny, he likes to go to the beach
    - If the weather is not sunny, he prefers to go to the cinema instead
  - For **food-related goals**:
    - Above all, Bob likes pizza
    - Additionally, he believes in sustainability and in local businesses
- His initial beliefs are like Alice's (excluding children-related beliefs)

From Bob, we expect him to, among other things:

- Attend any medical emergency, at any given moment
- Follow his goals in the right order and applying the right preferences
- Order local pizza from the big pizza chain
- Go biking anywhere by default if he cans, but switch to walking if it is cloudy, and if the weather turns bad, to use the car

- Have fun in the beach by if it is sunny, and go to the cinema if the weather is not clear

- Additionally, from Both Alice and Bob, and this scenario in general, we expect to see how complex can the situations we simulate get without using numbers in our implementations of preferences over goals and over plans. This was one of the original objectives of the thesis, and we intend to put it to the test here.

If we notice, Bob's goals are a subset of Alice's goals. That is why Bob's plans have the same structure as Alice's, and he essentially has the same plans. This, however, does not mean that Bob will act like Alice, as his personal preferences and moral beliefs differ quite a bit from Alice's. Since Bob's plans are the same as Alice's, we do not include figures of them.

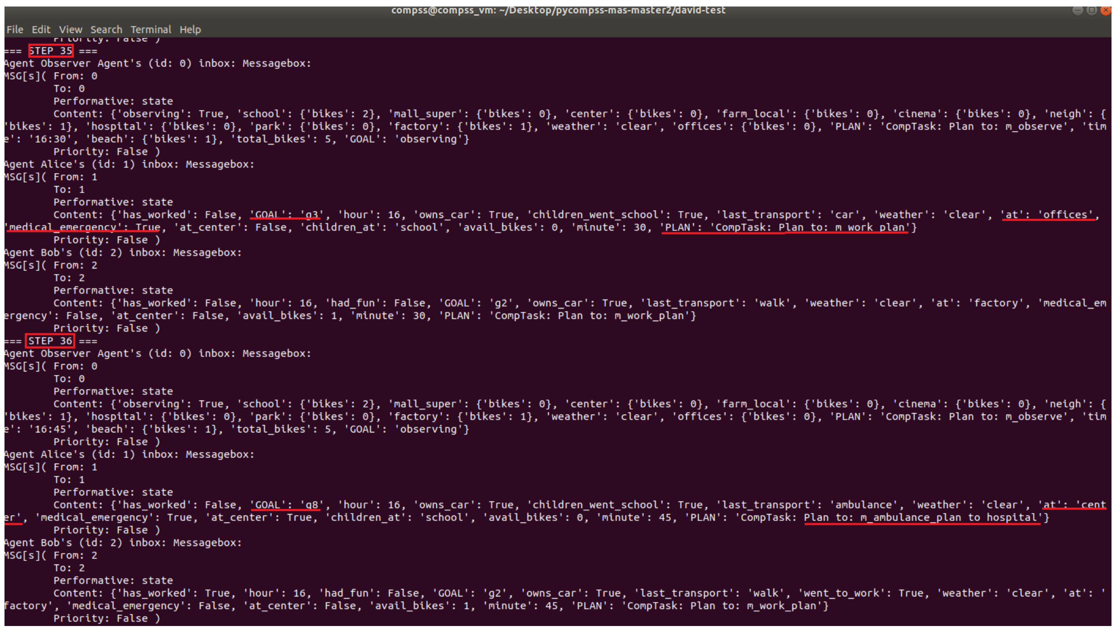
For more details in how either Alice or Bob are implemented, please refer to the delivered code, in file `ExampleBehaviors.py`, methods `init_alice(...)` and `init_bob(...)`.

5.3.2 Functionality-related results

For the functional part of this scenario, we will attempt to see that agents plan according to their preferences and values, and that they respond to changes in the environment that might cause them to reconsider their contextual preferences and, therefore, need to replan, or even reconsider their goals.

Let us first see a change in preferences *over goals*. Take a look at Figure 5.10. This is the result of a simulation with all default parameters except for `emergencyodds = 0.2` (20%). In it, we can see that Alice is working in her workplace, the offices, when he receives a medical emergency. Then, her conditional preferences over goals activate, she changes her current goal, and she rushes to the hospital, as we can see in the next step. Although not shown in the picture, when she goes to the hospital and is cured, her preferences over goals revert to default, and she goes back to the offices to continue working.

Now, let us see a change in preferences over plans in a action. Please refer to Figure 5.11. This is the result of a simulation with all default parameters except for `changeodds = 1`, `rainodds = clearodds = 0.5`, and `cloudoods = snowodds = 0`. As we can see, at step 43, both agents were having fun at the beach. However, it suddenly started to rain, and then their preferences over plans changed. Notice how the goal does not change, it remains the same, their goal to have fun. What changes, however, is *how* they decide to


compss@compss_vm: ~/Desktop/pycompss-mas-master2/david-test
File Edit View Search Terminal Help

```
        Priority: False )
=== STEP 35 ===
Agent Observer Agent's (id: 0) inbox: Messagebox:
MSG[s]( From: 0
        To: 0
        Performative: state
        Content: {'observing': True, 'school': {'bikes': 2}, 'mall_super': {'bikes': 0}, 'center': {'bikes': 0}, 'farm_local': {'bikes': 0}, 'cinema': {'bikes': 0}, 'neigh': {
'bikes': 1}, 'hospital': {'bikes': 0}, 'park': {'bikes': 0}, 'factory': {'bikes': 1}, 'weather': 'clear', 'offices': {'bikes': 0}, 'PLAN': 'CompTask: Plan to: m_observe', 'tim
e': '16:30', 'beach': {'bikes': 1}, 'total_bikes': 5, 'GOAL': 'observing'}
        Priority: False )
Agent Alice's (id: 1) inbox: Messagebox:
MSG[s]( From: 1
        To: 1
        Performative: state
        Content: {'has_worked': False, 'GOAL': 'g3', 'hour': 16, 'owns_car': True, 'children_went_school': True, 'last_transport': 'car', 'weather': 'clear', 'at': 'offices',
'medical_emergency': True, 'at_center': False, 'children_at': 'school', 'avail_bikes': 0, 'minute': 30, 'PLAN': 'CompTask: Plan to: m_work_plan'}
        Priority: False )
Agent Bob's (id: 2) inbox: Messagebox:
MSG[s]( From: 2
        To: 2
        Performative: state
        Content: {'has_worked': False, 'hour': 16, 'had_fun': False, 'GOAL': 'g2', 'owns_car': True, 'last_transport': 'walk', 'weather': 'clear', 'at': 'factory', 'medical_em
ergency': False, 'at_center': False, 'avail_bikes': 1, 'minute': 30, 'PLAN': 'CompTask: Plan to: m_work_plan'}
        Priority: False )
=== STEP 36 ===
Agent Observer Agent's (id: 0) inbox: Messagebox:
MSG[s]( From: 0
        To: 0
        Performative: state
        Content: {'observing': True, 'school': {'bikes': 2}, 'mall_super': {'bikes': 0}, 'center': {'bikes': 0}, 'farm_local': {'bikes': 0}, 'cinema': {'bikes': 0}, 'neigh': {
'bikes': 1}, 'hospital': {'bikes': 0}, 'park': {'bikes': 0}, 'factory': {'bikes': 1}, 'weather': 'clear', 'offices': {'bikes': 0}, 'PLAN': 'CompTask: Plan to: m_observe', 'tim
e': '16:45', 'beach': {'bikes': 1}, 'total_bikes': 5, 'GOAL': 'observing'}
        Priority: False )
Agent Alice's (id: 1) inbox: Messagebox:
MSG[s]( From: 1
        To: 1
        Performative: state
        Content: {'has_worked': False, 'GOAL': 'g8', 'hour': 16, 'owns_car': True, 'children_went_school': True, 'last_transport': 'ambulance', 'weather': 'clear', 'at': 'cent
er', 'medical_emergency': True, 'at_center': True, 'children_at': 'school', 'avail_bikes': 0, 'minute': 45, 'PLAN': 'CompTask: Plan to: m_ambulance_plan to hospital'}
        Priority: False )
Agent Bob's (id: 2) inbox: Messagebox:
MSG[s]( From: 2
        To: 2
        Performative: state
        Content: {'has_worked': True, 'hour': 16, 'had_fun': False, 'GOAL': 'g2', 'owns_car': True, 'last_transport': 'walk', 'went_to_work': True, 'weather': 'clear', 'at': '
factory', 'medical_emergency': False, 'at_center': False, 'avail_bikes': 1, 'minute': 45, 'PLAN': 'CompTask: Plan to: m_work_plan'}
        Priority: False )
```

Figure 5.10: Agent changing preferences over goals [Own making]

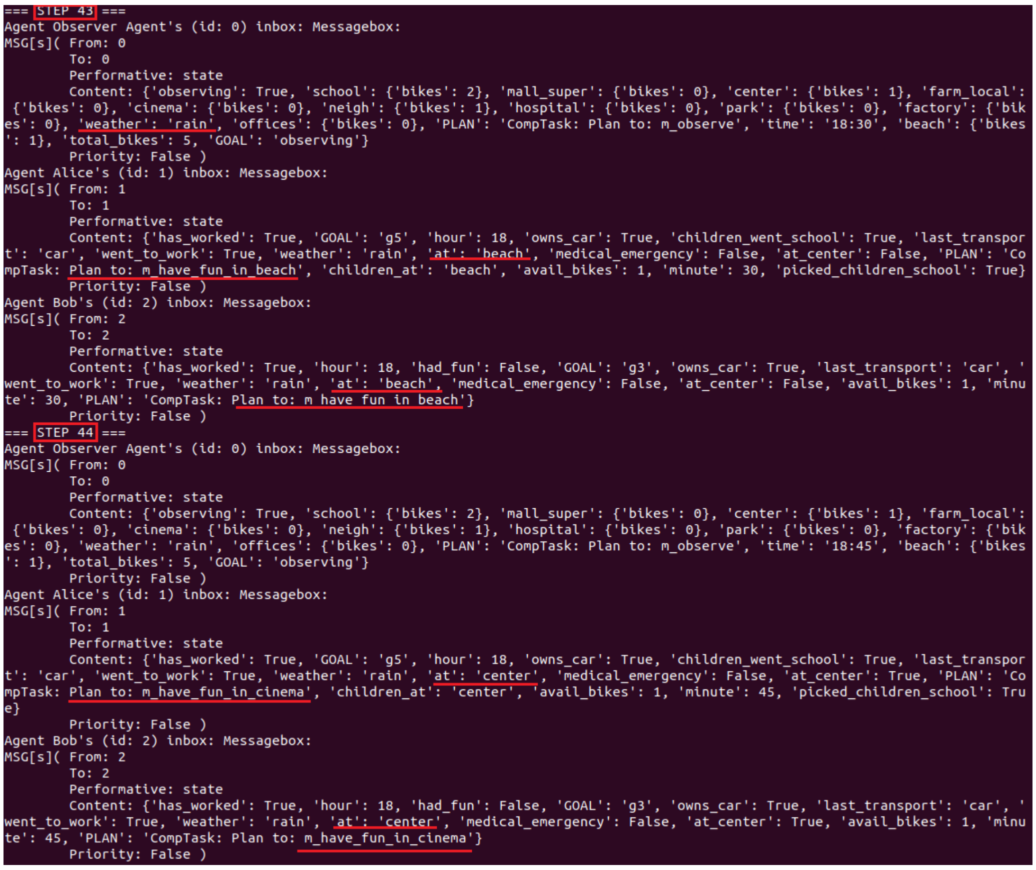
have fun. Under the previous conditions, sunny weather, they preferred to have fun by being at the beach. But when it started to rain, suddenly, while they still wanted to have fun, they did not want to have fun at the beach under those conditions (bad weather), and therefore replanned, and chose to have fun by going to the cinema instead.

For further examples of our features in action, please refer to Appendix C.

In general, we see that our agents react to changes in their current context (environment, circumstances), and they reflect that by changing their priorities, and always plan according to them. Additionally, by looking at the whole verbose dump of a simulation, we see that they function as expected: pursue their default goals in the correct order, change priorities over goals whenever they should, replan according to changes in both priorities over goals and plans, and make choices according to them, too.

Finally, we have also been able to see how 'far' we could go without using any numbers to express preferences over goals, plans, and moral values. As we have seen, we have been able to express conditional preferences over both, have these preferences change based on context, and agents replan based on changes of their environment.

Therefore, we conclude this test has been a success.



```
=== STEP 43 ===
Agent Observer Agent's (id: 0) inbox: Messagebox:
MSG[s]( From: 0
        To: 0
        Performative: state
        Content: {'observing': True, 'school': {'bikes': 2}, 'mall_super': {'bikes': 0}, 'center': {'bikes': 1}, 'farm_local':
 {'bikes': 0}, 'cinema': {'bikes': 0}, 'neigh': {'bikes': 1}, 'hospital': {'bikes': 0}, 'park': {'bikes': 0}, 'factory': {'bik
es': 0}, 'weather': 'rain', 'offices': {'bikes': 0}, 'PLAN': 'CompTask: Plan to: m_observe', 'time': '18:30', 'beach': {'bikes
': 1}, 'total_bikes': 5, 'GOAL': 'observing'}
        Priority: False )
Agent Alice's (id: 1) inbox: Messagebox:
MSG[s]( From: 1
        To: 1
        Performative: state
        Content: {'has_worked': True, 'GOAL': 'g5', 'hour': 18, 'owns_car': True, 'children_went_school': True, 'last_transpor
t': 'car', 'went_to_work': True, 'weather': 'rain', 'at': 'beach', 'medical_emergency': False, 'at_center': False, 'PLAN': 'Co
mpTask: Plan to: m_have_fun_in_beach', 'children_at': 'beach', 'avail_bikes': 1, 'minute': 30, 'picked_children_school': True}
        Priority: False )
Agent Bob's (id: 2) inbox: Messagebox:
MSG[s]( From: 2
        To: 2
        Performative: state
        Content: {'has_worked': True, 'hour': 18, 'had_fun': False, 'GOAL': 'g3', 'owns_car': True, 'last_transport': 'car', '
went_to_work': True, 'weather': 'rain', 'at': 'beach', 'medical_emergency': False, 'at_center': False, 'avail_bikes': 1, 'minu
te': 30, 'PLAN': 'CompTask: Plan to: m_have_fun_in_beach'}
        Priority: False )
=== STEP 44 ===
Agent Observer Agent's (id: 0) inbox: Messagebox:
MSG[s]( From: 0
        To: 0
        Performative: state
        Content: {'observing': True, 'school': {'bikes': 2}, 'mall_super': {'bikes': 0}, 'center': {'bikes': 1}, 'farm_local':
 {'bikes': 0}, 'cinema': {'bikes': 0}, 'neigh': {'bikes': 1}, 'hospital': {'bikes': 0}, 'park': {'bikes': 0}, 'factory': {'bik
es': 0}, 'weather': 'rain', 'offices': {'bikes': 0}, 'PLAN': 'CompTask: Plan to: m_observe', 'time': '18:45', 'beach': {'bikes
': 1}, 'total_bikes': 5, 'GOAL': 'observing'}
        Priority: False )
Agent Alice's (id: 1) inbox: Messagebox:
MSG[s]( From: 1
        To: 1
        Performative: state
        Content: {'has_worked': True, 'GOAL': 'g5', 'hour': 18, 'owns_car': True, 'children_went_school': True, 'last_transpor
t': 'car', 'went_to_work': True, 'weather': 'rain', 'at': 'center', 'medical_emergency': False, 'at_center': True, 'PLAN': 'Co
mpTask: Plan to: m_have_fun_in_cinema', 'children_at': 'center', 'avail_bikes': 1, 'minute': 45, 'picked_children_school': Tru
e}
        Priority: False )
Agent Bob's (id: 2) inbox: Messagebox:
MSG[s]( From: 2
        To: 2
        Performative: state
        Content: {'has_worked': True, 'hour': 18, 'had_fun': False, 'GOAL': 'g3', 'owns_car': True, 'last_transport': 'car', '
went_to_work': True, 'weather': 'rain', 'at': 'center', 'medical_emergency': False, 'at_center': True, 'avail_bikes': 1, 'minu
te': 45, 'PLAN': 'CompTask: Plan to: m_have_fun_in_cinema'}
        Priority: False )
```

Figure 5.11: Agent changing preferences over plans [Own making]

5.3.3 Performance-related results

No. of agents	Runtime (s)	Runtime per ag. (s)
Default (2)	8.11	4.055
4	10.79	2.6975
8	15.13	1.89125
16	23.84	1.49
32	43.8	1.36875
64	89.34	1.3959375
128	189.58	1.48109375

Table 5.3: Performance results of Goodsprings scenario [Own making]

Finally, for the performance version of this scenario, since we have no old version to compare it to, we will simply compute the performance metrics for our scenario, and see if the runtimes are acceptable, and the number of agents is scalable. Here is where we expect the most time-consuming results, because now agents have several goals, several preferences over goals, plans, and moral values, that they have to check at every step. Additionally, their set of beliefs, the environment they are in, and their plans, are all also more complex than in previous scenarios. Adding all that up, thus, we expect bigger runtimes.

We have, again, computed the runtime for the scenario with the default number of agents (3), 4, 8, 16, 32, 64, and 128, and all the same applicable metrics from the other scenarios. All this information is displayed in Table 5.3. For the extra agents, we have simply added copies of Alice and Bob, which is enough for the obtaining of performance-related metrics.

We can appreciate bigger runtimes for the same input size as in the previous performance tests. This is in line with what we expected and stated in the above paragraphs. Still, we have still managed to run a simulation with 128 agents in just 190 seconds. Taking into account that the simulation was 64 steps, this leaves us with a ratio of roughly 3 seconds per step, and at each step we have 128 agents running. For numbers of agents above or equal to 16, we get similar ratios as well. These figures are practical for real, complex situations. We also have to take into account that these results are from runs performed in a personal computer hosting a virtual machine, and would be much smaller if we were running them in an appropriate hardware oriented at running parallel simulations.

Chapter 6

Conclusions

In this work we have presented several improvements and additions to the platform presented in [11] and implemented in [10], focusing primarily on imbuing the platform's agents with social aspects such as preferences or moral values, as well as expanding other crucial aspects of it, like the addition of goals or the decoupling of plans from goals.

This last chapter works as a conclusion of all the work that has been done, to analyze how our initial goals have been addressed, and to lay out possible lines of future work and alternative ways in which our work could be improved.

This conclusion is structured as follows: in §6.1 we revisit the goals established in Chapter 1, assess to which degree they have been achieved, and how. Then, in §6.2, we discuss how this work contributes to the field of AI, and to the field of ABM in particular. Lastly, and to conclude the contents of this thesis, in section §6.3 we deliberate and examine several possible ways in which the work we have done could be expanded, complemented, improved and/or redesigned.

6.1 Revisiting objectives

The original list of objectives that we set before taking over this project were:

1. Regarding **goals**:
   (a) Adding an explicit structure that encodes goals in agents
   (b) Totally decoupling goals from plans, i.e., making plans and goals totally independent
   (c) Adding the capacity for agents to have many goals, not just one

2. Adding **preferences over goals**

3. Adding **preferences over plans, subplans, and actions**

4. Adding **values**

5. Make sure our additions are **scalable**

6. See **how 'far' we can go without using numbers** to encode preferences

Objective 1 has been fully covered, seeing as we added a structure that encoded goals for agents (1.a), have totally decoupled goals from plans (1.b), and our agents can have many independent goals (1.c). The formal definition of goals can be found in §3.1, and their implementation into the platform, in §4.1.

Objectives 2, 3, and 4, have also been totally achieved. Our agents can now express conditional preferences over the order in which to pursue their goals, conditional preferences over how to achieve their committed goal, and can endow their actions with moral values by using the same structures that are used to specify preferences over plans. The version of moral values that we have implemented is not necessarily constrained by moral objectivism, as different agents can associate the same actions with different moral values, as well as add different moral values to the same action done under different contexts. The formal definition of preferences over goals can be found in §3.2, and their implementation, in §4.2. The formal definition of preferences over plans can be found in §3.3, and their implementation, in §4.3. Finally, these additions were tested, and the tests we conducted are described, and their results, analyzed, in Chapter 5.

Objective 5 was also accomplished. This goal was attained by, on top of running functionality tests to verify that our implementations work as intended, also running performance test. The structure of these tests was also laid out, and they were all conducted throughout Chapter 5. After running these tests, we came to the conclusion that our additions scaled reasonably well with the number of agents, and that the richness they add to the agents reasoning capabilities was worth the slight decrease in runtime that they contributed to.

Finally, Objective 6 was also accomplished, and mainly demonstrated in §5.3. Here, we saw that we could express preferences over goals, plans, actions, and values without using numbers. Additionally, we also have been able to make these preferences conditional (i.e., context-dependent), and add as many conditional ones as desired. All that without using a single number in the back. The most apparent limitation that we have found when taking this approach is that agents can reason about whether they prefer X over Y (where X and Y can be goals, actions, etc.), bu they can only do so in *absolute* terms. In other words, an agent can express the statement "I prefer X over Y", but cannot add gradations or nuances to it, such as "I prefer X *twice* as much as I prefer Y", or "I prefer X over Y, by a margin of Z", they can only do so on absolute terms.

6.2 Main contributions of this work

This work adds to the field of artificial intelligence in different ways. In particular, its main additions are to the fields of MAS and ABM.

First of all, this work improves on an already-existing platform by endowing its agents with capabilities to reason over preferences over their goals, the actions they make, and also moral values. These additions have been focused on integration social aspects in the platform. Additionally, other improvements to the framework have also been made, such as introducing goals for agents, allowing them to have multiple goals, decoupling goals from plans, and having a library of plans as a way to reason on how to achieve said goals.

Additionally, we also had the goal of seeing how much we could accomplish, in terms of complex reasoning, by limiting our approaches on implementing preferences to be fully *qualitative* (i.e., not based on numbers). By doing that, we have seen the extent of social situations we are able to model, their complexities, and also have found some hard limitations, as stated in the last paragraph of §6.1.

Finally, as one of the main, foundational goals of the platform we took over was that it should be able to run *scalable* simulations of complex agents. Therefore, it was our objective too, to make sure our additions kept the system scalable. Thus, we have also contributed to the field of BDI-like agent micro-simulations on HPC, by making sure our implementations were scalable.

6.3 Possible lines of future work

Although our additions to the framework have been numerous, there are still many ways in which it could be enhanced and improved upon, both by adding completely new features, or by expanding or improving the already-existing ones. In this section, we present several possible lines of future work, classified by their nature.

6.3.1 Adding different types of goals

One of the main limitations of the system when it comes to goals is that it only really supports *maintain* goals (i.e., goals about *maintaining* some conditions about the state of the world), but these are not the only types of goals that exist. For instance, there are *perform* goals, which are goals whose main idea is that the plans associated to them should not necessarily reach their desired states, only that the plan should be executed, regardless of the results. In other words, if an agent acquires a maintain type of goal, he will draw a plan for the goal, execute the plan, and drop the goal after the plan has finished, regardless of whether the goal has been achieved or not.

Another type of goal worth mentioning is the *achieve* type of goal. These goals, when an agent acquires them, he draws a plan for the goal, and tries to achieve it. If the plan fails, the agent replans, and does so repeatedly until the goal has been achieved. Once that happens, the goal is dropped, and even if the conditions of the goal become false, the agent will not try to achieve it again.

Finally, agents could be expanded to be able to *commit* to more than one goal at once. This could perhaps be done through the addition of *compound* goals, which are the result of merging two separate goals into a single goal.

6.3.2 Improving preferences' trigger conditions

In the current version of the platform, the trigger conditions for both preferences over goals and preferences over actions are limited by a design choice: it tries to apply the preferences whose conditions apply, and it always starts checking by the preferences who have more conditions. Therefore, if preferences p_i have three assertions in their conditions, and preferences p_j have four assertions, and both trigger conditions are met, p_j will be selected. This becomes a problem in situations where we have more than one available preference, especially if their conditions have the same size, because all but one will be ignored.

This issue could be tackled in many ways. One of these ways would be to implement some sort of preferences merger method, which would take as input all the preferences that can be applied in a situation, and it will output a single set of preferences, product of merging all the preferences of the input. A different way in which this issue could be solved is by adding numbers to the way preferences are triggered.

6.3.3 Adding preferences over resource usage

Our current version of preferences over goals is limited in the way that the implementation of properties of goals we have made only let us assign discrete values to these properties. The full concept of properties of goals also includes resources. By expanding properties of goals to also include resources, and then expanding preferences to also be able to deal with resource usage, our agents could express preferences over plans of the type "I would rather not spend more than X dollars when attempting to achieve goal Y", which, as of right now, they cannot.

The best way to do this would simply be to implement the full concept of properties of goals and resource usage developed in [29].

6.3.4 Improving the efficiency on the evaluation of conditions for Goals and plans

In this thesis we have implemented the checking of which goals have been achieved in a given state of the world and which plans may be applied in a given state by performing a sequential check of their conditions against the belief base of the agent, and seeing whether the current assignment of variables is compatible with the assertions of the conditions. This is done sequentially, one condition after the other, for every goal, and for every plan, at each iteration.

One possible improvement to the way conditions are checked in our implementation could be to implement a version of Charles Forgy's Rete algorithm, creating a graph of partial activations of conditions for goals and plans. It would be interesting to investigate if the memory space requirements required by the Rete network (especially if implemented with some useful improvements such as hashed memories) would be compensated by the drastic time complexity reduction Rete provides.

6.3.5 Improving the monitoring capabilities of the framework

One of our original goals we had to drop was this one. The controller of the simulation could be expanded to better (i.e., more in detail) monitor the behavior of its agents. For instance, the controller could draw some performance-related metrics associated to every agent, it could track the state of agents and notify us when their preferences change, it could check the adherence of agents to simulation-wide policies, or it could add an explicability layer to the behavior of agents, that is, having a way to communicate with agents, where agents tell the controller *why* they performed X action over Y action, why their goals changed, etc.

6.3.6 Adding social norms

The use of social norms by BDI-like agents in social simulations is another addition which could complement very well the ones we have introduced. Norms are social rules that specify what should and should not be done, as well as the expected repercussions of different behaviors. In a complex setting with many options to achieve a desired state, checking the consequences of an agent's actions apropos of adherence (or non-adherence) to social norms may both add an extra layer to the reasoning of agents *and* also might reduce the size of the search space for farther planning, thus also helping to make the platform more scalable.

The most common implementations of norms include *obligations* (i.e., things that should be done) and *prohibitions* (i.e., things that should not be done). Common formalizations of computatitional norms tend to use **Deontic Logic** specifications whose influence on the agents' behaviours is formally defined by several options, including **possible worlds semantics**. This may require a modification of the language for conditions we introduced in this work to be able to model rich norm-based specifications in the platform.

This addition of social norms would also require agents to be expanded, in order to add different kinds of agents, classified by their both their *awareness* and their *compliance* with social norms. The former classification includes agents that are *norm-aware* and agents that are not, that is, agents that know that norms exist, and agents that do not. The latter classification includes agents that always try to follow the norms, agents that may sometimes not follow the norms, and even agents that try *not* to follow (i.e., break) the norms whenever possible. This would be very relevant for our platform, as the way social norms influence the agents preferences will be different depending on the level of norm awareness and norm compliance for each of the agents.

Bibliography

[1] Rosa M. Badia, Javier Conejero, Carlos Diaz, Jorge Ejarque, Daniele Lezzi, Francesc Lordan, Cristian Ramon-Cortes, and Raul Sirvent. Comp superscalar, an interoperable programming framework. *SoftwareX*, 3-4:32–36, 12 2015.

[2] Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. Jade - a fipa-compliant agent framework.

[3] Rafael H. Bordini and Jomi Fred Hübner. *Programming Multi-Agent Systems in AgentSpeak using Jason.* 2007.

[4] Craig Boutilier, Ronen I Brafman, Carmel Domshlak, Holger H Hoos, and David Poole. Cp-nets: A tool for representing and reasoning with conditional ceteris paribus preference statements, 2004.

[5] Michael Bratman. Intention, plans, and practical reason, 1987.

[6] Claudio Cioffi, Keith M Sullivan, Gabriel Catalin Balan, Sean Luke, Claudio Cioffi-Revilla, Liviu Panait, Keith Sullivan, and Gabriel Balan. Mason: A multiagent simulation environment. agent-based modeling of complex crises view project modeling the origins of conflict in east africa view project mason: A multi-agent simulation environment, 2014. URL: `http://cs.gmu.edu/eclab/projects/mason/`.

[7] Stephen Cranefield, Michael Winikoff, Virginia TU Dignum Delft MVDignum, and tudelftnl Frank Dignum. No pizza for you: Value-based plan selection in bdi agents, 2017.

[8] Mehdi Dastani. 2apl: A practical agent programming language. *Autonomous Agents and Multi-Agent Systems*, 16:214–248, 2008.

[9] Luciano Ferraro. Github project - regoap, 2016. URL: `https://github.com/luxkun/ReGoap`.

[10] Dmitry Gnatyshak. Adapting the smart python agent development environment for parallel computing, 2019. URL: `https://upcommons.upc.edu/bitstream/handle/2117/133128/139210.pdf?sequence=1`.

[11] Dmitry Gnatyshak, Luis Oliva-felipe, Sergio Ávarez Napagao, Julian Padget, Dario Garcia-Gasulla, Ulises Cortés, and Javier Vázquez-Salceda. Towards a goal-oriented agent-based simulation framework for high-performance computing, 2019. URL: `https://upcommons.upc.edu/bitstream/handle/2117/180887/high\_performance\_computing.pdf;sequence=1`.

[12] Koen V. Hindriks and Tijmen Roberti. Goal as a planning formalism. volume 5774 LNAI, pages 29–40, 2009.

[13] Erol Kutluhan, James Hendler, and Dana S. Nau. Htn planning: Complexity and expressivity. 1994.

[14] Winfried Lamersdorf, L Braubach, A Pokahr, and W Lamersdorf. Jadex: A short overview middleware view project complex objects view project jadex: A short overview, 2011. URL: `https://www.researchgate.net/publication/228981380`.

[15] Brian G Milnes, Garrett Pelton, Robert Doorenbos, Mike Hucka, John Laird, Paul Rosenbloom, and Allen Newell. A specification of the soar cognitive architecture in z, 1992.

[16] Ingrid Nunes. Github project - bdi4jade, 2010. URL: `https://github.com/ingridnunes/bdi4jade`.

[17] Ingrid Nunes, Carlos J P De Lucena, and Michael Luck. Bdi4jade: a bdi layer on top of jade, 2011.

[18] Jeff Orkin. Website - goal-oriented action planning (goap). URL: `https://alumni.media.mit.edu/~jorkin/goap.html`.

[19] Jeff Orkin. Symbolic representation of game world state: Toward real-time planning in games, 2004. URL: `https://alumni.media.mit.edu/~jorkin/WS404OrkinJ.pdf`.

[20] Jeff Orkin. Agent architecture considerations for real-time planning in games, 2005. URL: `https://alumni.media.mit.edu/~jorkin/aiide05OrkinJ.pdf`.

[21] Jeff Orkin. Three states and a plan: The a.i. of f.e.a.r. *M.I.T. Media Lab, Cognitive Machines Group*, 2006. URL: `https://alumni.media.mit.edu/~jorkin/gdc2006\_orkin\_jeff\_fear.pdf`.

[22] Mostafa Mohajeri Parizi, Giovanni Sileno, and Tom van Engers. Declarative preferences in reactive bdi agents. volume 12568 LNAI, pages 215–230. Springer Science and Business Media Deutschland GmbH, 2021.

[23] Stuart Russell and Peter Norvig. *Artificial Intelligence A Modern Approach Fourth Edition.* 1995.

[24] Shalom H. Schwartz. An overview of the schwartz theory of basic values. *Online Readings in Psychology and Culture*, 2, 12 2012.

[25] Bram Stolk. Github project - general purpose goap (gpgoap), 2012. URL: `https://github.com/stolk/GPGOAP`.

[26] Enric Tejedor, Yolanda Becerra, Guillem Alomar, Anna Queralt, Rosa M. Badia, Jordi Torres, Toni Cortes, and Jesús Labarta. Pycompss: Parallel computational workflows in python. *International Journal of High Performance Computing Applications*, 31:66–82, 1 2017.

[27] Seth Tisue and Uri Wilensky. Netlogo: A simple environment for modeling complexity, 2004.

[28] Simeon Visser, John Thangarajah, and James Harland. Reasoning about preferences in intelligent agent systems.

[29] Simeon Visser, John Thangarajah, James Harland, and Frank Dignum. Preference-based reasoning in bdi agent systems. *Autonomous Agents and Multi-Agent Systems*, 30:291–330, 3 2016.

[30] Michael J. Wooldridge. *An introduction to multiagent systems.* John Wiley & Sons, 2009.

[31] Kashif Zia, Andreas Riener, Katayoun Farrahi, and Alois Ferscha. A new opportunity to urban evacuation analysis: Very large scale simulations of social agent systems in repast hpc, 2012.

Appendix A

Project Planning

This chapter is about everything related to the planning and management of this project: laying the objectives, their requirements, breaking down into smaller tasks, the resources we will require, how we will carry out the agile planning and iterations, etc.

The project will take approximately 545 hours over 128 days. The work will take place between the 21st of February and the 28th of June (date of the project's oral defense), hence, 128 days. It is estimated that this project will take, according to the numbers above, an average of 4.26 hours per day.

A.1 Description of tasks

In this section, we will be covering all the tasks in which the project will be divided, defining them, giving time estimates for each tasks, establishing and justifying their dependencies, as well as stating the resources we will require in order to accomplish the tasks. We will also provide a table as a means of summary at the end of this section.

A.1.1 Definition and time estimates

Firstly, we need to strictly define and delimit what our tasks will be. They need to be as atomic as possible, so as not to overload tasks with an excessive number of actions or things that could be done better separately.

We will begin by stating the different kinds of tasks that we will have. These are **project management tasks**, **research tasks**, and **technical tasks**. Project management tasks will deal with everything related to the correct development of the thesis: planning, deciding a budget, setting deadlines, agile iterations, managing obstacles, re-planning, etc. Research tasks will involve searching for information, exploring the state of the art of MAS and reasoning models, reading scientific papers and books, etc. And finally, technical tasks will cover the implementations of all our ideas into code as well as executing them: the enhancing of the monitoring system, the improvement of the agents' reasoning capabilities, running the code in the supercomputer, etc.

Project management tasks
Project management tasks are of vital importance to the thesis's correct development. Without them, we would not be able to meet the deadlines, know when a task is finished, have a correct structure for the project, and we would not be able to keep correct track of our progress. Therefore, we need to be very careful when planning them. We have differentiated the following project management tasks:

- **Contextualization and scope:** We must state the project's overall goal(s), contextualize it, and justify why this particular topic matter was chosen.

- **Project planning:** This is a "meta-task", as it involves planning all the other tasks. It is crucial to know which tasks are more important, to meet deadlines, and to organize the project properly.

- **Budget and sustainability:** It is critical to understand the entire cost of a project as well as the influence that said cost will have on its progress. As a result, this work focuses on creating a budget and assessing the project's long-term viability.

- **Meetings:** Meetings with the project's director are arranged every two weeks as per our agile methodology. We'll talk about the current situation and the next steps to take. Due to the possibility of an unexpected need of extraordinary meetings, this task will be added extra time.

- **Final version of the document:** This task will deal with listing all the work we have actually done, clearly stating which objectives have been met, which have not been possible to meet (hopefully, none), and to set the final version of the scope.

- **Writing of the document:** This task is the actual writing of the thesis's document: putting into words and paragraphs all the work and findings we have done, as well as documenting all the process.

- **Preparing the thesis's defence:** We will have to rehearse the defence of this thesis, prepare questions in advance, set up some slides to smoothly present our ideas, etc.

Research tasks
This project features a significant amount of research. As a result, prior to beginning the experimental phase, it is necessary to do research into previous studies in order to see what has been done previously and recently in the related fields.

On the research part of this thesis, we will focus on reading research papers and books related to agents and MAS with the objective of finding ways to complement and enhance the agents' current reasoning mechanisms. Additionally, we will also be looking into similar platforms and frameworks in order to see what features they included in their monitoring systems with the intent to find additional functionalities that would go very well with our system. This part has been divided into the following tasks:

- **Research to improve the agents' reasoning abilities:** Explore the state of the art of agents and MAS in order to find different ways in which we can viably enhance our agents' reasoning capabilities. The exact ways or techniques that will be researched have not yet been defined, as that is also included in the purpose of the task, since this part of the thesis is about research.

- **Research to improve the systems' monitoring capabilities:** Explore methods on behaviour monitoring in agent-based social simulation platforms. This task was included in our original planning but, as we have explained in §1.1, in the first stages of project development it was decided to drop all tasks related to behaviour monitoring to focus on a richer agent reasoning cycle.

- **Other research:** As we progress in our search for information regarding the topics of agents and MAS, we may stumble upon concepts, constructs, or ideas that may fit well into our project but that we hadn't thought of before. This task will be about looking into all other interesting (and worth spending time into) things that we might come across as we develop this project.

Technical tasks
Once we have done sufficient research on different reasoning techniques and possible ways to enhance our platform's monitoring system, we will have to attempt to implement the features we have chosen. The technical part of this thesis has been subdivided among the following tasks:

- **Assess the requirements to implement each feature:** For each feature we decide to implement, we will need to elaborate a list of needs that we will require to implement them: code libraries, minimum hardware specs, extra necessary knowledge, etc.

- **Implement the improvements for the agents:** Add the necessary enhancements to the agents' code to include the improvements we have researched. Since we are working using an agile methodology, all these improvements will be reviewed every two weeks, and they may be further modified or even outright deleted if better and non-compatible alternatives are found later in time.

- **Implement the improvements for monitoring:** Add the necessary enhancements to the platform's monitoring systems to add the features we have found. As discussed in §1.1, this task was dropped to concentrate efforts in the improvements on the agents' reasoning cycle.

- **Validate the results:** After we carry out the implementation of any feature, we will have to test it to see how it affects the system. We will have to develop different, independent tests, to see if the agents reason better (or just differently), as well as some performance tests to see if the changes are worth keeping or if the improvements they provide come at a performance cost that is too high to be acceptable.

A.1.2 Sequence of tasks and dependencies

We have a set of several tasks that we have to accomplish in order to successfully conduct this thesis. Now, we need to add time estimates to each task, state what resources each will need, and set the dependencies between the tasks.

First, we will need to organize the tasks into groups, for better management. Some groups will be the same as mentioned before (management, research, etc.), but additional groups will need to be created for better clarity. We have devised the following groups:

- **T1: Project Management:** This group comprises all tasks in the **project management** group from §A.1.1, except for the last two: document writing and thesis defence preparation, as these two tasks will have to be done at the end of the project.

- **T2: Research:** This group comprises all tasks in the **research** group from §A.1.1.

- **T3: Technical:** This group comprises all tasks in the **technical** group from §A.1.1.
- **T4: Documentation:** This task is the actual writing of the document and all the findings and conclusions we produce.
- **T5: Oral defence:** This task is the preparation of slides, speech, and everything related to this thesis's defence.

A.2 Resources required

The purpose of this section is to list all the resources that are available to us, as well as which resource each task will need. We have divided all available resources between the following categories: software (SW), hardware (HW), human, and material.

SW resources group everything that is a program which we will be employing. These are: the Overleaf editor, Google Calendar and Google Meet, the repository to store and edit the code, an IDE and code editor, the Python programming language, a Gannt chart editor, and the previous code for the framework and the platform adaptation made by Dmitry.

HW resources group everything that is a computer or hardware which we will be using:

- **Microsoft Surface Pro 7 laptop:** 8 GB RAM, Intel(R) Core(TM) i5-1035G4 CPU @ 1.10GHz (8 cores).
- **ASUS desktop computer:** 16 GB RAM, Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz.
- **MareNostrum 4:** We have 256 processors (2048 cores) of the supercomputer at our disposal to run benchmarks and other tests. These are approximate numbers that are subject to change due to availability reasons.

Human resources are the human capital which will be involved in this project. This is me (the author), who will have to write, plan, and execute this project. Then we have the director, Javier, who will conduct this project and direct my efforts towards meaningful goals, as well as Dmitry and Sergio, who will play similar roles. Finally, we have the GEP tutor, Joan, who will oversee the management of this project and correct all deficiencies he detects in this process.

Material resources refer to books, scientific papers, articles, etc., that we will employ in order to broaden our knowledge and to get ideas from.

A summary of everything stated here, plus an initial estimate of hours for every task, can be consulted in Table A.1.

However, when we finished the project, we realized we had invested more hours than stated. Therefore, we provide an updated table with the real hours that have been worked in this project in Table A.2.

ID	Name of task	Time (h)	Dependencies	Resources required
T1	Project Management	**90**		
T1.1	Contextualization and scope	25		PC, Overleaf, GEP notes, D&R
T1.2	Project planning	10		PC, Overleaf, GEP notes
T1.3	Budget and sustainability	20	T1.2	PC, Overleaf, GEP notes
T1.4	Meetings	20		PC, Google Meet, D&R
T1.5	Final version of the document	15	T1.1, T1.2, T1.3	PC, Overleaf, GEP notes, D&R
T2	Research part	**110**		
T2.1	Research to improve agents	50		PC, papers, books
T2.2	Research to improve monitoring	50		PC, papers, books
T2.3	Other research	10		PC, papers, books
T3	Technical part	**150**		
T3.1	Assess the requirements of each feature	20	T2.1, T2.2	PC, papers, books
T3.2	Implement the improvements for the agents	55	T3.1	PC, papers, books, code, IDE
T3.3	Implement the improvements for monitoring	55	T3.1	PC, papers, books, code, IDE
T3.4	Validate the results	20	T3.2, T3.3	PC, D&R, supercomputer
T4	Documentation	**90**	T1.5, T3.*	PC, results obtained, Overleaf, code, D&R
T5	Oral defence preparation	**20**	T4	PC, Office 365, results obtained, code
Total		**460**		

Table A.1: Initial summary of all tasks [Own making]

By inspecting Table A.1, we draw the (one of the many) critical path: $T1.1 \rightarrow T1.2 \rightarrow T1.3 \rightarrow T1.5 \rightarrow T2.1 \rightarrow T2.2 \rightarrow T3.1 \rightarrow T3.2 \rightarrow T3.3 \rightarrow T3.4 \rightarrow T1.4 \rightarrow T5$

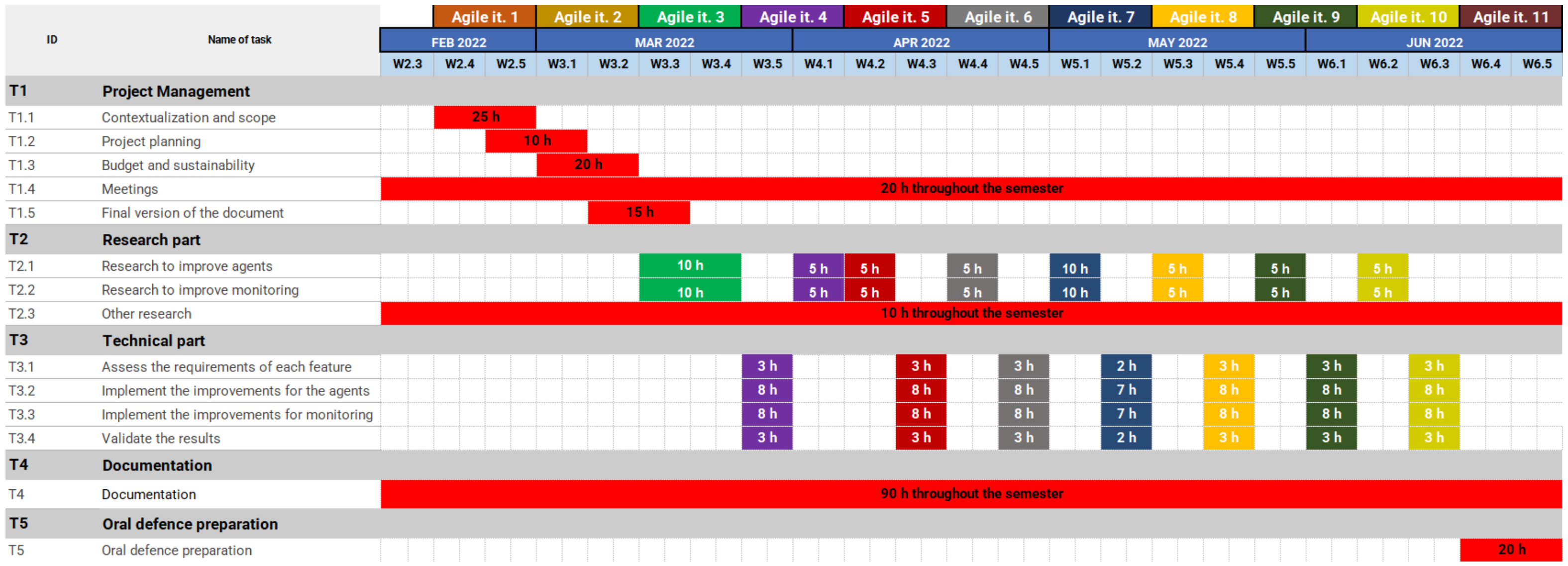
Which would take 430 hours, roughly the whole project. This project seems to be very linear, because the final tasks, the ones about unifying all the documentation, require all the research into the agents and the monitoring system to be finished beforehand. If we were in a situation that required to cut one of these two research options, the critical path would be reduced to 325 hours, assuming no extra hours.

In Table A.2 we can see that actually the tasks related to behaviour monitoring have dissapeared, but that effort was moved to other tasks (mainly T2.1 and T3,1, but also T2.3 and T3.4).

ID	Name of task	Time (h)	Dependencies	Resources required
T1	Project Management	**90**		
T1.1	Contextualization and scope	25		PC, Overleaf, GEP notes, D&R
T1.2	Project planning	10		PC, Overleaf, GEP notes
T1.3	Budget and sustainability	20	T1.2	PC, Overleaf, GEP notes
T1.4	Meetings	20		PC, Google Meet, D&R
T1.5	Final version of the document	15	T1.1, T1.2, T1.3	PC, Overleaf, GEP notes, D&R
T2	Research part	**150**		
T2.1	Research to improve agents	135		PC, papers, books
T2.3	Other research	15		PC, papers, books
T3	Technical part	**175**		
T3.1	Assess the requirements of each feature	25	T2.1, T2.2	PC, papers, books
T3.2	Implement the improvements for the agents	120	T3.1	PC, papers, books, code, IDE
T3.4	Validate the results	30	T3.2, T3.3	PC, D&R, supercomputer
T4	Documentation	**110**	T1.5, T3.*	PC, results obtained, Overleaf, code, D&R
T5	Oral defence preparation	**20**	T4	PC, Office 365, results obtained, code
Total		**545**		

Table A.2: Updated summary of all tasks [Own making]

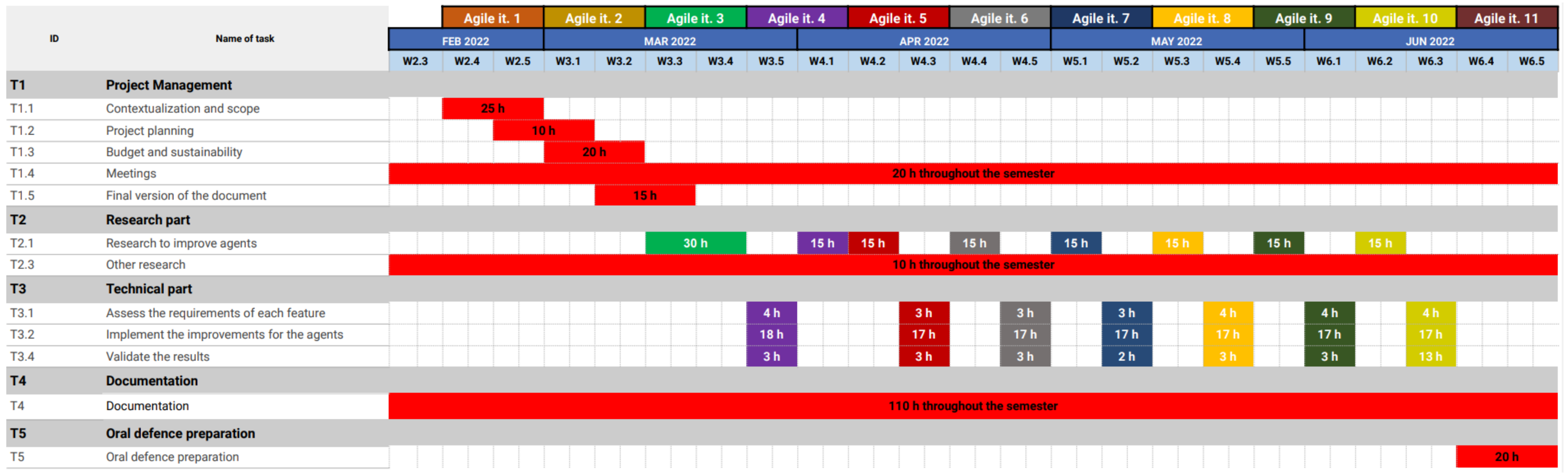
A.3 Initial Gantt chart



ID	Name of task	W2.3	W2.4	W2.5	W3.1	W3.2	W3.3	W3.4	W3.5	W4.1	W4.2	W4.3	W4.4	W4.5	W5.1	W5.2	W5.3	W5.4	W5.5	W6.1	W6.2	W6.3	W6.4	W6.5
			Agile it. 1		Agile it. 2		Agile it. 3		Agile it. 4		Agile it. 5		Agile it. 6		Agile it. 7		Agile it. 8		Agile it. 9		Agile it. 10		Agile it. 11	
		FEB 2022			MAR 2022					APR 2022					MAY 2022					JUN 2022				
T1	**Project Management**																							
T1.1	Contextualization and scope		25 h																					
T1.2	Project planning			10 h																				
T1.3	Budget and sustainability				20 h																			
T1.4	Meetings	20 h throughout the semester																						
T1.5	Final version of the document					15 h																		
T2	**Research part**																							
T2.1	Research to improve agents						10 h			5 h	5 h		5 h		10 h		5 h		5 h		5 h			
T2.2	Research to improve monitoring						10 h			5 h	5 h		5 h		10 h		5 h		5 h		5 h			
T2.3	Other research	10 h throughout the semester																						
T3	**Technical part**																							
T3.1	Assess the requirements of each feature								3 h			3 h		3 h		2 h		3 h		3 h		3 h		
T3.2	Implement the improvements for the agents								8 h			8 h		8 h		7 h		8 h		8 h		8 h		
T3.3	Implement the improvements for monitoring								8 h			8 h		8 h		7 h		8 h		8 h		8 h		
T3.4	Validate the results								3 h			3 h		3 h		2 h		3 h		3 h		3 h		
T4	**Documentation**																							
T4	Documentation	90 h throughout the semester																						
T5	**Oral defence preparation**																							
T5	Oral defence preparation																							20 h

Figure A.1: Initial Gantt chart for this project [Own making]

A.4 Updated Gantt chart



ID	Name of task		Agile it. 1		Agile it. 2		Agile it. 3		Agile it. 4		Agile it. 5		Agile it. 6		Agile it. 7		Agile it. 8		Agile it. 9		Agile it. 10		Agile it. 11	
		FEB 2022			MAR 2022					APR 2022					MAY 2022					JUN 2022				
		W2.3	W2.4	W2.5	W3.1	W3.2	W3.3	W3.4	W3.5	W4.1	W4.2	W4.3	W4.4	W4.5	W5.1	W5.2	W5.3	W5.4	W5.5	W6.1	W6.2	W6.3	W6.4	W6.5
T1	**Project Management**																							
T1.1	Contextualization and scope		25 h																					
T1.2	Project planning			10 h																				
T1.3	Budget and sustainability				20 h																			
T1.4	Meetings	20 h throughout the semester																						
T1.5	Final version of the document					15 h																		
T2	**Research part**																							
T2.1	Research to improve agents						30 h			15 h	15 h		15 h		15 h		15 h		15 h		15 h			
T2.3	Other research	10 h throughout the semester																						
T3	**Technical part**																							
T3.1	Assess the requirements of each feature								4 h			3 h		3 h		3 h		4 h		4 h		4 h		
T3.2	Implement the improvements for the agents								18 h			17 h		17 h		17 h		17 h		17 h		17 h		
T3.4	Validate the results								3 h			3 h		3 h		2 h		3 h		3 h		13 h		
T4	**Documentation**																							
T4	Documentation	110 h throughout the semester																						
T5	**Oral defence preparation**																							
T5	Oral defence preparation																							20 h	

Figure A.2: Updated Gantt chart for this project, taking into account the actual work done [Own making]

A.5 Risk management: alternative plans and obstacles

Throughout the whole life cycle of this project, some obstacles may arise and threaten to delay it, or even stop its progress. These potential risks are listed below, and we will assign a risk level to them, according to both their likelihood to materialize and how disruptive they would be if they occurred:

- **Delays in the deadlines [high chance] [medium impact]:** This risk can be caused by many factors: an underestimation of the workload, interference with exams, simply lack of time, getting stuck on a part of the project, etc. It is very likely that some internal deadlines will not be met, however, that could also be corrected internally and not end up affecting the whole project. If this risk were to materialize, the alternative action would be to re-plan the rest of the project with haste. This alternative task does not require any additional physical resources. In fact, the only additional resource we would need is extra time, not in the form of more days to work, but in the form of needing to work more hours per day. This obstacle would mostly affect tasks in the groups T4 and T5.

  If it was not solved, it could heavily affect the duration of the project by extending it, but we will take every precaution and action necessary to prevent this from happening.

- **Lack of familiarity with the platform [high chance] [medium impact]:** This risk is also highly likely to happen, as I, the author, have never employed this project's platform before. Its impact is still somewhat limited, as the solution is very similar to that of the risk above: re-schedule and re-plan in order to get back on schedule. Again, the only necessary extra resource we would need is time, and also in the form of working more hours per day (not extra days). This obstacle would mostly affect tasks in the group T3.

  If this obstacle was not solved, it would also heavily affect the duration of the project by extending it, and we will take all the necessary actions to avoid that as well.

- **Very time-inefficient simulation results [medium chance] [high impact]:** It may be the case that every new addition to the platform we produce makes running a multi-agent simulation on it unfeasible. While the chance of this risk materializing is not very high, its impact would indeed be: because it would mean we would not have improved the platform at all. If this were to happen, hopefully among the early stages of development, the only correcting actions available would be to either reduce the number of agents or, more in line with our objectives, invest more hours into this task to find ways to optimize the code. The only necessary resources here would be, once again, more working hours in order to produce better code, and in case that was not enough, we would need to make use of the MareNostrum to run the agents at much higher speeds and therefore allow more complex simulations to run smoothly (thus "hiding" or "disguising" the inefficiencies in the code). This obstacle would mostly affect tasks in the groups T3.

If this obstacle was not solved, it could also affect the duration of the project, but it would most likely affect the quality of the final results. It is not likely that this happens, but in the worst-case scenario (we produce inefficient code and we are unable to fix it), it would result in the quality of the end product being lower, i.e, we would deliver the project, and we would state that we have not been able to produce a more efficient version of it.

The task that would most effectively deal with all these risks would be, apart from the research and technical tasks, the meetings tasks, as with them we could meet with the researchers in an effort to make use of their hindsights and expertise on the field to correct the deficiencies as rapidly as possible.

Appendix B

Budget analysis and Sustainability

The purpose of this annex is to provide a budget estimate for this project, outlining all personnel costs per activity, generic costs, and other costs such as incidents, and compare them with a cost analysis after the end of the project. Furthermore, management control methods will be defined to control deviations that may occur due to unanticipated difficulties or obstacles. Finally, aspects regarding the sustainability of the project will be covered.

B.1 Budget

B.1.1 Staff costs

Here we will provide a full list of all the roles that a project of this magnitude needs in order to operate normally, as well as who will be covering those roles. Additionally, we will relate each employee position with the tasks defined in Appendix A, and how much it would cost per task and per employee.

In this project, we will distinguish between five different roles. Firstly, the **project manager** will be tasked with the correct planning and natural development of the project, as well as correcting the possible deviations that might occur. This role will be played mainly by me, the author, but also partly by the GEP tutor and the director of the project. Then, we have the positions of the **researcher**, whose main goal is to explore the state of the art on agents and to find different ways in which we can achieve the goals we set in Chapter 1. This role will be mainly carried out by me, but some help will be provided by the directors in case it were needed. The tasks regarding programming and adding new features to the platform, and testing

Role	**Cost (€/h)**
Project manager	25
Researcher	27
Programmer	20
Tester	15
Technical writer	18

Table B.1: Average hourly wages per position [Own making]

and verifying the results, will be carried out by the **programmer** and the **tester**, respectively. Both roles will be played mainly by me as well. Finally, we will need to document all the results we produce and the findings we discover, and this will be done by the **technical writer**, a role which will also be played by me, the author.

Now, we will try to estimate the average economic compensation that these positions would be receiving in a real project. The results have all been extracted from the *Glassdoor* online portal, a website which allows to compute the average salary for a position in a geographical area. The results obtained can be checked in Table B.1. These hourly wages are gross, that is, they already take into consideration the social charges and taxes in Spain. From now on, when we speak about the total cost of a worker, we will be referring to both their net salary, the taxes they pay in the form of personal income, and the taxes that we have to pay as their employers.

Now, according to the hours destined for each task, in Table B.2 can be seen the exact number of hours that each employee will be working and, thus, how much it would cost to compensate them economically for their work. In the table we have all the relevant information regarding cost per activity: how many hours each staff will need to invest in the project, as well as the total cost per task, which would be (without taking into account corrections) of 14020€. Again, it should be noted that this figure already includes taxation and social charges, since the hourly rates compiled in Table B.1 take into consideration net perceived salary, personal income tax, and the taxes that employers have to pay in Spain.

However, when actually doing the project, we had to update the numbers of hours, as well as divert the hours of research into monitoring in favor of research into improving the agents. All these new changes can be seen in Table B.3.

Task	**Proj. Man.**	**Researcher**	**Programmer**	**Tester**	**Tech. Wrt.**	**Total cost (€)**
Contextualization and scope	25h	-	-	-	-	625
Project planning	10h	-	-	-	-	250
Budget and sustainability	20h	-	-	-	-	500
Meetings	20h	20h	20h	20h	20h	2100
Final version of the document	5h	-	-	-	10h	305
Research to improve agents	-	50h	-	-	-	1350
Research to improve monitoring	-	50h	-	-	-	1350
Other research	-	10h	-	-	-	270
Assess the requirements of each feature	-	-	20h	-	-	500
Implement the improvements for the agents	-	-	55h	-	-	1375
Implement the improvements for monitoring	-	-	55h	-	-	1375
Validate the results	-	-	-	20h	-	300
Documentation	-	-	-	-	90h	1620
Oral defence prep.	20h (simult.)	20h (simult.)	20h (simult.)	20h (simult.)	20h (simult.)	2100
Total						**14020**

Table B.2: Initial costs per task and per role [Own making]

Task	**Proj. Man.**	**Researcher**	**Programmer**	**Tester**	**Tech. Wrt.**	**Total cost (€)**
Contextualization and scope	25h	-	-	-	-	625
Project planning	10h	-	-	-	-	250
Budget and sustainability	20h	-	-	-	-	500
Meetings	20h	20h	20h	20h	20h	2100
Final version of the document	5h	-	-	-	10h	305
Research to improve agents	-	135h	-	-	-	3645
Other research	-	15h	-	-	-	405
Assess the requirements of each feature	-	-	25h	-	-	500
Implement the improvements for the agents	-	-	120h	-	-	2400
Validate the results	-	-	-	30h	-	450
Documentation	-	-	-	-	110h	1980
Oral defence prep.	20h (simult.)	20h (simult.)	20h (simult.)	20h (simult.)	20h (simult.)	2100
Total						**15260**

Table B.3: Updated final costs per task and per role [Own making]

HW resource	**Years of use**	**Cost (€)**	**Amortization (€)**
Surface Pro laptop	3	1100	183.24
ASUS desktop	5	1000	99.95

Table B.4: Amortization per HW resource [Own making]

B.1.2 Amortization of the resources

Now, we will compute the amortization of the HW resources that will be used to develop this project. We will take into account the updated number of hours. However, we will still provide the previously computed costs, to see the difference, at the end of this major section. The amortization of any given piece can be calculated as follows:

$$Amort = price \times \frac{1}{years\_of\_use} \times \frac{1}{days\_of\_use} \times \frac{1}{hours\_per\_day} \times hours\_used$$

In Table B.4 then can be consulted the amortization of both computers that we will be employing. The MareNostrum will not be taken into consideration because the platform makes use of the COMPSs library, which allows scalability to be tested in any system, and we will only resort to using HPC (and, therefore, using the MareNostrum) if we deem it necessary due to our computers being too slow. In summary, we will not be using the MareNostrum from the beginning, only if need it during the development of the project, and in that case we would provide an analysis of the costs that making use of the supercomputer would carry. The work will be approximately evenly distributed between each HW piece, and the resources will be used all the time (including meetings (virtual), reading (PDFs and electronic books), etc.). Therefore, if the project would normally last 545 hours over 128 days (4.26 h/day), it will be assumed that each computer will be used 272.5, according to the previous assumption.

These numbers have been calculated as follows, using the aforementioned formula:

- **Surface Pro Laptop:** $Amort = 1100 \times \frac{1}{3yrs} \times \frac{1}{128days} \times \frac{1}{4.26hrs} \times 272.5hrs = 183.24$€.
- **ASUS Desktop:** $Amort = 1000 \times \frac{1}{5yrs} \times \frac{1}{128days} \times \frac{1}{4.26hrs} \times 272.5hrs = 99.95$€.

SW resources will not be taken into consideration because all SW we will be using are free-of-charge, and so are all the books and sources we will be consulting.

Therefore, the total amortization costs amount to 283.19€.

B.1.3 Other indirect costs

In order to make the budget more realistic, here we will include all the other indirect costs that are worth taking into account. That is, Internet access (crucial for the meetings, writing on Overleaf, etc.), electricity costs (electricity is crucial to work on a computer, for obvious reasons), and travel costs. These costs have also been updated to reflect the newly added hours.

- **Internet cost:** My usual Internet bill is of 100€. Therefore, the total Internet cost for this project has been of: (100€/(24*30)) * 545h = 75.69€. This has been calculated by computing the hourly cost of my Internet bill, and then multiplying by the total number of hours that the project will take (since every one of those hours will require access to the Internet).

- **Electricity cost:** The average power consumption by a laptop is 75 W/h, and the average power consumption by a desktop is 200 W/h. In terms of energy, that is 270 kJ/h and 720 kJ/h, respectively. Since we will be using each computer for 272.5 hours, the total consumption of the laptop is expected to be of 270 kJ/h * 272.5 h = 75.575 MJ, and for the laptop the figure would be of 720 kJ/h * 272.5 h = 196.2 MJ, for a total of 271.775 MJ of energy. Now, how to translate that into euros is a bit complicated, seeing how at the current year (2022), we are in the middle of a Europe-wide energy crisis, where the average price of electricity has seen an increase of over 10 times (1000%) what it used to be just a year ago. Nonetheless, we will assume that the last two week's average price of a MWh of energy will stay constant for the almost 5 months of the project. That cost is 350€/MWh. Given that 271.775 MJ are 0.0755 MWh, we get a total energy cost of 26.43€.

- **Transportation costs:** Since we will be working fully from home: from meetings, to doing research, to coding, etc., the costs associated with commuting will amount to a total of 0€.

- **Workplace costs:** Our employees need a place where they can meet, develop the project, and carry out their roles' designated activities. All the work of this project will be done either online (e.g., meeting through Google Meet) or at the author's home. Rent around their zone is of about 1100€/month on average, and since the project will take 4 months, the total workplace costs amount to 4400€.

The total indirect costs are of 4502.12€.

Therefore, the total genetic costs (amortization + indirect) amount to 283.19€+ 4502.12€= 4785.31€.

B.1.4 Deviations and contingencies

Contingency costs
In order to ensure the natural and correct development of the project, we need to take into account the very real possibility that our initial budget may not be enough. That is, we need to plan for contingencies and deviations in our original planning. Due to this, it is always necessary to prepare a contingency fund to act as a reserve of capital that we may need to make use of, should any deviations occur. Here, we provide these costs with the original number of working hours, since the whole point and nature of contingency costs is that they are calculated *before* work on the project starts.

For the staff costs, we will add a 15% of its cost as a contingency fund (15% of 14020€= 2103€).

Finally, for the indirect costs, due to the nature of the project (limited by a hard deadline), we consider that it is less likely that we will need to make further use of indirect resources than we have planned. Because of that, we will only be adding a 5% contingency fund for them: 5% of 433,48 = 21,68€.

Additionally, due to this current year's geopolitical and global context regarding energy prices, we will be including a flat 200€extra contingency fund that can only be spent on electricity, just in case energy prices go even higher.

Therefore, all contingency costs amount to a total of 2065.5€+ 21.68€+ 200€= 2324.68€.

Incidental costs:
We have identified possible risks that might delay the natural course of the project. Now, in this section, we will quantify the cost of each of these risks materializing, and then estimate the probability of them happening, in order to decide how much money we should be assigning to each potential incident. The results can be checked in Table B.5.

Since all the risks' solutions involved working more hours than planned, the estimated costs all come directly from having to cover higher staff costs than foreseen. The probability of a risk happening has been roughly estimated, always using conservative figures and trying to overestimate instead of underestimate. Finally, the actual cost is the result of multiplying the estimated cost by the estimated chance of it happening.

As can be seen in the aforementioned table, the total budget destined to incidental costs amounts to 915€.

Incident	Solution	Estimated costs (€)	Risk (%)	Final cost (€)
Delays in the deadlines	20h extra work for everyone	2100	25	525
Lack of familiarity with the platform	30h extra work for programmer	600	30	180
Very time-inefficient simulation results	30h extra work for programmer and tester	1050	20	210
Total				**915**

Table B.5: Summary of incidental costs [Own making]

Type	Cost(€)
Staff costs	14020
Amortization	285.07
Other indirect	4486.05
Contingency	2324.68
Incidents	915
Total	**22030.80**

Table B.6: Initial summary of all costs [Own making]

B.1.5 Total costs

In this section, we will provide a table summarizing all the costs calculated before starting the project. Such table is Table B.6. However, the actual costs of the project, calculated after it was finished, are displayed in Table B.7.

As can be seen, the total, final cost of the project, accounting for contingencies, incidents, etc., would amount to 23284.99€. The initially calculated costs are of 22030.80€. Therefore, we have spent more than initially planned, more concretely 1254.19€. However, this extra cost can be covered by the contingency funds, that exist specifically to deal with these deviations.

B.1.6 Management control

Despite the fact that we have described the subset of the budget set aside for incidents and contingencies, we still have not described how the potential

Type	Cost(€)
Staff costs	15260
Amortization	283.19
Other indirect	4502.12
Contingency	2324.68
Incidents	915
Total	**23284.99**

Table B.7: Final summary of all costs [Own making]

budget deviations will be modeled and detected.

The approach will be the following: during the development of each task, we will calculate how much it is costing so far, compare it to the estimated cost, and then deliberate. If at any given point we have consumed less or equal money than expected, then nothing is to be done besides writing down the ”good” deviation of the budget. If, however, at any given point we have consumed more money than expected at that point, then we would compute deviation = (estimated cost) - (current cost), and swiftly access the contingency and the appropriate incident funds in order to correct the deviation.

It should be noted that the deviation factor reveals how far the actual cost differs from the estimated cost. If the difference between the estimated and actual costs is positive, it indicates that the job cost was overestimated, and we might redirect this extra resources for other possible risks. If the difference is negative, on the other hand, we must allocate a portion of the contingency budget to the work in order to cover the deviation, as stated in the above paragraph.

How the process works
Every agile iteration, we will compute the following indicators:

- Current cost: this will be computed ad-hoc, by taking into account the hours invested so far, and using them to calculate how much we have spent up to the current agile sprint.

- Estimated cost: this represents how much we should have spent on a task given its current state and the duration of the task. For example, if the cost of a task is set to 200€, and we have currently completed 50% of the task, then the estimated cost would be 100€.

- Deviation: the deviation will simply be the difference between the estimated cost and the current cost. A positive deviation will be a good sign, and a negative deviation will trigger an extraordinary meeting to discuss which corrective actions to carry out and which contingency mechanisms to activate. Most of these corrective actions will probably be in the form of working more hours per day, at least until the deviation is fixed.

B.2 Sustainability

B.2.1 Self-assessment

Whenever we find ourselves involved with a new project in the 21st century, it is of the utmost importance to carefully measure the sustainability of the enterprise we are undertaking, not only to analyze its impact on the environment and society, but also to consider whether it is actually worth carrying out or if its impacts may be too high in comparison to its boons and benefits.

In today's modern world, it is crucial to analyze every detail of the work we conduct to ensure not only that it will have the minimum negative impact on the environment, but also how it will affect society, both on its economic dimension and its social dimension as well: how will our work be used? Who will benefit from it the most? Will it help empower minority groups that have had it rough in the past? Will it have negative economic impacts on groups already in a precarious situation? Or, in the contrary, could such groups benefit from our work?

Before taking this poll, my main thoughts regarding sustainability were always related to the environment: I always thought sustainability only meant having the smallest possible footprint on the environment and on Earth. However, upon deeper inspection and reflection, I have come to realize that sustainability is not as one-dimensional and most people (myself included) normally think of it. We also have to take into consideration the mark that our projects might have on society as a whole and also on individuals and groups of people.

All these very important questions will have an answer during this section, as we explore everything related to the sustainability of our project.

B.2.2 Economic dimension

Regarding PPP: Reflection on the cost you have estimated for the completion of the project
Section B.2.1 of this appendix is dedicated in its entirety to estimate the total cost of the project, breaking it down into different categories and also taking into account contingencies, deviations, and incidents.
Regarding Useful Life: How are currently solved economic issues (costs...) related to the problem that you want to address (state of the art)?, and how will your solution improve economic issues (costs ...) with respect other existing solutions?
The idea of using this agent platform is to make reasoning more scalable to

different computer architectures, which could allow people running simulators to take advantage of different infrastructure and be more efficient in their simulations, thus requiring a smaller computation time and, thus, consuming fewer energy or other valuable resources (computational time is one of such valuable resources, because time spent computing one thing is time not spent computing another thing).

B.2.3 Environmental dimension

Regarding PPP: Have you estimated the environmental impact of the project?
Since an important part of the nature of this project is research, it is difficult to estimate. However, the biggest environmental impact will probably come from running benchmarks at the MareNostrum, which we will try to minimize, or to outright avoid, and in case we need to run them, we will carefully analyze its cost and its impact.
Regarding PPP: Did you plan to minimize its impact, for example, by reusing resources?
Since we will be programming and researching, we will not use any physical resources to **produce** anything physical. And the computers we will be using are my personal computers which I will continue to use after this project.
Regarding Useful Life: How is currently solved the problem that you want to address (state of the art)?, and how will your solution improve the environment with respect other existing solutions?
We aim to do a powerful micro-simulation that will make it easier to analyze more complex social phenomena with fewer runs, and the reduction of runs may have a positive effect in the environment or, more accurately, a less negative impact.

B.2.4 Social dimension

Regarding PPP: What do you think you will achieve -in terms of personal growth- from doing this project
Related to the topic of the project: knowledge about agents, multi-agent systems, high-performance computing and parallel computational models, etc.

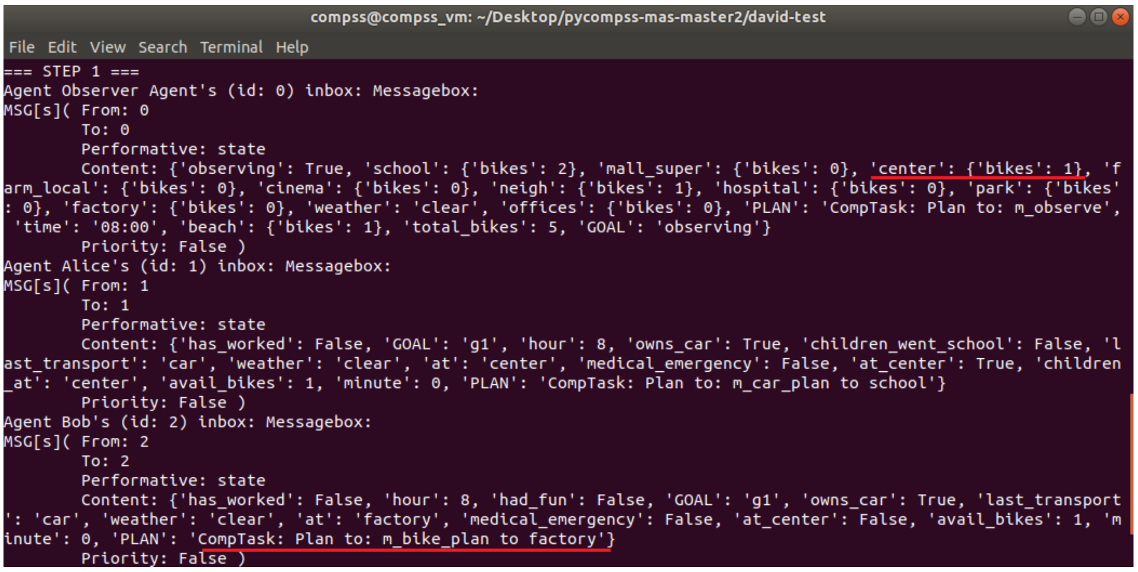
More general skills: project management skills, assessing the sustainability and impact of the project, planning using contingency plans, learn how to work using agile methodology, etc
Regarding Useful Life: How is currently solved the problem that you want to address (state of the art)?, and how will your solution

improve the quality of life (social dimension) with respect other existing solutions?
Because the study is exploratory, this is a question that will be answered once the project is completed. However, the techniques we will integrate are likely to already exist; our work will mostly focus on integrating and evaluating the scalability of the aforementioned techniques. In that sense, progress over the current state of the art will mostly consist in enabling researchers all over the world to run these methods in a more scalable manner.
Regarding Useful Life: Is there a real need for the project?
Yes. The field of MAS and agent-based simulations has a myriad of real-world applications that rise in demand every year. Some worth mentioning are social sciences, simulation of complex environments, traffic, policy-making, economic, behavioural sciences, etc.

Appendix C

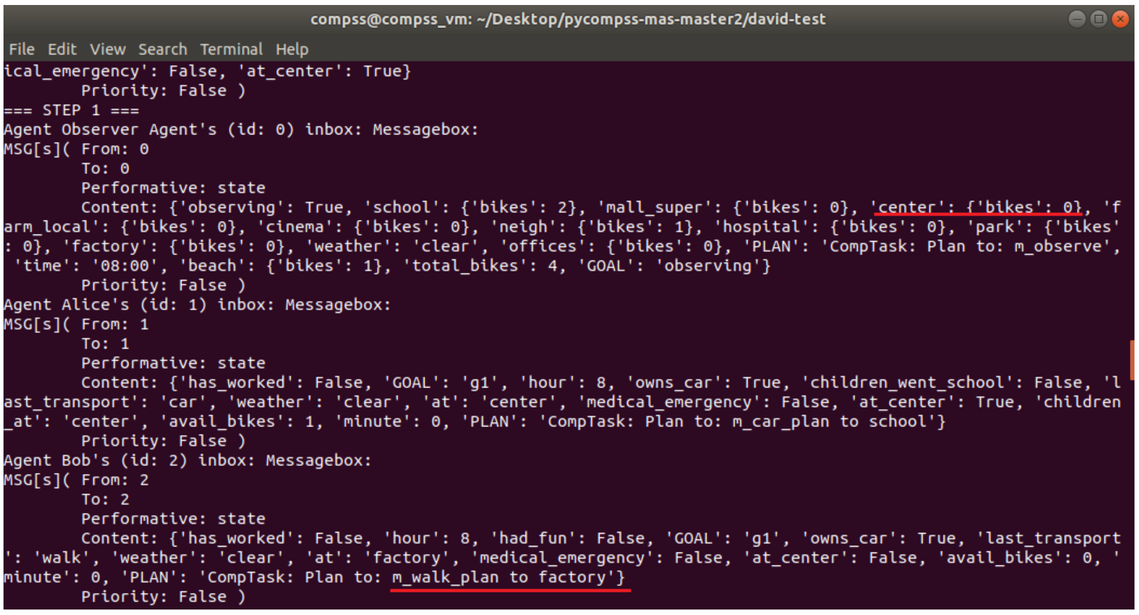
Extra Goodsprings examples

This appendix provides some screen captures to other examples of the executions of Bob and Alice in the Goodsprings scenario. These captures provide further proof that goal selection an task selection works properly with respect to the defined priorities.


compss@compss_vm: ~/Desktop/pycompss-mas-master2/david-test
File Edit View Search Terminal Help

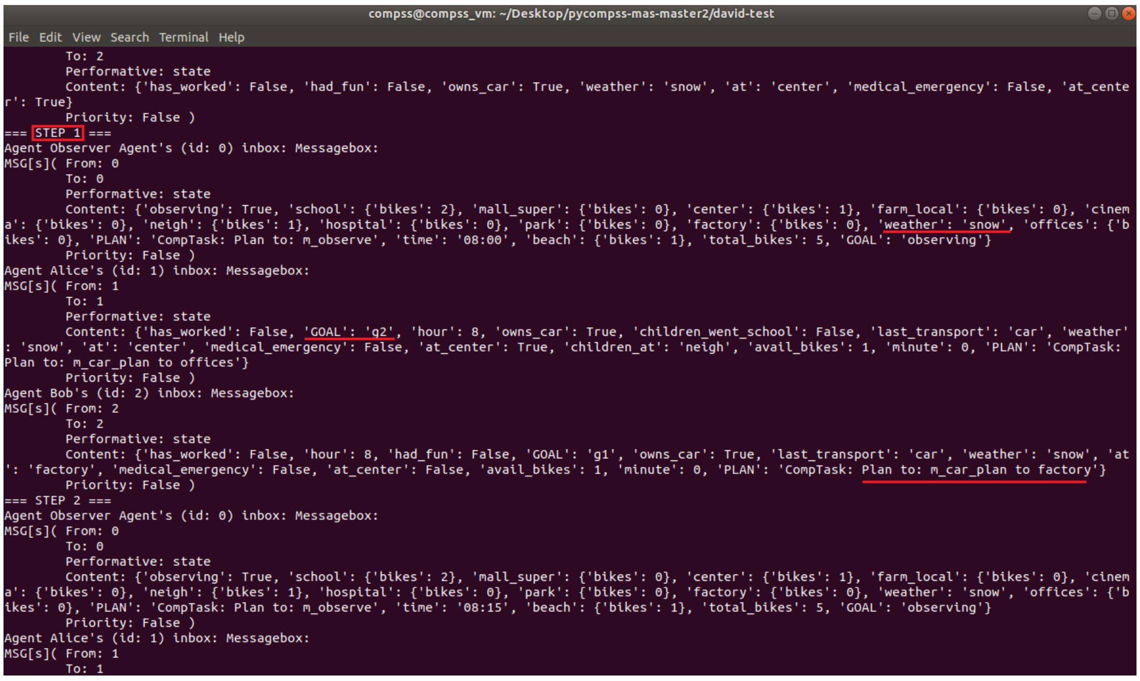
```
=== STEP 1 ===
Agent Observer Agent's (id: 0) inbox: Messagebox:
MSG[s]( From: 0
        To: 0
        Performative: state
        Content: {'observing': True, 'school': {'bikes': 2}, 'mall_super': {'bikes': 0}, 'center': {'bikes': 1}, 'f
arm_local': {'bikes': 0}, 'cinema': {'bikes': 0}, 'neigh': {'bikes': 1}, 'hospital': {'bikes': 0}, 'park': {'bikes'
: 0}, 'factory': {'bikes': 0}, 'weather': 'clear', 'offices': {'bikes': 0}, 'PLAN': 'CompTask: Plan to: m_observe',
 'time': '08:00', 'beach': {'bikes': 1}, 'total_bikes': 5, 'GOAL': 'observing'}
        Priority: False )
Agent Alice's (id: 1) inbox: Messagebox:
MSG[s]( From: 1
        To: 1
        Performative: state
        Content: {'has_worked': False, 'GOAL': 'g1', 'hour': 8, 'owns_car': True, 'children_went_school': False, 'l
ast_transport': 'car', 'weather': 'clear', 'at': 'center', 'medical_emergency': False, 'at_center': True, 'children
_at': 'center', 'avail_bikes': 1, 'minute': 0, 'PLAN': 'CompTask: Plan to: m_car_plan to school'}
        Priority: False )
Agent Bob's (id: 2) inbox: Messagebox:
MSG[s]( From: 2
        To: 2
        Performative: state
        Content: {'has_worked': False, 'hour': 8, 'had_fun': False, 'GOAL': 'g1', 'owns_car': True, 'last_transport
': 'car', 'weather': 'clear', 'at': 'factory', 'medical_emergency': False, 'at_center': False, 'avail_bikes': 1, 'm
inute': 0, 'PLAN': 'CompTask: Plan to: m_bike_plan to factory'}
        Priority: False )
```

Figure C.1: Agent Bob choosing his default preferred means of transport [Own making]



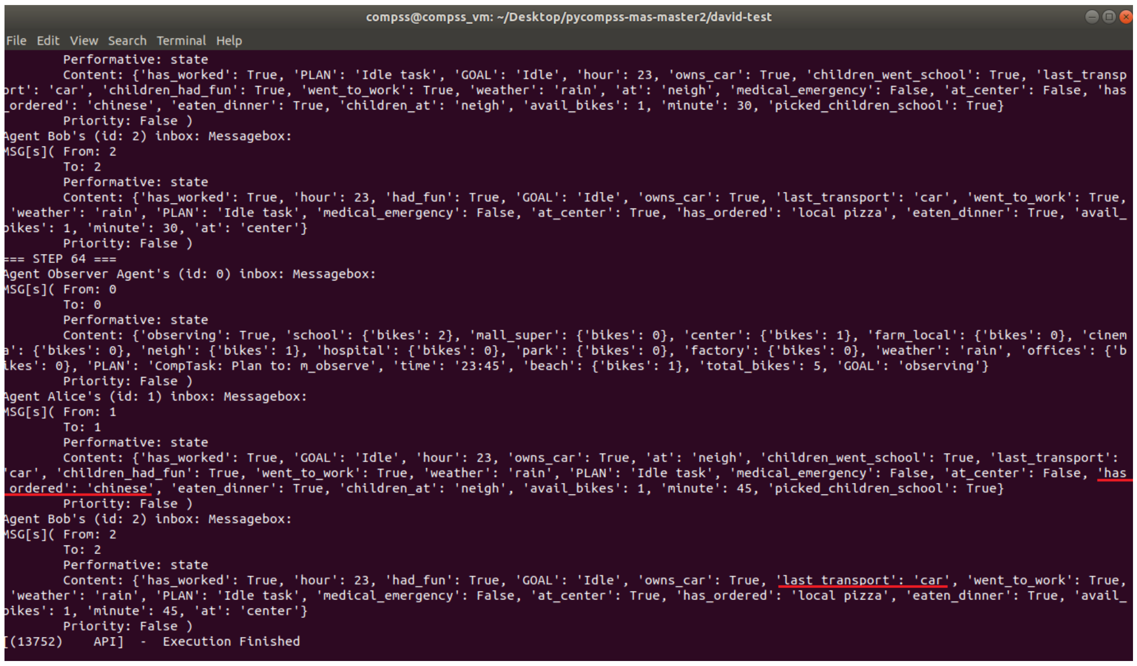
```
compss@compss_vm: ~/Desktop/pycompss-mas-master2/david-test
File Edit View Search Terminal Help
ical_emergency': False, 'at_center': True}
        Priority: False )
=== STEP 1 ===
Agent Observer Agent's (id: 0) inbox: Messagebox:
MSG[s]( From: 0
        To: 0
        Performative: state
        Content: {'observing': True, 'school': {'bikes': 2}, 'mall_super': {'bikes': 0}, 'center': {'bikes': 0}, 'f
arm_local': {'bikes': 0}, 'cinema': {'bikes': 0}, 'neigh': {'bikes': 1}, 'hospital': {'bikes': 0}, 'park': {'bikes'
: 0}, 'factory': {'bikes': 0}, 'weather': 'clear', 'offices': {'bikes': 0}, 'PLAN': 'CompTask: Plan to: m_observe',
 'time': '08:00', 'beach': {'bikes': 1}, 'total_bikes': 4, 'GOAL': 'observing'}
        Priority: False )
Agent Alice's (id: 1) inbox: Messagebox:
MSG[s]( From: 1
        To: 1
        Performative: state
        Content: {'has_worked': False, 'GOAL': 'g1', 'hour': 8, 'owns_car': True, 'children_went_school': False, 'l
ast_transport': 'car', 'weather': 'clear', 'at': 'center', 'medical_emergency': False, 'at_center': True, 'children
_at': 'center', 'avail_bikes': 1, 'minute': 0, 'PLAN': 'CompTask: Plan to: m_car_plan to school'}
        Priority: False )
Agent Bob's (id: 2) inbox: Messagebox:
MSG[s]( From: 2
        To: 2
        Performative: state
        Content: {'has_worked': False, 'hour': 8, 'had_fun': False, 'GOAL': 'g1', 'owns_car': True, 'last_transport
': 'walk', 'weather': 'clear', 'at': 'factory', 'medical_emergency': False, 'at_center': False, 'avail_bikes': 0, '
minute': 0, 'PLAN': 'CompTask: Plan to: m_walk_plan to factory'}
        Priority: False )
```

Figure C.2: Agent Bob choosing his second preferred means of transport, as the first was not available [Own making]



```
compss@compss_vm: ~/Desktop/pycompss-mas-master2/david-test
File Edit View Search Terminal Help
        To: 2
        Performative: state
        Content: {'has_worked': False, 'had_fun': False, 'owns_car': True, 'weather': 'snow', 'at': 'center', 'medical_emergency': False, 'at_cente
r': True}
        Priority: False )
=== STEP 1 ===
Agent Observer Agent's (id: 0) inbox: Messagebox:
MSG[s]( From: 0
        To: 0
        Performative: state
        Content: {'observing': True, 'school': {'bikes': 2}, 'mall_super': {'bikes': 0}, 'center': {'bikes': 1}, 'farm_local': {'bikes': 0}, 'cinem
a': {'bikes': 0}, 'neigh': {'bikes': 1}, 'hospital': {'bikes': 0}, 'park': {'bikes': 0}, 'factory': {'bikes': 0}, 'weather': 'snow', 'offices': {'b
ikes': 0}, 'PLAN': 'CompTask: Plan to: m_observe', 'time': '08:00', 'beach': {'bikes': 1}, 'total_bikes': 5, 'GOAL': 'observing'}
        Priority: False )
Agent Alice's (id: 1) inbox: Messagebox:
MSG[s]( From: 1
        To: 1
        Performative: state
        Content: {'has_worked': False, 'GOAL': 'g2', 'hour': 8, 'owns_car': True, 'children_went_school': False, 'last_transport': 'car', 'weather'
: 'snow', 'at': 'center', 'medical_emergency': False, 'at_center': True, 'children_at': 'neigh', 'avail_bikes': 1, 'minute': 0, 'PLAN': 'CompTask:
Plan to: m_car_plan to offices'}
        Priority: False )
Agent Bob's (id: 2) inbox: Messagebox:
MSG[s]( From: 2
        To: 2
        Performative: state
        Content: {'has_worked': False, 'hour': 8, 'had_fun': False, 'GOAL': 'g1', 'owns_car': True, 'last_transport': 'car', 'weather': 'snow', 'at
': 'factory', 'medical_emergency': False, 'at_center': False, 'avail_bikes': 1, 'minute': 0, 'PLAN': 'CompTask: Plan to: m_car_plan to factory'}
        Priority: False )
=== STEP 2 ===
Agent Observer Agent's (id: 0) inbox: Messagebox:
MSG[s]( From: 0
        To: 0
        Performative: state
        Content: {'observing': True, 'school': {'bikes': 2}, 'mall_super': {'bikes': 0}, 'center': {'bikes': 1}, 'farm_local': {'bikes': 0}, 'cinem
a': {'bikes': 0}, 'neigh': {'bikes': 1}, 'hospital': {'bikes': 0}, 'park': {'bikes': 0}, 'factory': {'bikes': 0}, 'weather': 'snow', 'offices': {'b
ikes': 0}, 'PLAN': 'CompTask: Plan to: m_observe', 'time': '08:15', 'beach': {'bikes': 1}, 'total_bikes': 5, 'GOAL': 'observing'}
        Priority: False )
Agent Alice's (id: 1) inbox: Messagebox:
MSG[s]( From: 1
        To: 1
```

Figure C.3: Agents Alice and Bob choosing their preferred means of transport for when it snows [Own making]


compss@compss_vm: ~/Desktop/pycompss-mas-master2/david-test
File Edit View Search Terminal Help

```
        Performative: state
        Content: {'has_worked': True, 'PLAN': 'Idle task', 'GOAL': 'Idle', 'hour': 23, 'owns_car': True, 'children_went_school': True, 'last_transp
ort': 'car', 'children_had_fun': True, 'went_to_work': True, 'weather': 'rain', 'at': 'neigh', 'medical_emergency': False, 'at_center': False, 'has
_ordered': 'chinese', 'eaten_dinner': True, 'children_at': 'neigh', 'avail_bikes': 1, 'minute': 30, 'picked_children_school': True}
        Priority: False )
Agent Bob's (id: 2) inbox: Messagebox:
MSG[s]( From: 2
        To: 2
        Performative: state
        Content: {'has_worked': True, 'hour': 23, 'had_fun': True, 'GOAL': 'Idle', 'owns_car': True, 'last_transport': 'car', 'went_to_work': True,
 'weather': 'rain', 'PLAN': 'Idle task', 'medical_emergency': False, 'at_center': True, 'has_ordered': 'local pizza', 'eaten_dinner': True, 'avail_
bikes': 1, 'minute': 30, 'at': 'center'}
        Priority: False )
=== STEP 64 ===
Agent Observer Agent's (id: 0) inbox: Messagebox:
MSG[s]( From: 0
        To: 0
        Performative: state
        Content: {'observing': True, 'school': {'bikes': 2}, 'mall_super': {'bikes': 0}, 'center': {'bikes': 1}, 'farm_local': {'bikes': 0}, 'cinem
a': {'bikes': 0}, 'neigh': {'bikes': 1}, 'hospital': {'bikes': 0}, 'park': {'bikes': 0}, 'factory': {'bikes': 0}, 'weather': 'rain', 'offices': {'b
ikes': 0}, 'PLAN': 'CompTask: Plan to: m_observe', 'time': '23:45', 'beach': {'bikes': 1}, 'total_bikes': 5, 'GOAL': 'observing'}
        Priority: False )
Agent Alice's (id: 1) inbox: Messagebox:
MSG[s]( From: 1
        To: 1
        Performative: state
        Content: {'has_worked': True, 'GOAL': 'Idle', 'hour': 23, 'owns_car': True, 'at': 'neigh', 'children_went_school': True, 'last_transport':
'car', 'children_had_fun': True, 'went_to_work': True, 'weather': 'rain', 'PLAN': 'Idle task', 'medical_emergency': False, 'at_center': False, 'has
_ordered': 'chinese', 'eaten_dinner': True, 'children_at': 'neigh', 'avail_bikes': 1, 'minute': 45, 'picked_children_school': True}
        Priority: False )
Agent Bob's (id: 2) inbox: Messagebox:
MSG[s]( From: 2
        To: 2
        Performative: state
        Content: {'has_worked': True, 'hour': 23, 'had_fun': True, 'GOAL': 'Idle', 'owns_car': True, 'last_transport': 'car', 'went_to_work': True,
 'weather': 'rain', 'PLAN': 'Idle task', 'medical_emergency': False, 'at_center': True, 'has_ordered': 'local pizza', 'eaten_dinner': True, 'avail_
bikes': 1, 'minute': 45, 'at': 'center'}
        Priority: False )
[(13752)    API]  -  Execution Finished
```

Figure C.4: Agent Alice having chosen her preferred meal for when it rains, while agent Bob has used his preferred means of transport for when it rains [Own making]