ALMA MATER STUDIORUM – UNIVERSITY OF BOLOGNA
CAMPUS OF CESENA

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
Second Cycle Degree in Computer Science and Engineering

A MULTI LEVEL EXPLAINABILITY FRAMEWORK FOR BDI MULTI AGENT SYSTEMS

Thesis in
PERVASIVE COMPUTING

Supervisor
Prof. ALESSANDRO RICCI

Co-supervisor
Prof. JOMI FRED HÜBNER
Dott. SAMUELE BURATTINI

Presented by
ELENA YAN

Academic Year 2022 – 2023

KEYWORDS

Agent-oriented software engineering
Multi-Agent Systems
Debugging agent program
Explainability
BDI agents
JaCaMo framework

"The only way to do great work is to love what you do."

Contents

List of Figures

List of Tables

Listings

Introduction

As software systems grow in size and complexity, it becomes difficult to program them but also to understand their behaviour and decisions. [32] At all stages of software development, a thorough understanding of the system is required. From the analysis and design phase, where capturing requirements and using a common language for designers and domain experts. To the *debugging* phase, in which developers identify and resolve system behaviour issues, to the *testing* and *validation* phase to check if the system meets its requirements.

This is even more of an issue when we consider those systems such as agent and multi-agent systems, which are able to behave autonomously to adapt to unpredictable external changes. At the same time, they are very complex systems with a high level of abstraction, bringing in some architectural concepts inspired by human reasoning. However, explaining, debugging, testing, and validating the behaviour of these systems is still a challenge [20, 40]. These tasks can be *time*- and *resource*-consuming if tools and methodologies are insufficient.

A common and valuable approach in the field of Explainable AI to increase efficiency in these phases is the use of **explanations**. [34, 1] In this context, the thesis introduces a *multi-level* explainability framework specifically for **Belief-Desire-Intention (BDI) multi-agent systems** to contribute to the methodologies of agent-oriented software engineering. The BDI model is one of the most well-known models of intelligent and cognitive agents, which incorporates an internal mental state and simulates the way humans use reasoning to pursue their goals. [14]

However, the internal complexity of the decision-making process of BDI agents poses challenges both in terms of the software development cycle and end-user reliability. The primary contribution of this thesis is the exploration and proposal of a comprehensive framework that constructs automated explanations to address several challenges in a *BDI multi-agent system*. This framework is designed to provide a *multi-level explanation* that serves *multiple purposes* and accommodates different user groups.

The thesis is structured as follows.

- Chapter 1 provides an overview of the context of the thesis and defines the project proposal, its motivations and approaches.
- Chapter 2 presents the concepts behind the research. Three main topics are discussed: the concept of agent and multi-agent systems, the definition of explainability and the state of the art in different areas, and debugging, testing, and validation in BDI systems. The concepts of agent programming language Jason are then presented.
- Chapter 4 described the design process of the framework and the identification of a multi-level explanation.
- Chapter 5 deals with the prototyping implementation of the tool for the multi-level explanability framework.
- Chapter 6 describes the evaluation phase of the system. There is presented a case study showing how the implemented tool can be useful in software engineering.
- Chapter 7 presents the final considerations and possible future developments.

Chapter 1

Motivations and Approach

1.1 Motivations

In this thesis, we focus mainly on agent and multi-agent systems, since with their high level of abstraction, they exhibit a high degree of complexity. Agents, by their nature, exhibit intelligent and autonomous behaviour by encapsulating their own decision-making process. They can therefore make autonomous and intelligent decisions reacting to external changes and can take independent actions based on their internal state and goals. [63]

Due to their complexity, high abstraction, and autonomy, many times understanding system behaviour becomes difficult without appropriate and valid supporting tools. Understanding the rationale behind their decision is important and beneficial at various stages of software development. From the analysis and design phase, the capture of requirements and the use of a common language and knowledge (particularly with the integration of domain-driven design [58]), stakeholders and domain experts require familiarity with the system. [17] During the *debugging* phase, developers identify and resolve issues in the system's behaviour. When dealing with complex systems, the ability to explain *why* some behaviour does not work as expected or fails becomes essential in order to diagnose and correct problems effectively. [60] In the *testing* phase, the system is subjected to various scenarios and inputs to evaluate its performance and correctness. [3, 55]

The importance of understanding system behaviour is underlined by the growing importance of explainable AI (XAI). XAI has gained prominence alongside the success of deep learning systems and the increasing adoption of AI in various commercial applications. [64] This trend underlines the need for systems that can be easily understood by non-technical users. The key idea is to provide this level of understanding through (usually text-based) explanations that can better convey the decision-making processes and enable people

to validate the motivation of choices taken by the system.

Motivated by the various use cases of the explanation and the different classes of target users, it is necessary to deal with different levels of abstraction in the generated explanation since they target specific classes of users with different requirements and objectives. This thesis introduces the idea of *multi-level explainability* as a way to generate different explanations for the same systems at different levels of detail. We introduce in the following sections the proposal and approaches adopted for the design of this explainability framework.

1.2 A Multi-Level Explainability Framework

The main objective of this thesis is to propose tools that facilitate the analysis and exploration of the behaviour of multi-agent systems, offering explanations from a multi-level perspective. The *explanations* aim to be presented in a narrative or storytelling form that can describe, reveal, or justify different aspects such as *what* are the chosen decisions, *why* something happens, *how* the agent arrives at certain decisions, and more. The basic concept of explanation in this work is based on a *logging approach* that captures events and actions, allowing us to reconstruct the history of events and present it to the user in a comprehensible way.

The proposed framework aims to address these challenges by providing **multi-level explanations** to contribute to agent-oriented software engineering methodologies. At a low level, the explanation algorithm provides a detailed understanding in terms of system code and internal agent reasoning. This more technical level can assist developers in debugging and testing the system. At the high level, the explanation will focus on providing in a user-friendly and easy-to-understand manner without delving into intricate code details. This level is most useful for end users and domain experts who may not have in-depth knowledge of agent programming.

This adaptable tool is designed to meet the unique needs and perspectives of different classes of users.

- **For users**, explanations should be presented at an appropriate level of abstraction that allows users to understand the domain of the system without overly technical system detail.
- **For designers and stakeholders**, to facilitate the process of acquiring models and understanding the system. It allows stakeholders to explore the system and formulate new requirements for effective collaboration and alignment.

- **For developers**, including technical experts and engineers, who need more detailed explanations to gain a complete view of the inner workings of the system to support the debugging and testing processes.

1.2.1 Benefits of the Multi-Level Explainability Framework

Our objective is to create a versatile explainability framework that accommodates a wide range of users, offering multi-level explanations tailored to their specific needs and perspectives: [5]

- **Debugging and diagnosis.** As agents operate autonomously and make decisions based on complex algorithms, explanations make it possible to analyse their behaviour and inner workings and to understand how and why agents make certain decisions or why something has failed. This could speed up the process of finding an error and the cause of the failure. [34]

- **Validation and testing.** The explanation of behaviour can also contribute to the validation and testing phases of software engineering, as the developer can use the explanation of behaviour to compare it with the requirements of the system and check whether they are fulfilled and correct. [55, 17, 26]

- **Collaborating and communicating with domain experts.** A clear understanding of the system can facilitate collaboration and communication between stakeholders, such as developers, users, and domain experts. By providing a common language and understanding of the behaviour, developers and experts can work together more effectively and reduce the gap between them. [17]

- **Trust, transparency, and user bias.** Explanations help build trust and confidence in an agent system. This is extremely important in cases where a small error can have significant or vital consequences and the ability to explain its reasoning is indispensable (scenarios such as healthcare, autonomous guidance, military, finance, etc.). Especially when the system is complicated, users tend to use it as a "black box" without knowing how the system works or the reasoning that leads to the result. In some cases, systems may be biased or unfair; one aim of explanation is also to make the system more transparent. When users have access to explanations, they can evaluate and validate how the data is used, how the system performs actions, and whether the results are correct. [13, 46]

- **Teaching and learning.** Explanations play a fundamental role in teaching and learning. In educational settings, explanation can be used as a tool to support teachers in the explanation of the principles and reasoning processes of intelligent agents for a better understanding of complex concepts by students.

1.3 Logging Execution Events

The process of recording execution events plays a key role in the development of an automatic explanation tool. In agent programming paradigms, various events within the agent system, such as beliefs, goals, intentions, actions, perceptions, and communication messages, are recorded as observations during run-time. Using the *logging* approach as the basis of an explainability tool allows the record of agent activities and interactions and access to the full program *history* for later analysis and explanation.

In this work, the logging approach adopted is primarily *implicit logging*. In traditional situations, where a lower-level paradigm prevails, *explicit logging*, where developers manually insert logging statements at certain key points, becomes necessary to ensure meaningful observations. For example, in languages such as Java, developers often need to add manual logging statements to gain insight into the execution of the program. [34] In contrast, the agent paradigm introduces a higher level of abstraction that allows the building of implicit logging. The *implicit logging* approach involves automatically capturing and building traces of agent behaviour without requiring explicit developer intervention. This level of abstraction facilitates the tracking of significant events based on their semantic meaning, providing a more abstract but insightful perspective on system behaviour.

Logs recording execution behaviour are stored and then categorised, organised, and ordered in a systematic way to allow programmers to navigate through the entire system history. It also establishes meaningful relationships and connections that give coherence to the observed data.

Operating within a *multi-level* context of explainability, the information recorded has the capacity to address the requirements of all levels. This is due to the fact that the logging mechanism captures information at a low level, including even the smallest details. Depending on the chosen level of explanation, the tool is able to generate abstract explanations at higher levels, building on the detailed log of information. Through the low-level log, we can analyse the whole sequence of events, construct the causality behind the events that occurred, and present a consistent and logical narrative at different levels.

The narrative presented can be more or less closely related to the agent

code and is suitable for both developers and domain experts, allowing them to gain a more or less detailed understanding of the system's behaviour.

1.4 Narrative of the System Behaviour

Presenting explanations in an understandable and **user-friendly** way is a fundamental requirement to ensure the effectiveness of an explanation system. When developing an automated explanation generation mechanism, it is important that the explanations are understandable and useful to users. Usability and user-friendliness are key aspects that influence how well users can understand and benefit from the explanations provided. To achieve this, instead of providing explanations in a traditional technical format or presenting directly with the raw format of the log, the approach is to present explainability at a higher level, in the form of a **narrative**. There are similar works in the literature [53, 55, 54] that present the explanation of behaviour in the form of stories. The stories are created about the agents so that their behaviour can be verified.

Our narrative-style explanation aims to describe how agents behave, covering the whole sequence of events, actions, and decisions that the agent makes. The narrative explains how the agent pursues goals, why certain actions were chosen, and the interactions between the agents and their environment from a multi-agent perspective.

The narrative is expressed in *first person* to make it more expressive, as if the agents themselves were narrating their experiences and thought processes. The explanation is more vivid and accessible, allowing users to enter into the agent's reasoning as if in the first person.

The narrative is particularly effective in making the explanation accessible to a wider audience. Developers, designers, and domain experts who may not be intimately familiar with the technical intricacies of multi-agent systems can easily understand concepts and motivations when presented in a familiar and story-like narrative form.

Chapter 2

Background

The aim of this thesis is to implement tools that support the phases and methodologies of agent-oriented software engineering. As described above, the reference model is intelligent and cognitive agents based on the BDI model. In particular, the reference framework on which the implementation of the explainability tool is built is **JaCaMo** [36]. Before going into the details of development and prototyping, in this chapter we will provide the necessary background, focusing on agent-based systems, agent-oriented programming, explainability, and JaCaMo framework. It also explores debugging, testing, and validating BDI (Belief-Desire-Intention) agents to provide a comprehensive foundation for subsequent chapters.

2.1 Agents and Multi-Agent Systems

Intelligent agents and multi-agent systems encompass a wide range of disciplines, each with unique characteristics that form a distinct and specialised field. [62] Multi-agent systems can be seen as a specialised subclass of *distributed and concurrent systems*. What distinguishes multi-agent systems, however, is the autonomy and self-interest of the individual agents. Unlike classical distributed systems, where entities typically pursue a common goal, intelligent agents in a multi-agent system have the ability to make independent decisions to achieve their own goals and promote their own welfare. These aspects are also closely related to the field of *artificial intelligence* (AI).

In addition to AI, the fields of *economics and game theory* contribute valuable insights to multi-agent systems by describing the interactions between self-interested agents. These concepts are relevant in multi-agent systems because agents often have their own objectives, and their impact on each other is significant. The overall behaviour of agents and their interactions can be distinguished as competitive or cooperative.

Furthermore, multi-agent systems have strong connections with *social science* disciplines, and in particular the study of human behaviour and social structures. In the design of multi-agent systems, there is an organisational and social component that defines groups, roles, and shared missions, which mirrors human society and structure.

The high complexity and abstraction of these agents make the technology highly flexible and ready to tackle real-world challenges. The concepts are now discussed in detail.

2.1.1 The Concept of Agent

An agent refers to an autonomous entity that can perceive and explore the environment and act upon it to achieve specific goals. A definition of the agent is given by Wooldridge in [62] who describes it as follows:

> "An agent is a computer system that is situated in some **environment** and that is capable of **autonomous action** in this environment in order to meet its **design objectives**."

Analysing the keywords in this definition, we can first focus on the term "*environment*." The environment provides the context in which the agent perceives and acts. The environment can take different forms: physical in the case of robots in the real world, or a software environment referring to a virtual domain. Secondly, an agent is capable of "*autonomous action*", which emphasises its ability to make independent decisions and take actions accordingly based on its own internal state and the information it perceives from the environment. As depicted in Figure 2.1, agents receive sensory input from the environment through their sensors and produce actions that affect the environment as output through their effectors or actuators. Agents typically do not possess full control over the environment, but rather have the ability to influence it. This limited control is primarily due to the presence of other agents sharing the same environment. Lastly, the definition mentions "*design objectives*". An agent is designed with specific goals or objectives in mind. These objectives guide the agent's actions and influence its behaviour.

For an agent to be considered *intelligent*, it should possess certain properties that enable it to act flexibly to achieve its design objectives. These properties can be summarised as follows: [62, 63]

- **Autonomy**, agents are capable of making decisions independently without the direct intervention of humans or others. They possess the ability to analyse their environment, evaluate available options, and autonomously choose the most appropriate action to achieve their goals and subgoals.

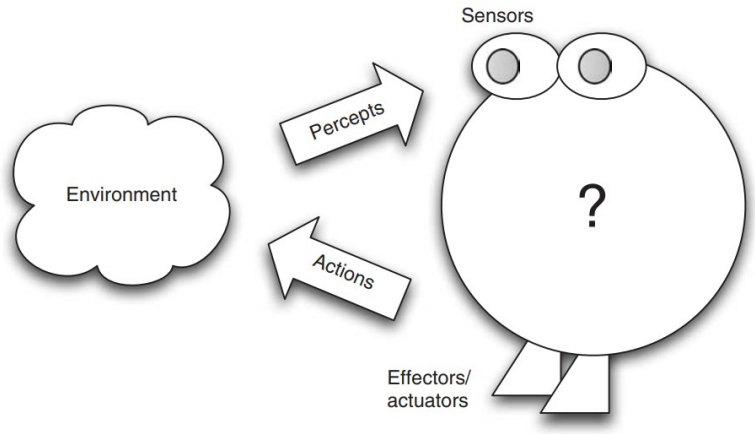

Figure 2.1: Agent and environment [36]

- **Proactiveness**, agents are proactive, they actively engage in actions by taking the initiative to satisfy their design objectives.

- **Reactivity**, agents often operate in dynamic and unpredictable environments, where changes can occur frequently and rapidly. They need to be *responsive* and adaptive to these changes. Reactivity implies that agents can promptly and effectively perceive and react to events in their environment.

- **Social ability**, agents possess social behaviours and the ability to interact with other agents or possibly humans. They have the ability to communicate, cooperate, and coordinate with other entities to achieve common goals. Social ability enables agents to collaborate effectively in multi-agent systems and be involved in complex social interactions.

These properties characterise the notion of "*weak agency*", where agents behave intelligently with "hard-coded goals" and exhibit these essential properties without *conscientious* behaviour.

To build a "*strong agency*", in addition to possessing these properties, they are modelled or implemented using *cognitive* concepts that are typically associated with humans. These include mentalistic notions such as knowledge, belief, intention, and obligation. An example of a model that provides practical support for a strong agency is the BDI (Belief-Desire-Intention) agent model.

2.1.2 The BDI Agent Model

The BDI (*Belief-Desire-Intention*) model is a popular abstraction to represent *strong* and intelligent agents. It is based on a *cognitive* model that focuses on human-like capabilities described in a philosophical model of practical human reasoning [14]. The BDI model conceptualises agents as having their own internal *mental state* which is expressed in terms of beliefs, desires, and intentions as follows: [62, 11]

- **Beliefs** represent all the information the agent believes about the world, including information that the agent senses from the environment and information about itself. This information may be inaccurate.
- **Desires** represents all the goals and objectives that the agent is willing (eventually) to achieve.
- **Intentions** represents the agent's decision to perform a specific action in order to achieve a particular goal. These are the activities that the agent is currently performing, possibly according to a plan.

BDI Abstract Control Loop

The BDI model includes two key processes of practical reasoning: *deliberation* and *means-ends reasoning* [62] described as follows:

- The **deliberation** process refers to deciding *what* goals to pursue from a set of available options. The agent evaluates its beliefs and desires, considers different possibilities, and selects the most appropriate goals to focus on.
- The **means-ends reasoning** process involves *how* the agent can achieve the selected goals. The agent develops a plan that outlines a sequence of actions and then attempts to execute that plan to achieve the goals.

The cycle of a BDI agent is also represented by **sense-plan-act** cycle [43], which in the *sense* phase, agents sense events from the environment through their sensors and receive messages from other agents. In the *plan* phase, the agent processes the information it has collected, updates its beliefs and goals, and then plans new actions to achieve its objectives. Once the agent has generated possible plans, it moves to the *act* phase. Here, the agent selects the most appropriate intention and executes the steps of the plan.

The cycle continues as the agent continuously senses, plans, and acts, adapting its behaviour according to its changing beliefs, desires, and intentions.

BDI Control Loop

The BDI reasoning cycle is quite abstract, as we follow the abstraction idea of the model.

The Listing 2.1 represents the BDI control loop in *pseudocode*, which describes how a cognitive agent thinks and acts based on the Belief-Desire-Intention architecture.

```
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
```

Listing 2.1: BDI Control Loop in *pseudocode*. [52]

At the beginning of each cycle, the process starts with the option generation phase. This phase produces a list of possible options. Next, the deliberation phase selects a subset of options in accordance with the agent's intentions. If at this stage, there is an intention to execute an atomic action at the current moment, the agent proceeds to execute it.

Subsequently, any external events that have occurred during the cycle are added to the event queue. At the same time, any internal events that have occurred are also added. In the final stages, the agent discards successful desires and fulfilled intentions, as well as unrealized desires and intentions.

2.1.3 Multi-Agent Systems

A Multi-Agent System (MAS) refers to a structured ensemble of agents working together to solve increasingly complex tasks within a shared environment. [9] At the individual level, each agent operates autonomously, pursuing its own set of goals and tasks by making independent decisions about what actions to take. However, as an ensemble, these agents typically coordinate and cooperate to achieve the collective goal of the multi-agent system as a unified organisation.

The analogy with the real world is quite straightforward: agents are like people who can observe and perceive something in the environment and then

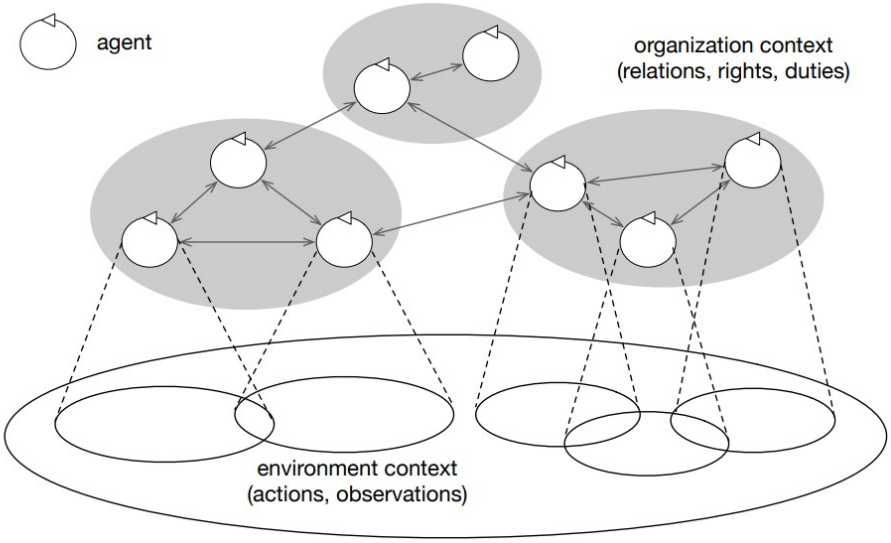

Figure 2.2: A representation of multi-agent systems [8]

decide what to do in order to achieve their own goals and actions. As with humans, the environment can also be seen as a set of resources and tools that agents can share and use to perform their tasks. When it comes to multi-agent systems, where there are multiple agents in the same environment that can communicate, interact, and collaborate, this aspect is also taken from human cooperation.

Above the environment, agents could belong to some organisations, analogous to human societies. The organisational level provides certain specifications that agents must follow and organisational objectives that individuals work together to achieve.

Programming multi-agent systems can be challenging due to the complexity of coordination and interaction among the agents, environment, and organisations. To simplify the development process and maintain consistency from the design phase to the implementation phase, the use of first-class abstractions is crucial.

One structured approach to programming multi-agent systems is called Multi-Agent Oriented Programming (*MAOP*). *MAOP* involves three main dimensions, each of which plays a crucial role in defining the behaviour and interactions within the system: [8]

- **Agent dimension.** This dimension includes the programming of individual agents and concepts for programming agents, including their beliefs, goals, and actions that influence their decision-making.

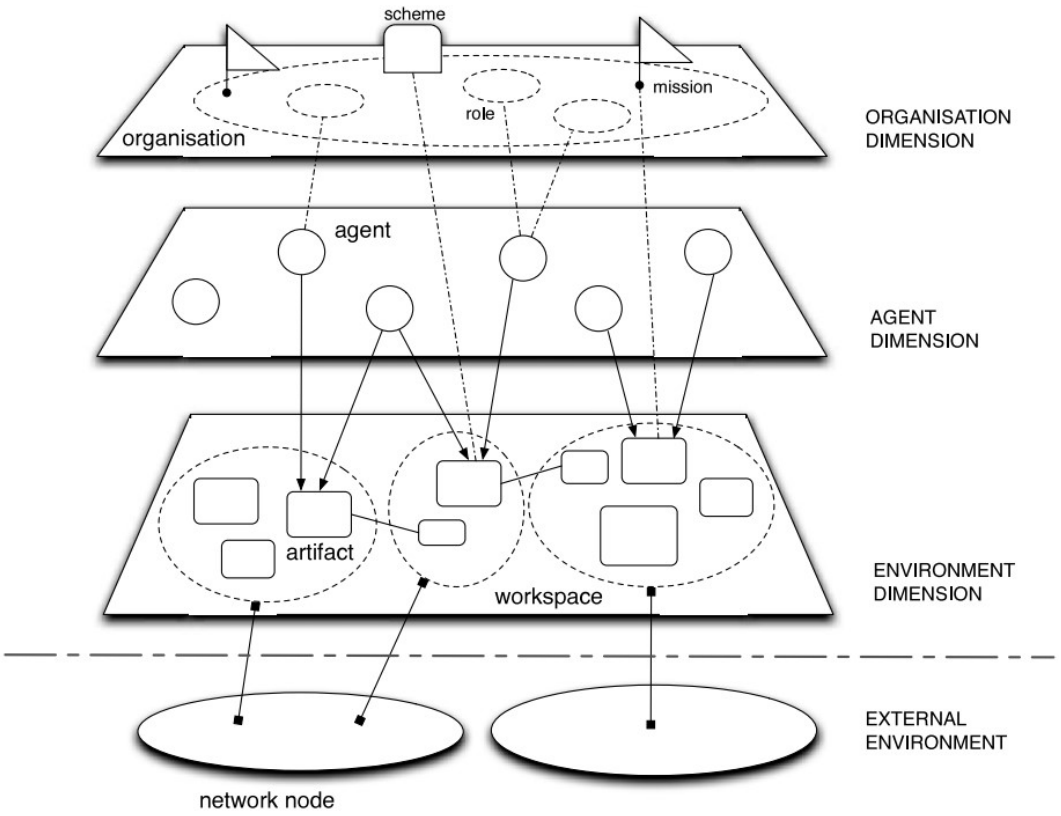

Figure 2.3: Overview of a JaCaMo multi-agent system, highlighting its three dimensions [9, 8]

- **Environment dimension.** This dimension includes concepts for the programming environment, including shared resources and means used by agents to interact, work, and connect with the real world. The environment represents the shared space in which the agents operate, this dimension provides context and influences agent behaviour based on the state of the environment.

- **Organisation dimension.** This dimension focuses on concepts related to how agents are structured, coordination aspects, and regulation of the agents working together to achieve global organisational goals.

2.2 The JaCaMo Framework

JaCaMo is the most notable framework for *MAOP*. The name "JaCaMo" stands for "*Jason*, CArtAgO, and $\mathcal{M}$OISE", which are the three main compo-

nents that constitute the framework. As depicted in the Figure 2.3, each of these components represents a distinct dimension of multi-agent development with its own set of programming abstractions, reference programming model, and meta-model [36, 10].

- ***Jason*** represents the agent dimension. It is a platform for developing cognitive agents following the BDI model with a logic-based language. agents. [38]
- **CArtAgO** represents the environment dimension. It is a framework for programming environment artefacts in multi-agent systems. It is based on the A&A metamodel, where the environment is programmed as a dynamic set of artefacts collected into *workspaces*. [18]
- $\mathcal{M}$**oise** for programming the organisation dimension. It provides support for modelling the organisation, with its structure and management infrastructure, and for organisation-based reasoning mechanisms at the agent level. [47]

2.3 Explainability

In this section, we will look at the definition of explainability, what it means for something to be explainable, its properties, and the different areas of application in research.

2.3.1 Definition

We start by understanding what we mean by *explainability* and what it entails in software engineering. Köhl et al. define that what makes a system explainable is access to *explanations*. [44]

While the Oxford English Dictionary does not have specific definitions for the terms "explainable" and "explainability", it defines "explanation" as follows: [25]

(i) a statement, fact, or situation that tells you why something happened; a reason given for something; and

(ii) a statement or piece of writing that tells you how something works or makes something easier to understand

As defined by the Oxford English Dictionary, explanations are used to make something clear and understandable.

In the context of "intelligent" or "knowledge-based" systems, Gregor and Benbasat (1999)' [31] define explanations as follows:

> "Explanations serve to clarify and make something understandable, or are a declaration of the meaning of words spoken, actions, motives, etc., with a view to adjusting a misunderstanding or reconciling different."

This definition assumes that explanations can be initiated with two main aims: (i) to provide information to clarify, justify, or convince; and (ii) to receive information to resolve misunderstandings or disagreements.

When providing explanations to clarify or justify information, motivations are often used. In fact, Achinstein (1983) [1] emphasises the concept of giving the causes of things and explaining why they happen as fundamental aspects of explanation.

Returning to the requirements for an explainable system, we have identified in the literature several aspects that need to be explained about a system: its reasoning processes, its internal logic, its model internals, its intention, its behaviour, its decision, its performance, and its knowledge about the user or the world. [19]

2.3.2 Explainable AI

Explainable AI (XAI) is a research field focused on developing AI systems that provide transparent and understandable explanations for their behaviour and decision-making processes. XAI aims to address the "black box" nature of many AI models and algorithms, where it can be challenging to understand how and why certain decisions are made.

According to the principles proposed in the European Requirements for Trustworthy AI [21], explainability is a fundamental property of trustworthy AI. Addressing the challenges of explainability and understandability is fundamental to meeting these requirements and ensuring the responsible and ethical use of intelligent agents. These requirements emphasise the need to address the challenges of explainability and understandability as fundamental steps in meeting the criteria for the ethical and responsible deployment of intelligent agents and to be *trustable* by humans. It defines "*explainability*" as the ability to provide clear and understandable explanations of both the technical processes of an AI system and the associated human decisions. Technical explainability enables human users to understand and comprehend the decisions made by the system.

2.3.3 Explainable BDI Agents

BDI (Belief-Desire-Intention) agents are a popular framework for designing intelligent agents that exhibit goal-directed behaviour. Explainable BDI agents

focus on improving the transparency and understandability of their decision-making processes.

The BDI model is advantageous for generating explanations because it relies on concepts that closely resemble those used by humans to explain their actions. [61] These explanations can follow human intuitions, as the agent's behaviour is attributed to the *beliefs*, *desires*, and *intentions* present in the system at a given time. In essence, the BDI-based agent programming approach follows declarative logic programming based on a folk-psychological model of reasoning.

Explainable BDI agents provide explanations for their beliefs, desires, and intentions, helping users understand why certain actions were chosen or decisions were made. Here is how this process generally unfolds:

- *context*, the agent operates within a specific context, which is shaped by a set of beliefs held by the system, its current goals, and potential events from the environment;
- *plan selection*, given this context, the agent identifies a subset of plans from its repository, selected on the basis of their potential to achieve the current goals;
- *intentions*, the instantiation of these selected plans represents the intentions of the system, they outline the specific courses of action that the agent intends to pursue in order to achieve its goals.

By making the agent's internal state and reasoning more accessible, these explanations facilitate collaboration, user trust, and effective use of the system. [15, 33] Techniques used in the development of explainable BDI agents include argumentation-based explanations, goal-based explanations, and plan-based explanations. These techniques allow users to understand the agent's decision-making process, what the agent's beliefs and desires are, and how the agent evaluates the choice of action to be performed.

In the work presented in [24], the authors introduce an approach to explaining the BDI-based agents through dialogues. They propose a formalised explanatory dialogue that helps identify and explain why a BDI system acts in a certain way. The paper focuses on a scenario where two participants have different views about a BDI program, where one is the program executor with the correct trace and the other is a human or system trying to understand the executor's behaviour. Identifying a disagreement serves as an explanation to highlight errors in the latter's assumptions or reasoning. These disagreements could involve variations in plans, priorities, inputs, or initial beliefs.

The work proposed in the thesis operates in a similar way. But instead of offering a dialogue, our idea is to generate a narrative of the story of the agents'

behaviour. The user can navigate through this narrative and understand all the reasons that led to a certain decision. The narrative focuses on the agent's beliefs, desires, and intentions in order to provide a reasonable explanation of the decisions and actions made and the reasons for that choice.

2.4 BDI Agent Debugging, Testing and Validation

In the software engineering process, debugging, testing, and validation are crucial parts of the software development cycle. Debugging, testing, and validating agents and multi-agent systems is not easy because of their non-deterministic behaviour and the need to manage the environment and organisation, which could be dynamic and complex.

However, given the nature of BDI agents, characterised by their ability to reason based on observations and objectives, it is possible to implement mechanisms that support the software engineering process.

2.4.1 Debugging in BDI Agents

Debugging, in the context of software development, is a critical process aimed at identifying, locating, and correcting defects or unexpected behaviour in a computer program. [35] It involves a systematic and often iterative approach to finding and solving problems. Without a good debugging tool, it is difficult and time-consuming for developers to determine the cause of the error and fix it. Various approaches to debugging agent-oriented systems have been studied in the literature. Some of these are discussed in the following paragraph.

Why? Questions One of the approaches first proposed for the Alice framework, and then in Java, and AgentSpeak, is to use the *Whyline* approach [41] with a series of 'Why? Questions', also known as *question-based debugging*. The idea is to allow programmers to ask, at each step of the execution trace, "why did this happen?" or "why did <something else> *not* happen?" [60]. Programmers can then identify the cause of the error step by step.

However, one limitation of this approach is that it is not user-friendly. The debugging process requires the user to ask a series of questions and understand each step of the answer in order to identify the cause of the error.

Algorithmic Debugging The idea of algorithmic debugging is to analyse the behaviour of the program and compare it with the expected behaviour. In

general, the approach creates an internal representation of the system and, by asking the user some questions or using a formal specification, identifies the location of the bug in the source code.

The algorithmic debugging approach presented in [2] for BDI agents builds a *debugging tree* of plans to describe the structure of the system. The idea is to check the plan of an unachieved goal or sub-goal, assuming that if a goal is achieved, then there are no bugs in that goal or its sub-goal. The program presents the tree to the user, and at each goal and sub-goal, asks the developer to check whether the goal has been achieved or not.

This form of debugging represents the behaviour of the system in terms of high-level concepts of goals and plans through the tree view. At the same time, debugging requires the user to have knowledge of the system in order to identify the cause of errors. Another limitation is that it may not be able to identify all types of errors, in fact, some errors may be caused by external factors such as changes in the environment.

Omniscient Debugging Omniscient debugging is another approach to debugging that differs from the traditional approach. It consists of recording the entire trace of the program and then allowing the user to navigate the history both forward and backward without re-executing the program [49]. This is useful in multi-agent systems because, due to their non-deterministic nature, the program will not always have the same output when the same program is run.

The Omniscient debugging introduced in [42] for the GOAL programming language records the program in a log. Due to the huge amount of information that is generated, it is not possible to record all states of the system, so in the presented work only the changed state of the agents for each cycle is stored. It has also integrated a mechanism for associating the location of the source code to facilitate the debugging process. Therefore, it constructs the trace of the run and presents an interface for visualising the traces through a space-time view with the possibility of sorting and filtering the trace.

Tracing Tool The Tracing Tool proposed in [12] for analysing intelligent agent systems, logs observations about agent behaviour and provides explanations in the form of relational graphs. It uses the agent's concepts, such as beliefs, goals, intentions, actions, events, messages, and relations between them, to create a concept graph for comprehending agent software. Developers can explore the graph, understand the systems, and explain why an agent performed some unexpected behaviour. [45]

Design Artefacts Another approach for debugging multi-agent systems is to look at the design artefacts. Diagrams such as class, sequence, or interaction diagrams support developers in comprehending the system's behaviour. The methodology proposed in [50] consists of analysing the interaction between agents by building a Petri net graph. It builds a Petri Net for monitoring the interaction between agents.

2.4.2 Testing and Validation BDI Agents

In the domain of the software development life cycle, testing is an important phase to ensure that the system behaves as expected. The correctness and robustness of these agents need to be assessed to ensure their effectiveness and reliability. This section presents some methodologies for testing BDI agents.

The work presented by Rodriguez et al. in [53, 55, 54], uses the concept of a *user story*, as a description of the system from the user's perspective. In [53] they extend this concept to *system stories* where, in this case, the perspective turns to the system. They present an agile approach to capturing system requirements, for each requirement, they identify a classical user story, map it to a system stories and define its acceptance criteria. During the development phase, they map the system stories to agent concepts to verify that all expected behaviours are met and for system traceability.

In the following research [55], they present an approach to testing whether the requirements specified in user and system stories are fulfilled. To do this, they developed an approach that validates whether the acceptance criteria are met by comparing them with the trace of the agent's execution. They also developed a fault model to classify the cause of failure.

The work presented in [54] introduces a novel approach to testing the behavioural requirements of agent systems in a Behaviour Driven Development (BDD) and Test Driven Design approach, in line with agile software development practices. They transition from user stories to *system and agent stories*, which represent the requirements from the agent's perspective. They extend the user and system story to capture *goals*, *plans*, *perceptions* and *beliefs*, with their own story with the aim of verifying their behaviour with the requirements.

In these very related works, they exploit user/system-stories to arrive at agent-related stories. The story they produce is very similar to our concept of storytelling which we would like to develop in our thesis research. The developed approach contributes to testing by applying the BDD approach. Another research in this context, presented in [17], introduced an agile testing methodology for multi-agent systems based on BDD, called BEAST Methodology, and a supporting tool BEAST Tool. The aim is to automatically generate *JUnit* test case skeletons from the specifications of BDD scenarios.

Another related work in [26] presents a *goal-oriented* approach, which considers the goal as the smallest unit in the multi-agent system to check the correctness of an agent. They proposed a new concept called 'test goal' to implement automatic tests for a single goal.

Regarding the testing, the work presented by Cleber et al. in [4] facilitates the testing and continuous integration in JaCaMo multi-agent systems. The research focuses on an approach for developers to write tests for Jason agents, from unit testing, and single-agent testing, to multi-agent testing with JaCaMo. In order to write tests for the agent in Jason, the proposed method involves the creation of plans for testing the agent's behaviour and then using assertions to verify its correctness. Additionally, the approach allows for the inclusion of multi-agents in the test file for the verification of functionality in a multi-agent context.

Chapter 3

Reference Technology

The study conducted in this thesis focuses on the development of a multi-level explainability tool for multi-agent BDI systems. In particular, the technology taken as a reference is the *JaCaMo* framework and we focus specifically on the agent programming language ***Jason***.

Since in this thesis, it was important to analyse the internal operation of ***Jason*** in order to extend it to include the logging part of the log track, in this chapter we present its general architecture and analysis of the operational semantics.

3.1 *Jason*

Jason is a Java-based interpreter for the extended version of AgentSpeak, proposed by Jomi F. Hübner, Rafael H. Bordini and colleagues in 2007 [38]. AgentSpeak(L), a theoretical foundation behind Jason, is a high-level programming language based on logic. It is specifically designed to model the mental attitudes and behaviours of agents within a Belief-Desire-Intention (BDI) architecture. This architecture, as stated before, is built upon the concepts of beliefs, desires, and intentions and serves as a cognitive framework through which agents' reason, decide, and interact with their environment.

An agent is represented by an entity composed of a set of *beliefs*, which encapsulate the agent's current state and knowledge about the environment and form the agent's belief base; a set of *goals*, which correspond to the objectives of the agent; and a set of *plans*, which define the ways to achieve goals.

In addition to interpreting the original AgentSpeak language, the ***Jason*** framework offers various powerful features that elevate its capabilities in the development of cognitive agents and multi-agent systems. Some of these notable features include [11, 38]:

- **Extensive language extensions.** The language extensions in ***Jason*** encompass meta events, declarative goal annotations, higher-order variables, treating plans as terms, imperative-style commands in plan bodies, and various other additions. These extensions give developers the flexibility to adapt agent behaviour to specific application requirements.
- **Agent communication.** While the BDI model is based on the philosophical literature on practical reasoning, the AgentSpeak language defines the internal structure of decision-making and the creation of goals or sub-goals and the execution of plans to achieve those goals or sub-goals. Agent communication in a multi-agent system is based on speech act theory. [6, 56] A speech act is also an action, with the difference that the domain of a speech act is typically limited to the mental state(s) of the receiver(s). A speech act could change the beliefs, desires, or intentions of the agent.
- **Handling of plan failure.** Just as human-like is full of infeasible plans that require the discovery of alternatives, in an ever-changing environment, not all plans in ***Jason*** are guaranteed success. The framework supports agents in dealing with failed plans by adapting and looking at other plans to fulfil their intentions.
- **Support for MAS.** ***Jason*** can be fully integrated with CArtAgO for developing environments and $\mathcal{M}$OISE for developing organisations.
- **Support for distributed execution.** ***Jason*** facilitates distributed execution of the multi-agent system over a network with the integration of SACI, JADE or other distribution infrastructures.

3.1.1 *Jason* Semantics

The ***Jason*** agent programming language is an extension of AgentSpeak, which is based on the BDI architecture. An agent's architecture in ***Jason*** consists of several important components that define its behaviour, categorised into beliefs, goals, and plans. The *belief base* is a critical component where an agent stores its knowledge, the agent's *goals* represent the objectives that the agent wants to achieve and *plans* outline the path for achieving the agent's goals.

The ***Jason*** programming language is rooted in logic-based principles, and its syntax is inspired by Prolog. Let's examine an example of the "hello world" program, which is presented in Listing 3.1.

In this illustrative example, we show the basic components of the BDI architecture and how they are represented in ***Jason***. The agent has a belief

```
happy(bob).                    // B
!say(hello).                   // D

+!say(X) : happy(bob)          // I
    <- .print(X).
```

Listing 3.1: BDI and ***Jason*** Hello World example [27]

that indicates a property, specifically the notion of being `happy` is attributed to `bob`. A desire or goal is represented with the construct `!say(hello)`. In this case, this goal serves as the agent's initial goal, indicating to say hello. A goal becomes an intention when it matches a plan. The example shows that if the agent expresses a desire to `say` something (denoted as `X`) and `bob` is `happy`, then the agent will commit to executing the plan. This represents an intention that the agent has decided to work towards and is committed to. The action of this plan is to print the value of `X`.

In the following sections, we will look at these different components and the language constructs in more detail.

Beliefs

The first component to explore is the agent's beliefs. Beliefs represent the agent's understanding and perception of the world. In the ***Jason*** framework, beliefs are managed through the agent's *belief base*, a repository that contains the agent's current set of beliefs. Beliefs are represented by literals:

```
functor(term_1, ..., term_n)[annot_1, ..., annot_m]
```

Beliefs have significant semantic meaning and are accompanied by *annotations* providing detailed information about a belief. One important annotation is the *source annotation*, which is used to indicate the source of the information from which the belief comes. Specifically, there are three different types of information sources for agents:

- **perceptual information** refers to the property of the environment (called a percept) that results from the agent's sensing of its environment and contributes to the formation of the agent's beliefs;
- **communication:** agent communication with other agents, this source includes the name of the agent who sent a communication message;
- **mental notes:** represents information that an agent records for its own future reference,used to facilitate the execution of plans at a later stage.

Within the agent's belief base, beliefs are stored together with their annotations. When a new belief is added, the process involves checking whether the belief already exists in the base. If it is not present, the belief is added. However, if a belief with identical content but a different source annotation already exists, only the new source information is appended to the existing belief.

Similarly, the procedure for removing a belief from a specific source follows a similar logic. If multiple sources within the belief base share the same belief, removing a source simply involves removing the corresponding source information from the belief. The agent continues to hold the belief based on other sources. The belief is only truly removed when no source remains associated with it.

Goals

Goals encapsulate the desired properties of the states of the world that the agent wants to achieve. Note that in ***Jason*** the term *goal* is used to refer to the agent's desire in the BDI definition. There are two types of goals:

- **achievement goal:** represents the objective, the state of the world that the agent wants to achieve. In this case, it is denoted by the "`!`" operator.

- **test goal:** is used to check and retrieve the information contained in a specific belief of the agent. It is denoted by the "`?`" operator.

We can define *initial goals* to indicate that this is a goal to be achieved with the exclamation mark:

```
!initial_goal.
```

The adoption of a new goal by an agent leads to the execution of *plans*, a sequence of actions that the agent will commit to execute in order to achieve that goal.

Plans

A plan is a foundational construct used to define a sequence of actions that an agent should execute in order to achieve a particular goal. A plan is composed of three distinct parts:

```
triggering_event : context <- body
```

Triggering Event Events represent changes, in terms of addition or deletion, in beliefs and goals. These allow us to express six different types of triggering events:

1. Belief addition
2. Belief deletion
3. Achievement-goal addition
4. Achievement-goal deletion
5. Test-goal addition
6. Test-goal deletion

Context Context plays a fundamental role in determining the *applicability* of a plan. When an agent is trying to achieve a goal, he may have different ways of doing so. The context of a plan is used to check the current situation, and the agent tries to determine which plan has the best chance of success by considering the current set of beliefs of the agent.

Body The body of a plan encapsulates the course of actions or steps that the agent will execute if the conditions of the plan are met and the plan is selected. The conditions are: (i) the event that has occurred matches the triggering event of the plan, and (ii) the context of the plan is true according to the agent's beliefs.

An action is represented by a formulæ, which can include various elements, such as:

- **actions**, operations that the agent can perform in order to reach its goals;
- **achievement goals**, another goal to achieve, which will create another goal to achieve (and then involve other plans) before continuing with the next actions;
- **test goals**, used to retrieve specific information from the agent's beliefs;
- **mental notes**, to create new information (involving the creation of a new belief) that the agent wants to remember for future reference;
- **internal actions**, executing ***Jason***'s standard internal actions;
- **expressions**, evaluating relational expressions or arithmetic expressions.

3.1.2 *Jason* Reasoning Cycle

The ***Jason*** *reasoning cycle* captures the fundamental process through which an agent reasons and interacts with its environment. It involves a continuous loop of perceiving the environment, reasoning about actions to achieve goals, and executing actions that may affect changes in the environment. The ***Jason*** reasoning cycle, depicted in Figure 3.1, consists of 10 main steps, each of which contributes to the agent's decision-making and behaviour. These steps are described below:

Step 1 - Perceiving the Environment. The reasoning cycle starts with perceiving the environment in order to update the agent's beliefs. This task is executed by the `perceive` method in the ***Jason*** implementation. By invoking this method, the agent retrieves a list of *percepts*, symbolically represented as literals. This step captures everything currently perceptible in the environment and can include both newly perceived information and previously acquired percepts.

Step 2 - Updating the Belief Base. After perception, the agent updates its belief base based on the perceptions it has acquired from the environment. This process, executed via the *belief update function* (`buf` method), involves adding new percepts to the belief base that were not previously present and removing percepts that are no longer observable. With the execution of this function, each change in the belief base generates an *event* characterised as an *external event*.

Step 3 - Receiving Communication from Other Agents. In the context of a multi-agent system, communication with other agents is fundamental. In this step, the agent receives messages from other agents. The `checkMail` method for this step checks the agent's 'mailbox' for incoming messages. The process then continues with a *message selection function*, which is responsible for prioritising and selecting messages for further processing.

Step 4 - Selecting 'Socially Acceptable' Messages. Before the messages are processed, they pass through the *social acceptance function* (`SocAcc method`). This function can be customised and determines whether incoming messages can be accepted by the agent or not. The default implementation accepts all messages from all agents.

Step 5 - Selecting an Event. BDI agents operate by continuously handling *events*. These events represent either perceived changes in the environment or changes in the agent's own goals. Moving forward, this stage

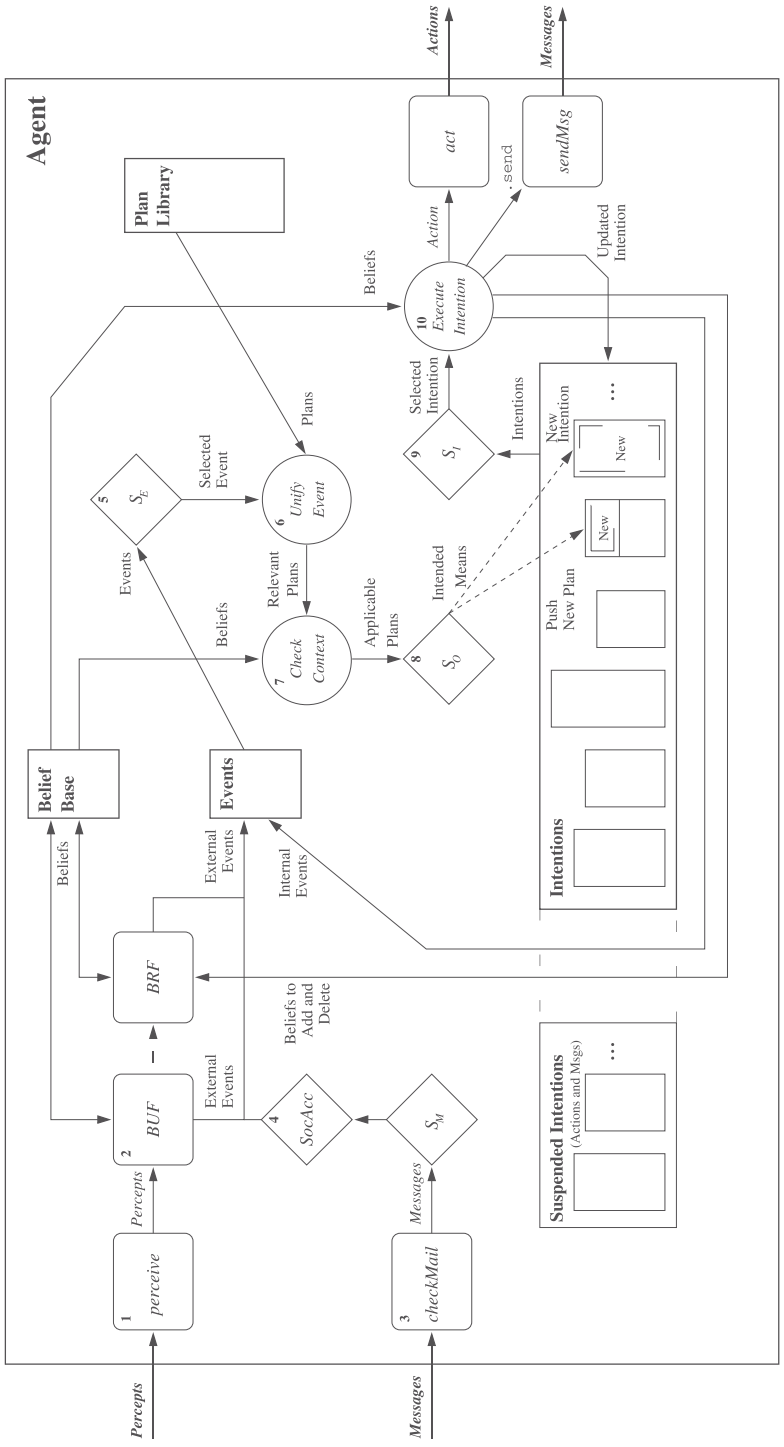

Figure 3.1: The ***Jason*** reasoning cycle [11]

involves the activation of an *event selection function*, responsible for prioritising and selecting events for subsequent processing. The default implementation selects the first event in the list.

Step 6 - Retrieving all Relevant Plans. Once the event has been chosen, the next step is to identify an appropriate plan to deal with the selected event. The process begins with an examination of the agent's *plan library* with the aim of identifying all plans that have a triggering event that can be *unified* with the selected event. This exploration ends with the retrieval of a set of *relevant plans*, ready for the next step.

Step 7 - Determining the Applicable Plans. Although we may have identified plans that are relevant to the chosen event, not all of these plans are immediately suitable for the agent's chosen course of action. Hence, a process of plan selection ensues to determine the plans that are currently *applicable*. The agent aims to select plans that have a probability of success given the scope of the agent's knowledge and current beliefs. To achieve this, the context of each relevant plan is evaluated by checking whether the context of each plan is consistent with its current belief base. This step ends with a list of applicable plans as alternative ways of dealing with the selected event.

Step 8 - Selecting One Applicable Plan. This step involves the selection of one of the alternative plans, which contains a course of action, followed by a commitment to execute the chosen plan. The selection of the plan is done by the *option selection function* (S_O). The commitment implies the agent's *intention* to pursue the prescribed course of action outlined in the selected plan. The chosen plan is called *intended means* and will be included in the agent's *set of intentions*.

Step 9 - Selecting an Intention for Further Execution. With an intended means established through the previous steps, the reasoning cycle moves on to considering which intention to prioritise for further execution. In most scenarios, an agent's set of intentions will contain multiple intentions, each reflecting a different focus of attention. Similar to earlier stages, the choice of which intention to pursue is guided by a selection function. Specifically, this selection function is referred to as the *intention selection function* (SI). This function selects which intention should be prioritised for continued execution, allowing the agent to maintain focus and move forward with purpose.

Step 10 - Executing One Step of an Intention. Based on the previous step of selecting a specific intention, the next step involves the execu-

tion of a single action related to that intention. The first formula in the body of the plan is executed and once completed, the formula is deleted from the body of the plan. Then the updated intention is moved back to the set of intentions.

Chapter 4

A Multi-Level Explainability Framework

A common research path in the literature about explainability and BDI agents is the use of logs and traces as a foundation for understanding agent behaviour in support of agent-oriented software engineering. The key point is that the literature typically focuses on a *single level* of tracing, and for a *single agent*, our research introduces a novel approach by proposing multiple levels of abstraction in logging and explainability generation. The idea of proposing tools for multi-level explanation is motivated by the different contexts of use for different user groups.

This chapter delves into the fundamental concepts behind this innovative framework. It will discuss the main components of the tool, the multi-level concept and the levels covered. Subsequently, it will present the narrative identification phase for the two levels and the mapping between them.

4.1 Main Components of the Explainability Tool

The main challenge of this project is to investigate the behaviour of the agent and identify a narrative for different types of users. Because depending on the user, the explanation could be different. If the user is a developer, the explainability is more related to the code. If it is the end user, we don't pay attention to all the details of the individual agents, it could just be related to the BDI level or something at a higher level.

Figure 4.1 depicts the main components of the multi-level tool that we want to build in this project. Our approach is to include a **logging** component for each agent and system component. This means that we produce a log for each agent in the system, and this is essential because each agent has its own reasoning cycle with its own time to operate on. In addition, when we consid-

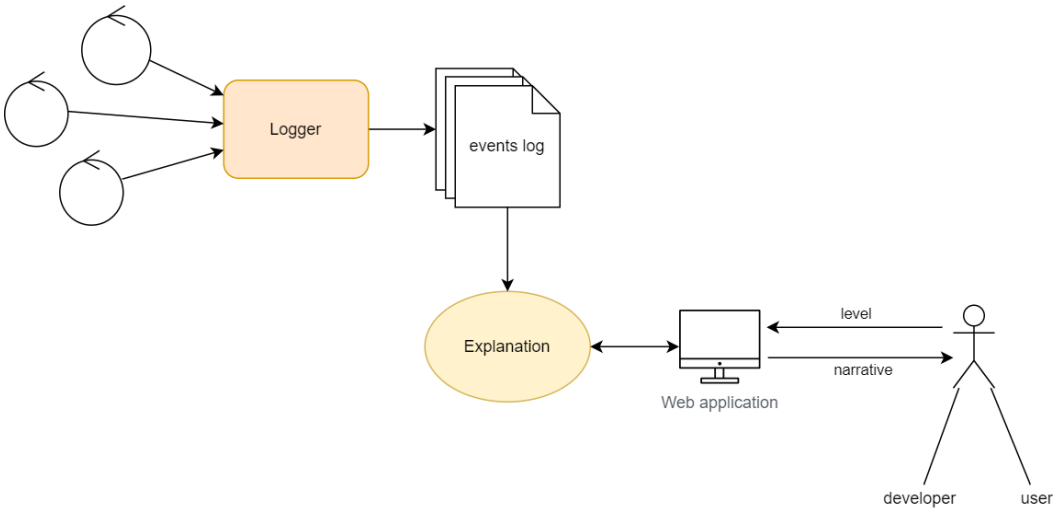

Figure 4.1: Overview of the main components of the framework

ered distributed scenarios, agents could even be executed on separate nodes, further emphasising the need for individual logs. This logging component captures events related to a lower level of the reasoning cycle during execution. These captured events are then stored in *file-based* formats, preserving them for later analysis and processing.

However, logging every detail of every agent and artefact is not feasible due to the enormous number of events generated per second. Therefore, we need to select the most relevant events to log, ensuring their usefulness for explanation purposes. Important event logs are related to reasoning events such as belief updates, goal and plan applicability, agent perceptions, speech-act messages, etc.

A model component then includes the **explanation** algorithm, which reads the log and generates an explanation and narrative of the system's behaviour at multiple levels. This explanation is presented to the user through a *web application*. Different classes of users such as developers, designers, or end users, can access the application and explore the generated narrative at different levels for understanding and analysis.

4.2 Multiple Levels of Abstraction

In the context of understanding BDI agents and their use for testing and validating the system or for other use cases, multiple levels of abstraction are essential. We refer to this approach as "*multi-level*" explainability, as its purpose is to provide insights from the developer's perspective to the user's

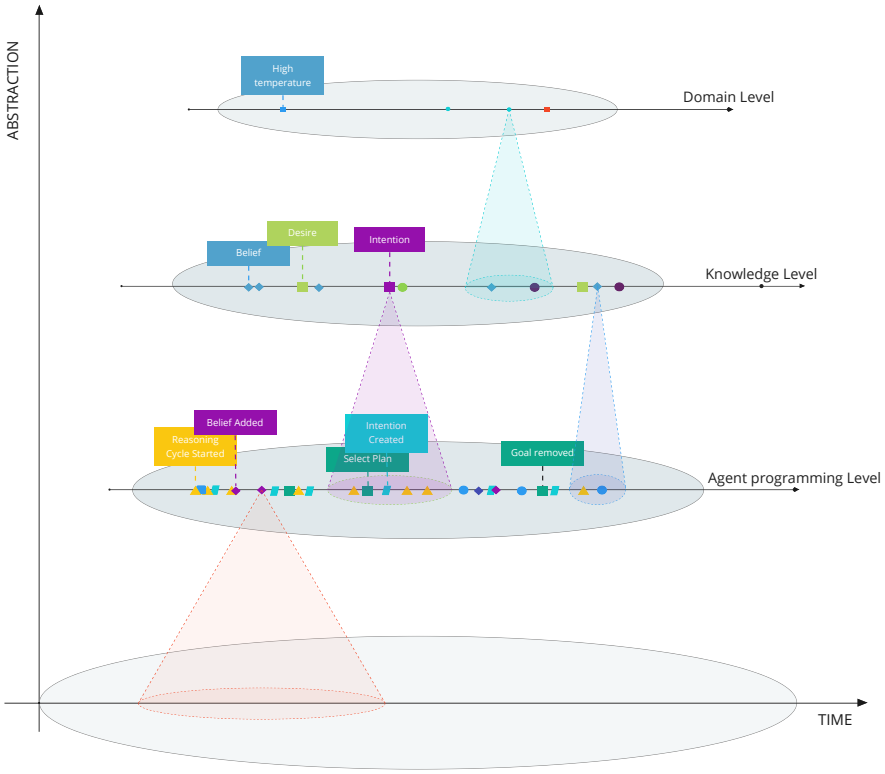

Figure 4.2: Multiple levels of abstraction in the agent dimension

perspective. These levels adapt to different uses and user profiles, providing a comprehensive understanding of agent behaviour.

The concept is illustrated in Figure 4.2, where different levels of abstraction are depicted. This multi-level approach operates at different levels of granularity ranging from a fine-grained, low-level perspective closely related to the underlying code to high levels of abstraction that encapsulate the BDI model or address the user perspective. The primary levels of abstraction identified in this work are:

- ***Jason* level.** At the low level, we have the ***Jason*** level. This level records every event of the agent and observes every detail of the behaviour. It follows the *reasoning cycle* of the system and produces a complete, unfiltered, raw understanding of the system. Developers who want to enter the details of the code and see what and why something happened can navigate through this level and analyse the provided narrative.

- **BDI level.** Moving up a level, we encounter the BDI level. Here the

> focus shifts to the beliefs, desires, and intentions of the conceptual abstraction of the agents. It is like zooming in to see the main behaviour of the system. At this level, we are observing actions from a higher perspective, highlighting the cognitive processes and behaviour of the agents.

These abstraction levels are designed to be flexible, allowing different users to access explanations from different perspectives. Each level of explanation already contains all the information useful for understanding that abstraction. The levels are therefore *self-contained* respect to other levels, which means that the user already has all the explanations, properties, and tools for which we can verify the behaviour at that level with that abstraction. So if, for example, the user is at the ***Jason*** level, thanks to its clear semantic operation, he knows how the agent works and its reasoning cycle and, therefore he can understand how and why events are produced with that abstraction. When he goes to the BDI level, he is not interested in knowing the details of the ***Jason*** level because levels are closed, but more in the cognitive aspects with the BDI abstraction.

Users can simply change the level of abstraction according to their needs. Consider a scenario in which a user, while examining the BDI level, perceives a lack of clarity. In such cases, he can smoothly move to the ***Jason*** level and delve into the complex state of the agent to gain a deeper understanding. Conversely, a developer who examines the ***Jason*** level may find it difficult to understand a complex system due to the large volume of low-level events. In this case, the developer can easily switch to the BDI level, which provides a simplified and abstract overview of the system's behaviour. This dynamic adaptability and simplicity in switching between fine and higher levels ensure that the tool is able to address a wide range of users and scenarios.

The multi-level explainability framework can be extended to a higher level to incorporate the knowledge and domain-specific insights. This vision provides a comprehensive understanding related to the system's high-level behaviour and requirements that involve end-users and domain experts.

Events at higher levels of abstraction are composed of a *set* of events at lower levels. This modular process involves maintaining the most relevant events while presenting them in a more abstract and higher-level way according to the abstraction. This perspective allows users to obtain a higher and more conceptual view of essential aspects of the agent's behaviour without being filled with excessive detail.

The discussion now focuses only on the agent dimension. Nevertheless, it is very useful to extend the tool with other *orthogonal dimensions*, including *environments* and *organization dimensions* [10], as depicted in Figure 4.3. When dealing with multiple agents in a complex system, the concept of narratives

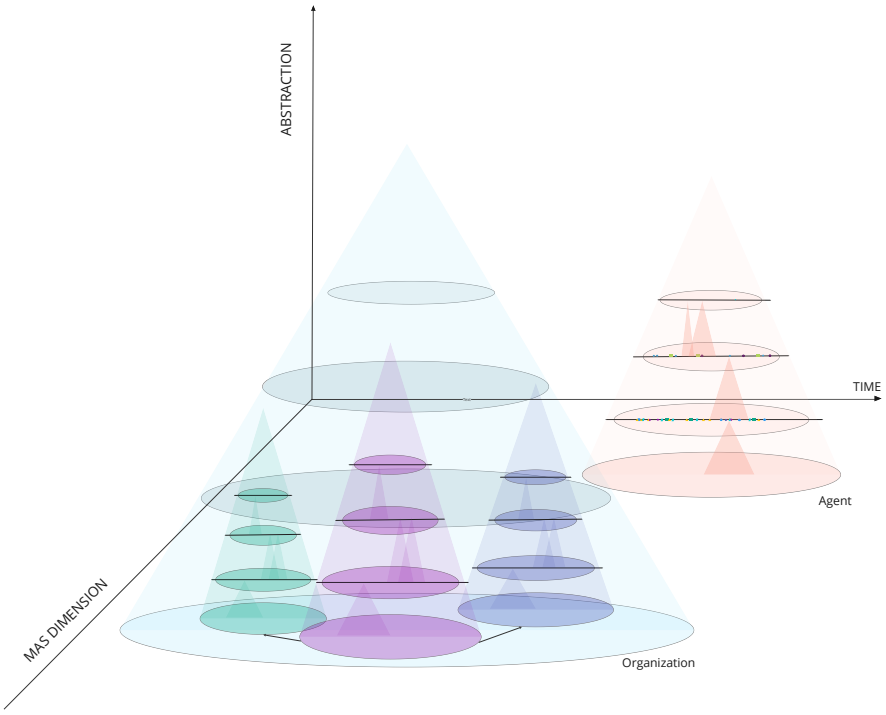

Figure 4.3: Multiple *dimensions* and *levels* of explainability in a multi-agent system

can be extended beyond individual agents to include multiple agents working together with organisational concepts such as norms and social structures.

This multi-level idea is very innovative because by going into a broader vision with the addition of the organisation dimension, the explanation of the organisation can also be extended to several levels. A low level could be the explanation at $\mathcal{M}$OISE level, as it can easily integrate with ***Jason***, while at a higher level, it could provide a knowledge level explanation based for example on missions and norms.

4.3 *Jason* Level

This section will discuss the main events identified at the ***Jason*** level for the building of the narrative. The events identified follow Jason's reasoning cycle and are presented under two macro concepts: belief and goal.

4.3.1 Belief

A belief represents everything the agent knows about the environment and itself. In ***Jason***, beliefs are stored in the agent's belief base. The two main operations related to beliefs are addition and deletion. An important piece of information associated with beliefs is the *source annotation*, which specifies the origin of the belief. An agent may believe the same belief but from different sources. We therefore distinguish the events we have identified related to the belief in:

- **Belief Added:** This event indicates the addition of a *new* belief to the belief base.
- **Belief From Src Added:** This event occurs when the belief is already present in the belief base, and a new source annotation is added. Each origin annotation provides context on the origin of the belief.
- **Belief From Src Removed:** This event signifies the elimination of a particular source from a belief. However, it is important to clarify that the belief itself may still be maintained by the agent through other sources.
- **Belief Removed:** This event indicates the total elimination of a belief from the agent's belief base. It occurs when all the sources of a particular belief have been removed, indicating that the agent no longer holds that belief.

Other events that may cause changes in the belief base include:

- **New Percept:** This event occurs when the agent senses a new perception from the environment, which updates its belief about the current state of the environment.
- **New Speech Act Message:** This event relates to communication between agents. Only when the agent receives a *tell* message from another agent, it is updated in the agent's belief base, becoming a belief of the agent.

4.3.2 Goal

In the ***Jason*** framework, *goals* represent the desired states that an agent aims to achieve. These goals are categorised into different states based on their current status and the agent's commitment to achieving them.

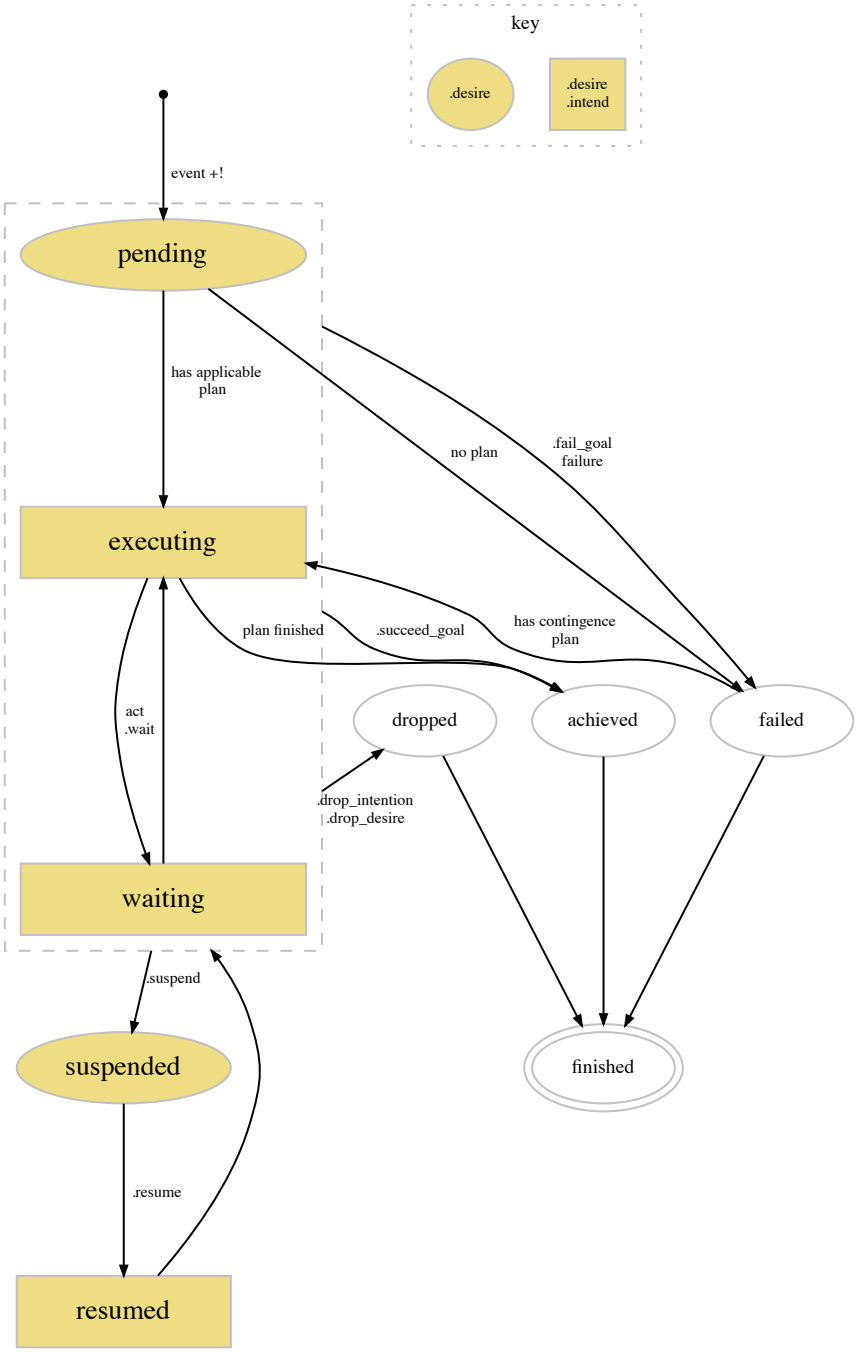

Figure 4.4: Goal states in ***Jason*** [37]

The goal states are depicted in Figure 4.4. Each goal can be in one of these states: pending, executing, waiting, suspended, resumed, or finished. The initial state of a goal is *pending*, denoting that at this moment it is essentially a desire of the agent. If an applicable plan is discovered, the goal becomes an **intention** and the agent is fully committed to pursuing it, the state then moves to *executing*. From this point, it can cause events that put the desire on hold, with the state of *waiting*, and which can then be *resumed* and executed again.

From the states pending, executing, and waiting, the goal may also enter the *suspended* state, indicating that the agent has deliberately suspended its intention to pursue the goal. This is called by an internal action. The goal is no longer intended. The goal, however, may be *resumed* from this state and return to waiting.

Finally a goal, from the executing state, may conclude its lifecycle by moving into one of the three terminal states: *dropped* - by special internal action, *achieved* - successfully completed or *failed* - if there is no plan or by internal action.

By examining this diagram, we can effectively map events related to goals. The log's events are identified based on these states:

- **Goal Created** A goal is created when the goal is added to the event queue, the initial state is pending.

- **Plan Selected** This event occurs when a plan is chosen for a particular goal. It occurs once the goal has been created and the agent selects a plan to fulfil it. This corresponds to the agent's commitment to the goal, moving its state to "executing."

- **Goal Suspended** When a goal is suspended, it transitions back to being a mere desire, reverting from an intention. This event mirrors the suspension of intention.

- **Goal Removed** The event represents the removal of a goal. Goals are removed when they are finished and can assume these possible states: *achieved*, *dropped* or *failed*.

As discussed before with the *goals* state, when the agent commits to the goal, the goal becomes an intention. Therefore, the intention is created and moves to the *running* state. From this state, the intention could move to two other states: *suspended* or *waiting*. These states help us to define the concept of intention along with its associated events:

- **Intention Created.** This event signifies the creation of an intention. It occurs when an agent is committed to pursuing a specific goal and selects a plan to achieve it.

- **Intention Waiting.** This event is triggered when an intention is put on hold, typically due to external factors or dependencies that need to be fulfilled before proceeding with the intention.

- **Intention Suspended.** When an intention is intentionally suspended by the agent, this event is recorded. It implies that the agent has decided to temporarily defer pursuing the goal associated with the intention.

- **Intention Removed** the intention is removed, it is equivalent to Goal Removed.

4.3.3 Other *Jason* Level Concepts

Apart from belief and goal events, there are other significant events in the ***Jason*** framework that contribute to the detailed representation of the agent's behaviour:

- **Reasoning Cycle Started** This event marks the beginning of a new reasoning cycle for the agent.

- **Plan Events** These kinds of events refer to a change in the agent's plan library. It includes:

  - **Plan Added** This event occurs when a new plan is added to the agent's plan library.
  - **Plan Removed** When a plan is removed from the agent plan library, this event is triggered.

- **Action Events.** Recall that there are six different types of formulæ that can appear in a plan body: internal or external actions, achievement goals, test goals, mental notes, or expressions.

  - **Internal Action Finished.** This event is triggered when an execution of an internal action is completed. Internal actions are part of the predefined set of actions that agents can perform.
  - **External Action Triggered.** When an external action is initiated or triggered by the agent's reasoning cycle, this event is recorded. It marks the start of executing the external action.

  - **External Action Finished.** This event occurs when the external action is completed.
  - **Executed Deed.** Other actions (achievement/test goals, mental notes, or expressions) are represented by a generic "executed deed" event that includes information about the type of deed that was executed.

- **Signal Messages** The semantics of signals are that the receiver will have the event added to its event queue. Signals are similar to perceptions but do not get added directly to the belief base. There are two types of signals:
  - **Agent Signal:** signal sent from other agents.
  - **Artefact Signal:** signal send from the environment artefact.

- **Speech Act Messages.** In a multi-agent system, agent communication is an important aspect. To comprehensively capture these interactions, the recording of speech act events is essential. This category includes the following events:
  - **Mailbox Messages.** This event is triggered when a new message arrives in the agent's mailbox. It presents all unprocessed messages that have arrived in the mailbox.
  - **Selected Message.** Associated with the reasoning cycle's *message selection* function, this event represents the agent's choice of a specific message for further processing.
  - **New Speech Act Message.** When a message is selected, this event is triggered and generates a new speech act message for the agent. The message can be of different types: tell, achieve, or signal.
  - **Send Message.** This event is generated when the agent sends a message to another agent.

4.4 BDI Level

This section will present events related to the BDI level. The identification of events at this level focuses on the cognitive aspects of the agent following the BDI abstraction: belief, desire and intention.

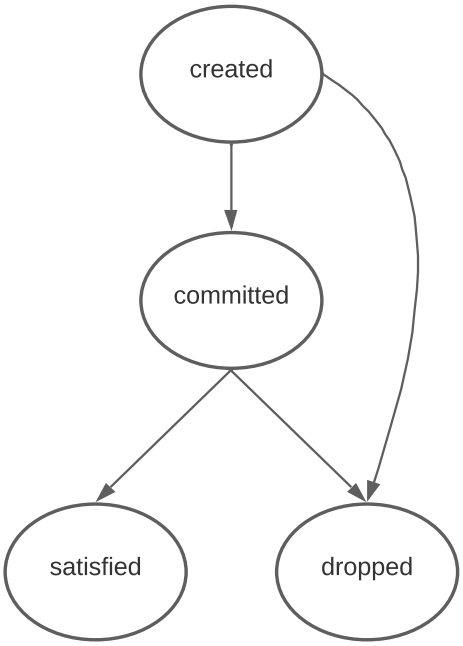

Figure 4.5: Desire states and lifecycle at the BDI level

4.4.1 Belief

At the BDI level, we identified three main events concerning possible operations related to a belief: addition, removal and update.

- **New Belief.** The addition of a new belief means that the agent believes that this information is true. The belief information can come from perception, from himself to note down information, or from messages.
- **Belief Removed.** The removal of a belief indicates that the agent no longer believes that specific information.
- **Belief Updated.** The update of a belief consists of a modification of the information associated with the previous belief.

4.4.2 Desire

In the BDI architecture, desires represent the agent's motivations or objectives. These desires go through different states and transitions that can be captured by several main events. The lifecycle of desires at the BDI level is illustrated in Figure 4.5.

The main events that can be identified in the desire lifecycle are the following:

- **New Desire.** It occurs when the agent identifies a new objective to be achieved. This event can be triggered in several scenarios:
  - initial desire
  - new desire created from another agent by an achieved message
  - new desire created from an intention
  - new desire formed with the addition of a belief
  - told by someone
  - agent receives a signal creating a desire
  - agent perception created a new desire
- **Desire committed.** When an agent commits to pursuing a particular desire by selecting a plan to achieve it, this event takes place. It signifies that the agent is actively committing to the pursuit of the desire.
- **Desire satisfied** When a desire is successfully achieved, it enters the state of satisfaction. This event indicates that the agent has successfully accomplished the desired objective.
- **Desire dropped.** If a desire cannot be realised for various reasons, such as a lack of adequate plans or other constraints, this event is triggered. As can be seen from the lifecycle, this event can occur either after the desire has been committed or after it has been created.

4.4.3 Intention

The lifecycle and the states of the intention are presented in Figure 4.6. It can be seen that there is a perfect correspondence between the lifecycle of the intention and the last states of the desire's lifecycle, it coincides exactly with the states when the desire is committed and becomes an intention. Specifically, the events identified for intention at the BDI level are:

- **Intention Created.** When the agent commits to a desire, an intention is created.
- **Execute action.** This event occurs when the agent finds the means to achieve the intention and execute the action.

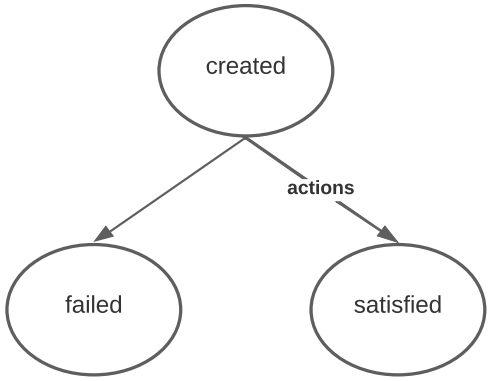

Figure 4.6: Intention states and lifecycle at the BDI level

- **Intention satisfied.** When the entire sequence of actions to satisfy the desire has been completed, the intention ends in the satisfied state.

- **Intention failed.** This event occurs if the agent realises that the means are not feasible or decides to give up the intention.

4.5 Mapping BDI Level and *Jason* Level

Once we have identified events of both levels, we need to establish a rule that enables the connection of events in *Jason* to the corresponding BDI knowledge concepts. Given the events at the ***Jason*** level and knowing their operation semantics, we can devise some patterns of events that can be mapped and connected to a new event at a higher level of abstraction. An illustrative example of this event mapping is presented in Figure 4.7. For instance, a Desire Committed event at the BDI level, in which the desire is chosen and the agent commits to it, is grounded by Select Plan, Plan Selected, and Intention Created events at the ***Jason*** level. While a Belief Updated event at the BDI level is associated with the events Belief Added and Belief Removed at the ***Jason*** level. It is important to note that these events may not necessarily occur sequentially, there can be interleaving events, such as between the group of events to commit to the desire, the agent may receive new perceptions from the environment or messages from other agents.

To provide a more detailed overview of this mapping process, in this section, we explain and show the connections between the two levels.

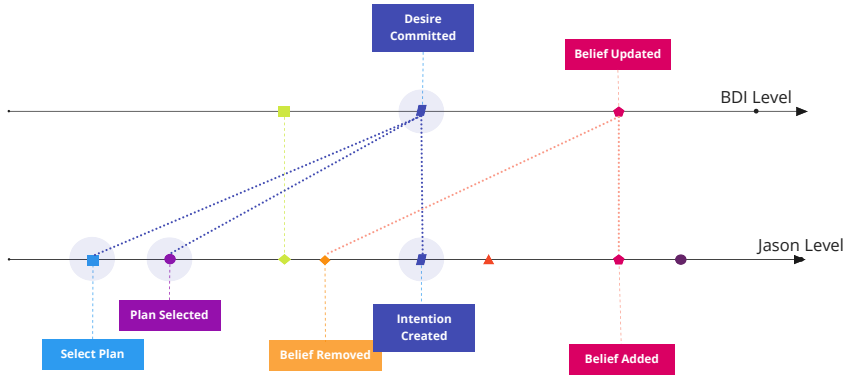

Figure 4.7: Example of the connection of the BDI and ***Jason*** levels by the mapping of events of the two levels

4.5.1 Belief Events

The mapping of belief events between the BDI level and the ***Jason*** level is presented in table 4.1.

For belief events, a **New Belief** event refers to the addition of a belief in the agent's belief base. However, because it is possible to have multiple beliefs with the same information but from different sources at the *Jason* level, we only consider the first belief added to the belief base as a new belief event at the BDI level. This new belief event can be triggered also by a new perception or a new speech act message, specifically for "tell" messages.

In fact, as illustrated in the table, the BDI's **New Belief** event can correspond to different events at the ***Jason*** level, contingent on the *source* of the information. At the BDI level, the *reason* behind the event changes depending on the source of the information. Specifically, if the source is the agent *itself*, the reason for the event is "because I noted it in my mind for future reference". However, if the source is a *percept* indicating an environmental property from an entity like an artefact *c1*, the reason transforms to "I perceived it from *c1*". On the other hand, if the source does not correspond to either of these but matches the name of another agent, it is aligned with a **New Speech Act Message** type event and a subsequent **Belief Added** event at the ***Jason*** level. This implies that someone, like *Alice* in this example, communicated the information to the agent as a *tell* message, leading the agent to believe it. In this scenario, the narrative is presented as "I believe *b* because *Alice* told me".

For the removal of a belief, the **Belief Removed** takes place when all sources of that belief have been eliminated from the agent's belief base. It is associated with the same **Belief Removed** event at the ***Jason*** level. Recall

BDI Level	**Jason Level**
New Belief "I believe *b* because I noted it in my mind for future reference"	**Belief Added** "Added belief *b*"
"I believe *b* because I perceived it from *c1*"	**New Percept** "New percept from *c1* : *b*" **Belief Added** "Added belief *b*"
"I believe *b* because *Alice* told me"	**New Speech Act Message** "New speech act message from *Alice*: *b*" **Belief Added** "Added belief *b*"
Belief Removed "I no longer believe in *b*"	**Belief Removed** "Removed belief *b*"
Belief Updated "I update the belief *b(1)* to *b(2)*"	**Belief Removed** "Removed belief *b(1)*" **Belief Added** "Added belief *b(2)*"

Table 4.1: Correspondence of Belief Events in BDI and Jason level

that the Belief Removed event in Jason implies that all beliefs with the same information are removed, whatever the source of the belief. In this case, the narrative is represented with "I no longer believe in *b*".

And finally, we have the update of a belief, **Belief Updated**. This involves deleting a belief and adding the same belief, with an updated value. This is why the event is mapped to two consecutive events: **Belief Removed** and **Belief Added** at the ***Jason*** level. The narrative in this case is simply: "I updated the belief from *b(1)* to *b(2)*".

4.5.2 Desire events

The mapping of desire events is illustrated in table 4.2. In particular, the **New Desire** event at the BDI level corresponds directly to the **New Goal Created** event at the ***Jason*** level. The narrative of this event is expressed as "I have a new desire *g*". The reason for the desire varies according to its source, whether it is an initial desire or a desire derived from another desire, belief, perception, or message, as previously discussed.

The **Desire Committed** event at the BDI level finds its equivalent in a group of events at the ***Jason*** level. This includes the **Select Plan Event**, which encapsulates the selection context and information about the chosen plan, the **Plan Selected** event which indicates the effective selection of the plan and execution of the goal, and the **Intention Created** event which marks the creation of an intention. The narration of this set of events conveys the commitment of the desire in a concise manner: "I committed to desire *g*, and it became a new intention *g/1*". Intentions are represented using the notation: `[desireName/intentionId]`.

Lastly, the removal of the intention (**Intention Removed**) and goal (**Goal Removed**) is associated with the final state of the desire according to the result. When the intention is finished and the goal is *achieved*, it corresponds to a **Desire Satisfied** event, narrated as "I have satisfied my desire *g* because its intention *g/1* has finished". Conversely, if the intention and goal result in *failure* state, it aligns with a **Desire Dropped** event, narrated as "I gave up desire *g* because its intention *g/1* failed".

4.5.3 Intention events

The mapping of intention events between the BDI level and the ***Jason*** level is illustrated in table 4.3.

The correspondence between intention events in these two levels is similar. When an intention is created, indicated by the **Intention Created** event at ***Jason*** level, it aligns with a **Desire Committed** event at BDI level, as

BDI Level	**Jason Level**
New Desire	**New Goal Created**
"I have a new desire *g* because it is an initial desire"	"Goal *g* created, state: *pending*"
"I have a new desire *g2* because it is a desire created from *g*"	"Goal *g2* created, state: *pending*"
Desire Committed	**SelectPlanEvent**
"I committed to desire *g*, and it became a new intention *g/1*"	"Plan options for *g* are [...] The plan selected for *g* is `g : (count(X) & (X < 3)) <- a1; !g2; a3.`"
	PlanSelected
	"Plan *g* selected, state: *executing*"
	Intention Created
	"Intention *1 g* created, state: *running*"
Desire satisfied	**Intention Removed**
"I have satisfied my desire *g* because its intention *g/1* finished"	"Intention 1 removed, state: *undefined*"
	Goal Removed
	"Goal *g* removed because the goal is *achieved*"
Desire dropped	**Intention Removed**
"I gave up desire *g* because its intention *g/1* failed"	"Intention 1 removed, state: *undefined*"
	Goal Removed
	"Goal *g* removed because the goal is *failed*"

Table 4.2: Correspondence of Desire Events in BDI and Jason level

BDI Level	**Jason Level**
Desire Committed "I committed to desire *g*, and it became a new intention *g/1*"	**SelectPlanEvent** "Plan options for *g* are [...] The plan selected for *g* is `g : (count(X) & (X < 3)) <- a1; !g2; a3.`" **PlanSelected** "Plan *g* selected, state: *executing*" **Intention Created** "Intention *1 g* created, state: *running*"
Executed action "I'm executed action *a1* because of intention *g/1*"	**Intention Created** "Intention 1 *g* created, state: *running*, current step: *a1* **External Action Finished / Internal Action Finished / Execute Deed** "External action / Internal Action / Deed *a1* executed"

Table 4.3: Correspondence of Intention Events in BDI and Jason level

discussed with desire events. This alignment is reinforced by the narrative, which not only communicates the commitment of the desire but also includes relevant information about the intention itself.

Subsequently, when the agent proceeds to execute the actions following the plans of the intention, the **Executed action** event at the BDI level is connected with **Intention Created** and a **Execute Deed**, **External Action Finished** or **Internal Action Finished** event at ***Jason*** level. In this way, the information of intention can be added to the motivation of the action performed, so the narration presents as "I'm executing action *a1* because of intention *g/1*".

4.6 Formalized Mapping for Prototype Development

In the previous sections, we delved into the theoretical foundations of our multilevel explainability framework. We introduced the core concepts, identified and explored various levels of abstraction, defined relevant events at each level, and presented a mapping between the ***Jason*** and BDI levels. This identification framed the conceptual elements necessary for the development of our prototype.

Although our current focus has been on the ***Jason*** language, which is a BDI-based technology, we propose that the identification of abstraction levels through explainability can have broader visions. These levels can be adapted to systems that are not necessarily designed as BDI systems. Specifically, we argue that provided a mapping from the lower level, it would be beneficial to describe any intelligent systems using the BDI framework since it would be easier for humans to understand the mentalistic explanation of a system that behaves rationally given its desires and current beliefs i.e. following the so-called intentional stance [23].

With these formalized mappings and connections of different levels of abstraction, we are now ready for the practical implementation of the multi-level explainability tool, as presented in the next chapter.

Chapter 5

Prototype Implementation

As described in section 4.1, the tool presented with this thesis consists of two main components: the **logging component**[1] to be integrated with the multi-agent system to track its behaviour, and the **explanation component**[2], a web application to visualise the multi-level system narrative. This chapter deals with the prototyping and implementation phases of both components.

5.1 Logging Component

The prototyping and implementation of the Logging component involve extending the ***Jason*** language with new functionalities to enable the logging and tracing of events within the system's behaviour. As previously discussed in section 3.1, since ***Jason*** is developed in Java, these customisations and extensions can be achieved through the development of libraries and components using the Java language and the object-oriented paradigm.

The following sections will describe how the ***Jason*** interpreter is extended and customised for developing the logging functionality and how the overall architecture of the logging component is structured.

5.1.1 Architecture

The customised components of the *Jason* interpreter are depicted in Figure 5.1. In particular, the *Agent* class and the *Agent Architecture* are extended and improved to support the logging functionality through the creation of the `LoggerAg` and `LoggerArch` classes, respectively.

[1]Logging component: `https://github.com/yan-elena/agent-logging`

[2]Explanation component: `https://github.com/yan-elena/agent-explanation`, a deployed application can be accessed directly via this link: `https://yan-elena.github.io/agent-explanation/`

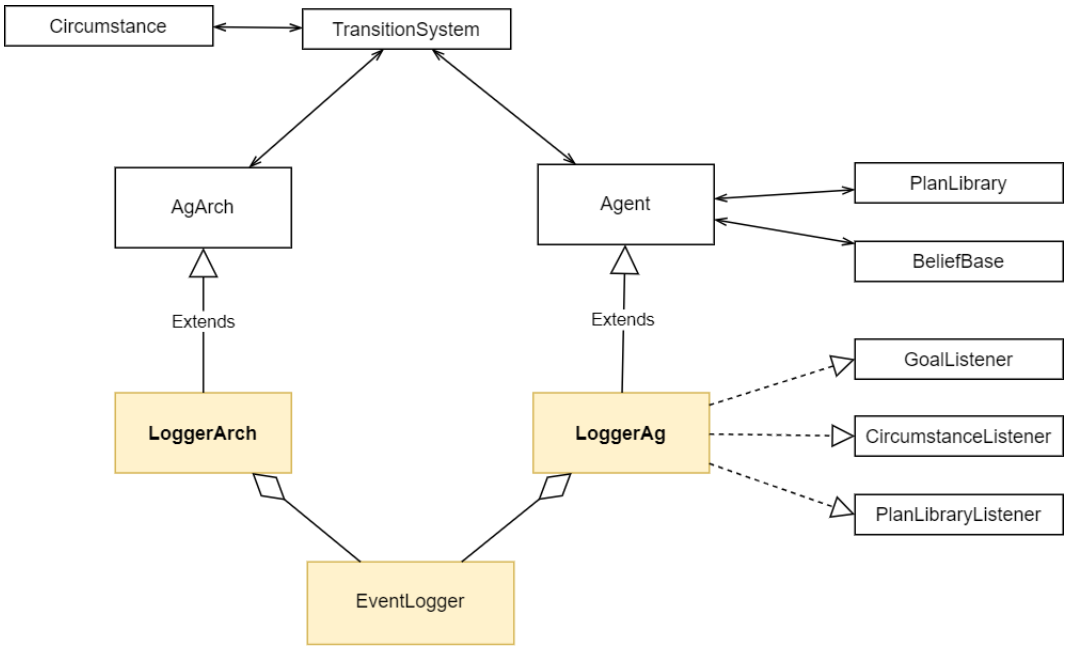

Figure 5.1: Customising the ***Jason*** interpreter

The `Agent` class serves as a crucial part of the interpreter, responsible for various functions such as belief revision and selection functions used in the reasoning cycle. It contains essential elements, including the `BeliefBase`, `PlanLibrary`, user-defined selection and trust functions for messages, belief update and revision functions, and a `Circumstance` that contains information about pending events, intentions, and other relevant structures.

To enable logging for agents, the `Agent` class is extended by the `LoggerAg` class. This extension involves overriding methods to introduce logging functionality with the agent's behaviour. Additionally, the `LoggerAg` class implements various listener interfaces that enable the agent to listen for specific events. These interfaces include:

- `GoalListener`: This interface provides methods to listen for events related to goals. These events contain state changes such as *pending*, *executing*, *suspended*, *resumed*, *waiting*, *achieved*, *dropped*, *failed*, and *finished*.

- `CircumstanceListener`: This interface enables the agent to listen for events related to its circumstance. Events include `eventAdded` and changes in intention states, such as *added*, *dropped*, *suspended*, *waiting*, *resumed*, and *executing*.

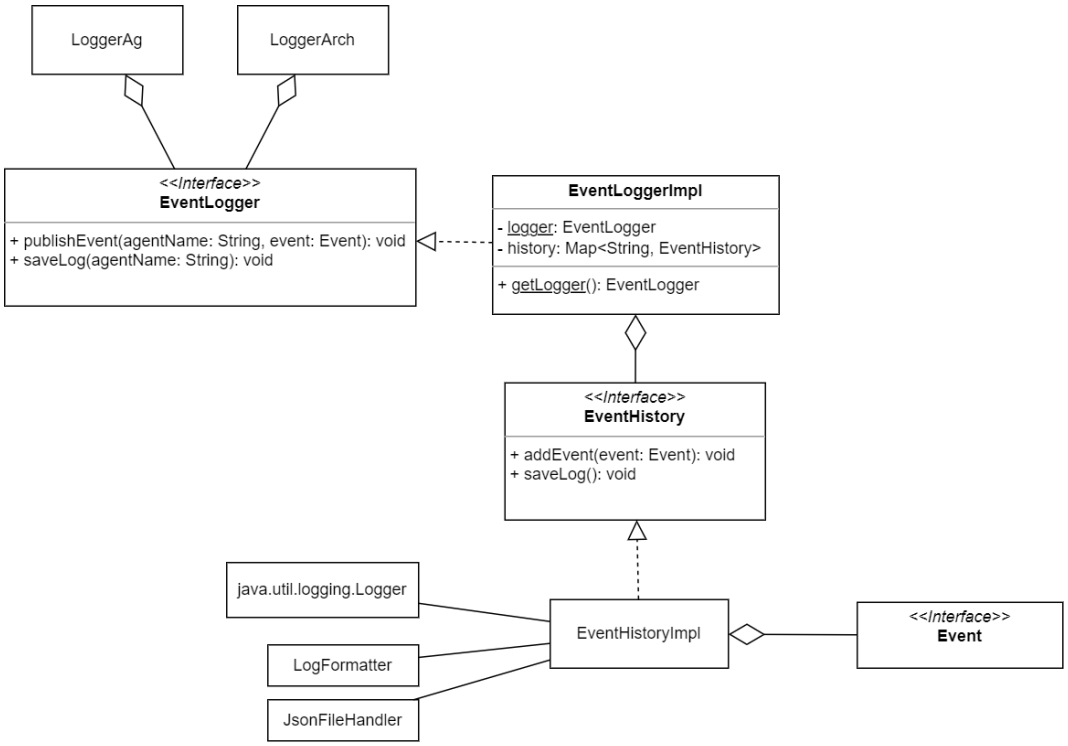

Figure 5.2: Overall architecture of the logging component

- `PlanLibraryListener`: This interface allows the agent to monitor changes to the PlanLibrary, particularly when plans are added or removed from it.

On the other side, the `LoggerArch` class extends ***Jason***'s `AgArch`, which serves as the foundation of the agent's architecture, dictating how the agent interacts with the external world, specifically with the environment and other agents. The agent architecture encapsulates the functionality of the agent's sensory perception, action execution, and communication mechanisms. These roles are realised through the `perceive` method for handling sensor input, the `act` method for managing actuator actions, and methods for handling message reception and communication. The `LoggerArch` class, in conjunction with `LoggerAg`, is associated with agents that require logging functionality, in the multi-agent system configuration file.

Within this customization, each of these methods designated for monitoring relevant events incorporates calls to methods of `EventLogger` class. These invocations are used to manage and store events, ensuring their proper recording in the log.

The overall architecture of the logger component is presented in Figure 5.2.

Specifically, the central element of this structure is the `EventLogger`, which orchestrates the management of logs from all agents in the system. This component has an interface includes methods that are dedicated to publishing the events associated with a specific agent and subsequently saving the log to a file. To ensure the consistency of this architecture, the `EventLogger` is designed as a singleton, guaranteeing the existence of a single instance for the entire system.

The workings of the `EventLogger` class exploit a map-based data structure, which houses the event history of all agents. The individual log of a given agent is represented by the `Agent History` component. The interface of this component provides methods for including events and storing the log. It uses the Java Logging API to implement the logging mechanism, using two auxiliary classes to simplify the process: `LogFormatter`, responsible for formatting the log, and `JsonFileHandler`, able to persist the log in a JSON file.

Events are encapsulated by the corresponding `Event` interface, which provides a uniform structure to represent different types of events. Every ***Jason*** level event outlined in section 4.3 is instantiated as a class that implements the `Event` interface, adapting its structure to the specific characteristics of the event type.

5.1.2 Events Prototype

Events are unified under a common interface, which can be implemented by specific classes tailored to different event types. This design choice is motivated by the need for specialised properties and components to handle the diverse range of events.

As depicted in Figure 5.3, the image provides a comprehensive overview of all event types at the ***Jason*** level. These events are systematically classified into several distinct fields, each of which has a unique purpose:

1. reasoning cycle event,
2. percepts events,
3. belief events,
4. signal events,
5. goal events,
6. intention events,
7. plan events,

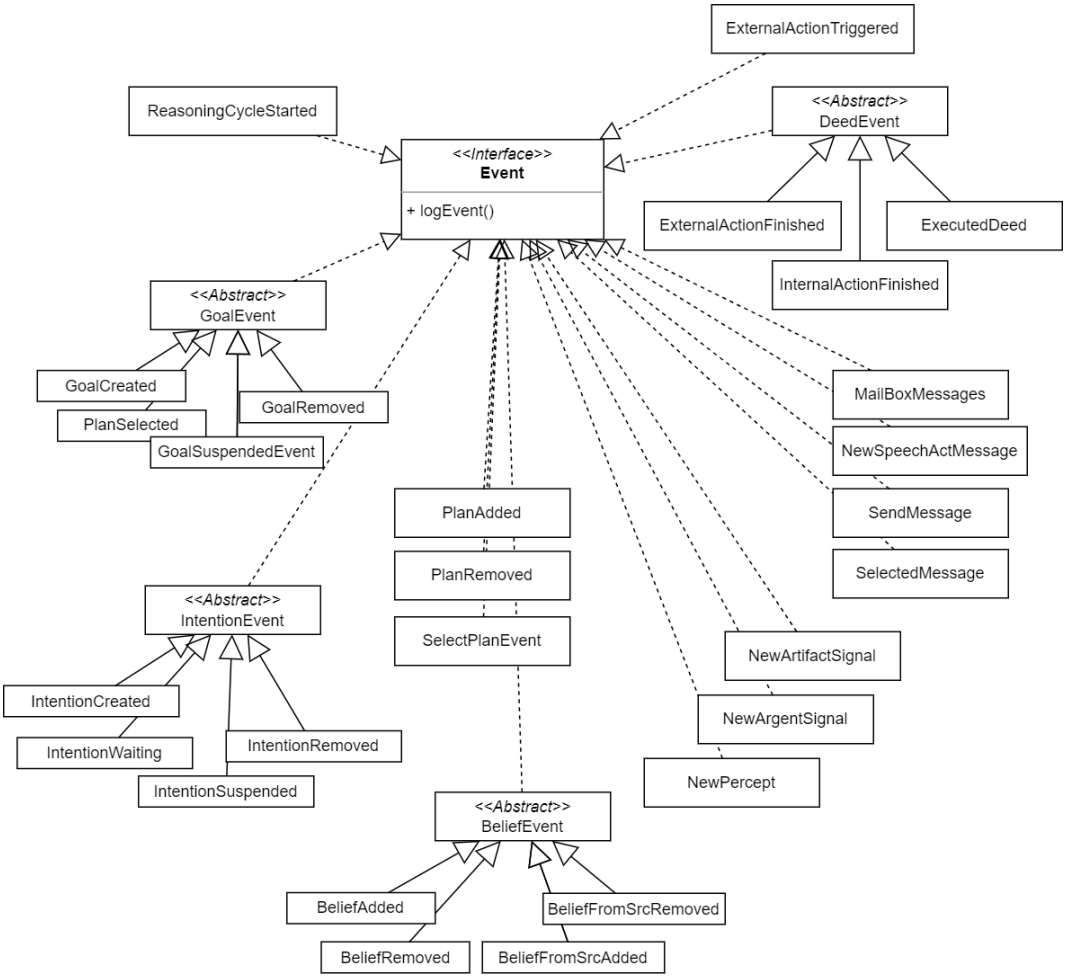

Figure 5.3: Overview of all events

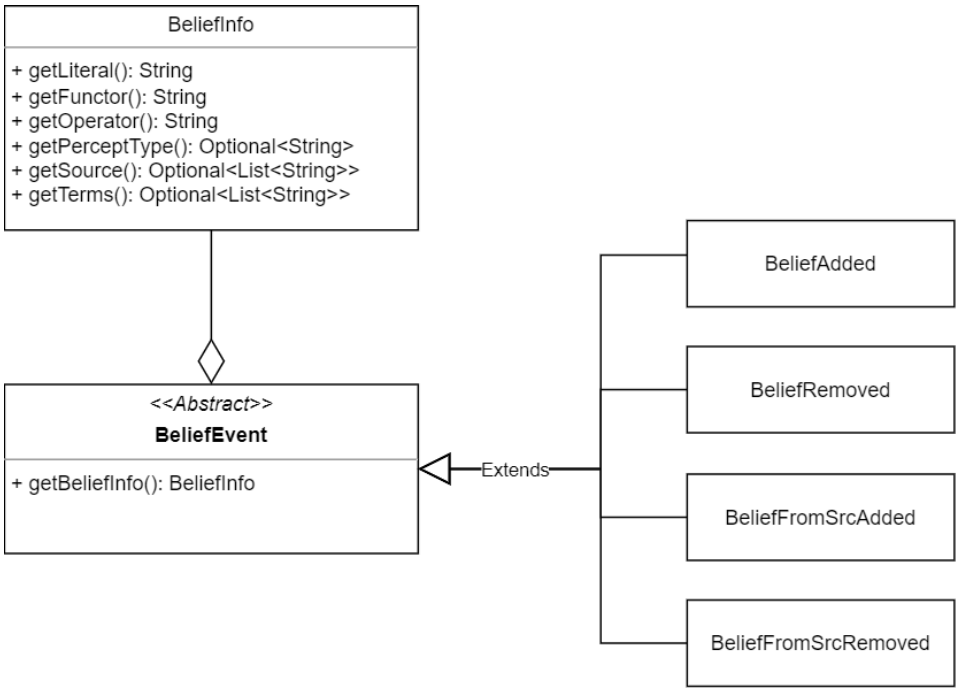

Figure 5.4: Class diagram of belief events

8. action events, and

9. speech act message events.

In the subsequent parts, we will provide a detailed description of the design prototype for the relevant categories.

Belief Events

In the Figure 5.4 is illustrated the class diagram of belief events. A belief is encapsulated by the `BeliefInfo` class, which contains relevant information about the literal, functor, operator, type, terms, and sources of the belief.

Belief events are generated with the `eventAdded` method of the `CircumstanceListener` interface. These events include different scenarios depending on the source and operation of the belief, as described in section 4.3.1. In particular, there are: `BeliefAdded`, triggered when a new belief is added to the agent's *belief base*, `BeliefFromSrcAdded`, in the case of an addition of a new source in an existing belief, `BeliefRemoved`, if all sources of a belief are removed, and `BeliefFromSrcRemoved`, when a specific source is removed from an existing belief.

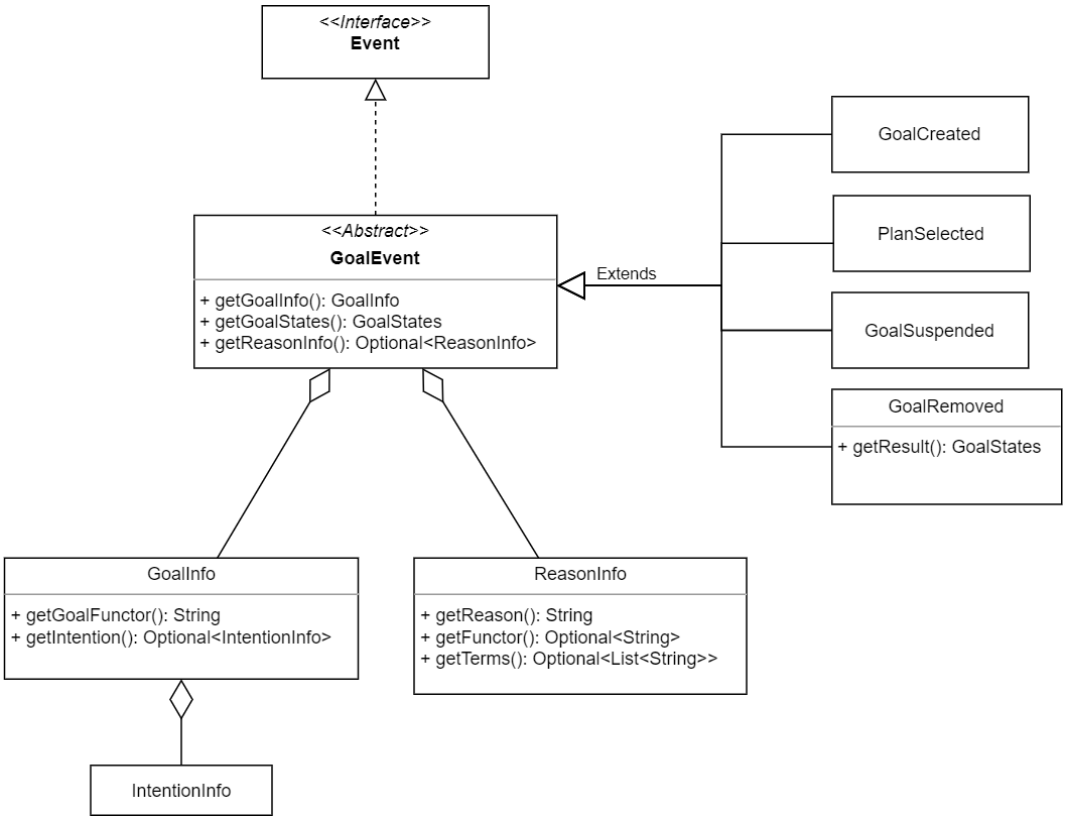

Figure 5.5: Class diagram of goal events

Goal Events

The class diagram depicted in Figure 5.5 illustrates the structure of the goal events. A *goal* is represented by the object `GoalInfo`, encapsulating essential details such as its *functor* and associated *intention*.

The `GoalEvent` is an abstract class that contains the common property of a goal event, it could be extended by a specific goal's events. The class encompasses information about the goal, its current state (*pending*, *executing*, *suspended*, or *finished*) and the *reason* behind its state transition. The `ReasonInfo` is a structure with a functor and a list of terms.

Goal events are triggered by the `GoalListener` interface, and following the discussion in section 4.3.2 we have these classes, each representing a specific type of goal event:

- `GoalCreated`. This event is recorded when a `goalStarted` method from the `GoalListener` interface is called. This event signifies the creation of a goal. At this stage, the goal is in a *pending* state, awaiting execution.

- `PlanSelected.` This event is recorded when the `goalExecuting` method is triggered. This event marks the transition of a goal from *pending* state to *executing* state. The goal is selected, committed by the agent, and becomes a new intention.
- `GoalSuspended.` This event is added when the `goalSuspended` method is triggered. This event denotes the suspension of a goal, moving it to the *suspended* state.
- `GoalRemoved.` Two distinct methods led to the generation of this event: `goalFinished` and `goalFailed`, both indicating that the goal has reached a *finished* state. This class also contains information about the result of the goal, which can be categorised as *dropped*, *achieved* or *failed*.

Intention Events

As illustrated in Figure 5.6, the structure of intention events is similar to goal events. An intention is represented by the class `IntentionInfo`. It encapsulates the essential attributes, such as the *id*, *states* and relevant information about the selected applicable plan, which are found in `IntendedMeansInfo`. The plan's course of action serves as the intended means by which the agent intends to achieve its goal.

The `IntentionEvent` class contains complete information about the intention and the reason associated with its happening. These events are generated through interactions with the `CircumstanceListener` interface. Four distinct types of intention events are identified, as discussed in section 4.3.2:

- `IntentionCreated` This event is generated when a new intention is introduced, triggered by the `intentionAdded` method of the `CircumstanceListener`. It represents the creation of a new intention.
- `IntentionWaiting` Triggered by the `intentionWaiting` method of the `CircumstanceListener`, this event denotes the scenario where an intention is in a *waiting* state. Additionally, it captures the reason behind the intention's transition to the waiting state.
- `IntentionSuspended` Resulting from the `intentionSuspended` method, this event indicates that an intention has been suspended. Like the `IntentionWaiting` event, it also documents the rationale behind the suspension.
- `IntentionRemoved` Recorded when an intention is dropped, this event denotes the conclusion of an intention's lifecycle. Following this event, the intention's state becomes *undefined*.

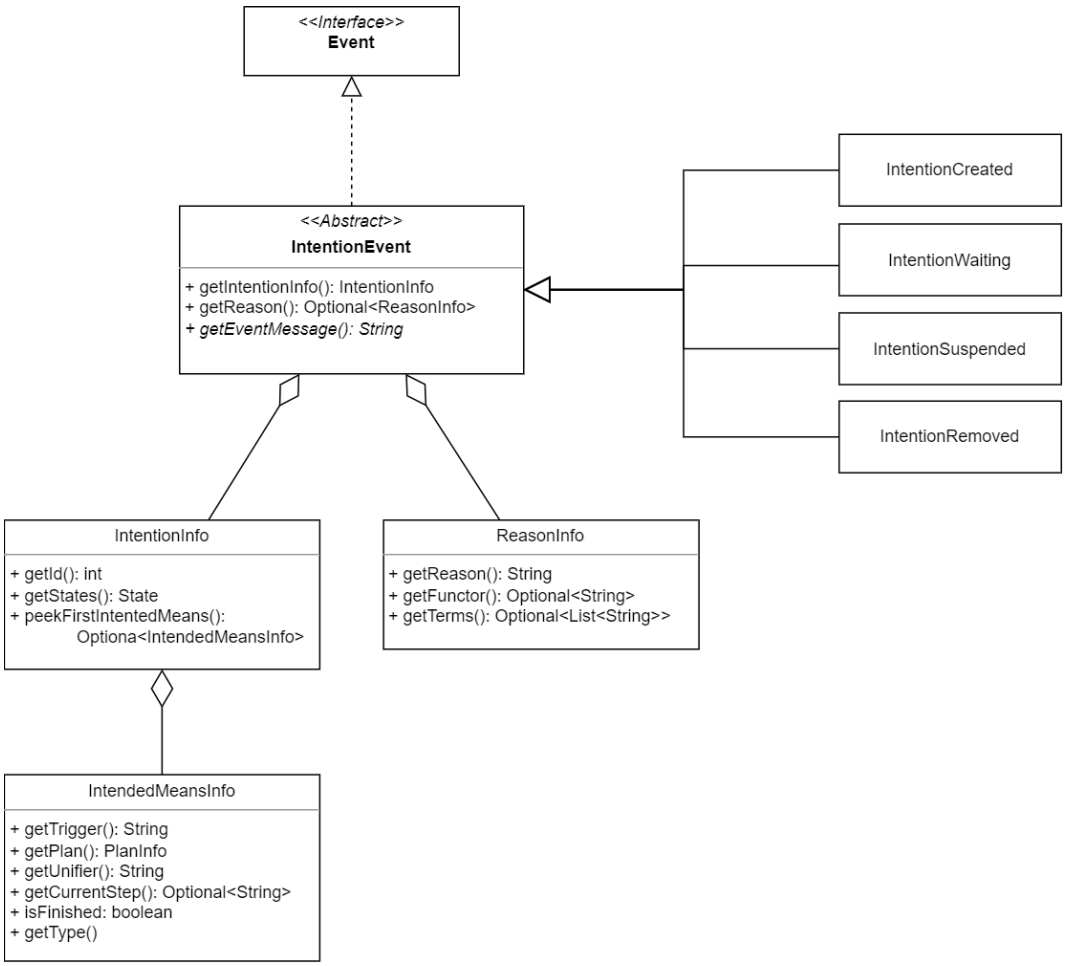

Figure 5.6: Class diagram of intention events

5.2 Explanation Component

The *explanation* component is responsible for presenting the explanation of the multi-agent system in a comprehensible and user-friendly manner to users.

To fulfil this objective, we chose to create a **web application** for its ease of use and accessibility. A web application provides an interface for users to interact with and understand the intricate reasoning and behaviour of the multi-agent system. Through this web application, users can navigate explanations, view event narratives, and comprehend the agents' decision-making process.

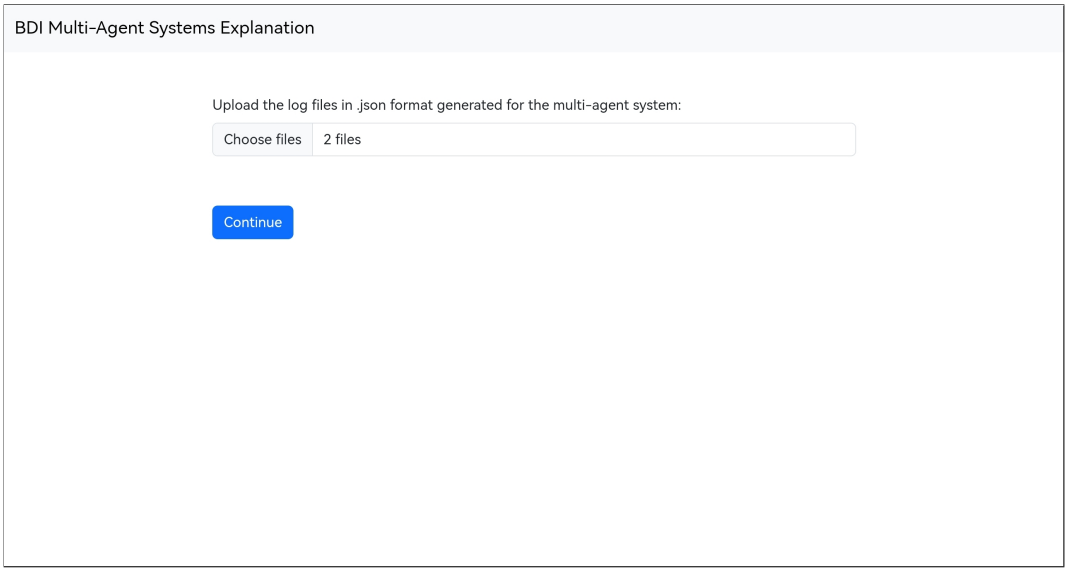

Figure 5.7: Web Application - Initial page for uploading log files

As illustrated in Figure 5.7, the web application provides an initial page for users to upload the log files generated by the *logging component*. Given these log files, the application will be able to generate a complete multi-level explanation.

Once the log file has been uploaded, the web application provides users with a comprehensive overview of all agents present in the multi-agent system, as shown in Figure 5.8. On this page, users have the option of selecting the agent to explore and the possibility of delving into different narrative perspectives, including the *complete narrative*, the *belief narrative*, the *desire narrative*, or the *intentions narrative*.

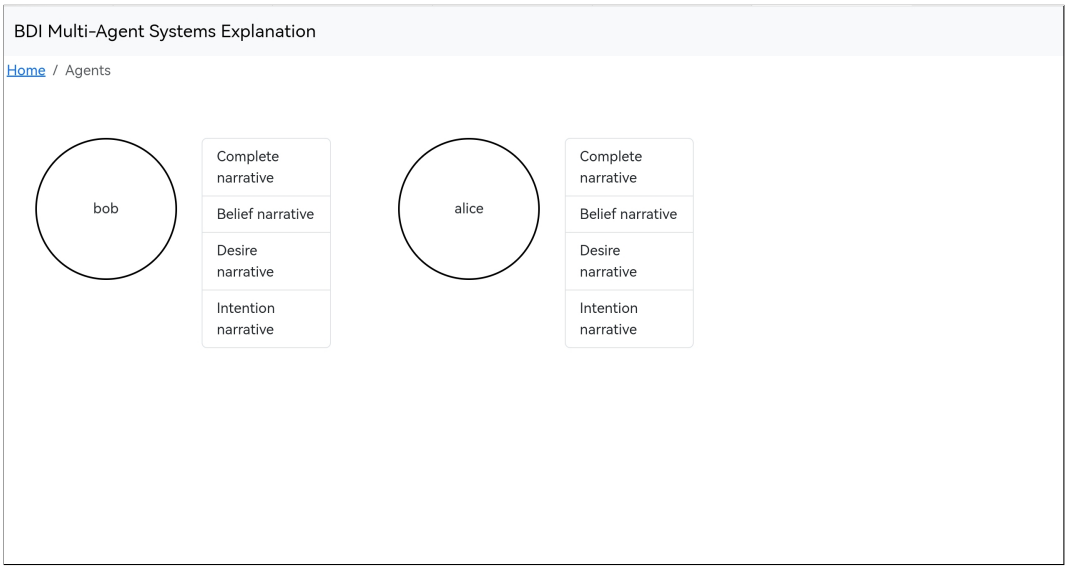

Figure 5.8: Web application - overview of all agents in the multi-agent system

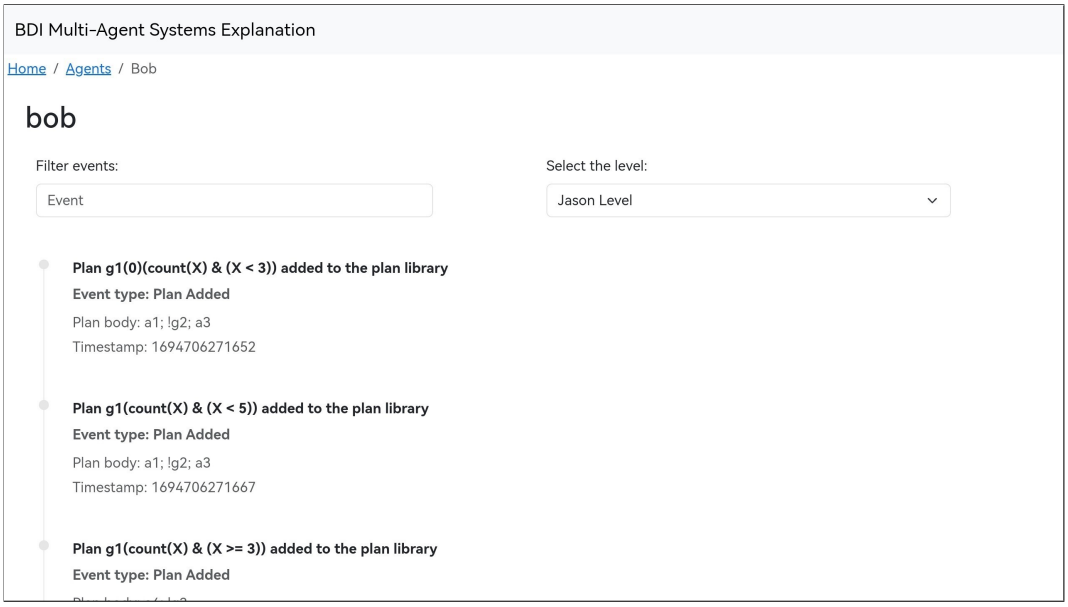

Figure 5.9: Web application - explanation page with Jason level

If the user chooses to explore a complete narrative for a specific agent, the web application provides an in-depth view of all event narratives at ***Jason*** level, as exemplified in Figure 5.9. For each event, the narrative is elaborated, detailing the sequence of actions, decisions, and their reasons. The type of event is clearly indicated, together with its *timestamp*. In addition, some

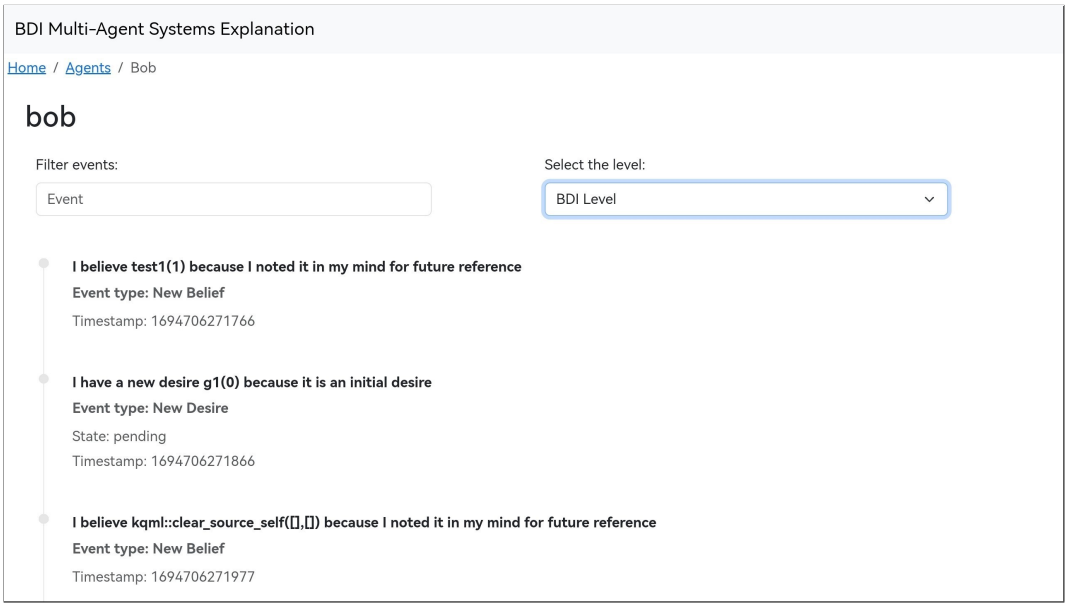

Figure 5.10: Web application - explanation page with the BDI level

events are accompanied by additional contextual *information*, which enhances the user's understanding of the meaning and implications of the event.

The web application provides an interface that allows users to refine the exploration according to their preferences. On this page, users can apply a *filter* to events and choose the *narrative level* they wish to explore.

The Figure 5.10 illustrates the sequence of narrative at a BDI level. This BDI-level narrative presentation offers a higher-level perspective, focusing on concepts related to beliefs, desires, and intentions. The BDI level narratives are realised with a first-person approach, which gives the explanation a more personal and immersive quality.

The filter component is a fundamental tool that allows users to selectively focus on events of interest to them. This component not only allows users to filter events according to their type (belief, intention, desire, etc.), but also allows them to focus on events associated with specific behaviours or the entire lifecycle of a particular desire. The Figure 5.11 shows an example of practical use of the filter. In this case, the application is configured to present a comprehensive narrative, including all events related to the desire *g1*. This powerful feature explains the *g1* lifecycle, tracing its path from the beginning as a new desire, through its commitment and transformation into an intention, the execution of associated actions, and finally its realisation or failure. This functionality offers a valuable way to fully understand the lifecycle of a specific event, along with the intentions and actions associated with it.

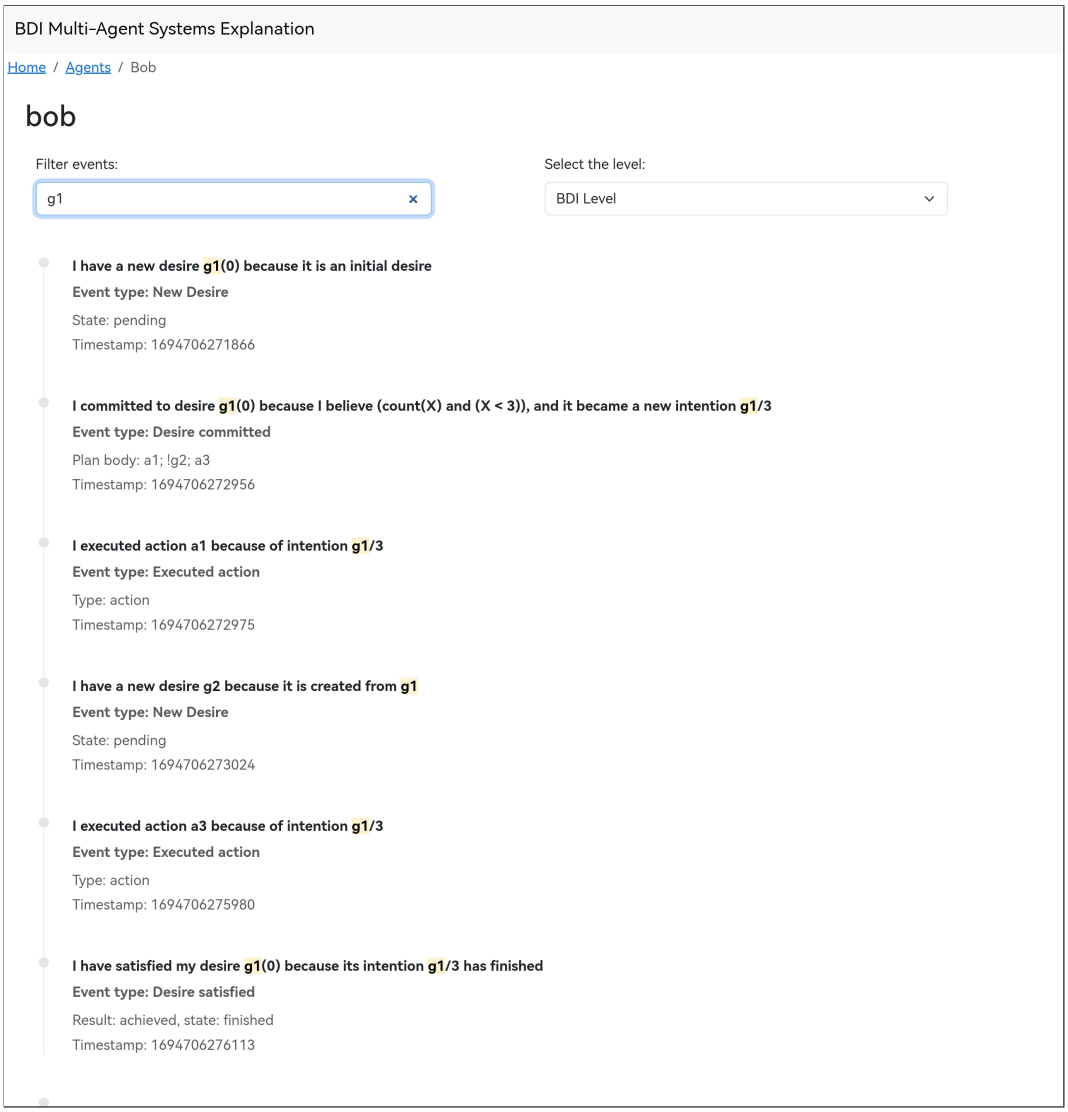

Figure 5.11: Web application - filtering and *g1* desire lifecycle explanation

Chapter 6

Evaluation

For the evaluation part, we move to evaluating the effectiveness of the tool by taking the example of domestic robots from the Jason distribution. The narrative produced for the example in the two explanation levels shows how the tool can be used for debugging or understanding the system.

6.1 Domestic Robot Application

Consider the following *"domestic robot"* example[1], which comes with ***Jason***'s distribution. [11]

> "A domestic **robot** has the goal of serving beer to its **owner**. Its mission is quite simple, it just receives some beer requests from the owner, goes to the fridge, takes out a bottle of beer, and brings it back to the owner. However, the robot should also be concerned with the beer stock (and eventually order more beer using the **supermarket**'s home delivery service) and some rules hard-wired into the robot by the Department of Health (in this example this rule defines the limit of daily beer consumption)." [11]

The example is composed of three agents: the *robot*, the *owner*, and the *supermarket*.

6.1.1 Configuration of the Tool

To use the implemented tool, the following dependency of the *logging component* must be included in the project's *gradle* file.

[1] `https://github.com/yan-elena/domestic-robot-example`

```
dependencies {
    implementation 'io.github.yan-elena:agent-logging:0.1.0'
}
```

Listing 6.1: Domestic robot example: configuration for the logging component dependency in the project gradle file

Then, we proceed to configure the customised `agentArchClass` and `agentClass` to enable the logging component for the interested agents. The configuration of the domestic robot example is as follows:

```
MAS domestic_robot {
    environment: example.HouseEnv(gui)

    agents:
            robot
                agentArchClass log.LoggerArch
                agentClass     log.LoggerAg;

            owner
                agentArchClass log.LoggerArch
                agentClass     log.LoggerAg;

            supermarket
                agentArchClass log.LoggerArch
                agentClass     log.LoggerAg;

    aslSourcePath: "src/agt";
}
```

Listing 6.2: Domestic robot example: configuration for the domestic robot MAS application

6.1.2 Running the System with the Logging Tool

When the application is launched, events are recorded as they occur in the system. Two log files are produced. The first one contains a low-level narrative very closely related to the code, an extract of the log of the robot agent can be seen in Listing 6.3.

```
//...
[1693582280618] ReasoningCycleStarted: New reasoning cycle started: 2
[1693582280627] MailBoxMessages: Messages in mailbox:
   achieve message from owner: has(owner,beer)
```

```
[1693582280635] SelectedMessage: Selected Message: has(owner,beer)
[1693582280641] NewSpeechActMessage: New speech act message [achieve] from
    owner: has(owner,beer)
[1693582280655] GoalCreated: Goal has created
[1693582280713] SelectPlanEvent: Plan options for has are:
   has(owner,beer) : (available(beer,fridge) & not (too_much(beer))) <-
       !at(robot,fridge); open(fridge); get(beer); close(fridge);
       !at(robot,owner); hand_in(beer); ?has(owner,beer); .date(YY,MM,DD);
       .time(HH,NN,SS); +consumed(YY,MM,DD,HH,NN,SS,beer).
The plan selected for has is  has(owner,beer) : (available(beer,fridge) &
    not (too_much(beer))) <- !at(robot,fridge); open(fridge); get(beer);
    close(fridge); !at(robot,owner); hand_in(beer); ?has(owner,beer);
    .date(YY,MM,DD); .time(HH,NN,SS); +consumed(YY,MM,DD,HH,NN,SS,beer).
[1693582280717] PlanSelected: Plan has selected, state: executing
[1693582280721] IntentionCreated: Intention 3 has created, state: undefined
   current step: !at(robot,fridge); open(fridge); get(beer); close(fridge);
       !at(robot,owner); hand_in(beer); ?has(owner,beer); .date(YY,MM,DD);
       .time(HH,NN,SS); +consumed(YY,MM,DD,HH,NN,SS,beer)
[1693582280724] GoalCreated: Goal at (sub-goal of has) created
[1693582280729] ExecutedDeed: Deed at(robot,fridge) of type achieve
    executed - from file:src/agt/robot.asl:20

//...
```

Listing 6.3: Domestic robot example: extract of the `.log` file generated for the *robot* agent

The second log generated is of the `.json` type, this log file will be used to load into the web application for multi-level explainability. An extract of this log for the robot agent is represented in Listing 6.4.

```
[
  //...
  {
    "timestamp": 1693582280618,
    "message": {
      "type": "ReasoningCycleStarted",
      "event": {
        "cycleNumber": 2
      },
      "log": "New reasoning cycle started: 2"
    }
  },
  {
    "timestamp": 1693582280627,
    "message": {
      "type": "MailBoxMessages",
      "event": {
        "messages": [
```

```
        {
          "id": "mid1",
          "type": "achieve",
          "sender": "owner",
          "receiver": "robot",
          "message": "has(owner,beer)"
        }
      ]
    },
    "log": "Messages in mailbox: \n\tachieve message from owner:
        has(owner,beer)"
  }
},
{
  "timestamp": 1693582280635,
  "message": {
    "type": "SelectedMessage",
    "event": {
      "selected": {
        "id": "mid1",
        "type": "achieve",
        "sender": "owner",
        "receiver": "robot",
        "message": "has(owner,beer)"
      }
    },
    "log": "Selected Message: has(owner,beer)"
  }
},
{
  "timestamp": 1693582280641,
  "message": {
    "type": "NewSpeechActMessage",
    "event": {
      "message": {
        "id": "mid1",
        "type": "achieve",
        "sender": "owner",
        "receiver": "robot",
        "message": "has(owner,beer)"
      }
    },
    "log": "New speech act message [achieve] from owner: has(owner,beer)"
  }
},
{
  "timestamp": 1693582280655,
  "message": {
    "type": "GoalCreated",
```

```
    "event": {
      "goalInfo": {
        "goalFunctor": "has",
        "intention": {
          "value": null
        }
      },
      "goalStates": "pending",
      "reasonInfo": {
        "value": null
      }
    },
    "log": "Goal has created"
  }
 },

 //...
]
```

Listing 6.4: Domestic robot example: extract of the `.json` file generated for the *robot* agent

6.2 Narrative of the Domestic Robot Application

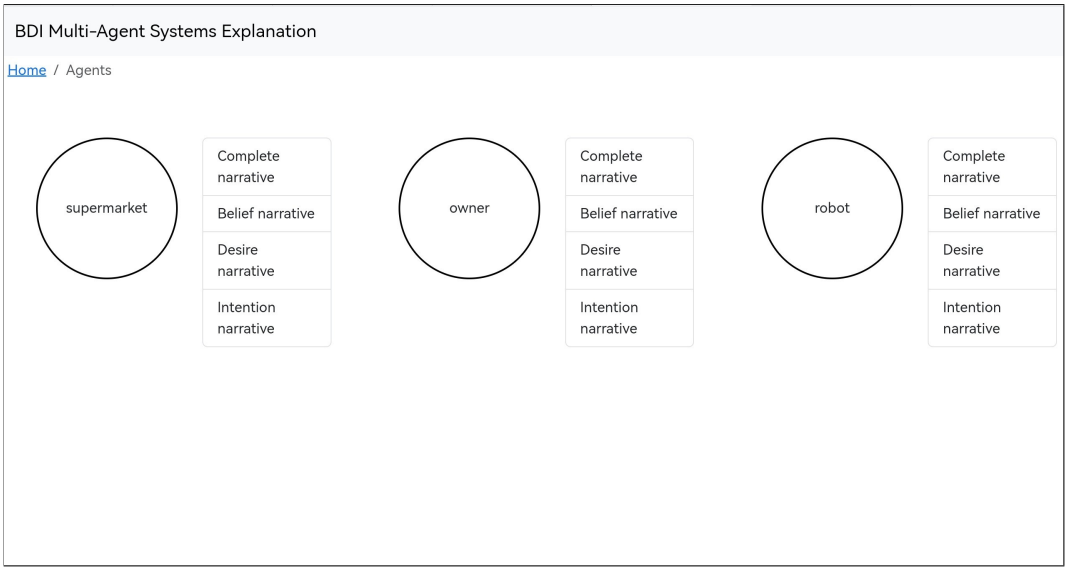

Figure 6.1: Domestic robot example: an overview of all agents of the application

Once the log files have been uploaded to the web application, the user can select the agent with which he wants to view the narration. (Figure 6.1)

Through narration, it is possible to reconstruct the whole history of the agents, which helps in explaining and understanding the behaviour of the system. Let us now visualise the narrative at the BDI level for each agent in the example application.

6.2.1 Owner

The ***Jason*** code for the owner agent in this example is as follows.

```
/* Initial goals */

!get(beer). // initial goal: get a beer
!check_bored. // initial goal: verify whether I am getting bored

+!get(beer) : true
   <- .send(robot, achieve, has(owner,beer)).

+has(owner,beer) : true
   <- !drink(beer).
-has(owner,beer) : true
   <- !get(beer).

// while I have beer, sip
+!drink(beer) : has(owner,beer)
   <- sip(beer);
     !drink(beer).
+!drink(beer) : not has(owner,beer)
   <- true.

+!check_bored : true
   <- .random(X); .wait(X*5000+2000); // i get bored at random times
      .send(robot, askOne, time(_), R); // when bored, I ask the
         robot about the time
      .print(R);
      !check_bored.

+msg(M)[source(Ag)] : true
   <- .print("Message from ",Ag,": ",M);
      -msg(M).
```

Listing 6.5: Domestic robot example: *owner* agent code

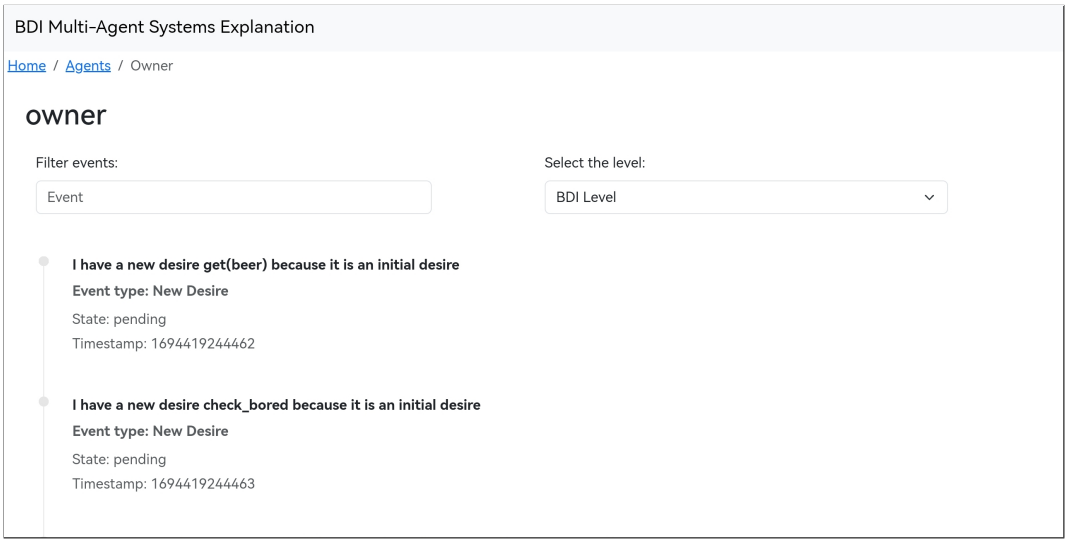

Figure 6.2: Domestic robot example: *owner* agent initial desires

The code represented may not be immediate for users who are not experts in the language. Looking at the narrative at the BDI level (see Figure 6.2), we first notice that the owner agent has two initial desires: `get(beer)` to request beer and `check_bored`, to check whether the owner is getting bored.

We can view the narrative of the desire `get(beer)` by filtering the related events in the search bar, as illustrated in Figure 6.3. At this point, we see that the desire `get(beer)` is committed, and it becomes a new intention. The plan consists of sending the robot a message to achieve the goal `has(owner, beer)`. The desire concludes with satisfaction. The desire narrative is very clear and concise, including all the stages of the desire life cycle and all the elements needed for understanding.

Once the robot has achieved the goal `has(owner, beer)`, the owner perceives it and believes in `has(owner,beer)`, as depicted in Figure 6.4. This belief generates an event in which the owner starts drinking beer with the desire `drink(beer)`. To drink the beer, the owner takes sips, with the creation of more desires, until the beer is finished.

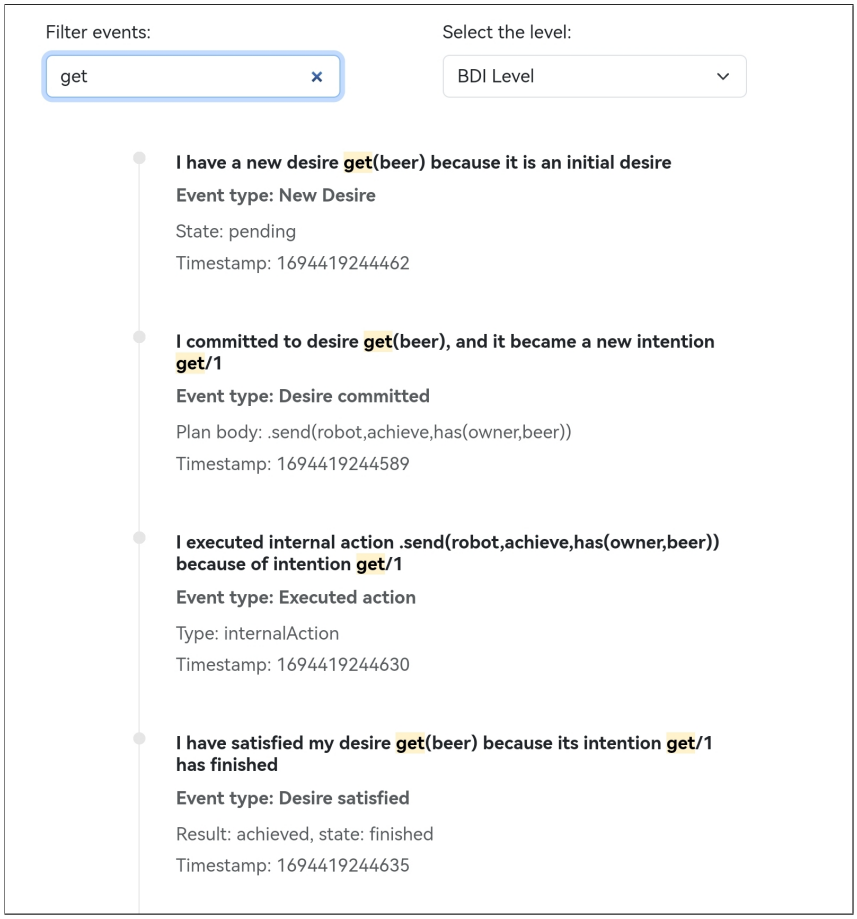

Figure 6.3: Domestic robot example: *owner* agent `get(beer)` desire lifecycle

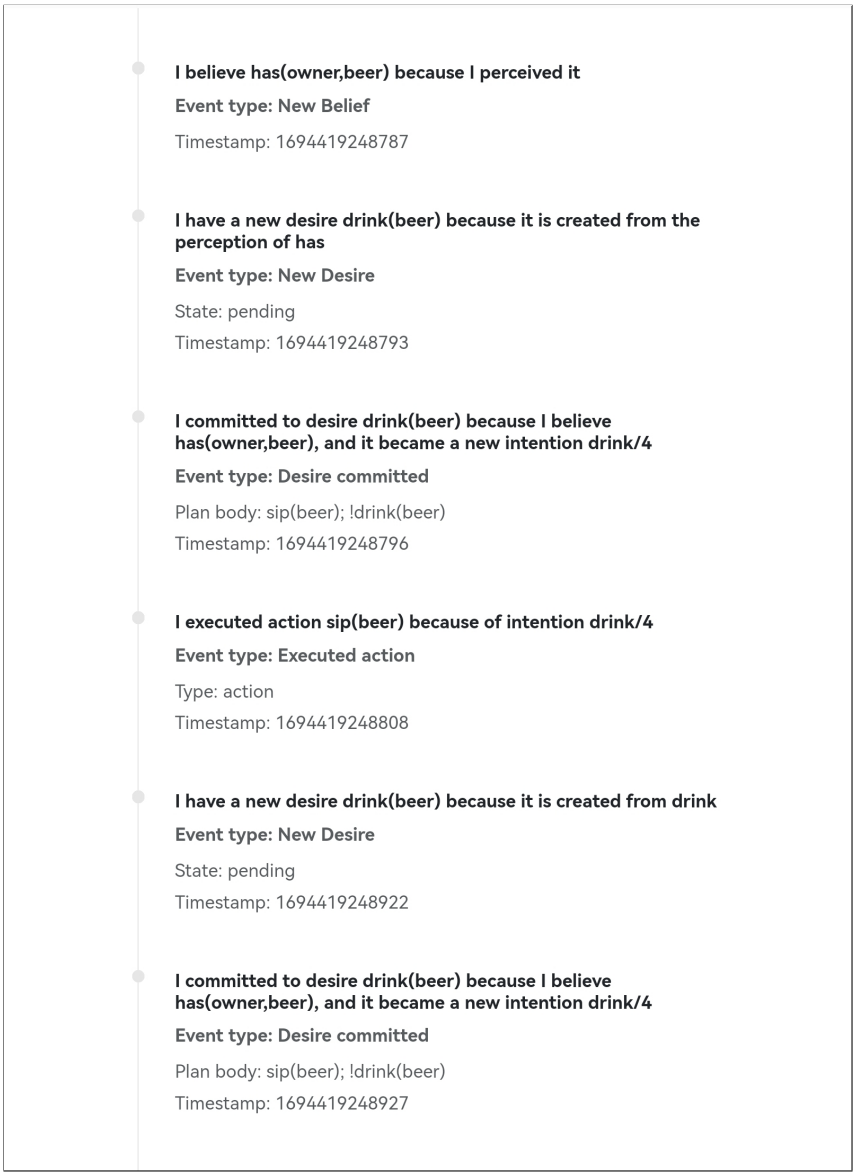

Figure 6.4: Domestic robot example: *owner* agent `has(owner,beer)` desire lifecycle

6.2.2 Robot

```
/* Initial beliefs and rules */

// initially, I believe that there is some beer in the fridge
available(beer,fridge).

// my owner should not consume more than 10 beers a day :-)
limit(beer,10).

too_much(B) :-
   .date(YY,MM,DD) &
   .count(consumed(YY,MM,DD,_,_,_,B),QtdB) &
   limit(B,Limit) &
   QtdB > Limit.


/* Plans */

+!has(owner,beer)
   :  available(beer,fridge) & not too_much(beer)
   <- !at(robot,fridge);
      open(fridge);
      get(beer);
      close(fridge);
      !at(robot,owner);
      hand_in(beer);
      ?has(owner,beer);
      // remember that another beer has been consumed
      .date(YY,MM,DD); .time(HH,NN,SS);
      +consumed(YY,MM,DD,HH,NN,SS,beer).

+!has(owner,beer)
   :  not available(beer,fridge)
   <- .send(supermarket, achieve, order(beer,5));
      !at(robot,fridge). // go to fridge and wait there.

+!has(owner,beer)
   :  too_much(beer) & limit(beer,L)
   <- .concat("The Department of Health does not allow me to give
       you more than ", L,
              " beers a day! I am very sorry about that!",M);
      .send(owner,tell,msg(M)).
```

```
-!has(_,_)
   :  true
   <- .current_intention(I);
      .print("Failed to achieve goal '!has(_,_)'. Current intention
         is: ",I).

+!at(robot,P) : at(robot,P) <- true.
+!at(robot,P) : not at(robot,P)
  <- move_towards(P);
     !at(robot,P).

// when the supermarket makes a delivery, try the 'has' goal again
+delivered(beer,_Qtd,_OrderId)[source(supermarket)]
  :  true
  <- +available(beer,fridge);
     !has(owner,beer).

// when the fridge is opened, the beer stock is perceived
// and thus the available belief is updated
+stock(beer,0)
   :  available(beer,fridge)
   <- -available(beer,fridge).
+stock(beer,N)
   :  N > 0 & not available(beer,fridge)
   <- -+available(beer,fridge).

+?time(T) : true
  <-  time.check(T).
```

Listing 6.6: Domestic robot example: *robot* agent code

The robot code in this example is the most complex. A more compact representation of the robot's narrative can be visualised by selecting only events relating to desires, as depicted in the Figure 6.5.

As can be seen from the narration in the figure, the robot's desire to `has(owner, beer)` is requested by the owner via an achieved message. The robot finds a plan and commits to it. At this point, the robot performs a series of actions defined by the plan, leading to the creation of new sub-desires and intentions.

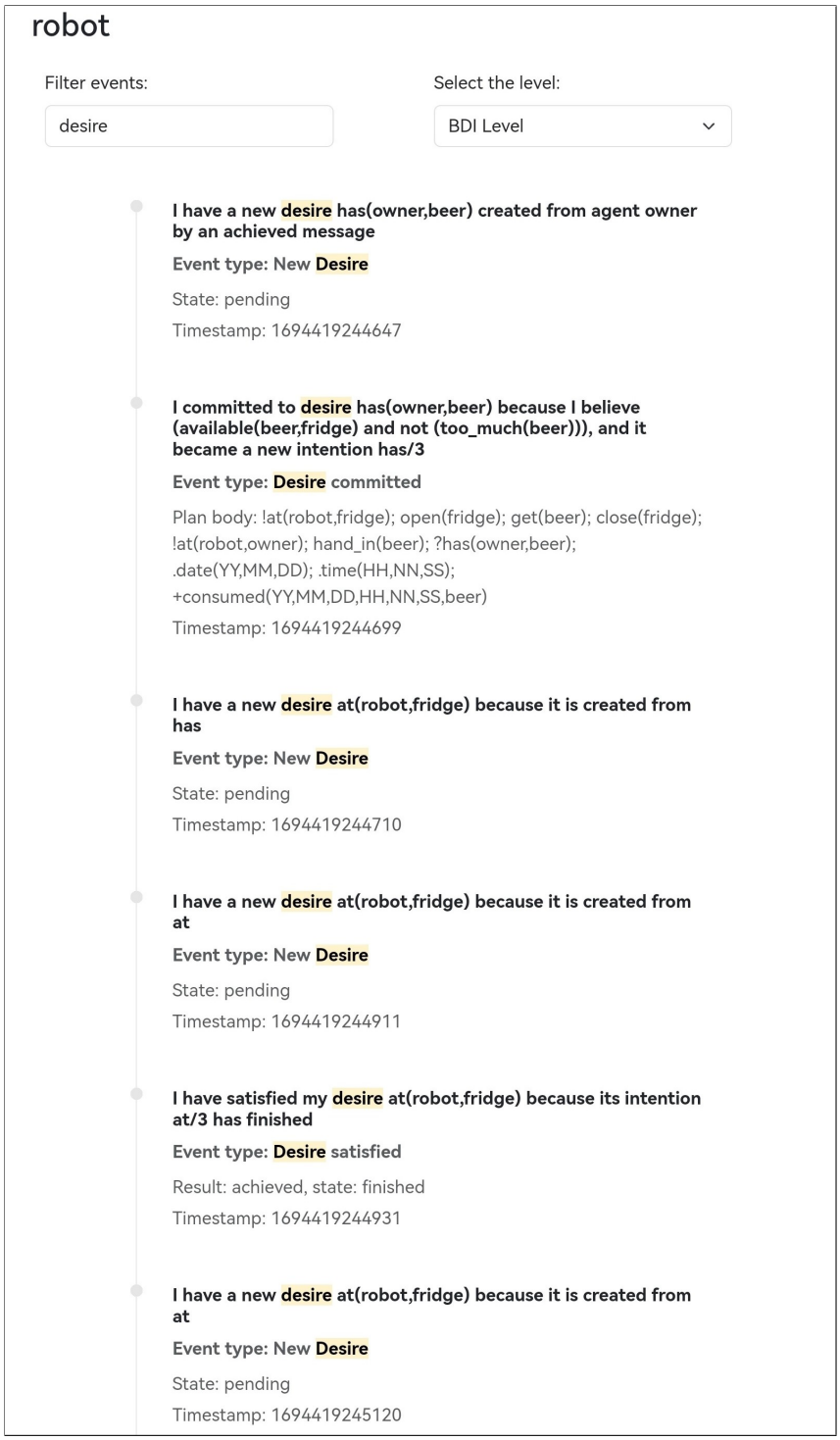

Figure 6.5: Domestic robot example: narrative based on the *robot* agent's desires

6.2.3 Supermarket

```
last_order_id(1). // initial belief

// plan to achieve the goal "order" for agent Ag
+!order(Product,Qtd)[source(Ag)] : true
  <- ?last_order_id(N);
     OrderId = N + 1;
     -+last_order_id(OrderId);
     deliver(Product,Qtd);
     .send(Ag, tell, delivered(Product,Qtd,OrderId)).
```

Listing 6.7: Domestic robot example: *supermarket* agent code

The code (see Listing 6.7) and the narrative (see Figure 6.6) for the supermarket agent are quite simple.

Initially, the supermarket agent has an initial conviction indicating the number of the last order. The agent only performs actions when someone places an order. At this point, a new desire `order(beer, 5)` is created with the specification of the type and quantity of the product. The actions of the intention consist of updating the belief on the order number to `last_order_id(2)`, delivering the product with the action `deliver` and notifying the agent of the successful delivery with a *tell* message.

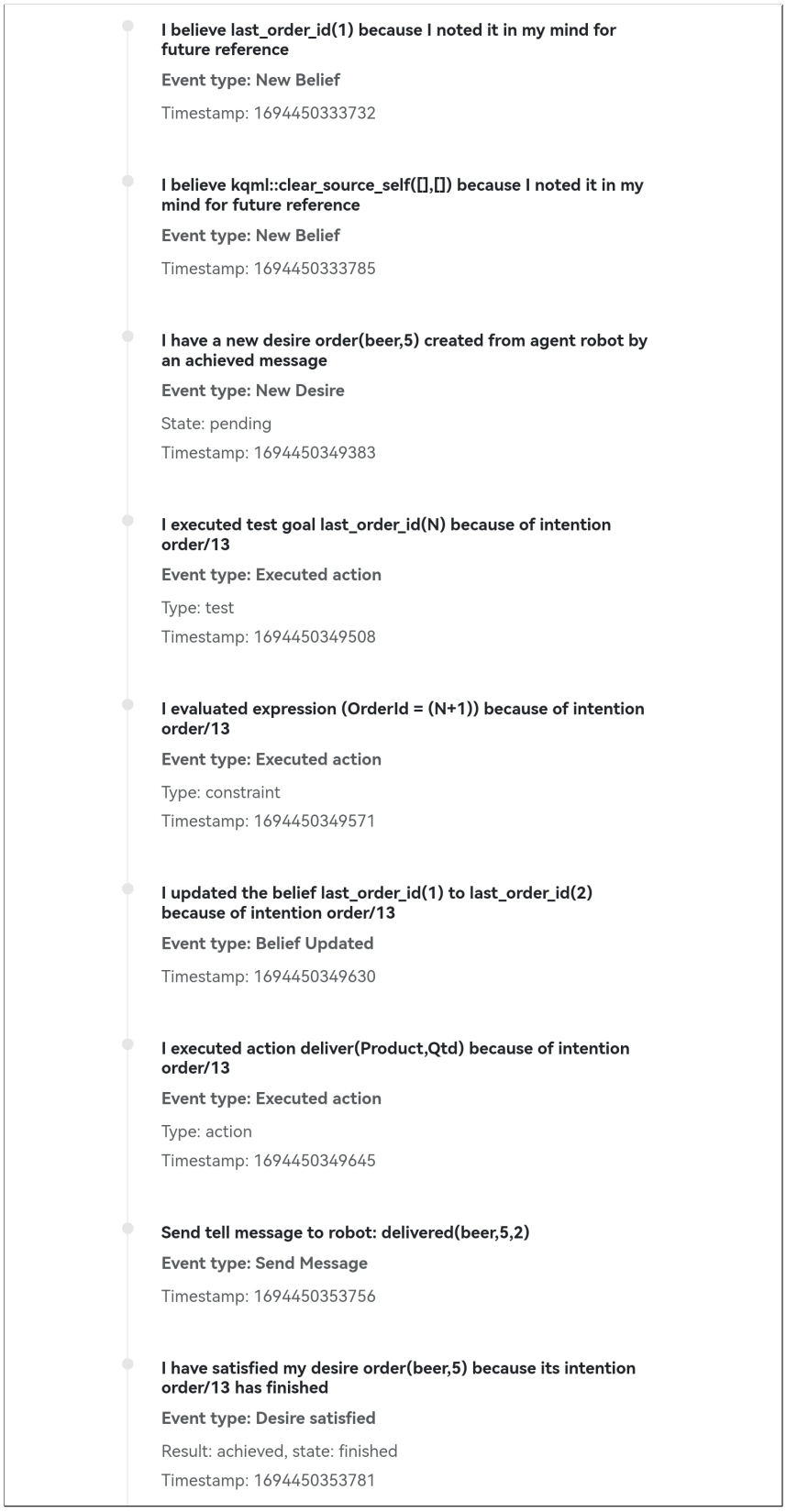

Figure 6.6: Domestic robot example: *supermarket* agent narrative

6.3 Debugging with the Explanation Tool

The implemented tool serves as an important tool to support the debugging phase for developers. Thanks to the multi-level explanation, it is possible to switch from one level to another depending on the level of detail required.

Let us see an example of its use for debugging. Let us take this piece of code from the file `robot.asl` as an example and suppose that the programmer forgot to deliver the beer to the owner (`hand_in(beer)`) as displayed below in the Listing 6.8.

```
+!has(owner,beer)
   :  available(beer,fridge) & not too_much(beer)
   <- !at(robot,fridge);
      open(fridge);
      get(beer);
      close(fridge);
      !at(robot,owner);
      //hand_in(beer);
      ?has(owner,beer);
      .date(YY,MM,DD); .time(HH,NN,SS);
      +consumed(YY,MM,DD,HH,NN,SS,beer).
```

Listing 6.8: Domestic robot example: *robot* agent code with a bug

The developer can observe that the owner asked the robot for a beer with the `get(beer)` message, but then it was not delivered and he did not drink it. The developer asks why the owner did not get the beer.

The developer at this point goes on to examine the robot's narrative after it has received the desire to take the beer (see Figure 6.7). He noted that the request to have beer was correctly delivered to the robot, the robot actually has this desire and commits to it. The first step in which the robot moves towards the fridge is done correctly, and the developer sees that it gradually creates sub-desires to move forward and that they are all fulfilled.

When the robot arrives in front of the fridge, he opens the fridge, takes the beer, closes the fridge, and moves towards the owner (see Figure 6.8).

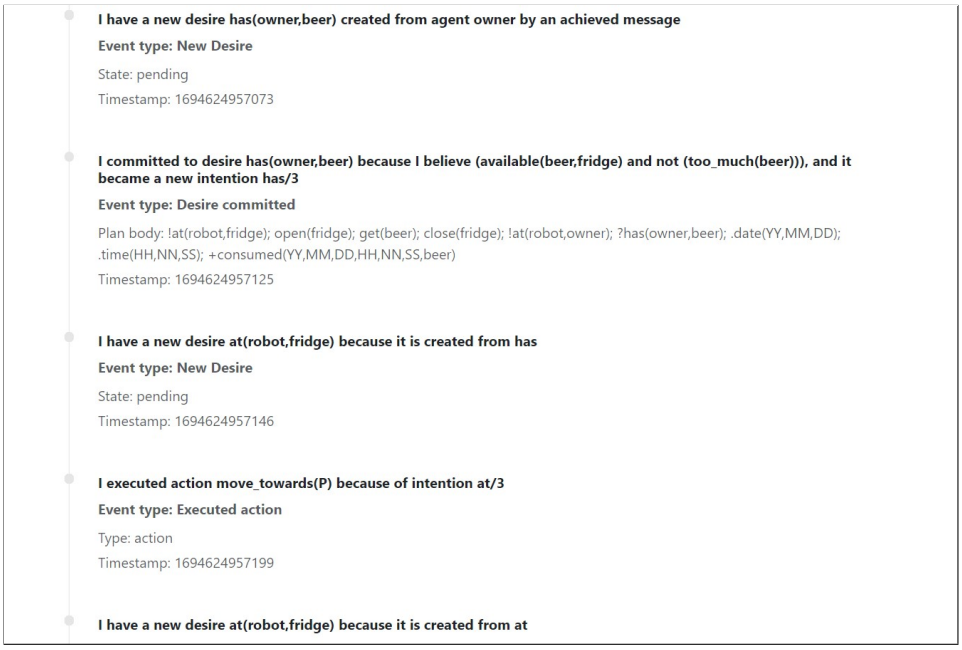

Figure 6.7: Debugging *robot* agent: narrative of the desire *has(owner, beer)*

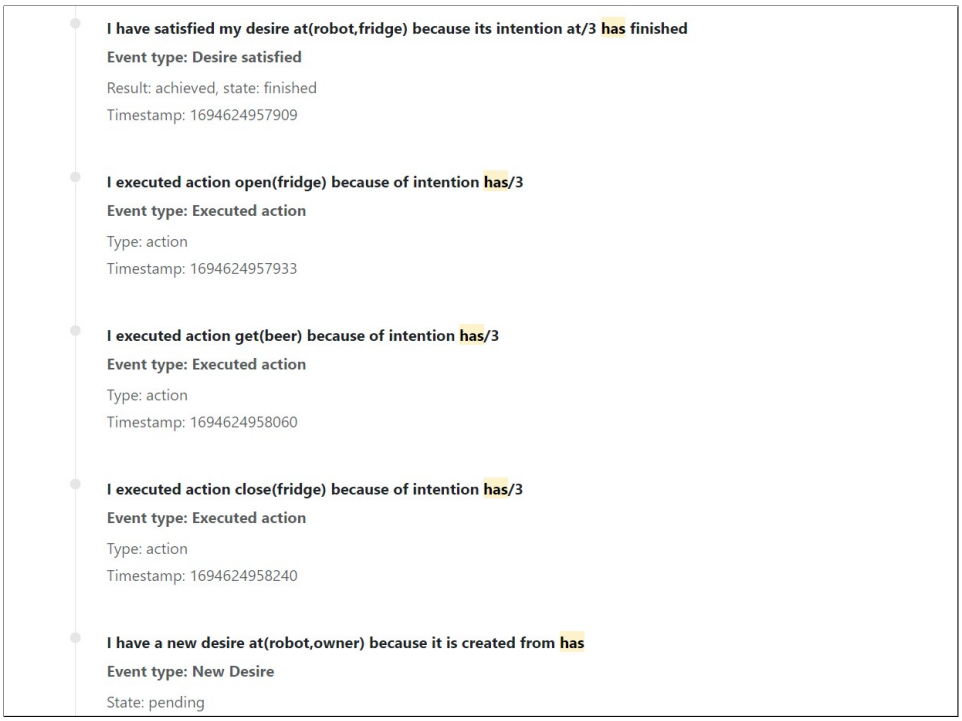

Figure 6.8: Debugging *robot* agent:narrative of the desire *has(owner, beer)*

The developer sees that all these steps have been performed correctly and are satisfied, but the next event shows that the robot gave up the desire `has(owner, beer)` because its intention failed, as illustrated in Figure 6.9.

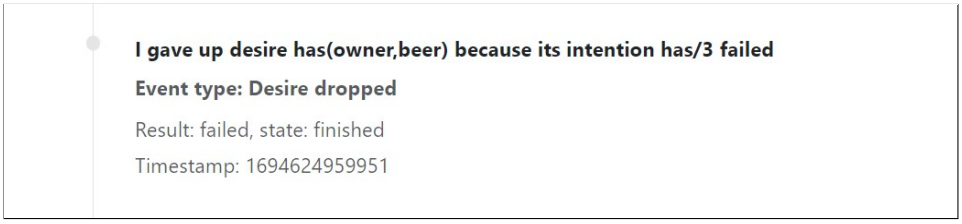

Figure 6.9: Debugging *robot* agent: desire *has(owner, beer)* dropped

In order to debug and find the reason of the failure, the developer decides to go into a lower level of detail. Moving on to the ***Jason*** level, the developer can observe that the error consists of *test_goal*, in fact, it is shown that the execution of *test_goal* `has(owner, beer)` on line 26 of the robot code failed.

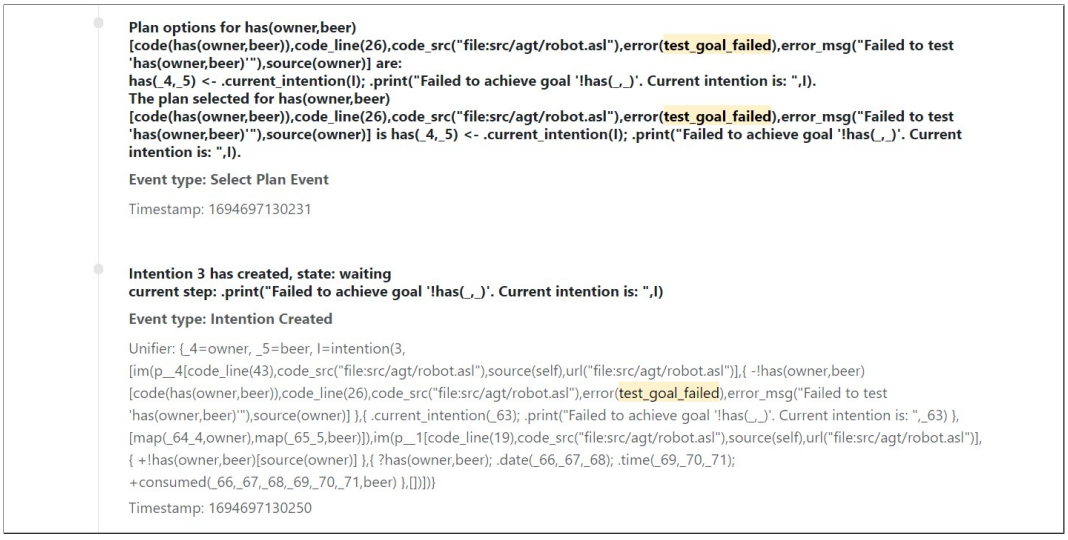

Figure 6.10: Debugging *robot* agent: test goal *has(owner, beer)* failed

Once the cause of the error has been identified, it is easier to continue with the correction of that error. In fact, the developer realises that the owner did not get the beer because it was not delivered to his hands. So there is a missing action for the robot to take, which is `hand_in(beer)`, before checking whether the owner has the beer in his hands.

Chapter 7

Conclusion and Future Work

Software is becoming more and more opaque due to its increasing complexity and autonomy. Sometimes even domain experts and system engineers struggle to understand certain aspects of a system. [44, 32]

This thesis introduces the concept of *"multi-level explainability"*, developed on the basis of multi-agent BDI systems. In particular, our aim is to develop a tool that supports various phases of agent-oriented software engineering for various stakeholders: domain experts, designers, or developers. We took cognitive agents based on the BDI (Belief-Desire-Intention) model as the reference technology because they use high-level concepts closer to human reasoning and are sometimes so complex that without a valid tool as support, it becomes difficult to understand their code and behaviour.

The research that led to this thesis was to implement a tool to provide an explanation for the behaviour of multi-agent BDI systems. In most of the works already developed in the literature, the focus is mainly on the explanation of a *single agent* that produces a *single explanation* for a *single purpose*. Our research introduces a different approach by presenting an explainability framework for agents and multi-agent systems that deals with multiple levels of abstraction that can be used with different purposes by different classes of users. The multi-level explanations facilitate communication between stakeholders and developers, as the high-level explanations can be understood by both technical and non-technical people. It also supports the debugging, testing, and validation phases for developers.

We experimented with the automatic generation of explanations starting from a BDI-based technology, and we propose that the levels of abstraction at which an explanation can be given are general and can be applied to systems that have not necessarily been designed as BDI systems.

The implemented tool is very easy to use, to generate the logs, the user only needs to specify the dependency and indicate which agents they want to

log. After execution, the user can upload the logs to the web application, and you can browse the whole multi-agent system narrative. The user is able to change the level of detail in the narrative and filter the events according to his needs. In the evaluation phase of the thesis, an example was shown of how this tool works well in various contexts, both for comprehension and for system debugging.

7.1 Future Works

In the current state, the implemented explainability tool can be used as a basis for comprehending multi-agent systems. However, we present below some valuable suggestions and possible directions for research to further improve the framework.

The first main direction of advancement involves expanding the dimension of the framework to incorporate *environmental* and *organisational* considerations, as cited in section 4.2. This expansion will involve the integration of dedicated logging components tailored to environmental artefacts and organisational specifications. The information gathered by these components could then be presented within the explanation component, enhancing the overall comprehensibility of the system and offering a complete view of the system.

In addition to expanding the dimensions regarding multi-agent systems, it would be interesting to consider expanding the explanability level as well. One useful direction is to add a *user level* to explain behaviour related to the system's domain. The explanations could include higher level *domain-related* aspects, facilitating the work of domain experts and improving comprehensibility for end users as well.

Another direction that needs to be improved in order to achieve a more robust framework is the integration of *cause-effect* relationships to sequences of events. [39] The complexity lies in identifying the relationships between the various events, *linking* them and creating causal chains. This is feasible because we use the BDI model as an abstraction since it is a logical language, and knowing its semantics, it is possible to formulate and link the causes of events. This feature further improves the debugging process by obtaining chains of explanations of *why* certain actions occur and allowing the root cause to be traced in a robust manner.

Acknowledgements

In concluding this work and coming to the end of this extraordinary journey, I would like to sincerely thank all the people who have supported and accompanied me on this path.

First of all, I truly thank Prof. Alessandro Ricci for being my supervisor both in my Bachelor's degree and now in my Master's degree. I thank him for providing me with very useful insights and valuable advice and for all the time spent discussing and developing the thesis, which helped me complete it better. His passion and dedication shown in the research and during the lectures have made me interested in this area, where I hope to continue my research going forward. I will always be extremely grateful for all the possibilities he has given me and for guiding me throughout my academic journey.

I also sincerely thank my thesis co-supervisors, Prof. Jomi Fred Hübner and Dr. Samuele Burattini for following me throughout this research project. Thanks to Prof. Jomi Fred Hübner for teaching me a lot about the Jason framework, its internal workings, and many technical and implementation aspects. I thank him for all the valuable discussions and all his ideas and insightful suggestions on the research. Also, thanks to Dr. Samuele Burattini for his support, patience, and precise and detailed comments throughout the project and thesis.

With their great availability, they helped me in the thesis development process through suggestions, critiques, and always targeted and very useful observations. I am grateful to them.

I would like to thank my friends and classmates. Thanks for their sincere encouragement and support in studying and in life and for all the beautiful memories and moments together. A special thank go to Anna, I always feel really lucky that on the first day of my Bachelor's degree, I am sitting next to her. Since that time, she has been my study and project partner, and we shared decisions, anxieties, difficulties, and most importantly, satisfaction and happiness during these five years of study in Cesena.

Naturally, I would like to express my heartfelt thanks to my family. A big thanks to my parents for supporting me in pursuing my dreams, even if they sometimes seemed crazy and irrelevant. Thanks for their love, understanding,

and always being there for me and encouraging me when I am down or lose faith. And not to forget my sister, Elisa, who is now taking her first steps into this intricate world of computer science. I wish her all the best and thank her for being by my side and for all the moments and joys spent together.

At last, I would like to thank myself. Thanks for the hard work and dedication to becoming the best version of myself. The road to the future is still long and challenging, and I want to say something to myself who has been insisting: *"Please maintain this love, even in the most difficult moments".*

Thank you to everyone. *Dal cuore, vi dico grazie.*

山水一程，感恩遇见。愿我们前路漫漫亦灿灿。

Elena Yan, September 2023

Bibliography

[1] Peter Achinstein. *The nature of explanation*. Oxford University Press, USA, 1983.

[2] Tobias Ahlbrecht. An algorithmic debugging approach for belief-desire-intention agents. *Annals of Mathematics and Artificial Intelligence*, pages 1–18, 05 2023.

[3] Aditya Ghose Ahmad Alelaimat and Hoa Khanh Dam. Mining and validating belief-based agent explanations. *EXTRAAMAS2023*, 2023.

[4] Cleber Jorge Amaral, Jomi Fred Hübner, and Timotheus Kampik. Tdd for aop: Test-driven development for agent-oriented programming. In *Proceedings of the 2023 International Conference on Autonomous Agents and Multiagent Systems*, pages 3038–3040, 2023.

[5] Sule Anjomshoae, Amro Najjar, Davide Calvaresi, and Kary Främling. Explainable agents and robots: Results from a systematic literature review. In *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems*, AAMAS '19, page 1078–1088, Richland, SC, 2019. International Foundation for Autonomous Agents and Multiagent Systems.

[6] John Langshaw Austin. *How to do things with words*, volume 88. Oxford university press, 1975.

[7] Amal Azazi, Deborah Richards, Hedieh Richards, and Samuel Mascarenhas. Belief-based agent explanations to encourage behaviour change. In *Proceedings of the 19th ACM International Conference on Intelligent Virtual Agents*, pages 176–178, 07 2019.

[8] Olivier Boissier, Rafael H Bordini, Jomi Hubner, and Alessandro Ricci. *Multi-agent oriented programming: programming multi-agent systems using JaCaMo*. Mit Press, 2020.

[9] Olivier Boissier, Rafael H. Bordini, Jomi Hübner, Alessandro Ricci, and Andrea Santi. Multi-agent oriented programming with JaCaMo. *Science of Computer Programming*, 78(6):747–761, June 2013. Special section on Agent-oriented Design Methods and Programming Techniques for Distributed Computing in Dynamic and Complex Environments.

[10] Rafael Bordini, Jomi Hübner, and Alessandro Ricci. Dimensions in programming multi-agent systems. *The Knowledge Engineering Review*, 34, 01 2019.

[11] Rafael Bordini, Jomi Hübner, and Michael Wooldridge. *Programming Multi-Agent Systems in AgentSpeak Using Jason*, volume 8. John Wiley & Sons, 10 2007.

[12] Tibor Bosse, Dung Lam, and K. Barber. Tools for analyzing intelligent agent systems. *Web Intelligence and Agent Systems*, 6:355–371, 01 2008.

[13] Dawn Branley-Bell, Rebecca Whitworth, and Lynne Coventry. User trust and understanding of explainable ai: Exploring algorithm visualisations and user biases. In Masaaki Kurosu, editor, *Human-Computer Interaction. Human Values and Quality of Life*, pages 382–399, Cham, 2020. Springer International Publishing.

[14] Michael Bratman. *Intention, Plans, and Practical Reason*. Cambridge: Cambridge, MA: Harvard University Press, 1987.

[15] Joost Broekens, Maaike Harbers, Koen Hindriks, Karel Bosch, Catholijn Jonker, and John-jules Meyer. Do you get it? user-evaluated explainable bdi agents. In *Multiagent System Technologies: 8th German Conference, MATES 2010, Leipzig, Germany, September 27-29, 2010. Proceedings 8*, volume 6251, pages 28–39. Springer, 09 2010.

[16] C3.ai. Explainability. `https://c3.ai/glossary/machine-learning/explainability/`. (Accessed on August 2023).

[17] Álvaro Carrera, Carlos A Iglesias, and Mercedes Garijo. Beast methodology: An agile testing methodology for multi-agent systems based on behaviour driven development. *Information Systems Frontiers*, 16:169–182, 2014.

[18] CArtAgO. Cartago. `https://cartago.sourceforge.net/`. (Accessed on July 2023).

[19] Larissa Chazette, Wasja Brunotte, and Timo Speith. Exploring explainability: A definition, a model, and a knowledge catalogue. In *2021 IEEE 29th International Requirements Engineering Conference (RE)*, pages 197–208, 2021.

[20] Rem Collier. Debugging agents in agent factory. In Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni, editors, *Programming Multi-Agent Systems*, pages 229–248, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[21] European Commission. Requirements of trustworthy ai — futurium — european commission. `https://ec.europa.eu/futurium/en/ai-alliance-consultation/guidelines/1.html`. (Accessed on July 2023).

[22] Mehdi Dastani, Jaap Brandsema, Amco Dubel, and John-Jules Ch. Meyer. Debugging bdi-based multi-agent programs. In Lars Braubach, Jean-Pierre Briot, and John Thangarajah, editors, *Programming Multi-Agent Systems*, pages 151–169, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[23] Daniel C Dennett. *The intentional stance*. MIT press, 1989.

[24] Louise A Dennis and Nir Oren. Explaining bdi agent behaviour through dialogue. *Autonomous Agents and Multi-Agent Systems*, 36(2):29, 2022.

[25] Oxford Dictionary. explanation noun - definition, pictures, pronunciation and usage notes — oxford advanced learner's dictionary at oxfordlearnersdictionaries.com. `https://www.oxfordlearnersdictionaries.com/definition/english/explanation?q=explanation`. (Accessed on 09/06/2023).

[26] Erdem Eser Ekinci, Ali Murat Tiryaki, Övünç Çetin, and Oguz Dikenelli. Goal-oriented agent testing revisited. In Michael Luck and Jorge J. Gomez-Sanz, editors, *Agent-Oriented Software Engineering IX*, pages 173–186, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[27] Jomi Fred Hübner. Agent oriented programming with jason. `https://jomi.das.ufsc.br/mas/slides/slides-aop.pdf`. (Accessed on August 2023).

[28] Jomi Fred Hübner, Jaime Simão Sichman, and Olivier Boissier. Moise tutorial (for moise 0.7). `https://moise.sourceforge.net/doc/tutorial.pdf`. Accessed on May 2023.

[29] Alma M. Gómez-Rodríguez and Juan C. González-Moreno. Comparing agile processes for agent oriented software engineering. In *Product-Focused Software Process Improvement*, pages 206–219, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[30] Juan C. González-Moreno, Alma Gómez-Rodríguez, Rubén Fuentes-Fernández, and David Ramos-Valcárcel. *INGENIAS-Scrum*, pages 219–251. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[31] Shirley Gregor and Izak Benbasat. Explanations from intelligent systems: Theoretical foundations and implications for practice. *MIS Q.*, 23(4):497–530, dec 1999.

[32] David Gunning. Broad agency announcement explainable artificial intelligence (xai). *Defense Advanced Research Projects Agency (DARPA), Tech. Rep.*, 2016.

[33] Maaike Harbers, Karel van den Bosch, and John-Jules Meyer. Design and evaluation of explainable bdi agents. In *2010 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology*, volume 2, pages 125–132, 2010.

[34] Koen V. Hindriks. Debugging is explaining. In Iyad Rahwan, Wayne Wobcke, Sandip Sen, and Toshiharu Sugawara, editors, *PRIMA 2012: Principles and Practice of Multi-Agent Systems*, pages 31–45, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[35] ISO ISO. Iso/iec/ieee international standard - systems and software engineering–vocabulary. *ISO/IEC/IEEE 24765:2017(E)*, pages 1–541, 2017.

[36] JaCaMo. Jacamo project — multi-agent programming framework. `https://jacamo.sourceforge.net/`. (Accessed on May 2023).

[37] Jason. Jason. `https://github.com/jason-lang/jason/tree/master`. (Accessed on August 2023).

[38] Jason. Jason — a java-based interpreter for an extended version of agentspeak. `https://jason.sourceforge.net/wp/`. (Accessed on July 2023).

[39] Shakil M Khan and M Rostamigiv. On explaining agent behaviour via root cause analysis: A formal account grounded in theory of mind. In *Proceedings of the 26th European Conference on Artificial Intelligence ECAI*, pages 30–09, 2023.

[40] Mark Klein and Chrysanthos Dellarocas. Exception handling in agent systems. *Proceedings of the International Conference on Autonomous Agents*, 03 2000.

[41] Amy J. Ko and Brad A. Myers. Debugging reinvented: Asking and answering why and why not questions about program behavior. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, page 301–310, New York, NY, USA, 2008. Association for Computing Machinery.

[42] Vincent J. Koeman, Koen V. Hindriks, and Catholijn M. Jonker. Omniscient debugging for cognitive agent programs. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pages 265–272, 2017.

[43] Robert Kowalski and Fariba Sadri. From logic programming towards multi-agent systems. *Annals of Mathematics and Artificial Intelligence*, 25(3–4):391–419, feb 1999.

[44] Maximilian A. Köhl, Kevin Baum, Markus Langer, Daniel Oster, Timo Speith, and Dimitri Bohlender. Explainability as a non-functional requirement. In *2019 IEEE 27th International Requirements Engineering Conference (RE)*, pages 363–368, 2019.

[45] D. N. Lam and K. S. Barber. Comprehending agent software. In *Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems*, AAMAS '05, page 586–593, New York, NY, USA, 2005. Association for Computing Machinery.

[46] Avleen Malhi, Samanta Knapic, and Kary Främling. Explainable agents for less bias in human-agent decision making. In Davide Calvaresi, Amro Najjar, Michael Winikoff, and Kary Främling, editors, *Explainable, Transparent Autonomous Agents and Multi-Agent Systems*, pages 129–146, Cham, 2020. Springer International Publishing.

[47] Moise. Moise organisational model. `https://moise.sourceforge.net/`. (Accessed on July 2023).

[48] Allen Newell. The knowledge level. *Artificial intelligence*, 18(1):87–127, 1982.

[49] Guillaume Pothier and Éric Tanter. Back to the future: Omniscient debugging. *IEEE Software*, 26:78–85, 11 2009.

[50] David Poutakidis, Lin Padgham, and Michael Winikoff. Debugging multi-agent systems using design artifacts: The case of interaction protocols. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems: Part 2*, AAMAS '02, page 960–967, New York, NY, USA, 2002. Association for Computing Machinery.

[51] David Poutakidis, Lin Padgham, and Michael Winikoff. An exploration of bugs and debugging in multi-agent systems. In Ning Zhong, Zbigniew W. Raś, Shusaku Tsumoto, and Einoshin Suzuki, editors, *Foundations of Intelligent Systems*, pages 628–632, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[52] Anand Srinivasa Rao and Michael P. Georgeff. Bdi agents: From theory to practice. In *International Conference on Multiagent Systems*, 1995.

[53] Sebastian Rodriguez, John Thangarajah, and Michael Winikoff. User and system stories: An agile approach for managing requirements in aose. In *Proceedings of the 20th International Conference on Autonomous Agents and MultiAgent Systems*, AAMAS '21, page 1064–1072, Richland, SC, 2021. International Foundation for Autonomous Agents and Multiagent Systems.

[54] Sebastian Rodriguez, John Thangarajah, and Michael Winikoff. A behaviour-driven approach for testing requirements via user and system stories in agent systems. In *Proceedings of the 2023 International Conference on Autonomous Agents and Multiagent Systems*, pages 1182–1190, 2023.

[55] Sebastian Rodriguez, John Thangarajah, Michael Winikoff, and Dhirendra Singh. Testing requirements via user and system stories in agent systems. In *Proceedings of the 21st International Conference on Autonomous Agents and Multiagent Systems*, pages 1119–1127, 2022.

[56] John R Searle. *Speech acts: An essay in the philosophy of language*, volume 626. Cambridge university press, 1969.

[57] Yoav Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, 1993.

[58] Ömer Uludağ, Matheus Hauder, Martin Kleehaus, Christina Schimpfle, and Florian Matthes. Supporting large-scale agile development with domain-driven design. In Juan Garbajosa, Xiaofeng Wang, and Ademar Aguiar, editors, *Agile Processes in Software Engineering and Extreme*

Programming, pages 232–247, Cham, 2018. Springer International Publishing.

[59] Yves Wautelet, Samedi Heng, Soreangsey Kiv, and Manuel Kolp. User-story driven development of multi-agent systems: A process fragment for agile methods. *Computer Languages, Systems & Structures*, 50:159–176, 2017.

[60] Michael Winikoff. Debugging agent programs with why? questions. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*, AAMAS '17, page 251–259, Richland, SC, 2017. International Foundation for Autonomous Agents and Multiagent Systems.

[61] Michael Winikoff, Galina Sidorenko, Virginia Dignum, and Frank Dignum. Why bad coffee? explaining bdi agent behaviour with valuings (extended abstract). In Lud De Raedt, editor, *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI-22*, pages 5782–5786. International Joint Conferences on Artificial Intelligence Organization, 7 2022. Journal Track.

[62] Michael Wooldridge. *Introduction to MultiAgent Systems*. John Wiley and Sons, 2002.

[63] Michael Wooldridge and Nicholas R. Jennings. Intelligent agents: theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.

[64] Feiyu Xu, Hans Uszkoreit, Yangzhou Du, Wei Fan, Dongyan Zhao, and Jun Zhu. Explainable ai: A brief survey on history, research areas, approaches and challenges. In *Natural Language Processing and Chinese Computing: 8th CCF International Conference, NLPCC 2019, Dunhuang, China, October 9–14, 2019, Proceedings, Part II 8*, pages 563–574. Springer, 2019.