UNIVERSITY OF MACEDONIA
DEPARTMENT OF APPLIED INFORMATICS
GRADUATE PROGRAM

# A Prolog Meta-Interpreter for AgentSpeak(L)

M.Sc. THESIS

*of*

Dimitrios Dimitriadis

Thessaloniki, June 2019

# A Prolog Meta-Interpreter for AgentSpeak(L)

**Dimitrios Dimitriadis**

Hellenic Military Academy (Evelpidon), 2008

B.Sc. in Business Administration, Department of Accounting & Finance
University of Macedonia, 2014

M.Sc. Thesis

submitted as a partial fulfillment of the requirements for

THE DEGREE OF MASTER OF SCIENCE IN APPLIED INFORMATICS

Supervisor: Assist. Prof Ilias Sakelariou

*Approved by examining board on ….. 2019*

# Abstract

Many studies and surveys have focused on software agents and agent-oriented programming. This study proposes an approach towards agent programming, bridging the gap between the theory and the practical issues concerned with software agents' programming. Based on the BDI (Belief Desire Intention) model, AgentSpeak(L) is considered as a language with a remarkable contribution to the ideas in the field. The Prolog meta-interpreter for AgentSpeak(L) proposed in the current project, tries to "exploit" all the features of modern logic programming system implementations in order to provide a practical implementation of AgentSpeak(L), that can be used in numerous applications. Staying close to the syntax defined by Rao in the original paper (1996), the current meta-interpreter is capable of parsing and executing programs written in AgentSpeak(L), and with small modifications other similar programming languages based in the agent oriented programming paradigm. SWI-Prolog has been chosen as the implementation platform of choice, since it is very actively developed, with wide adoption in the research community. The interpreter's design heavily exploits multithreading, focusing on the features of autonomous and parallel execution. The proposed meta-interpreter also supports communication between agents, allowing encoding of agents capable of exchanging messages following the FIPA standards.

## Key Words and Phrases

Agents, Multi-Agent Systems (MAS), Agent-oriented programming, Logic Programming, Artificial Intelligence, Agent-programming languages, AgentSpeak(L), Jason interpreter, BDI model, Prolog, SWI-Prolog, Meta-interpreter, Multithreading, Agents' Communication, FIPA Standards, Failure Handling.

# Acknowledgements

Reaching the end of a demanding period, lasting almost a year, I strongly believe that writing this note of acknowledgements, is the finishing touch on my thesis. Working on such a project, was a great challenge for me, not only in the scientific arena, but also on a personal level. Writing this thesis has had a big impact on me and I would like to reflect on the ones who have supported and guided me throughout this period.

I am deeply indebted to my supervisor Assistant Professor Ilias Sakelariou for his fundamental role in this project. Ass an original pedagogue, he provided me with every bit of guidance, advice, assistance and expertise that I needed to write this M.Sc. thesis. It is a blessing for every student to come across a brilliant academic with such a pleasant benevolent character.

This thesis though would have been impossible without especially my wonderful beloved wife and my three precious little ones, which are always a source of inspiration for me, reminding me of the bright side of life. I would like to thank them for all the love, support and continuous encouragement throughout this period of study and the process of researching and writing this thesis. A piece of each one of you is present in this work. Thank you all.

# Table of Contents

# List of figures

# List of tables

# CHAPTER 1

# Introduction

Software agents have gained much attention among the research community in the last years. It is considered as a technology with many applications both in industry and in research field. With characteristics as autonomy, reactivity and social ability, software agents are capable of acting independently without requiring supervision or guidance from another higher level entity. However, the idea of inventing machines capable of acting like humans, has been accompanying people years before the inception of software agents. Thus, there has been developed a whole range of different theoretical models that could be used, such as BDI model, as well as languages that satisfy efficiently the requirements of multi-agent systems. So far, AgentSpeak(L), is considered as the most commonly known approach to a concrete language in agent-programming. Jason, an actively developed interpreter of AgentSpeak(L), is written in Java, seems like a contradiction, since AgentSpeak(L) was introduced as an extension of Logic Programming.

In the current project a Prolog meta-interpreter of AgentSpeak(L) is presented, capitalizing on the features of logic programming that seem to be very close to the ideas of agent-oriented programming. Tackling the challenge of implementing a robust and efficient framework for software agents, under which they would be able to act autonomously and communicate with each other, this project aims to combine the features of modern logic programming with the agent-oriented systems.

The proposed meta-interpreter stays close to the original syntax of AgentSpeak(L) as well as extends the set of features, implementing new ones, such as strong negation, belief and plan annotations, complicated mental rules and failure handling, as those were proposed in later research in the AgentSpeak(L) model.

The communication mechanism is based on the Pedro platform, a subscription/notification system. Based on the server-client model, agents are able to exchange messages directly and in an asynchronous way. The established communication network is managed from an independently implemented entity, the registry, which is responsible for updating the agent list, including all the active agents that take part in the Multi-Agent system.

## 1.1 Aim & objectives of the thesis

This thesis aims to provide a comprehensive approach towards agent programming, bridging the gap between theory and the practical issues concerned with software agents' programming.

Software agents and agent-oriented systems gained a great amount of attention in the last years. Since there is a bunch of agent-oriented languages, as well as proposed models and frameworks it is still under research, which should be the features of a robust and efficient agent-oriented platform.

In this framework, theory and modern logic programming applications seem to be relative to software agents and their features. Real-world conditions and parallel execution compose a demanding environment for the agents, as it takes significant effort for an implementation to achieve monitoring, managing and controlling their actions.

This thesis intends to work out these challenges, aiming to propose a robust and efficient framework for software agents, enhancing their capabilities. The meta-interpreter proposed in this project aims to provide such a framework.

## 1.2 Thesis Structure

A brief description of the chapters of this thesis is provided below.

Chapter 2 describes software agents, focusing on their most indicative attributes, as they are referenced in the respective literature. It also presents, a categorization of agents concerning the basic types such as reactive agents, model-based, goal-based and utility-based ones. The chapter includes a short survey, of the most common in use agent oriented languages and concludes with a description of the main features of AgentSpeak(L) and Jason.

Chapter 3 presents the technological background of the proposed meta-interpreter. An extensive analysis of the syntax of AgentSpeak(L), is also provided. A significant number of examples are presented, with the aim to describe the use of the language and especially the attributes implemented in the current project, such as strong negation, the belief and plan annotations and the support of complicated mental rules and higher order Prolog predicates.

Chapter 4 describes the mechanism that the parser adopts in order to translate an AgentSpeak(L) program, using Prolog terms. A thorough explanation is given, along with the use of examples, concerning the internal representation of the basic components of the AgentSpeak(L), such as beliefs, mental rules and plans. A special section is dedicated to describe the function of the solver. A set of examples is used to present the mechanism for the execution of achievement and test goals, goals pursued as separate intentions, belief addition/deletion goals, plans triggered by belief addition/deletion, as well as the execution of expressions and Prolog predicates. The chapter also presents the implemented failure handling mechanism and concludes with the presentation of the multithreading technique that is used, concerning the agent's design.

Chapter 5 refers to agents' communication. Multithreading is considered as the main technique in use, concerning the overall communication handling mechanism. The messages that the agents exchange, follow the principals of ACL. The implemented set of performatives is presented, such as *tell, untell, achieve, test* and *ask*. The description of message processing, illustrates the significant role of multithreading in the overall agent's design. Concluding this chapter, a set of test cases is presented, highlighting the function of each one of the implemented types of messages that the agents are capable of exchanging.

# CHAPTER 2

# Literature review

This chapter provides a survey of literature, in the area of agent-oriented programming. It starts by giving a definition of software agents, highlighting the most representative features that agents adopt. Consequently, the chapter presents different types of agents as they are defined in bibliography. Special focus is given to the BDI model and its interpretation cycle, noticing the model's influence to the agent-oriented systems. The next section intends to present the main features of the recently proposed agent oriented languages. AgentSpeak(L) is explicitly analyzed, considered as a language, with a remarkable contribution to the ideas in the field. Also, a special section refers to Jason, an interpreter of AgentSpeak(L), with wide adoption in the research community and similar philosophy with the interpreter proposed in this project.

## 2.1 Software Agents

There is much discussion among the research community, about the definition of agents and the attributes that clearly distinguish them, from any other kind of program (Franklin and Graesser, 1996). However, the central idea is to create advanced programs, capable of acting on our behalf.

The dominant attribute in agents is autonomy, which allows agents to operate in complex environments (e.g. planets' exploration where human's presence or remote control are not feasible), to perform exhausting, time-consuming and repetitive tasks in a fast and efficient way.

Searching for the most representative features, that distinct agents from other programs, one could highlight the autonomy, the social ability, the reactivity and the pro-activeness, in line with the definition of (Wooldridge and Jennings, 1995). Hence, agents are self-contained programs, that could operate, requiring neither supervision nor guidance from another entity, they are able to communicate and as a result, to coordinate their actions and cooperate with each other, aiming at the achievement of desirable goals. In other approaches, we could find more features, imputed to agents, such as mobility, self-adaptivity, known also as self-* properties and rationality (Wooldridge, 2003).

Agents are able to build “societies”, cooperate and negotiate with each other in a peer relationship, in order to achieve system’s goals. This kind of society is called Multi-Agent System (MAS). The great challenge for such systems, is the efficient management and control. Among the characteristics of a MAS, we should highlight the asynchronous computations, decentralized data, lack of a central control mechanism and locality, meaning that no agent has global view of the system.

## 2.2 Basic types of Software Agents

With respect to the structure of the agent, we could define four basic types of agents, such as simple reflex agents (i.e. reactive), model-based (i.e. agents with internal state), goal-based and utility-based ones. This categorization (Russell and Norvig, 2016), p.47) ranges from the simple to the intelligent model. Having said that, we should stress the fact that there is no useless model and it depends on the purposes and the nature of the system, in order for the developer to choose among the one agent type or the other.

In a real-world example, consider that unmanned solar vehicles are equipped with sensors and effectors and controlled by agents. According to the information given from the sensors, the agent generates its actions, with respect to a set of rules, while each type of agent adopts a different kind of behavior.

Simple reflex agent’s actions, are based on the current environment percept. So for example, one can naturally imagine a rule that dictates the appropriate safety distance that the vehicle should keep from the other vehicles or obstacles, and the agent through its effectors accelerates or decelerates accordingly, respecting the safety distance limit. The agent, of course, is responsible to cope with a large number of such rules, thus an intelligent behaviour emerges.

Agents with internal state, have to match not only the rule conditions with the environmental state in order to decide for the next action, but with their internal state as well. Such agents are capable of maintaining a percept history, based on what they have experienced before and may not be observable at the current state. So, following the previous example, the agent is able to keep track of other solar vehicles’ position and thus avoid accidents, while changing its course.

One level above, a goal based agent is able to select the action that brings it closest to achieving its goals. The latter describe the desired system state. A goal based agent incorporates search and planning capabilities, while the decision for its next action takes into consideration, to an extent, any consequences of these actions. This makes the decision making process more complicated, since agents have to predict future changes, while they may have to cooperate with each other and coordinate their actions. Following the previous example, agents are able to find a plan to reach a given destination, if they are tasked to do so.

In case that there are alternative actions that all satisfy the goal, it is preferable to choose the most "efficient" one. Utility-based agents are capable of comparing different solutions for the same problem, estimating certain parameters, finding the most desirable way to achieve the goal. Thus, a utility-based agent in the previous scenario, could search for the fastest route or the safest one, in case it was tasked with such a goal.

## 2.3 BDI Agents

Beyond these basic types of agent architectures, a particular type of rational agent, the Belief-Desire-Intention (BDI) agent, has received a great deal of attention. BDI model has a strong philosophical underlying theory towards simulating human behavior that derives from the respectively conceived theory of (Bratman, 1987).

The three mental attitudes of Belief, Desire and Intention represent, respectively, the informative, motivational and deliberative components of a system (Rao and Georgeff, 1995). Although research community has questioned the necessity of these three mental attitudes, it is widely accepted that they are determinative for the system's behaviour. Hence, a rapidly updated belief base is vital for the agent to be kept, taking into account that both modern systems and environments are non-deterministic. It is also necessary for the system, to keep track of the variety of objectives that is asked to accomplish. Desires, could be viewed as a way to describe these objectives, expressed in an abstract way, however decomposing the latter in a hierarchical manner. In modern systems, environmental changes may occur faster than the execution of the action selection function, or the specific action determined by it. In this framework, the system has either to re-execute the selection function, considering the environmental changes, or execute the chosen course of action (plan) to completion. In any case, the agent should have a way to keep "stored" the current course of action and this could be achieved through the third component, the system's intentions.

One of the key points for the decision making process of the agents, is the concept of commitment to the agents' intentions. Apart from the commitment condition, which the agent has to maintain, the model focuses on the different termination conditions that define the context, under which the agent quits its commitment.

More specifically, the model introduces three different commitment strategies that determine agent's behavior. Hence, depending on the application, the user is allowed to choose between the blind-committed, the single-minded or the open-minded agent (Rao and Georgeff, 1991). For example, we could imagine of three agents having to deliver the same task, each one of them following a different commitment strategy, from the above mentioned: agents are tasked to evacuate a room, but unfortunately, the main door is stuck. The blindly committed agent, would keep on trying to open the stuck door, refusing to drop its intention, insisting on its initial plan. So, this type of agent, maintains its intention, until it believes that it has been achieved. The single-minded agent, would quit trying to open the door, assuming the intention to be unachievable. Thus, a single-minded agent drops its intentions, in case it believes that they could not be achieved. In the same scenario, the open-minded agent would try to find another door to exit the room, changing its goals, dropping its commitments accordingly. Hence, the open-minded agent does not maintain its intentions, if the last are not compatible with its goals.

```
BDI-interpreter
initialize-state();
repeat
  options := option-generator(event-queue);
  selected-options := deliberate(options);
  update-intentions(selected-options);
  execute();
  get-new-external-events();
  drop-successful-attitudes();
  drop-impossible-attitudes();
end repeat
```

*Figure 2.1 The BDI interpretation cycle as it is presented in the original paper from* (Rao and Georgeff, 1995)

One of the success factors of the BDI architecture is that it has led to concrete implementations, i.e. executable systems. The BDI interpreter follows an event-driven cycle

(Fig. 2.1). The selected event from the list, determines the corresponding applicable list of plans, with the latter being refined to take the best one, in terms of efficiency. This plan is being added to the intention structure. The next step is the execution of agent's atomic action, as part of an intention. Any new external events are added in the event queue. After the execution, all the accomplished or unachievable desires and intentions are dropped, so the system is updated and the cycle resumes from the beginning. Although abstract, the architecture recommends a number of representation formats, increasing the practicality of the system such as ground set of literals for beliefs, plans and intention stacks.

The BDI model has been proved as one of the most successful agent models. However, there is much discussion about the limitations of the model among the research community (Georgeff et al., 1998). Referring to some of the model's weak points, one could highlight the inability of the model to distinct goals and events, to support learning agents and the fact that the model avoids to establish any mechanism for goal deliberation. Certain proposals could be found in the literature (Ancona and Mascardi, 2003; Guerra-Hernández et al., 2004), aiming to extend the BDI architecture, incorporating such types of behaviour. Although imperfect, the BDI model is a prominent model of reasoning agents, with great influence among the research community and thus, a significant number of implementations and successful applications.

## 2.4 Programming Agent-Oriented Languages

Huge research efforts have been devoted in formulating languages for programming agents. These efforts are reported in several proposals found in Multi-Agent Systems (MAS) literature, and vary from declarative, to strictly imperative approaches.

AgentSpeak(L) is a typical paradigm of a logic-based programming language, ideal for encoding complex agent-based systems, with a lot of influence over several implementations dealing with BDI agents. Estimated as the core of the present project, the language will be discussed in more details, in Section 2.6.

Beyond AgentSpeak(L), a significant number of languages have been proposed from researchers, referenced also in several surveys that could be found in the literature (Bordini et al., 2006; Mascardi et al., 2005; Wooldridge and Jennings, 1994). In the rest of the section, we would try to provide a high-level overview of the most recent ones, or the ones most commonly used. The most important and distinctive characteristics of these languages are introduced and discussed informally.

The Artificial Autonomous Agents Programming Language or 3APL (Dastani et al., 2003; Hindriks et al., 1999) has a combination of imperative and logic programming features. In 3APL, the knowledge base (beliefs) is represented in terms of a logical language. However, the language incorporates a procedural rather than a declarative notion of goals, also called goals-to-do. 3APL recognizes the fact, that goals need not necessarily be consistent, so there should be established a goal deliberation mechanism. In such a goal-oriented design, three types of goals exist: basic actions, achievement and test goals. Through basic actions, 3APL agents update their mental state, performing changes in their belief base and the environment. Achievement goals describe, in an abstract way, the system state, that the agent is tasked to bring about. A test goal allows the 3APL agent to check the consistency between a certain formula and its belief base.

The set of plans (i.e. practical reasoning rules), is tightly coupled with goals, in the same way that goals are coupled with beliefs, since they enhance monitoring and planning of agent's goals. Practical reasoning rules, follow the very well-known structure of head-context (here it is called guard) -body, providing also a mechanism for plan and goal revision, adding failure handling capabilities. Goal modifications are available through the use of goal variables in rules. Respectively, a rule classification is proposed, where failure, reactive, plan and optimization rules are distinguished and ordered, according to their priority. The above ordering is required from the nature of the system, since it is urgent for the agent first to avoid failure and react and then to achieve its goals.

Focusing on control mechanisms, 3APL implements a policy following the glass-box assumption, concerning agents' decision, about which goals to deal with first and which rules to use. The above means that the specification of selection mechanisms that agents use, is readily available to the programmer. An imperative, set-based language, is proposed to play the role of the meta-language, dealing with the control structure, which in turn, is based on the update-act cycle that is followed in several agent languages. The basic outline of this control structure, is presented in Figure 2.2, where the first two steps of the cycle refer to the application phase and the second two, to the execution one.

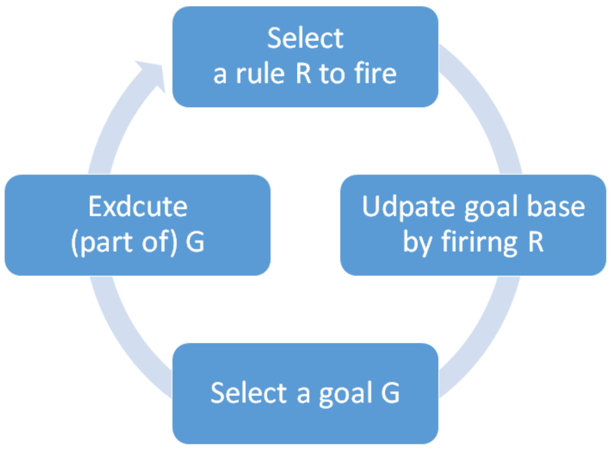

*Figure 2.2 The Update-act cycle of a 3APL agent*

The above concludes our discussion for 3APL, a language close enough to AgentSpeak (L), since both are based on the BDI architecture. The most important difference between the two languages, one could highlight is the extended set of rules of 3APL, that enables goal revision and modification.

Furthermore, several implementations tried to accomplish the task of bridging the gap between logic theory and practice, in agent-programming. GOAL (Goal-Oriented Agent Language, (Hindriks et al., 2000) is a logic-based agent-programming language. In contrast with other agent-programming languages, GOAL uses logical terms to represent goals, as well as beliefs, adopting a declarative perspective. The language introduces the goals-to-be notion, instead of the goals-to-do one. Consequently, GOAL focuses on the description of the state, that the agent desires to reach and not on the actions that are required for this.

Besides goals and beliefs, a set of basic actions is associated with the GOAL agent, defining its capabilities. These actions are to be considered for execution, depending on the mental state condition of the GOAL agent. Through the execution of these actions, GOAL agents are able to drop goals or adopt new ones, noted that it is not required for the new goals to be consistent with the old ones. GOAL agents follow the blind commitment strategy, with respect to which, a goal is being dropped, if and only if it is considered, from the agent, as achieved.

GOAL agents follow a reasoning cycle that consists of two phases. First, they update their knowledge about their environment, which derives from the events, such as percepts and messages. Second, based on this up-to-date knowledge base, GOAL agents have to choose their next action through the decision making process.

The language allows distributed execution of the MAS implemented on multiple machines, communication between agents and is also supported by an IDE offering some debugging tools such as agent introspectors, message sniffers and breakpoints. It is worth mentioning that GOAL was the winner of Multi-agent Programming Contest (MAPC) for the year 2011 (Behrens et al., 2011), represented by the team HACTAR V2 from TU Delft, NL.

JADE (Java Agent DEvelopment Framework, (Bellifemine et al., 1999) is a FIPA (Foundation for Intelligent Physical Agents (O'Brien and Nicol, 1998)) compliant software framework for agent applications. JADE is a distributed application platform that supports lightweight message exchanging, adopting a thread-per-agent concurrency model. Messages are encoded as Java objects. APIs are independent from the underlying network and Java version. A suite of graphical tools is provided, enhancing debugging and monitoring agents' behaviour, at the execution phase.

Jadex (Pokahr et al., 2005), could be viewed as an extension to the JADE agent platform. Hence, a variety of features of JADE, are also incorporated in Jadex, such as the communication infrastructure, agent management services, debugging and development tools. However, Jadex, as a reasoning engine, aims to deal also with the agents' decision making process, focusing on internal agent concepts.

In Jadex architecture, beliefs are represented as Java objects, while plans have to be implemented as Java classes. Jadex defines four different types of goals, such as perform, achieve, query and maintain. Perform goals refer to actions that Jadex agents have to take, without focusing on the overall outcome. Achieve goals describe the general scope that has to be reached, while the query ones are similar to the test goals defined in other languages. Maintain goals, allow Jadex agents to monitor the desired system state. The overall Jadex execution model is strongly based on the BDI interpretation cycle. Apart from the JADE tools already available, Jadex also supports the debugger, for introspecting agents' behavior and the logger, a message sniffer, monitoring the communication between the agents.

SRI Procedural Agent Realization Kit (SPARK, (Morley and Myers, 2004)), is another agent framework, based on the BDI model. The properties of SPARK agent, include the knowledge base, the library of procedures (i.e. plans) and the set of intentions. Similarly to other frameworks, a set of logic clauses is used to model SPARK agent's knowledge base. On the other hand, the plans consist of the name, the event that triggers the plan, the context and the body. The last includes the tasks that should be achieved, represented as finite state machines (FSMs). During the execution cycle, the SPARK agent receives feedback from its sensors, activates the applicable procedures, which in turn can cause changes in its knowledge base, triggering new procedures.

An important feature of SPARK, is agents' guidance. Much research has gone into finding the most efficient way for the agents to prioritize their actions in order to avoid failure and maintain their autonomy, at the same time. According to this attribute, the user is allowed to put some limitations to this autonomy, guiding the agents on how to accomplish their tasks. Two types of guidance are defined, the strategy preference and adjustable autonomy. Regarding the first type, the user is able to implement some restrictions on agent's procedures, concerning for example the sources of information to be retrieved; or to limit the range of solutions, managing specific parameters. According to the second type, the agent is forced to consult or ask for permission from the user, in order to be endorsed to proceed to the next action.

Having clearly defined formal semantics and enhancing practicality, failure handling and scalability, SPARK claims to be sufficient for large-scale real-world applications.

Although a significant number of other approaches have been proposed, we limited our review to the above in order to provide a context for presenting in more detail AgentSpeak(L), which is the main focus of the present project.

## 2.5 AgentSpeak(L)

AgentSpeak(L) (Rao, 1996) is a language, with a remarkable contribution to the ideas in the field, investigated at a time when agent programming was something of a black box and definitely a daunting task for the researchers. This language is with no doubts, the most commonly known approach to a formulated language in agent-programming.

AgentSpeak(L) is built on and claim to be viewed as a simplified textual language of the Procedural Reasoning System (PRS, (Georgeff and Lansky, 1986; Ingrand et al., 1992)), which can be considered as an "ancestor" of the language. As a BDI implemented system, PRS played the role of the starting point for AgentSpeak(L), while the latter could be viewed as an abstraction of the former. PRS, endowed with the attitudes of BDI is a both highly reactive and goal-directed agent architecture, promising to be sufficiently reactive and useful in real-world applications. The PRS interpreter, is responsible for the execution of the entire system and aims to manage the sets of beliefs, knowledge areas (i.e. plans) and the process stack (i.e. intentions), in order to achieve goals. Multiple asynchronous PRS instances could be monitored, as instantiations of the basic system, which run in parallel, used for simulating the communication and the message handling. A fully fledged implementation of PRS, in C++, was the distributed Multi-Agent Reasoning Systems (dMARS, (d'Inverno et al., 1997)).

The mental structure of an AgentSpeak(L) agent, consists mainly of the belief base and the set of plans. Beliefs, are represented using ground literals, similarly to facts in the logic programming sense, formulating belief atoms or their negations. Some examples of beliefs, are showed in Figure 2.3.

```
block(3234,5471).
quantity(wood,23).
~like(john,movies).
book_price(prolog,120).
```

*Figure 2.3 Beliefs in AgentSpeak(L)*

Agents perceive their environment and take actions according to their plans, in order to achieve their goals. A goal is defined as a state of the system, which the agent wants to reach. Bluntly put, a goal puts the agent in an alert state, in order to accomplish a task. Two types of goals are distinguished, achievement (!) and test (?) goals. Concerning the first type, the agent has to reach certain states of affairs, desired situations. As for the second type, the agent has to check its belief base, for compliance with the respecting goal formula.

Plans could be viewed as context-sensitive instruction sets, which are triggered by events. The syntax of a plan follows a head-context-body form, where the head defines the related event, the context, in which the conditions under which the plan is applicable are defined and finally the body, describes the primitive actions or subgoals that have to be achieved. In order for a plan to be applicable, the agent's current beliefs must comply with plan's context. A plan example is shown in Figure 2.4.

```
+!sell_Flowers : flowers(Q) & price(P) & not flowers(0)
<-   broadcast(achieve, propose(Q,P));
     wait(1500);
     !eval_proposals.
```

*Figure 2.4 An achievement plan in AgentSpeak(L)*

The above plan (Figure 2.4) refers to the initial call of an auctioneer, addressed to the bidders, during a flower auction. As for the structure of the plan, the head contains plan's related event, prepended by +! , which means this is an achievement plan. The context part, includes some checks for the sufficient quantity and the price limit, while the body includes the broadcast to the bidders, in order to be informed and prepare their bids, followed by the "wait" action, ensuring a time gap for receiving the bids. Finally, an important part is the sub-goal !evalproposals, which triggers a "lower" level next plan, that would describe the respective actions for proposal evaluating.

Any change in the environment of the agent is denoted in the belief base, as an addition or a deletion of the corresponding belief. In the same way, we could have addition or deletion of goals. Such changes, are called triggering events and in most of the cases, result in invocation of respective plans. Events could be characterized as external, when triggered from an environmental change or another user that asked the system to adopt a goal and as internal, when generated by the agent itself.

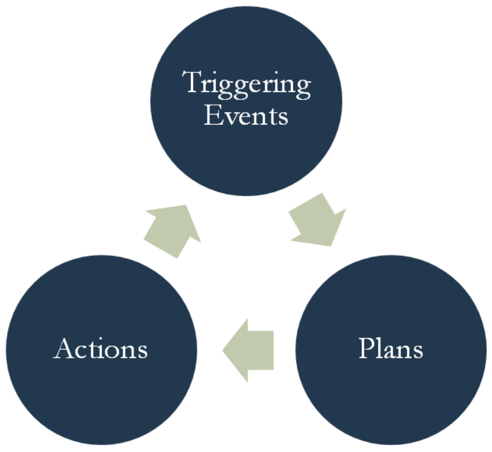

*Figure 2.5 AgentSpeak(L) agent's lifecycle*

From a point of view, in Figure 2.5 is presented the (simplified) life cycle of the AgentSpeak(L) agents: events trigger plans and so, actions are taken, which result in new events completing the cycle.

According to the informal semantics of the language, apart from the sets of beliefs, plans, events, actions and intentions there are also three selection functions SE, SO, SI for events, options (i.e. applicable plans) and intentions respectively. The interpretation cycle is event-driven. Thus, events are added to the respective set, as well as selected from the respective selection function (SE). Consequently, relevant plans are triggered. Provided that the belief base complies with plan's context, a plan becomes applicable and available to be activated from the selection function (SO). Depending on the type of the event, either a new intention is created (in case of an external event), or a new sub-plan is pushed (in case of an internal event) to the intention stack. In the next step, an intention is selected (SI), in order to be executed and this, results in agent's commitment with that intention.

Hence, the interpreter has to execute the formula at the top of intention's body. The above means that, for the agent, there are four different cases, in relation to the type of the goal or action that appears as the first literal at intention's body. In case of an achievement goal, a new internal event is generated and added in the appropriate set, while the goal remains on the stack. If the literal is a test goal, after the potential variable binding, the system seeks to update the stack, as well as the rest of the plan, accordingly. In case of an atomic action, it is directly executed, and dropped from the stack. Finally, if the formula is true, the respective satisfied goal is removed and the substitution is applied to the rest of the body of the intention.

The above described interpretation cycle is followed until all the plans are executed and removed from the intention stack, leaving it empty. That cycle is similar for deletion of goals, as well as for addition and deletion of beliefs. Further analysis of other aspects, such as non-deterministic plans, although referred, are not thoroughly explained.

Concluding our discussion for AgentSpeak(L), we stress once more language's influence to other approaches as mentioned above. An indicative example of that influence and worthy to be further discussed is Jason, claiming to be a fully-fledged interpreter of the language.

## 2.7 Jason

Jason (Bordini and Hübner, 2005) is an AgentSpeak(L) interpreter, written in Java, aiming to extend the language, in terms of practicality. Jason implements a significant number of Prolog features. Among others, one can mention the use of "_" for the anonymous variables, Prolog lists, relational operators and rules. At the same time, other additional features are introduced, such as strong negation (~), as well as belief and plan annotations, with the last to redefine the way that unification is performed.

Annotations provide a simple way to "comment" formulas, enhancing the communication between the agents and useful for programming purposes. An indicative example is given in Figure 2.6, where the belief annotations provide supplementary information, concerning the confidence and the source.

```
offer(book,40)[expires(sunday),confidence(0.2)]
have_good(price(23))[source(agent 23)]
```

*Figure 2.6 Belief annotations in Jason*

On the other hand, plan labels (Figure 2.7) assign symbolic names to plans, useful for debugging and communication purposes, associated with meta-level information. These plan labels, allow the programmer to yield or suspend a plan among alternative ones, interfering in the execution cycle.

```
@ask_to_open_door
+!open_door : true <-
     .send(bob, achieve, open(door));
     .wait(1000);
     .send(bob, askOne, window(Status), Reply);
```

*Figure 2.7 Plan labels in Jason*

Standard and user-defined (written in Java) internal actions (Figure 2.8) for the agents are available with the use of symbol ".", enhancing the controlling mechanisms in a MAS, while selection functions are also customizable in Java.

```
+!list(books) : true <-
     .findall(Book,price_book(Book,_), Books);
     .print(Books).
```

*Figure 2.8* Internal actions in Jason

Another salient characteristic of Jason is failure handling: when an action fails or the interpreter is not able to find an applicable plan to execute, the whole failed plan triggering event (+!g) is replaced with an internal event (−!g), that is generated in the same intention. If there is not a provision (an alternative plan to be triggered) for that failure, the system discards the intention, throwing a warning message, otherwise the failure is handled, according to the plan. The last promotes an open-minded commitment strategy.

The already known from the AgentSpeak(L) execution cycle selection functions (SE, SO, SI), along with Belief Update Function (BUF), Belief Revision Function (BRF), as well as message handling functions, manage Jason’s execution cycle. BUF deals with environmental percepts, updates the Jason agent’s belief base and generates the corresponding external events, but is unable to guarantee belief consistency, which is up to the programmer. BRF is responsible to catch the feedback from the environment, after the execution of Jason agent’s plans.

The compelling reason of BRF existence (as an extension to AgentSpeak(L)’s execution cycle), has to do with the communication needs of the Jason agents. The last are able to send and receive messages in a MAS, adopting a social behaviour. The communication concept in MAS, is heavily influenced by speech act theory (Austin et al., 1975; Searle, 1980). This theory refers to the effect of speech actions that indicate speaker’s intention, with respect to their illocutionary force.

The structure of the messages that are exchanged between Jason agents, follows the schema of Figure 2.9. The first argument provides information about the sender of the message, while the second one (illocutionary force), contains Knowledge Query and Manipulation Language (KQML) performatives (tell, achieve etc.). KQML (Finin et al., 1994), is a language that provides a communication protocol, defining a specific set of performatives that should be used from the agents. In this framework, KQML ensures that Jason agents share a common syntax and semantics, concerning their communication. Finally, the third component of a message in Jason, includes the content, which could be a Prolog literal, a triggering event, a plan or a list.

```
[sender, illocutionary_force, content]
.send(bob,achieve, close(window));
.send(alice,tell, nice(weather));
```

*Figure 2.9 Message structure and message examples in Jason*

Hence, Jason agents are able to send and receive messages, seeking or providing information and tasking each other. However, apart from the above mentioned, Jason agents are able to exchange plans, which means to consult each other, on how to accomplish a task.

The Jason user is also allowed to create environments, directly in Java classes, that extend the default environment class, of Jason. The user is also endorsed to customize the selection functions of Jason agents, managing their behavior. The language is also supported by an IDE, under *jedit*, which enhances MAS programming and facilitates the debugging task, by providing the respective debugging tools (mind-inspector) for agent introspection.

## 2.8 Summary

Among a variety of agent-programming languages that have been implemented through the last decades, AgentSpeak(L) is considered as the distinguished representative in the field and the most common in use. Jason, is a JAVA-based AgentSpeak(L) interpreter, which has gained much of attention lately, among both the research and the academic community. The continuous research and the wide variety of implementations in the field, show the significant effort that it takes, to design and implement a robust and efficient platform, concerning agent oriented programming. The next chapter intends to address this challenge, presenting the Prolog meta-interpreter, implemented in this project.

# CHAPTER 3

# Syntax and Use of AgentSpeak(L)

This chapter presents the technological background of the project, as well as a thorough analysis of the identified features of AgentSpeak(L), along with the respective syntax. Since AgentSpeak(L) is considered the most commonly known approach to a formulated language in agent-programming, it has influenced many projects in the domain. Concerning the syntax, the current interpreter stays close to the original one defined from Rao (1996), offering to the programmer the ability, to use code, written in AgentSpeak(L). As for the use, the presented interpreter implements an extended set of the features of the language, highlighting the ones of strong negation, belief and plan annotations and complicated mental rules, which are briefly explained in the chapter.

## 3.1 A Prolog based meta-interpreter

AgentSpeak(L), as described in the previous chapter, is an agent-programming language with a strong theoretical background, based on the BDI model. This background has a close relationship with logic programming, and AgentSpeak is considered as an extension of logic programming.

On the other hand, logic programming languages as Prolog, have been proven ideal to support programming of meta-interpreters. The fact that Prolog treats data the same way as program clauses, facilitates the development of meta-interpreters. The declarative nature of a logic programming language that focuses on the definition of the problem and not on the explicitly described solution steps, as it happens in the imperative programming could be viewed as the key point, which makes such a language useful for agent-oriented programming. As cognitive agents, have the ability to act, move, reason, with a high level of flexibility, the declarative approach seems ideal for their specification.

Thus, building on a strong background, such as first-order logic, Prolog can be considered as a robust tool, tackling the challenge of filling the gap between the logical theory and the practical issues concerned with software agents' programming.

Agents, require a precisely defined framework, inside which, beliefs, goals and real-world's assumptions would be clearly stated. Hence, a logic programming language, as Prolog, is a powerful conceptual tool, providing the ability to express the complex real-world environment and also to represent both agents' beliefs and desires, in a way similar to the human way of thinking.

## 3.2 Technological Background

SWI-Prolog has been chosen as the language platform of choice, since it is a very actively developed with wide adoption in the research community. SWI provides a variety of development tools, from which, the Eclipse Plugin, a Prolog Development Tool (ProDT), has been chosen, aiming to support the coding, as well as the debugging of this demanding project. Considered as key features of SWI, connectivity, scalability and multi-threading support, could also be thought as the strong pillars of the project. SWI also supports DOCKER, a useful platform for running distributed applications. On the other hand, Pedro (Robinson, 2010), a platform that establishes the connection between the agents, following the server-client model, gives agents the ability to communicate with each other, inside the Multi-Agent System (MAS), respecting the FIPA standards.

The present interpreter, allows the programer to use all the basic features, of the original AgentSpeak(L). Besides these basic features, our Prolog interpreter, includes all the features of modern logic programming system implementations.

| | |
|---|---|
| SWI-Prolog<br><br>http://www.swi-prolog.org/ | |
| Docker<br><br>https://www.docker.com/ | docker |

| | |
|---|---|
| Eclipse IDE – Prolog Development Tool<br>https://www.eclipse.org/eclipseide/<br><br>https://marketplace.eclipse.org/content/prolog-development-tools-prodt | |
| FIPA<br><br>http://www.fipa.org/ | |
| Pedro<br>http://staff.itee.uq.edu.au/pjr/HomePages/PedroHome.html | *No Logo* |

*Table 3.1 The technological background of the current project*

## 3.3 The AgentSpeak(L) Syntax

Concerning the syntax, the present interpreter stays close to that defined from Rao in the original paper (1996) and enriched from Winikoff in his Meta-Interpreter (2006).

### 3.3.1 Beliefs

Regarding beliefs, they are handled the same way as facts in logic programming. Apart from the simple beliefs, that follow the standard form, for example `light(green)`, the interpreter also supports a form of strong negation, expressed by facts prefixed with the operator tilde (~). This powerful feature comes to highlight the difference between a completely absent (from the code) belief and a strong negated one, explained further in the next Section. Following the extensions from Jason (Bordini et al., 2005), belief annotations, that can be used to add information for an existing belief, indicate the source of the information and other user defined attributes the beliefs may have. The infix operator that is used for annotations, is the hash (#). Some examples that show the belief syntax, are presented below.

```
book(prolog).
~book(erlang).
object(front)#[source(agent1)].
object(front)#[source(agent2)].
car(volvo)#[source(ag1),color(white)].
(~like(cookies))#[source(nick)].
```

In the last example, we could notice the parenthesis, which is put to separate the belief from the annotation, in case that a belief is both strong negated and annotated. The parenthesis is necessary to distinguish between two cases:

```
(~like(cookies))#[source(nick)].
~like(cookies)#[source(nick)].
```

In the first case, the use of the parenthesis means that the agent does not like cookies and the respective belief derives from nick. In the latter case, the negation refers to the whole expression. That means, the agent's belief is a negated expression of the above form, while the source, which is omitted, is the agent itself `(source(self))`.

### 3.3.2 Mental Rules

Mental Rules follow the same syntax as Rules in logic programming with the difference being the use of ampersand '&', instead of comma ',' that separates the relations, which form the rule. It is worth highlight that the programmer may use all the predicates supported in Prolog (more significantly in SWI-Prolog, concerning that project), as well as universal facts (beliefs) and recursive definitions, to formulate strong declarative rules, as in logic programming. An example of a mental rule could be the following:

```
findMaxTemp(Temp):-
     temp(Temp) &
     not ( (temp(Y) & Y > Temp) ).
```

Similarly to Rules used in logic programming that introduce a relationship and could be also viewed as means of expressing new or complex queries, Mental Rules in agent-oriented programming could be viewed as means of extending agents' deliberative mechanism. Agents, consult their mental rules, to test conditions, that may appear in the context of a plan or to execute test goals.

Apart from the above mentioned example of a simple mental rule, there is also the option of using more complicated Prolog rules enhancing the programming procedure. This type of rules, available to the programmer as an extra mean, adds robustness to the code and could be viewed as a very powerful feature, especially for the programmers familiar with logic programming. The below given rule is used to gather different solutions, creating the respective list, using the predicate `findall/3`, a well-known predicate in Prolog. After gathering, it also orders the list, with the use of another Prolog predicate, sort/2.

```
findAllTemp(List):-
     findall(X,temp(X),List1),
     sort(List1,List).
```

### 3.3.3 Goals

Agents act and behave, with respect to their beliefs, rules and plans, aiming to accomplish their intended task. That task is represented in terms of agents' goals. There are two basic types of goals, as defined in AgentSpeak(L), according to the agent's objective.

*Achievement goals (`!g`)* - They refer to actions that the agent should take, in order to achieve a certain state that is described. It is the most common type of goals. For example:

```
!close_valve
!shut_window
!turn_left
```

*Test goals (`?g`)* - They ask from the agent to check if the referenced formula (`g`), is true or not, i.e. represent informational tasks that the agent should carry out. For example:

```
?has_leak
?sufficient_funds
?cheapest_book
```

*Goals of the form* `^^g` - Extending the below referenced two basic types of goals, there is another type that also may facilitate the programming procedure. In some cases, there is a need to pursue the goal as a separate intention, rather than following the sequential order of achieving goals. This could be useful, in a large goal sequence, for the goals that are expected to be more time-consuming, in order to avoid delays in the sequence, or when goals should be achieved independently. An example could be the following:

```
^^book_table;
!start_car;
!pick_up_friends;
!go_for_dinner.
```

### 3.3.4 Events

Agents observe their environment, noticing any changes. These environmental changes are represented as events and trigger agents to take the appropriate actions, according to their plans. The function of this mechanism, will be explained in details, in the next Section. The types of triggering events of AgentSpeak are:

```
+b(t)     Belief Addition
-b(t)     Belief Delition
+!g(t)    Achievement Goal
-!g(t)    Achievement Goal (Failure Handling)
+?g(t)    Test Goal
-?g(t)    Test Goal (Failure Handling)
```

### 3.3.5 Plans

Plans as defined in AgentSpeak(L), consist of three basic components, the Head, the Context and the Body. The Head of the plan, consists of the triggering event. Depending on the type of the event that triggers it, the plan is categorized accordingly. Thus, we could define achievement plans (`+!E`), test plans (`+?E`), but also plans triggered from a belief addition or a belief deletion (`+E/-E`). Another category of plans, is used for failure handling (`-!E/-?E`). The above mentioned plan categories will be explained further, in the next Section. Different types of plans are presented through examples, as below:

```
+!speak :: true <- actions.
+?move :: true <- action1;action2;action3.
-book(Prolog) :: true <- actions.
+book(Erlang) :: true <- actions.
-!walk :: true <- actions.
-?check :: true <- actions.
```

Regarding the syntax of the plan, there is a slight difference between the original AgentSpeak(L) syntax and our approach. The operator which is used to separate the head from the context of the plan is '::' instead of ':'. On the contrary, the symbols of ';' and '<-', used for sequencing and implication correspondingly, are implemented exactly the same way as in Rao's (1996) approach. This change was due to the fact that modern Prolog Systems reserve the ":" operator for modules.

## 3.4 Using AgentSpeak(L)

A typical AgentSpeak(L) program, consists of beliefs, mental rules and plans. These are the basic components that combined accordingly, determine the agent's behaviour. Thus, it is of high importance to highlight the different behaviour that the agent adopts, depending on the component of the program used.

### 3.4.1 Beliefs

The agent uses its belief base to store all the available information for the environment, expressed in facts. For example, the fact `weather(nice)` is used to inform the agent about the weather conditions, while `object(front)` is used to warn the agent that an object is in front of it. These facts are formulated, as a result of the feedback from agent's sensors.

However, it would be wrong to say that this is the only "type" of beliefs. Unlike the majority of beliefs that derive from the agent's environment, there is another category of beliefs that has to do with the agent itself. This category involves "internal" beliefs of the agent and in most of the cases has no relation with its environment. Such beliefs, can be used to distinguish one (type of) agent from another, giving the programmer the opportunity to create different tribes of agents. Thus, one could include beliefs, like the following, that "characterize" the agents:

```
like(cookies)
size(small)
gas(petrol)
```

Agent's belief base could be considered as a robust tool in programmer's hands, as it is used as a huge storage (agent's memory), as well as the field in which are stored the features that structure the agent's profile.

### 3.4.2 Strong Negation

Consider chess pieces, controlled by agents. With respect to chess rules, the bishop is able to move only in a diagonal direction, but the rook has to make only horizontal or vertical moves. So, a reliable practice for the programmer of the above MAS, could be to include a belief for the agent bishop, of the form `move(diagonally)`. On the contrary, bishop's belief base should include a belief like `~move(vertically)`. Thus, agents' beliefs should not only show what the agent is able to do, but also what is not able to.

This additional attribute, that is used to bring beliefs more closely to the human way of thinking, is called strong negation. Another example could be the belief, `~like(cakes)` which means that the respective agent does not like (dislike) cakes. The use of strong negation in beliefs, has completely different impact, from the (simple) negation (using `not`). For the previous example, in case there is no belief in the base referring to cakes, this does not mean that the respective agent does not like cakes. In fact, in that case there is no way to decide for the agent's preferences, concerning cakes, just because the program does not include any clue for this. Thus, expressions of the form `not(like(cakes))`, that could be found inside the context of a plan, succeed when the respective belief (`like(cakes)`), is simply absent from the base. On the other hand, when the context includes an expression like `~like(cakes)`, the respective strong negated belief should be included in the belief base.

### 3.4.3 Belief Annotations

One could say that every kind of information could be expressed, using facts, just like in logic programming. However, in agent-programming we have to go one step ahead. Consider that a simple belief, may come from different sources: in many cases, the source of the information is of great importance, as it may increase or decrease the level of trust on the information. So, there must be a way to separate beliefs from different sources, an attribute that is not reflected in simple facts. Annotated beliefs, is an efficient way to handle such belief attributes. Attributes like source, confidence or any other characteristics of beliefs, that may be useful in their handling, should be included in a list, annotating the beliefs. These attributes, could play a vital role, while processing belief's content. Some examples of annotated beliefs, are given below.

```
light(green)#[source(agent26)].
weather(cloudy)#[probability(0.8)].
champion(teamA)#[confidence(0.7)].
```

It is worth to notice that, especially for the case in which a belief derives from the same agent (`source(self)`), it is acceptable to write only the belief with no annotation, concerning the source. So, the following beliefs have exactly the same influence in the code:

```
colour(yellow).
colour(yellow)#[source(self)].
```

Hence, beliefs may have several annotations, adding complexity to the unification procedure. With respect to the type of the annotation, beliefs with the same attribute, for example the source, could be unified, provided that the annotations of the one are included in the annotations of the other. The treatment of the annotated beliefs, is thoroughly described, through the following examples.

```
object(front)#[source(self)].
object(front)#[source(agent22)].

object(front)#[source(agent21)].
object(front)#[source(agent21),source(agent22)].
```

Concerning, the first pair of beliefs, although they refer to the same content, they could not be unified, because of the different annotations that they have. However, concerning the second pair of beliefs, the annotation of the first, is included in the list of annotations of the second belief. Thus, the beliefs of the second pair should be unified, as the first is a subset of the second.

### 3.4.4 Plans

As stated, plan categorization is derived from the categorization of their related triggering events. The most commonly used are achievement plans, which are triggered from achievement goals and describe all the actions that are required, for the agent to reach the desired state. An example of an achievement plan is given below:

```
% Bookstore Example
% Beliefs
bookstore_book(prolog,25).
bookstore_book(java,23).
bank_account(100).
% Plans
+!can_buy_book(Book)  ::  bookstore_book(Book,  Price)
<-
?bank_account(Balance) ;
Balance > Price ;
write([‘You have the money to buy ‘, Book]);
!buy_book(Book);
+my_books(Book).
[...]
```

The above example presents the deliberative mechanism that the agent follows to take actions, in the effort to achieve its goals. The agent of the example, desires to buy a book from the bookstore. Thus, the initial goal that the agent has to achieve is the `!can_buy_book(Book)`. Sequentially, the event that is generated is the `+!can_buy_book(Book)`, which triggers the respective plan. Before the execution of the plan, the agent has to check its applicability, which means that the plan's context should be true. In the current example, the plan's context is used to check if a book is available in the bookstore. The available books are represented with beliefs of the type `bookstore_book(Book,Price)`, so what is needed for this plan to be applicable is simply to exist a belief (at the agent's belief base), like the one referenced in plan's context. Provided that the plan is applicable (the book exists), the agent is able to start executing what is written inside plan's body.

The body of a plan could include the following:

- Actions
- Internal Actions
- Achievement Goals
- Test Goals
- Mental Notes
- Expressions

*Actions.* Actions' syntax is similar to ground predicates, used in logic programming. Agents perform actions, causing changes to the environment. In case that an action fails, the whole plan will also fail.

*Internal Actions.* The internal actions refer to actions that do not have impact on the external environment of the agent, but aim to improve the level of functionality for the whole program. As internal actions, could be used all the implemented Prolog predicates (the ones supported in the current version of SWI, that has been chosen for the current project). Examples of such predicates, practical for agent-programming purposes are the following:

```
write/1
max/2, min/2
findall/3, setof/3, bagof/3
```

It is worth mentioning, that as in logic programming, the programmer is allowed to define new predicates, according to the needs of the program. An example could be the predicate `wait/1`, which causes a time delay, often useful at the agent oriented programming.

```
%%% Internal action wait _time in milliseconds
wait(Time):-
     get_time(T1),
     repeat,
     get_time(T2),
     Z is Time/1000,
     T2 > T1 + Z.
```

*Achievement Goals.* Achievement goals inside plan's body, trigger achieve plans accordingly, which are usually called sub-plans. In most cases, achievement goals are included inside plan's body, as a way to divide a large task (parent plan) into other subordinate ones. Such an example is given below:

```
+!make_cake :: true <-
     !heat_oven;
     !add_ingredients;
     !mix;
     !bake.
```

The parent plan (`+!make_cake`) that describes the main task, is divided into sub-tasks, expressed as sub-plans. Also, in the previously referenced bookstore example, the plan `!buy_book(Book)`, could be viewed as a sub-plan of the "parent" plan `+!can_buy_book(Book)`. Such a programming practice, increases code's readability, allowing the whole program to be more easily debugged.

*Test Goals.* Unlike achievement goals that their only use is to trigger achieve plans, test goals could be used in several ways as referenced below:

- Checking the agent's belief base
- Checking an agent's mental rule

- Triggering a test plan

The bookstore example, as presented above is helpful to analyze better the first and the latter case, concerning the use of test goals. As for the second case, in which test goals trigger the execution of mental rules, it will be analyzed in a dedicated part below, called Mental Rules.

Hence, in the bookstore example, the bank account balance is checked through the test goal `?bank_account(Balance)`, which seeks for the respective belief, at the agent's base, returning to the variable Balance, the respective value.

It is worth to notice, that in case of failure of such a test goal, the whole plan also fails. This happens because the only way for such a test goal to succeed, is to unify with the respective belief. Thus, if a belief of the form `bank_account(Balance)` does not exist in the code, the whole plan (`+!can_buy_book(Book)`) will fail.

However, a robust program should provision against such kind of failure. Thus, an efficient practice is to avoid including test goals, that have a possibility to fail, inside plan's body. Instead of the previous syntax, it is more preferable to follow the one presented below:

```
+!can_buy_book(Book)  ::  bookstore_book(Book,  Price)
<-
     ?check_bank_ac(Balance,Price);
     Balance > Price;
     write([‘You have the money to buy ’, Book]);
     !buy_book(Book);
     +my_books(Book).
+?check_bank_ac(Bal,Pr) :: true
<-
     ?bank_account(Bal).
```

Following the above syntax, the test goal `?check_bank_ac(Balance,Price)` triggers the execution of another plan (`+?check_bank_ac(Bal,Pr)`), which is called test plan. During the execution of this plan, the belief `bank_account(Bal)` is checked and the appropriate value is returned to the respective variable (`Bal`). The mechanism that is followed is exactly the same with the achieve sub-plans that described before.

Although the execution cycle is better arranged and everything seems to be under control, the possibility to face the same failure, in case that the test goal `?bank_account(Bal)` fail, should still be considered. A possible failure of the respective test goal would cause a failure to the test plan, driving the parent plan to fail.

The most highly effective mean, that would efficiently decrease the possibilities of failure for the program, is the use of alternative plans. Once the execution mechanism follows the top down approach, every time a plan fails, another plan should be applicable, in order to handle such a failure. This mechanism will be detaily explained, in the respective chapter titled Failure Handling.

As a general rule, it is preferable to avoid failure inside plan's body, moving any test conditions in the context of a plan. Once the conditions inside the context fail, the whole plan would be considered as inapplicable and another applicable plan would be searched. Including (at the end of the procedure) a plan with no context (always true), could be proved an efficient practice to avoid failure. The version given below, of the previously presented code, is an example that implements such a practice.

```
+!can_buy_book(Book)  ::  bookstore_book(Book,  Price)
<-
     ?check_bank_ac(Balance,Price);
     write([‘You have the money to buy ‘, Book]);
     !buy_book(Book);
     +my_books(Book).
+!can_buy_book(Book) :: true <-
     write([‘The book ‘ ,Book, ‘ does not exist.’]).
+?check_bank_ac(Bal,Pr) :: true <-
     ?bank_account(Bal).
+?check_bank_ac(Bal,Pr) :: true <-
     alternative_actions[…]
```

In the above program, there is provision against failure in two cases. First, in case that the belief `bookstore_book(Book, Price)` fails (book does not exist). Second, in case that the test goal `?bank_account(Bal)` fails (the belief does not exist).

Of course, the effort to decrease the possibilities of failure is endless, because always there should be a case for the whole program to fail. But, it is worth to take the appropriate measures, keeping the code robust and safe.

Hence, although test goals has been proved a handy tool, which could be used in several ways, it is worthy of great attention, as it could make the code extremely fragile.

*Mental Notes.* Bluntly put, agents' belief base is a huge information storage, that plays the role of agent's memory, that is able to change during the execution of the program. Similarly to humans, agents need a way to keep track of environmental or internal changes. Through mental notes, agents keep notice of such changes, by adding or removing information from their base. Mental notes are included in plan's body and add (+belief#[annotation]) or remove (-belief#[annotation]) beliefs. It is considered as a useful practice to keep mental notes, enhancing both the processing and the debugging procedure, but maintaining also a high degree of consistency for the whole MAS. In the example given above (Figure XX), the +!can_buy_book(Book) plan, after the necessary actions and sub plans that includes, for the "buy" procedure, it finally includes the +my_books(Book) mental node. This node plays a vital role, as it allows the agent to keep track of the transaction, that just took place. After adding such a belief, the agent keeps the evidence, which might be proved very useful for it in the future.

Such mental notes, could also trigger relevant plans. Sometimes, it could be very useful for the agent to take appropriate actions, depending on certain changes of the environment or changes regarding its beliefs. The above mechanism means, that an event of the form +/-b, could trigger a plan of the form +/-b :: true -> actions. The examples below, show two cases, where a belief's addition or deletion, trigger relevant plans:

```
+car(broken) :: true <-
     !go_on_foot.
-weather(rainy) :: true <-
     !go_for_a_ride.
```

*Expressions.* Sometimes it is useful to apply mathematical calculations inside plan's body or compare values, as part of the calculating process. Expressions like the one included in the plan below are commonly in use, not only in the body of the plan but also in the context.

```
% Beliefs
book(anne_frank, 19).
book(harry_potter, 25).
book(hamlet, 28).
money(30).
% Plans
+!buy_book(Book) :: book(Book,Price) <-
     ?money(Amount) ;
     Amount >= Price ;
     write([‘You can afford buying’ , Book]).
```

It should be highlighted that once the expression fails, the whole plan would fail, too. Thus, it is for the programmer to avoid that failure, including an alternative plan, similarly to the cases described earlier.

### 3.4.5 Plan Annotations

Annotations do not only refer to beliefs, but also to plans. Considering the basic structure of the plan, with head, context and body, the head refers to the (internal or external) change, called event that triggers the plan. Annotations could be also used, to define precisely this triggering event. For example, the plans below, would be triggered under different conditions.

```
+!g#[confidence(A)] :: true <- actionsA.
+!g#[confidence(B)] :: true <- actionsB.

+b#[source(A)] :: true <- actionsA.
+b#[source(B)] :: true <- actionsB.
```

The first pair of achievement plans, describes the alternative courses of action, that would be followed, according to the case, while the second pair of plans, separates the actions that should take place, with respect to the source of the added belief. The use of annotated plans, could be proved convenient for the programmer, defining exactly the circumstances, under which the one or the other plan would be triggered. The use of plan annotations, increases code modularity and flexibility, but also allows a large program to be more easily debugged.

### 3.4.6 Mental Rules

Apart from simple beliefs, cognitive agents can employ rules, to derive conclusions. In a way similar to Prolog, Mental Rules are used to express relationships and allow agents to combine simple beliefs to draw more complex ones. As mentioned, mental rules are useful in test conditions, as part of plans' context, but also in test goal "execution". The code below, illustrates the above mechanism, presenting two examples of mental rules, employed in different cases.

In the first case, the mental rule is used in plan's context (`over_salary_limit(S)`), checking if the given salary satisfies the limit, which is set in the belief base (`limit(#)`). In the latter case, the best job is chosen (from the ones given), through the execution of a test goal (`?highest_salary(S,P)`), triggering the respective mental rule.

```
%% Salaries’ Example
%% Beliefs %%
limit(10000).
salary(8000,cashier).
salary(12000,employee).
salary(18500,doctor).
salary(25000,broker).
%% Mental Rules %%
over_salary_limit(S) :-
     limit(L) & L=<S.
highest_salary(Salary,Description) :-
     salary(Salary,Description) &
     not ( (salary(S2,_) & S2 > Salary) ) .
%% Plans %%
% The Mental rule is inside plan's context %
+!is_good_job(S,P) :: over_salary_limit(S) <-
     write(['Job ', P , 'provides a good salary,
     offering ', S , 'per year.']),nl.
```

```
+!is_good_job(S,P) :: true <-
     write('Not accepted job. Too low wages.'),nl.
% The Mental rule's getting triggered from a test goal
+!find_best_job :: true <-
     ?highest_salary(S,P);
     write(['Job ', P , 'provides the best salary,
     offering ', S , 'per year.']),nl.
```

Noticed that in the second case that the mental rule searches for the highest salary, there is always a case, in which the rule succeeds (provided that there is at least one given belief that refers to a salary and a job). However, in the first case, there is always the possibility for the rule to fail, so it is a good practice for the programmer to include an alternative plan (`+!g2`) as referenced before, to avoid failure.

## 3.5 Summary

The chapter starts be describing the technological background of this project. The proposed meta-interpreter, is written in SWI-Prolog, which has been chosen as the most actively developed version of Prolog language. Also the whole project, is based on a robust technological background, capitalizing on modern tools such as Docker®.

After a thorough description of the original syntax of AgentSpeak(L), it is provided a detailed analysis of the implemented features, concerning the use of the proposed meta-interpreter. The features of strong negation, belief and plan annotations and the use of complicated mental rules, combined with the basic features of the language, compose a robust framework that extends software agents' capabilities.

# CHAPTER 4

# Implementation

In this chapter we are going to present the implementation of the meta-interpreter. As it is already mentioned earlier, AgentSpeak(L)'s syntax stays close to the syntax that is used in modern logic programming languages as SWI-Prolog. This logic-familiar syntax, enhances the implementation of the parser. Thus, the parser mainly focuses on the definition of the specific operators that are used in AgentSpeak(L). As for the use of the meta-interpreter, it stays close to the one defined from Rao in the original paper ((Rao, 1996)). The basic features of AgentSpeak(L) are enriched with new ones, with the aim to increase the efficiency of the language. Examples of these new features are the implementation of a sophisticated mechanism for belief handling, the support of new types of goals (goals pursued as a separate intention) and also the use of higher order Prolog predicates.

Section 4.4 refers to another interesting feature in agent programming, which is failure handling. The meta-interpreter's implemented failure handling mechanism, aims to support the execution process, allowing the program to run smoothly. Finally, the design of the system is described in the last Section. The multithreaded model is followed for the design of the current project, as it enhances both parallel execution and communication between agents.

## 4.1 The Parser

In our implementation, the user program follows closely the syntax of AgentSpeak(L), with some minor differences regarding operators, as mentioned in the previous chapter. The parser loads the program, by creating a set of Prolog facts for every program statement it parses, using a failure driven loop. During parsing, we obtain the initial definition of the agent, expressed in beliefs, mental rules and plans.

Aiming to achieve the highest level of compatibility with the original AgentSpeak(L) and at the same time exploit the parsing capabilities of Prolog, we defined a set of operators, extending those already defined in Prolog (Appendix 1).

Thus, operators part of the AgentSpeak(L) syntax, should be defined in Prolog, respecting the precedence and associativity in order for the whole program to be parsed correctly, but also to maintain its readability. The new operators are defined, using the predicate `op/3`, with the following syntax:

```
op(Precedence, Type, Name)
```

The variables that are used in the predicate refer to the precedence (values between 0-1200), the type (postfix, infix etc.) and the name of the operator. These operators are presented below (Table 4.1).

| **Precedence** | **Type** | **Name** | **Functionality** |
|---|---|---|---|
| `1190` | `xfx` | `<-` | as in AgentSpeak(L) |
| `1150` | `xfx` | `::` | as '`:`' in AgentSpeak(L) |
| `1050` | `xfy` | `&` | Conjunction Operator |
| `900` | `fy` | `not,~` | Negation, Strong Negation |
| `200` | `fx` | `!,?,+?,-?` | as in AgentSpeak(L) |
| `200` | `fx` | `^^` | as '`!g`' but with '`^^g`' the goal is pursued as a separate intention |
| `190` | `xfx` | `#` | Annotations |

*Table 4.1 Operators defined in the Prolog meta-interpreter for AgentSpeak(L)*

Noticed also, that the '`;`' operator, already defined in Prolog, is used for sequencing actions inside plan's body. As disjunctive operator, the '`|`' predefined in Prolog was used.

## 4.2 Internal Representation

As mentioned, the user program is represented internally as Prolog facts. In the sections that follow we describe the internal representation of each AgentSpeak construct.

### 4.2.1 Beliefs

Beliefs are parsed, to facts of the form: `belief(Belief,Annotations)`. A user statement *`object(front)#[source(self)]`* is translated by the parser to the fact: *`belief(object(front), [source(self)])`*.

Notice that belief annotations should always appear as a list. In the case that the belief originates from the agent itself (`source(self)`), the annotation could be omitted. Each belief addition given by different sources, alters the belief annotations accordingly. For example, consider the following program:

```
    object(front).
    object(front)#[source(agent1)].
    object(front)#[source(agent2)].
```

The parser should create a fact like the following:

```
    belief(object(front),[source(self),source(agent1),source(
agent2)]).
```

### 4.2.2 Mental Rules

Similar to beliefs, mental rules are parsed creating a fact of the form *`rule(Belief,Body)`*. For example, the following rule is used to find the maximum of given temperatures, expressed as `temp(Temp)` beliefs in the program:.

```
    findMaxTemp(Temp):-
```

```
        temp(Temp) &
        not ( (temp(Y) & Y > Temp) ).
```

For the above rule, the parser would create the fact given below:

```
    rule(findMaxTemp(VAR_Temp),    temp(VAR_Temp)   &   not   (
(temp(VAR_Temp_2) & VAR_Temp_2 > VAR_Temp) ).
```

### 4.2.3 Plans

As already mentioned, plans play the role of the recipes for the agent and describe the options that it has, according to the events that may occur.

The basic syntax for a plan is the one that includes the event (`E`), the guard (context) and the plan's body. Hence, every plan is parsed to a fact following the form `plan(Event,Context,PlanBody)`. An example which shows the plan parsing, is the following:

```
    +!speak :: weather(nice) <-
        !say_hello;
        !go_for_a_ride.
```

The above plan written in AgentSpeak(L) should be parsed, creating a Prolog fact, like the one below:

```
    plan(+!speak,weather(nice),!say_hello;!go_for_a_ride).
```

Apart from the "usual" syntax of a plan, the programmer is allowed to write "incomplete" plans, lacking one or more components of the three basic ones (Event, Context, PlanBody), as mentioned above. For example, in many cases it is useful for the programmer to omit the context, writing the following:

```
+!speak <-
     !say_hello.
```

The above is equivalent to a plan that has *true* in its context, but more simple to be written. The parser would still create the following, without throwing any errors.

```
plan(+!speak,true,!say_hello).
```

In the same framework, acceptable forms of plans are also those which lack a body i.e. which consist only from the event. The last, could only be achievement plans (`+!speak`). Examples of that kind of plans, after parsed, would give the following internal representations:

```
plan(+!speak,context,true).
plan(+!speak,true,true).
```

## 4.3 The Interpreter

After parsing and loading the program, the interpreter is responsible to proceed with its execution. Once the belief and the plan base have been structured, the next step is to define the goals of the agent. These goals originate either from the agent itself (according to the code written by the programmer), or from another agent that asks for them. Additionally, the agent adopts goals, triggered by changes on its beliefs, while the last derive either from environmental percept or through communication with agents. Hence, it is critical to distinguish between these cases, where the agents adopt new goals:

- Goals that originate from the agent itself
- Goals asked by other agents to be achieved
- Changes in beliefs as a result of environmental percept
- Changes in beliefs as a result of communication with other agents

The mean for reaching (achieving) such goals, is the agent’s plans, which include the necessary actions, sub-plans or expressions, as mentioned earlier. The execution of such plans, reflects the agent’s intentions.

The interpreter receives the above described events as input reacts based on the agent’s plans. The main part of the interpreter (*solver*), has been based on the meta-interpreter for the AgentSpeak(L), proposed by Michael Winikoff (2006) and is presented below:

```
solve(true).
solve(fail):- !,fail.
% Sequencing actions in plan body
solve(P1;P2):-
  !,nb_setval(remaining,P2),
  solve(P1), !, solve(P2).
% Solving an achieve goal
solve(!E):-
  !,
  find_plan(+!E,_,P),
  %% commit to an applicable achieve plan
  !,
  solve(P).
% Solving a test goal
solve(?E):-
     test_condition(E).
% Solving a test goal (with test plan)
solve(?E):-
  !,
  find_plan(+?E,_,P),
  %% commit to an applicable test plan
  !,
  solve(P).
% Solving a goal pursued as a separate intention
solve(^^E):-
     generate_agentspeak_event(achieve(!E)).
```

```
%%%%%%%  Adding, Removing & Updating Beliefs %%%%%%%%
solve(+B#A):-
     !,add_belief(B,A).
solve(+B):-
     !,add_belief(B,[source(self)]).
solve(-B#A):-
     !,remove_belief(B,A).
solve(-B):-
     !,remove_belief(B,[source(self)]).
%%%% Plans Triggered from belief addition/deletion %%%%
solve(addb(B,A)):-
     find_plan(+B#A,_,P),
     % commit to an applicable triggered plan
     !,solve(P).
%% In case there is no plan for belief addition.
solve(addb(_,_)):-!.
solve(delb(B,A)):-
     find_plan(-B#A,_,P),
     % commit to an applicable triggered plan
     !,solve(P).
%% In case there is no plan for belief deletion.
solve(delb(_,_)):-!.
solve(E):- E.
```

The solver above, is responsible to handle the input, unifying the *events* with the existing *plans* in the agent's plan library, as below:

- Solve an achievement goal (`!g`)
- Solve a test goal (`?g`)
- Solve a goal pursued as a separate intention
- Solve a belief addition/deletion
- Solve a plan triggered by a belief addition/deletion

- Solve an action/expression/Prolog predicate

The cases listed above, cover the ones defined in AgentSpeak(L). Hence, the interpreter is capable of reading and executing a program written in AgentSpeak(L), with respect to the minor differences regarding the syntax, as already noticed earlier. Ensuring that the level of compatibility with AgentSpeak(L), as well as the smooth functionality were checked, through a wide set of test cases, it is important to present the most representative ones, along with the respective console output, at the end of every part, as following.

### 4.3.1 Achievement and Test goals

Concerning the achievement and test goals, the solver focuses on the unification of the event (`!g, ?g`) with an applicable corresponding plan (`+!g, +?g`). Once this applicable plan exists, the predicate `find_plan`, which is explained later on, succeeds. Sequentially, what is included inside plan's body, as content, is executed (`solve(P)`). For example:

```
%% Test Case 1
%% Beliefs
time(morning).
day(friday).
%% Plans
+!test1 :: true <-
     !message(friday);
     !testAnotPlan#[source(agent13)];
     ?is_limit(100)#[source(agent21)].
+!message(D) :: time(morning) & day(D) <-
     write('test 3 -ok. (Checking  Plan Conditions)');nl.
+!testAnotPlan#[source(agent13)] :: true <-
     write("test  24  -ok.  (Annotated  Plan  asked  to  be
     achieved)");nl.
+?is_limit(L)#[source(agent21)] :: L>10 <-
     write("test 32 -ok. (Solving a test goal, by executing
     an annotated test plan) "),nl.
```

Console output:

```
?- load_agent('tester.pl').
true.
?- test 3 -ok. (Checking  Plan Conditions)
test 24 -ok. (Annotated Plan asked to be achieved)
test 32 -ok. (Solving a test goal, by executing an annotated
test plan)
```

However, as it is mentioned in the previous Chapter, test goals are not only used to trigger test plans, but also to check if a belief exists or to execute a mental rule. The part of the code that handles these cases, calls the predicate `test_condition/1`, which will be explained in detail later in the corresponding Section. The example below presents such a case, where a mental rule is executed:

```
%% Test Case 2
%% Beliefs
salary(10000,captain).
salary(15000,major).
salary(18000,colonel).
%% Rules
lowest_salary(Salary,Person) :-
     salary(Salary,Person) &
     not ( (salary(S2,_) & S2 < Salary) ) .
%% Plans
+!evaluate_salary :: true <-
     ?lowest_salary(S,P);
     S=10000;
     write("test 28 ok. (Solving a test goal following a
     mental rule) "),nl.
```

Console output:

```
?- load_agent('tester.pl').
true.
?- test 28 ok. (Solving a test goal following a mental rule)
```

### 4.3.2 Goals pursued as separate intentions

Concerning the case where a goal is pursued as a separate intention, the difference is the use of the predicate `generate_agentspeak_event (achieve(!E))`. Noticed that this case refers only to achievement goals, as indicated by the respective event, that is generated. It is worth to mention, that the predicate above, is also used for the creation of a new intention stack, while its behaviour will be further explained, in the Design Section. The example that follows, shows such a case:

```
%% Test Case 3
%% Beliefs
%% Plans
+!test :: true <-
     ^^separateIntention.
+!separateIntention :: true <-
     write("test  33  ok.  (Goal  pursued  as  a  separate
     intention)"),nl.
```

Console output:

```
?- load_agent('tester.pl').
true.
?- test 33 ok. (Goal pursued as a separate intention)
```

### 4.3.3 Belief Addition/Deletion Goals

The part of the solver that handles the changes in the agent's belief base, refers to the events of +B, -B, +B#A, -B#A, in relation to a belief addition or removal, either it is annotated or not. Depending on the desired goal, the interpreter calls one of the respective predicates, which function is defined as below:

```
%% The Belief already exists with different annotation(s)
add_belief(B,Atts):-
     retract(belief(B,PreviousAtts)),
     !,
     union(Atts,PreviousAtts,NewAtts),
     assert(belief(B,NewAtts)),
     generate_agentspeak_event(addb(B,Atts)).
%% Completely New Belief
add_belief(B,Atts):-
     is_list(Atts),!,
     assert(belief(B,Atts)),
     %% this is to work with higher order predicates.
     assert(B),add_record(B),
     generate_agentspeak_event(addb(B,Atts)).
%%% The belief exists
remove_belief(B,Atts):-
     belief(B,PreviousAtts),
     !,
     subtract(PreviousAtts,Atts,NewAtts),
     retract(belief(B,PreviousAtts)),
     !,
     length(NewAtts,N),
     %% if the belief has no other annotations,
     %% there is no reason to be kept in BB
     (N > 0 -> assert(belief(B,NewAtts)) ; retract(B)),
     generate_agentspeak_event(delb(B,Atts)).
%%% If the belief does not exist, then simply succeed
%%% without triggering any plan
remove_belief(_,_).
```

The behaviour of every one of the predicates above, distinguish between two cases: the added/removed belief already exists in the agent's belief base, or not.

Concerning the behaviour of the predicate `add_belief/2`, considering the case where the added belief exists, it is enough to add the new annotation in the list of the ones that already exist. On the other hand, if the added belief does not exist, a completely new record of such a belief, is added in the belief base. This new record is of the form `belief(B,Atts)`, as defined in parsing.

However, it is worth to mention a special mechanism regarding belief addition. Apart from the record that corresponds to the form `belief(B,A)`, the content of the belief (`B`), is also (double) tracked at the Prolog base (`assert(B)`). The reason for this, is related with the execution of mental rules. Consider that the agent has to check the rule given below:

```
findMaxTemp(Temp):-
    temp(Temp) &
    not ( (temp(Y) & Y > Temp) ).
```

During the execution of such a rule, the interpreter seeks for records of the form `temp()`, while the beliefs of the agent respect the form `belief(B,A)`. So, despite the fact that the agent's base might include, for example, a belief like `belief(temp(43),source(self)`), the rule would not be executed successfully, because of this forms' inconsistency. Thus, by using both the two forms for the belief recording, the above described deficiency is resolved. Mental rules can normally be executed, when written following the Prolog syntax, similarly to the rules in logic programming.

The predicate `remove_belief/2` , assuming the case where the removed belief exists, the has as a first action is to remove the respective annotation from the set of the annotations that corresponds to that belief. However, after this action, it is critical to check if the annotation of the removed belief is the only one present in the respective list. This check is of great importance, because in that case, the belief should be completely removed from the base, since there is no reason for a belief without annotation(s), to be kept. In case that the removed belief does not exist, the predicate simply succeeds. It is worth to mention, that in this case, even if there is a plan to be triggered from such a remove (`-b`), it is not triggered.

The example presented below refers to the cases where a belief is added or removed from the agent's belief base:

```
%% Test Case 4
%% Beliefs
obstacle(back)#[source(agent1)].
obstacle(back)#[source(agent99)].
%% Plans
+!test :: true <-
     +obstacle(back)#[source(agent100)];
     write("test  16  -ok.  (Add  directly  an  annotated
     belief)");nl;
     -obstacle(back)#[source(agent99)];
     write("test  19  -ok.  (Remove  directly  an  annotated
     belief)");nl.
```

Console output:

```
load_agent('tester.pl').
true.
?- test 16 -ok. (Add directly an annotated belief)
test 19 -ok. (Remove directly an annotated belief)
```

Noticed also, that after the execution of the code above, the agent's belief base should include the belief `obstacle(back)#[source(agent100), source(agent1)].`

## 4.3.4 Plans triggered by Belief Additions/Deletions

Changes in the belief base could trigger the execution of other plans, written for this purpose. In this case, a special "pseudo-event" is generated, corresponding to the belief addition (`addb(B,A)`) or deletion (`delb(B,A)`), accordingly. A detailed explanation for this kind of "pseudo-events", which are generated using the predicate `generate_agentspeak_event`, will be given in the Design Section.

The solver handles these events in a way similar to the one that follows for the achievement and test goals. It seeks for a plan (`+b`/`-b`) that unifies with the belief added or removed and executes it, provided that it is applicable. For example:

```
%% Test Case 5
%% Beliefs
bad(mood)#[source(agentB)].
%% Plans
+!test :: true <-
     +good(day)#[source(agentA)];
     -bad(mood)#[source(agentB)].
+good(day)#[source(agentA)] :: true <-
     write('test 21 -ok. (Plan triggered by an annotated
     belief addition)');nl.
-bad(mood)#[source(agentB)] :: true <-
     write('test 23 -ok. (Plan triggered by an annotated
     belief deletion)');nl.
```

Console output:

```
load_agent('tester.pl').
true.
?- test 21 -ok. (Plan triggered by an annotated belief
addition)
test 23 -ok. (Plan triggered by an annotated belief
deletion)
```

Noticed that after the execution of the code above, the belief `good(day)` should have been added in, while the belief `bad(mood)` should have been removed from, the agent's belief base.

### 4.3.5 Actions/Expressions/Prolog Predicates

Finally, for the rest of the cases, which refer to the execution of (Prolog) predicates or checking expressions, the solver simply evaluates the predicate or the expression as true or false and proceeds accordingly:

```
?- solve(1>2).
false.
?- solve(3>2).
true.
?- solve(write("Prolog")).
Prolog
true.
```

As it was mentioned above, there are two basic predicates, that play an important role in the interpretation of a program. These are the `find_plan/3` and the `test_condition/1`.

More significantly, the `find_plan/3` predicate, takes three arguments, the head, the context and the body of a plan. Also, there could be distinguished three basic cases, in which the current predicate is used for searching for a type (test, achievement etc.) of plan. The code presented below, refers to test plans (the implementation of the process for the other types of plans is similar) and includes these basic cases:

```
%%%% Plans getting triggered from test goals - TEST PLANS
%% Searching for a test plan where source(self) is omitted
find_plan(+?E#[source(self)],C,P):-
```

```
          plan(+?E,C,P),
          test_condition(C),
          !.
     %% Searching for an annotated triggered test plan
     find_plan(+?E#A,C,P):-
          plan(+?E#Annotations,C,P),
          subset(A,Annotations),
          test_condition(C),
          !.
     %% Searching for a non-annotated triggered test plan
     find_plan(+?E,C,P):-
          plan(+?E,C,P),
          test_condition(C),
          !.
```

As it can be seen in the first case, when the interpreter searches for a self-originated plan, it is allowed to unify with a plan with no annotations, assuming that the source is the agent itself.

The second clause, refers to the case where the annotation of the plan that is searched, is included in the list of annotations of a corresponding plan that exists, in the agent's plan library. This is adequate for such a plan, in order to be considered as applicable.

Finally the last case, is the most straightforward, where the interpreter seeks for a single (non-annotated) plan.

In any of the cases above, after the unification, the predicate `test_condition` is responsible to check plan's applicability. This predicate takes an argument, which refers to the context of the plan that is searched. Once the context is true, the predicate succeeds. Thus, once the predicate `find_plan/3` succeeds, it delivers the body of the plan (`P`) back to the solver, in order for the last to proceed with the execution of the plan.

Concerning the behaviour of the predicate `test_condition`, it is defined in the part of the code presented below:

```
     %%% Testing Conditions
     test_condition(true):-!.
```

```
%%% Conjunction
test_condition(G1 & G2):-
     !,
     test_condition(G1),
     test_condition(G2).
%%% Disjunction
test_condition(G1 | G2):-
     !,
     (test_condition(G1) ; test_condition(G2)).
%% Strong negation on beliefs
test_condition(~G):-
     !,exists_belief(~G).
%%% Negation NOT on anything
test_condition(not G):-
     test_condition(G),!,fail.
test_condition(not _):-
     !,true.
%%% Annotated beliefs
test_condition(G # A):-
     exists_belief(G,A),!.
%%% Non-annotated Beliefs
test_condition(G):-
     exists_belief(G,_).
%%% Handling a mental rule
test_condition(G):-
     rule(G,B),
     test_condition(B).
%% A belief exists in BB.
exists_belief(B):-
     belief(B,_).
%% Belief Annotation on lists of annotations
%% A belief condition holds if it is a subset of beliefs.
exists_belief(B,Att):-
     belief(B,Attributes),
          subset(Att,Attributes).
```

The predicate `test_condition/1`, is used to check the argument that takes, evaluating it as true (succeeds) or false (fails). Although that argument in most cases refers to the plan's context, it could also refer to a belief or a mental rule. This case was defined earlier, when describing the process for solving test goals:

```
% Solving a test goal
solve(?E):-
          test_condition(E).
```

As mentioned in the previous Chapter, the interpreter supports strong negation and annotated beliefs. Thus, the implementation of the predicate `test_condition/1`, in those cases is quite simple. The predicates `exists_belief/1` and `exists_belief/2` are used to check the existence of non-annotated or annotated beliefs.

Hence, in case the interpreter has to check the existence of an annotated belief in the agent's belief base, it simply requires that the annotation should be included in the list of the annotations for the current belief. This practice is in line with the one already explained for the annotated plans above.

Regarding the execution of mental rules, the interpreter evaluates the body of the rule, as in logic programming. Notice the wide variety of mental rules that could be implemented and therefore interpreted, as in most of the implementations of modern logic programming languages.

Concluding the presentation of the solver, it is worth to highlight once more, that the current interpreter is capable of executing a program written in AgentSpeak(L) and since it is written in SWI-Prolog, all the implemented Prolog library predicates, are compatible with the interpreter and therefore, executable.

## 4.4 Failure Handling

Failure Handling is definitely a demanding aspect of agent programming, as well as a great challenge for the research community. There are several implementations of failure handling in agent programming languages (Hübner et al., 2006; Sardina and Padgham, 2011). It is the nature of the agent oriented systems, where the time is considered as a critical factor, in combination with the complexity of such systems, concerning the parallel execution and the rapid changes that highlight the importance of such a feature. Although it could be limited, failure exists in any kind of system and thus, it should be provisioned (where it could) and monitored, restricting its consequences.

Agents perform actions, causing changes to their environment and receiving feedback from it. These changes concern the real world environment and as a result, in many cases it seems complicated to estimate their impact. Thus, it is essential to monitor agents' actions in a MAS, in order to avoid malfunctions that could make the whole system to adopt a chaotic behaviour.

For example, consider a MAS implemented in a traffic control system, where agents are responsible to control the function of the traffic lights. The function of a traffic light depends on the traffic conditions in the area close to it and thus, the function of other traffic lights, in a certain radius. Once an agent that controls a traffic light fails for some reason, the respective light would become inoperable. Sequentially, the other lights nearby it, would be influenced. Rapidly, this initially estimated as a simple conflict, would drive the whole network to chaos, causing huge traffic delays. The loss, in such a system could be hardly estimated, while restoring the system and start from the beginning, where everything seems perfect, is almost impossible, in real-world conditions.

Hence, it is vital to find an efficient way to recover timely from failure, since it is impossible to avoid it. In agent oriented programming, similarly to the logic programming, the level of flexibility is high. Thus, restricting agents' actions, seems to be an inconsistent practice, for such a programming environment. However, it is more efficient to provision, as many cases as we could, aiming to protect agents from deadlocks. So, the general scope is to provide agents with options, preparing them to face any conditions, peculiar or not. Agents should know what to do, which action to perform in every moment, in any case. Options, in agent oriented programming, means plans.

The failure mechanism in the current project, aims to give the opportunity to the programmer to provision and handle failure. Similarly to Jason approach, once a plan (`+g`) fails, another recovery plan (`-g`), provided by the programmer, is executed, since it is applicable. Thus, this plan (`-g`), plays the role of an alternative, for achieving g. Also, the execution of -g, might be used to bring the system back to a reasonable state, in which the regular plans (`+g`), could be applicable again.

The failure handling mechanism, expands the `find_plan` process, as it is defined in the interpreter. First, the interpreter searches for a plan of the form `+g` that corresponds to an event `g`. Once the process fails for some reason, the interpreter seeks for a plan of the form `-g`. Of course, there are many cases, which could cause failure. Distinguishing between these cases, it is necessary to define the following ones:

- *No applicable plan exists*. It describes the case where none of the provided plans (`+g`), has true context, in order to be executable.
- *No relevant plan exists*. It refers to the case where there are no plans of the form `+g`, corresponded to the event `g`.
- *Other reasons of failure*. It is also possible for a plan to fail, during its execution. All the contents of a plan, such as actions, internal actions, sub-plans and expressions run the risk of failure.

Although it is possible to define the courses of failure, it depends on the programmer's ability, to provision and handle failure, providing recovery to the system. Arranging the failure mechanism in a more efficient way, it is preferable for the programmer to use annotations, with respect to the case of failure. Thus, the following annotated failure handling plans could be used:

```
-g#[error(non_applicable_plan)]
-g#[error(no_relvevant_plan)]
```

Of course, the use of annotations in failure handling plans is not necessary for the programmer, who has also the option to use non-annotated plans of the form `-g`, to handle failure. These "naked" failure handling plans (`-g`), are executed in case that no other failure handling plan is applicable, always depending on the case of failure.

Besides the failure handling mechanism, there is also a variety of messages, used to inform the user about the case of failure. The following table, presents all the possible cases during the execution of the program, including "regular" execution, failure and recovery (Table 4.2).

| **Plan Executed** | **Description** | **Recovery** |
|---|---|---|
| `+g` | Plan found and executed with success | ✓ |
| `-g#[error(non_applicable_plan)]` | No applicable plan found, but an annotated plan for that case was provided and executed | ✓ |
| `-g#[error(no_relvevant_plan)]` | No relevant plan found, but an annotated plan for that case was provided and executed | ✓ |
| `-g` | A (known or unknown) failure occurred, but an annotated plan was provided and executed | ✓ |
| `-` | Failure handling plan found but it is non-applicable | ✗ |
| `-` | No applicable plan found | ✗ |
| `-` | No relevant plan found | ✗ |

*Table 4.2 Cases of Failure Handling during the execution of the program*

The part of the code, concerning both the failure handling and the message throwing mechanism, is presented below:

```
%%%%%%%%%%%% Failure Handling Mechanism%%%%%%%%%%%%
%%%%%%%%%% Failure Conditions in Achievement Plans%%%%%%%%%
%% Annotated Failure Handling Plan(error(non_applicable_plan))
find_plan(+!E,_,_):-
     % at least one plan (+!E) exists but is not applicable
     plan(+!E,_,_),
     % error provisioned
     plan(-!E#[error(non_applicable_plan)],C,P),
     test_condition(C),
     !,
     solve(P).
%% Annotated Failure Handling Plan (error(no_relevant_plan))
find_plan(+!E,_,_):-
     not(plan(+!E,_,_)),
     plan(-!E#[error(no_relvevant_plan)],C,P),
     test_condition(C),
     !,
     solve(P).
%% NON-ANNOTATED Failure Handling Plan (catch all)
find_plan(+!E,_,_):-
     plan(-!E,C,P),
     nl,
     test_condition(C),
     solve(P),
     !.
%% NON-applicable Failure Handling Plan
find_plan(+!E,_,_):-
     plan(-!E,_,_),
     nl,
     write(['No Applicable Plan found for goal event ',+!E]),nl,
     write(['Failure  Plan  found  but  NOT  applicable  for  goal
     event ',+!E]),nl,
     nb_getval(remaining,P2),
     write(['Could not finish intention:' ,P2]),nl,
     !,
     fail,
     !.
%% In case there is NO applicable plan at all
find_plan(+!E,_,_):-
     plan(+!E,_,_),
     nl,
     write(['No Applicable Plan found for goal event ',+!E]),nl,
     write(['No Fail event was generated for goal ',!E]),nl,
     nb_getval(remaining,P2),
     write(['Could not finish intention:' ,P2]),nl,
     !,
     fail,
     !.
%% In case there is NO relevant plan at all
find_plan(+!E,_,_):-
     not(plan(+!E,_,_)),
     nl,
     write(['No Relevant Plan found for goal event ',+!E]),nl,
     nb_getval(remaining,P2),
     write(['Could not finish intention:' ,P2]),nl,
     !,
     fail,
     !.
```

It is worth to notice that the code above, refers to the handling of the achievement plans (`+!g`). The code concerning test plans (`+?g`), is completely equivalent to the one presented above.

As for the message throwing mechanism, it is highlighted that it provides information to the user not only concerning the “type” of failure, but also the remaining actions, which are included in the intention stack that is discarded. This is achieved through the use of the global variable `remaining`, which stores the actions that sequence the one executed, inside plan’s body. This storing is performed in the solver and more specifically, in the part of the code presented below:

```
        solve(P1;P2):-
          !,nb_setval(remaining,P2),
          solve(P1), !, solve(P2).
```

Finally, the example below, illustrates the function of the failure handling mechanism, as described above.

```
% Plans
+!testFailureHandlingMechanism :: true <-
     !failureHandlingAchievePlan_NA;
     !failureHandlingAchievePlan_NR;
     !failureHandlingAchievePlan_.
+!failureHandlingAchievePlan_NA :: 1>2 <-
     %this is expected to fail, the message should not appear
     write("test !! -ok. (Solving an achieve plan - )"),nl.
-!failureHandlingAchievePlan_NA#[error(non_applicable_plan)] ::
true <-
     write("test FH_A1 ok. Annotated failure handling plan for
     error(non_applicable_plan)"),nl.
-!failureHandlingAchievePlan_NR#[error(no_relvevant_plan)]   ::
true <-
     write("test FH_A2 ok. Annotated failure handling plan for
     error(no_relvevant_plan)"),nl.
-!failureHandlingAchievePlan_ :: true <-
     write("test  FH_A3  ok.  Non-annotated  failure  handling
     plan"),nl.
```

While the console output, should be the following:

```
?- start_agent.
true.
?- load_agent('failurehandling.pl').
true.
?-  test  FH_A1  ok.  Annotated  failure  handling  plan  for
error(non_applicable_plan)
```

```
test FH_A2 ok. Annotated failure handling plan for
error(no_relvevant_plan)
test FH_A3 ok. Non-annotated failure handling plan
```

## 4.5 The Agent as a Multi-Threaded Application

In this section we describe the design of the agent as a multi-threaded application. Following the model of multithreading, multiple threads co-exist in the framework of a single process, sharing system's resources. These threads are used either for the simultaneous execution of agent's intentions or for communication purposes, as it will be explained in details later on. Hence, the current project capitalizes on the feature of parallel execution, as a very interesting one, also considered as a natural feature of the agent-oriented systems.

Agents perform several different actions, causing changes to their environment and receiving feedback from it. This continuous interaction becomes seriously more complicated, taking into account the relations that are created between a large number of agents, in the context of a MAS. It takes significant effort to monitor such a volatile behaviour, that the agents adopt. Additionally, it is a great challenge to control and direct the great number of actions that take place in *parallel*, inside a MAS.

Thus, an efficient agent-oriented design would be based on parallel, fast and autonomous execution. Additionally, such a design should be intolerant in interruptions. As referenced, in a MAS, there are several actions that take place simultaneously and there is no criterion to decide which one should be postponed or executed immediately. Parallel execution is the only way to address this challenge. Time is considered as the key point in a MAS, as it is the most critical factor in agent-oriented programming. Agents, should reason and act rapidly, since significant changes in the environment, that might influence their actions or plans, may occur every single second. Every agent has its own beliefs, desires and intentions, as it is clearly defined in the respective BDI model, so it is not always useful to group the agents, in order to manage their behaviour. Thus, autonomy is a non-negotiable characteristic of agents.

On the other hand, multithreading allows the programmer to address the challenges specified above, both managing agents and allocating system's sources. Having said that, it is worth to highlight that multithreading is an efficient way to increase the utilization of a single application by using *thread-level parallelism*, allocating system's sources in a smooth way. Apart from the above mentioned features, multithreading allows the programmer to handle the communication between the agents. This aspect will be thoroughly explained in a separate section. Thus, multithreading is considered as a robust technique for supporting project's design and especially the agent's execution cycle.

The part of the code that refers to thread handling, concerning the current project, is given below:

```
%%% Code for thread handling - threads.pl
%%% start_agent/0
%%% Starts the main_agent_thread
start_agent:-
     thread_create(main_agent,_,[alias(main_agent_thread)]).
%%% stop_agent/0
%%% Stops the agent
stop_agent:-
     thread_send_message(main_agent_thread,end),
     thread_join(main_agent_thread,_).
%%% main_agent/0
%%% Main Agent thread.As usual this is a repeat fail loop.
main_agent:-
     repeat,
     thread_get_message(Mess),
     once(process_message(Mess)),
     once(clear_processes),
     off(Mess).
%%% off/0
%%% Succeeds and exits the main failure drive loop (Shut Down)
off(end):-!.
%%% Message to stop agent Execution.
process_message(end).
%%% The role of the clear message is to do any joins
process_message(clear).
%%% creates a thread to handle any plans triggered from adding
a belief in BB
process_message(addb(X,Atts)):-
     name_thread(add_belief(X,Atts),Alias),
     thread_create(solve(addb(X,Atts)),_,[alias(Alias)]).
%%% creates a thread to handle any plans triggered from deleting
a belief from BB
process_message(delb(X,Atts)):-
     name_thread(del_belief(X,Atts),Alias),
     thread_create(solve(delb(X,Atts)),_,[alias(Alias)]).
%%% creates a thread to handle an achieve message
process_message(achieve(X)):-
     name_thread(achieve(X),Alias),
     thread_create(solve(X),_,[alias(Alias)]).
%%% creates a thread to handle an ask message
process_message(asks(X,Content)):-
     name_thread(asks(X,Content),Alias),
     thread_create(answer_message(X,Content),_,[alias(Alias)]).
%%% name_thread/2
%%% Create a unique alias for the thread.
name_thread(Task,Alias):-
     get_time(T),
     term_string(iStack(Task,T),S),
     atom_string(Alias,S).
%%% clear_processes/0
%%% Succeeds by joining any threads that have succeeded.
clear_processes:-
     thread_property(I,status(true)),
     thread_join(I,_),
     write([I,Status]),nl,
     fail.
%% Clearing threads that have failed.
clear_processes:-
        thread_property(I,status(false)),
        thread_join(I,_),
        write(['***',I,failed]),nl,
       fail.
clear_processes.
```

```
%%% Predicates used by others
generate_agentspeak_event(AGL_Event):-
     thread_send_message(main_agent_thread,AGL_Event).
```

The code above, describes the mechanism that every agent is obligated to respect, once it joins the MAS. As multithreading is already implemented in SWI-Prolog, there are plenty of predicates, dedicated for multithreaded applications (predicates in use for the current project in Appendix 2).

Hence, initially, a thread is created for every agent that enters the MAS, called `main_agent_thread`. It could be considered playing the role of the main hub, for the events to be handled by the agent. Thus, once the `start_agent` is executed, the `main_agent_thread` is running, putting the respective agent in an “alert” state, ready to receive “processing messages”.

It is of high importance, to separate this type of “processing messages” from “communication messages”, as these messages are completely different, from the ones used in communication, between the agents of a MAS.

Once a “processing message” is received from the `main_agent_thread`, the latter is responsible for generating other threads to handle it. These “processing messages” correspond to goals that the agent has to achieve. On the other hand, threads that are generated from the main thread, correspond to the *intentions* of the agent. The role of the predicate `generate_agentspeak_event/1`, is to send a message to the main thread, triggering it to start processing and therefore, to create another thread. This reflects the creation of a new intention stack, as in the following cases:

- Goal to be achieved
- Goal pursued as a separate intention (`^^g`)
- External belief addition/deletion (*through communication with other agents*)
- Internal belief addition/deletion (*parsing procedure*)

As presented earlier in the code for thread handling, there are dedicated threads for handling a belief addition or deletion, as well as an `achieve` or an `ask` message. Of course, the “achieve” thread is used every time the agent intends to accomplish a task, either it derives from an internal or an external event. Thus, this thread is generated when the agent has to achieve a goal, triggering the execution of an achievement or test plan (`solve(X)`).

As for the second case, concerning a goal pursued as a separate intention (`^^g`), the code below is to be executed:

```
solve(^^E):-
    generate_agentspeak_event(achieve(!E)).
```

It is also worth to mention the importance of the predicate `clear_processes`, which removes all the succeeded, as well as the failed threads, ensuring the sustainability of the system's sources.

By following the above described mechanism, agent's intentions could be executed in parallel, autonomously, but still centrally controlled (by the main thread). The schema below, is used to present the vital role of multithreading in agent's execution cycle (Figure 4.1).

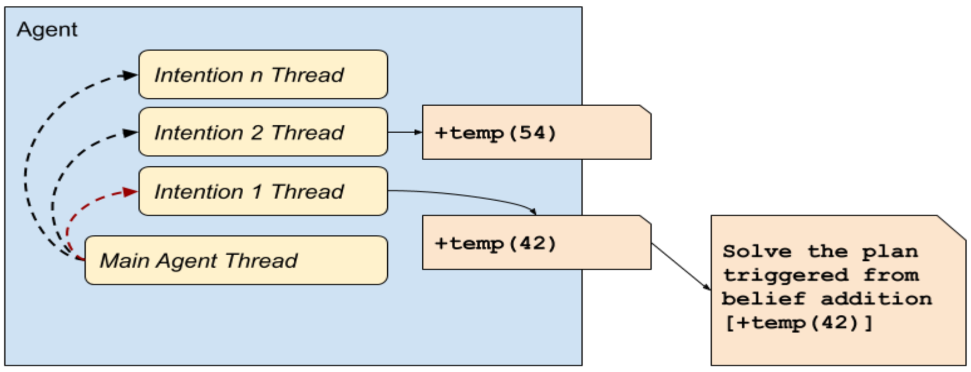

*Figure 4.1 Multithreading in Agent's Execution Cycle*

The example presented in the above schema, refers to the addition of the beliefs `temp(43)` and `temp(54)`, in the belief base of the agent. This is performed by the parser, when a program is loaded, or when the belief is formulated as an environmental percept or sent as a message from another agent. Considering the first case, the program that is loaded is presented below, we focus on specific parts of the code that the interpreter would consult, executing a belief addition (Table 4.3).

```
%% Belief Addition Example add_belief.pl
%% Beliefs
temp(42).
temp(54).
%% Plans %%
% plan triggered for every belief addition of the form temp()
+temp(A) :: true <-
     write(['temperature', A , 'added']),nl.
```

| **Executed code** | **Function** |
|---|---|
| `parse_term(Belief):-`<br>` !,`<br>` add_belief(Belief,[source(self)]),`<br>` !,`<br>` fail.` | *parsing the belief* |
| `add_belief(B,Atts):-`<br>`     is_list(Atts),!,`<br>`     assert(belief(B,Atts)),`<br>`     assert(B),add_record(B),`<br>`     generate_agentspeak_event`<br>`                         (addb(B,Atts)).` | *sending a "process message" to the* `main_agent_thread` |
| `process_message(addb(X,Atts)):-`<br>` name_thread(add_belief(X,Atts),Alias),`<br>` thread_create`<br>`     (solve(addb(X,Atts)),_,[alias(Alias)]`<br>`).` | *creating a* ***thread*** *to execute the belief addition* |
| `solve(addb(B,A)):-`<br>` find_plan(+B#A,_,P),`<br>` !,`<br>` solve(P).` | *executing the plan that is triggered from the belief addition* |

*Table 4.3 Executed Code in case of a Belief Addition*

Thus, before loading the above program, the only thread that exists, is the `main_agent_thread` (as it is created after the `start_agent` command execution). Once the program is loaded successfully, we get the message (printed in the console). Additionally, one could check the threads list, by executing the respective command (`threads`). The last thread that is active during the execution phase, is the one appeared in the list, under the name while checking the threads that have been created during the execution, we could observe the two new threads (named as defined in the code for thread handling) as described above. A console snapshot is given below:

```
?- start_agent.
true.
?- threads.
%     Thread      Status     Time  Stack use    allocated
% -----------------------------------------------------
% main_agent_thread running 0.000      800      120,808
true.
?- load_agent('add_belief.pl').
true.
?- [temprature,42,added]
[temprature,54,added]

threads.
%     Thread      Status     Time  Stack use    allocated
% -----------------------------------------------------
% main_agent_thread running 0.000      800      120,808
%'iStack(add_belief(temp(54),[source(self)]),1558510406.0
96266)' true
true.
```

As already mentioned, during the execution, the predicate `clear_processes` removes all the succeeded threads. Thus, the thread that finally remains in the corresponding list, is the last one generated. In the case above, this thread is the one used for the belief addition (`+temp(54)`).

## 4.6 Summary

This chapter describes the main functions of the current meta-interpreter, focusing on the parser and the solver. The internal representation of the basic components of an AgentSpeak(L) program, such as beliefs, mental rules and plans, is thoroughly explained. The next section refers to the solver, which is responsible to handle the input, unifying the *events* with the existing *plans* in the agent's plan library. Hence, it is provided an extended analysis, as well as a categorization of goals, given directly or triggered to be executed through the solver. The next section describes the implemented failure handling mechanism, while the chapter concludes presenting the overall design of the agent as a multi-threaded application, focusing on the parallel execution ability.

# CHAPTER 5

# Agents' Communication

Probably the most interesting feature in agent oriented architecture, is their ability to communicate with each other, when living in the same environment (MAS). Implementing agents' communication, introduces a significant number of challenges concerning the timing and the control of the messages exchanged, between the agents of a MAS. Various approaches could be found in the literature, concerning this demanding implementation (Becker et al., 2009; Werner, 1989).

In the current framework, agents might have same or similar goals and therefore, cooperate with each other, with the aim to achieve them. However, there are agent environments, where the agents compete each other, aiming to achieve individual goals. In both cases, a robust communication mechanism should be defined, with respect to which, the agents would be able to communicate, cooperating or coordinating their actions, aiming to reach their goals.

Consider a traffic control system, where agents are in charge of traffic lights. In this MAS, agents need to cooperate with each other, sending and receiving data regarding the traffic conditions. This communication needs to be based on a robust and efficient specific mechanism that the necessary protocols will be implemented on. On the other hand, agents that control automated vehicles compete each other, aiming to achieve their goal(s), by reaching their destination, in a fast and safe way. Considering this second case, although agents are not cooperate, they have to coordinate their actions, exchanging communication messages, with the overall aim to avoid accidents.

## 5.1 Communication Design

As already mentioned, multithreading is considered as an efficient technique for the design of the agent oriented systems. It is worth to remind the role of the `main_agent_thread`, which is responsible for the creation of the intention handling threads. In a similar manner, a thread is responsible for communication handling, called `com_thread`. This thread is also responsible to invoke/post appropriate events to the `main_agent_thread`, after receiving a message.

The Figure below (5.1), gives an overview of the multithreaded processing, focusing on the agent's communication cycle.

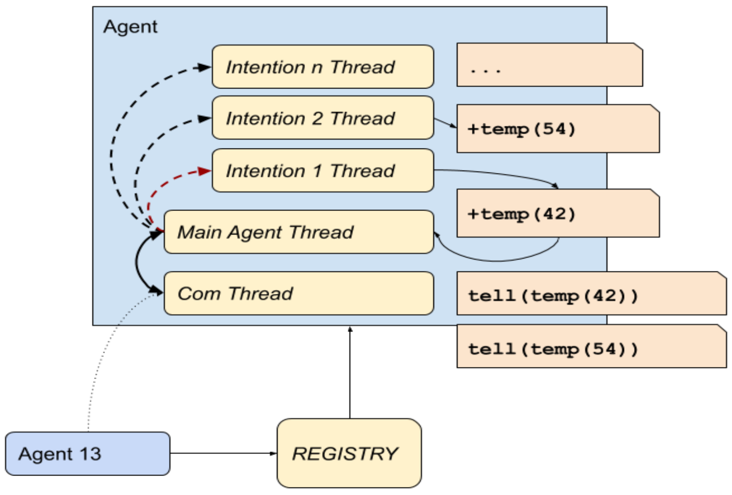

*Figure 5.1 Multithreading processing in the Agent's Communication Cycle*

In the current work, communication between agents is achieved through the use of the messaging platform *Pedro* (Robinson and Clark, 2010). *Pedro (ver. 1.9 Mar 2019)* is a message subscription system that supports Prolog like messages.

*Pedro* also provides a mechanism for peer-to-peer communication, along with the subscription framework. The communication between each agent (client) and the Pedro server is achieved through a pair of sockets. The one is used for notifications (message passing), while the other is used for the server to acknowledge messages from the client.

Every agent of the MAS (client), is registered to *Pedro*, establishing connection with *Pedro* server. Once registered, the agent takes a unique name, which is also used by other agents, when communicating with them (peer-to-peer communication). It is vital for the system, to keep track of the mapping between subscriptions and names, and this is the role of another entity, part of the system, the *Registry*.

The role of the Registry is very important for the system. Once the Registry is connected to Pedro, it is responsible for receiving messages from the agents that belong to the MAS and handling them accordingly. Registry is also responsible for keeping and updating a list of all registered agents. The last is vital for the connection of the agents, which is achieved following these simple steps:

- Connect to Pedro
- Register to the Registry
- Subscribe to the agent list

For the connection to the server, it is needed the agent's Name and the port. The default port that is used for the current project, is `4550`. The provided Name of the agent, is used for the address that the agent should have, in order to be able to communicate with the other agents in the MAS. After registering, an address like the following is given to every agent:

```
agent_Name@IP
```

Registry is responsible for keeping the agent list. Every time an agent enters the MAS and completes its registration successfully, the Registry adds its name to the respective agent list, confirming the update by printing a message that refers to the last registered agent:

```
    p2pmsg(registry@10.140.5.212,bob@10.140.5.212,registering
_agent)
```

The opposite should happen when some agent is deregistered from the MAS. The agent list should also be kept updated.

Thus, both *Registry* and every agent included in the MAS, are defined as different entities, connected through *Pedro* server. Through this connection, the agents exchange messages with the registry in order to be tracked, while they are also capable of sending messages "directly" to each other.

```
start_agent_communication(nocom):-!.
%% The default port for Pedro server is 4550
start_agent_communication(Name):-
  establish_connection(Name,4550),
thread_create(main_agent_com_code,_,[alias(com_thread)]).
stop_agent_communication:-
  whoami(D),pedro_send_pp_message(D,off),
  wait(500),
  pedro_deregister,
  close_pedro_connection.
main_agent_com_code:-
  repeat,
  once(get_pedro_message_data(Rock,Content)),
  once(process_p_message(Rock,Content)),
  fail.
```

When a registered agent wants to send a message to another registered agent, it simply sends a notification of the form `p2p(Addr,Sender,Content)` to *Pedro*. The first argument (`Addr`) refers to the receiver of the message, the second argument (`Sender`) refers to the agent who sends the message, while the last one (`Content`) is the actual message. For example:

```
p2pmsg(alice@192.168.43.251,bob@192.168.43.251,tell(hello))
p2pmsg(bob@192.168.43.251,alice@192.168.43.251,tell(speak))
```

Once Pedro receives such a notification, forwards it to the address of the receiver, after checking the respecting agent list. The code presented below, refers to this mechanism that handles point to point messages:

```
%%% Sending point to point agent messages.
pedro_send_pp_message(Receiver,Content):-
  match_list(Receiver,Addr),Addr \= none,
  whoami(Me),
       send_ack_pedro_message(p2pmsg(Addr,Me,Content),suc
cess),
  !.
pedro_send_pp_message(Receiver,Content):-
  write(['Failed to send message',Receiver,Content]).
%%% Already in correct form.
match_list(Receiver@IP,Receiver@IP):-!.
%%% Matching a name
match_list(Receiver,Receiver@IP):-
    agent_list(ListofAgents),
    member(Receiver@IP,ListofAgents),!.
match_list(_Receiver,none).
```

Pedro also gives the opportunity to the client to categorize messages, according to its own purposes. This is achieved through the use of the variable Rock, which pairs the messages. In the current project, this value is used to distinguish between peer-to-peer messages and the ones sent to update the agents' list. Thus, for this reason, agents exchange specific messages (`Rock = 1`), which allow them to keep always an updated copy of the list that includes the agents of their MAS. On the other hand, peer-to-peer messages, which allow agents to communicate with each other, take a different value for the respecting variable (`Rock = 0`).

For example, the messages presented below illustrate the use of variable `Rock`:

```
[1,agent_list([alice@192.168.43.251])]
[1,agent_list([alice@192.168.43.251,bob@192.168.43.251])]
```

```
[0,p2pmsg(alice@192.168.43.251,bob@192.168.43.251,test(ok))]
[0,p2pmsg(bob@192.168.43.251,alice@192.168.43.251,tell(yes))]
```

The first pair of messages, is used for updating the agents' list, while the second pair is an example of peer-to-peer communication (`p2pmsg`). Thus, in the current project, messages with `Rock = 1`, are used to extend the agent list, by adding a new agent that is connected to the MAS. On the other hand, when rock is equal to `0`, the message refers to communication between the agents and it is processed accordingly. Details for this processing are given in the corresponding Section.

## 5.2 Message Syntax

The syntax that is followed when sending a message, it is compliant with the ACL standards, using performatives (i.e. *tell*, *untell*, *ask* etc.) and follows the form of:

```
send(Receiver, ACT, Term).
```

Obviously, the first argument refers to the agent that receives the message, the second is the action that corresponds to one of the defined KQML performatives (*tell*, *untell* etc.), while the third contains the term that the sender wants to deliver to the receiver.

The code presented below is to define the syntax that the program should follow when sending a message:

```
send(Receiver,ACT,Term):-
  Content=..[ACT,Term],
  pedro_send_pp_message(Receiver,Content).
```

Thus, as it is defined from the code above, the variable `Content`, includes both the performative and the term to be delivered. Some examples of `Content`, are the following:

```
tell(temp(42))
untell(weather(rainy))
ask(colour(green))
```

## 5.3 Message Processing

After describing the way that messages are handled from Pedro server and the syntax of the messages that the sender should follow, it is worth to refer to the process that takes place at the side of the receiver.

Once the `com_thread` is created, i.e. the thread responsible for the communication, the agent is ready to receive messages. Depending on the content of the message received, the `com_thread` generates and sends a corresponding message to the `main_agent_thread`. This message includes the content and the id of the sender of the message that was received, following one of the types below:

| **Message received from `com_thread`** | **Message to the `main_agent_thread`** |
|---|---|
| `tell(Content)` | `addextb(Content,[source(Sender)])` |
| `untell(Content)` | `delextb(Content,[source(Sender)])` |
| `achieve(Content)` | `extachieve(Content#[source(Sender)])` |
| `test(Content)` | `exttest(Content#[source(Sender)])` |
| `ask(Content)` | `asks(Sender,Content)` |

*Table 5.1 Message processing*

The Table above (5.1), provides the correspondence between the communication messages and those generated internally addressed to the main agent thread. The part of the code below, presents the processing of the peer-to-peer messages, showing the conversion of the message received to the dedicated one forwarded to the `main_agent_thread`, as well as the generation of the respecting event:

```
process_p_message(_,p2pmsg(_,Sender,Message)):-
  fix_message_for_main(Sender,Message,ToMain),
  generate_agentspeak_event(ToMain).
%% Forward process messages to main agent thread
```

```
     fix_message_for_main(Sender,tell(Content),addextb(Content
,[source(Sender)])).
     fix_message_for_main(Sender,untell(Content),delextb(Conte
nt,[source(Sender)])).
     fix_message_for_main(Sender,achieve(Content),extachieve(C
ontent#[source(Sender)])).
     fix_message_for_main(Sender,test(Content),exttest(Content
#[source(Sender)])).
     fix_message_for_main(Sender,ask(Content),asks(Sender,Cont
ent)).
```

Every type of communication message, generates an event that triggers some agent action, i.e. a change in its belief base (`tell/untell`), execution of an achievement (`achieve`) or a test goal (`test`), or checking a condition (`ask`). The code below defines the procedure that is followed by the `main_agent_thread` of the receiving agent, concerning message handling:

```
     process_message(addextb(Content,Atts)):-
             add_belief(Content,Atts).
     process_message(delextb(Content,Atts)):-
             remove_belief(Content,Atts).
     process_message(extachieve(X)):-
             name_thread(extachieve(X),Alias),
             thread_create(solve(!X),_,[alias(Alias)]).
     process_message(exttest(X)):-
             name_thread(exttest(X),Alias),
             thread_create(solve(?X),_,[alias(Alias)]).
     process_message(asks(X,Content)):-
             name_thread(asks(X,Content),Alias),
             thread_create
                 (answer_message(X,Content),_,[alias(Alias)]).
```

Below we detail how the agent behaves upon reception of the different messages.

`tell message.` Messages of this type, are used to provide information to the receiver. Once an agent gets a tell message, it stores its content, by adding a belief on its belief base. Of course this annotated belief addition (`add_belief(Content,Atts)`) is to generate a new (*external*) event for the agent (`generate_agentspeak_event(addb(B,Atts))`), triggering any plans that may exist for this purpose.

`untell message.` The `untell` message is used to recall the information sent previously to the receiver. It is very important to highlight that using this type of message, the sender is capable of removing a belief from the receiver's belief base, provided that it was originated from it earlier. This means that the sender is not allowed to remove any beliefs from different sources, at the receiver's base, even if they refer to the same content.

`achieve message.` This message triggers the receiver to execute an achievement plan that exists in its plan library. This plan should have the respective annotation, corresponding to the source of the sender (`+!g#[source(sender)]`), otherwise it cannot be triggered.

`test message.` The use of such type of message is similar to the previous one, but the test message refers to the case where a test plan is to be triggered (`+?g#[source(sender)]`).

`ask message.` This message is used for the sender to check, if a condition is true, according to the receiver. This condition might refer to an existing belief or it may trigger a mental rule in the receiver's base. If that condition is satisfied, the receiver replies to the sender, sending a special message (`answer(Content)`). Once this message is received from the sender of the initial message, it triggers a belief addition to its base. This belief is annotated with the source of the receiver of the initial message. The code presented below, refers to the handling of such type of messages:

```
answer_message(Sender,Content):-
     test_condition(Content),
     pedro_send_pp_message(Sender,answer(Content)).
fix_message_for_main(Sender,answer(Content),addextb(Content,[s
ource(Sender)])).
```

The implemented types of messages, could cover a variety of communication purposes. Obviously this set can be extended to cover more cases, as for example `tellHow`, `untellHow` for plan exchanging, or `broadcast` for sending messages to the whole agent list. As for the use of the implemented types, is highlighted the next section with a test program that covers these cases.

## 5.4 Test Cases

This Section provides a test program, which includes all the appropriate test cases that correspond to the types of messages described earlier. For the needs of the program, the agent system consists of two agents (`alice` and `bob`), exchanging messages with each other. Apart from the two agents, the full system includes the *Registry*, which is used for the subscription and of course the *Pedro* server. Aiming to run the program smoothly, the following steps should be followed:

1. Start the *Pedro* server.
2. Start the *Registry*.
3. Start the agent `Alice`.
4. Start the agent `Bob`.

The steps above, start four different program instances each corresponding to one of the different entities, as described above. Hence, the connection between the two agents and the Registry should be established, and the following message should be displayed in the registry console:

```
    Registry Ready
    Waiting
    p2pmsg(registry@192.168.1.10,alice@192.168.1.10,registeri
ng_agent)
```

```
Waiting
p2pmsg(registry@192.168.1.10,bob@192.168.1.10,registering
_agent)
Waiting
```

Sequentially, the agents' list should include both the two registered agents:

```
[1,agent_list([alice@192.168.1.10,bob@192.168.1.10])]
```

Every agent that belongs to the MAS, executes its own code. The code of the two agents (`alice` and `bob`) that are created for the needs of the testing, is presented below:

```
%%% This is alice
%%% Beliefs
test(aa).
test(bb).
%%% Plans
+hello#[source(S)] :: true <-
     write([received,hello,S]);
     send(S,tell,hithere).
-hello#[source(S)] :: true <-
     write([received,delbel,S]);
     send(S,tell,delbelief).
+!speak#[source(S)] :: true <-
     write([received,speak,S]);
     send(S,tell,speak).
+?check#[source(S)] :: true <-
     write([received,test_check,S]);
     send(S,tell,checked).
---------------------------------------------------------------------------------------------------------
%%% This is bob
%%% Beliefs
test(1).
test(2).
%%% Plans
```

```
+!com_test:: true <-
     send(alice,tell,hello);
     send(alice,untell,hello);
     send(alice,achieve,speak);
     send(alice,test,check);
     send(alice,ask,test(aa));
     wait(300);
     write('test 5 -ok. ask message');nl;
+hithere#[source(_)] :: true <-
     write('test 1 -ok. message received');nl;
     !testBel.
+!testBel :: hithere <-
     write('test 2 -ok. tell message');nl.
+!testBel::true <-
     write('test 2 -failed. tell message').
+delbelief#[source(_)] ::true <-
     write('test 3 -ok. untell message');nl.
+speak#[source(_)] ::true <-
     wait(200);
     write('test 4 -ok. achieve message');nl.
+checked#[source(_)] :: true <-
     wait(500);
     write('test 6 -ok. test message').
```

In the program above, agent `bob` sends a set of messages to `alice`, testing every implemented message type. After receiving these messages, `alice` confirms them by printing a special confirmation message for each one of them (`write([received,Content,Sender]))`. Additionally, for every message that `alice` receives, it sends a `tell` message to `bob`, confirming that its message was received. The content of this `tell` message that `alice` sends back to `bob`, corresponds to the content of the initial message that was sent from `bob` to `alice`.

On the other side, when this confirmation `tell` message from `alice` is received from `bob`, it triggers the execution of a plan, depending on the content of the belief that is added at `bob’s` belief base. Thus, when bob receives `hithere` from `alice`, it executes the respective `+hithere` plan, printing a message to the console that informs the user for the accomplishment of the procedure.

The Figure below (5.2) shows this message exchanging between the agents `alice` and `bob`:

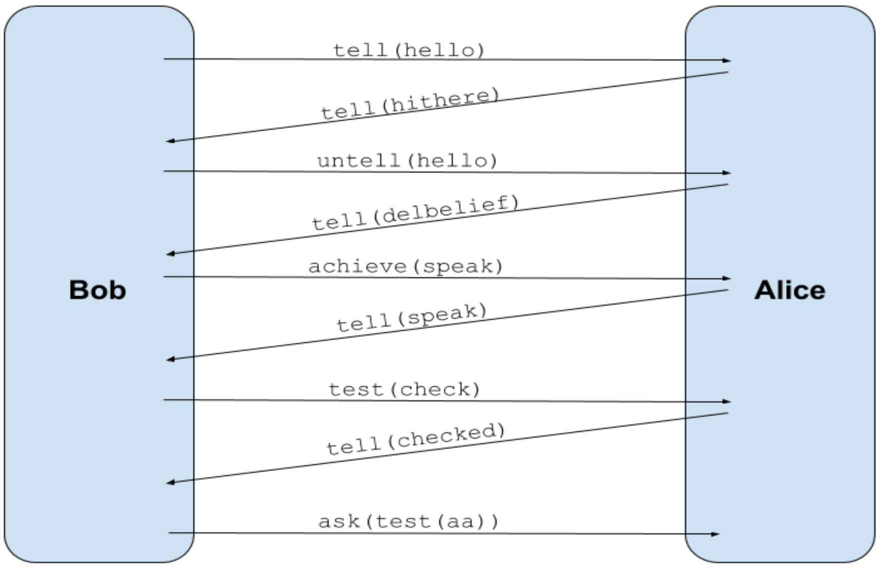

*Figure 5.2 Message Exchanging between Agents Bob & Alice*

At the end of the execution, it is also useful to check agent `bob’s` belief base, ensuring that the messages exchanged, have triggered the appropriate belief additions. This is performed, executing the command `belief(Content,Annotation)` at `bob’s` console:

```
Content = test(1),
Annotation = [source(self)] ;
Content = test(2),
Annotation = [source(self)] ;
Content = hithere,
Annotation = [source(alice@'192.168.43.251')] ;
```

```
Content = speak,
Annotation = [source(alice@'192.168.43.251')] ;
Content = delbelief,
Annotation = [source(alice@'192.168.43.251')] ;
Content = checked,
Annotation = [source(alice@'192.168.43.251')] ;
Content = test(aa),
Annotation = [source(alice@'192.168.43.251')] ;
```

It is worth to highlight that the last printed belief, is generated as a result of the `ask` message, sent from `bob` to `alice`. That belief has the role to confirm that `alice's` belief base includes the belief `test(aa)`, as it was asked earlier from `bob`. It is automatically generated, as part of the `ask` message procedure and not as a result of a `tell` message sent from `alice` to `bob`, as it happens with the rest messages in the above program.

The above presented test program, concludes the section referring to agent's communication. As mentioned earlier, the ability of the agents to communicate with each other is considered as a very interesting feature. As mentioned earlier, the programmer is allowed to expand this capability, adding new features, message types or forms, aiming to improve agents' functionality.

## 5.5 Summary

This chapter presented the communication design of the proposed agent framework. Significant focus was given in describing the multithreaded processing, concerning agents' communication. Pedro, the subscription/notification server that was used for the implementation and its role to the overall architecture of the system, were also described. In the next section, the general format that is followed in message syntax is presented. Extensive analysis of message processing and the role of the dedicated threads is presented in the corresponding section. The chapter concludes with the presentation of a set of test cases that illustrates the agents' message exchanging capabilities.

# CHAPTER 6

# Discussion & Conclusions

This thesis presented a Prolog meta-interpreter for AgentSpeak(L). The combination of the features of modern logic programming with the already implemented features of the language, results in a constructive approach towards modern agent programming. The meta-interpreter is fully compatible with programs written in AgentSpeak(L), supporting all the features defined in the original version, as well as implementing new ones as strong negation, belief and plan annotations, the support of complicated mental rules and higher order Prolog predicates and finally, the failure handling mechanism. The meta-interpreter's design is based in multithreading, enhancing parallel execution in agent's actions and supporting asynchronous communication. Finally, the operability and reliability of the current interpreter was extensively tested, in a wide variety of test cases, demonstrating both the single agent function and the communication between the agents, through message exchanging.

## 6.1 Research contributions

The presented Prolog meta-interpreter for AgentSeak(L), exploiting modern logic programming features, aims to bridge the gap between theory and the practical issues concerned with software agents' programming.

So far, the only commonly used interpreter for AgentSpeak(L), is the one implemented in Java, the actively developed Jason. In this project, the proposed meta-interpreter supports all the basic AgentSpeak(L) features, as well as intend to add new ones, such as strong negation, belief and plan annotations and support of complicated mental rules and higher order Prolog predicates. These attributes, along with the implementation of a failure handling mechanism intend to compose a robust and efficient framework for agent oriented programming.

Multithreading as a widely used and efficient technique, allows the current meta-interpreter to support parallel execution, an attribute with significant impact on the software agents' design. Capitalizing on the benefits of multithreading, the agents are capable of handling independently and thus, executing in parallel their intentions. Consequently, with respect to multithreading design, agents are able to exchange messages, supporting asynchronous communication.

## 6.2 Limitations of research

Limitations of the composed research mainly derive from the diversity that characterizes the field of the agent oriented programming. The variety of approaches, concerning the agent programming languages, as well as the significant number of theoretical models, compose a wide but uncertain field, concerning the implementation of a robust and efficient agent oriented framework.

Another limitation derives from the nature of a meta-interpreter implementation in general, as it takes significant effort to overcome the inconsistency, concerning the syntax. The combination of features (e.g. operators, special characters etc.) of the already implemented language (in our case SWI-Prolog), with features of the interpreted language (in our case AgentSpeak(L)) is considered as a hard to be tackled challenge. In the current project, these incompatibility issues in some cases required the adoption of syntactic forms that differ from the original ones, aiming to support the whole set of both languages' features.

Another limitation pinpointed from the examination of the failure handling mechanism, was that the nature of logic programming does not allow the complete control of the program's execution. This, in combination with the inherent deficiency, regarding the control of agents' actions in real-world conditions inside a MAS, makes the debugging, as well as the implementation of a failure handling mechanism, even more demanding.

Finally, there are certain limitations that derive from the character of synchronous communication that the agents establish. Timing, is considered as a critical factor, influencing the execution of message exchanging process, especially when in most of the cases, there should be established a mechanism, enhancing automatically triggered reactions, from the agents' side. Also the demanding role of the registry, in the communication design, is considered as vital, monitoring and updating the agents' list.

## 6.3 Future work

In light of the potential and the advantages of this project and the limitations identified during the process, there is a number of issues that could be potentially explored. A future extension related to the proposed meta-interpreter could be the use of constraint programming in agent's belief base. Capitalizing on the benefits that derive from the use of Constraint Logic Programming (CLP), the current meta-interpreter could increase the range of problems that the interpreter could handle.

The work carried out in this project, could be also further elaborated with the incorporation of planning from first principals, the implementation of synchronous communication and the extension of the semantics of the language.

In addition, failure handling mechanism could be extended, with the aim to provide recovery even when an agent's action fails. This requires explicit monitoring of the execution of agent's plans, allowing the agent to track the action that triggered the failed one and therefore recover from failure. This multi layering architecture regarding the execution cycle, is considered as demanding task in agent oriented programming.

Also, future research could focus on debugging techniques that could facilitate the use of the current meta-interpreter. Finally, a potential future extension could incorporate the implementation of testing in truly distributed cloud environments.

# Appendix 1

| ***Precedence*** | ***Type*** | ***Name*** |
|---|---|---|
| *1200* | *xfx* | -->, :- |
| *1200* | *fx* | :-, ?- |
| *1150* | *fx* | dynamic, discontiguous, initialization,meta predicate, module transparent, multifile, public, thread local, thread initialization, volatile |
| *1100* | *xfy* | ;,\| |
| *1050* | *xfy* | ->,*-> |
| *1000* | *xfy* | , |
| *990* | *xfx* | := |
| *900* | *fy* | \+ |
| *700* | *xfx* | <, =, =.., =@=, \=@=, =:=, =<, ==, =\=, >, >=, @<, @=<, @>,@>=, \=, \==, as, is, >:<, :< |
| *600* | *xfy* | : |
| *500* | *yfx* | +,-,/\,\/,xor |
| *400* | *yfx* | *, /, //, div, rdiv, <<, >>, mod, rem |
| *200* | *xfx* | ** |
| *200* | *xfy* | ^ |
| *200* | *fy* | +,-,\ |

| | | |
|---|---|---|
| *100* | *yfx* | *.* |
| *1* | *fx* | *$* |

*Table A1.1 System operators SWI-Prolog as defined in http://www.swi-prolog.org*

# Appendix 2

| ***Predicate*** | ***Content*** | ***Description*** |
|---|---|---|
| `thread_create/3` | Goal, Id, Alias | Creates a thread to handle a specific task or the `main_agent_thread` |
| `thread_send_message/2` | Id(Receiver),Term | Communicate a “process message” to the respective thread |
| `thread_get_message/1` | Term | Receive a “process message” |
| `thread_join/2` | Id, status(true) | Terminates a succeeded thread, reclaiming all resources associated with it |
| `thread_property/2` | Id,status(true) | Gives the id of a thread with status true (used in combination with `thread_join/2`) |

*Table A2.1 Multithreading Predicates in use for the current project*

# References

Ancona, D., Mascardi, V., 2003. Coo-BDI: Extending the BDI model with cooperativity, in: International Workshop on Declarative Agent Languages and Technologies. Springer, pp. 109–134.

Austin, J.L., Urmson, J.O., Sbisà, M., 1975. How to do things with words. Harvard University Press, Cambridge, Mass.

Becker, R., Carlin, A., Lesser, V., Zilberstein, S., 2009. Analyzing myopic approaches for multi-agent communication. Computational Intelligence 25, 31–50.

Behrens, T., Köster, M., Schlesinger, F., Dix, J., Hübner, J.F., 2011. The multi-agent programming contest 2011: A résumé, in: International Workshop on Programming Multi-Agent Systems. Springer, pp. 155–172.

Bellifemine, F., Poggi, A., Rimassa, G., 1999. JADE–A FIPA-compliant agent framework, in: Proceedings of PAAM. London, p. 33.

Bordini, R.H., Braubach, L., Dastani, M., Seghrouchni, A.E.F., Gomez-Sanz, J.J., Leite, J., O'Hare, G., Pokahr, A., Ricci, A., 2006. A survey of programming languages and platforms for multi-agent systems. Informatica 30.

Bordini, R.H., Hübner, J.F., 2005. BDI agent programming in AgentSpeak using Jason, in: International Workshop on Computational Logic in Multi-Agent Systems. Springer, pp. 143–164.

Bratman, M., 1987. Intention, plans, and practical reason. Harvard University Press Cambridge, MA.

d'Inverno, M., Kinny, D., Luck, M., Wooldridge, M., 1997. A formal specification of dMARS, in: International Workshop on Agent Theories, Architectures, and Languages. Springer, pp. 155–176.

Dastani, M., van Riemsdijk, M.B., Dignum, F., Meyer, J.-J.C., 2003. A programming language for cognitive agents goal directed 3APL, in: International Workshop on Programming Multi-Agent Systems. Springer, pp. 111–130.

Finin, T., Fritzson, R., McKay, D., McEntire, R., 1994. KQML as an agent communication language, in: Proceedings of the Third International Conference on Information and Knowledge Management. ACM, pp. 456–463.

Franklin, S., Graesser, A., 1996. Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents, in: International Workshop on Agent Theories, Architectures, and Languages. Springer, pp. 21–35.

Georgeff, M., Pell, B., Pollack, M., Tambe, M., Wooldridge, M., 1998. The belief-desire-intention model of agency, in: International Workshop on Agent Theories, Architectures, and Languages. Springer, pp. 1–10.

Georgeff, M.P., Lansky, A.L., 1986. Procedural knowledge. Proceedings of the IEEE 74, 1383–1398.

Guerra-Hernández, A., El Fallah-Seghrouchni, A., Soldano, H., 2004. Learning in BDI multi-agent systems, in: International Workshop on Computational Logic in Multi-Agent Systems. Springer, pp. 218–233.

Hindriks, K.V., De Boer, F.S., Van Der Hoek, W., Meyer, J.-J.C., 2000. Agent programming with declarative goals, in: International Workshop on Agent Theories, Architectures, and Languages. Springer, pp. 228–243.

Hindriks, K.V., De Boer, F.S., Van der Hoek, W., Meyer, J.-J.C., 1999. Agent programming in 3APL. Autonomous Agents and Multi-Agent Systems 2, 357–401.

Hübner, J.F., Bordini, R.H., Wooldridge, M., 2006. Programming declarative goals using plan patterns, in: International Workshop on Declarative Agent Languages and Technologies. Springer, pp. 123–140.

Ingrand, F.F., Georgeff, M.P., Rao, A.S., 1992. An architecture for real-time reasoning and system control. IEEE expert 7, 34–44.

Mascardi, V., Demergasso, D., Ancona, D., 2005. Languages for Programming BDI-style Agents: an Overview., in: WOA. pp. 9–15.

Morley, D., Myers, K., 2004. The SPARK agent framework, in: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems-Volume 2. IEEE Computer Society, pp. 714–721.

O'Brien, P.D., Nicol, R.C., 1998. FIPA—towards a standard for software agents. BT Technology Journal 16, 51–59.

Pokahr, A., Braubach, L., Lamersdorf, W., 2005. Jadex: A BDI reasoning engine, in: Multi-Agent Programming. Springer, pp. 149–174.

Rao, A.S., 1996. AgentSpeak (L): BDI agents speak out in a logical computable language, in: European Workshop on Modelling Autonomous Agents in a Multi-Agent World. Springer, pp. 42–55.

Rao, A.S., Georgeff, M.P., 1995. BDI agents: from theory to practice., in: ICMAS. pp. 312–319.

Rao, A.S., Georgeff, M.P., 1991. Modeling rational agents within a BDI-architecture. KR 91, 473–484.

Robinson, P.J., Clark, K.L., 2010. Pedro: A publish/subscribe server using Prolog technology. Software: Practice and Experience 40, 313–329.

Russell, S.J., Norvig, P., 2016. Artificial intelligence: a modern approach. Malaysia; Pearson Education Limited,.

Sardina, S., Padgham, L., 2011. A BDI agent programming language with failure handling, declarative goals, and planning. Autonomous Agents and Multi-Agent Systems 23, 18–70.

Searle, J.R., 1980. Minds, brains, and programs. Behavioral and brain sciences 3, 417–424.

Werner, E., 1989. Cooperating agents: A unified theory of communication and social structure, in: Distributed Artificial Intelligence. Elsevier, pp. 3–36.

Wooldridge, M., 2003. Reasoning about rational agents. MIT press.

Wooldridge, M., Jennings, N.R., 1995. Intelligent agents: Theory and practice. The knowledge engineering review 10, 115–152.

Wooldridge, M., Jennings, N.R., 1994. Agent theories, architectures, and languages: a survey, in: International Workshop on Agent Theories, Architectures, and Languages. Springer, pp. 1–39.