ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Ingegneria e Architettura
Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

JACOP: Programming BDI Agents with Pluggable Concurrency Model

Tesi di laurea in
Intelligent Systems Engineering

Relatore
Prof. Giovanni Ciatto

Candidata
Martina Baiardi

Correlatori
Prof. Andrea Omicini
Prof. Jomi F. Hübner

Anno Accademico 2021-2022

Abstract

Nowadays, more and more disciplines adhere to Agent-Oriented Programming paradigm to break down complex problems. Various methodologies have been formalized over the years to model the behaviour of an agent, one of them is the well-known BDI (Beliefs, Desires, Intentions) model [11]. This model, after receiving great interest from researchers, was formalized into an abstract language called AgentSpeak(L), and later implemented in Jason [1]. Jason let users express multi-agent system (MAS) specification via an *ad-hoc* language and a clear concurrency model.

In this work we propose a novel interpreter for AgentSpeak(L) – namely, Jacop – supporting both concurrency model *pluggability*, and interoperability with mainstream programming languages – namely, the JVM-based ones –, via a Kotlin-based *domain specific language* (DSL).

On the one hand, concurrency model pluggability let users choose the best concurrency model for their applications. In this way, the *same* MAS specification can run on resource-constrained devices, as well as on parallel, distributed, and simulated architectures.

On the other hand, our DSL aims at specifying BDI agents directly in Kotlin. In this way, users can easily develop MAS without having to learn a new syntax, and, therefore, smoothly blend BDI agents into other projects.

To demonstrate the functionality of our contribution we test Jacop over several MAS specifications, aimed at assessing if and to what extent: *(i)* the Jacop interpreter covers AgentSpeak(L) semantics, *(ii)* the Jacop DSL covers Jason syntax, *(iii)* the same MAS specification can run on different concurrency models. A qualitative analysis of the result code snippets and their execution shows that all such features are satisfied.

To the dreams that come true.

Contents

Chapter 1

Introduction

Computer systems became more complex and more powerful over the years. The more the complexity of the system grows, the more sophisticated techniques – such as decomposition and abstraction – are useful. However, the construction of reliable, maintainable and extensible systems requires design methodologies and technologies to manage their inherent complexity.

Agents. Agents are a form of abstraction for complex systems, they can perform tasks and interact with each other. Agents are also relevant in the field of artificial intelligence (AI), where they are used to model *intelligent* behaviour. Each one of them has an explicit representation of the world that depends on the *environment* in which are situated and deliberate about the best course of action to take to achieve their goals.

Multi-Agent System (MAS). A MAS is a *society* composed of multiple Agents that interact with each other in order to achieve common goals. Multi-Agent Systems play a key role in the field of artificial intelligence, where agents are used as an abstraction to model complex systems. Agents' behaviour is strictly coupled with the *environment* in which they live, this means that the environment is also an essential entity of a Multi-Agent System.

Recent fields of AI, such as XAI (eXplainable AI), show that the use of sub-symbolic techniques is at the core of many successful agent-based models and technologies. XAI is a research field that aims to "open the black box" of neural networks and turn them into transparent systems [7], possibly by leveraging symbolic AI techniques—e.g., computational logic [4]. A thorough literature review of MAS solutions involving logic-based technologies is reported in [3]. There, MAS have the potential to bridge the two branches of AI (symbolic and sub-symbolic)

under a coherent, unified conceptual framework.

BDI model and AgentSpeak(L). Many researchers proposed different design methodologies and languages to model agents and multi-agent systems. One of the most popular and successful frameworks to develop agents is the *Beliefs, Desires, Intentions* (BDI) model [12]. Rao and Georgeff proposed a way to design agents specifying three main components: *beliefs*, *desires* and *intentions*. Using these three pieces of information, agents can reason about the best course of action to take to achieve their goals. However, the BDI model is not a programming language, but an agent theory to explain complex behaviours. For this reason, Rao proposed AgentSpeak(L) [10], a logic-based programming language specification that provides an alternative formalization of BDI agents.

Jason. AgentSpeak(L) only defines an abstract language: to adopt the BDI model on real systems applications a concrete implementation was necessary. One well-established technology is Jason [1], that in 2007 concretely implemented the abstract definition. Jason provides a language that supports the development of agents that reason, plan, and interact with each other and with the environment in which they are situated.

Currently, developers need to follow the Jason language to develop BDI agents. This means that they can not use the programming language of their own choice, because it could not be compatible to work with it. This bought us to believe that a more flexible approach could be useful to let developers design (and run) agents on-the-fly inside their applications.

Moreover, the Jason interpreter comes with its own concurrency models—i.e., the interpreter is in charge of allocating agents on threads. While this choice makes perfectly sense in most scenarios, we argue that there may be cases where developers might want to have more control over the execution model of their agents. Along this line, users should be able to *adopt* – and possibly realise – their preferred execution strategy, hence controlling agents' scheduling in a fine-grained way.

Jacop. This work proposes Jacop (Jason-like Agents where COncurrency is Pluggable): a library to develop a Multi-Agent Systems (MAS) inspired by Jason's syntax, that separates the framework into different abstraction layers, which developers may plug and replace according to their needs. The library supports both concurrency model *pluggability*, and interoperability with mainstream programming languages – namely, the JVM-based ones –, via a Kotlin-based *domain specific language* (DSL).

Jacop separates the execution model from the domain specification, this means that developers can *plug*, and also *customize*, the execution model that better fits their needs. This implies a complete decoupling among the multi-agent system specification and the architecture on which it is executed, as well as parallel, distributed and simulated ones.

This library provides also a Kotlin-based DSL to define MAS and its components, which can be used as an elegant way to inject MAS into other systems. Moreover, our DSL aims at specifying BDI agents directly in Kotlin. In this way, users can easily develop MAS without having to learn a new syntax, and, therefore, smoothly blend BDI agents into other projects.

To demonstrate the functionality of our contribution we test Jacop over several MAS specifications, aimed at assessing if and to what extent: *(i)* the Jacop interpreter covers AgentSpeak(L) semantics, *(ii)* the Jacop DSL covers Jason syntax, *(iii)* the same MAS specification can run on different concurrency models. Finally, a qualitative analysis of the result code snippets and their execution shows that all such features are satisfied.

The following chapters are organized as follows. Chapter 2 discusses the State of the Art of the concept analyzed within this thesis, including references to previous works and technologies. Chapter 3 summarizes the requirements that the proposed solution satisfies and discusses the design of the library and its components. Chapter 4 describes the implementation of relevant aspects of Jacop. Chapter 5 portray tests and usages of the proposed solution. Finally, Chapter 6 concludes this thesis by summarising its main contribution and future works.

Chapter 2

State of the Art

This chapter provides an overview of the scientific literature which led to the definition of Jacop language. It starts describing Multi-Agent Systems and their usefulness to solve complex systems. Then it discusses the BDI model, one of the best-known approaches to represent agents' capabilities and then it describes the contribution given by AgentSpeak(L) language to actually adopt them. Finally, it describes Jason language, the most acknowledged language that implements AgentSpeak(L) definitions, and discusses the main differences between the two of them.

2.1 Multi-Agent Systems

The Agent-Oriented Paradigm (AOP) was introduced by Shoham in 1993 [13]. The AOP framework was introduced to be a *specialization* of the Object-Oriented Programming (OOP) framework. The main difference between the two of them is that while OOP let users define generic modules that can communicate with each other, AOP constrains the state – or *mental state* – of those modules, transforming them into *agents*.

The agent concept was first introduced in fields related to Distributed Artificial Intelligence (DAI) to simplify complex distributed systems. However, each specific field of study involved with agents gave its nuance for the entity definition, even if the notion of agent is more generic. A shared and generalized definition for an Agent is the following [6]:

> *An agent is an entity which is placed in an* ***environment*** *and senses different* ***parameters*** *that are used to make a decision based on the goal of the entity. The entity performs the necessary* ***action*** *on the environment based on this decision.*

Agents are placed within a shared *environment*, they observe changes in its state in order to perform decision-making operations. However, these entities can not perceive any nuance of such a complex state, indeed they only sense specific types of data, that are their *parameters*. The environment parameter they observe strictly depends on the purpose of each agent. After the decision-making process, agents could decide to execute actions to operate on the environment state in which they are inserted. The goal of each agent is to solve tasks for which is developed, respecting some constraints that it senses from the environment.

For Wooldridge [15], an agent is considered *intelligent* when it *flexibly* performs its choices, and agents compliant with these features are considered to have the capabilities to solve complex tasks [6]. A flexible agent satisfies these features:

- *Sociability*: Agents are capable of interacting with each other in order to share their knowledge.
- *Autonomy*: Each agent can take decisions and actions, without human intervention, during their execution.
- *Reactiviy*: Agents perceive the environment and undertake actions, based on its changes, in order to satisfy their goals.
- *Proactivity*: Each agent exhibits a goal-directed behaviour, indeed they take the initiative to choose the actions to take based on their knowledge.

The case in which only one agent exists in the environment does not require complex techniques to manage their knowledge representation. The real benefit of this technology is exploited when they work collaboratively. Multiple agents that collaborate to solve a complex task are known as Multi-Agent Systems (MAS).

Multi-agent systems (MAS) represent a means to solve complex problems by subdividing them into smaller tasks. Each agent can solve the allocated task with any level of knowledge, introducing high flexibility.

2.2 BDI model

BDI is a model for rational agents that is based on the idea that agents can be modeled through *Beliefs*, *Desires* and *Intentions*. Bratman in 1987 [2] argued for the first time the role played by intentions and plans in the design of rational agents, to shape their behaviour and decisions. He stated that intentions play a significant role in practical reasoning, indeed they represent the commitment of an agent to perform a course of actions to fulfill its goals.

This idea was well recognized in the AI literature so, later on, Rao and Georgeff [11][12] formalized a semantics for the theorized BDI model. Authors observe that

agents, at a certain time during their execution, will select a course of action to perform to reach their goals, depending on what is their perception of the environment. However, the environment is mutable over time, so an agent must keep a memory of its past decisions that it is going to follow. This commitment is represented by an agent's intention. In addition, agents can carry out multiple intentions at the same time alternating their action execution, this shows that an agent does not carry a single intention over its execution but a *set* of them.

The BDI model described by Rao and Georgeff in [12] brings a way to describe a system composed of multiple agents, that act over a shared environment to achieve their goals. The environment is the entity where agents live and act, so it is composed of information that agents can *perceive* and, possibly, *modify*. Agents might *act* over the environment to change its state, to do so they have to select which actions to execute from various *plans* of actions available. The system can achieve its primary objectives by selecting the appropriate actions to perform, given the characteristics of the environment in which agents are situated.

The following sections describe the main concepts of the BDI model: beliefs, desires and intentions.

Beliefs

To take decisions, agents need to have a representation of the world in which they live. Given that an environment is an extremely mutable entity, agents who perceive it must have a component that let them keep track of the changes that occur over time. This knowledge perceived by agents is called *Belief*, the latter enables them to describe the environment's state that agents can collect. An agent can store multiple perceptions in its memory, this set of beliefs is called *Belief Base*.

In [8] authors define the Belief model as follows:

> *A Belief Model describes the information about the environment and internal state that an agent of that class may hold, and the actions it may perform. The possible beliefs [...] are described by a belief set. In addition, one or more belief states – particular instances of the belief set – may be defined and used to specify an agent's initial mental state.*

The authors' idea is that an agent has a *mental state* that changes over time performing actions and perceiving back consequent environment's changes. The mental state is made up of *beliefs states*, which represents an instance of the *belief set* held by the agent.

Desires

The system's primary objectives are called Desires, they represent the *motivational* state of the system. Agents are provided with a set of plans that they can use to achieve their goals, each plan is composed of a set of actions that the agent must execute to contribute to the achievement of its goals. From the user's viewpoint, the most suitable plan choice is based on two factors: the agent's and the environment's actual state. The whole process for the plan selection is declared *Selection Function*, which specific design is a field of decision theories.

Intentions

After a single plan selection, or also during action execution, the environment in which the agent is situated could change. This means that the system tracks what actions are committed to execution, and also it must keep track of the progress of each one of them. This is the main purpose of the Intentions set: they capture the *deliberative* component of the system.

2.3 AgentSpeak(L)

After the BDI model was formalized, under both the theoretical and the design perspective, and after the raise of interest from the researcher, Rao proposed AgentSpeak(L) [10] as a semantic language for the BDI model application. This language enables developers to describe agents in horn-clause logic similar to programs.

AgentSpeak(L) mainly follows the BDI model, indeed representing agents with a set of beliefs, desires and intentions as its three primary mental attitudes. The current state of the agent, that is the model of itself, the environment and other agents, can be viewed as its current *belief base*. States that the agent wants to bring the system are represented as its *desires*, and the actions that the agent is committed to performing to reach its desires are its *intentions*. However, the language also investigates other notions such as commitment, capabilities, know-how and others, resulting in a more refined syntax.

Another feature brought by AgentSpeak(L) is the introduction of the operational semantics of the language. Authors explain that a run-time agent can be described with a unique tuple of elements, this is described in the following sections.

2.3.1 Language definition

An AgentSpeak(L) agent is described using five components: *beliefs*, *goals*, *events*, *actions* and *plans*.

Listing 2.1: Horn Clause example for belief representation

```
grandmother(X, Z) :- mother(X, Y), parent(Y, Z).
```

Beliefs. Agent's beliefs semantics follows the description given in the BDI model [12]. They are represented as *Horn Clauses*, where the predication in the head of the rule represents the belief's name.

The Listing 2.1 is a representation of a Horn Clause, where X, Y and Z are variables that can be instantiated with any value. Predicates *mother* and *parent* assert statements from the domain of the discourse, which can be *true* or *false*. Following the example, *grandmother* belief is true only if X is the mother of some Y and the same Y is the parent of Z.

Horn Clauses can be represented as *rules*, like in Listing 2.1, and as *facts*. The difference between these two notations is that a rule can be true only if its head and all of its body predicates are true, while a fact is always true in the domain concerned by the agent. A peculiar instance of fact is a predicate with no arguments, this denotes a statement that holds in the agent's domain.

Goals. Goals are the state of the system that the agent wants to reach. In AgentSpeak(L) there is a slight difference from the BDI definition, indeed it introduces goals as the agent's *adopted* desires. There are two types of goals that an agent can manage: *achievement goals* and *test goals*. Achievement goals represent the state of the world that the agent wants to achieve, which eventually will hold by its belief base. Test goals state that the agent wants to know if a certain belief is true or not.

Events. When an agent gets a new goal to achieve or when it notices some changes in the environment, it may generate an addition (or deletion) from its goals or beliefs. These occurrences are called *triggering events*. There are six types of triggering events accordingly:

1. Addition of a belief
2. Deletion of a belief
3. Achievement of an achievement goal
4. Deletion of an achievement goal
5. Achievement of a test goal
6. Deletion of a test goal

Listing 2.2: Plan definition compliant with AgentSpeak(L) syntax

```
+location(waste, X) : location(robot, X) &
                      location(bin, Y)
                    <- pick(waste);
                       !location(robot, Y);
                       drop(waste).
```

Actions. The purpose of an agent is to execute some actions in order to reach its goals. Actions are the primitive operations that the agent can perform.

Plans. An agent is provided with a set of plans, each one of them is composed of actions that the agent may execute to contribute to the achievement of its goals. Plans consist of a head and a body, the head is composed itself by a *triggering event* and a *context*, while the body is a sequence of goals. The plan's context specifies the beliefs that should hold in the agent's belief base when the plan is triggered, to execute it.

An example of AgentSpeak(L) plan syntax is shown in Listing 2.2: it describes a robot agent that performs the waste collection operation. The highlighted plan is triggered when waste appears in a particular position in the environment, and its body actions can be executed only if the robot is at the same location. The agent then can act as picking up the waste, followed by the action of reaching the bin location and finally executes the action of dropping the waste inside of it. Of course, the described robot must also have appropriate plans to make it able to reach the desired position in the environment.

2.3.2 Operational semantics

The state of an agent, during its whole lifecycle, can be described by a tuple of elements:

$$\langle E, B, P, I, A, S_E, S_O, S_I \rangle$$

where E represents the set of events, B is the set of beliefs, P is the set of plans, I is the set of intentions, A is the set of actions, S_E is the selection function that selects from the E set, S_O is the selection function that selects the applicable plans, and S_I is the selection function that selects an intention from I.

The agent execution is ruled by events that occur during its lifecycle. An event can be generated during the execution of a plan by another agent's request, or triggered by a change in the environment's state. There are two types of events: *internal* events and *external* ones. The difference between the two of them is found in the source of the event: internal events are generated by the agent itself, while external events are generated by the environment or other agents. At any moment, an agent may have multiple events to manage: selection function S_E is used to choose the next event to be processed.

Once an event is selected, the agent uses it to unify with the triggering events of the plans in its set P, the ones that unify are called *relevant* plans. The relevant unifier is then applied to the context condition of the plan, if the computed context is a logical consequence of the agent's belief base, the plan is considered *applicable*. For each event managed by the agent, there could be many applicable plans to adopt, for this reason, the selection function S_O is used to select which plan to execute between the applicable ones. Once a plan is selected, the agent associates it with an intention to execute it, indeed each intention is essentially a stack of partially instantiated plans. If the selected triggering event is an external one this will generate a new intention, otherwise, the plan will be added to the top of the intention that generated the internal event.

Lastly, an agent must select which intention to execute, this job is done by the selection function S_I. When an intention is selected for execution only the first goal (or action) on the top of the intention is executed by the agent. Goal execution may trigger new events, and then the whole cycle is repeated.

2.4 Jason

Jason is the most used and acknowledged language that implements AgentSpeak(L), a logic-based agent-oriented programming language, proposed by Jomi F. Hübner and Rafael Bordini in 2007 [1]. Agents in Jason are developed following the BDI definition, which is a model that is based on the idea that agents can be modeled through Beliefs, Desires and Intentions. The language not only adheres at the AgentSpeak(L) notion of an agent but also provides extensions to make the language more expressive and powerful for multi-agent systems' definition.

The framework introduces *annotations* to use wherever an atomic formula was allowed in the original language, enclosed in square brackets immediately following the formula. This information inside the belief base can be used, for example, to specify the source of the belief, the time when it was acquired and others. In other components, such as plans, annotations can be used to adopt sophisticated selection functions, such as the one that selects the plan with the highest confidence level.

Moreover, Jason introduces events for handling plan failure. Agent's actions can fail, or there could not be any applicable plan to execute related to it, in this case, the agent can generate a failure event. Actually, the deletion of a goal is not an interesting event to model within an agent, in fact, this triggering event represents the failure that occurred during the goal execution.

Lastly, the language discerns the action concept in two different ones: *internal* actions and *external* actions. The first ones are directed to query (or modify) the agent itself, while the second ones operate over the environment. There are several implementations provided by the framework for internal and external actions, but developers can create their custom ones using Java language.

Chapter 3

Design

The Jacop library provides a tool to develop agents who perceive and act within a shared environment and communicate with each other through a form of message-passing communication.

The library brings a framework to build a Multi-Agent System composed of agents compliant with the BDI model and inspired by Jason implementation. A Multi-Agent System is composed of two fundamental entities:

- *Agents*: the agent is the main entity of the library.
- *Environment*: the environment is the entity where agents live.

MAS agents observe the environment state during their execution and they choose independently to act, depending on which information they perceive. Their actions may affect the environment's state, and the change will be percept by all other agents that operate over it. Indeed, agents are provided with goals they are committed to achieve, which together bring the entire Multi-Agent System's goal achievement. Regarding the environment entity, a user can:

- *Create* an environment
- *Add* capabilities to the environment, such as:
  - Definition of actions that agents within the environment can use.
  - Definition of the environment's state, as the set of information that agents inside can perceive and, possibly, modify.
- *Add* and *remove* agents from the environment

Furthermore, agents situated in the same environment can communicate through messages. Messages enable agents to share some information about their state, or also share a goal completion, this feature makes the Multi-Agent System a truly *society* of agents. Regarding the agent entity, a user can:

- *Create* agents and customize their behaviour, following the BDI model.
- *Add* actions that one single agent can perform. For example, users can define agent-specific operations that they can perform with their sensors that do not affect the environment state, such as the movement of a robot arm.
- *Customize* agents' behaviour during their execution, for example, providing them with Artificial Intelligence algorithms that change their deliberative operations.

The library let users define how their BDI agents will run. This means that the same agents can be executed both in a single thread or between multiple of them, in an almost completely transparent way. To enable the MAS to it, users must define its *execution model*, i.e. how its agents are executed. This execution model must be *pluggable* because it enables the dynamic choice of which type of execution users want to adopt within the library.

To make it possible for users to customize how the system is executed, it is necessary to define an abstraction layer that governs its iterations, separating them from the physical machine on which is executed. Indeed, an agent is an active entity that has its own life cycle, which *transits* through different *states* before reaching its completion. These operations together denote a state machine that regulates the entire execution of each entity in the system.

Finally, an abstraction layer is required above the definition of domain entities, which guides users during their definition. This abstraction is represented by a Domain Specific Language (DSL) which let users define only the essential concepts necessary for the system execution, making it more intuitive to understand.

This chapter describes the design choices adopted for the proposed library components. It first describes the domain entities of the library in Section 3.1, then it discusses the library's design choices made before the implementation phase Section 3.2.

3.1 Domain Model

The purpose of Jacop is to provide a flexible framework, inspired by Jason, that let developers run multi-agent systems (MAS) on different execution models. Three

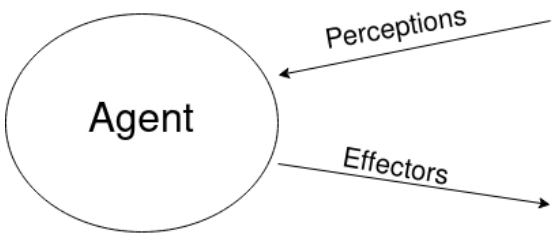

Figure 3.1: Agent representation through perceptions and effectors

entities compose a MAS: *agents*, *environments* and *execution strategy*.

3.1.1 Agents

An agent is an entity that can perceive the environment, reason about changes that happens in it, plan and act.

Agents can be imagined as in Figure 3.1: they can be reactive thanks to their perceptions, indeed, they are used to elaborate the appropriate effectors – the Agent's Actions that act over the environment – to adopt in order to reach the desired state of the environment.

Agents share an environment and interact with each other to achieve their goals. Jacop agents comply with the Jason extension of AgentSpeak(L), indeed they are developed using the BDI framework and a similar first-order logic syntax.

Following the AgentSpeak(L) definition in Section 2.3.2, agents can be described using the following elements: Events, Beliefs, Plans, Intentions, Actions, Event Selection Function, Plan Selection Function and Intention Selection Function.

Events

Agents, during their execution, decide which operations to perform depending on which events occur. Events can be generated from a change in the environment observed by the agent, from other agent actions or from the execution of a previously scheduled plan. Two different types of events can be discerned accordingly: *internal* events and *external* ones.

One of the goals the agent could encounter can be the achievement of another goal, in this case, the agent must execute all the steps that satisfy the other goal before continuing the previous intention execution. What happens is that an internal event for the new goal is generated, because the agent must reason over the most suitable plan to adopt to achieve that goal, but it also needs to keep track of the previously interrupted intention to continue its execution after the new goal

is reached. Other event sources are pretty similar, they ask the agent to react to something that happened outside of its context, and these are external events.

The difference between the two types of events is simply the presence of the intention that generated it, needed to let the agent resume its execution.

Events can also be classified from the trigger that generated them, indeed they could be generated from:

1. An addition inside of the belief base
2. A deletion from the belief base
3. A test goal invocation
4. A test goal failure
5. An achievement goal invocation
6. An achievement goal failure

Beliefs

Apart from the ability to perceive the environment, an agent could have some prior knowledge. Each piece of knowledge is represented by a *belief*, which together compose the agent's belief base. Beliefs are defined as Horn Clauses, in particular, there could be two different types of them: *rules* and *facts*. A rule belief can hold only if its head and all of its body predicates are true, while a fact belief always holds.

Developers can see beliefs as a logic theory that the agent will use – during its execution – to adopt the most suitable plan of action. Moreover, beliefs represent the agent's knowledge about the environment, in fact, an addition (or deletion) of them generates triggering events, that eventually will perform a suitable course of action. This language detail enables modeling agents that adapt their behaviour to environmental changes.

As in Jason, agents' beliefs have another detail: they also are provided with a list of annotations that let users distinguish *knowledge* based on some properties. We decided to provide a similar feature, although the initial design of this property is simple: instead of a list of them, beliefs are provided with only one annotation, that stores the source of the piece of knowledge.

Plans

Agents are provided with goals that they want to achieve, these goals could also enable other goals to be satisfied, however, the main objective of the agent is to

satisfy them all. Plans represent the capabilities of the agent, indeed the only way to make them able to perform actions is through a plan's body.

AgentSpeak(L) provided an excellent description of what a BDI-compliant plan should be composed of, which is the one on which we made our design choices. Plans are composed of three components: *triggering event*, a *context* and a *body*. The first element let agents determine which plans are *relevant* against an event selected from the event set. After determining relevant plans, agents select only *applicable* plans from relevant ones. A certain plan is applicable only if its context is a logical consequence of the agent's belief base. The context is important for plan definition because a plan should not only rely on triggering events happening, but also on the actual knowledge of the agent. Lastly, the body of the plan is a list of operations that the agent must perform to reach its main goals. These operations are modeled as *goals*, indeed they represent tasks the agent must perform to satisfy the overall plan execution.

Once an applicable plan is selected, the agent can commit itself to its execution associating it with an intention. If the plan was triggered by an internal event, the agent will recover the previous intention that generated it and add the new plan's goals on top of its execution stack, otherwise, an empty intention will be filled with those goals.

One plan succeeds only if all its goals are executed with success. If a failure occurs during plan execution, a failure event should be raised and also there should be a plan to manage the occurrence. This is useful for developers because they can provide agents with capabilities to recover themself after some errors appear.

In the framework there are six kinds of goals:

1. *Belief addition*: which execution adds a new belief inside the agent's belief base. The goal execution will then trigger a new belief base addition internal event.

2. *Belief deletion*: that deletes the specified belief from the agent's belief base. The goal execution will then trigger a new belief base removal internal event.

3. *Belief update*: that modifies a current belief value into a new one.

4. *Achievement goal invocation*: which execution will generate an achievement goal internal event. The execution of the current intention does not continue until the plan goals, assigned to satisfy the new event, are executed successfully.

5. *Spawn goal invocation*: the same as an achievement goal invocation, but the execution of the current intention and the new one is concurrent.

6. *Test goal invocation*: it checks whether a belief is believed by the agent's knowledge. It is typically used to retrieve values from the belief base.

7. *External action execution*: runs an action that potentially changes the environment's state.

8. *Internal action execution*: runs an action that potentially changes the agent's state.

Intentions

Intentions are the core aspect related to the BDI agent's execution, indeed they represent the commitment of an agent to perform some goals. Intentions' goals can also trigger new events that need to be satisfied before continuing their execution. Each element of the Agent's Intention set represent a different *Focus of Attention*, which means that all of them are competing for the Agent's attention, but only one of them is managed at a time. We can imagine that an intention is composed of a stack of partially instantiated plans, the latter are queues of goals provided by a previously selected plan. We call them *Activation Records*, they represent the execution state of the intention. At each step of the agent's execution, it will perform a single goal, this one will be the next goal in the activation record at the top of the intention stack. A graphic example of this description is shown in Figure 3.2, where we can see a sample snapshot of an intention. An intention can be viewed as a stack that grows upwards, this happens when a goal requires another plan execution to reach success. Indeed, we can notice from Figure 3.2 that the agent, at some time during its execution, encountered the execution of the goal $b1$. But to satisfy the latter, goal $a1$, $a2$, $a3$, $a4$ and $a5$ must be executed. This results in the addition of a new queue on the top of the stack that will contain the required goals, respectively. In the example is also shown that the next goal that the agent will execute is $a1$ because is the next one in the queue on the top of the stack of the executed intention.

After a single goal is executed with success, the behaviour of the agent is to drop that goal. A more sophisticated design could save those dropped goals, this would lead to agent memory of which operation it has performed, enabling users to perform advanced queries. For example, we could query: "Did the agent perform X in the past". The main drawback which led to not preferring the introduction of this feature is the huge amount of memory that a single agent could require to save that information.

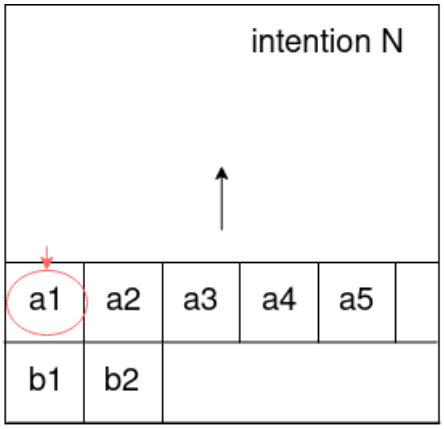

Figure 3.2: Intention stack graphic representation

Actions

Actions represent the operative capabilities of agents. Through action definition, users can build agent capabilities to operate on a wide range of scenarios. This is the main reason that led us to model a sufficiently generic entity for an Action. Just like Jason, there are two types of actions: *internal* actions and external ones.

Internal actions are visible only from the agent's perspective, indeed they can alter the agent's state through their execution, for example, they can model a dynamic plan provisioning for the agent. External actions are visible at the environment level, this means that all the agents situated in the same environment can use them. The latter type of action is used to operate over the environment state, for example, external actions can be used to add dynamically another agent into the same environment.

Actions definition in Jacop is a little more strict than Jason's one, indeed the latter applies all the changes performed to the agent (or environment) state during the action execution, instead Jacop provide a limited set of changes that can be performed during action execution (a sort of side effect) and other alterations are not considered. Developers must explicitly indicate wherever a change should be applied, then the actual change is performed by the Jacop framework.

Selection Functions

AgentSpeak(L) introduces the concept of selection functions, there are three types of them:

- *Event Selection Function.* The selection of which event the agent will manage is a duty of the event selection function. The latter takes the agent's set of events as a parameter and returns the one that the agent is going to satisfy.

- *Plan Selection Function.* Once the agent has computed all the applicable plans over an event that needs to be satisfied, only one of them can be committed for execution at each entity iteration. This selection is performed by the plan selection function.

- *Intention Selection Function.* Each agent must select one intention for execution, this choice is made by the intention selection function. Indeed, every agent could have multiple intentions that need to be executed at the same time and this algorithm could change completely the behaviour of the entity.

The three of them let users refine the reasoning behaviour of agents. Although there is a default implementation of each one of them, users should be free to customize them as they need, at the agent's definition time. Indeed, a simple change in one selection function could drastically change the course of action taken by the agent.

3.1.2 Environment

A multi-agent system has many agents that work together in a shared environment. The environment is the entity on which agents act and observe changes, this means that its design should be as generic as possible to let users shape it following the domain they want to represent. Indeed, an environment could be simulated or connected to real-world scenarios, both scenarios should be representable easily by users through this entity.

These motivations brought us to model the environment with an entity that is not operating directly on the MAS but is a shared object between agents. However, the framework must also support users' decision to turn the environment into an active entity. The latter is the reason why there's also a third component on Jacop MAS definition: the *execution strategy.*

The environment is also responsible for the management of agents' *messages*, indeed, agents are capable of communicating with each other thanks to this shared entity between them. There will be a queue of messages associated with each agent living in the environment, and the environment manages all of them.

Lastly, as a shared entity between agents, the environment is also the place where *external* actions are located. Whenever an agent wants to use an external action, it will query environment actions to check if there's an applicable operation.

3.1.3 Execution Strategy

The flexibility of Jacop framework to shape each scenario users want to create is also attributed to the *execution strategy.* This entity separates the concept of

model definition and its execution, making it possible to customize the machinery on which the framework is actually executed.

With the separation of these two notions, users can potentially run the same multi-agent system both on physical machinery using single-threads and also multi-threads. This will result in a fully customizable framework that enables users to adopt BDI agents on every machinery they have.

3.1.4 Agent State Machine

To manage agents' execution in a shared common way, between different execution strategies, an execution control layer is provided. This let users build an execution strategy flexibly because the execution of the framework becomes only the decision on when to start the execution of an agent.

We wanted to explicitly associate each agent with its life cycle, which passes through different states until it ends. We have chosen to call this component *Agent State Machine* (Agent SM).

Conceptually, any activity that must be performed – in our case the agent – will pass through an initial state in which it will perform a series of configuration operations, which are typically performed only once before the actual execution. An activity lifecycle finally ends when it reaches its final state. We can model this machine behaviour with a state diagram where all the state transitions are ruled by explicit routines invocations.

Figure 3.3 shows the graphical representation of the state machine of an entity. There are five reachable states during an activity lifecycle: `Created`, `Started`, `Paused`, `Paused` and `Stopped`. All the activities start from the created state, then they transit into started after a constant amount of time ϵ. After the transition is triggered, and the routine `onBegin` is invoked, the activity starts its concrete execution with `onRun` execution reaching the running state. Running the state where the activity belongs most of the time, as shown on Figure 3.3, because this is the state that each activity reaches without performing changing-state operations.

Four operations can trigger a change over an activity state: `restart`, `pause`, `resume`, `stop`. Apart from the first one, each one of the others can be called by the activity only if is currently in a specific state: pause and stop are invocable from paused and running state and resume is only invocable from paused state. Intuitively, paused state enables an activity to stop its execution and then resume it, while the stopped state precedes the end of the lifecycle. As for the startup of the activity, there's also a routing for the end of it called `onEnd`. The latter is run once before the end of the loop and after its invocation the execution ends.

This design for agents executions is quite generic to enable execution on a wide range of machinery. The key concept is that there are three *callbacks* that rule

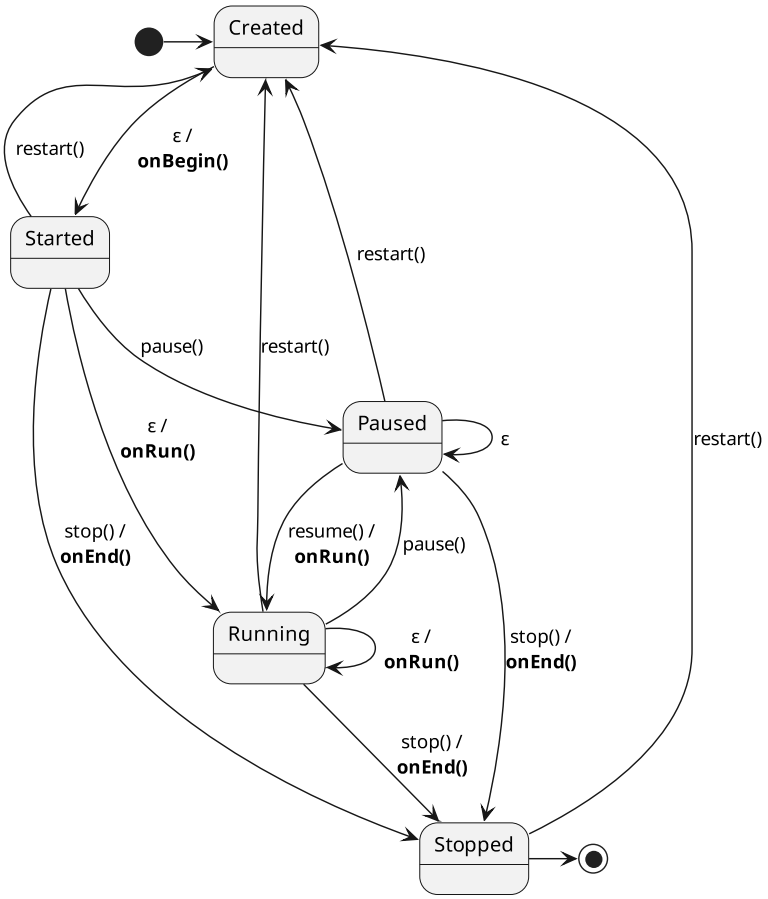

Figure 3.3: Agent State Machine graphic representation

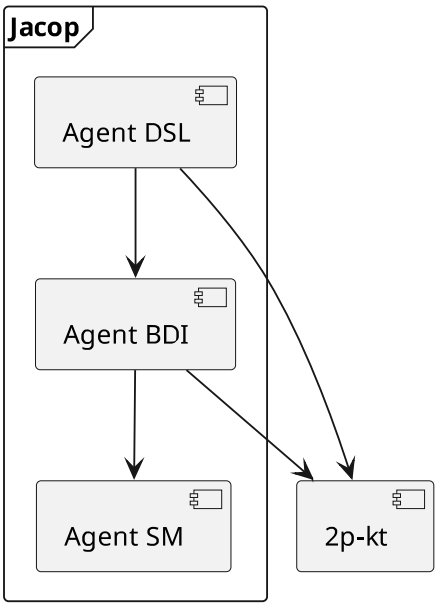

Figure 3.4: Mudules decomposition of Jacop library

the whole execution of agents: `onBegin`, `onRun` and `onEnd`. Each activity modeled using the three of them could be completely independent from the concrete execution paradigm.

3.2 Implementation design

This section explores the design choices made for the main components of Jacop framework described in Section 3.1.

In order to develop a clear framework and simple to debug, we decided to implement it with an *immutable* structure. An object is immutable if its state doesn't change once the object has been created. The immutable pattern increases the robustness of objects that share references to the same object and reduces the overhead of concurrent access to them.

As it results after the Domain Model section, each module of the library is designed to be completely independent from others. This distinction is highlighted from the project decomposition, as shown in Figure 3.4. From the users' perspective, modules mean that each one of them could be completely changed – following the interface definition – and the framework still works using the new implementation.

From the graphical representation, there are four modules that compose the Jacop framework: *Agent DSL*, *Agent BDI*, *Agent SM* and *2P-kt*.

The first one is the layer built over the model definition that defines the language users can use to facilitate their development using the library. This language is defined using *Domain Specific Language* (DSL) technology, which exposes only the main domain information that agents need to be specified to work properly. A DSL is also useful because it can be easier to understand for people that do not know how to use the language on which the library is implemented and it is also interchangeable within mainstream language files.

Agent BDI is the core module of the library, it contains the definition of Jacop interfaces and their implementations. This module implements all the BDI model entities described in Section 3.1.1 and contains the whole engine that enables the execution of the Multi-Agents System.

The Agent SM model contains entities that enable transparency between the model execution and the underlying infrastructure. These entities indeed are a form of abstraction for the technology used to execute the framework, enabling it to be executable in a single-thread, multi-threaded or simulated physical environment.

The last module is the one represented outside the Jacop jurisdiction, indeed, is the only external dependency of the library. The 2P-kt module is a well-established library that implements a Prolog logic language syntax.

The following sections describe the main choices made for entities definition and also the relationships between them, together with a detailed description of their expected behaviour.

3.2.1 Agent SM module

The state machine (Figure 3.3) is the engine that rules the state transitions that could happen during an entity's lifetime, in our case the entity is an agent. As described in Section 3.1.4, there are three ingredients needed to model it: a *state*, the *transitions trigger* and the *routines*.

Depending on which state is currently the agent and also depending on which course of transitions the entity selected, a different routine is run. Inside routines there is the definition of the whole behaviour of the entity, there are three of them: `onBegin()`, `onRun()` and `onEnd()`. As represented in Figure 3.3, the first is run once when the agent starts its execution, the second one is run repeatedly when the agent is running and the last one is also run once right after the agent stops its execution. Each entity modeled with these three routines is called *Activity*. As visible in the graphical representation, some state transitions are ruled by explicit operations run by the activity, that are: `restart()`, `pause()`, `resume()` and `stop()`. We modeled them as activity's *Controller*. The latter is the entity that gives, to users that define an entity, the ability to *control* state transitions.

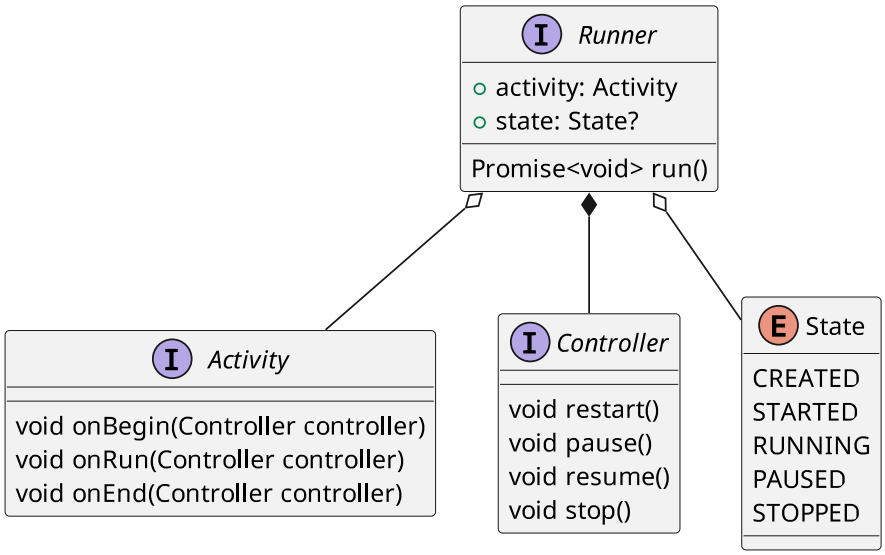

Figure 3.5: UML diagram of Agent SM main interfaces

An activity is only a formal description of entities' behaviour, to enable them to run on whatsoever machinery another component is essential: the *Runner*. A Runner embeds all the necessary code to model state transitions over all the possible scenarios, which makes it possible to completely abstract execution from entity definition. Runners manage the execution technology, the actual state, the routine execution and the change of state. This layer of abstraction is fundamental to make it possible to map an activity execution over any underlying machinery desired.

Figure 3.5 shows the interaction between the main entities inside Agent SM module, as described above. Firstly, to execute these entities, a user needs to define the expected behaviour of the activity, dividing it into the three routines that describes ad Activity, then he needs to choose which Runner will execute it, choosing between one of the available runners, or also implementing a new one by itself. As shown in the UML, after associating the activity to the chosen Runner, the latter exposes the method that must be called by users to finally run the entity.

3.2.2 2P-kt module

Jason's implementation is provided with an internal logic language engine that let users use this syntax within the agents' definition. Since a reliable implementation of this system component would require a large effort, we have decided to rely on the stable 2P-kt [5] library, which is flexible enough to be adapted to the library's needs. However, it is not possible to use it directly as the logical engine for the

library, because, even if AgentSpeak(L)'s agents definition is inspired by the horn clauses, the syntax is slightly different.

2P-kt's programs are called *theories*, which are lists of Horn Clauses. A Clause in prolog can be *Rule*, *Fact* or *Directive*.

A Rule is a predicate that holds true only if a number of other predicates also hold true. A Fact is a predicate that is known to hold in the domain concerned and a Directive is the goal of the logic program, which is the predicate that must be proven over a Prolog theory.

The syntax of a Rule is the following:

```
<HEAD> :- <BODY>.
```

Where `<HEAD>` is a single *Predicate* and `<BODY>` is a sequence of them. Predicates represent statements that can be true or not in the domain of the discourse.

Both rules and theories can be exploited to represent the agent's beliefs and belief base, respectively. Indeed, as described in the following sections, a belief is represented through a clause, in case it states something that the agent knows, and rules, that let the agent infer knowledge over multiple predicates that must hold in the domain concerned by the agent itself. Usually, the agent stores complex knowledge composed of a list of beliefs: this representation can is done using a 2P-kt theory entity.

3.2.3 Agent BDI module

The definition of all BDI model entities resides in the Agent BDI module. The latter is responsible for providing users with all the components needed to build a multi-agent system.

As shown in Figure 3.6 a MAS consists of three essential elements: An *Execution Strategy*, an *Environment* and a set of *Agents*. The main model decisions of the three of them are described below.

Agent

Within a MAS there is a set of agents, who cooperate to achieve a common goal, possibly by changing the state of the environment in which they live. Agents are the essential entities operating within the Multi-Agent System's environment. As described in Section 2.3.1, AgentSpeak(L) agents are composed of five elements: beliefs, goals, events, plans, and actions.

The execution of an agent is described by a static and a dynamic part, the latter is the one defined by users. The static part is part of Jacop's implementation and contains all the logic related to the entity execution, as well as the Agent Lifecycle.

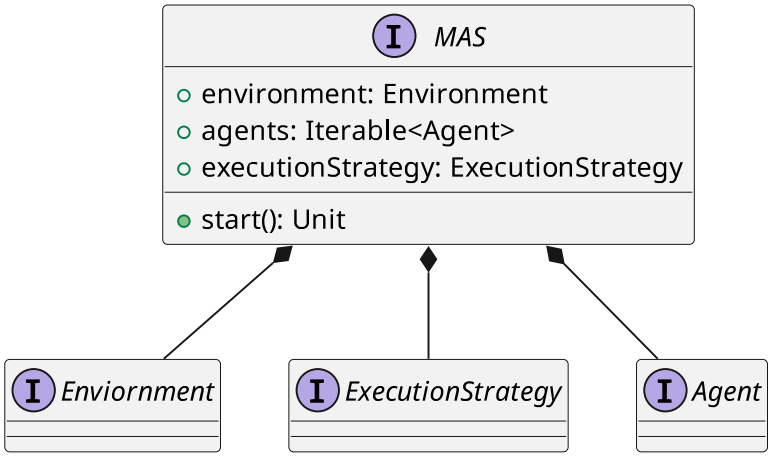

Figure 3.6: UML diagram of Jacop main interfaces

Users can specify agent behaviour in terms of BDI notions such as Beliefs, Plans and Goals, but they can also specify the three selection functions, as described in AgentSpeak(L) agent definition. Despite these functions have a trivial default implementation, users can refine them to obtain the desired Agent behaviour.

Beliefs. The first Beliefs modeling was thought of as 2P-kt *Clause*. The problem associated with this choice was that such an entity can be of three types: a *fact*, a *rule* or a *goal/directive*. Since a belief does not represent a goal that users want to verify, but statements of the domain known to the agent, the entity was then modeled as an extension of the *Rule* interface. This choice is optimal because it is possible to represent both a Rule and a Fact through the same entity; in fact, a Rule is nothing more than a Fact that is always verified in the domain concerned by the agent. As with Jason's implementation, a Belief can also contain within it a list of annotations. These annotations are useful to distinguish essential information about each piece of information believed by the agent.

Jacop's implementation introduced a simplified concept of annotation: they are used to distinguish the source of a belief using a single Struct, instead of a list of them as in Jason.

An agent does not store a single belief, but a set of them, so a data structure is needed to store this information within it in an optimal way. The data structure used for this task is the 2P-kt *MultiClauseSet*, however, this implementation – as all 2P-kt specific ones – is completely hidden inside BeliefBase implementation. A graphical representation of the interaction between these interfaces is reported in Figure 3.7.

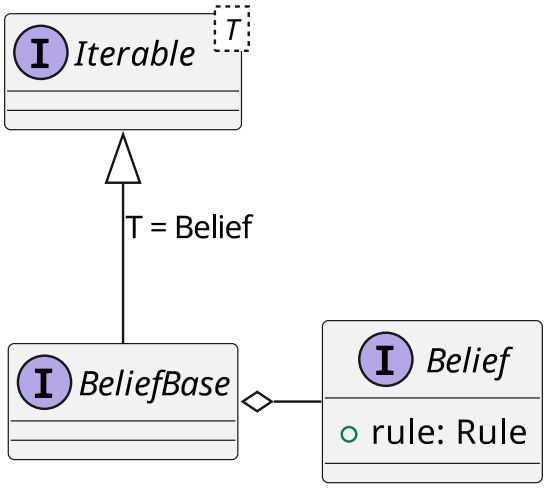

Figure 3.7: UML diagram of Agent's BeliefBase and Belief

Goals. Agent goals represent their capabilities, more specifically, all the operations they can perform within a Multi-Agent System. Indeed, each Agent's Plan contains a list of Goals that must be met to lead to the success of the overall Plan. Eight possible goals can be specified: *Belief Addition Goal*, *Belief Removal Goal*, *Belief Update Goal*, *Achievement Goal Invocation*, *Test Goal Invocation*, *Spawn Goal Invocation*, *External Action Goal Invocation* and *Internal Action Goal Invocation*.

The first three of them are all Goals that eventually modify the BeliefBase of the agent; respectively, they are the addition of a new belief, its deletion and the update of an existing one.

Achievement Goals are the ones that enable the agent's composite plans, indeed they generate a new event for the achievement of that goal that the agent will eventually satisfy. The Plan selected by the agent to meet the Achievement Goal will itself have a list of Goals to be met; the Agent will have to execute all of them before it can continue execution of those belonging to the initial plan. A different behaviour is instead provided by Spawn Goals, in fact, these latter create a new event to find the plan to be adopted to satisfy the Spawn Goal – in the same way as Achievement Goals – but the previously running Plan does not wait for the completion of all the Goals of the new Plan to continue its execution: these two will execute concurrently.

Test Goals can be adopted by users who need to retrieve information from the agent's BeliefBase, they fail if there is not any Belief that is a logical consequence of the goal's logic structure.

The Action goal invocation triggers the execution of an action from the agent.

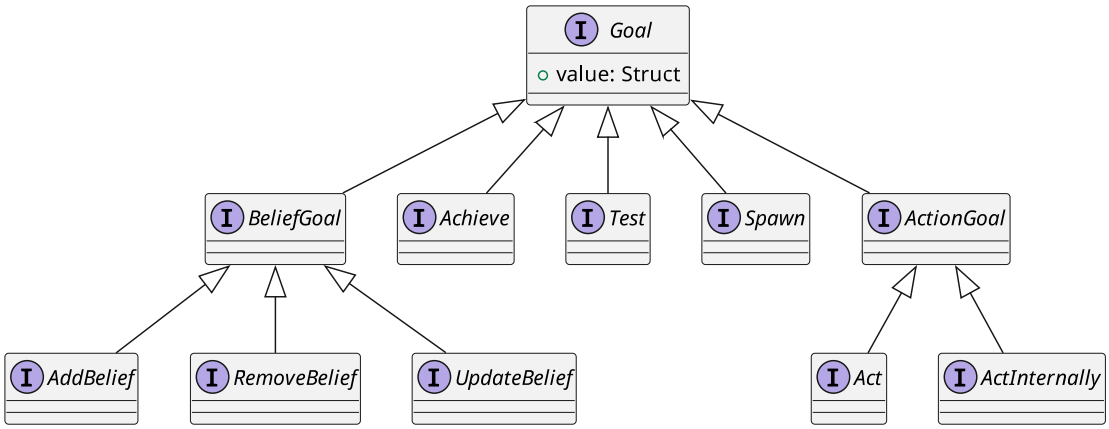

Figure 3.8: UML diagram of Goals interfaces decomposition

Actions group the ability of agents to operate; in fact, users can provide Multi-Agent System's entities with all sorts of capabilities through the definition of Actions. Actions can be of two types, *internal* or *external*; the differences between the two will be detailed in the next sections, but the goal distinction is necessary to discern from which set of Actions the one to be performed should be found by the Agent's execution engine.

The Figure 3.8 shows the decomposition of all the Goal types described above, it also shows the entity through which the goal will be met: a 2P-kt *Struct*. During the execution of a Goal, the agent will try to apply that structure according to the Goal it wants to satisfy: in the case of BeliefGoal sub-types and Test types it will search for an existing logical consequence of the Goal value in the agent's BeliefBase, for Achieve and Spawn the agent will look for an existing plan which trigger is unified with the Goal value and lastly, if it is an ActionGoal sub-type, it will search for an Action to execute which name is equal to the predicate of the Goal value.

Events. Events rule the occurrence of an Agent's Actions; in fact, only after an Event is raised then one Plan is selected to satisfy it. There are two types of Events: *internal* and *external*. Internal Events are generated by the Agent itself while executing one of its Intentions. This type of event keeps track of the Intention that triggered it since the latter cannot continue its execution until the event is satisfied (unless it is a Spawn Goal type that runs it concurrently). Six types of events can be generated during an agent's lifetime: *BeliefBase Addition*, *BeliefBase Removal*, *Test Goal Invocation*, *Test Goal Failure*, *Achievement Goal Invocation* and *Achievement Goal Failure*.

Intuitively, BeliefBase addition and removal are triggered when a Belief is re-

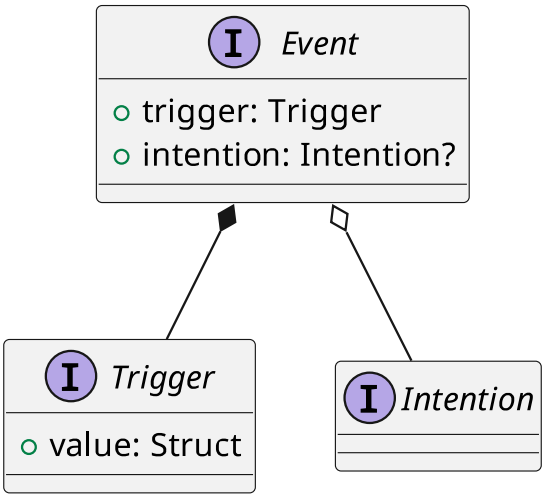

Figure 3.9: UML diagram of Event interface

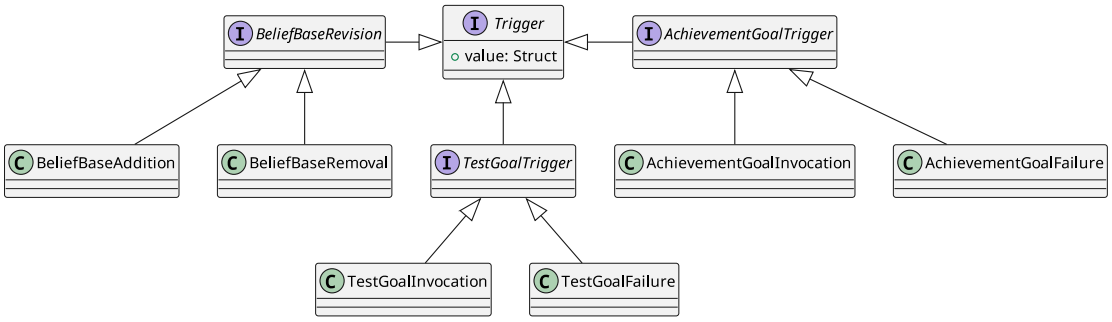

Figure 3.10: UML diagram of Trigger implementations

spectively added or removed from the Agent's set. These two Events are useful because enable the agent to a *reactive* behaviour because, for example, an Agent can react to a change in the Environment state with an appropriate Plan designed for it.

Figure 3.9 shows how an Event is defined: it could be an internal or an external one depending on the presence of the Intention, as explained above. The Trigger interface is the entity that identifies the trigger type that generates an event, which decomposition is represented in Figure 3.10.

Plans. Plans represent an agent's cognitive capabilities; in fact, they describe which actions agents are capable of executing based on the event that has occurred. As described in Section 3.1.1, a plan consists of three elements: a *triggering event*, a *context* and a sequence of *goals*.

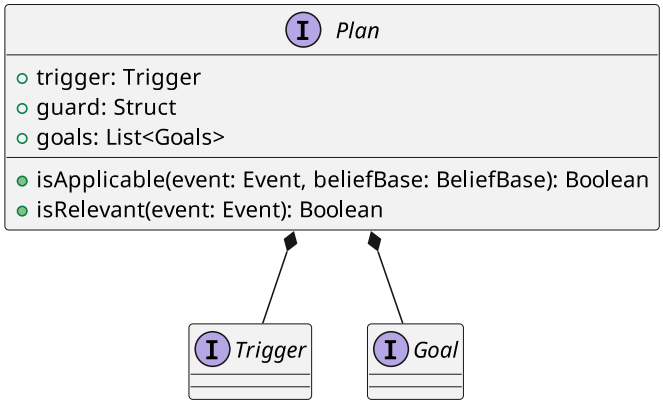

Figure 3.11: UML diagram for Plan interface

As visible in Figure 3.11, a triggering event is the same interface that describes the trigger of an Event. This is because Event triggers will be applied, during agent execution, to the plan's triggering event to determine its relevance. Once all relevant Plans are highlighted, their applicability is evaluated based on the knowledge the Agent has about the Environment at that particular time. This last check is made with the context of a plan, which determines if a Plan is applicable if its context is a logical consequence of the BeliefBase.

Once the set of applicable Plans is selected, the agent's engine will choose which one to apply, this means that its goals will be appropriately placed within an Intention, chosen based on the nature of the event: if the event is external it will be a new Intention, otherwise the one that generated the internal event will be retrieved and properly updated. The new Goals inserted within the Intention are *partially instantiated plans*, this means that all the variables contained within them – that were present in the triggering event and inside the plan's context – will be replaced with their value, while instead all others will be replaced during the execution of the goals.

Plan's context is represented as a single 2P-kt Struct as visible in Figure 3.11, this means that this component is a predicate that could hold or not in the domain concerned by the agent. A Struct is flexible because, with a single entity, it can model complex conditions with also logic operators such as *and*, *or* or *not*.

Intentions. Intentions represent the mental state of the agent, as described by the BDI model (Section 2.2). Intentions are not managed directly by users during the agent definition but are totally controlled by the agent's internal engine. An intention is created when a Plan is chosen to satisfy an external Action. Within an Intention, there is a set of partially instantiated Goals, indeed, variables contained inside Plans' body are substituted during the operation of evaluating the Plan

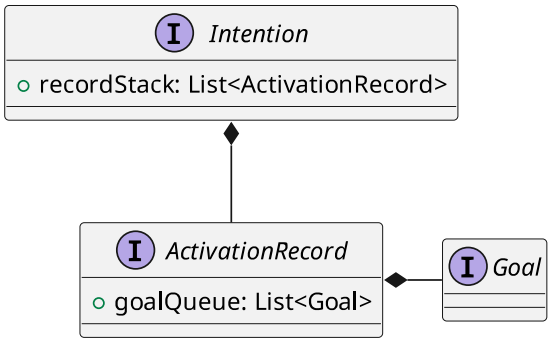

Figure 3.12: UML diagram for Intention interface

applicability, while variables for which a value could not be associated will be appropriately replaced during the execution of the Intention. The agent's lifecycle executes only one goal per intention; the Intention Selection Function is in charge to select the Intention – from the Intention Set – for which to continue the execution.

An intention, as visible in Figure 3.12 is composed of a stack of *Activation Records*. The stack is used to keep track of all the subgoals generated during the Intention execution; so that the agent is always able to know what is the next goal that it needs to manage in order to continue its execution. The motivation behind this data structure is intuitive: as soon as the agent encounters a new subgoal to satisfy, all of its Goals are inserted into a new Activation Record situated at the top of the stack. Consequently, an Activation Record is nothing more than a queue of partially instantiated Goals, derived from the application of the plan that was chosen by the Agent.

At each iteration of the agent lifecycle one goal is executed, taken from the first queued event in the Activation Record at the top of the record stack of the selected Intention; if the chosen one is an Achievement Goal then another Activation Record will be added to the top of the stack, otherwise, the chosen goal will be managed by the agent and removed from the queue.

Actions. Actions represent agents' capabilities, indeed, a user can simply extend what the agent is capable to do through the definition of customized Actions. Also, Actions are divided into two categories: *internal* and *external*. Internal Actions can only see the Agent's state which is executing them, and possibly modify it. To ensure that such changes do not break the agent's execution, users are only provided with a *defense* copy of the agent's state so that no changes are considered by the Agent engine. However, state modification is not forbidden, this can still be done through appropriate methods that can be invoked explicitly by users within the action. These methods will then be executed by the Agent engine, ensuring

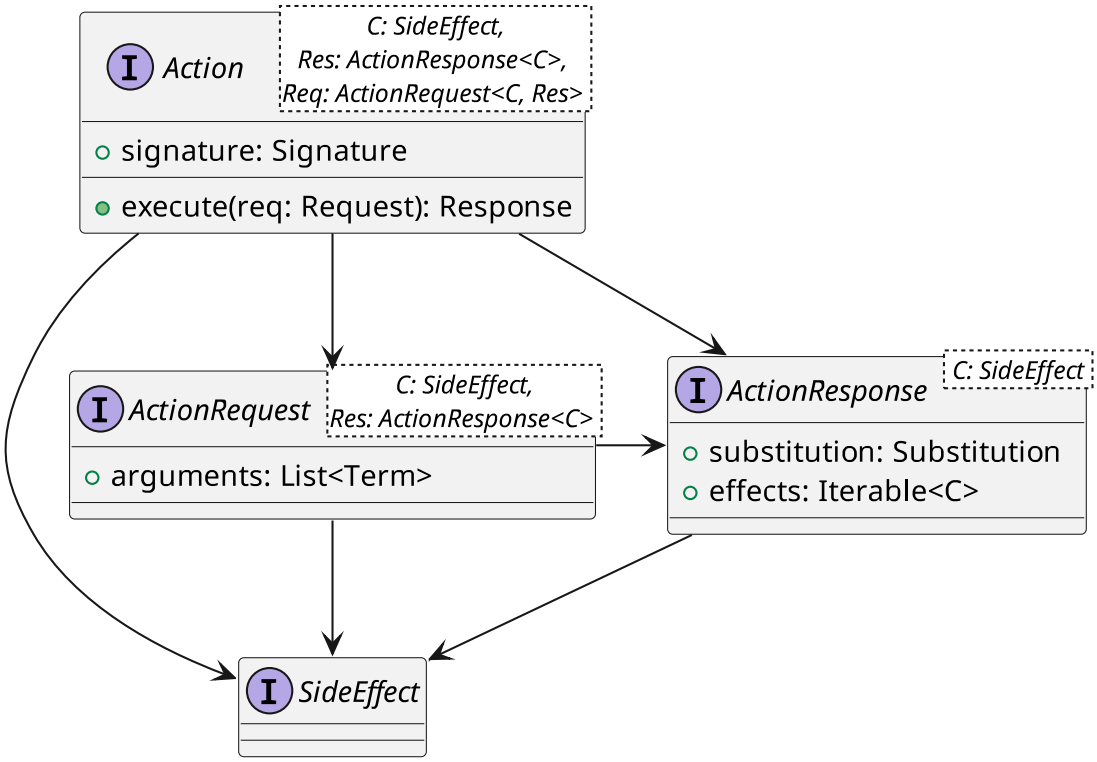

Figure 3.13: UML diagram for Action interface. Arrows represent assignments constraints on generic variables

that the change is safe as well as successful.

External actions follow the same reasoning, but instead of operating on the state of the agent, they only see the state of the Environment. A user, within an internal action, cannot view the internal status of the agent who is performing it, but can only know its reference. External actions are defined at the Environment level and, as it is shared, can be performed by all the agents located inside it.

As shown in Figure 3.13, an Action is designed to be strictly defined by three elements: *SideEffect*, *ActionRequest* and *ActionResponse*.

SideEffect is the interface that denotes the types of changes that can be applied by the Action, this entity enables the control over state changes described above. Figure 3.14 shows that SideEffect types follow the definition of the two types of actions, indeed, there is *EnviornmentChange* that enables to perform some changes over the environment and *AgentChange* that possibly modify the Agent's state. The changes that can be performed over the environment are: *(i)* addition of a new agent, *(ii)* removal of an existing agent, *(iii)* send a message and *(iv)* broadcast a message. The changes that can be performed over the agent's state are:

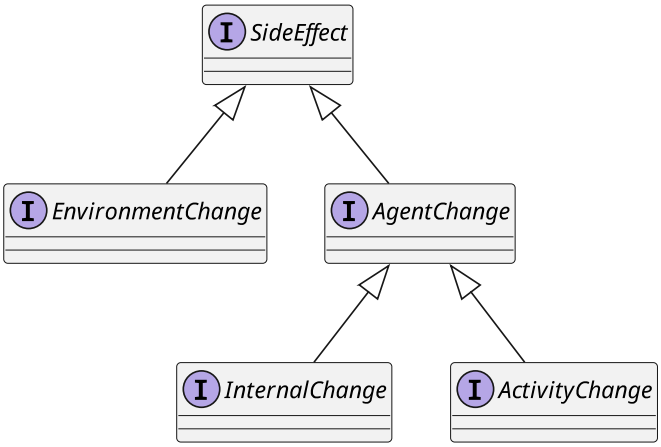

Figure 3.14: UML diagram for SideEffect interface

(i) addition/removal of a belief, *(ii)* addition/removal of an intention, *(iii)* addition/removal of an event, *(iv)* addition/removal of a plan, *(v)* stop the agent execution, *(vi)* pause the agent execution, *(vii)* pause the agent execution for a number of milliseconds. Beyond the previous description, there is another distinction inside AgentChange interface: *InternalChange* and *ActivityChange*. The first one represents changes over the Agent's BDI model state, such as its events, beliefs, plans and other elements. The latter denotes changes over the Agent SM's Activity that is currently running the entity, this enables users to modify the current running Activity execution through the Action implementation, for example, users can stop and resume the Agent's lifecycle.

The Action UML representation shows that the interface provides the `execute` method, which takes an ActionRequest and returns an ActionResponse: this method will execute the operations specified by users. Another detail captured by Figure 3.13 is that a Jacop Action definition strictly depends on which SideEffect is used because it also defines uniquely the ActionResponse, which in turn defines uniquely the ActionRequest. This means that users cannot define, for example, internal Actions provoking changes in the Environment (and vice versa), as shown in Figure 3.15.

During Agent Goals execution, Actions are invoked by users using a logic Predicate as if it is a function call, indeed, the predication – the name that defines the Predicate – represents the Action's name to be searched, while the arguments are themselves the arguments of the Action. This design requires that each Action stores the name to which it can be called and the number of arguments it expects as input: in fact, this information is saved within the signature field, which has the purpose of storing precisely these information.

The ActionResponse (Figure 3.16) is the type of data that is returned after the

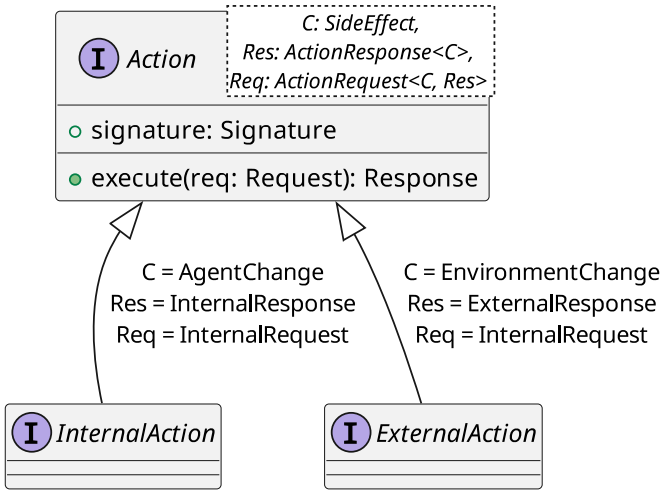

Figure 3.15: UML diagram that describes InternalAction and ExternalAction

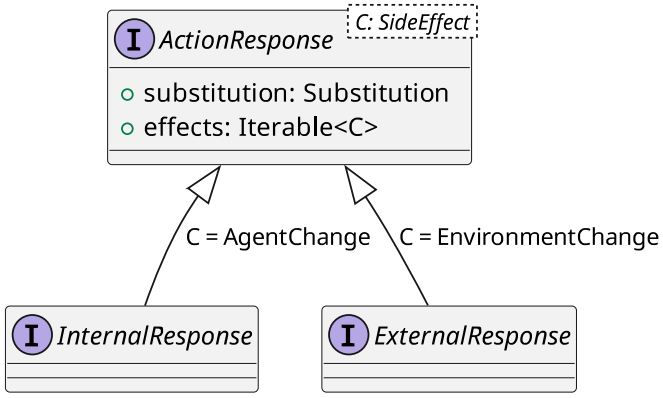

Figure 3.16: UML diagram for ActionResponse interfaces decomposition

invocation of the Action's `execute` method, indeed, an Action Response contains two elements: a 2P-kt substitution and a sequence of SideEffects. The substitution represents the result of the Action execution which can be true or false; it can also contain some mappings between variables and their associated value, that will be appropriately replaced in the subsequent Goals variables of the Intention. This means that the substitution field is useful for both to know if the Action was completed successfully and to retrieve possible execution results.

The effects, on the other hand, represent all the changes that the user wants to perform after the execution of the Action on the state of the Agent, in the case an Internal Response is being created, or on the state of the Environment, in the case of an External Response. This field is then used by the engine in charge of

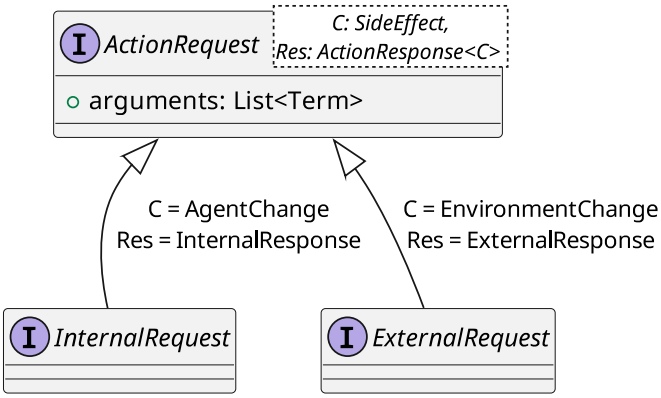

Figure 3.17: UML diagram for ActionRequest interfaces decomposition

running the agent to apply those desired changes.

Finally, once the type of SideEffect has been fixed, ActionResponse is uniquely defined and consequently also the type of the request. The ActionRequest represents the input passed to the method that performs the Action, consequently, it will contain a reference to the parameters that are passed by users during the Action invocation, as visible in Figure 3.17.

Agent Lifecycle. An Agent Lifecycle is nothing more than the engine that takes care of the entity execution. Indeed, at each iteration of the latter, a series of operations must be performed to guarantee the functioning of its BDI model.

The first operation that the agent must perform, at each iteration of its life cycle, is to observe whether any changes have occurred within the environment. Indeed, to make the agent as responsive as possible to these changes, before it decides which operations to perform, it must observe the current state of the Environment. Once the Agent knows the new state of the Environment, it must update its knowledge base, i.e. the BeliefBase, with the new information retrieved. This step is carried out by a *Belief Update Function*, which has the task of optimizing this procedure. In order to react to changes in the Environment state, and therefore to be a reactive entity, the agent must generate events. These events are determined by evaluating differences between its knowledge base and the new perception of the state around itself. Events can be of two types: addition or deletion of a Belief.

Another event that the agent must respond to is the receipt of a new message. Messages are the component that agents can use to communicate with each other within the Environment. This means that, at each iteration of the lifecycle, the Agent has to check if there are any new messages that it needs to handle. Messages can trigger two different scenarios related to their type, indeed, there could be

`Achieve` Messages that trigger a new Achievement Goal invocation Event or also `Tell` Messages that are handled exactly like adding a new Belief, the only difference with a generic one of them is given by the annotation, because it refers to the *sender* of that message. As with the perception phase, the latter type of Message also generate a Belief addition Event.

After checking all the Event sources that have occurred, the Agent selects one Event to handle. The latter is selected using the *Event Selection Function* defined within the Agent. If there is an Event to handle, then it is removed from the Agent's Event set and a suitable Plan is searched to fulfill it.

The Plan selection phase follows three steps. First, the Plans which have a triggering event that unifies with the selected Event are filtered among Plans available to the Agent, these are the *Relevant Plans* Then the applicable ones are selected, this is done by verifying that the Plan's context is a logical consequence of the agent's BeliefBase. Of the remaining Plans, the one actually applied is chosen using the Plan Selection Function. If it was possible to outline a plan that satisfies all these three steps, then this is assigned to an Intention.

If the Event that triggered the Plan application was external, then a new Intention, with the partially instantiated Goals of that Plan's body, is added to the Intention set. Otherwise, if the Event was an internal one, the Lifecycle retrieves the related Intention and adds another ActivationRecord, with those Goals inside of it, on the top of its stack.

Lastly, the Agent executes one Intention Goal. The Intention to execute is selected with the Agent's *Intention Selection Function*, and if there's at least one of them to execute then it's scheduled for execution.

The execution phase is where the agent could potentially perform changes over its state or on the Environment's one. However, Agent Lifecycle does not operate directly over the Environment state, as for Action's SideEffect, the whole reasoning procedure described above returns a list of EnvironmentChanges, which application will be managed by the Multi-Agent System ExecutionStrategy.

Environment

The Environment is the entity in which agents live, its state is changed by agents to achieve their goals. In the initial model definition of the library, the Environment is not foreseen as an active entity, which means that it does not have its own control flow, unlike the agents, but this entity is generic enough that users could choose to implement custom behaviour for it. Although its behaviour can be manipulated by users, this entity has only the goal of agents' synchronization, this means that the update of its state must be managed appropriately by users in multi-threaded systems, although immutable modeling helps to avoid critical runs.

The Environment encapsulates four important pieces of information: a set of

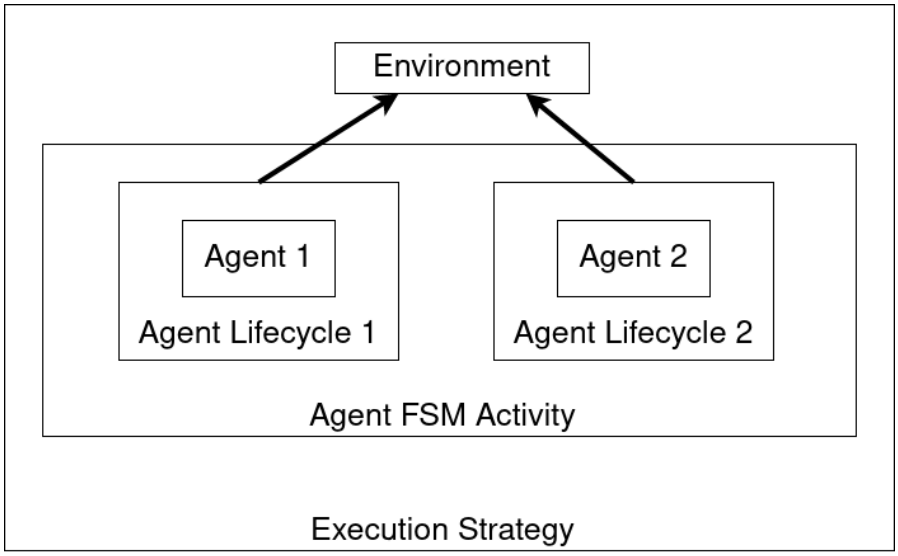

Figure 3.18: Representation of Execution Strategy behaviour

agents living within the Environment, Agent's mailboxes on which the others can send messages, the set of external Actions visible at Environment level and, finally, the component that returns the current state of the Environment itself, in terms of Belief.

Execution Strategy

The Execution Strategy is the element that builds the entities needed to execute the MAS, indeed, this entity is seen as the connection layer between Agent BDI module and Agent SM one. It also represents the synchronization point between the agents and the environment. Furthermore, while the Agent state machine implementation provides the abstraction of the physical machine on which the agent will be executed, the Execution Strategy has the task of establishing how and when to execute the agents inside it other.

The implementation of this entity is in charge of managing the update of the Environment state in the most appropriate way, indeed, the execution of the Agent Lifecycle does not modify it directly, following the philosophy of the immutable design. This means that without an explicit update management, the Environment would never percept the changes desired by the Agents.

The importance of this entity is visible in Figure 3.18, indeed, this is the entity that encapsulates all the Multi-Agent System's Agents to their Agent Lifecycle instance, and then deploy them on the proper Activity. The example in the figure represents an Execution Strategy implementation where Agents are running over

the same Activity and the Environment is shared between them. Although the Environment has been designed as a synchronization entity, in reality within an Execution Strategy a user can specify a different behaviour for this entity, indeed, it is possible to carry out an implementation in which the Environment is itself an active entity of the domain executed, for example, on a different Activity among Agents.

3.2.4 Agent DSL module

The entity definition described above is not intuitive enough to program BDI agents because it requires, in addition to some specific knowledge of the language used to implement it, advanced knowledge of the treated domain interfaces. For this reason, we decided to define a layer of abstraction that simplifies the Jacop adoption by users, i.e. a Domain Specific Language (DSL) inspired by the Jason syntax.

The DSL is built through the concept of *scopes*, the latter is nothing more than a builder of a certain object of the domain. Nested scopes let users create more complex objects transparently, enabling them to specify only the essential concepts.

As described in Section 3.2.3, a Jacop MAS is composed of three elements: an environment, a set of agents and an execution strategy.

Environment. Inside the environment entity, users can specify the external actions that can be executed by all the agents living within it. An external action can query the environment state and, possibly, change it through explicit side effects, that can be explicitly invoked during action execution. The environment state can not be directly modified by agents, this ensures that all changes are made safely without breaking the MAS execution.

Accordingly, an action response – i.e. the result produced by the action execution – contains the list of all the side effects that users want to perform over the environment. The list of these changes is then managed by the execution strategy implementation.

Inside an external action, users can perform side effects over the environment using the syntax described in Listing 3.1. Certainly, not only side effects can be performed inside of the action, the instructions specified in the Listing 3.1 can be alternated with standard Kotlin statements. Moreover, the figure shows that an action needs two additional pieces of information for its definition: a *name* and an *arity*. The name represents the string with which the action will be invoked by agents, the arity represents the number of parameters that the action expects in input.

Listing 3.1: External action side effects

```
environment {
    actions {
        action("name", /* arity */) {
            addAgent(agent)                    // adds a new agent in the env.
            removeAgent("agent")               // removes "agent" from the env.
            sendMessage("receiver", message)   // sends a message to "receiver"
            broadcastMessage(message)          // sends a message to all agents
        }
    }
}
```

Listing 3.2: Multiple external actions definition

```
environment {
    actions {
        action("first_action", /* arity */) {
            /* possibly contains a side effect */
        }

        action("second_action", /* arity */) {
            /* possibly contains a side effect */
        }

        ...
    }
}
```

Users can also describe more than a single external action inside the environment, this can be performed adding another `action` scope inside of the environment's actions, as shown in Listing 3.2.

Agent. An agent is described by at least two elements: initial goals and plans. The initial goals represent their purpose, agents are going to search for the most suitable plans to satisfy them. Initial goals can be of two types, achievement and test, and they can be described through the DSL as shown in Listing 3.3.

Plans describe all the capabilities that an agent can do, they present a triggering event, a context and a body of goals. The latter is a list of goals that the agent must execute to satisfy the overall plan, they can be described in the DSL as shown in Listing 3.4.

Plans' context represents the condition that must hold in order to execute the plan's body. This information is not mandatory, indeed, users can specify the `iff` block containing it or not, as visible in Listing 3.6. If they omit the context, it is simply treated as an always true predicate. Plans context is the information on which agents can discern applicable plans over relevant ones.

The trigger of a plan represents the event to which the plan responds, they can

Listing 3.3: Initial goals definition

```
goals {
    achieve("f")      // achievement goal invocation with "f" event trigger
    achieve("f"(X))   // achievement goal invocation with "f"(X) event trigger
    test("f")         // test goal invocation with "f" event trigger
    test("f"(X))      // test goal invocation with "f"(X) event trigger
}
```

Listing 3.4: Plan body definition

```
plans {
    + achieve("f"(X)) then {
        achieve("g"(X)) // achievement goal invocation with "g"(X) event trigger
        achieve("g")    // achievement goal invocation with "g" event trigger
        test("g"(X))    // test goal invocation with "g"(X) event trigger
        test("g")       // test goal invocation with "g" event trigger
        spawn("g"(X))   // spawn goal invocation with "g"(X) event trigger
        spawn("g")      // spawn goal invocation with "g" event trigger

        + "g"(X)        // belief base addition of "g"(X)
        add("g"(X))     // belief base addition of "g"(X)
        - "g"(X)        // belief base removal of "g"(X)
        remove("g"(X))  // belief base removal of "g"(X)
        update("g"(X))  // belief base update of "g"(X)

        act("g")              // execution of external action "g"
        act("g"(X))           // ... with one parameter X
        act("g"(X, Y, ...))   // ... with many parameters

        iact("g")             // execution of internal action "g"
        iact("g"(X))          // ... with one parameter X
        iact("g"(X, Y, ...)) // ... with one parameter X
    }
}
```

be defined using `+` or `-` depending on if the user wants to intercept the achievement of the failure of a certain event, all the possible combinations are shown in Listing 3.5.

Then, there are agents' pieces of information that can be added to its definition but are not mandatory, such as the initial belief base and the internal actions. The agent's beliefs can be specified both as rules and clauses, as shown in Listing 3.7. Agents can be customized with the addition of internal actions, these can be listed in the same way as the environment's `actions` scope, but the state changed by internal actions is the agent one. The side effects that internal actions can perform over the agent's state are listed in Listing 3.8.

Lastly, an agent can be composed of the elements described above as in Listing 3.9.

Listing 3.5: Different plan's triggers types

```
plans {
    // achievement goal invocation triggered by "f"(X)
    + achieve("f"(X)) then { /* plan's body */ }
    // achievement goal invocation triggered by "f"
    + achieve("f") then { /* plan's body */ }
    // achievement goal failure triggered by "f"(X)
    - achieve("f"(X)) then { /* plan's body */ }
    // achievement goal failure triggered by "f"
    - achieve("f") then { /* plan's body */ }

    // test goal invocation triggered by "f"(X)
    + test("f"(X)) then { /* plan's body */ }
    // test goal invocation triggered by "f"
    + test("f") then { /* plan's body */ }
    // test goal failure triggered by "f"(X)
    - test("f"(X)) then { /* plan's body */ }
    // test goal failure triggered by "f"
    - test("f") then { /* plan's body */ }

    // plan triggered by belief base addition of "belief"
    + "belief" then { /* plan's body */ }
    // plan triggered by belief base removal of "belief"
    - "belief" then { /* plan's body */ }
}
```

Listing 3.6: Plan context definition

```
plans {
    + achieve("f"(X)) then {
        /* plan's body */
    }

    + achieve("f"(X)) iff {
        (N lowerThan M) and (S `is` (N + 1))  // plan's context
    } then {
        /* plan's body */
    }
}
```

Listing 3.7: Initial belief base definition

```
beliefs {
    fact { "weather"("snowing") }           // a fact belief definition
    fact { "started" }                      // a fact belief without parameters
    rule { "s"("nat"(X)) impliedBy "s"(X) } // a clause belief definition
}
```

Listing 3.8: Internal action definition

```
actions {
    action("name", /* arity */) {
        addBelief(belief)           // adds belief into the belief base
        removeBelief(belief)        // removes belief from the belief base
        addIntention(intention)     // adds intention into the intentions set
        removeIntention(intention)  // removes intention from the intentions set
        addEvent(event)             // adds event into the events set
        removeEvent(event)          // removes event from the events set
        addPlan(plan)               // adds plan into the plan library
        removePlan(plan)            // removes plan from the plan library
        stopAgent()                 // stops agent activity execution
        sleepAgent(millis)          // pause the agent's execution for millis ms
        pauseAgent()                // pauses the agent's execution
    }
}
```

Listing 3.9: Agent definition

```
agent("name") {
    beliefs {
        /* beliefs definition */
    }
    goals {
        /* goals definition */
    }
    plans {
        /* plans definition */
    }
    actions {
        /* actions definition */
    }
}
```

Listing 3.10: Execution strategy definition

```
executionStrategy {
    // executes each agent on a different thread
    ExecutionStrategy.oneThreadPerAgent()
    // executes all the agents over the same thread
    ExecutionStrategy.oneThreadPerMas()
    // executes the multi-agent system in a discrete event simulation
    ExecutionStrategy.discreteEventExecution()
    // executes the multi-agent system in a discrete time simulation
    ExecutionStrategy.discreteTimeExecution()
}
```

Listing 3.11: Time distribution description for the agent

```
agent {
    ...
    timeDistribution {
        Time.continuous((it as SimulatedTime).value + 5.0)
    }
    ...
}
```

Execution Strategy. The execution strategy represents how the multi-agent system is executed because maps agents' execution to their runners. In the actual version of Jacop framework, users are provided with four implementations of execution strategy, but they can also define a custom implementation of them following their needs. In both cases, users can specify with execution strategy to adopt inside their multi-agent system using the DSL, as shown in Listing 3.10.

If users want to run the MAS using the discrete event simulation execution strategy they also need to specify another piece of information: the agent's time distribution. The latter can be implemented inside agents as in the Listing 3.11, in this case, the actual time is simply added to a constant number.

MAS. After the description of all the single components of a multi-agent system, its definition is made as in Listing 3.12.

Thanks to the DSL, users can freely define a Multi-Agent System in its entirety or compose it of its individual parts, indeed, all the information can also be defined within variables and inserted transparently within the `mas` scope.

Listing 3.12: Multi-Agent system definition

```
mas {
    environment {
        /* environment definition */
    }
    agent("agent1") {
        /* agent definition */
    }
    agent("agent2") {
        /* agent definition */
    }
    executionStrategy {
        /* execution strategy definition */
    }
}
```

Chapter 4

Implementation

This chapter describes the development phase of the library, in particular, it focuses on the implementation of some of the main components of the system. This chapter also describes the most relevant implementation choices made during the development of the framework.

During the definition of a totally abstract execution model from the physical entity that executes it, we noticed that it could be abstract in its entirety. This enables the framework to be used within *simulation tools* transparently. In this way, a user can define a Multi-Agent System independently from the physical entity that runs it, dynamically deciding whether to run it in production or over a simulated environment. There are various opportunities for simulating Multi-Agent Systems, precisely because they can be applied in a wide range of application domains.

The reference language for the implementation of Jacop is Kotlin, the choice is made not only because nowadays it is a mainstream language, but also because it can be integrated into projects written in more popular languages such as Java, thanks to the underlying JVM. Another reason that led to this choice is the simplicity that this language enables for the definition of a Domain Specific Language. Thanks to the DSL, also users that are not confident with Kotlin development can still decide to integrate a Multi-Agent System within their project without using language-specific features.

4.1 Agent State Machine

The first implemented system module is the state machine to manage the execution of a generic Activity. This implementation is essential to abstract between the

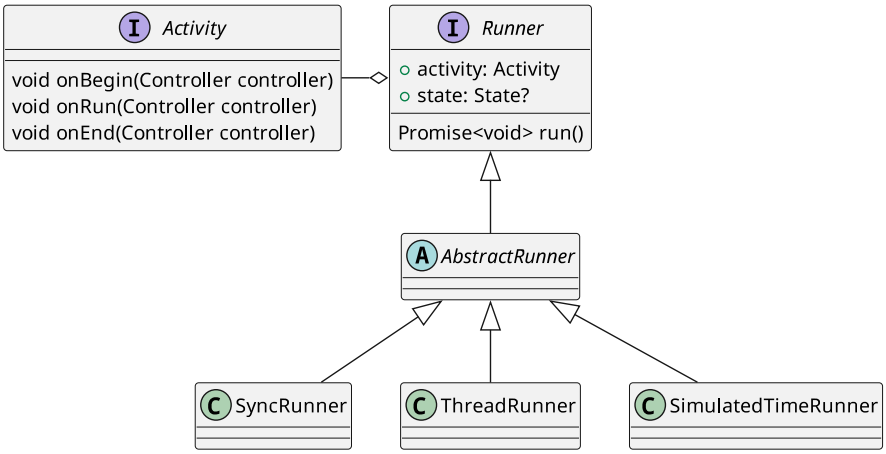

Figure 4.1: Agent Final State Machine implementations

entity definition and the physical machine on which it is actually executing.

At this abstraction layer, users define an agent through an Activity, i.e. using the three designed routines: `onBegin()`, `onRun()` and `onEnd()`. To run an Activity, users must associate it with a Runner. A Runner implementation contains all the low-level implementations necessary to let the entity be executed as desired by the user.

This module is designed in such a way that the user can, according to his needs, use an already implemented Runner, through static factory methods, or define a customized one following the definition of its interface.

The library provides three basic implementations of the runner interface, as shown in Figure 4.1. The first, *SyncRunner*, runs the Activity on the current thread and manages *time* as the same of the physical machine's one on which it is running. *ThreadRunner*, on the other hand, is developed to run the Activity on a separate thread and manages time like the previous one. The last one, *SimulatedTimeRunner*, also runs on the current thread but abstracts the concept of time, *simulating* it.

The notion of time within the state machine is also abstracted: this enables agents to know which is the current time perceived by them during actions execution, whether it is real or simulated.

Figure 4.2 shows Time definitions inside Agent's Runner, there are two types of time, *EpochTime* and *SimulatedTime*, respectively to describe it as the one seen by the physical machine and as the one simulated. This is also visible from the value stored by those entities: EpochTime wraps a value with *Long* type, which is the standard used to represent the current timestamp in milliseconds for computing machines, and SimulatedTime stores a *Double* value and its meaning depends on

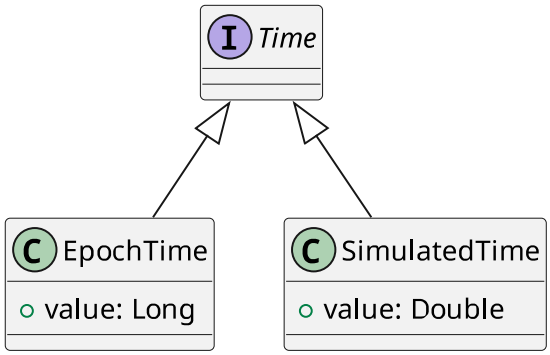

Figure 4.2: Time abstraction inside the Agent's Runner

the chosen simulated execution strategy.

If the three implementations provided differ from the user's requirements he can extend the AbstractRunner, the abstract class will guide him in the implementation of a customized Runner that can be used within the Multi-Agent System.

4.2 BDI Agents

Once the implementation of the state machine was consolidated, we started the implementation of the core module of the library. This module is the one that took the longest to define clearly, in fact, it has been conveniently improved several times.

The package organization within the module tries to hide the implementation details as much as possible, showing users only the interfaces that they can query to better understand the entities. However, users can still use those implementations thanks to static factory methods suitably inserted within the interfaces. The implementation of these objects was bottom-up, which means that the individual components of the Multi-Agent System were first implemented and then subsequently integrated to let users define and execute them.

To make the definition of an agent more clear to the user we decided to separate its behaviour definition from its state. A user observing the Agent interface (Figure 4.3) notices that it is made up of an AgentContext, which is its BDI state representation.

The Figure 4.3 shows that an agent is also an extension of the 2P-kt Taggable interface, this is useful because enables Agent entities to store key-value information transparently.

Lastly, an Agent needs the definition of its behaviour through the definition of the three selection functions. The Event Selection Function is specified with

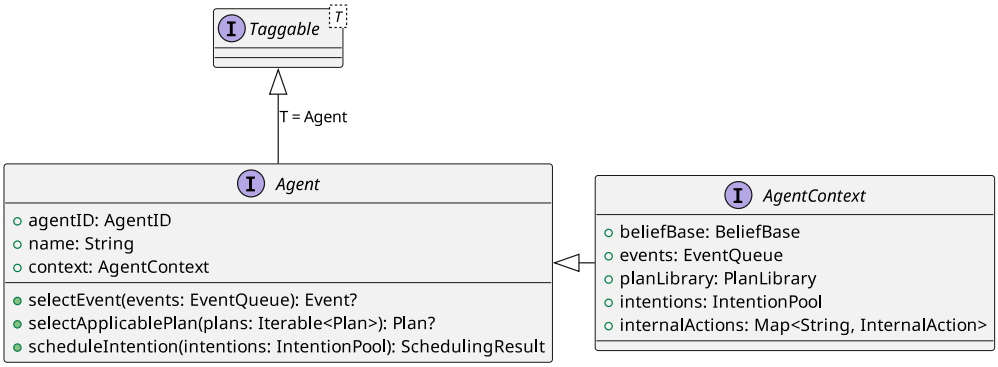

Figure 4.3: Agent representation through Agent and AgentContext interfaces

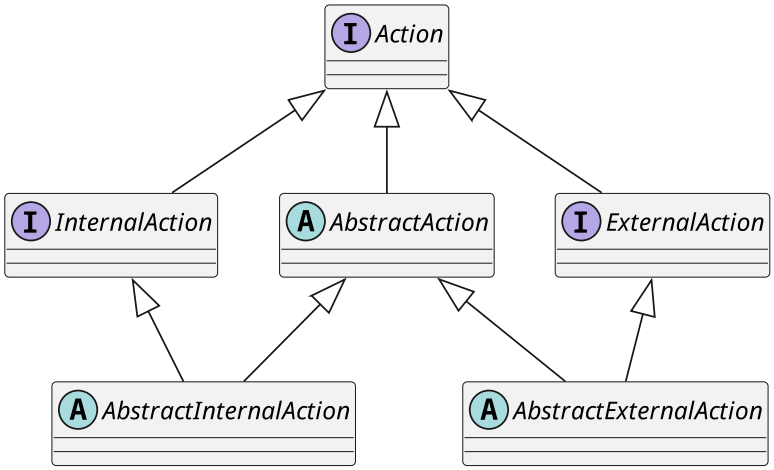

Figure 4.4: Actions implementation overview class diagram

`selectEvent` method, Plan Selection Function with `selectApplicablePlan` and Intention Selection Function with `scheduleIntention`. Jacop proposes a default implementation of the Agent interface, that implements the selection functions trivially: they simply return the first element in the list.

From what concerns Agent actions, their implementation required a lot of effort to be able to guarantee that a user can create them following his needs but without breaking the constraints defined by the type of Actions. Internal Actions cannot operate on the state of the Environment and, otherwise, external Actions cannot interact with the Agent's AgentContext. The design phase has, indeed, made sure that the types used to define an action are consistent with each other, leading to the class decomposition figured in Figure 4.4.

To define an Action, a user must extend the appropriate abstract class, this

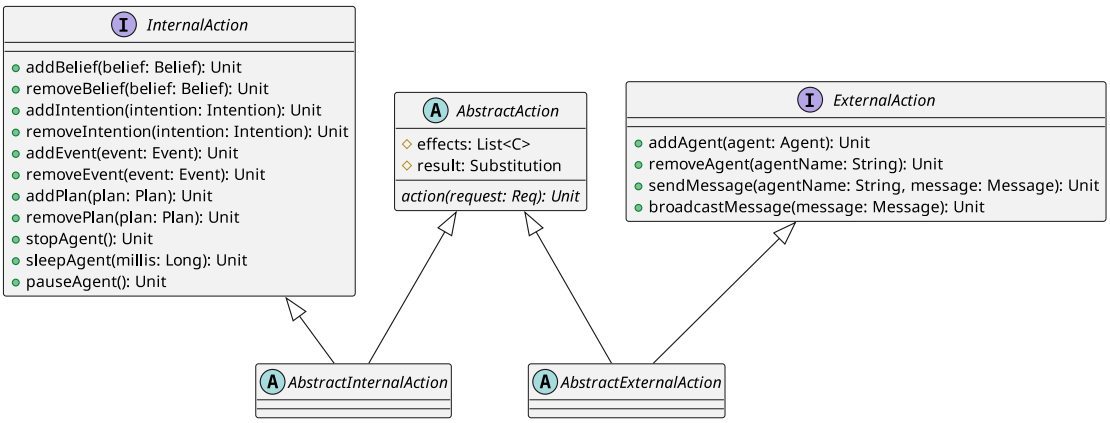

Figure 4.5: Actions implementation detailed class diagram

means that if he wants to implement an InternalAction he will have to extend an AbstractInternalAction, and vice versa. The reason to extend abstract implementations is that they provide the methods that the user can use to change the state of the referenced entity. This means that changing the state inside the action implementation is not enough to propagate it, users must specify such changes with the provided methods invocation, otherwise, they will be ignored by the Agent Lifecycle.

The Action performs the operations specified within the abstract `action` method, which is the only one that the user must implement when extending an abstract action. The detailed Figure 4.5 also shows the constraints over Action execution side effects, they differ a lot depending on the entity on which they operate.

During the implementation phase, it was decided to not equip Agents with a large number of predefined Actions, indeed, only five simple internal Actions are provided to let users test the library and no external Action is implemented by default. These actions are useful for testing purposes because enable users to understand how Jacop agents works, and are: *print*, *fail*, *stop*, *pause* and *sleep*.

This choice makes it possible to have a totally generic purpose framework, while still making its complex implementation transparent, which can be useful for a lot of different scenarios. The result of what is described above is visible in Listing 4.1, where only the essential information are specified by users, such as the name of the action, its arity – i.e. the number of parameters it expects in input – and lastly its body.

As regards the ExecutionStrategy entity, this component represents the meeting point between the BDI domain interfaces and those of the state machine. As for the Actions, only a few implementations are provided for this entity, they can be

Listing 4.1: Example of an ExternalAction implementation using the library

```
object : AbstractExternalAction("name", 0) {
    override fun action(request: ExternalRequest) {
        println("External Action body")
    }
}
```

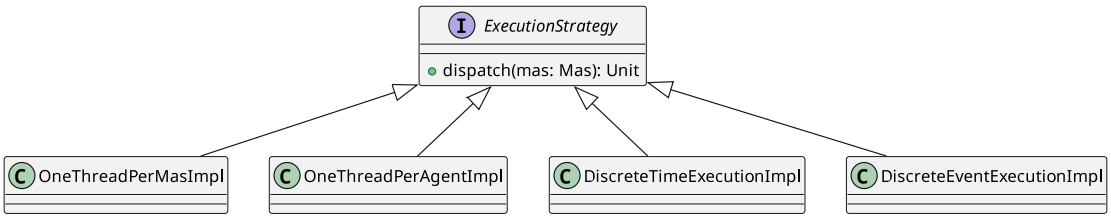

Figure 4.6: Execution Strategy implementations

suitably extended according to the needs of the users.

Jacop is provided with four Execution Strategy implementations. The first one executes all the agents over the same Activity, so it means that executes them on the same thread. The second one executes each agent over a different Activity, enabling the MAS concurrent execution. The Discrete Time Execution and Discrete Event Execution implementations enable the framework to run on simulated scenarios. The last two of them are described in the following section.

4.2.1 About simulation-oriented execution strategies

A computer simulation is a program that is a computer program that executes step-by-step a mathematical model to explore its approximate behaviour [14]. The most relevant aspect of the simulation is the modeling of time, indeed, the simulation tools can be divided into two categories [9]:

- *Time-driven simulations*: In these simulations, time is simulated through discrete time slots where at every tick the model is updated to the new state.

- *Event-driven simulations*: During these simulations, each event is executed one by one and after the execution of each one of them the time is shifted forward. An event is an instantaneous change in the value of one or more state variables, performed during a bounded procedure.

In order to enable the user to choose the most appropriate simulation technology for his use-case, we decided to totally abstract the concept of time perceived by the running agent. Indeed, as visible in Figure 4.2, time is discerned in two types inside the agent SM Activity: EpochTime and SimulatedTime. While the former represents an instant of time as seen by the physical machine, the latter represents the time value managed during the simulation execution.

This abstraction let users decide freely which type of simulation to adopt for their MAS execution: in the first one events depend on the passage of time, while in the second one, the time value depends on events occurrences.

For this scope, two additional execution strategy implementations are provided in Jacop: Discrete Time Execution and Discrete Event Execution. Both of them run the entire Multi-Agent System on the same thread and they also abstract the concept of time, instantiating it appropriately within the Activity's SimulatedTime structure. In discrete time execution, the time is simply a counter that is incremented each time the Activity has executed a single step of all the agents' lifecycles. The discrete event execution, instead, calculates what is the next time value depending on which agent is executed. This means that, when running the latter, it is necessary to specify – in each agent – what is the *probability distribution* that the next iteration will occur. Time distribution is nothing more than a function that takes an instant of time as input and returns the time when the next event will occur. Users can specify this information on agents, enabling the execution strategy to determine which are the next agents to be executed in the next iteration of the Activity.

4.3 Domain Specific Language

The last concept developed is the Domain Specific Language. Kotlin, thanks to its functional tendency, enables the DSL definition by exploiting higher-order functions and operator overloading.

We can see each domain entity as a different *scope*, indeed, each object needs a different number of elements required to define them. The user is helped to describe correctly object behaviour, because each scope shows him the information that can be specified inside of it. These scopes are nothing more than object Builders, in fact, each one of them exposes a set of methods to describe the object behaviour and then the build method is invoked, the latter returns an instance of the object customized with the user's preferences.

For example, a MAS consists of three elements: Environment, ExecutionStrategy and the Agents' list. We can imagine a MAS as a scope in which the builders of Environments, Agents and ExecutionStrategy can be invoked. Those builders are different scopes themself because, for example, an Agent can in turn contain

Beliefs, Goals, Plans, etc. and so on. This scope hierarchy leads to the generation of a Multi-Agent System using the described DSL.

The result obtained from language implementation differs slightly from the initial idea thought during the design phase. This is due to the language chosen for the implementation which enable overloading only for a subset of operators, in which there are not some that were intended to be used within the library. The next chapter describes how to use the language that has been produced.

Chapter 5

Validation

During the definition of the domain entities, we implemented unit tests through the *Kotest* library, which let us determine whether the entities' behaviour was working as expected. These tests also ensure that all new feature introductions do not compromise the functioning of the previously defined domain entities.

However, once the library reached a certain maturity, we chose to test its functionality (and execution) with several Jacop MAS definitions, inspired from Jason examples. These applications, in addition to verifying the integration of the various components, provide the end user with an example of how the language features can be exploited to build a MAS. These examples are also useful because they can represent a starting point for the development of more complex applications.

The following sections describe three different MAS descriptions through the Jacop DSL syntax, in order they describe a multi-agent system that: *(i)* performs a plan's recursion, *(ii)* exploits external actions to perform the ping-pong scenario, *(iii)* exploits the simulated execution strategy, *(iv)* exploits the multi-threaded execution strategy.

5.1 Agent's plan recursion

To develop an Agent it is necessary to describe at least one Goal and a Plan that can satisfy it. This can be done using the syntax shown in Listing 5.1 using the Jacop DSL.

This first application shows a trivial use case of the library, that is a print of values from 0 to 10, however, it let us clearly understand many details of the Jacop language. In fact, agent Alice is described with a single goal and two plans that can achieve it. The designated goal is of the type *achieve*, meaning that she will try to fulfill it through the application of one or more plans, based on those provided by the user. In the agent description, two plans are described: the first is triggered

when the two parameters of the `start` predicate – specified within her goal – have the same value, and the second one is when they differ. Therefore, in the first iteration of her lifecycle, the only relevant plan is the second one, because the triggering event match with her goal. Then, the agent looks at whether the plan is also applicable, checking if its context is a logical consequence of its knowledge and, since it is, then the plan is scheduled for execution.

The plan's context in this application shows that it works just like a standard Prolog theory, but it also concatenates predicates using the `and` construct. The execution of that plan contains two goals, the invocation of an internal action and the achievement of another goal. The internal action executes one of the agent's built-in actions that perform a print over the console, this is helpful for testing scenarios – like the one in consideration. After printing the actual value of the variable `N`, the achievement of the next goal is triggered. We can notice, from the plan's context in Listing 5.1, that the value adopted for the next goal invocation is incremented by one unit, provoking the plan invocation that prints the incremented value, and so on. The last plan invocation happens when `S` variable reaches value `10`, this will trigger the first plan that does not generate any other events, ending the Agent execution.

5.2 Ping-Pong multi-agent system

A complete application that also shows the interaction between the agents living in the environment is the Ping-Pong MAS. This multi-agent system contains two agents, described in Listing 5.3 and Listing 5.4, who exchange a message within the environment (Listing 5.2) in which they live.

Listing 5.2 shows that the environment scope is now specified within the Ping-Pong multi-agent system, when in the previous example it was not necessary. The entity definition is now necessary because specifies an external action that let agents communicate with each other. Indeed, without the presence of the Environment, the exchange of messages between agents would be impossible, because they are delivered and queued within this entity. For this reason, an action with the name `send` is defined within the environment and expects two input arguments: the receiver's name and the message content. These arguments can be manipulated within the action body by calling the `arguments` method, as shown in Listing 5.2. Then, to carry out the effective delivery of the message inside an agent's queue, it is necessary to use one of the SideEffects described in Section 3.2.4 to be able to modify its state, in this case `sendMessage` is used.

Within the Ping-Pong MAS definition two agents are defined, the first one is named *pinger* (Listing 5.3), while the other is named *ponger* (Listing 5.4).

Pinger knows only two pieces of information: it knows who should execute at

Listing 5.1: Finitely recursive agent MAS definition using Jacop

```
mas {
    agent("alice") {
        goals {
            achieve("start"(0, 10))
        }

        plans {
            + achieve("start"(N, N)) then {
                iact("print"("Hello World!", N))
            }

            + achieve("start"(N, M)) iff {
                (N lowerThan M) and (S `is` (N + 1))
            } then {
                iact("print"("Hello World!", N))
                achieve("start"(S, M))
            }
        }
    }
}
```

Listing 5.2: Environment definition for Ping-Pong MAS using Jacop language

```
environment {
    actions {
        action("send", 2) {
            val receiver: Atom = argument(0)
            val message: Struct = argument(1)
            sendMessage(
                receiver.value,
                Message(this.sender, Tell, message)
            )
        }
    }
}
```

a given moment and it knows the name of the other agent existing within the environment. This agent contains a single goal, which is to send the ping, that tries to satisfy it using the plans that the agent is provided with. In order, the first plan is executed, which in turn will trigger the execution of the third, which finally sends the message to the other agent. When Pinger receives the reply from ponger, it will print a string on the console acknowledging its receipt. An interesting thing that can be seen from this implementation is that the receipt of the message is seen as the addition of a new belief within the knowledge base of the agent, this is highlighted by the plan that manages its reception, which is triggered from an event of new belief addition.

Finally, Listing 5.4 shows the ponger agent, which definition looks a like the pinger's one but is slightly different. In fact, the latter does not have a goal that must be satisfied, but it only reacts to a message receipt with a confirmation printed on the console.

As can be seen from the examples above, none of them highlight the type of execution strategy that the multi-agent system must adopt. This is because, by default, all multi-agent systems are defined following the strategy that allocates the same thread for all agents defined within the system.

5.3 Simulated execution strategy

Users might want to modify multi-agent system execution, perhaps choosing a simulated one. Jacop enables to change easily the way agents run, as visible in Listing 5.7.

Listing 5.3: Pinger Agent definition for Ping-Pong MAS using Jacop DSL

```
agent("pinger") {
    beliefs {
        fact("turn"("me"))
        fact("other"("ponger"))
    }
    goals {
        achieve("send_ping")
    }
    plans {
        + achieve("send_ping") iff {
            "turn"("source"("self"), "me") and "other"("source"("self"), R)
        } then {
            update("turn"("source"("self"), "other"))
            achieve("sendMessageTo"("ball", R))
        }

        + "ball"("source"(R)) iff {
            "turn"("source"("self"), "other") and "other"("source"("self"), R)
        } then {
            update("turn"("source"("self"), "me"))
            iact("print"("Received ball from ", R))
            -"ball"("source"(R))
            iact("print"("Pinger hasDone"))
        }

        + achieve("sendMessageTo"(M, R)) then {
            iact("print"("Sending message ", M))
            act("send"(R, M))
        }
    }
}
```

Listing 5.4: Ponger Agent definition for Ping-Pong MAS using Jacop DSL

```
agent("ponger") {
    beliefs {
        fact("turn"("other"))
        fact("other"("pinger"))
    }
    plans {
        + "ball"("source"(S)) iff {
            "turn"("source"("self"), "other") and "other"("source"("self"), S)
        } then {
            update("turn"("source"("self"), "me"))
            -"ball"("source"(S))
            achieve("sendMessageTo"("ball", S))
            achieve("handle_ping")
        }

        + achieve("handle_ping") then {
            update("turn"("source"("self"), "other"))
            iact("print"("Ponger has Done"))
        }

        + achieve("sendMessageTo"(M, R)) then {
            iact("print"("Sending message ", M))
            act("send"(R, M))
        }
    }
}
```

We can highlight this with an additional example: Listing 5.5 describes a MAS where the agent named Alice recursively prints on the console the current time perceived during the action's execution. This agent is running using the default implementation for the execution strategy – i.e. one thread for all the mas agents – that in this example is made explicit to appreciate the pluggability of the DSL.

If we run the Listing 5.5 we get an infinite list of time values – in milliseconds – taken from the machine that is executing the agent. Indeed, the output obtained from this mas execution is shown in Listing 5.6: it shows that each time is of type EpochTime with a long value, that represents the timestamp.

If the execution strategy is changed, as in Listing 5.7, the output of the MAS execution is different. Indeed, the latter example modifies only how the MAS is executed, but the entity definition is the same as Listing 5.5. The execution of this multi-agent system will obtain Listing 5.8 as result, which shows that the agent percept time differently than the previous one, but in a transparent way. This output is actually showing that its time is of type SimulatedTime, with a value that represents the time computed inside the execution strategy implementation.

A detail captured from the latter output is that the simulated time value perceived by the agent is not sequential, indeed, it prints it every two iterations. This happens because the agent's lifecycle executes only a single intention's goal at ev-

Listing 5.5: One thread per multi-agent system execution using Jacop DSL

```
mas {
    agent("alice") {
        goals {
            achieve("time")
        }
        actions {
            action("time", 0) {
                println("time: ${this.requestTimestamp}")
            }
        }
        plans {
            + achieve("time") then {
                iact("time")
                achieve("time")
            }
        }
    }
    executionStrategy {
        ExecutionStrategy.oneTreadPerMas()
    }
}
```

Listing 5.6: Execution output of Listing 5.5

```
time: EpochTime(value=1678287490881)
time: EpochTime(value=1678287490939)
time: EpochTime(value=1678287490976)
time: EpochTime(value=1678287491008)
time: EpochTime(value=1678287491040)
...
```

Listing 5.7: Discrete time execution using Jacop DSL

```
mas {
    agent("alice") {
        goals {
            achieve("time")
        }
        actions {
            action("time", 0) {
                println("time: ${this.requestTimestamp}")
            }
        }
        plans {
            + achieve("time") then {
                iact("time")
                achieve("time")
            }
        }
    }
    executionStrategy {
        ExecutionStrategy.discreteTimeExecution()
    }
}
```

Listing 5.8: Execution output of Listing 5.7

```
time: SimulatedTime(value=0.0)
time: SimulatedTime(value=2.0)
time: SimulatedTime(value=4.0)
time: SimulatedTime(value=6.0)
time: SimulatedTime(value=8.0)
...
```

ery iteration. As visible in Listing 5.7, indeed, the plan's body contains two goals: the first one executes the internal action, and the second one generates the event for the achievement of another goal, managed in the next iteration of the lifecycle.

The main difference between simulated executions and others is, as shown from the results, the abstraction of time. Users can choose which execution strategy adopt in their multi-agent system transparently, but the semantics of time value is different. EpochTime will always be a different number, even after restarting the MAS execution, but SimulatedTime value starts from zero each time the execution is restarted. This is because the first one represents a timestamp, while the second is a value that is relevant only during the system's execution.

5.4 Multi-threaded execution strategy

Another interesting application that shows the execution strategy pluggability of Jacop is the dynamic decision wether to adopt a single-threaded execution strategy or a multi-threaded one.

We discern two examples that show this feature: a multi-agent system composed of two agents – named alice and bob – that print their current thread during an external action execution named `thread`.

The only difference between Listing 5.9 and Listing 5.11 is that they run, respectively, on a single-threaded and a multi-threaded environment, as visible in their execution strategy specification. It is interesting that, without changing the way on which these agents are described, they can be executed transparently over many architectures. This is visible from the result of their execution: the output of the execution over the same thread – reported in Listing 5.10 – shows that both agents are running over the same thread, while the output of the `oneThreadPerAgent` execution strategy – visible in Listing 5.12 – shows that the two agents are running on different threads, as well as expected.

Listing 5.9: MAS definition running on a single thread and printing its agents' thread name

```
mas {
    environment {
        actions {
            action("thread", 0) {
                println("${this.sender} thread: ${Thread.currentThread().name}")
            }
        }
    }
    agent("alice") {
        goals {
            achieve("my_thread")
        }
        plans {
            + achieve("my_thread") then {
                act("thread")
            }
        }
    }
    agent("bob") {
        goals {
            achieve("print_thread")
        }
        plans {
            + achieve("print_thread") then {
                act("thread")
            }
        }
    }
    executionStrategy {
        ExecutionStrategy.oneThreadPerMas()
    }
}
```

Listing 5.10: Listing 5.9 execution result

```
alice thread: Thread-0
bob thread: Thread-0
```

Listing 5.11: MAS definition where each agent is mapped to a different thread and prints its thread name

```
mas {
    environment {
        actions {
            action("thread", 0) {
                println("${this.sender} thread: ${Thread.currentThread().name}")
            }
        }
    }
    agent("alice") {
        goals {
            achieve("my_thread")
        }
        plans {
            + achieve("my_thread") then {
                act("thread")
            }
        }
    }
    agent("bob") {
        goals {
            achieve("print_thread")
        }
        plans {
            + achieve("print_thread") then {
                act("thread")
            }
        }
    }
    executionStrategy {
        ExecutionStrategy.oneThreadPerAgent()
    }
}
```

Listing 5.12: Listing 5.11 execution result

```
bob thread: Thread-1
alice thread: Thread-0
```

Chapter 6

Conclusions

This work led to the development of a framework for developing a Multi-Agent System following the BDI modeling and the abstract syntax proposed in AgentSpeak(L). The detailed design has contributed to obtaining a highly customizable and pluggable result, in order to contain an easy adoption in all areas in which agent-based modeling can be exploited to solve a problem of a complex nature.

The developed framework let users customize and plug each component, to be able to adapt it efficiently to any execution model needed by them, in addition to those already provided by the library. Thanks to Jacop, users can define a Multi-Agent System composed of agents and environments, together with a description of how they should be run.

The abstraction of the various components that make up the framework has enabled the language to be applied both in physical environments, in which it reflects real behaviour, or equivalently in a simulation, where the entities simulate their conduct with respect to what would happen in reality.

Finally, the potential of the framework can be exploited thanks to a Domain Specific Language. Indeed, the defined language simplifies users to adopt agent modeling avoiding knowing all the detailed implementations. This language can be used both inside of a single file or moduled in more of them transparently, simplifying the decomposition of the various elements with which the Multi-Agent System is composed.

We hope that our contribution will involve other research groups besides ours and that, in the future, it can be expanded to meet completely the needs of Multi-Agent Systems developers.

6.1 Future Works

Jacop implementation represents a way to be able to efficiently separate the execution of the Multi-Agent System and its association on a physical machine, through a simple DSL provided by the library. However, some details have been simplified in order to concentrate the design on the key aspects concerning the modeling of the library.

As introduced in the previous sections, some concepts could be engineered more subtly to exploit MAS features as efficiently as possible by its users to represent real situations. This also comprehends the Beliefs' annotations, that in the current implementation is possible to represent only one of them, constrained to the source of the piece of knowledge. Users may want to insert additional information over this entity, for example, an evaluation of the reliability of the belief, the latter enables users to describe more sophisticated conditions to execute plans.

Another interesting feature with which the framework could be extended is the ability to keep track of the operations performed by agents. This addition would let users carry out conditions that are much more complex and that come closer to modeling a real behaviour, for example, the agent could be queried if it has performed a certain action in the past. However, we decided not to implement this feature because it would burden the execution of the MAS framework, in fact, each agent would require a large amount of memory to be able to memorize its current and past execution statuses.

Lastly, the actions' execution could be improved because, at the moment, every step of the agent's execution runs all the code specified in the body of the actions. This implementation could be optimal only if it is guaranteed that the user never puts too heavy computations inside of an agent action, or that it even never ends. However, there is no guarantee that users will comply with this convention, which could lead to a loss of the MAS's performance during its execution. For this reason, users could model a feature that let them perform actions asynchronously for Jacop agents, they would be invoked to perform heavy workload without damaging the performances of the whole system execution. The design of these actions is not trivial, because developers have to consider that they never have the awareness, at the action level, that the action is running on single-threaded or multi-threaded machinery. For this reason, it must be properly engineered.

These are just some of the things that could be added to the library. They were drawn from external opinions regarding the MAS execution, which at the time of the design we decided not to consider.

Bibliography

[1] Rafael H Bordini, Jomi Fred Hübner, and Michael Wooldridge. *Programming multi-agent systems in AgentSpeak using Jason*. John Wiley & Sons, 2007.

[2] Michael E. Bratman. *Intention, Plans, and Practical Reason*. Harvard University Press, November 1987.

[3] Roberta Calegari, Giovanni Ciatto, Viviana Mascardi, and Andrea Omicini. Logic-based technologies for multi-agent systems: A systematic literature review. *Autonomous Agents and Multi-Agent Systems*, 35(1):1:1–1:67, 2021. Collection "Current Trends in Research on Software Agents and Agent-Based Software Development".

[4] Roberta Calegari, Giovanni Ciatto, and Andrea Omicini. On the integration of symbolic and sub-symbolic techniques for XAI: A survey. *Intelligenza Artificiale*, 14(1):7–32, September 2020. Special Issue for the Twentieth Edition of the Workshop 'From Objects to Agents'.

[5] Giovanni Ciatto, Roberta Calegari, and Andrea Omicini. 2P-Kt: A logic-based ecosystem for symbolic AI. *SoftwareX*, 16:100817:1–100817:7, December 2021.

[6] Ali Dorri, Salil S. Kanhere, and Raja Jurdak. Multi-agent systems: A survey. *IEEE Access*, 6:28573–28593, 2018.

[7] Riccardo Guidotti, Anna Monreale, Salvatore Ruggieri, Franco Turini, Fosca Giannotti, and Dino Pedreschi. A survey of methods for explaining black box models. *ACM Comput. Surv.*, 51(5), aug 2018.

[8] David Kinny, Michael P. Georgeff, and Anand S. Rao. A methodology and modelling technique for systems of BDI agents. In Walter Van de Velde and John W. Perram, editors, *Agents Breaking Away, 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, Eindhoven, The Netherlands, January 22-25, 1996, Proceedings*, volume 1038 of *Lecture Notes in Computer Science*, pages 56–71. Springer, 1996.

[9] Richard K Nance. On time flow mechanisms for discrete system simulation. *Management Science*, 18(1):59–73, 1971.

[10] Anand S. Rao. Agentspeak(l): BDI agents speak out in a logical computable language. In Walter Van de Velde and John W. Perram, editors, *Agents Breaking Away, 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, Eindhoven, The Netherlands, January 22-25, 1996, Proceedings*, volume 1038 of *Lecture Notes in Computer Science*, pages 42–55. Springer, 1996.

[11] Anand S. Rao and Michael P. Georgeff. Modeling rational agents within a bdi-architecture. In James F. Allen, Richard Fikes, and Erik Sandewall, editors, *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR'91). Cambridge, MA, USA, April 22-25, 1991*, pages 473–484. Morgan Kaufmann, 1991.

[12] Anand S. Rao and Michael P. Georgeff. BDI agents: From theory to practice. In Victor R. Lesser and Les Gasser, editors, *Proceedings of the First International Conference on Multiagent Systems, June 12-14, 1995, San Francisco, California, USA*, pages 312–319. The MIT Press, 1995.

[13] Yoav Shoham. Agent-oriented programming. *Artif. Intell.*, 60(1):51–92, 1993.

[14] Eric Winsberg. Computer Simulations in Science. In Edward N. Zalta and Uri Nodelman, editors, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Winter 2022 edition, 2022.

[15] Michael Wooldridge. Intelligent agents. *Multiagent systems: A modern approach to distributed artificial intelligence*, 1:27–73, 1999.