Extending BDI Agents to Model and Reason with Uncertainty

By

AIDAN SCANNELL

Supervisors:
Professor WEIRU LIU
Dr KEVIN MCAREAVEY

A dissertation submitted to the University of Bristol and the University of the West of England in accordance with the requirements of the degree of MASTER OF RESEARCH in Robotics and Autonomous Systems.

SEPTEMBER 2018

Dedication and Acknowledgements

I would like to acknowledge my supervisors, Professor Weiru Liu and Dr Kevin McAreavey for their continued support and guidance throughout the project. I would also like to thank Dr Kim Bauters for all of his help and contributions.

This work was supported by the EPSRC Centre for Doctoral Training in Future Autonomous and Robotic Systems (FARSCOPE) at the Bristol Robotics Laboratory.

Author's declaration

This Masters level study was completed as part of the FARSCOPE Centre for Doctoral Training at the University of Bristol and the University of the West of England, Bristol. The work is my own. Where the work of others is used or drawn on it is attributed

SIGNED: ..... DATE: .....

Word count: 21883

Abstract

This work attempted to extend BDI agents, in particular AgenSpeak(L) agents, to model and reason with uncertain information. The extended AgentSpeak(L) language uses epistemic states to model an agent's uncertain beliefs about the world. The implemented extended AgentSpeak(L) language effectively allows agents to model and reason with uncertainty in a computationally efficient manner. Agents are capable of modelling their beliefs as either probabilistic or possibilistic compact epistemic states and the implementation of epistemic states provides a base for easily implementing instantiations of different uncertainty theories, e.g. Demspter-Shafer theory.

A language $\mathscr{L}_{\geq}$ for constructing formulas that can reason over these uncertain beliefs is detailed. This language is capable of forming sentences of the form: ϕ is more plausible than ψ ($\phi > \psi$). This language is used to construct agent's plan contexts and test goals, which were previously just a conjunction of literals. This provides the agents with extended reasoning capabilities. The agent's belief base was modelled as a Global Uncertain Belief (GUB), acting as a set of formulas from the language $\mathscr{L}_{\geq}$. This enables agents to select applicable plans from the set of relevant plans by querying if a logical formula is entailed by the GUB.

A development environment for defining and simulating MASs written in the extended AgentSpeak(L) language is introduced. A simulation environment that a programmer can easily extend to a specific scenario is detailed. It uses a multi-threaded approach to enable multiple agents to run on a single machine and ensures that agents act on and perceive an environment without any thread interference or memory inconsistency issues. All of the work in this project is then brought together in a mars exploration MAS that demonstrates the extended modelling and reasoning capabilities as well as the power of the development environment as a whole.

Accompanying code: *https://github.com/aidanscannell/uncertain-agentspeak*.

Table of Contents

LIST OF TABLES

TABLE **Page**

List of Figures

Figure **Page**

CHAPTER 1

INTRODUCTION

Multi-agent systems (MAS) are computer systems which allow multiple decision-making entities, known as agents, to interact with one another to achieve their predefined goals. These systems are well suited for environments that dynamically change. This is due to their ability to respond and interact with their environment to make appropriate decisions based on their current state (Jennings, Sycara, and Wooldridge 1998).

Intelligent autonomous systems offer great potential, however, the development of systems that are robust enough to operate in the real-world is extremely difficult. There is currently a lack of frameworks for modelling and reasoning with uncertainty. Traditional Belief-Desire-Intention (BDI) architectures for designing and developing intelligent agent systems are less adequate for modelling uncertain knowledge, beliefs and evidence, whilst the real-world is almost always pervaded with uncertainty (Kwisthout and Dastani 2006).

Most systems require perfect information in order to operate successfully. In the highly competitive business world, systems need to be capable of handling complex scenarios where this is not the case. In real-world scenarios the environment may be dynamic, pervaded with uncertainty and only provide partial information. The frequency with which these systems are required to change is also increasing, resulting in architectures and languages that need to provide reduced complexity as well as specification and modification time (M. Georgeff et al. 1999). Software agents, particularly Belief-Desire-Intention (BDI) agents, provide the essential tools for dealing with the real world.

There has been significant research in the Multi-Agent System (MAS) field and many logic-based approaches for representing and reasoning about uncertainty have been proposed. However, there is currently a lack of implementations and frameworks that allow for combining multiple approaches when implementing a system (Kern-Isberner and Lukasiewicz 2017).

These form the key motivations for conducting research into a development environment for defining and testing BDI agents that are capable of modelling and reasoning with uncertainty.

1.1 Background

Architectures such as the Belief-Desire-Intention (BDI) architecture (Bratman 1987) allow dynamic knowledge and beliefs to be explicitly modelled through decomposing a complex problem into a set of autonomous and interacting agents. BDI agents consist of their own 'Beliefs', 'Desires' and 'Intentions'; beliefs model the agent's understanding of the environment, desires represent the states that the agent wants to achieve and intentions are the desires that the agent has acted upon. Using these, the agents are able to respond accordingly to the situations that they find themselves in. In recent years, Belief-Desire-Intention (BDI) agents have received a great deal of interest due to their potential for replacing current methods for analysing, designing and implementing complex, large-scale intelligent systems (Herzig et al. 2017).

Many agent-based programming languages based on the BDI architecture have been proposed over the years, including Procedural Reasoning System (PRS) (Ingrand, M. P. Georgeff, and Rao 1992), AgentSpeak(L) (Rao 1996) and A Practical Agent Programming Language (2APL) (Dastani 2008). Although these languages have successfully been used to model some modern systems, they are not well-suited to model more complex systems as they are not able to model or reason about uncertain information. In most real-world scenarios agent's beliefs do not take the form of binary true or false values but rather are uncertain. Uncertainty can arise from many different sources, for example, sensor noise or incomplete information. The computational complexity of uncertainty theories extends this issue further as BDI agents rely on reactive behaviour and most uncertainty theories do not consider tractability.

Uncertain input can be dealt with in different ways: (i) it can act as a constraint that must be satisfied after belief revision, or (ii) it can be treated as a new belief with an associated weight. Ma and Liu (2011a) proposed a framework for dealing with uncertainty where new information from multiple sources is used to strengthen or weaken existing beliefs and not necessarily cancel out existing beliefs. However, this framework relies on semantic belief change operators, restricting it's practical applicability due to it's computational cost. There have been approaches suggested that use syntactic operators, however, these are limited to classical beliefs (Alchourron, Gardenfors, and Makinson 1985; Nebel 1995). There are far less approaches that use syntactic operators for iterated belief revision. Ordinal Conditional Functions (OCF) are a popular approach to defining epistemic states (Spohn 1988) and Williams (1995) proposed a syntactic representation of OCF. However, this representation is not a general framework and leads to issues when attempting to instantiate into different uncertainty theories. Bauters et al. (2017) propose an approach for managing different sources of uncertainty in a BDI framework. They offer a new approach for modelling the beliefs of an agent as well as a novel syntactic approach

for revising beliefs with uncertain information. This paper offers the theoretical foundations and the base for a lot of the work presented here.

There are many implementations of the AgentSpeak(L) agent programming language (Rafael H. Bordini and Jomi F. Hübner 2006; Machado and R. Bordini 2003a). Jason has become an extremely popular development environment for defining and simulating AgentSpeak(L) agents. Jason provides an interpreter for AgentSpeak(L) agents extended with speech acts as well as an environment for simulating the interactions of multiple agents with each other and an environment. The lack of tractable syntactic operators for belief revision with uncertain information has left a significant gap for AgentSpeak(L) interpreters and environments extended to model and reason with uncertain information.

1.2 Research Challenges

The primary research questions that this work seeks to address are as follows:

1. How can an agent-based programming language extended to model and reason with uncertain information be implemented as a scalable BDI framework?

2. How should an agent's belief base be implemented and how should it reason about uncertain information? What impact does it have on the agent's beliefs and reasoning capabilities?

3. How can a general simulation environment capable of hosting "extended" agents be developed? How can application specific environments be developed as an extension of such a general environment?

4. How can systems be implemented in such a framework and how do they compare to implementations in existing frameworks? What are the advantages and limitations of an extended framework?

5. How can a new BDI framework be evaluated with regards to existing frameworks?

1.2.1 Aims and Objectives

The main aims and objectives are detailed below.

A.1 Implement the AgentSpeak(L) programming language.

- **O.1.1** Study the AgentSpeak(L) agent programming language.
- **O.1.2** Implement the underlying data types of AgentSpeak(L).
- **O.1.3** Implement the AgentSpeak(L) interpreter and required mechanisms.
- **O.1.4** Create a parser that can read files and instantiate AgentSpeak(L) agents.

A.2 Extend this implementation for modelling and reasoning with uncertain information.

O.2.1 Study uncertainty modelling in relation to BDI agents.

O.2.2 Extend this implementation to incorporate modelling of uncertain beliefs.

O.2.3 Extend this further to incorporate reasoning about such uncertain beliefs.

A.3 Develop a platform based on the extended language for designing, simulating and testing Multi-Agent System (MAS)s.

O.3.1 Develop a system that can simulate multiple "extended" agents in a user defined simulation environment.

O.3.2 Develop a general simulation environment that can host "extended" agents.

A.4 Compare the performance of the extended language with the original.

O.4.1 Construct an example scenario that demonstrates the performance of the extended framework.

O.4.2 Develop the required agents in the extended language.

O.4.3 Implement the example scenario's simulation environment by extending the platforms base environment.

O.4.4 Simulate and analyse the performance of the extended language in the example scenario.

1.3 Summary

The rest of this document will be outlined as follows; a review of relevant literature is presented in Chapter 2, first introducing the BDI architecture and detailing the AgentSpeak(L) language. It will then introduce probability theory and possibility theory, including how uncertainty has previously been integrated into the BDI architecture. This chapter will finish with a brief overview of the relevant MAS development tools that are currently being used by researchers. Chapter 3 then introduces and discusses the research methodology behind implementing the extended AgentSpeak(L) language. Chapter 4 then details the development environment that was created, including the simulation environment and the configuration and running of MASs. Chapter 5 then discusses the work presented in this dissertation, relating it back to the initial research objectives and finally, Chapter 6 will conclude the dissertation.

CHAPTER 2

LITERATURE REVIEW

This chapter provides a comprehensive review of the work related to this dissertation. It will begin with an introduction to the BDI architecture, including a comparison of the most popular BDI agent-based programming languages. After selecting AgentSpeak(L) as the best language to be extended with uncertainty modelling capabilities, AgentSpeak(L) itself will be reviewed in detailed.

This is followed by a comprehensive introduction to well-known uncertainty theories used to model uncertain information and knowledge across various domains, including, probability theory and possibility theory. Particular focus is given to their practical applicability in MAS, in terms of their complexity and resulting scalability. The next section details the state-of-the-art in BDI uncertainty modelling, where particular focus is given to the tractability of these frameworks.

The next topic of this literature review covers existing MAS platforms, which are defined here as tools that provide a user with the ability to design and test MASs. As the AgentSpeak(L) programming language has been selected for this work, this section will mainly focus on platforms that are based upon the AgentSpeak(L) language.

2.1 Belief Desire Intention (BDI) Architecture

Within the research community, the notion of an intelligent agent, which appears to be the subject of it's beliefs, desires, commitments and other mental attitudes is widely accepted. The philosopher Dennett coined the term intentional agent to define such systems (Clement 1987).

Many different logics have been proposed in order to formalise intentional systems, amongst these are theory of intentions (Cohen and Levesque 1990) and BDI logic (Rao and M. P. Georgeff 1991). However, logical agents cannot be developed using ad-hoc logical languages but instead require programming in executable languages.

The BDI architecture is a computational model that has gained much interest to fill this gap. Figure 2.1 illustrates the basic architecture. As it's name suggests, BDI agents consist of:

- **Beliefs**: representing the agents knowledge about the world,
- **Desires**: states of the world that the agent wants to bring about,
- **Intentions**: the actions currently under execution for achieving the agent's desires.

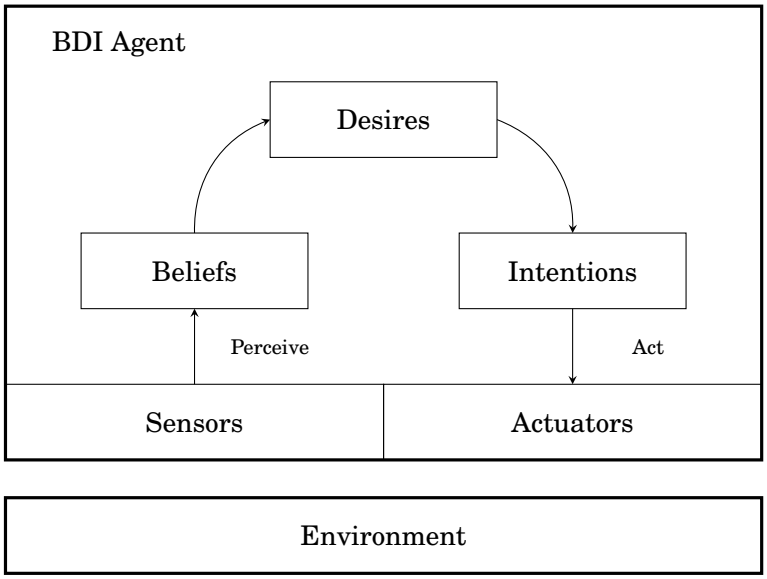


Figure 2.1: Belief-Desire-Intention (BDI) agent architecture.

As well as these components agents also consist of a predefined plan library and event queue. The plan library consists of plans representing the procedural knowledge of the agent. The event queue consists of external events perceived from the environment, internal events generated by the agent (to update it's belief base) and internal subgoals (to help achieve it's desires).

A typical BDI interpretation cycle is detailed below:

1. Observe the agents environment and internal state and update the event set accordingly.
2. Select the relevant plans whose triggering event matches an event in the event set. Select the applicable plans whose preconditions are also satisfied.
3. Select one applicable plan to execute.
4. Put the new plan either into a new or existing intention stack.
5. Select an intention and execute the next step from the top plan. If it is an action then perform it and if it is a subgoal add it to the event set.

There are many BDI implementations and agent programming languages, each with their own strengths and weaknesses. There are too many to review here but the reader can be

referred to (Kravari and Bassiliades 2015a) and (Mascardi, Demergasso, and Ancona 2005), providing a survey on BDI agent programming languages and a survey on general agent platforms respectively.

AgentSpeak(L) is a theoretical agent-based programming language that is an extension of logic programming for the BDI architecture. It has gained much interest within the MAS research community with popular implementations such as Jason (Rafael H. Bordini, Jomi Fred Hübner, and Wooldridge 2007). AgentSpeak(L) retains the most important aspects of BDI-based reactive planning systems, which makes it particularly interesting (Á. F. Moreira, Vieira, and Rafael H. Bordini 2004). It's relation to BDI logics and formal semantics are also being widely studied (A. F. Moreira and Rafael H. Bordini 2002; Rafael H. Bordini and Á. F. Moreira 2002; Rafael H. Bordini and Á. F. Moreira 2004).

2.2 AgentSpeak(L)

(Rao 1996) provide an operational and proof-theoretic semantics of a language AgentSpeak(L), an abstract language used for describing and programming BDI agents. It is an extension of logic programming for the BDI architecture, which is the predominant approach for implementing intelligent agents (Andrew 2001).

An AgentSpeak(L) agent is defined by specifying a set of base beliefs, known as the belief base and a set of plans, known as the agent's plan library. A belief atom takes the form of a first-order predicate. Each belief from the belief base consists of a belief atom or it's negation, which are known as belief literals.

Agents define two types of goals, test goals and achievement goals:

- **Achievement Goals**: Achievement goals are predicates prefixed with the achievement operator '!'. They represent the states of the world (where the predicate evaluates to true) that the agent wants to achieve. Achievement goals are usually used to trigger subplans.

- **Test Goals**: Test goals are predicates prefixed with the test operator '?'. Agents use test goals to check whether or not a predicate unifies with it's belief base, returning a unifier or failing otherwise.

Agents maintain an event queue that is used to instantiate plans. Events can take the form of the addition $'+'$ or deletion $'-'$ of both beliefs and goals. These events can be either external or internal events. External events are generated from belief updates resulting from the agents perception of it's environment. Internal events are generated when a subgoal needs to be achieved.

Each plan consists of three components:

- **Triggering Event**: Triggering events define which events make a plan relevant.

- **Context**: The plan context consists of a conjunction of belief literals. If the context is a logical consequence of the agent's belief base then the plan becomes applicable.
- **Plan Body**: The plan body consists of a sequence of basic actions or (sub)goals that the agent must achieve (or test) if the plan is applicable and chosen for execution.

Figure 2.2 shows some examples of AgentSpeak(L) beliefs, goals and plans. The initial beliefs tell us that the agent is currently at location(1) and that there is water at location(2). The agent has an initial goal to find water, which is added to the event set. The $+!findWater$ goal is then selected from the event set and used to select the relevant plans (both of the $+!findWater$ plans are relevant). The plan contexts are then evaluated resulting in the variable A unifying as A/2 due to the belief $water(location(2))$. As the agent is not at location(2) the second plan context is not a logical consequence of the belief base and thus will fail. The first plan's context, however, is a logical consequence and therefore is an applicable plan, returning the unifier A/2. This plan is then selected for execution and the subgoal $+!move(location(2))$ will be added to the event set. This will trigger the $+!move(location(A))$ plan resulting in the environment action $travel(location(1),location(2))$. The agent will now be at location(2) and will update it's belief base accordingly. The second action from the first plan $+!findWater$ will now be added to the event set, leading to both of the $+!findWater$ plans becoming relevant. This time, however, only the second plan's context is a logical consequence of the belief base as the agent is now at location(2). The agent will then perform the environment action $sampleWater(location(2))$.

```
1  // initial beliefs
2  water(location(2)).
3  at(location(1)).
4
5  // initial goals
6  +!findWater.
7
8  // plan library
9  +!findWater : water(location(A)) &  ~at(location(A)) <- +!move(location(A)); +!findWater.
10 +!findWater : water(location(A)) & at(location(A)) <- sampleWater(location(A)).
11 +!move(location(A)) : at(location(B)) <- travel(location(B), location(A)).
```

Figure 2.2: Example of AgentSpea(L) beliefs, goals and plans

2.2.1 AgentSpeak(L) Syntax

The language for specifying AgentSpeak(L) agents is shown by the grammar in Figure 2.3.

An AgentSpeak(L) agent is defined as a set of beliefs bs (the belief base) and a set of plans ps (the plan library). The atomic formulae of the language at are predicates with predicate symbol P and standard terms of first-order logic $t_1...t_n$. All atomic formulae at in the belief base must be grounded, i.e. they cannot contain variables.

A plan p is composed of three components: the triggering event te, the context ct and a sequence of actions, goals or belief updates h. The triggering event and context $te : ct$ are referred

```
ag  ::=  bs ps
bs  ::=  at1. . . . atn.
at  ::=  P(t1,...,tn)
ps  ::=  p1 ... pn
p   ::=  te : ct <- h .
te  ::=  +at | -at | +g | -g
ct  ::=  true | l1&...&ln
h   ::=  true | f1;...;fn
l   ::=  at | not at
f   ::=  A(t1,...,tn) | g | u
g   ::=  !at | ?at
u   ::=  +at | -at
```

Figure 2.3: AgentSpeak(L) grammar (Rafael H Bordini and Jomi F Hübner 2007)

to as the head of the plan and h is referred to as the body of the plan. The triggering event te can be the addition or deletion of a belief atom or goal ($+at$, $-at$, $+g$, $-g$). The plan context ct is a conjunction of belief literals $l_1 \wedge ... \wedge l_n$ where belief literals l are either a belief atom at or it's negation $\neg at$.

It is assumed that an agent has a predefined set of actions that it can perform. Each action is defined by an action symbol A which takes standard first-order logic terms as arguments. There are two types of goals, achievement goals $!at$ and test goals $?at$. Finally, there are also two belief updates u, these are the addition and deletion of beliefs ($+at$ and $-at$ respectively).

2.2.2 Informal Semantics

The AgentSpeak(L) interpreter also manages a set of events, a set of intentions and utilises three selection functions in order to operate.

- S_ϵ - The event selection function is used to select a single event from the set of events. This is the event that the agent has chosen to act upon.
- S_O - The option selection function selects a single plan from the set of applicable plans.
- S_I - The intention selection function selects a single intention from the set of intentions.

The selection functions are not defined in AgentSpeak(L) and instead are left to be designed on a per agent basis (i.e. they should be agent specific).

Intentions are stacks of partially instantiated plans and represent the "course of action" an agent has committed to in order to respond to a given event. Events can lead to the execution of plans and can be either external or internal. External events arise due to an agent's perception of it's environment (e.g. the addition of a new belief) and internal events are generated from the execution of a plan (e.g. the addition of a new achievement goal from a subgoal within a plan). When internal events trigger applicable plans they are added to the top of the intention that

generated them, whereas external events lead to the creation of new intentions, representing a new focus for the agents acting in the environment.

Figure 2.4 shows the interpretation cycle for an AgentSpeak(L) agent, where rectangles represent sets (e.g. belief, event, plan and intention sets), diamonds represent selection (of a single element from a set) and circles represent more complex processing required for interpreting agent programs (Machado and R. Bordini 2003b).

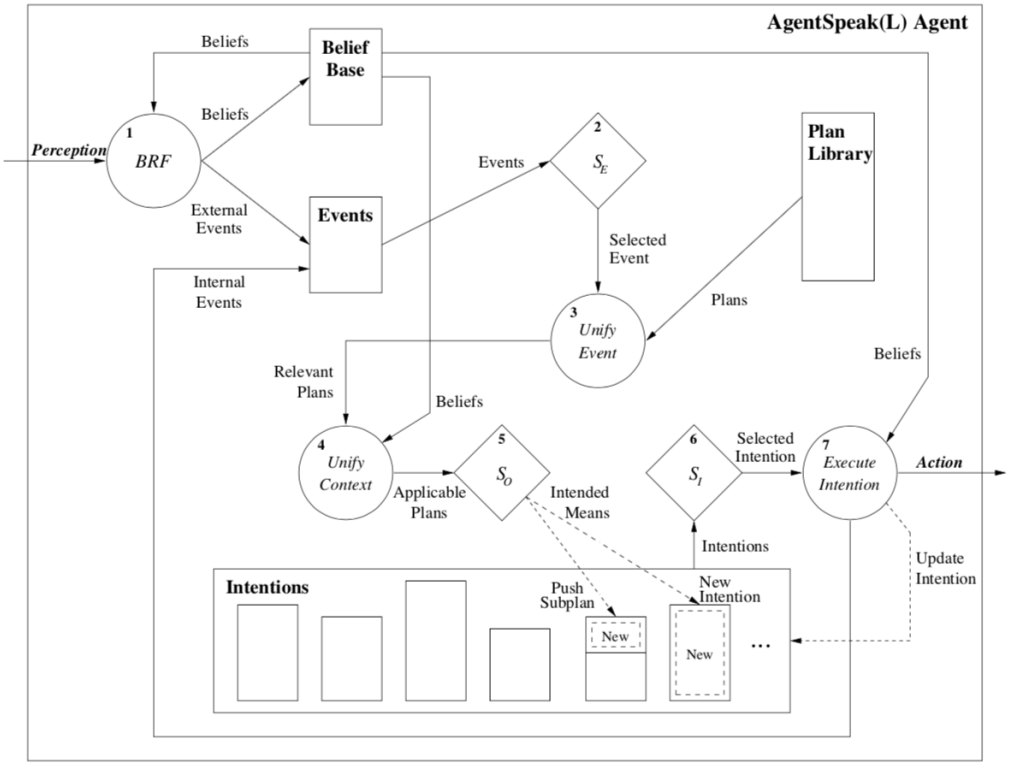


Figure 2.4: AgentSpeak(L) interpreter (Machado and R. Bordini 2003b)

At every interpretation cycle an agent updates the list of events. The belief base is updated through the agent's perception of it's environment and every time it is updated a corresponding event is added to the event set. A description of an AgentSpeak(L) interpretation cycle is given below:

1. The event selection function selects a single event from the event set.

2. The interpreter then attempts to unify this event with the triggering events of each plan in the plan library. These plans are known as the relevant plans.

3. Next, for each relevant plan the agent checks whether the plan's context is a logical consequence of the belief base. This returns a set of applicable plans, each with their own

unifiers. These are plans that can be used at the current time to deal with the selected event.

4. The "option" selection function S_O selects one of the applicable plans to become the intended means for the selected event. If the event is an internal event the intended means is added to the top of the current intention and if it is an external event then a new intention is created.

5. Next, the intention selection function s_I selects a single intention (a stack of partially instantiated plans) from the set of intentions.

6. At the top of this intention there is a plan and it's body is taken forward for execution.

7. The formulas in the plan body are then executed one-by-one. These may take the form of the agent performing an action on it's environment, generating an internal event (when it is an achievement goal) or performing a test goal.

   a) If the agent performs an action or a test goal then the intention set needs to be updated.

   b) If an action is performed then it is removed from the intention and the relevant components of the agent that control the action are informed.

   c) If a test goal is to be performed then the agent queries the belief base to find a belief atom that unifies with the test goal. If it succeeds then the rest of the partially instantiated plan will be further instantiated with that unifier. The test goal is then removed from the intention.

8. Once all of the formulas in the plan body have been executed the plan is removed from the intention. If it was triggered by an achievement goal then it is also removed.

9. This cycle is then repeated, starting with the agent perceiving it's environment.

2.2.3 Summary

The AgentSpeak(L) belief base consists of first-order logic terms that act as binary true or false representations of agent's beliefs. Agent's query their belief base when attempting to unify a plan context or a test goal in order to determine if the context is a logical consequence of the agent's current beliefs or to obtain a substitution evaluating the test goal as true. Plan contexts and test goals take the form of a conjunction of beliefs (first-order logic terms). In order to extend AgentSpeak(L) agents to model and reason with uncertainty the method for modelling beliefs must first be extended. This would then require new mechanisms for belief revision.

Following this, the language for constructing plan contexts and test goals must be extended so that agents are capable of reasoning about these belief (i.e. determining if they more strongly

believe "belief A" or "belief B"). Extending this language would require new mechanisms for determining if a formula written in the language is a logical consequence of the agent's belief base. Achieving all of these in a tractable and efficient manner is an active area of research. Bauters et al. (2017) present a theoretical approach for achieving this, which will be discussed in Section 2.4.

2.3 Uncertainty Modelling

This section discusses the theories that have been proposed to manage uncertain information, including probability theory (Ash 2012) and possibility theory (Dubois, Moral, and Prade 1998; Zadeh 1999). There are other theories that deal with uncertain information but these will not be discussed here. This section will focus on introducing different uncertainty theories and reviewing both their expressive power and computational complexity. The desired output of this work is the integration of expressive, scalable uncertainty theories into the AgentSpeak(L) language discussed in Section 2.4.

2.3.1 Probability Theory

Probability theory is a statistical, subjective uncertainty theory. It is used to capture variability through repeated observations (randomness) and to represent belief in situations with information deficit (partial knowledge) (Edwin 2003).

Probability can be defined from a frequentist point of view, where it is related directly to the frequency that events occur i.e. an event's probability is defined as the limit of it's relative frequency over a large number of trials. For example, if 50.9% of babies born in the UK are female then the frequentist probability of a newly born baby being female is P=0.509. Conversely, probability can be defined from a Bayesian perspective, where the probability is interpreted as reasonable expectation representing the state of knowledge or quantification of a belief. For example, the probability that the sun will rise again tomorrow will be high due to our strong prior belief that it will rise again tomorrow.

In the case of probabilistic modelling of beliefs, a probability distribution, P, on the set of all possible worlds, Ω, can be defined as follows:

Definition 2.3.1. If Ω is the set of all possible worlds, then a mapping $P : \Omega \rightarrow [0,1]$, such that $\sum_{w \in \Omega} P(w) = 1$, is called a probability distribution.

By convention, $P(w) = 1$ implies that the world w represents the true state of the world and $P(w) = 0$ implies that w is definitely not the true state of the world. Ignorance is modelled by insufficient information, represented by the uniform distribution $P(w) = \frac{1}{|\Omega|}$.

2.3.2 Possibility Theory

Possibility theory is an uncertainty theory that is capable of handling incomplete information (Dubois, Moral, and Prade 1998). It is defined as a possibility distribution $\pi : \Omega \rightarrow [0,1]$, mapping every possible world to a value between 0 and 1. By convention, $\pi(w) = 0$ implies that w is impossible and $\pi(w) = 1$ implies that none of the available evidence prevents w from representing the true state of the world. A possibility distribution corresponds to both a possibility measure and a necessity measure. For nested mass assignments in Dempster-Shafer theory the belief measure is called a necessity measure and the plausibility measure is called a possibility measure.

Possibility distributions are usually normalised as un-normalised distributions indicates the presence of conflicting information. Beliefs instantiated with possibility theory are usually compared qualitatively as they provide us with the ability to formulate expressions of the form $\psi_1 \geq \psi_2$ as $N(\psi_1) \geq N(\psi_2)$. This represents the intuition that ψ_1 is at least as plausible as ψ_2.

Definition 2.3.2. A possibility measure is a mapping $\Pi : 2^{\Omega} \rightarrow [0,1]$, $\forall A \subseteq \Omega$, defined as $\Pi(A) = max\{\pi(w) | w \in A\}$.

Definition 2.3.3. A necessity measure is a mapping $N : 2^{\Omega} \rightarrow [0,1]$, $\forall A \subseteq \Omega$, defined as $N(A) = 1 - \Pi(\Omega \backslash A)$.

2.3.3 Summary

Both probability theory and possibility theory fall into the NP (non-deterministic polynomial time) complexity class. NP problems are hard to solve but proofs are verifiable by deterministic computations performed in polynomial time. Possibility theory comes with two measures of possibility and necessity, which ensure that not as much information is thrown away as in probability theory. This makes possibility theory perfect for expressing degrees of ignorance. Possibilistic logic can be seen as an extension of propositional logic tolerant to inconsistencies that provides a semantic setting for non-monotonic reasoning (Benferhat, Dubois, and Prade 1998), whilst offering complexity close to propositional logic (Dubois and Prade 2015). This provides the ability to formulate logical expressions of the form $\phi_1 \geq \phi_2$ i.e. ϕ_1 is at least as plausible as ϕ_2. In the next section the application of different uncertainty theories in the BDI setting will be introduced and discussed.

2.4 Uncertainty Modelling in BDI Agents

There are many BDI agent programming languages that have been developed, "Procedural Reasoning System" (Ingrand, M. P. Georgeff, and Rao 1992), "AgentSpeak" (Rao 1996) and "A Practical Reasoning System" (2APL) (Dastani 2008). These current BDI implementations are not capable of modelling the next generation of systems as they are not able to model or reason with uncertain information. The real world is pervaded with uncertainty, therefore an agents

beliefs would be uncertain (due to sensor noise, incomplete information etc). Most uncertainty theories do not consider tractability, resulting in computational complexity issues (Bauters et al. 2017). This aggravates their integration into BDI implementations as BDI agents rely on reactive behaviour.

Currently, the majority of MASs model their agent's belief as binary (true or false) representations. This significantly limits the ability of BDI agents to react in a satisfactory manner in an uncertain environment. There are several sources of uncertainty that arise in the agent's beliefs:

- Environmental perceptions:
  - Uncertainty in what the agent has sensed,
- Actions:
  - Uncertainty in how successfully the agent completed it's intended action,
  - Uncertainty in a specific effect that can change the epistemic state/belief set.

Previous work has integrated uncertainty into BDI, known as Graded BDI (Ana, Lluis, and Carles 2005). In Graded BDI the beliefs, desires and intentions of an agent are modelled with a measure of uncertainty. This was extended further in (Casali, Godo, and Sierra 2011) to incorporate norms (patterns of behaviour to be followed). However, although this work is of theoretical value there is a mismatch between theory and practice. This is because Graded BDI uses complex modal logic axiomatisation, similar to that of BDI, making it hard to implement.

Previous work has also looked at the relationship between BDI and POMDP (Nair and Tambe 2005). They were successful in developing a hybrid BDI-POMDP framework that was able to reason about uncertainty, outperforming both BDI and POMDP. However, this framework is limited by the constraints of POMDP (modelling power and computational complexity).

In (Bauters et al. 2017) the operational semantics of Conceptual Agent Notation (CAN) are extended to deal with uncertain information. This is achieved by representing the uncertain beliefs of an agent as a set of epistemic states, which are stratified to make them commensurable and to enable reasoning with uncertain beliefs. They introduce the concept of a Global Uncertain Belief (GUB), a set of local epistemic states to represent agent's uncertain beliefs. Bauters et al. (ibid.) consider tractability throughout their theoretical work and thus provide a solid starting point for implementing an uncertainty capable agent-based programming language.

2.4.1 CAN+ (Bauters et al., 2017)

In this section the relevant theory for extending the BDI architecture to model and reason with uncertain information will be presented and discussed. This section is based on work done by (ibid.) and (Ma and Liu 2011b), in particular their method for managing different sources of uncertainty in a BDI framework. This section will first introduce Ma and Liu's epistemic

state, including it's formal definition and revision operator. After focusing on the semantic representation of epistemic states, an extended language that enables reasoning about uncertain beliefs is introduced and the mechanisms for determining if a logical formula is entailed by an epistemic state is detailed. Next, the extension to multiple epistemic states is outlined, including an introduction to the Global Uncertain Belief (GUB) and mechanisms for GUB revision and entailment. Following this, the tractable syntactic approach from Bauters et al. (2017) for implementing epistemic states is outlined, including probabilistic and possibilistic instantiations.

2.4.1.1 Epistemic States

First of all, we must define an epistemic state as seen in (Ma and Liu 2011b).

Definition 2.4.1. Let Ω be the set of possible worlds. An epistemic state Φ is a mapping $\Phi : \Omega \rightarrow \mathbb{Z} \cup \{ -\infty, +\infty \}$.

An epistemic state Φ is used to define the mental state of an agent and the value $\Phi(w)$ represents the strength (weight) of an agent's belief in a possible world w. $\Phi(w) = \infty$ represents a possible world w that the agent believes fully plausible, $\Phi(w) = -\infty$ indicates that the possible world is not plausible and $\Phi(w) = 0$ represents the case when the agent is totally ignorant about the plausibility of w.

Unlike other representations of epistemic states this definition does not apply any more meaning to the values. As a result this definition provides a general epistemic state that can be instantiated as any other representation.

The language $\mathscr{L}$ is defined in Backus-Naur Form (BNF) as: $\phi ::= a \mid \neg a \mid (\phi_1 \wedge \phi_2) \mid (\phi_1 \vee \phi_2)$, where a is a belief atom and ϕ is a formula such that $\phi \in \mathscr{L}$. New information consisting of a proposition $\phi \in \mathscr{L}$ and an associated weight $m \in (\mathbb{Z} \cup \{-\infty, +\infty\})$ can update existing beliefs. This information $input(\phi, m)$ can be represented as an epistemic state Φ_{in} such that $\Phi_{in}(w) = m$ iff $w \models \phi$ and $\Phi_{in}(w) = 0$ otherwise. This naturally leads to the epistemic state revision operator $\circ$ proposed by (ibid.): $\forall w \in \Omega$, $(\Phi \circ \Phi')(w) = \Phi(w) + \Phi'(w)$. This operator is both commutative and associative which is desirable for handling revision with uncertain information. This definition of an epistemic state forms the basis for how uncertain beliefs can be modelled.

In order to model and reason about uncertain beliefs a language $\mathscr{L}_{\geq}$ (an extension of $\mathscr{L}$) is defined in BNF as:

$$\psi ::= a \mid \neg a \mid (\psi_1 \wedge \psi_2) \mid (\psi_1 \vee \psi_2)$$
$$\phi ::= a \mid \neg a \mid (\phi_1 \wedge \phi_2) \mid (\phi_1 \vee \phi_2) \mid (\psi_1 \geq \psi_2) \mid (\psi_1 > \psi_2) \mid not\ \psi$$

This new language supports the intuition that one formula is more strongly believed than another ($\psi_1 > \psi_2$), i.e. it has a higher weight, that a formula is at least as plausible as another ($\psi_1 \geq \psi_2$) and it also supports negation-as-failure ($not\ \psi$) i.e. "ψ is assumed not to hold".

The semantics of the extended language $\mathscr{L}_{\geq}$ requires the definition of the λ-value, a mapping from formula $\phi \in \mathscr{L}_{\geq}$ to $\mathbb{Z} \cup \{-\infty, +\infty\}$. This value represents how strongly an agent believes the formula ϕ to be true.

Definition 2.4.2. Let $\phi \in \mathscr{L}_{\geq}$. If $\phi \in \mathscr{L}$ (i.e. ϕ is a propositional statement) then $\lambda(\phi) = \max\{\Phi(w) \mid w \models \phi\}$, where $\max(\emptyset) = 0$. Otherwise it is defined as $\lambda(\phi) = \lambda(pare(\phi))$, where $pare$ is:

$$
\begin{aligned}
pare(\phi \oplus \psi) &= check(\phi) \oplus check(\psi) \\
pare(\phi \geq \psi) &= \begin{cases} \top, & \text{if } \lambda(\neg\phi) \leq \lambda(\neg\psi) \\ \bot, & \text{otherwise} \end{cases} \\
pare(\phi > \psi) &= \begin{cases} \top, & \text{if } \lambda(\neg\phi) < \lambda(\neg\psi) \\ \bot, & \text{otherwise} \end{cases} \\
pare(not\ \phi) &= \begin{cases} \top, & \text{if } \phi \in \mathscr{L}\ and\ \lambda(\neg\phi) \geq \lambda(\phi) \\ \bot, & \text{otherwise} \end{cases} \\
check(\phi) &= \begin{cases} \phi, & \text{if } \phi \in \mathscr{L} \\ pare(\phi), & \text{otherwise} \end{cases}
\end{aligned}
$$

The conjunction $\wedge$ and disjunction $\vee$ operators are checked to ensure that both operands are formulas from language $\mathscr{L}$. If they are not then they are pared down to propositional formulas ($\phi \in \mathscr{L}$). Qualitative operators such as $\geq$ are expressed as an ordering, e.g. $\phi \geq \psi$ reads as "ϕ is more plausible that ψ" or, "$\neg\psi$ is more certain than $\neg\phi$". For negation-as-failure ($not\ \phi$) the classical negation of ϕ is checked to see if it is more believed than ϕ. It is worth noting that as λ is not normalised $\lambda(\top)$ can take different values in different epistemic states and also that the negation-as-failure and qualitative operators (such as $>$) only make sense when the operands ϕ are classical formulas.

Using the λ-value it is possible to determine if a formula ϕ is entailed by an epistemic state Φ.

Definition 2.4.3. Let $\phi \in \mathscr{L}_{\geq}$ and ϕ be an epistemic state, then $\Phi \models \phi$ iff $\lambda(\phi) > \lambda(\neg\phi)$.

This definition states that a formula ϕ is entailed by an epistemic state Φ if an only if $\lambda(\phi) > \lambda(\neg\phi)$. For a belief atom $a \in At$ it may be the case that $\lambda(a) = \lambda(\neg a)$, this represents total ignorance about the value of a. This is the reason both expressions must be mapped onto distinct values.

2.4.1.2 Multiple Epistemic States

Agent's usually require multiple epistemic states to represent their beliefs. This may be because different subsets of beliefs do not influence each other. Thus to prevent complexity issues that

arise due to the exponential size of each epistemic state, they can be separated into multiple epistemic states. As each epistemic state can be instantiated with different uncertainty theories it may also be practical to separate subsets of beliefs into different epistemic states.

Bauters et al. (2017) introduce the Global Uncertain Belief (GUB), which is a set of epistemic states.

Definition 2.4.4. A GUB $\mathscr{G}$ is a set of epistemic states $\{\Phi_1,...,\Phi_n\}$ such that each Φ_i represents an epistemic state epistemic state over the domain $A_i \subseteq At$ where $\{A_1,...,A_n\}$ is a partition of At.

Each epistemic state within the GUB represents beliefs that are semantically related and instantiated with the same uncertainty theory. A GUB is not an epistemic state which has the following consequences:

- By partitioning the beliefs it simplifies the exponential representation of epistemic states.
- Enables each epistemic state to be instantiated with a different representation.
- No revision strategy is defined as each local epistemic state has their own.

In the BDI setting an agent wishes to check if a plan context is entailed by it's belief base. In our setting the belief base has taken the form of a GUB and the logical formula ϕ has become the plan context. A formula ϕ can be evaluated if it is associated with a single epistemic state as $\Phi \models \phi$. However, more commonly the context is associated with multiple epistemic states, so it needs to be broken up and evaluated directly.

It is trivial to split an expression with conjunctive ($\wedge$) and disjunctive ($\vee$) connectives, however, it is not so simple with qualitative connectives such as $>$ and $\geq$. If the operands are instantiated with different uncertainty theories then it is not possible to evaluate the expression. For the case when the operands are from different epistemic states their different underlying structures also makes it not possible to evaluate the expression.

A new language on the GUB level is defined as $\mathscr{L}_{\mathscr{G}}$. The language $\mathscr{L}_{\mathscr{G}}$ consists of every formula $\phi \in \mathscr{L}_{\geq}^{A_i}$ over A_i with $i \in \{1,k\}$ i.e. it consists of every formula contained within the GUB. It also contains conjunctions and disjunctions i.e. for $\phi_1,\phi_2 \in \mathscr{L}_{\mathscr{G}}$ both $\phi_1 \wedge \phi_2$ and $\phi_1 \vee \phi_2$ are formulas within $\mathscr{L}_{\mathscr{G}}$.

Using this language a formula ϕ can be broken down and evaluated by considering the operands. If the connectives $(>,\geq)$ are not comparable then it will return a contradiction.

Definition 2.4.5. Let $\mathscr{G}$ be a GUB and $\phi \in \mathscr{L}_{\mathscr{G}}$ be a plan context, then $\mathscr{G}$ is said to entail ϕ ($\mathscr{G} \models \phi$) iff $val_{GUB}(\phi) \equiv \top$, where val_{GUB} is defined as:

$$val_{GUB}(\phi) = \begin{cases} \top, & \text{if } \phi \in \mathscr{L}_{\geq}^{A_i},\ \Phi_i \models \phi \\ \bot, & \text{if } \phi \in \mathscr{L}_{\geq}^{A_i},\ \Phi_i \not\models \phi \\ simplify(\phi), & \text{otherwise} \end{cases}$$

$$simplify(\phi \otimes \psi) = val(\phi) \otimes val(\psi) \qquad \otimes \in \{\vee, \wedge\}$$

A GUB must also be capable of revising with an uncertain belief $b = (\phi, m)$, where $\phi \in \mathscr{L}$. This is denoted as $\mathscr{G} \circ b$. Although a GUB can be revised as an epistemic state it is not computationally efficient to do so (Bauters et al. 2017). The belief revision can instead take the form of a marginalisation of the formula ϕ followed by revision according to each epistemic states revision strategy.

Definition 2.4.6. Let $\mathscr{G}$ be a GUB, $b = (\phi, m)$ be an uncertain belief revising $\mathscr{G}$ and $A_{in} = \{a^* | a \in lit(\phi)\}$ (the set of atoms in the formula ϕ). Now, for $\Phi_i \in \mathscr{G}$ (every epistemic state in the GUB), $refine(b, \Phi_i)$ is defined as:

$$refine(b, \Phi_i) = \begin{cases} forget(b, \Phi_i), & \text{if } A_{in} \bigcap A_i \neq \emptyset \\ \langle\rangle, & \text{otherwise} \end{cases}$$

where $forget(b, \Phi_i) = \langle (\bar{\alpha}, m) \mid w \in Mod(\phi),\ \alpha = w \bigcap lit(A_i) \rangle$ defines the sequence of inputs and $Mod(\phi)$ returns the set of all models of ϕ.

When the domain A_i of an epistemic state Φ_i does not intersect with the domain A_{in} of the input b, $refine$ returns an empty sequence of inputs i.e. the epistemic state Φ_i is not affected by the input b. When the domains do intersect ($A_{in} \bigcap A_i \neq \emptyset$) the formula is broken down into a sequence of inputs for the the relevant epistemic state Φ_i. Each epistemic state can then be iteratively revised for each input according to it's own revision strategy.

The final GUB revision can be defined as follows:

Definition 2.4.7. Let $\mathscr{G}$ be a GUB and b be an uncertain input. Revision can be denoted as $\mathscr{G} \circ b = \{\Phi_i \circ refine(b, \Phi_i) \mid \phi_i \in \mathscr{G}\}$ where $\circ$ is the corresponding revision operator for Φ_i.

Figure 2.5 shows visual representations of entailment and revision of a GUB.

2.4.1.3 Compact Epistemic Sates

As BDI agents require reactive behaviour it is extremely important that any approach for modelling and revising uncertain beliefs is not prohibitive. Bauters et al. (ibid.) propose a tractable syntactic approach for modelling and revising with uncertain inputs.

The approach utilises a compact epistemic state and only considers literals l as inputs, these can be either positive literals (a) or negative literals ($\neg a$). For this reason the weights associated

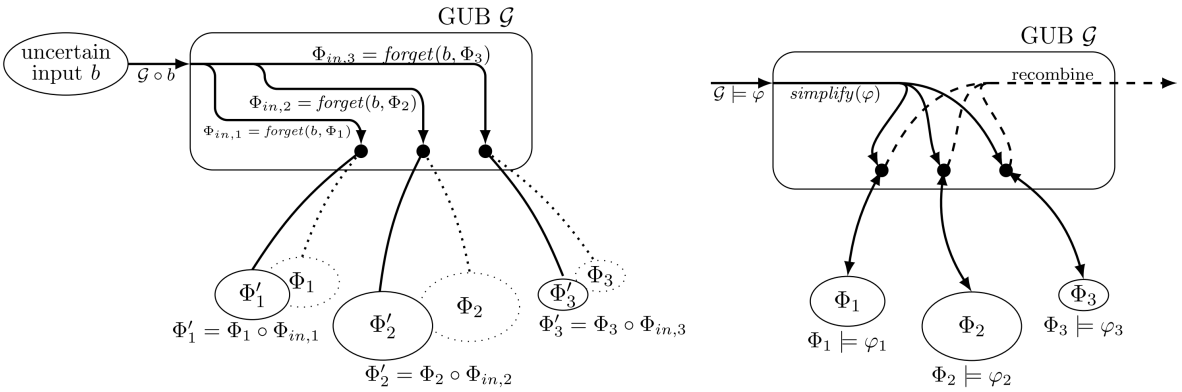


Figure 2.5: Diagrams showing Global Uncertain Belief (GUB) entailment (left) and revision (right) (Bauters et al. 2017)

with both literals $(a, \neg a)$ of a given atom $a \in At$ are maintained ($\overset{+}{\mu}$ and $\overset{-}{\mu}$ respectively). Here At refers to the finite set of atoms associated with the compact epistemic state, also known as the domain. Lit is used to define the corresponding set of literals.

Definition 2.4.8. A compact epistemic state W is defined as a mapping $W : At \rightarrow (\mathbb{Z} \cup \{-\infty, +\infty\})^2$, such that $W(a) = (\overset{+}{\mu}, \overset{-}{\mu})$.

This definition shows that each belief atom a is mapped to two weights ($\overset{+}{\mu}$ and $\overset{-}{\mu}$), associated with the positive a and negative $\neg a$ literals of the belief atom a. A tractable belief change operator adjusts the weight associated with the literal given as an input.

Definition 2.4.9. Let $W(a) = (\overset{+}{\mu}, \overset{-}{\mu})$ be a compact epistemic state and (l, μ) be an uncertain input, then $W' = W \circ_t (l, \mu)$ can be defined as:

$$W'(a) = \begin{cases} (\overset{+}{\mu} + \mu, \overset{-}{\mu}), & \text{if } l = a \\ (\overset{+}{\mu}, \overset{-}{\mu} + \mu), & \text{if } l = \neg a \\ W(a), & \text{otherwise} \end{cases}$$

A compact epistemic state can be implemented using a sorted map. This map contains elements between belief atoms and a pair of values representing the positive and negative weights $(\overset{+}{\mu}, \overset{-}{\mu})$. Revising such a compact epistemic state can be achieved using an algorithm of complexity $log_2(|At|)$ as it involves a binary search over the keys and a constant time revision (addition of value) (ibid.).

As well as belief revision an efficient belief entailment operator is required. The belief set of a compact epistemic state can be defined as $Bel(W) = \bigwedge\{l \in Lit \mid w_W(l) > w_W(\neg l)\}$. As $Bel(W)$ is a conjunction of literals it is easy to determine if a formula $\phi \in \mathscr{L}$ is a logical consequence of $Bel(W)$. Evaluating a formula $\phi \in \mathscr{L}$ requires replacing each occurrence of literals from $Bel(W)$ in ϕ as a tautology ($\top$) and all others as a contradiction ($\bot$). Although determining if a formula

$\phi \in \mathscr{L}$ is a logical consequence of $Bel(W)$ is straight forward, it is more complicated to reason about the uncertainty.

In order to reason about uncertainty efficiently a restricted language must be used. It comprises a fragment of the language where it is straight forward to find the literals whose weight is known (bounded literals). For example, when determining the weight of a formula the weights associated with the bounded literals (literals in the formula) must be used, whereas the highest weight associated with an unbounded literal l (weight of a or $\neg a$) can be used. Therefore the fragment of the language must make it easy to determine the set of bounded literals. The language $\mathscr{L}_t \subseteq \mathscr{L}$ is defined in BNF as follows:

$$\begin{aligned} disj &::= a \mid \neg a \mid disj_1 \vee disj_2 \\ conj &::= a \mid \neg a \mid conj_1 \wedge conj_2 \\ \phi &::= a \mid \neg a \mid disj \wedge conj \mid \phi_1 \vee \phi_2 \end{aligned}$$

The language ensures that whenever a conjunction occurs, one branch will contain only conjunctions whilst the other only disjunctions. In order to obtain the weight associated with a formula $\phi \in \mathscr{L}_t$ a new notion is introduced $T_W = \sum_{a \in At} \max(W(a))$. This represents the sum of all the maximum weights associated with each atom (the maximum sum of weights if no literals are bounded). This sum can be calculated easily from belief revision. For $W' = W \circ_t (l, \mu)$ it can be calculated as follows, $T_{W'} = T_W - \max W(l^*) + \max W'(l^*)$. From this new language the λ-values of a formula $\phi \in \mathscr{L}_t$ can be determined as follows:

Definition 2.4.10. Let W be a compact epistemic state, $\phi \in \mathscr{L}$ and L a set of literals. The λ-value is recursively defined as $\lambda_t(\phi, L)$:

$$\begin{aligned} \lambda_t(\phi_1 \vee \phi_2,\, L) &= \max(\lambda_t(\phi_1, L),\, \lambda_t(\phi_2,\, L)) \\ \lambda_t(disj \wedge conj, L) &= \lambda_t(disj,\, L \cup lit(conj)) \\ \lambda_t(l,\, L) &= \begin{cases} -\infty, & \text{if } inconsistent(L \cup \{l\}) \\ \max_{T_W}(l, L), & \text{otherwise} \end{cases} \end{aligned}$$

where $inconsistent(S)$ is true if $\exists a \in At \cdot \{a, \neg a\} \subseteq S$ and $\max_{T_W}(l, L) = T_W - \sum_{l' \in L \cup \{l\}} |w_W(l') - \max W(l'^*)|$.

This definition keeps track of the bounded literals as required. A disjunction results in the maximum λ-value being taken forward without any change to the bounded literals, whereas a conjunction defines what literals are bounded and thus adds literals to the set of bounded literals. When the formula is reduced to a literal l, both l and the set of bounded literals L are checked for consistency. If they are inconsistent then the λ-value is set to $-\infty$. Otherwise, it is calculated

starting from T_W and removing all of the maximum weights associated with the bounded literals $L \cup \{l\}$ and then adding the correct bounded weights back.

The language $\mathscr{L}_t$ can be extended to include operators such as $\phi_1 > \phi_2$ and $\phi_1 \geq \phi_2$. The $\lambda(\phi)$ value of a formula ϕ can be computed and then the $pare(\phi \geq \psi)$ function defined earlier can be used. The language $\mathscr{L}_t^{\geq}$ is defined in BNF as:

$$
\begin{aligned}
disj &::= a \mid \neg a \mid disj_1 \vee disj_2 \\
conj &::= a \mid \neg a \mid conj_1 \wedge conj_2 \\
\phi &::= a \mid \neg a \mid disj \wedge conj \mid \phi_1 \vee \phi_2 \mid \phi_1 \geq \phi_2 \mid \phi_1 > \phi_2
\end{aligned}
$$

2.5 Multi-Agent System Development Tools

Recent research in MAS has led to the development of programming languages and development tools for implementing such systems. This new programming paradigm is an extremely important part of MAS research. This work is focused on creating and testing a development environment for agents programmed in Uncertain AgentSpeak(L). There are many MAS platforms that can be reviewed and Kravari and Bassiliades (2015b) provide a comprehensive survey. As this work is based on an extension of AgentSpeak(L) it is logical to review the MAS platforms that are based on AgentSpeak(L).

Jason is the most widely used platform based on AgentSpeak(L) (Rafael H. Bordini and Jomi F. Hübner 2006). It is an interpreter for an extended version of AgentSpeak(L) that is implemented in Java. It implements the operational semantics of AgentSpeak(L) and provides a development environment for defining and simulating AgentSpeak(L) agents. It provides multiple infrastructures for implementing MASs: centralised, Simple Agent Communication Infrastructure (SACI) and Java Agent DEvelopment Framework (JADE).

A centralised architecture enables all of the agents and the environment to be run on a single computer. In order for this to be achieved the environment is executed on it's own separate thread and attends to the actions requested by agents. Each agent is assigned a thread which executes the agent's reasoning cycle. As a result the environment is capable of handling multiple agent requests concurrently. However, this infrastructure is limited in the number of agents that can be run and depends on the capabilities of the JVM and the operating system.

The Simple Agent Communication Infrastructure (SACI) (J. Hübner and Sichman 2009) is a Java API that provides a set of tools for distributed groups of agents. The SACI API has two important features: (i) composing and sending/receiving messages and, (ii) removing architecture design from agent designers (Fernández et al. 2010). The agents are grouped into societies that have mailboxes, providing communication between them. This can be seen in Figure 2.6.

A more popular infrastructure that is provided by Jason is the Java Agent DEvelopment Framework (JADE) (TILAB 2009). JADE uses a middle-ware that complies with FIPA specifi-

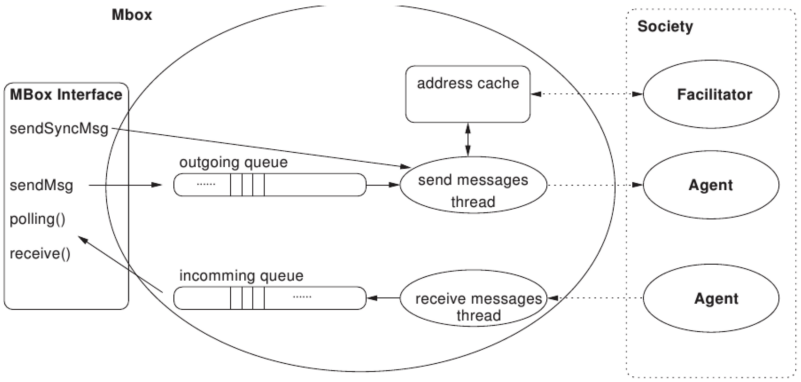


Figure 2.6: Simple Agent Communication Infrastructure (J. Hübner and Sichman 2009)

cations and simplifies the implementation of MASs. A MAS platform can be distributed over multiple machines using JADE, as can be seen in Figure 2.7.

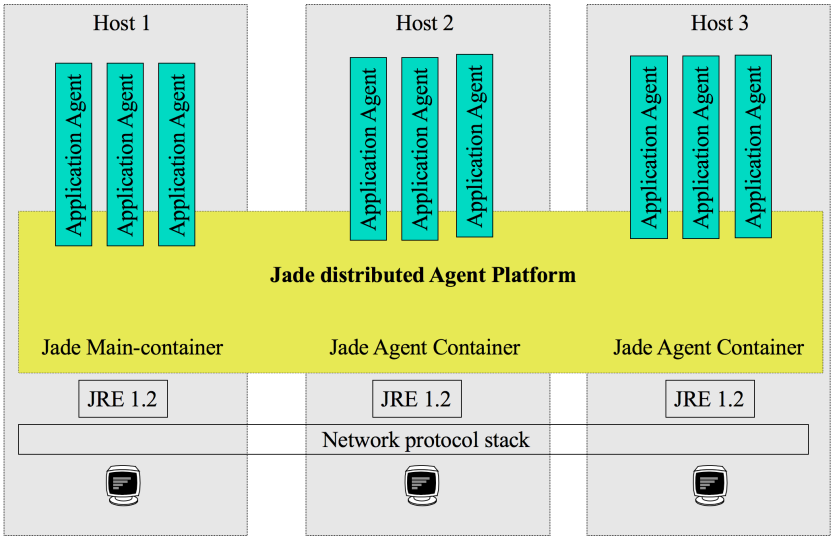


Figure 2.7: Java Agent DEvelopment Framework (JADE) (TILAB 2009)

In order to provide portability when designing a framework it is logical to implement the framework in Java as programs written in Java can be run on any machine that can run the JVM. The different infrastructures introduced in this section provide different functionalities to a designer and the overall MAS. Although distributed infrastructures can provide enhanced performance, most MAS development tools provide a centralised infrastructure as it offers a simple and effective means of developing and testing MASs. The complexity behind implementing distributed infrastructures has resulted in their application mainly in development tools tailored towards implementing industrial scale systems.

2.6 Summary

The aim of this chapter was to review the relevant literature that will be encountered throughout this dissertation. This chapter has provided the necessary background as well as a comprehensive review of the technologies related to this research. The main areas that were covered in this section are as follows:

- the BDI architecture
- the AgentSpeak(L) agent programming language
- modelling uncertain information using probability theory and possibility theory
- modelling and reasoning with uncertain information in a BDI setting
- MAS development tools for designing and testing AgentSpeak(L) agents.

CHAPTER 3

EXTENDING AGENTSPEAK(L) TO MODEL AND REASON WITH UNCERTAINTY

This chapter details the research methodology that was employed in response to the initial research objectives for implementing uncertainty modelling and reasoning capabilities in the AgentSpeak(L) programming language. Initially this chapter highlights the similarities and differences between the languages and the rationale behind the design decisions that were made. Next, the implementation of standard first-order terms is discussed, along with the underlying mechanisms for implementing unification. These form the basic building blocks for both AgentSpeak(L) and the extension. This is followed by the implementation of the language $\mathscr{L}_{\geq}$ for constructing logical formula that are used in plan contexts and test goals. This language is what provides the extended reasoning capabilities of the agents enabling them to select plans by reasoning about their uncertain beliefs. Next, the extension of the belief base is outlined, first focusing on the implementation of epistemic states and their mechanisms for revision and entailment. It will then focus on how these are used by a GUB for modelling the uncertain beliefs of an agent and determining when a logical formula (plan context or test goal) is a logical consequence (entailed) of the GUB. The remainder of this section will then outline the implementation of the remaining AgentSpeak(L) components (goals, triggering events, actions, plans) and will finish by detailing how the agents interpreter functions by utilising all of the components mentioned previously.

3.1 AgentSpeak(L) Modifications

In order to implement the extended language some fundamental changes were made. The first modification involved implementing the language $\mathscr{L}_{\geq}$ so that both the plan context ϕ from

$e : \phi \leftarrow P$ and test goals $?\phi$ could be extended to take the form of any sentence from $\mathscr{L}_{\geq}$, as apposed to just a conjunction of belief literals (as in AgentSpeak(L)). The language $\mathscr{L}_{\geq}$ is shown below:

$$
\begin{aligned}
disj &::= a \mid \neg a \mid disj_1 \vee disj_2 \\
conj &::= a \mid \neg a \mid conj_1 \wedge conj_2 \\
\phi &::= a \mid \neg a \mid disj \wedge conj \mid \phi_1 \vee \phi_2 \mid \phi_1 \geq \phi_2 \mid \phi_1 > \phi_2
\end{aligned}
$$

In AgentSpeak(L), beliefs are implemented as belief atoms, which are not capable of modelling uncertainty. Therefore, beliefs had to be extended so that they could take the form of epistemic states, where each belief literal has an associated weight, corresponding to the strength of the agent's belief.

In AgentSpeak(L), belief revision is simply the addition and deletion of belief atoms from the agent's belief base, however, this is not the case for epistemic states. For this reason, the mechanisms for belief revision also required modification.

Belief entailment occurs when an agent checks that a plan's context is a logical consequence of the agents belief base. In AgentSpeak(L) this is as simple as checking that a belief literal is present in the agent's belief base. In the extended version, entailment is much more complicated due to plan contexts taking the form of any sentence from the extended language $\mathscr{L}_{\geq}$. As a result, the extended version required modification of the mechanisms for implementing belief entailment.

Belief revision and entailment occur on both an individual belief level and on the belief base level. The modification to modelling beliefs as epistemic states lead to the belief base requiring extending to a GUB. As described in Section 3, a GUB is a set of logical formulas over the language $\mathscr{L}_{\geq}$ that also supports belief revision. As such, a GUB offers tractable modelling and revision of uncertain beliefs in the BDI setting.

3.2 Syntax

In Section 2.2 the standard AgentSpeak(L) grammar was presented. In Figure 3.1 the BNF grammar for the extended AgentSpeak(L) language is shown.

In the grammar <VAR> is an identifier beginning with an uppercase letter (representing a variable), <ATOM> is an identifier beginning with a lowercase letter, <NUMBER> is any integer number and <DECIMAL> is any floating point number. There are five key differences:

- The agent's beliefs are no longer simply belief atoms but instead can be instantiated as probabilistic or possibilistic compact epistemic states.
- There are no longer event triggers for belief addition and deletion but instead a single event trigger for belief revision.

```
uncertainAgentspeak     ::= init_bels init_goals plans EOF

// Initial beliefs
init_bels               ::= (probabilistic_es | possibilistic_es )*
probabilistic_es        ::= probabilistic_bel+
possibilistic_es        ::= possibilistic_bel+
probabilistic_bel       ::= '**(' belief_literal ',' number ')' '.'
possibilistic_bel       ::= '***(' belief_literal ',' number ')' '.'

// Initial goals
init_goals              ::= ( achievement_goal '.' )*

// Plans
plans                   ::= ( plan )*
plan                    ::= event ':' context '<-' body '.'

// Event
event                       ::= belief_event_trigger | goal_event_trigger | 'true'
belief_event_trigger        ::= '*(' belief_literal ',' term ')'
goal_event_trigger          ::= add_goal_event_trigger | delete_goal_event_trigger
add_goal_event_trigger      ::= '+' goal
delete_goal_event_trigger   ::= '-' goal
goal                        ::= achievement_goal | test_goal
achievement_goal            ::= '!' term
test_goal                   ::= '?' log_expr

// Plan Context
context                 ::= log_expr | 'true'

// Logical Expressions
log_expr                ::= or_expr ('&&' or_expr)*
or_expr                 ::= less_than_expr ('||' less_than_expr)*
less_than_expr          ::= less_equals_expr ('<' less_equals_expr)*
less_equals_expr        ::= greater_than_expr ('=<' greater_than_expr)*
greater_than_expr       ::= greater_equals_expr ('>' greater_equals_expr)*
greater_equals_expr     ::= equals_expr ('>=' equals_expr)*
equals_expr             ::= not_equals_expr ('==' not_equals_expr)*
not_equals_expr         ::= negation_expr ('\\==' negation_expr)*
negation_expr           ::= 'not' belief_atom_expr
                          | '~' belief_atom_expr
                          | belief_atom_expr
belief_atom_expr        ::= belief_atom | '(' log_expr ')'

// Plan body
body                    ::= body_statement (';' body_statement)* | 'true'
body_statement          ::= belief_action | goal | environment_action
belief_action           ::= '*(' belief_literal ',' term ')'
environment_action      ::= term

// Beliefs
belief_literal          ::= positive_literal | negative_literal
positive_literal        ::= belief_atom
negative_literal        ::= '~' belief_atom
belief_atom             ::= term

// Terms
term                    ::= constant | variable | structure
constant                ::= atom | number
variable                ::= <VAR>
structure               ::= <ATOM> '(' arguments_list ')'
atom                    ::= <ATOM>
number                  ::= intNum | doubleNum
arguments_list          ::= term | term ( ',' term )+
intNum                  ::= <NUMBER>
doubleNum               ::= <DECIMAL>
```

Figure 3.1: Extended AgentSpeak(L) grammar

- Test goals no longer take the form of a standard first-order logic term but can instead take the form a formula ϕ from the language $\mathscr{L}_{\geq}$.

- Plan contexts are no longer simply a conjunction of literals but instead can be expressed as

a formula ϕ from the language $\mathscr{L}_{\geq}$.

- There are no longer actions for belief addition and belief deletion but instead a single action for belief revision.

The rest of this section will outline the methodology and justify the design decisions that were made whilst implementing the extended language.

3.3 Terms

The underlying first-order logic terms that form the basis for writing agent programs are introduced here. These are the terms from the bottom of the grammar in Figure 3.1. The implemented alphabet consists of:

1. *A set of constant symbols* - Constant names start with lowercase letters (atoms) e.g. john. Integers and double numbers are also allowed.

2. *A set of variable symbols* - Variables are used to denote the same elements in case the name of an element is not known. Variables names start with uppercase letters e.g. Name.

3. *A set of structure symbols* - Structures serve as complex object constructors. Here they are used to construct predicates that take a number of terms as arguments e.g. parent(john, steve).

Figure 3.2 shows the class diagrams and inheritance structure that was used to implement the terms. These form the basic "data types" that were used to implement both AgentSpeak(L) and the extension.

3.3.1 Unification

At the heart of any computational model for a logic programming language is it's unification algorithm. In this work a unifier class was created to act as a mapping from variables to terms. The unifier class extends the hash map class with variable's as keys and term's as values. Hash maps ensures that every key (variable) is unique.

Each term contains a method for substituting a unifier into the term and returning the result. This is used throughout AgentSpeak(L) and the extended version to enable the instantiation of variables.

Unification was achieved by recursive descent (Baader and Snyder 1999). The term class contains a unify(Term) method that unifies the term associated with the class with the given input term. It checks if the term is an instance of a constant, variable or structure and calls the relevant unify() method for that term type.

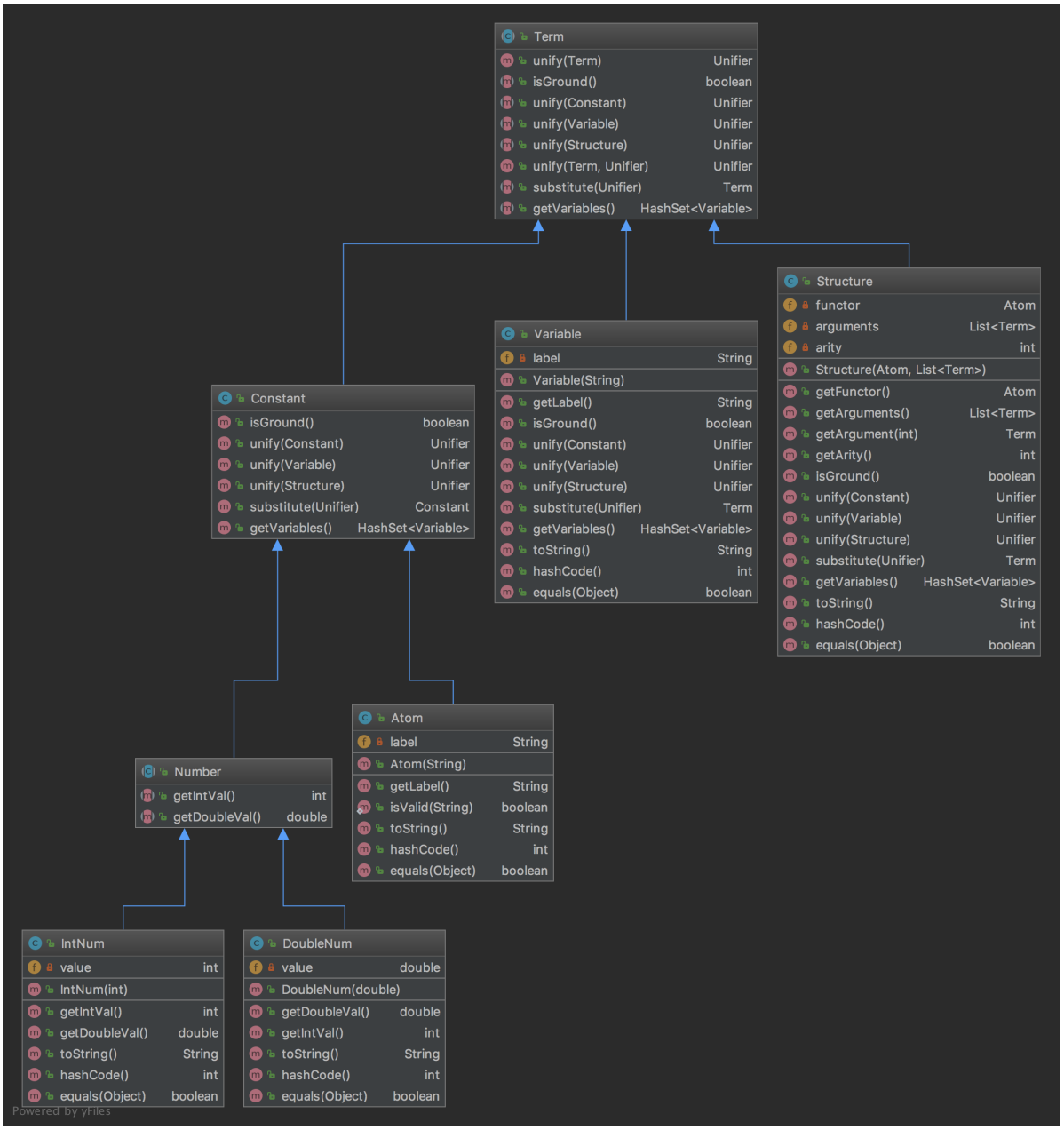


Figure 3.2: Class diagrams of first-order logic terms

- **Constants** - If a constant attempts to unify with another constant it will return *null*, unless the two constants are the same, in which case it will return an empty unifier. If a constant attempts to unify with a variable a unifier with a mapping from the variable to the constant will be returned. If a constant attempts to unify with a structure it will return *null*.

- **Variable** - If a variable attempts to unify with another variable it will return a unifier mapping itself to the other variable, unless the two variables are the same, in which case it

will return an empty unifier. If a variable attempts to unify with a structure it will return a unifier with a mapping from the variable to the structure.

- **Structure** - Unifying two structures is slightly more complicated than the other cases. A method unify(Term, Unifier) was created that first substitutes the unifier into both terms and then attempts to unify them, returning the most general unifier. The unify(Structure) method from the structure class first checks that the two structures have the same predicate symbol and that they are of the same arity. It then iterates through each argument and calls the unify(Term, Unifier) method with each argument and a unifier. Originally this unifier is empty but any unifiers returned by the unify(Term, Unifier) method are added to the unifier for subsequent arguments. This procedure ensures that the most general unifier will be returned as the hash map will replace any previous mappings with the new input if they have the same key (variable).

3.3.1.1 Example

An example term consisting of constants (IntNum, DoubleNum, Atom), variables and structures is given below:

```
1 move(john, location(X1,Y1), location(2,1.0)).
```

Figure 3.3 shows the parse tree for the example using the grammar shown in Figure 3.1. It is clear that the parser is cable of detecting all types of terms including nested terms.

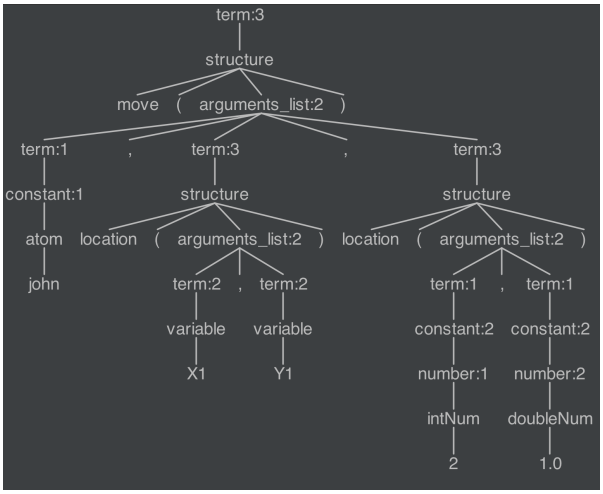


Figure 3.3: Parse tree for the term - move(john, location(X1,Y1), location(2,1.0)).

3.4 Logical Expressions for Plan Contexts and Test Goals

Now that the standard first-order terms have been introduced the implementation of the language $\mathscr{L}_{\geq}$ for writing logical formulas used in plan contexts and test goals will be detailed. Figure 3.4

shows the class inheritance structure that was used to implement the extended language $\mathscr{L}_{\geq}$. This class inheritance structure forms the base for all logical formulas that can be constructed in the extended version of AgentSpeak(L), enabling reasoning with uncertain beliefs.

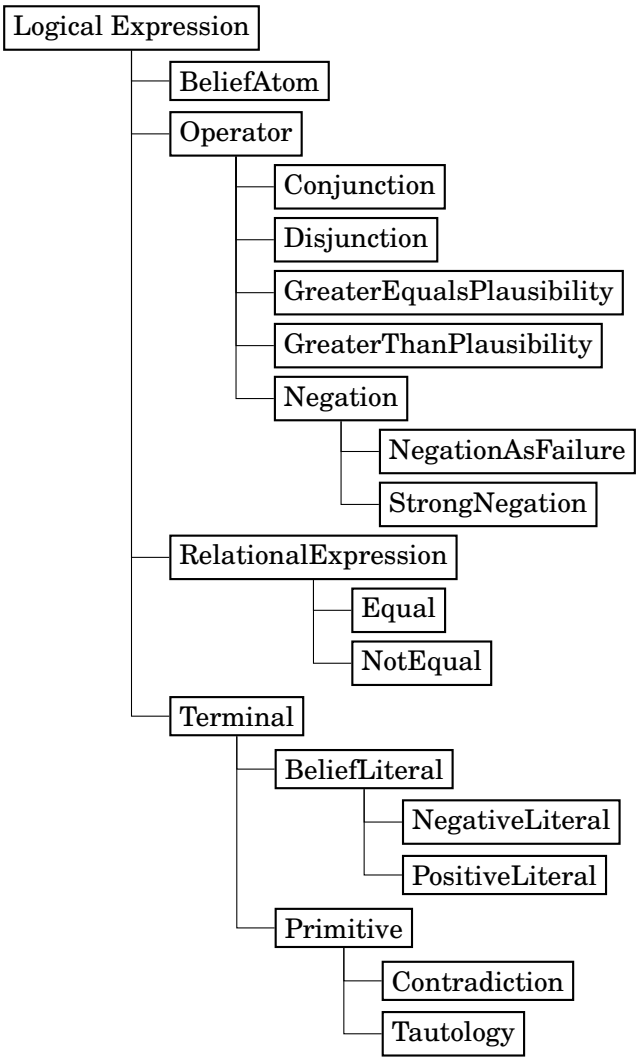


Figure 3.4: Class inheritance structure for Logical Expressions in the extended language

This class structure includes all of the operators in the language $\mathscr{L}_{\geq}$ and also includes relational expressions for equals and not equals. All formulas constructed as a logical expression consist of terminals, which are either positive or negative literals (consisting of an underlying belief atom) or are a tautology or contradiction, if they are true or false respectively. Relational expressions consist of two expressions (taking the form of logical expression's) and an underlying relational operator. The conjunction, disjunction, greater equals plausibility and greater than plausibility operators also consist of two operands that are logical expressions. The negation operator consists of a single operand and has two implementations, strong negation and negation as failure.

A parser generator called ANTLR (Parr 2013) was used for parsing the extended language and instantiating agents. Figure 3.1 contains the grammar rules that were used for logical expressions. The grammar enforces a well-defined precedence for the order in which operators are evaluated, as seen in Table 3.1. The grammar permits parentheses to override precedence as most programmers do not remember precedence rules.

Table 3.1: Table showing operator precedence for logical expressions

Level	Operator	Description	Associativity
9	()	Parentheses	Left to right
8	AND	Logical AND	Left to right
7	OR	Logical OR	Left to right
6	<	Relational	Left to right
6	≤	Relational	Left to right
5	>	Relational	Left to right
5	≥	Relational	Left to right
4	==	Equality	Left to right
3	\==	Equality	Left to right
2	not	Negation as failure	Right to left
1	¬	Strong negation	Right to left

An example logical expression (a formula written in the language $\mathscr{L}_{\geq}$) is given below and Figure 3.5 shows it's corresponding parse tree.

```
1 water_or_ice(location(X)) >= water_or_ice(location(Y1)) && water_or_ice(location(X)) >= water_or_ice
    ( location(Y2)) && at(location(Z)) > at(location(X)) && X \== Z.
```

When the formula is parsed it is instantiated into the corresponding logical expressions shown by the parse tree.

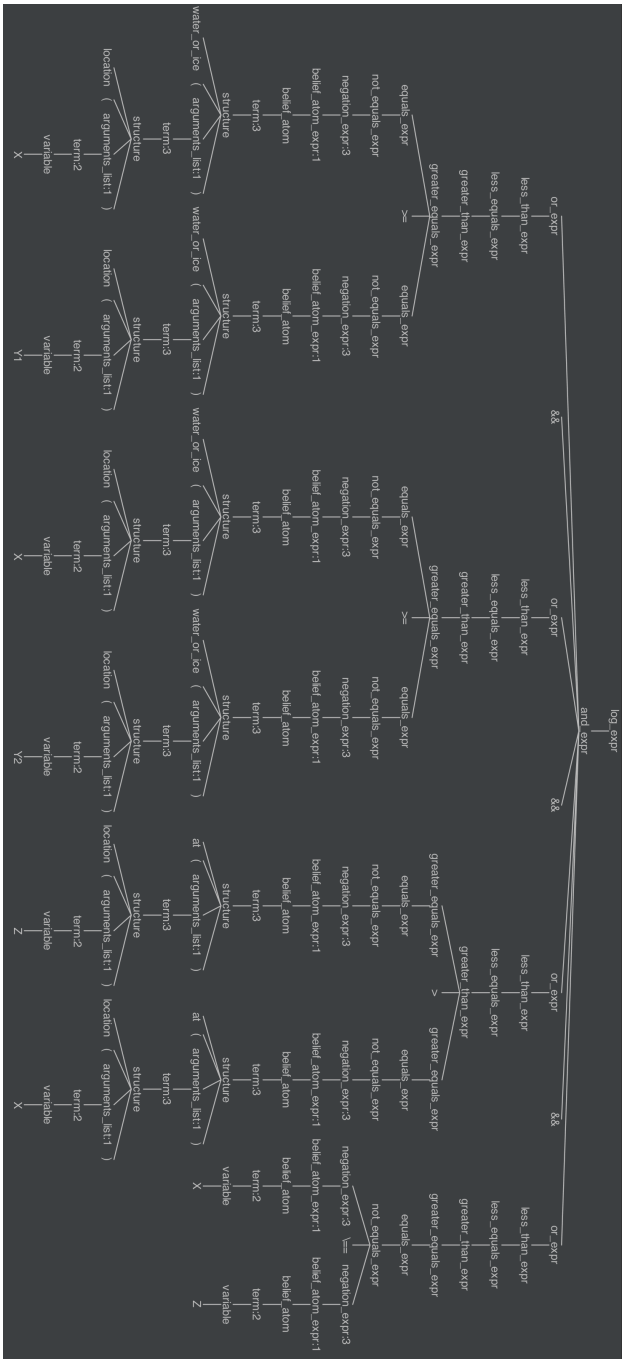


Figure 3.5: Parse tree for example logical expression

Figure 3.6 shows the classes derived from the logical expression class. These classes were extended further and their class inheritance structures can be seen in Figures B.1-B.3. Of particular interest are the properties and methods of the logical expression class. These are described below:

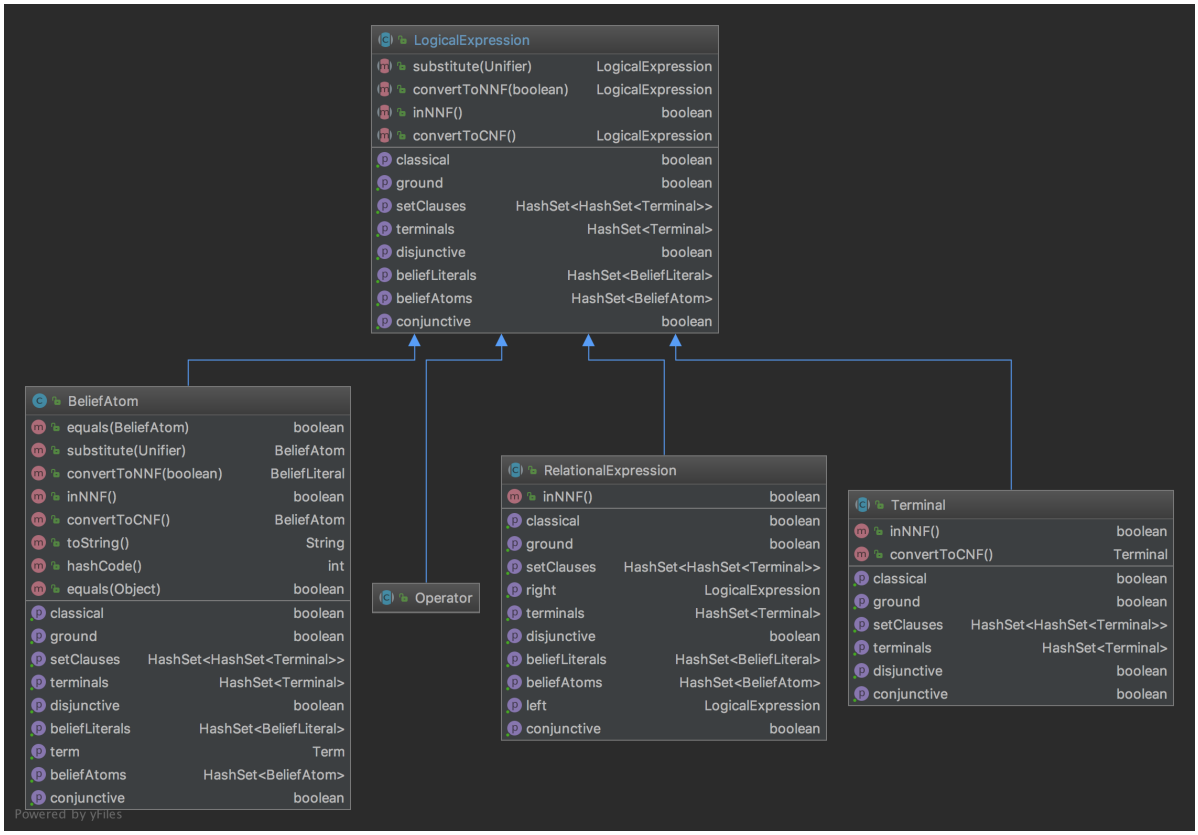


Figure 3.6: Logical Expression Class Diagrams

- **substitute(Unifier)** - this method performs substitution on any free variables in the logical expression using the unifier passed to the method.

- **convertToNNF(boolean)** - this method converts the logical expression into Negation Normal Form (NNF). The boolean input signifies if a strong negation needs to be propagated e.g. convert a negated conjunction ($\neg(a \wedge b)$) into a conjunction of the negation of the conjuncts ($\neg a \wedge \neg b$). This is used in the entails(LogicalExpression) method in the epistemic state class and also in the getLambda(LogicalExpression) method in the compact epistemic state class. These methods require formulas in the language $\mathscr{L}_{\geq}$ and the NNF of a formula ϕ is an expression in the language $\mathscr{L}_{\geq}$.

- **inNNF()** - this method returns true if the logical expression is in NNF. This is also used by the getLambda(LogicalExpression) method in the compact epistemic state class. If the formula is not in NNF then it is converted to NNF.

- **convertToCNF** - this method converts the logical expression into Conjunction Normal Form (CNF). The SAT solver that was used (SAT4J) requires the formula to be in CNF.

- **classical** - this property is true if the logical expression is a classical formula. The negation as failure and qualitative operators are only valid when the operands are classical formulas. This method is used in their constructors and throws an exception if their operand(s) are not classical formulas. It is also used by the epistemic state class in the pare() method to ensure that the operand(s) are calssical formulas, if they are not then the method will return a contradiction.

- **ground** - this property is true if the formula contains no free variables and false otherwise. Whenever a compact epistemic state is instantiated all of the belief atoms must be ground, otherwise an exception is thrown. The getLambda() method in the compact epistemic states class also utilises the property as it cannot determine a λ-value for a formula with uninstantiated variables. When revising a GUB with a belief literal and associated weight the revise() method checks that the belief literal is ground. Revising a belief literal with uninstantiated variables would require finding all unifiers and then performing the revise() method for each instantiation. This would be too computationally complex, which is why the belief literals must be ground.

- **setClauses** - represents the set of all clauses of the formula. The getLambda(Conjunction) method in the compact probabilistic epistemic state requires a SAT solver, which requires the set of clauses to operate.

- **terminals** - this property is the set of terminals associated with the formula.

- **disjunctive** - this property is true if the formula is disjunctive and false otherwise. It is used by the getLambda(Conjunction) method in the compact epistemic state class as required by Definition 2.4.10.

- **beliefLiterals** - this represents the set of belief literals associated with a formula. This is also used by the getLambda(Conjunction) method in the compact epistemic state class and is used to add belief literals from a conjunction into the set of bounded literals as required by Definition 2.4.10.

- **beliefAtoms** - this represents the set of belief atoms associated with a formula. It is used by the languageContains(LogicalExpression) method in the epistemic state class to check that all of the belief atoms in a formula are contained within the epistemic states domain. This is consistent with enforcing formulas to be in the language $\mathscr{L}_{\mathscr{G}}$ from Definition 2.4.5. (The language $\mathscr{L}_{\mathscr{G}}$ consists of every formula $\phi \in \mathscr{L}_{\geq}^{A_i}$ over A_i with $i \in \{1,k\}$ i.e. it consists of every formula contained within the GUB. It also contains conjunctions and disjunctions i.e. for $\phi_1, \phi_2 \in \mathscr{L}_{\mathscr{G}}$ both $\phi_1 \wedge \phi_2$ and $\phi_1 \vee \phi_2$ are formulas within $\mathscr{L}_{\mathscr{G}}$.)

- **conjunctive** - this property is true if the formula is conjunctive and false otherwise. It is used by the getLambda(Conjunction) method in the compact epistemic state class as required by Definition 2.4.10.

The logical expression class that has been detailed here is consistent with the theory from Bauters et al. (2017) for implementing the extended language $\mathscr{L}_{\geq}$ for reasoning about uncertain beliefs. In AgentSpeak(L) test goals take the form of terms and plan contexts are just a conjunction literals. In the extension of AgentSpeak(L) both plan contexts and test goals are logical formulas ϕ from the language $\mathscr{L}_{\geq}$ and are implemented as instantiations of the logical expression class. Now that the language for constructing logical formulas has been detailed the implementation of the extended belief base is introduced. The belief base requires methods for checking if a logical expression object is a logical consequence (entailed) by it.

3.5 Belief Base

This section will detail how the belief base was implemented for the extended AgentSpeak(L) language. It will first detail how the AgentSpeak(L) belief base was implemented and highlight the key modifications that were required for the extension.

In AgentSpeak(L) beliefs are simply a belief atom or it's negation i.e. a belief literal. The belief base was implemented as an array list of belief's (ArrayList<Belief>) as it provided all of the list methods required for the belief base. In AgentSpeak(L) plan contexts (logical formulae) consist of a conjunction of belief literals. These are then evaluated by ensuring that every belief literal in the context is a logical consequence of the belief base i.e. they are all contained in the belief base. This was implemented by defining a class (Context) of type LinkedList<Belief>. Plan contexts are evaluated when selecting applicable plans from the set of relevant plans.

This required iterating through each belief literal in the belief base, checking if the belief literal and context belief literal were either both negative or both positive literals, substituting the relevant unifier and checking that the context belief literal unifies with the belief literal. Upon successful unification a unifier is returned, which is then taken forward and used for substitution in the evaluation of subsequent context beliefs.

It is clear that in AgentSpeak(L) belief revision is as simple as adding and removing beliefs and that entailment is easily evaluated. The rest of this section will now detail how the belief base for the extended version of AgentSpeak(L) was implemented. It will first outline the implementation of an epistemic state (representing a single belief), including the compact epistemic state and it's probabilistic and possibilistic instantiations. This will include the mechanisms for revision and entailment of these epistemic states. After this, the implementation of the GUB as the belief base will be discussed. Again, this will include the mechanisms for revision and entailment.

3.5.1 Epistemic States

The concept of epistemic states was introduced in Section **??** in order to model uncertain beliefs. In particular, a general epistemic state that can be instantiated as any other type was detailed. The implementation of the logical expression class (the extended language $\mathscr{L}_{\geq}$), capable of modelling and reasoning about uncertain beliefs was introduced in Section 3.4. This language required the definition of a λ-value, a mapping from formula $\phi \in \mathscr{L}_{\geq}$ to $\mathbb{Z} \cup \{-\infty, +\infty\}$, representing how strongly an agent believes ϕ to be true. Definition 2.4.2 details how a formula $\phi \in \mathscr{L}_{\geq}$ can be pared down so that it's λ-values can be determined directly. Using the λ-value agents are able to determine if a formula $\phi \in \mathscr{L}_{\geq}$ is a entailed by the epistemic state.

Figure 3.7 shows the class diagram for the implementation of epistemic states. The epistemic state class is abstract and provides basic functionality, making it easier to implement different epistemic state instantiations. The compact epistemic state was implemented as it offers a tractable syntactic approach to modelling and revising with uncertain inputs. Both of the probabilistic and possibilistic compact epistemic state instantiations were also implemented as they provide powerful representations of beliefs whilst achieving good efficiency. Appendix A provides the java code for implementing the GUB and epistemic states.

An epistemic state is defined by it's domain (the set of belief atoms it covers) and provides the following base methods:

- **languageContains** - The languageContains() method takes a LogicalExpression as an input and returns true if the epistemic state's domain contains all of the belief atoms in the logical expression. This method is used by the GlobalUncertainBelief class to ensure that each belief atom in a formula ϕ is contained within the domain of an epistemic state.

- **pare** - This method takes a LogicalExpression $\phi \in \mathscr{L}_{\geq}$ as an input and recursively pares it down as in Definition 2.4.2. It checks whether the qualitative operators ($\phi > \psi, \phi \geq \psi, not\ \phi$) are true or false, returning a Tautology or Contradiction respectively. Otherwise it returns a LogicalExpression $\psi \in \mathscr{L}$ for which a λ-value can be determined directly.

- **entails** - This method takes a LogicalExpression as an input and creates a new empty unifier. It checks that the formula is in NNF and that all of the belief atoms in the formula are contained within the epistemic state's domain. It then substitutes the unifier in order to ground the formula and then determines the λ-values of the grounded formula ϕ and it's (strong) negation $\neg\phi$. If $\lambda(\phi) > \lambda(\neg\phi)$ then the method returns the unifier and returns *null* otherwise. This entailment function is used for all of the epistemic state instantiations and is called by the global uncertain belief class when checking that a formula $\phi \in \mathscr{L}_{\geq}$ is entailed by a given epistemic state.

The EpistemicState class is provided in Appendix A, which outlines the implementation of entailment for the abstract epistemic state class. The entails() method returns the relevant

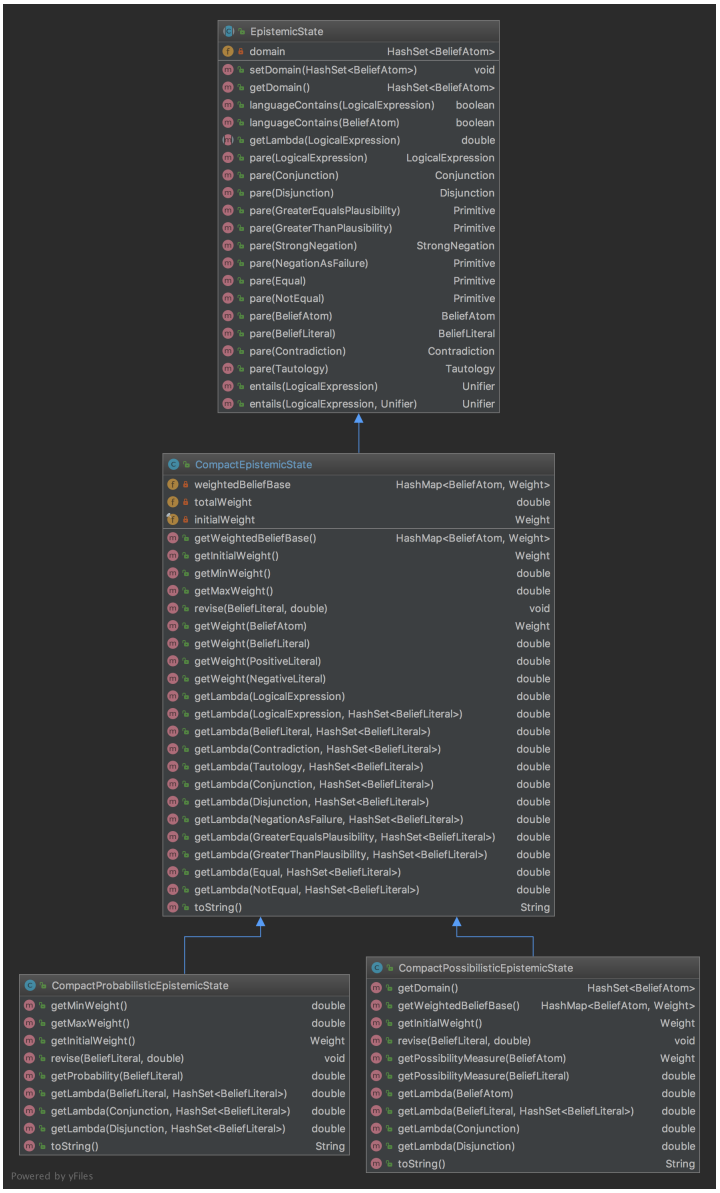


Figure 3.7: Epistemic States Class Diagrams

unifier for the logical expression ($\phi \in \mathscr{L}_{\geq}$) if it is a logical consequence of the epistemic state and *null* otherwise.

3.5.1.1 Compact Epistemic States

In Section 2.4 the compact epistemic state was introduced as an approach to achieve tractable syntactic modelling and revision with uncertain input. This compact epistemic state is an extension of the more general epistemic state detailed previously. It is defined as a mapping $W : At \rightarrow (\mathbb{Z} \cup \{-\infty, +\infty\})^2$ such that for a belief atom a, $W(a) = (\overset{+}{\mu}, \overset{-}{\mu})$ i.e. the weights associated

with the positive and negative literals (a and $\neg a$ respectively).

A class named Weights was created to maintain the positive and negative weights associated with a belief atom in a compact epistemic state. It contained two class variables of type double, representing the positive and negative weights. The compact epistemic state was implemented as a class with a variable named weightedBeliefBase (of type HashMap<BeliefAtom, Weight>), representing the mapping from belief atoms to their associated weights. It also contains a class variable called totalWeight, representing $T_W = \sum_{a \in At} \max(W(a))$, i.e. the total of all maximum weights associated with each atom a. This value is used for determining the λ-value of a formula and is updated easily during belief revision. The hash map ensures that every belief atom is unique. When a compact epistemic state is instantiated it sets the domain in the EpistemicState class, initialises the weightedBeliefBase and sets totalWeight to zero.

The theory behind revising a compact epistemic state was outlined in Section 2.4 and Bauters et al. (2017) suggest that an implemented algorithm of complexity $O(log_2|At|)$ could be used. Figure A.3 shows the implemented revision strategy. The use of HashMap's and HashSet's provides constant-time $O(1)$ operations for the get(), containsKey(), put() and remove() methods. As a result, for a sequence of revisions $I = \langle i_1, ..., i_n \rangle$, each taking the form (l, μ), where $l \in Lit$ and μ is the associated weight, the revision algorithm is linear-time $O(n)$.

The algorithm implemented in the EpistemicState class for entailment is inherited by the CompactEpistemicState class. However, the algorithm requires the child class (CompactEpistemicState) to implement methods for determining the λ-value of a given formula. Figure A.3 shows the implemented method getLambda(LogicalExpression). It takes a formula $\phi \in \mathscr{L}_{\geq}$ as an input and checks that it is ground. It then pares down the formula using the pare(LogicalExpression) method inherited from the EpistemicState class. This simplifies the formula into the language $\phi \in \mathscr{L}$ where the λ-value can be determined directly.

If any of the qualitative operators ($\phi > \psi$, $\phi \geq \psi$ or $not\phi$) do not hold then the pare() method will return a Contradiction. Next the formula is converted to Negation Normal Form (NNF) if it is not already in NNF. The λ-value is then determined by traversing the formula tree and collecting the bounded literals as outlined in Definition 2.4.10.

The formula's type is recursively checked (e.g. conjunction, disjunction, greater than) and then passed to the relevant getLambda() method. If the formula is a disjunction then the maximum value of the two operands is returned. If the formula is a conjunction then it defines what literals are true and adds them to the set of bounded literals.

If a formula has been reduced to a belief literal then it is evaluated by considering the set of bounded literals. If the set of bounded literals contains the negation of the belief literal then it is evaluated as inconsistent and returns the λ-value of a contradiction. If it is not inconsistent then the weight is calculated as follows: starting from the sum of all maximum weights associated with each atom (T_W), the maximum weight associated with each bounded literal is removed and then the corrected weight of each bounded literal is added back. The result is the λ-value of the

original formula.

If the formula ϕ is a tautology then the maximum weight associated with the models of ϕ is returned. Conversely, if it is a contradiction then the minimum weight ($-\infty$) is returned. For all other formulas ϕ that are not contained in the language $\mathscr{L}$ (e.g. $not\phi$, $\phi > \psi$, $\phi \geq \psi$, $\phi = \psi$ and $\phi \neq \psi$), their λ-values are determined by $\lambda(\phi) = \lambda(pare(\phi))$.

The implementation of epistemic states detailed above is capable of modelling beliefs instantiated with probability theory and possibility theory. Figures A.4 and A.5 show the Java classes for the implemented probabilistic and possibilistic compact epistemic state instantiations, including their modified revision strategies and methods for determining the λ-value of a logical expression (formula). The epistemic state and compact epistemic state classes provide a general framework that can be used to instantiate any other epistemic state representations. Future work could address implementations of epistemic states instantiated with Dempster-Shafer theory.

3.5.2 Global Uncertain Belief

Here we discuss how an agents belief base can be implemented when utilising epistemic states as the underlying representation of beliefs. As described in Section 2.4, a GUB is a set of logical formulas over the language $\mathscr{L}_{\geq}$ that also supports belief revision. As such, a GUB offers tractable modelling and revision of uncertain beliefs in the BDI setting.

A GUB $\mathscr{G}$ is defined as a set $\mathscr{G} = \{\Phi_1, ..., \Phi_n\}$ where each Φ_i is an epistemic state over a domain $A_i \subseteq At$ i.e. a partition of the GUBs domain At. Therefore a GUB is defined by it's domain and a set of epistemic states.

Figure 3.8 shows the class diagram for the GlobalUncertainBelief class that was implemented. It contains two class variables: one named "domain" of type HashSet<BeliefAtom>, representing the entire domain of the GUB and one named epistemicStates of type HashMap<HashSet<BeliefAtom>, CompactEpistemicState>, representing a mapping from each compact epistemic state's domain to itself.

The main methods for implementing the GUB class are detailed below:

- **addEpistemicState(CompactEpistemicState)** - this methods adds an epistemic state to the set of epistemic states associated with the GUB. This method is used by the agent parser when configuring an agent and adds all of the agents initial beliefs (epistemic states) to its GUB.

- **getRelevantEpistemicStates(LogicalExpression)** - this method returns all of the epistemic states from the GUB that are relevant to the logical expression (formula) passed to it. It is utilised by the entails() methods in the global uncertain belief class to retrieve the relevant epistemic states. These epistemic states are then individually used to check if the logical expression (formula) is entailed by them.

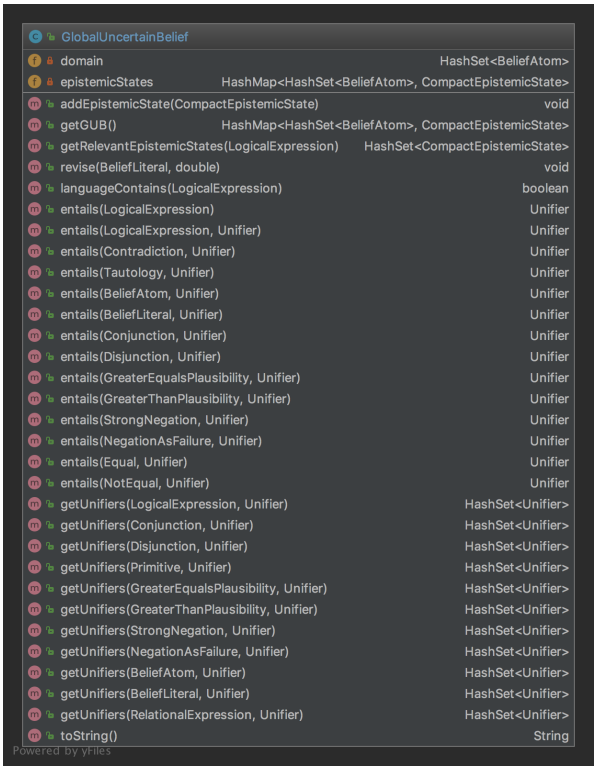


Figure 3.8: GlobalUncertainBelief Class Diagram

- **revise(BeliefLiteral, double)** - this method is used to revise the GUB for a given belief literal and weight pair (l, μ). Definition 2.4.6 defines GUB revision with an input formula ϕ and associated weight. However, as only compact epistemic states were considered the revision method was only required to revise the GUB for inputs of the form (l, μ). This method is used by the executeAction() method in the revise belief action class. This method would be integrated with an agents perception system and would be called when an agent wishes to revise it's belief base with newly perceived information. It can also be called if a revise belief action is present in a plan body. The method is implemented as follows:

  1. First it checks that the belief literal is grounded (contains no free variables). It does this for the same reasons mentioned for epistemic state revision.
  2. It then loops through each epistemic state and checks if the belief atom associated with the input belief literal is contained within the epistemic state's domain.
  3. If it is, then it revises that particular epistemic state (with it's own revision strategy) and updates the GUB with the new epistemic state.

- **languageContains(LogicalExpression)** - this methods checks that the formula (represented as a logical expression) is contained in the language $\mathscr{L}_{\mathscr{G}}$. Any formula $\phi \in \mathscr{L}_{\geq}^{A_i}$ for $i \in \{1, k\}$ (the epistemic states in the GUB) is also a formula in $\mathscr{L}_{\mathscr{G}}$. This method

iterates through the epistemic states contained in the GUB and calls the languageContains(LogicalExpression) method within the epistemic state class. This returns true if all of the belief atoms in the formula are contained within one of the epistemic states and false otherwise.

- **getUnifiers(LogicalExpression)** - this method returns the set of possible unifiers for a logical expression (formula) with the GUB. It is called by the entails() method in the global uncertain belief class because it needs to iterate through each possible unifier to check for entailment. This method first retrieves all of the free variables from the logical expression. It then iterates through each of the relevant epistemic states for the logical expression and retrieves all of the possible substitutions that each variable could take. A recursive method was then used to create the set of possible unifiers. Each unifier contains a different combination of substitutions for each free variable. This method has a significant impact of the computational complexity of the implementation but performs better than the original depth first search algorithm that was implemented. Obtaining the set of possible unifiers limits the practicability of the extended AgentSpeak(L) language and will be discussed further in the discussion.

- **entails(LogicalExpression)** - this method is used to check if a logical expression (formula) is a logical consequence (entailed) by the GUB and returns the unifier that made it a logical consequence. This method is called when selecting applicable plans from the set of relevant plans. This is where the plan context is evaluated to check if it is a logical consequence of the belief base and therefore an applicable plan. If it is a logical consequence then the plan is added to the set of applicable plans and substituted with the unifier that was found.

  This method required finding all of the possible unifiers of the formula with the GUB and then for each one attempting to traverse the formula tree and check for entailment. A preorder traversal was used to traverse the formula tree and check for entailment. A preorder traversal is logical here because it evaluates a particular node before proceeding to it's children. If successful it will return a unifier, backtrack and then evaluate the next node at the same depth, thus preventing the need to evaluate it's children. If unsuccessful (and a binary expression) then the left and right child expressions will be evaluated (in that order). If a particular node returns a null unifier then the traversal is aborted and restarted with the next possible unifier. This method is an implementation of Definition 2.4.5 and works as follows:

  1. First, it substitutes the unifier it is passed into the logical expression (formula).
  2. It then creates a HashSet of all the possible unifiers using the getUnifiers() method. The HashSet ensures that no duplicates are present in the set.
  3. It then attempts to find an applicable unifier by iterating through the HashSet and performing a preorder tree traversal with following steps for each unifier:

a) It checks what type of logical expression the formula is e.g. conjunction, disjunction, equals etc. The formula and the unifier are then passed to the appropriate entails() method for that type of logical expression (formula).

b) In accordance with Definition 2.4.5 these methods are implemented as follows:

   i. A **tautology** returns the unifier that it was passed as it signifies that the formula is entailed.

   ii. A **contradiction** returns null as it is not entailed and their is no applicable unifier.

   iii. For **belief atoms**, **belief literals**, **strong negations**, **negation as failures**, **equals** and **not equals**, the formula is first grounded using the possible unifier that it was passed. It then checks that the formula is contained in the language $\mathscr{L}_{\mathscr{G}}$ by using the languageContains(LogicalExpression) method. If it is, then it iterates through the relevant epistemic states (found using the getRelevantEpistemicStates() method) and passes the formula and possible unifier to the entails() method within the epistemic state class. This utilises the pare() and getLambda() methods introduced earlier and returns an applicable unifier if the formula is entailed and *null* otherwise. If the unifier is not *null* then the tree traversal is continued and it is aborted if it is *null*.

   iv. For **conjunctions** the same process is followed, however, if the formula is not contained in the language $\mathscr{L}_{\mathscr{G}}$ then the operands are evaluated separately. This is in line with the preorder tree traversal. First there is an attempt to evaluate the conjunction directly but if this fails then the left and right child expressions are evaluated. The left conjugate is passed into the entails() method with the possible unifier. The entails method is thus recursive. If the left conjugate is entailed with the possible unifier it will return a new unifier containing further substitutions. Otherwise it will return *null*, indicating that the formula is not entailed with the possible unifier. The right conjugate is then passed into the entails() method with the unifier returned from the call to the entails() method with the left conjugate and possible unifier. This returns a unifier that determines if a formula is entailed by the GUB. If the unifier is *null* then it indicates that the formula is not entailed and the tree traversal is aborted. Otherwise the traversal is continued with the new unifier.

   v. For **disjunctions** a similar approach is followed, however, this time only one of the operands (disjuncts) is required to return a unifier. If one of the operands returns a unifier then the algorithm backtracks and evaluates the next node.

   vi. For **greater than** expressions the same process is followed as for belief atoms

(i.e. it attempts to evaluate the expression directly), except for if the formula is not contained in the language $\mathscr{L}_{\mathscr{G}}$. In this case the formula is checked to ensure that it is ground and that it's left and right operands are both contained within the language $\mathscr{L}_{\mathscr{G}}$. If they are, then for each operand the relevant epistemic states are found for their strong negations. The epistemic states associated with the left and right operands are compared and if they are not of the same type then an exception is thrown. This is because epistemic states instantiated with different uncertainty theories cannot be compared with qualitative operators. The λ-values associated with the two epistemic states are then retrieved using the getLambda() method from the compact epistemic state class. These two λ-values are then compared in accordance with Definition 2.4.2. If the left operand's λ-value is less than the right operands λ-value then the possible unifier is returned, signifying that the expression is entailed. This results in the next node being evaluated. Otherwise *null* is returned and the tree traversal is aborted.

vii. For **greater equals than** expressions the same process is followed as for greater than expressions, except that in the comparison of the λ-values the $\leq$ operator is used instead of $<$.

4. If for a given possible unifier the evaluation of a node returns *null*, then the tree traversal is aborted and restarted with the next possible unifier.

5. If for a given possible unifier the whole formula tree has been traversed and the resulting unifier is not *null*, then the formula is said to be entailed by the GUB. The relevant plan is then added to the set of applicable plans with the resulting unifier substituted in it's plan body.

3.6 Goals

Now that the main extensions have been detailed (logical expressions and the belief base) the remainder of this section will outline how the rest of the extended version of AgentSpeak(L) was implemented. It will focus on how the new belief base and language for writing plan contexts and test goals was incorporated into the remainder of the implementation.

AgentSpeak(L) defines two types of goals, achievement goals and test goals. Achievement goals are formulated by an atomic formula prefixed with a '!'. Test goals are formulated by an atomic formula prefixed with a '?'. Achievement goals define states of the world that the agent wants to achieve where the atomic formula is evaluated as true. Test goals state that the agent wishes to check whether the atomic formula can be unified with it's belief base.

Achievement goals are found in plan body's (taking the form of an action) which when executed add an equivalent internal event (representing a goal the agent wishes to achieve) to the event set.

Such events can then trigger plans as relevant plans by unifying with event triggers containing the same achievement goal. The functionality of achievement goals is thus dependent on whether they are associated with an event trigger or an action. Test goals, however, are only found in plan body's as actions. They are used to further instantiate free variables in the remainder of the plan body by unifying themselves with the belief base. Goals were implemented the same in both AgentSpeak(L) and the extended version and the class inheritance structure can be seen in Figure 3.9.

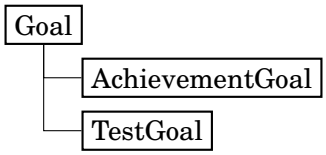


Figure 3.9: Class inheritance structure for AgentSpeak(L) goals

In AgentSpeak(L) the belief base consists of a set of belief literals and test goals take the form of first-order logic terms. Test goals were implemented as terms which meant that they could be tested for unification with each of the terms associated with the belief literals in the belief base. In the AgentSpeak(L) implementation all goals contain a term as their main class variable. As goals are found in both event triggers and in plan body's they require a substitution method that can replace any variables with the relevant substitution. Below is an example of a test goal (in a plan body) in AgentSpeak(L):

```
1 // Belief Base
2 at(location(1)).
3
4 // Plan Body
5 ?at(location(X)); move(location(X)).
```

The test goal consists of a term and would return a unifier containing the first substitution between $at(location(X))$ and the belief base. For example, if the belief base contained the belief atom $at(location(1))$ then it would return the substitution $\{X/1\}$. This would lead to the remainder of the plan body being instantiated with the substitution $\{X/1\}$, resulting in the action $move(location(1))$.

In the extended implementation test goals contain class variables for both terms and logical expressions (formulas). However, as the belief base no longer contains belief literals it is not possible to use test goals as terms (representing belief atoms). In future work the belief base could be extended to incorporate both a set of belief literals and a GUB. Test goals could then take the form of either terms or logical expressions. However, in this implementation test goals must take the form of a logical expression as shown in the following example:

```
1 // Belief Base
2 **(at(location(2)), 0.9).     // probabilistic
3 **(at(location(1)), 0.1 ).    // probabilistic
4
```

```
// Plan Body
?at(location(X)) > at(location(Y)); move(location(X)).
```

In this scenario the test goal would check if an epistemic state in the belief base entailed the formula $at(location(X)) > at(location(Y))$. It would return the substitution $\{2/X, 1/Y\}$, resulting in the remainder of the plan being instantiated as $move(location(2))$. The functionality surrounding goals will be detailed in the sections on event triggers and actions when discussing their corresponding event triggers and actions.

3.7 Triggering Events

When an AgentSpeak(L) agent acquires new goals or perceives it's environment it may trigger the addition or deletion of goals or beliefs. These are referred to as triggering events. There are four triggering events defined in AgentSpeak(L): the addition and deletion of both beliefs and goals. The class inheritance structure shown in Figure 3.10 was used to implement triggering events for AgentSpeak(L).

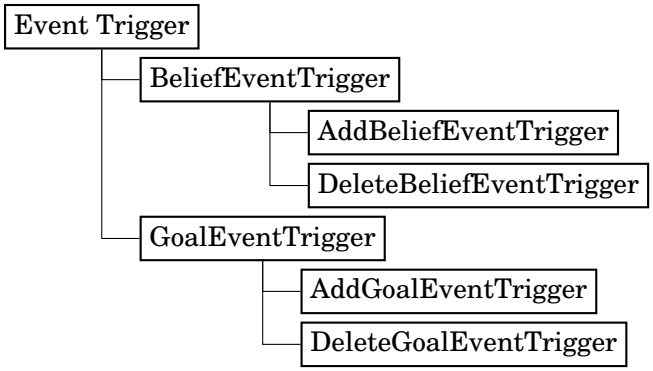


Figure 3.10: Class inheritance structure for AgentSpeak(L) event triggers

In the extended implementation there are only three triggering events as the addition and deletion of beliefs is no longer supported. Instead, agents are only capable of revising their beliefs. However, these classical rules can be defined as belief revisions: $\circ(\phi, \max_G)$ for belief addition and $\circ(\phi, \min_G)$ for belief deletion.

The main role of the event trigger class is to unify events in the event set with relevant plan's event triggers. This functionality is used by the agent's interpreter to select all of the relevant plans from the plan library for a given event. As a result, the event trigger class is used by plans for their event triggers and by events in the event set. The class inheritance structure and class diagrams used to implement event triggers are shown in Figure 3.11. The belief event trigger and goal event trigger classes contain variables for their associated belief literal or goal respectively.

The revise belief event trigger class also contains a class variable of type term, representing the corresponding revision weight. This was implemented as a term for unification and substitution purposes. For example, an agent may perceive it's environment and revise it's belief

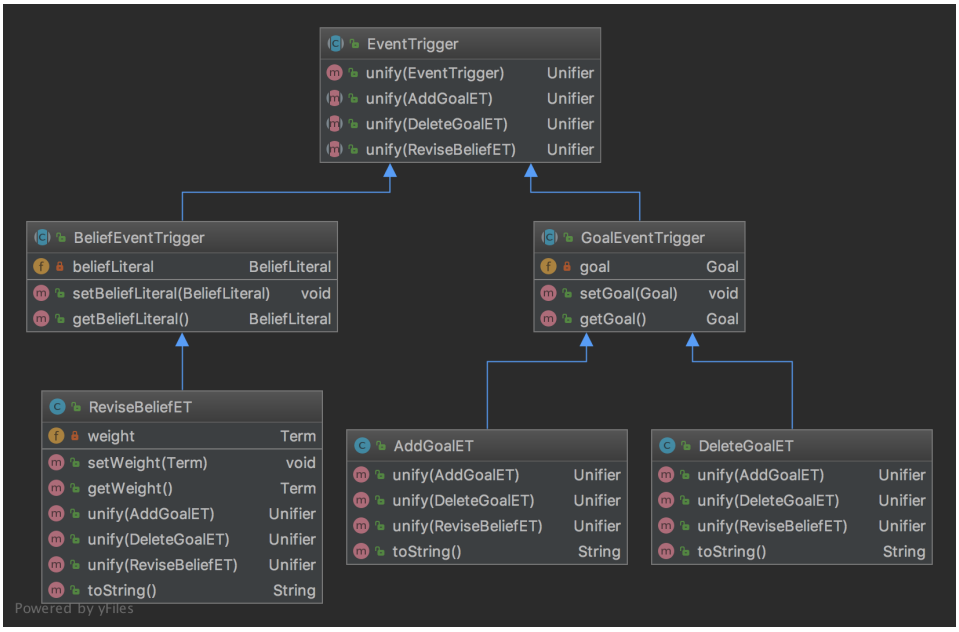


Figure 3.11: Event Trigger Class Diagram

base with $*(water(location(1)), 0.9)$ and add a corresponding external event with a revise belief event trigger to the event set. This indicates that it believes their is water at location(1) with a probability of 0.9. A plan in the plan library may take the form:

```
1 *(water(location(X)), W) : at(location(X)) > 0.9 && W >= 0.9 <- collectWaterSample.
```

If the event is then selected from the event set then this plans event trigger would unify with the event trigger associated with the selected event and generate the relevant unifier $\{X/1, W/0.9\}$. The plan would then become a relevant plan and it's context (substituted with the relevant unifier) would then be evaluated.

3.7.1 Unification

The implementation of event trigger unification is detailed here. Event triggers only unify if they are of the same type, for example, two add goal event trigger's, two delete goal event triggers or two revise belief event trigger's. If two event triggers of different types attempt to unify the unify() method will return null. If the event triggers are of the same type then the unify() method operates as follows:

- For **goal event triggers** the unify() method retrieves the goal associated with each event trigger. It then retrieves the term associated with each goal and calls the term's unification algorithm, which returns a unifier if successful or *null* otherwise.
- For **revise belief event triggers** the unify() method retrieves the belief literal associated with each event trigger. It then calls it's associated belief literal's unify() method, passing

to it the other belief literal. Belief literal's only unify if they are either both positive or both negative. The belief literal's unify() method then retrieves the belief atom associated with each belief literal and then the term associated with each belief atom. It then calls the term's unification algorithm, which returns a Unifier if successful or *null* otherwise. If this Unifier is not *null* then the algorithm retrieves the weights of each event trigger (which are term's) and performs substitution using the previously found unifier. These new substituted weights (which are still term's) are unified by calling their unification method. If this unification is successful then the unifiers are combined and returned, otherwise *null* is returned.

The implementation of goal event trigger's was the same in AgentSpeak(L) and the extension. However, as detailed above, the add and delete belief event trigger classes were removed and replaced with the revise belief event trigger class. The add and delete belief event trigger classes utilised a unification algorithm very similar to that of the goal event trigger's. The only difference being that the term's were retrieved from the associated belief atom and not from an associated goal.

The new event trigger can be utilised in the same way as both the add and delete event triggers and also possesses more expressive functionality. For example, a plan's event trigger may only unify with the selected event if the weight associated with the revision is of a specific value. The plan library could therefore contain multiple plans with revise belief event triggers for the same belief literal but with different associated weights.

The weight associated with a plan's revise belief event trigger may take the form of a variable which is instantiated by the weight associated with the selected event. Multiple plan's could then become relevant and their context's would be instantiated with the substitution. Their context's could then reason over the value of this weight and the appropriate plan would be selected as an applicable plan. For example, an agent that is trying to collect water samples may have a compact probabilistic epistemic state regarding the presence of water at different locations. This agent may then have the following plans in it's plan library.

```
1 *(water(location(X)), W) : W > 0.7                <- +!collectWaterSample(location(X)).
2 *(water(location(X)), W) : W >= 0.3 && W <= 0.7   <- +!perceive(location(X)).
3 *(water(location(X)), W) : W < 0.3                <- +!explore.
```

These plans represent three different cases:

- **W > 0.7** - This indicates that the agent is fairly certain that there is water at location(X) and thus creates an internal event with an achievement goal to collect a water sample from location(X).
- **W >= 0.3 && W <= 0.7** - A probability value of 0.5 indicates that the agent is ignorant about the presence of water at location(X). It therefore creates an internal event with an achievement goal to perceive location(X), with the hope of strengthening or weakening it's belief.

- **W < 0.3** - This represents the case that the agent is fairly certain that there is no water present at location(X). It therefore creates an internal event with an achievement goal to continue exploring it's environment (and ignore location(X)).

3.8 Actions

In AgentSpeak(L) the body of a plan consists of a sequence of actions that can take different forms. An agent has a predefined set of actions that it can perform on it's environment and these are referred to as environment action's. These actions are represented as first order terms, such as $move(location(2))$. The action is defined by the atom $move$ and has an argument $location(2)$ that provides a specific instance of the action. Environment actions inform the relevant section of the agent's control architecture, which then performs the action. In simulated environments the action is defined as part of the environment.

AgentSpeak(L) also defines actions for test goals (?), achievement goals (?), addition of belief literals ($+b$) to the belief base and deletion of belief literals ($-b$) from the belief base. In the extension there is no addition or deletion of beliefs from the belief base but instead revision of the GUB with a belief literal and associated weight. Figure 3.12 shows the class inheritance structure that was used to implement AgentSpeak(L) actions. Figure 3.13 shows the class inheritance structure and the class diagrams for the implemented actions in the extended version.

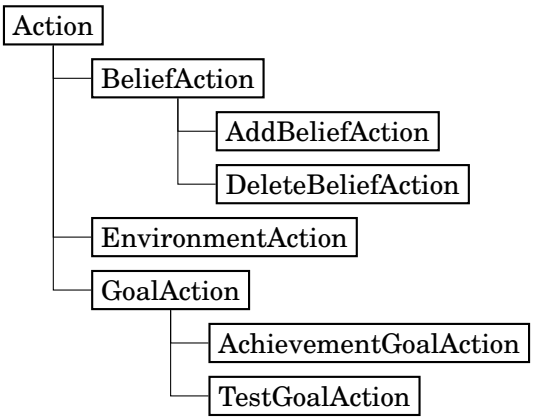


Figure 3.12: Class inheritance structure for AgentSpeak(L) actions

All action classes contain an executeAction(agentName, intention, unifier, beliefBase, eventSet, environment) method, which performs the required action and returns the unifier to be used for subsequent actions in that plan body. All of the executeAction() methods return the original unifier that they are passed except for the test goal action executeAction() method, where any further variable instantiations are added to the unifier that is returned. The executeAction() method for each action class is detailed below:

- **EnvironmentAction** - For environment actions the executeAction() method starts by

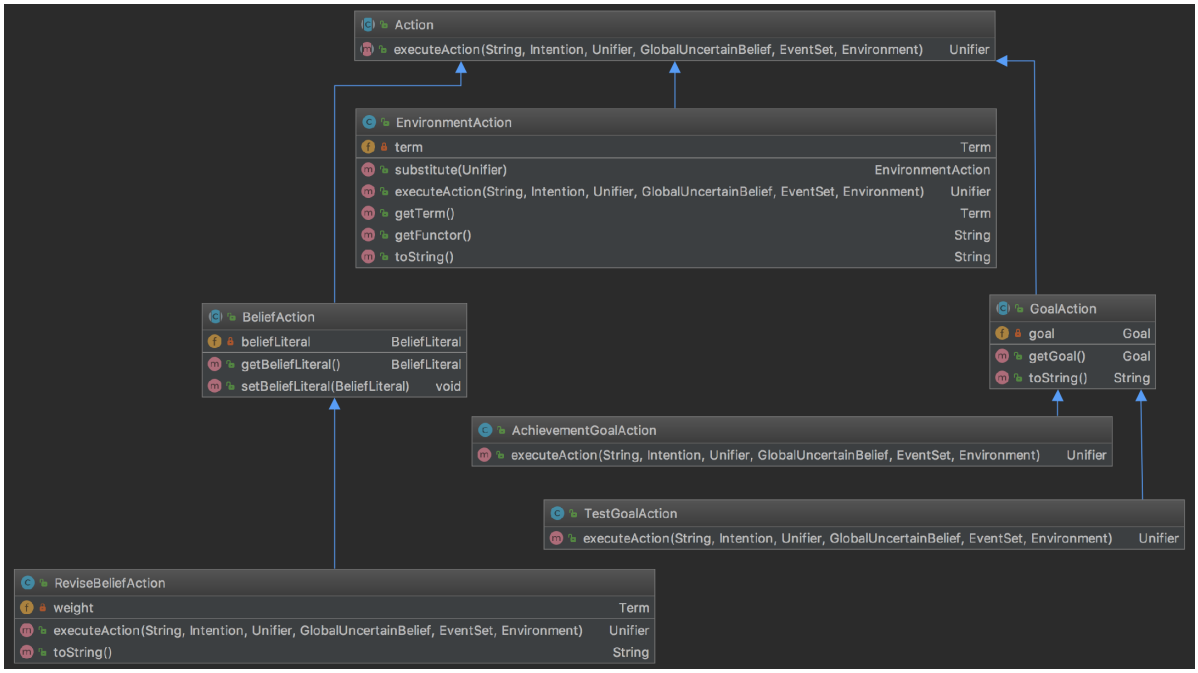


Figure 3.13: Action Class Diagrams

substituting the unifier into the action. In non simulated environments the substituted action would be passed to the agent's control architecture and the relevant action would be executed. However, this platform uses a simulated environment so it then calls the scheduleAction() method from the environment class and passes it the agents unique name and the substituted action. The scheduleAction() method is synchronised, ensuring that only one agent (thread) can execute the method at any given time. This prevents the environment from experiencing a corruption of state. It also ensures that any changes to the environment (by a single agent) are visible to all other agents (threads).

- **AchievementGoalAction** - For achievement goal actions the executeAction() method substitutes the applicable unifier into the achievement goal. This is the unifier that made the plan's context a logical consequence of the belief base. It then generates an internal event, which requires an event trigger and an intention.
  - A new add goal event trigger is instantiated with the goal associated with the achievement goal action. This is used as the event trigger.
  - The intention that was passed to the executeAction() method is used as the intention because internal events are associated with the intention that created them.

  This internal event is then added to the agent's event set.

- **TestGoalAction** - For test goal actions the executeAction() method simply retrieves the logical expression (formula) associated with the test goal action and passes it to the belief

base's entails() method along with the unifier it is to be instantiated with. The entails() method returns a new unifier with further variable instantiations which is then returned by the executeAction() method. This new unifier is then used for variable instantiation in the rest of the plan body.

- **ReviseBeliefAction** - For revise belief actions the executeAction() method first substitutes the unifier into both the belief literal and the weight (which is a term) associated with the revise belief action. It then calls the revise() method of the GUB and passes it the substituted belief literal and weight, which is now a double number. It then instantiates an external event, which requires an event trigger.
  - A new revise belief event trigger is instantiated with the substituted belief literal and weight associated with the action.
  - The instantiation of an external event does not require an intention because they generate their own, representing a new focus for the agents acting in the environment.

  This external event is then added to the agent's event set.

3.9 Plans

As discussed in Section 2.2, AgentSpeak(L) agents consist of a predefined set of plans. Each plan represents an agent's response to a given scenario. Plans consist of three components:

1. **Event trigger** - The AgentSpeak(L) interpreter operates by selecting an event to respond to from it's event set. A plan's event trigger determines when a plan is a relevant plan for handling that event. If the plan's event trigger unifies with the event trigger associated with the event then it is added to the set of relevant plans.
2. **Context** - A plan's context is used to select plans that are applicable to the agent's current state from the set of relevant plans. The agent performs detailed reasoning by checking if the context is a logical conqeuence of it's current belief base. If it is, then the plan is selected as an applicable plan. Plans are applicable in different scenarios dependent on the agent's current belief base.
3. **List of actions** - The body of a plan consists of a sequence of actions or (sub)goals that the agent should perform. It specifies goals that the agent wishes to achieve or test and actions that it should execute.

Figure 3.14 shows the class diagram for the implementation of plans in the extended version of AgentSpeak(L). Plans were implemented almost the same in AgentSpeak(L) and the extension. The plan class consists of three class variables, representing the plan's event trigger, context and

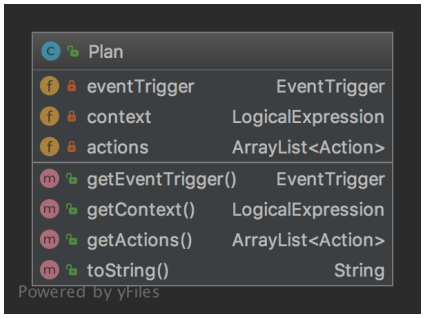


Figure 3.14: Plan class diagram

sequence of actions. The event trigger was of type event trigger and the context of type logical expression because they take the form of any formula from language $\mathscr{L}_{\geq}$. In AgentSpeak(L), however, the context took the form of a queue of belief literals. This was because the context was simply a conjunction of belief literals and thus did not require any complicated methods for entailment, just the ability to iterate over each belief literal. The plan body (sequence of actions) was implemented as an array list of actions as it provided O(1) complexity for it's get() and add() methods. When executing a plan body the interpreter simply accesses the relevant action using the get() method. Its uses an integer index maintained as a simple counter.

The enhanced plan selection capabilities can be showed by means of an example. Below are two plans, the first written in AgentSpeak(L) and the second written in the extended version. The goal of the plans is to explore mars and find water or ice samples.

```
1 +!exploreMars : water_or_ice(location(X)) && at(location(Z)) <- !proceed(location(Z),location(X)).
```

```
1 +!exploreMars : water_or_ice(location(X)) >= water_or_ice(location(Y1)) && water_or_ice(location(X))
      >= water_or_ice( location(Y2)) && water_or_ice(location(X)) >= water_or_ice( location(Y3)) &&
    at(location(Z)) >= at(location(X)) && X \== Z && X \== Y1 && X \== Y2 && X \== Y3 && Y1 \== Y2
    && <- !proceed(location(Z),location(X)).
```

An agent has a set of predefined plans, known as the plan library. The plan library class was implemented as an array list of plans. The only operation applied to the plan library consists of iterating through each plan, in an attempt to unify each plan's triggering event with the event that the agent has selected to respond to. Array list's provide constant time complexity when performing iterator operations.

3.10 Operational Semantics

So far the underlying components and structure of "Uncertain" AgentSpeak(L) have been introduced. This section will now focus on how these components and their mechanisms are utilised by the agent to achieve autonomy i.e. how the agent's interpreter functions. Agent's also manage a set of events and a set of intentions that are used by it's interpreter. These will be introduced here.

3.10.1 Events

In the original AgentSpeak(L) events can be defined as either external or internal, dependent on whether they are generated due to the agents perception of it's environment or from a subgoal within a plan. This is the same in the extended language and the class inheritance structure and diagrams used to implement events are shown in Figure 3.15. The event class consists of two class variables, an event trigger and their corresponding intention. External events generate their own new intention and internal events inherit the intention from the plan that generated them.

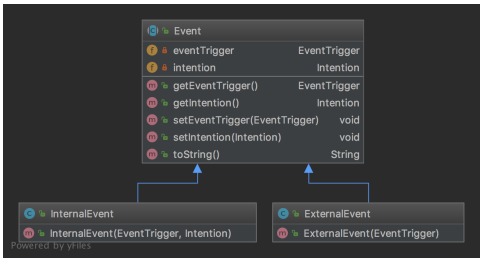


Figure 3.15: Uncertain AgentSpeak(L) Event class diagram

Agents manage a set of events and use it to select plans from their plan library that are relevant to that particular event. The event set is required to add and remove events, however, the event selection function is not defined in AgentSpeak(L). In Uncertain AgentSpeak(L) the selectEvent() method was implemented as a queue. This was achieved by implementing the event set as a linked list. It's add method appends to the end of the list and it's poll method retrieves and removes the head (first element) of the list. The event set was implemented as a linked list because adding and removing elements provides O(1) performance. The selectEvent() method is implemented in the agent class so that a programmer can easily modify it by extending the agent class and overriding the method.

3.10.2 Intentions

Intentions represent an agent's course of action to respond to a particular event. They take the form of a stack of partially instantiated plans. Figure 3.16 shows the class diagram for the implementation of intentions.

In order to implement intentions first the intended means class shown in Figure 3.17 had to be created. The intended means class represents a partially instantiated plan and so contains class variables for a plan and unifier. It also contains a class variable (index) that is used by the class for selecting the relevant action from the plan. The plan body consists of a list of actions and once an action is executed the intended means class increments it's index variable to signify that the next action should be considered.

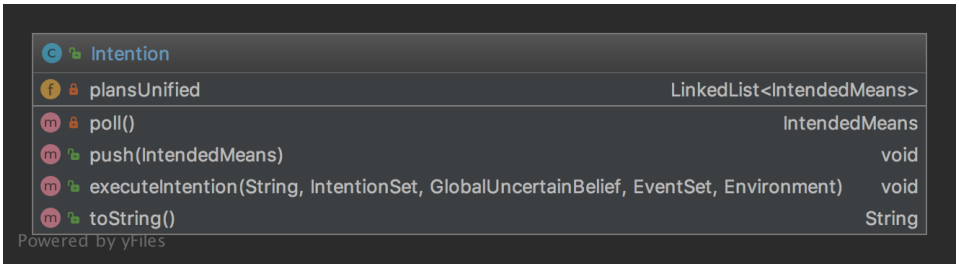


Figure 3.16: Uncertain AgentSpeak(L) Intentions class diagram

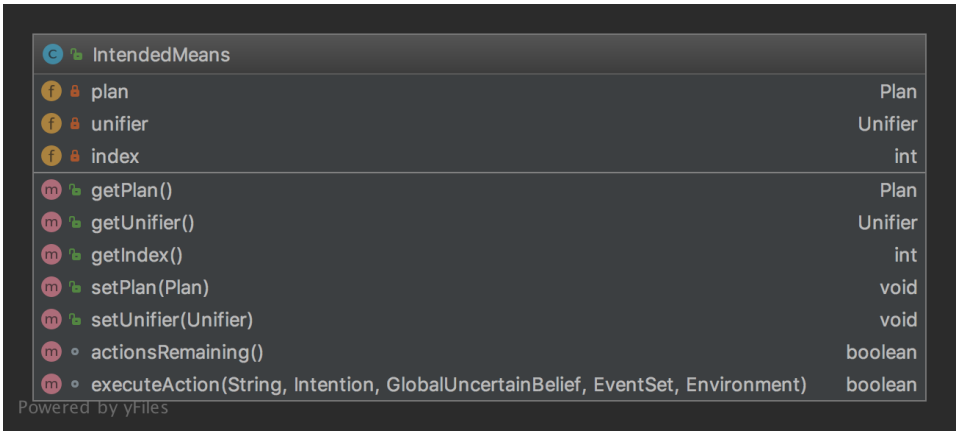


Figure 3.17: Uncertain AgentSpeak(L) Intended Means class diagram

The main functionality of the intended means class is to execute the relevant action from the plan body. The executeAction() method selects the relevant action from it's plan's body and calls it's executeAction() method, which performs the required action. The action's executeAction() method returns a unifier that the intended means sets as it's unifier and will use in the execution of subsequent actions. All actions return the original unifier except test goals that return a new unifier with any extra substitutions added.

The intention class contains a variable called plansUnified, representing the stack of partially instantiated plans associated with an intention. This was implemented as a linked list of intended means as linked lists can act as stacks with add and remove methods of performance O(1). The add method adds the intended means to the end of the list and the pollLast() method retrieves the last element from the list, essentially acting as a stack.

The main functionality of an intention is it's execution, which is performed by it's executeIntention() method. This method retrieves the intended means from the top of the stack and calls it's executeAction() method. This causes the next action from the intended means to be executed and if an achievement goal action is executed then the method returns true and false otherwise. If it returns true (i.e. an achievement goal is created) then the agent should stop executing the current intended means and restart it's interpretation cycle by selecting a new event from the event set. For this reason if it returns true then the executeIntention() method finishes. If it

returns false then it signifies that the intended means has been executed and so it is removed from the top of the intention. When an intention is selected from the intention set it is removed from the intention set. For this reason if not all of the intended means have been executed then the intention is re-added to the intention set.

At a high level there are no major differences between the implementation of intentions in AgentSpeak(L) and Uncertain AgentSpeak(L). The only differences occur at a lower level in the structure of plans and the mechanisms associated with the execution of different types of actions.

The set of intentions is implemented as a stack using a linked list (for similar reasons). The intention selection function is not defined in AgetnSpeak(L) and is left for the user to define. In Uncertain AgentSpeak(L) the selectIntention() method simply pops an intention from the top of this stack. Again, this method was implemented in the agent class so that a programmer can easily modify the method.

3.10.3 Agent Interpreter

An AgentSpeak(L) (and Uncertain AgentSpeak(L)) agent is defined as a class containing it's interpreter. Figure 3.18 shows the class diagram for the Uncertain AgentSpeak(L) agent class.

In AgentSpeak(L) an agent consists of a belief base, a pre-defined plan library, a set of events and a set of intentions. In order to develop a simulation environment, agents were also given a name, id, an environment that they could act on and perceive from and event listeners so that the GUI could visualise the agent's "thoughts". The main difference in Uncertain AgentSpeak(L) is that the belief base takes the form of a GUB and the plan library consists of plans written in the extended language.

The main difference in Uncertain AgentSpeak(L) is the agent's interpretation cycle, which is defined in the run() method. The three selection functions (event selection, plan selection and intention selection) were also implemented in the agent class for the reasons discussed previously. The remaining methods in the agent class were added for the development of the simulation environment and will be discussed in Section 4.1. Figure 3.19 shows the interpretation cycle for an Uncertain AgentSpeak(L) agent. This will now be discussed here.

1. **Event Selection** - When an agent is initialised it's initial goals are added to it's event set. At the start of an interpretation cycle the agent's selectEvent() method is called. As mentioned earlier this is implemented as a queue because it is not defined in AgentSpeak(L) but intended to be designed by the user. This method was implemented in the agent class so that it can easily be modified if the programmer requires. Extending the agent class and overriding the method would enable a programmer to define their own selection function.

2. **Plan Selection** - The agent's selectPlan(Event) method is then called and passed the event that was selected from the event set.

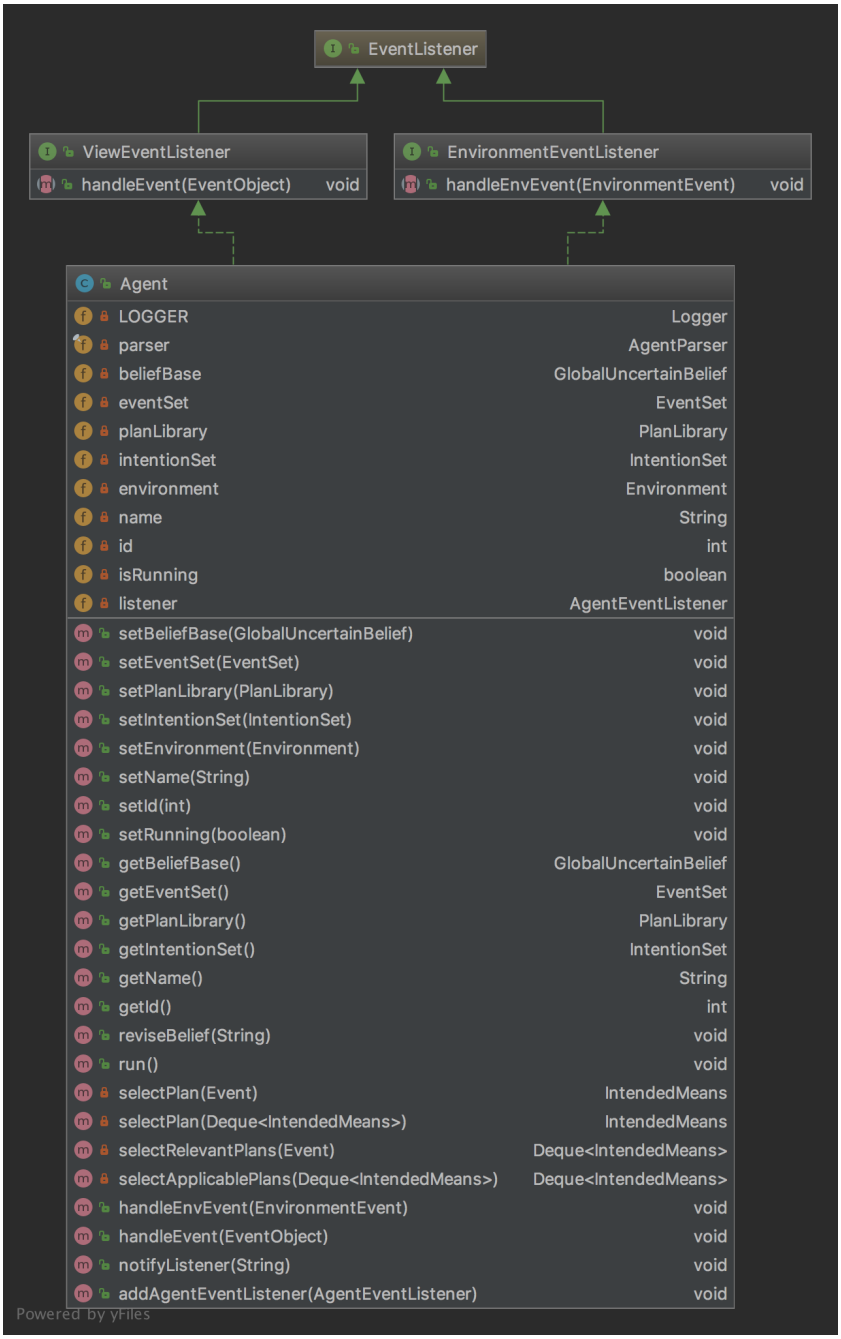


Figure 3.18: Uncertain AgentSpeak(L) Agent class diagram

a) **Relevant Plan Selection** - The agent's selectRelevantPlans() method is then called to select the the relevant plans from the plan library. This method iterates through all of the plans and calls the event trigger's unify() method in attempt to unify it with the plan's event trigger. This is the event trigger associated with the selected event. If unification is successful then it will return a unifier and *null* otherwise. If a unifier is

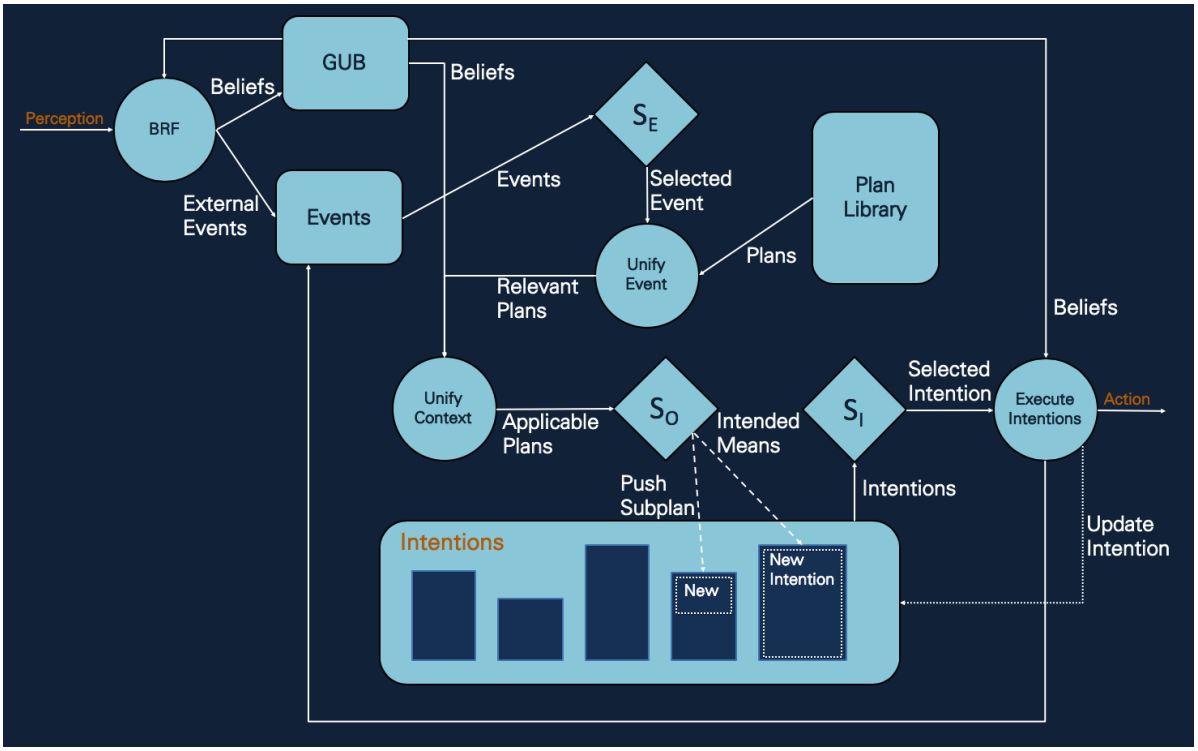


Figure 3.19: Uncertain AgentSpeak(L) interpreter

returned then a new intended means is created with the plan and the relevant unifier, which is then added to the set of relevant plans.

b) **Applicable Plan Selection** - If the set of relevant plans is not empty then the agent passes the set of relevant plans (set of intended means objects) to the agent's selectApplicablePlans() method. This method then pops the top intended means from the set of relevant plans one by one and calls the belief base's entails() method in an attempt to unify the plans context with the belief base. If successful this will return a unifier and a new intended means is created with the plan associated with the intended means and the new (applicable) unifier. This new intended means is then added to the set of applicable plans. Once all of the relevant plans have been tested the selectApplicablePlans() method returns the set of applicable plans (which is a set of intended means).

Finally, the selectOption(Deque<IntendedMeans>) method is called to select a single plan from the set of applicable plans. This selection function S_O is also not defined in AgentSpeak(L) and is left for the programmer to implement. In Uncertain AgentSpeak(L) this is implemented as a queue, giving priority to plans that are defined earlier in the plan library. Again, this selection function can easily be altered by a programmer by extending the agent class.

3. **Add Intention** - The selected plan is then added to the top of the intention stack associated with the selected event. If the event is external then the intention stack will be empty and if internal the intention stack will contain partially instantiated plans (intended means) associated with that particular event. The intention is then added to the intention set.

4. **Select Intention** - The agent then calls it's selectIntention() method that removes the last intention from the list. This is equivalent to popping the top element from a stack. This is logical as the agent should select a course of action (intention) for dealing with the most recent event. Again, this method can easily be modified by a programmer.

5. **Execute Intention** - The final step in the agent's interpretation cycle is to execute the selected intention. This is achieved by calling the executeIntention() method associated with the intention. This method was detailed earlier in Section 3.10.2.

CHAPTER 4

MAS SIMULATION ENVIRONMENT

This chapter details the research methodology that was employed in response to the initial research objectives regarding a platform for developing and simulating extended agents. Section 4.1 focuses on how the general simulation environment was constructed so that agents can perform actions on and perceive an environment. This is a general environment that a programmer would extend to implement a custom simulation environment. Section 4.2 then outlines how the agent programming language and the environment can be combined to define and simulate a MAS. In the final section an example mars exploration MAS that was implemented using the development environment is detailed and discussed.

For the readers convenience the overall system architecture is detailed in Figure 4.1. It gives a basic outline of how the different components interact with each other. The individual components will be discussed in more detail throughout this chapter.

Each of the agent files in Figure 4.1 is written in this extended language detailed in Chapter 3. When these files are parsed the JVM instantiates Agent objects for each agent defined in the MAS project file. Essentially, each agent block on the diagram consists of an extended AgentSpeak(L) interpreter that acts on and perceives the environment. The overall process for creating and simulating MAS in the implemented platform is detailed below:

1. The user creates:

    - Files for each type of agent they wish to specify. Each agent file contains the agent's initial beliefs, the goals they wish to achieve and their predefined plan library. These files are written in the extended AgentSpeak(L) programming language that is defined in Section 3.
    - An environment class for their system by extending the base environment class

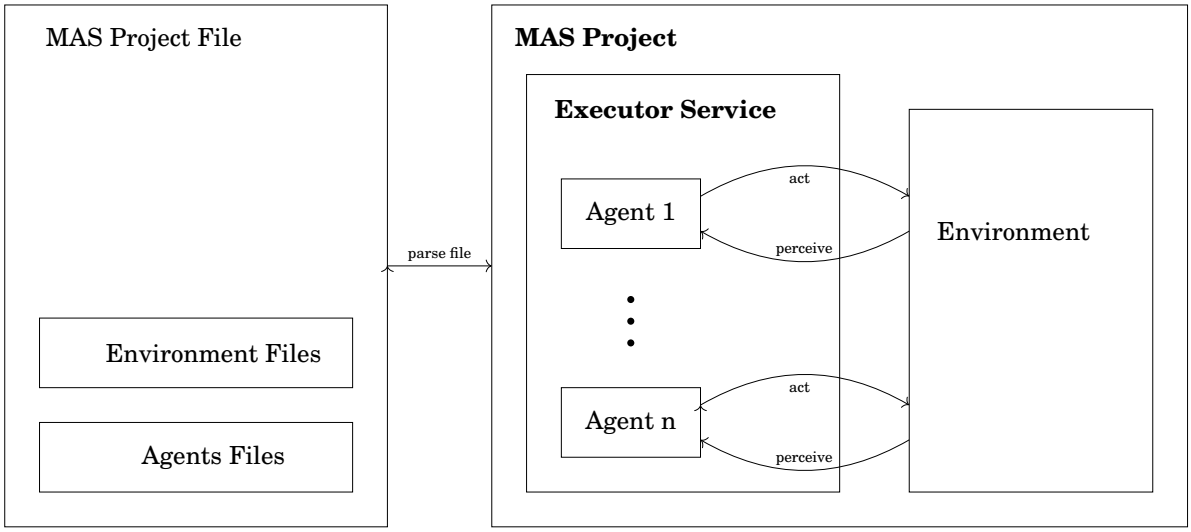


Figure 4.1: MAS project architecture

provided. This models how the agents actions effect the environment and how the agents update their percepts from the environment.

- A MAS project file that defines the projects environment and agents.

2. The MAS file is then parsed and an instance of the MASProject class is created. This instantiates:
   - The environment that was specified.
   - An executor service with a thread for each agent to run on.
   - The number of each type of agent specified in the project file. The extended AgentSpeak(L) parser is used to parse the agent files and create instances of the Agent class for each agent.
3. The MASProject class then runs all of the agents simultaneously using the executor thread.

The rest of this chapter will now introduce, describe and justify the implementation of the development environment.

4.1 Simulation Environment

Chapter 3 outlined the implementation of the Uncertain AgentSpeak(L) programming language. Agent's programmed in this language can be tested and deployed in real world systems by creating a custom interface between Uncertain AgentSpeak(L) and the system. The only requirement is that the hardware is capable of running the Java Virtual Machine. However, in most scenarios it is preferred to develop agents and test them in a simulated environment. This section will

detail the implementation of a general simulation environment. The simulation environment provides the core functionality for simulating agents. Programmers can develop custom simulation environments by extending the environment class. MASs can then be simulated by programming agents (in .agent files), extending the base environment class and writing a MAS configuration file (.mas) specifying the agents and environment to be simulated.

4.1.1 Environment

The main class for implementing simulated environments is shown in Figure 4.2. This class has two main functions: enable agents to act on the environment and to perceive the environment. Each agent is ran on a separate thread so the environment had to be able to function without any thread interference or memory consistency issues. The main functionality of the environment is detailed below.

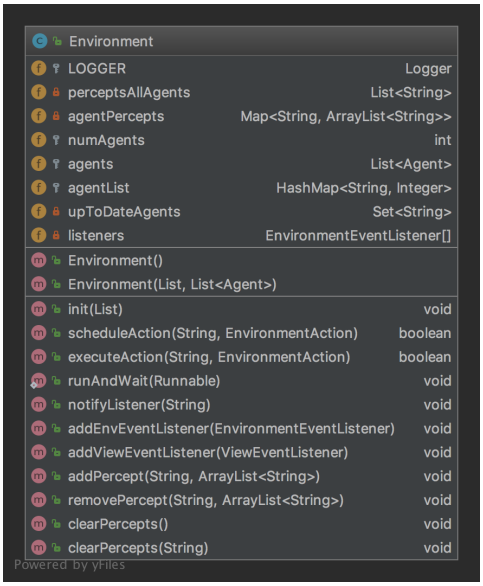


Figure 4.2: Environment class diagram

1. **Event Listeners** - In order for the environment to function and inform agents of their environment perceptions an EnvironmentEventListener interface was created. An accompanying EnvironmentEvent class was created that contains an array list of strings containing a particular agent's perceptions.

    a) The agent class implements the environment event listener and contains a method handleEnvEvent(EnvironmentEvent). This method iterates through the agent's perceptions and calls the agents reviseBelief(String) method, which parses the string and instantiates a new revise belief action object. It then calls the executeAction() method associated with revise belief action method.

b) The environment class also contains methods for adding listeners and notifying listeners.

2. **Perception** - The environment class has a class variable called agentPercepts. This is a concurrent hash map containing mappings from an agents name to an array list of strings containing that agent's percepts. This variable is used alongside event listeners to inform agents when they have new percepts.

   a) **upToDateAgents** - this class variable is a synchronised hash set that maintains a record of which agents have been sent their perceptions. When an agent is sent it's perceptions the agents name is added to the upToDateAgents set and when an agent receives a new perception it's name is removed. This ensures that only agents with new perceptions are sent their perceptions.

      The environment class also has four methods for dealing with an agents percepts:

      i. **addPercept()** - this method accepts a string containing the agents name and an array list of it's perceptions. This method adds the agents perceptions to the agentPercepts class variable. It should be used by a programmer to add perceptions.

      ii. **removePercept()** - this method removes a percept for a single agent and again should be used by the programmer whenever they wish to remove a perception from agentPercepts.

      iii. **clearPercepts()** - this method clears all perceptions from agentPercepts.

      iv. **clearPercepts(String)** - this method clears all the percepts for the agent whose name is passed to the method.

3. **Actions** - When an agent seeks to perform an action on the environment it calls the environment's scheduleAction() method and passes the environment it's name and the environment action it wishes to perform.

   a) The environment class has a method called executeAction() that a programmer must override in their environment class. This method is required to parse the environment action string and perform the required action on the environment. This method is synchronised to ensure that only one agent (thread) is able to modify the environment at a given time. This prevents issues with thread interference and memory consistency errors.

   b) The environment class also has a method called runAndWait() which accepts a runnable, queues it on the environment thread (JavaFX thread) and waits for it's execution. This ensures that the environment's view (discussed later) is updated without any issues.

The scheduleAction() method creates a runnable that calls the environments executeAction() method and then it's notifyListener(String) method. This runnable is then passed to the runAndWait() method that ensures all agents actions are performed without any interference and that the environments view is updated without freezing. It also ensures that the agent receives it's perceptions of the environment.

4.1.2 Grid World

When creating a custom environment for simulating agents it is usually preferred to maintain a model class for the environment that maintains all of the data and a view class that is used to visualise the model. Many systems are simulated in a grid world so base implementations of a grid world model and grid world view were created. The class diagrams of these base classes are shown in Figure 4.3.

If a programmer wishes to implement a grid world environment then they can create a model class extending GridWorldModel and a view class extending GridWorldView. The implementation of the grid world will not be discussed here as the method names are enough to describe their function.

4.2 Multi-Agent System Projects

This section will detail how all of the previous work is combined to enable a user to define and simulate a system with multiple Uncertain AgentSpeak(L) agents. A MAS project requires an environment and multiple instances of agents programmed in Uncertain AgentSpeak(L).

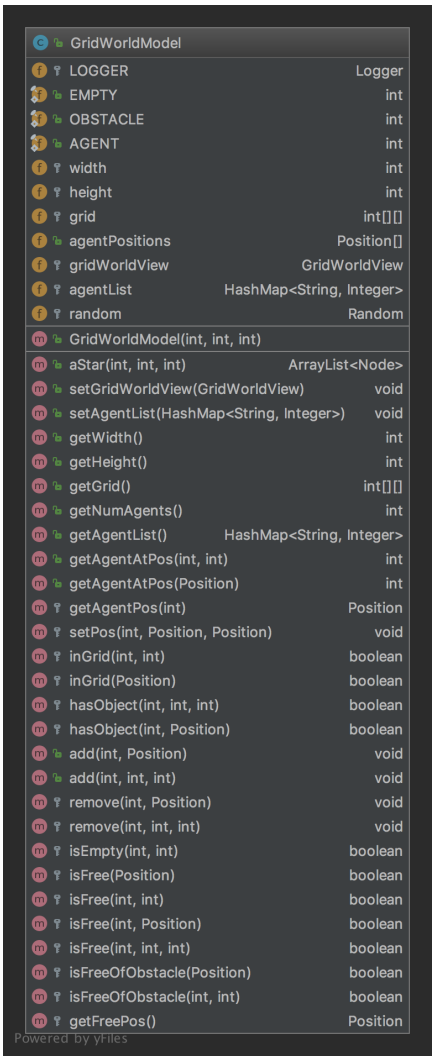
4.2.1 Syntax

A very simple text file can be used to define a MAS, the grammar for writing these files is shown in Figure 4.4.

A MAS project requires the definition of four components:

1. **name** - A name for the MAS project.
2. **infrastructure** - An underlying infrastructure for the the MAS. Only a centralised infrastructure was implemented but in the future it could be extended to include others. For example, Simple Agent Communication Infrastructure (SACI) (J. Hübner and Sichman 2009) and Java Agent DEvelopment Framework (JADE) (TILAB 2009).
3. **environment** - The name of the environment class must be specified along with any arguments used in it's constructor.
4. **agents** - The file names of each agent (written in Uncertain AgentSpeak(L)) and the number to instantiate.

a



b

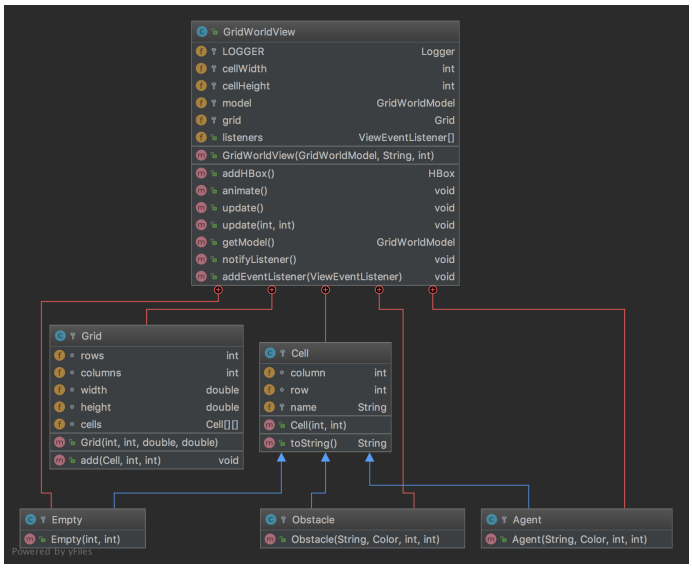


Figure 4.3: Base class diagrams for implementing a grid world environment, figure (a) shows the base grid world model class diagram and figure (b) shows the base grid world view class digram

The MASPorject class was created to manage a project. When the .mas configuration file is parsed it instantiates a MASProject object. It's class digram can be seen in Figure 4.5.

This class contains a list of the agents instantiated by the parser as well as the environment object that was created by the programmer. It also contains an executor service that is used execute each agents reasoning cycle on a separate thread and an agent console that is used to display the mental "thoughts" of each agent.

4.2.2 Infrastructure

At the start of this chapter the overall MAS development environment architecture was introduced. Figure 4.1 shows the centralised architecture that was implemented. This enabled

```
mas_project     ::= 'MAS' name '{' content '}' EOF
name            ::= <ATOM>
content         ::= infrastructure environment agents
infrastructure  ::= 'infrastructure:' centralised
centralised     ::= 'centralised'
environment     ::= 'environment:' 'env.' env_class_name '(' arguments_list ')'
env_class_name  ::= CLASS_NAME
agents          ::= 'agents:' agent+
agent           ::= agent_name NEWLINE | agent_name '#' numAgents NEWLINE
agent_name      ::= <ATOM>
numAgents       ::= <INTEGER>
arguments_list  ::= arg | arg ( ',' arg )+
arg             ::= <NUMBER> | <STRING> | 'true' | 'false'
```

Figure 4.4: BNF Grammar for defining MASs

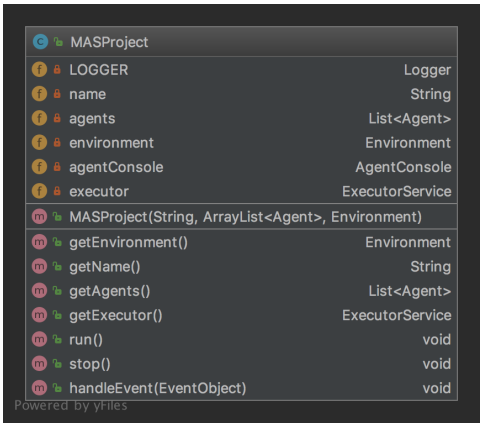


Figure 4.5: MASProject class diagram

all of the agents and the environment to be run on a single computer. However, as discussed previously, this infrastructure is limited in the number of agents that can be run and depends on the capabilities of the JVM and the operating system. As future work the ability to add a configurable pool of threads that are shared by the agents could be used to remove this limitation. The agents would share a reduced number of execution threads, thus reducing the computational burden but also reducing concurrency.

4.3 Mars Exploration Scenario

In order to measure the performance of both the extended AgentSpeak(L) language and the implemented framework an example Mars exploration scenario was constructed.

4.3.1 Background

The aim of this system is to maximise the scientific discovery from a team of exploration agents on Mars. In particular, discovering evidence for the presence of:

- Water/ice
- Fossils
- Living organisms

Different types of locations (characterised by land features) have been identified as more likely to contain evidence of each item. The objectives of the system are as follows:

1. Explore local terrain, locating areas of interest and improving certainty in the presence of items in existing areas of interest.
2. Collect samples from areas of interest.
3. Analyse collected samples for evidence of:
   (A) Water/ice
   (B) Fossils
   (C) Living organisms

The only prior knowledge going into the mission is that of nearby land features (determined from satellite imagery). Using this information the agents are initialised with beliefs regarding the likelihood of finding water/ice, fossils and living organisms at different locations. Predicting the presence of water/ice, fossils and living organisms is difficult due to insufficient information. For this reason, the beliefs associated with their presence at different locations are instantiated with possibility theory. In light of new information, agents are able to update their beliefs. If new information invalidates previous beliefs then this will be captured by the beliefs possibilistic instantiation.

The system consists of 3 types of agents: 1) type A sampling agents, 2) type B sampling agents and 3) sample analysis agents. The overall system consists of 3 agents: 1 type A sampling agent, 1 type B sampling agent and 1 sample analysis agent.

Sampling agents navigate the environment and collect samples. There are two types of sampling agents, each capable of collecting samples of different types:

- **Type A sampling agents** can collect samples determining the presence of water,
- **Type B sampling agents** can collect samples determining the presence of fossils and living organisms.

Sample analysis agents receive samples from sampling agents. They analyse these samples and determine the presence of water/ice, fossils or living organisms. Analysis agents update beliefs associated with the presence of water/ice, fossils or living organisms to be certain. In future work this would be used to coordinate excavation agents that are capable of performing further analysis on high priority locations.

4.3.2 Environment

The Mars environment was created by extending the environment class and implementing a model to manage the data and a view to visualise it. The environment is a 2D grid world that was created by extending the GridWorldModel and GridWorldView classes. The class diagrams for the implemented mars environment are shown in Figure 4.6.

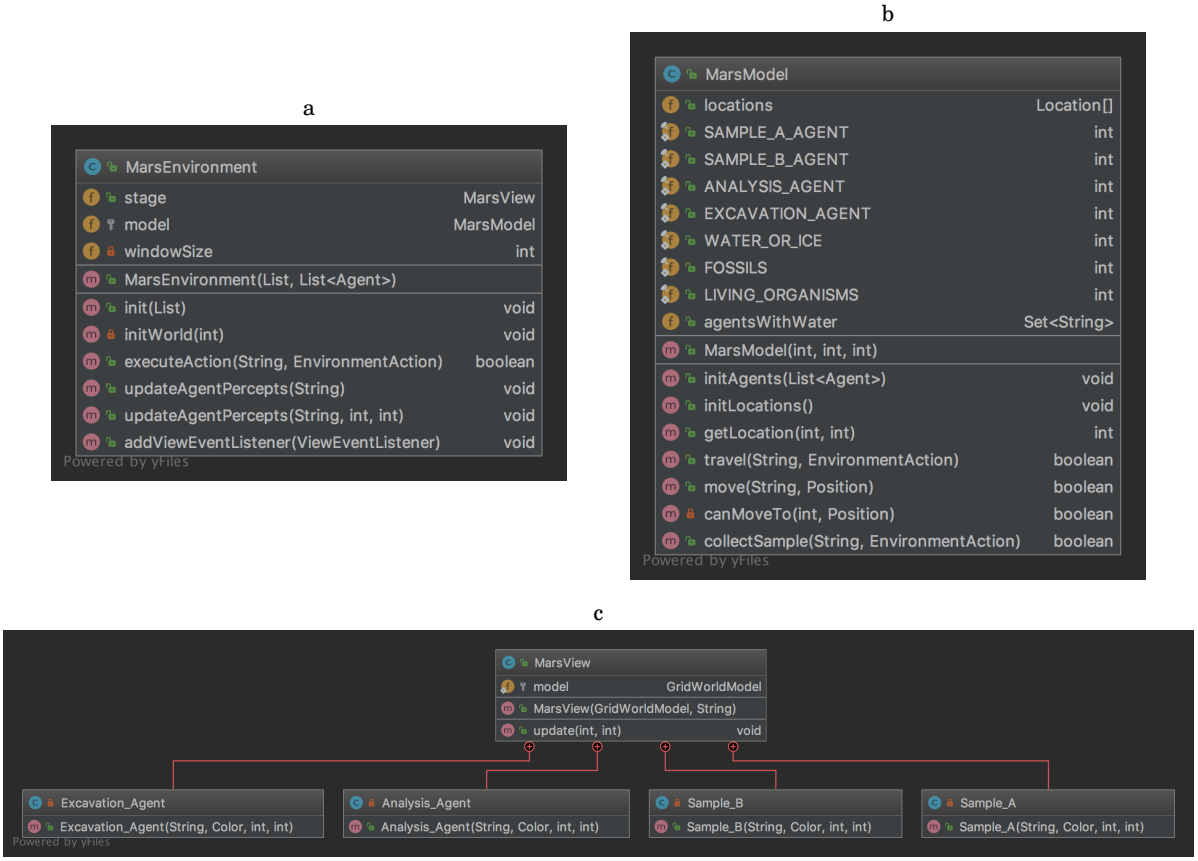


Figure 4.6: Class diagrams for (a) the mars environment class (b) the mars model class and (c) the mars view class

The model extends the grid world model class which used a class variable called grid, which is a 2D array of integers to model the world. The integer value at a particular position in the grid defines what object is present in the grid. The mars model defines the following objects that can be present in a grid cell:

- Sample A Agent
- Sample B Agent
- Analysis Agent
- Water or Ice

- Fossil
- Living organisms

The mars model also implements the agent's actions on the model using the following methods:

- **travel()** - This method utilises an A* search algorithm to provide a collision free path from the agent's current position to the target location. It then calls the move() method to iteratively move the agent along the path.
- **collectSample()** - This method checks that the agent is not already carrying a sample and that there is an object of the correct type in a neighboring cell. It then removes the object from the grid and adds the agent's name to the relevant carrying item list (e.g. agentsWithWater).
- **depositSample()** - This method checks that the agent is carrying a sample and that there is an empty cell that the agent can deposit it into. It then updates the grid accordingly.

The grid world view has two nested classes representing the grid and the cells on the grid. It also extends the cell class to define agent cells, obstacle cells and empty cells. The mars view extends the agent class to create new cell classes for each type of agent. It also defines new cell classes for each type of item (water/ice, fossil and living organisms). These define how the cells are visualised and an animation timer is used to update the view based on the model.

Figure 4.7 shows the 15x15 grid world used to model the environment. The red boxes indicate the nine locations that agents hold beliefs regarding the presence of each item. Each location is a 5x5 grid. Agents are capable of perceiving items that are contained in the same location that they are located in.

The mars environment class overrides the environments executeAction() method and provides methods for creating the agent's percepts and notifying the agents.

- **executeAction()** - This methods parses the string corresponding to the environment action and calls the models relevant action. If the action is successful then it calls the updateAgentPercepts for that particular agent.
- **updateAgentPercepts()** - This method updates the agentPercepts variable defined in the base environment class. The environment class automatically handles notifying the agents when they have new perceptions so this method purely updates the agentPercepts variable based on the mars model.
  - It checks the location of the agent and searches through the 5x5 section of the grid corresponding to that location for the presence of each item. If there is no item present then it increases the weight associated with the literal representing not having an item by a random amount. For example, if the agent was at $location(1)$

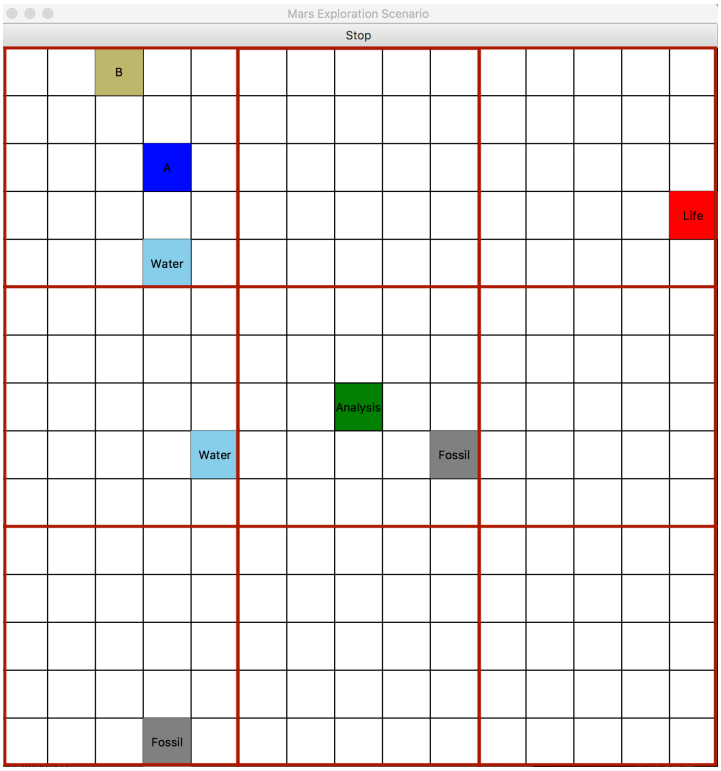


Figure 4.7: Mars world view showing the nine locations marked with red boxes. The dark blue box marked A represents a sample A agent, the brown cell marked B represents a sample B agent, the green cell marked analysis represents an analysis agent, the light blue boxes marked water represent water samples, the grey boxes marked fossil represents fossil samples and the red box marked life represents a living organism sample.

and there is no water or ice at $location(1)$ then agent would update it's GUB with $*(\neg water\_or\_ice(location(1)), w)$, where w is a random amount between the previous weight associated with the literal and the maximum weight, 1. If there is an item present then it increases the weight associated with the literal represnting that there is an item present (e.g. $water\_or\_ice(location(1))$)

- If the agent's name is on a carrying items list then the perception $*(carrying(A), 1)$ is added to the agentPercepts variable. This represents that the agent is certain that it is carrying what it believes to be a sample containing the presence of A.

- If the agent moves to a new location then the weight (probability) associated with it's belief in the old location is reduced and the weight associated with the new location is increased.

4.3.3 Agents

In order to solve this problem the agent's were implemented using the extended AgentSpeak(L) language. The agent's belief bases are represented as GUBs with five epistemic states, W_1, W_2 and W_3, representing the agent's beliefs in finding water/ice, fossils or living organisms at a given location respectively, W_4 representing the agent's belief in it's current location and W_5 representing the agent's belief in whether or not it is carrying a particular item. W_1, W_2 and W_3 are instantiated as possibilistic compact epistemic states and W_4 and W_5 are instantiated as probabilistic compact epistemic states as shown below,

```
1 ***(water_or_ice(location(X)), w).
2 ***(fossil(location(X)), w).
3 ***(living_organism(location(X)), w).
4 **(at(location(X)), p).
5 **(carrying(X), p).
```

where w is the weight associated with the belief literal l, such that $N(l) \geq w$ and p is the weight associated with the belief literal l, such that $P(l) = p$. For any epistemic state instantiation W over domain At, the initial weights for $b \in At$ are defined as follows,

$$W(b) = \begin{cases} (1,1), & \text{if } W \text{ is possibilistic,} \\ (0.5,0.5), & \text{if } W \text{ is probabilistic,} \end{cases}$$

where $W(b) = (\overset{+}{\mu}, \overset{-}{\mu})$ represents that $\overset{+}{\mu}$ is the λ-value for b and $\overset{-}{\mu}$ is the λ-value of it's negation $\neg b$. The definition of the sample agent A is detailed below along with details of it's interpretation cycle. The other agents operate in a similar manner.

4.3.3.1 Sample Agent A

The sample agent A program is listed in Figure 4.8 and lines 1-38 show the agent's initial beliefs being specified. Line 41 shows the definition of the agents initial goal $!exploreMars$. This is used by the agent to trigger relevant plans. The agent's plans are specified on lines 43-58.

The agents GUB and revised weights are shown in Table 4.1, where W_i represents the compact epistemic state i, $b \in At_i$ represents the domain of epistemic state i and $W_i(b)$ the weights associated with the positive and negative literals $W_i(b) = (\overset{+}{\mu}, \overset{-}{\mu})$. The weights associated with the belief literals in the epistemic state representing the agent's belief in it's current location are all set to zero. This is because when the agent is randomly initialised it perceives it's environment and updates it's GUB accordingly.

Tables 4.2 and 4.3 provide a summary of the agent's reasoning cycle for the first six steps of operation. The events generated due to the agents location have been omitted for clarity. Informally, the agent receives the goal $!exploreMars$ as a new event when it is instantiated, which in turn triggers the agent's first plan p_1. In this scenario there are nine locations that

```
# Initial beliefs.
***(water_or_ice(location(1)),0.9).
***(water_or_ice(location(2)),0.85).
***(~water_or_ice(location(3)),0.5).
***(water_or_ice(location(4)),0.35).
***(water_or_ice(location(5)),0.5).
***(water_or_ice(location(6)),0.4).
***(water_or_ice(location(7)),0.3).
***(~water_or_ice(location(8)),0.2).
***(~water_or_ice(location(9)),0.1).
***(~fossil(location(1)),0.9).
***(~fossil(location(2)),0.85).
***(~fossil(location(3)),0.5).
***(fossil(location(4)),0.35).
***(fossil(location(5)),0.8).
***(fossil(location(6)),0.4).
***(fossil(location(7)),0.9).
***(~fossil(location(8)),0.2).
***(~fossil(location(9)),0.7).
***(~living_organism(location(1)),0.9).
***(~living_organism(location(2)),0.85).
***(living_organism(location(3)),0.6).
***(living_organism(location(4)),0.35).
***(living_organism(location(5)),0.3).
***(living_organism(location(6)),0.4).
***(~living_organism(location(7)),0.9).
***(~living_organism(location(8)),0.4).
***(~living_organism(location(9)),0.6).
**(at(location(1)),0).
**(at(location(2)),0).
**(at(location(3)),0).
**(at(location(4)),0).
**(at(location(5)),0).
**(at(location(6)),0).
**(at(location(7)),0).
**(at(location(8)),0).
**(at(location(9)),0).
**(carrying(water),0).

# Initial goals.
!exploreMars.

# Plan library.
+!exploreMars : water_or_ice(location(X)) > water_or_ice(location(Y)) &&
                water_or_ice(location(X)) > water_or_ice(location(Y1)) &&
                X \== Y && X \== Y1 && Y \== Y1 <- !findWater(location(X)).

+!findWater(location(X)) : water_or_ice(location(Z)) > water_or_ice(location(X)) &&
                           Z \== X <- !findWater(location(Z)).

+!findWater(location(X)) : water_or_ice(location(Z)) && Z \== X <- travel(location(X)).

*(water_or_ice(location(X)), W) : water_or_ice(location(X)) > water_or_ice(location(X1)) &&
                                  at(location(X))  <- collectSample(water).

*(water_or_ice(location(X)), W) : ~water_or_ice(location(X) <- !exploreMars.

*(carrying(A), W) : carrying(A) <- travel(location(8,7)); depositSample(A); !exploreMars.
```

Figure 4.8: Sample Agent A program listing for Mars exploration scenario

the agent holds beliefs about the presence of water/ice. Figure 4.9 shows the plan that an agent would require in order to determine the most plausible location for finding water/ice.

Table 4.1: Initial GUB definition and revised weights

W_i	Instantiation	$b \in At_i$	$W_i(b)$
		$water\_or\_ice(location(1))$	(1, 0.9)
		$water\_or\_ice(location(2))$	(1, 0.85)
		$water\_or\_ice(location(3))$	(0.5, 1)
		$water\_or\_ice(location(4))$	(1, 0.35)
W_1	Possibility	$water\_or\_ice(location(5))$	(1, 0.5)
		$water\_or\_ice(location(6))$	(1, 0.4)
		$water\_or\_ice(location(7))$	(1, 0.3)
		$water\_or\_ice(location(8))$	(0.8, 1)
		$water\_or\_ice(location(9))$	(0.9, 1)
		$fossil(location(1))$	(0.1, 1)
		$fossil(location(2))$	(0.15, 1)
		$fossil(location(3))$	(0.5, 1)
		$fossil(location(4))$	(1, 0.35)
W_2	Possibility	$fossil(location(5))$	(1, 0.8)
		$fossil(location(6))$	(1, 0.4)
		$fossil(location(7))$	(1, 0.9)
		$fossil(location(8))$	(0.8, 1)
		$fossil(location(9))$	(0.3, 1)
		$living\_organism(location(1))$	(0.1, 1)
		$living\_organism(location(2))$	(0.15, 1)
		$living\_organism(location(3))$	(1, 0.6)
		$living\_organism(location(4))$	(1, 0.35)
W_3	Possibility	$living\_organism(location(5))$	(1, 0.3)
		$living\_organism(location(6))$	(1, 0.4)
		$living\_organism(location(7))$	(0.1, 1)
		$living\_organism(location(8))$	(0.6, 1)
		$living\_organism(location(9))$	(0.4, 1)
		$at(location(1))$	(0, 1)
		$at(location(2))$	(0, 1)
		$at(location(3))$	(0, 1)
		$at(location(4))$	(0, 1)
W_4	Probability	$at(location(5))$	(0, 1)
		$at(location(6))$	(0, 1)
		$at(location(7))$	(0, 1)
		$at(location(8))$	(0, 1)
		$at(location(9))$	(0, 1)
W_5	Probability	$carrying(water)$	(0, 1)

Table 4.2: Sample Agent A's reasoning cycle - planning stage

Step	Event Set	Relevant Plans	Applicable Plans	Adopt Intention
1	$\{e_1\}$	$\{\langle p_1, \emptyset\rangle\}$	$\{\langle p_1, \{X/7, Y/9, Y1/8\}\rangle\}$	$i_1 = [\langle p_1, \{X/7, Y/9, Y1/8\}\rangle]$
2	$\{e_2\}$	$\{\langle p_2, \{X/7\}\rangle, \langle p_3, \{X/7\}\rangle\}$	$\{\langle p_2, \{X/7, Z/2\}\rangle, \langle p_3, \{X/7, Z/9\}\rangle\}$	$i_2 = [\langle p_2, \{X/7, Z/2\}\rangle, \langle p_1, \{X/7, Y/9, Y1/8\}\rangle]$
	$\{e_3\}$	$\{\langle p_2, \{X/2\}\rangle, \langle p_3, \{X/2\}\rangle\}$	$\{\langle p_2, \{X/2, Z/1\}\rangle, \langle p_3, \{X/2, Z/4\}\rangle\}$	$i_3 = [\langle p_2, \{X/2, Z/1\}\rangle, \langle p_2, \{X/7, Z/2\}\rangle, \langle p_1, \{X/7, Y/9, Y1/8\}\rangle]$
3	$\{e_4\}$	$\{\langle p_2, \{X/1\}\rangle, \langle p_3, \{X/1\}\rangle\}$	$\{\langle p_3, \{X/1, Z/4\}\rangle\}$	$i_4 = [\langle p_3, \{X/1, Z/4\}\rangle, \langle p_2, \{X/2, Z/1\}\rangle, \langle p_2, \{X/7, Z/2\}\rangle, \langle p_1, \{X/7, Y/9, Y1/8\}\rangle]$
4	$\emptyset$	-	-	-
5	$\{e_5\}$	$\{\langle p_4, \{X/1, W/0.9\}\rangle\}$	$\{\langle p_4, \{X/1, W/0.9\}\rangle\}$	$i_5 = [\langle p_4, \{X/1, W/0.9\}\rangle]$
6	$\{e_6\}$	$\{\langle p_5, \{\}\rangle\}$	$\{\langle p_5, \{\}\rangle\}$	$i_6 = \{\langle p_5, \{\}\rangle\}$

Table 4.3: Sample Agent A's reasoning cycle - acting stage

Step	Intention Set	Execute Step	Generate Event	Intention Set
1	$\{j_1\}$	$!findWater(location(7))$	$e_2 = \langle +!findWater(location(7)), i_1\rangle$	$\emptyset$
2	$\{j_2\}$	$!findWater(location(2))$	$e_3 = \langle +!findWater(location(2)), i_2\rangle$	$\emptyset$
3	$\{j_3\}$	$!findWater(location(1))$	$e_4 = \langle +!findWater(location(1)), i_3\rangle$	$\emptyset$
4	$\{j_4\}$	$travel(location(1))$	$e_5 = \langle *(water\_or\_ice(location(1)), 0.98), []\rangle$	$\{i_4\}$
5	$\{j_5\}$	$collectSample(water)$	$e_6 = \langle *(carrying(water), 1), []\rangle$	$\{i_5\}$
6	$\{j_6\}$	$travel(location(5))$	-	$\{i_6\}$

```
+!exploreMars :
    water_or_ice(location(X)) >= water_or_ice(location(Y)) && water_or_ice(location(X)) >= water_or_ice(location(
     Y1)) &&
    water_or_ice(location(X)) >= water_or_ice(location(Y2)) && water_or_ice(location(X)) >= water_or_ice(location(
     Y3)) &&
       water_or_ice(location(X)) >= water_or_ice(location(Y4)) && water_or_ice(location(X)) >= water_or_ice(
     location(Y5)) &&
       water_or_ice(location(X)) >= water_or_ice(location(Y6)) && water_or_ice(location(X)) >= water_or_ice(
     location(Y7)) &&
       water_or_ice(location(X)) >= water_or_ice(location(Y8)) &&
       X \== Y && X \== Y1 && X \== Y2 && X \== Y3 && X \== Y4 && X \== Y5 && X \== Y6 && X \== Y7 && X \== Y8 &&
       Y \== Y1 && Y \== Y2 && Y \== Y3 && Y \== Y4 && Y \== Y5 && Y \== Y6 && Y \== Y7 && Y \== Y8 &&
       Y1 \== Y2 && Y1 \== Y3 && Y1 \== Y4 && Y1 \== Y5 && Y1 \== Y6 && Y1 \== Y7 && Y1 \== Y8
       Y2 \== Y3 && Y2 \== Y4 && Y2 \== Y5 && Y2 \== Y6 && Y2 \== Y7 && Y2 \== Y8 &&
       Y3 \== Y4 && Y3 \== Y5 && Y3 \== Y6 && Y3 \== Y7 && Y3 \== Y8 &&
       Y4 \== Y5 && Y4 \== Y6 && Y4 \== Y7 && Y4 \== Y8 &&
       Y5 \== Y6 && Y5 \== Y7 && Y5 \== Y8 &&
       Y6 \== Y7 && Y6 \== Y8 &&
       Y7 \== Y8  <- travel(location(X)).
```

Figure 4.9: Non-recursive plan for determining most plausible location containing water or ice

This plan contains nine free variables and the relevant epistemic state contains nine belief atoms that each free variable could unify with. The getUnfiers() method in the GUB class suffers from complexity issues because it is required to return every possible unifier for the plan context. As a result the getUnifiers() method would return 387,420,489 possible unifiers. Future work should look to improve the efficiency of this method. However, this issue was dealt with by defining recursive plans for the agent. Plan p_2 and plan p_3 both contain the same triggering event so are both relevant plans when the achievement goal $!findWater(location(X))$ is added (and selected) from the event set. Plan p_2 recursively calls itself until it's context can no longer evaluate to true i.e. the agent could not find any locations that it believes contains water/ice more than the current instantiation of X (acting as an accumulator). Plan p_3 then becomes the only applicable plan and the environment action $travel(location(1))$ is executed on the environment.

At this point the agent waits for a new event to act upon. As the agent has now moved to a new

location it receives new perceptions from the environment regarding the presence of water/ice in location 1 and it's new location. The agent's new perceptions create three revise belief actions $*(water\_or\_ice(location(1)), 0.98)$, $*(at(location(1)), 0.98)$ and $*(at(location(5)), 0.25)$. These actions update the agent's GUB and create new external events that are added to the event set that the agent is managing. The event associated with $*(water\_or\_ice(location(1)), 0.98)$ is then selected from the event set and unifies with plans p_4 and p_5, making them relevant plans. As the λ-value associated with $water\_or\_ice(location(1))$ is higher than the λ-value for $\neg water\_or\_ice(location(1))$ and similarly for it's belief about it's location, only plan p_4 is applicable and the agent performs the environment action $collectSample(water)$. The $collectSample(water)$ action informs the agent's controller, which automatically moves the agent to the cell containing water/ice and collects it. If the plan context does not evaluate to true then plan p_5 will be an applicable plan. This plan will create a new achievement goal $!exploreMars$, which will restart the reasoning cycle by triggering plan p_1. The events associated with the agent's current location will then be selected and removed from the event set. These do not result in any relevant plans and as there are no un-executed actions in the intention set the agent once again waits for a new event.

At this point the agent has picked up the item and receives a new perception from the environment which creates a new revise belief action $*(carrying(water), 1)$. The resulting external event unifies with plan p_6, making it a relevant plan. As the agent believes that it is carrying water the plan is applicable and it executes the $travel(location(8,7))$ environment action, which moves it to the analysis agent's location. The agent receives new perceptions regarding it's location but once again the corresponding events do not trigger any plans. The agent then executes the $depositSample(water)$ environment action which deposits the sample next to the analysis agent. The agent once again receives new perceptions that it is no longer carrying the sample and revises it's GUB accordingly. The corresponding event triggers plan p_6, however, as the agent does not believe that it is still carrying the sample the plan is not applicable. The agent then generates a new achievement goal $!exploreMars$ and the process continues. As the environment has been modified along with the agent's GUB, the agent continues to collect samples, checking more plausible locations first and stopping once it believes that all of the locations no longer contain samples.

4.3.4 MAS Definition and Runtime

Figure 4.10 shows the (.mas) configuration file that was used to define the mars exploration MAS. The system uses a centralised infrastructure and specifies an environment class called MarsEnvironment that has two constructor parameters: the environment configuration ID that specifies the model to be used from the MarsWorldFactory class and the size of the window containing the environment's view. The MarsWorldFactory class was used to create mars worlds with different layouts and number of items.

```
MAS marsExplorationScenario {

    infrastructure: centralised

    environment: env.MarsEnvironment(2, 800)
//                                  1. environment configuration id (int)
//                                  2. Window width (int)

    agents:
        sampleAgentA ;

        sampleAgentB ;

        analysisAgent ;
}
```

Figure 4.10: Mars exploration MAS configuration file (.mas)

Figure 4.11 shows the IDE that was created for Uncertain AgentSpeak(L) with the mars exploration MAS loaded into it. Figure 4.12 shows a screen shot of the full mars exploration MAS running, including the IDE, agent console and mars environment view.

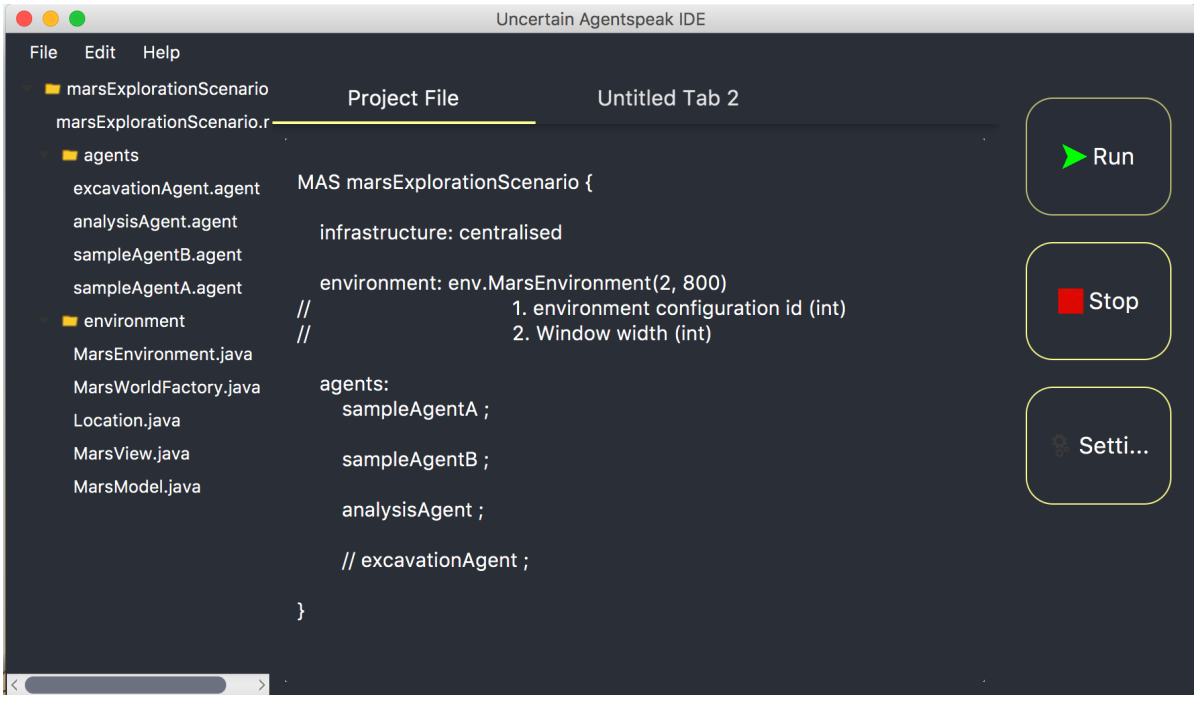


Figure 4.11: Screen shot of the mars exploration MAS loaded into the Uncertain AgentSpeak(L) IDE

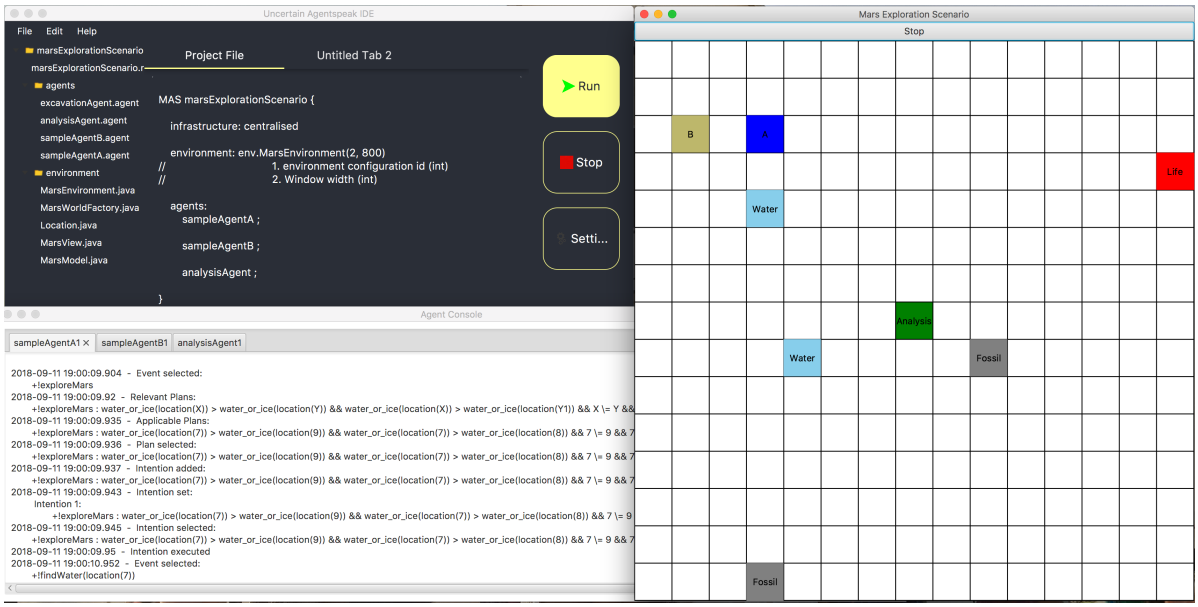


Figure 4.12: Screen shot of the IDE, agent console and environment view for the mars exploration MAS

CHAPTER 5

DISCUSSION

In the previous two chapters the implementation of the extended AgentSpeak(L) programming language was detailed, along with the development environment that was created for defining and simulating MASs written in the extended language. The mars exploration MAS presented in Section 4.3 provides a means of analysing the performance of both the extended language and the development environment.

This project had four main objectives: to implement the AgentSpeak(L) agent programming language, extend this implementation for modelling and reasoning with uncertain information, to develop a platform where MASs written in the extended language could be defined and simulated, and to analyse the performance of the extended language utilising the platform. The first objective was intended to provide a base for implementing the extended language as well as providing a bench mark to asses it's performance.

The epistemic state class that was implemented provides the base functionality for instantiating epistemic states with different underlying uncertainty theories. This is in coherence with Definition-2.4.1. The class provides the basic functionality of paring down a logical formula $\phi \in \mathscr{L}_{\geq}$ to be a propositional statement $\phi \in \mathscr{L}$. Bauters et al. (2017) presented a compact epistemic state (an extension of the general epistemic state discussed previously) providing a tractable approach to modelling and revising uncertain beliefs. The compact epistemic state class implements efficient mechanisms for revision and determining the λ-value of a formula. The probabilistic and possibilistic compact epistemic state instantiations achieve probability and possibility distributions over the belief literals contained within the epistemic states, as desired. Dempster-Shafer theory is a powerful uncertainty theory that is limited in it's applicability due to it's computational complexity. Future work could investigate how the epistemic state class could be instantiated with Dempster-Shafer theory and how efficient revision and entailment

operators could be defined and implemented.

A GUB was used to group an agents epistemic states and act as it's belief base. Although the underlying mechanisms for evaluating whether a logical expression is entailed by the GUB are efficient, the need to obtain all of the possible unifiers between the logical expression and the GUB results in complexity issues when evaluating formulas with many free variables. In the example scenario, the agents plan libraries consisted of recursive plans to obtain the most plausible location containing an item. This was due to the computational complexity of the getUnifiers() method that scales exponentially with the number of free variables in the plan context. Future work could define syntax for obtaining the most plausible belief literal from an epistemic state using recursion (outside of the agent's interpreter). This would prevent the agent's interpreter from performing execution steps to find the most plausible location and would instead perform the recursive operation "behind the scenes".

In the mars exploration MAS the agent's were capable of reasoning about uncertain beliefs regarding the presence of items in different locations. A MAS consisting of standard AgentSpeak(L) agents would have to model each location as either containing or not containing an item. This binary representation of beliefs does not permit agents to prioritise locations based on the strengths of their beliefs (instantiated with uncertainty theories). Instead, they would be required to deploy random search which would result in significantly lower performance. Although the GUB is a powerful tool for implementing beliefs instantiated with uncertainty theories, it is also quite restrictive. It would be beneficial for an agent to maintain both a GUB and a standard AgentSpeak(L) belief base. There are certain beliefs that are better modelled as standard belief atoms. Representing an agents location in the example scenario consisted of defining a belief literal for each location ($at(location(1))$, ..., $at(location(9))$). This required the agent having prior knowledge of it's environment and also leads to complexity issues when determining the most plausible belief when the epistemic state's domain is large. In situations like this it would be more beneficial to simply add and delete beliefs, as in AgentSpeak(L).

Although the mars exploration MAS was simple it proves that agents are capable of modelling their beliefs as epistemic states instantiated with probability and possibility theory. The mechanisms for belief revision and entailment were tested in the scenario and were capable of providing reactive behaviour. The language for constructing plan contexts and test goals is much more expressive than in AgentSpeak(L), where they simply consist of a conjunction of literals.

This work only implemented a centralised infrastructure for running MASs. This executes all of the agents and the environment on the same machine using separate threads. This has limited performance arising from the capabilities of the JVM and the underlying operating system. Implementing a distributed architecture such as SACI or JADE would enable large MASs to be defined and simulated. Currently the centralised architecture is limited to running no more than 5 agents (on a 2.3GHz Intel Core i7 MacBook Pro). It would be interesting to see how well both the extended AgentSpeak(L) implementation and the simulation environment scale with large

systems.

The environment is capable of hosting multiple agents that can both perceive from and act on the environment. This is made possible by synchronized methods that ensure no concurrency issues arise. There is a field of researching looking into non-blocking algorithms for concurrent environments that utilise low-level machine instructions to ensure data integrity. Java has implementations of atomic variables which work along these lines preventing locking issues arising from synchronised methods. Future work should consider integrating atomic variables into the simulation environment.

CHAPTER 6

CONCLUSION

The overall aim of this project was to implement an extension of the AgentSpeak(L) agent programming language for modelling and reasoning with uncertain information. The implemented extended AgentSpeak(L) language effectively allows agents to model and reason with uncertainty in a computationally efficient manner. Agents are capable of modelling their beliefs as either probabilistic or possibilistic epistemic states and the implementation of epistemic states provides a base for easily implementing instantiations of different uncertainty theories, e.g. Demspter-Shafer theory. The language for constructing plan contexts and test goals was extended and implemented as the logical expression class. This included the ability to use qualitative operators such as $>$, not and $\geq$. Using a GUB to group together an agent's uncertain beliefs (represented as epistemic states) and the logical expression class, agents are capable of reasoning over uncertain beliefs in the context of plans and in the bodies of plans using test goals. This enables agents to select applicable plans from the set of relevant plans by querying if a logical formula is entailed by the GUB.

The development environment enables a user to easily create a simulation environment that is capable of hosting multiple agents that are modelling and reasoning with uncertain information. It uses a multi-threaded approach to enable multiple agents to run on a single machine and ensures that agents act on and perceive an environment without any thread interference or memory inconsistency issues. Base classes were created for implementing a grid world environment, including a grid world model and a grid world view for visualising the environment. All of the work in this project was then brought together in a mars exploration MAS that demonstrated the extended modelling and reasoning capabilities as well as the power of the development environment as a whole.

Humans are a great example of autonomous agents as we successfully achieve our desires in a

world pervaded with uncertainty. The field of artificial intelligence faces a significant challenge to develop intelligent agents that are capable of reasoning as human beings do. The BDI architecture and as a result AgentSpeak(L), has gained great interest for achieving such intelligent agents. The work presented here has extended the capabilities of AgentSpeak(L) agents to model uncertain information using epistemic states and to reason about these uncertain beliefs in a tractable manner. Further to this, it has provided a development environment for defining and simulating multiple extended AgentSpeak(L) agents on a single machine.

APPENDIX A

BELIEF BASE JAVA CLASSES

This appendix details the Java classes that were used to implement the belief base, including the global uncertain belief class, epistemic state class, compact epistemic state class, probabilistic compact epistemic state and possibilistic compact epistemic state class.

A.1 GlobalUncertainBelief Class

```
1  package main.java.uncertain_agentspeak.uncertainty;
2
3  public class GlobalUncertainBelief {
4
5      private final Logger LOGGER = LogManager.getLogger("Global Uncertain Belief");
6
7      private HashSet<BeliefAtom> domain;
8      private HashMap<HashSet<BeliefAtom>, CompactEpistemicState> epistemicStates;
9
10     public GlobalUncertainBelief() {
11         domain = new HashSet<>();
12         epistemicStates = new HashMap<>();
13     }
14
15     public void addEpistemicState(CompactEpistemicState epistemicState) throws
       Exception {
16         if(!this.domain.isEmpty() && !epistemicState.getDomain().isEmpty()) {
17             for (BeliefAtom t : epistemicState.getDomain()) {
18                 if (domain.contains(t)) {
19                     throw new Exception("Intersects with existing epistemic state");
20                 }
21             }
22         }
23         this.domain.addAll(epistemicState.getDomain());
24         epistemicStates.put(epistemicState.getDomain(), epistemicState);
```

```
25      }
26
27      public HashMap<HashSet<BeliefAtom>, CompactEpistemicState> getGUB() {
28          return epistemicStates;
29      }
30
31      private HashSet<CompactEpistemicState> getRelevantEpistemicStates(LogicalExpression
         logicalExpression) {
32          HashSet<CompactEpistemicState> relevantEpistemicStates = new HashSet<>();
33          for (Map.Entry<HashSet<BeliefAtom>, CompactEpistemicState> epistemicState :
    epistemicStates.entrySet()) {
34              CompactEpistemicState compactEpistemicState = epistemicState.getValue();
35              if (compactEpistemicState.languageContains(logicalExpression)) {
36                  relevantEpistemicStates.add(compactEpistemicState);
37              }
38          }
39          return relevantEpistemicStates;
40      }
41
42      public void revise(BeliefLiteral beliefLiteral, double weight) throws Exception {
43          if (!beliefLiteral.isGround()) {
44              throw new NotGroundException(beliefLiteral + "is not ground");
45          }
46          for (Map.Entry<HashSet<BeliefAtom>, CompactEpistemicState> epistemicStateEntry
    : epistemicStates.entrySet()) {
47              BeliefAtom beliefAtom = beliefLiteral.getBeliefAtom();
48              HashSet<BeliefAtom> domain = epistemicStateEntry.getKey();
49              if ( domain.contains(beliefAtom) ) {
50                  CompactEpistemicState epistemicState = epistemicStateEntry.getValue();
51                  epistemicState.revise(beliefLiteral, weight);
52                  epistemicStates.put(domain, epistemicState);
53                  return;
54              }
55          }
56          LOGGER.error("No local epistemic state for: " + beliefLiteral);
57          throw new Exception("No local epistemic state for: " + beliefLiteral);
58      }
59
60      private boolean languageContains(LogicalExpression logicalExpression) {
61          for (Map.Entry<HashSet<BeliefAtom>, CompactEpistemicState> epistemicState :
    epistemicStates.entrySet()) {
62              CompactEpistemicState compactEpistemicState = epistemicState.getValue();
63              if (compactEpistemicState.languageContains(logicalExpression)) {
64                  return true;
65              }
66          }
67          return false;
68      }
69
70      private HashSet<Unifier> getUnifiers(LogicalExpression logicalExpression, Unifier
    unifier) throws Exception {
71
72          HashSet<BeliefAtom> beliefAtoms = logicalExpression.substitute(unifier).
    getBeliefAtoms();
73          ArrayList<Variable> freeVariables = new ArrayList<>();
```

```
74
75          // Get the free variables in the logical expression
76          for ( BeliefAtom beliefAtom : beliefAtoms) {
77              if (!beliefAtom.isGround()) {
78                  freeVariables.addAll(beliefAtom.getTerm().getVariables());
79              }
80          }
81
82          // Get the set of Term's each free variable could be instantiated with
83          HashMap<Variable, HashSet<Term>> varTermMap = new HashMap<>();
84          for (Variable variable : freeVariables) {
85              for (CompactEpistemicState compactEpistemicState :
    getRelevantEpistemicStates(logicalExpression.substitute(unifier))) {
86                  for (BeliefAtom beliefAtom : beliefAtoms) {
87                      if (beliefAtom.getTerm().getVariables() != null) {
88                          if (beliefAtom.getTerm().getVariables().contains(variable)) {
89                              varTermMap.put(variable, compactEpistemicState.getUnifiers(
    beliefAtom, variable));
90                          }
91                      }
92                  }
93              }
94          }
95
96          // recursive method to create a Unifier for every combination of variable
    substitution
97          return combine(0, unifier, varTermMap, new HashSet<>());
98      }
99
100     public HashSet<Unifier> combine(int index, Unifier current, Map<Variable,HashSet<
    Term>> map, HashSet<Unifier> list) {
101         if(index == map.size()) {
102             Unifier newUnifier = new Unifier();
103             for(Variable key: current.keySet()) {
104                 newUnifier.put(key, current.get(key));
105             }
106             list.add(newUnifier);
107         } else {
108             Object currentKey = map.keySet().toArray()[index];
109             for(Term value: map.get(currentKey)) {
110                 current.put((Variable)currentKey, value);
111                 combine(index + 1, current, map, list);
112                 current.remove(currentKey);
113             }
114         }
115         return list;
116     }
117
118     public Unifier entails(LogicalExpression logicalExpression) throws Exception {
119         return entails(logicalExpression, new Unifier());
120     }
121
122     public Unifier entails(LogicalExpression logicalExpression, Unifier unifier) throws
     Exception {
123         HashSet<Unifier> unifiers = this.getUnifiers(logicalExpression, unifier);
```

```
124
125         for (Unifier u : unifiers) {
126             Unifier unifierValid = null;
127             if (logicalExpression instanceof Conjunction) {
128                 unifierValid = entails((Conjunction) logicalExpression, u);
129             } else if (logicalExpression instanceof Disjunction) {
130                 unifierValid = entails((Disjunction) logicalExpression, u);
131             } else if (logicalExpression instanceof GreaterEqualsPlausibility) {
132                 unifierValid = entails((GreaterEqualsPlausibility) logicalExpression, u
    );
133             } else if (logicalExpression instanceof GreaterThanPlausibility) {
134                 unifierValid = entails((GreaterThanPlausibility) logicalExpression, u);
135             } else if (logicalExpression instanceof Negation) {
136                 unifierValid = entails((Negation) logicalExpression, u);
137             } else if (logicalExpression instanceof BeliefAtom) {
138                 unifierValid = entails((BeliefAtom) logicalExpression, u);
139             } else if (logicalExpression instanceof BeliefLiteral) {
140                 unifierValid = entails((BeliefLiteral) logicalExpression, u);
141             } else if (logicalExpression instanceof RelationalExpression) {
142                 unifierValid = entails((RelationalExpression) logicalExpression, u);
143             } else if (logicalExpression instanceof Contradiction) {
144                 unifierValid = entails((Contradiction) logicalExpression, u);
145             } else if (logicalExpression instanceof Tautology) {
146                 unifierValid = entails((Tautology) logicalExpression, u);
147             }
148             if (unifierValid != null) {
149                 return  unifierValid;
150             }
151         }
152         return null;
153     }
154
155     private Unifier entails(Contradiction contradiction, Unifier unifier) {
156         return null;
157     }
158
159     private Unifier entails(Tautology tautology, Unifier unifier) {
160         return unifier;
161     }
162
163     private Unifier entails(BeliefAtom beliefAtom, Unifier unifier) throws Exception {
164         BeliefAtom groundBeliefAtom = beliefAtom.substitute(unifier);
165         if (this.languageContains(groundBeliefAtom)) {
166             HashSet<CompactEpistemicState> relevantEpistemicStates =
    getRelevantEpistemicStates(groundBeliefAtom);
167             for (CompactEpistemicState compactEpistemicState : relevantEpistemicStates)
     {
168 //                System.out.println(compactEpistemicState.toString());
169                 Unifier unifierValid = compactEpistemicState.entails(beliefAtom,
    unifier);
170                 if (unifierValid != null) {
171                     return unifierValid;
172                 }
173             }
174         }
```

```
175         return null;
176     }
177 
178     private Unifier entails(BeliefLiteral beliefLiteral, Unifier unifier) throws
    Exception {
179         BeliefLiteral groundBeliefLiteral = beliefLiteral.substitute(unifier);
180         if (this.languageContains(groundBeliefLiteral)) {
181             HashSet<CompactEpistemicState> relevantEpistemicStates =
    getRelevantEpistemicStates(groundBeliefLiteral);
182             for (CompactEpistemicState compactEpistemicState : relevantEpistemicStates)
     {
183                 Unifier unifierValid = compactEpistemicState.entails(beliefLiteral,
    unifier);
184                 if (unifierValid != null) {
185                     return unifierValid;
186                 }
187             }
188         }
189         return null;
190     }
191 
192     private Unifier entails(Conjunction conjunction, Unifier unifier) throws Exception
    {
193         Conjunction groundConjunction = conjunction.substitute(unifier);
194         if (this.languageContains(groundConjunction)) {
195             HashSet<CompactEpistemicState> relevantEpistemicStates =
    getRelevantEpistemicStates(groundConjunction);
196             for (CompactEpistemicState compactEpistemicState : relevantEpistemicStates)
     {
197                 Unifier unifierValid = compactEpistemicState.entails(conjunction,
    unifier);
198                 if (unifierValid != null) {
199                     return unifierValid;
200                 }
201             }
202             return null;
203         } else {
204             Unifier unifierLeft = this.entails(conjunction.getLeft(), unifier);
205             Unifier unifierRight = null;
206             if (unifierLeft != null) {
207                 unifierRight = this.entails(conjunction.getRight(), unifierLeft);
208             }
209             return unifierRight;
210         }
211     }
212 
213     private Unifier entails(Disjunction disjunction, Unifier unifier) throws Exception
    {
214         Disjunction groundDisjunction = disjunction.substitute(unifier);
215         if (this.languageContains(groundDisjunction)) {
216             HashSet<CompactEpistemicState> relevantEpistemicStates =
    getRelevantEpistemicStates(groundDisjunction);
217             for (CompactEpistemicState compactEpistemicState : relevantEpistemicStates)
     {
```

```
218                 Unifier unifierValid = compactEpistemicState.entails(disjunction,
    unifier);
219                 if (unifierValid != null) {
220                     return unifierValid;
221                 }
222             }
223             return null;
224         } else {
225             if ( this.entails(disjunction.getLeft(), unifier) != null || this.entails(
    disjunction.getRight(), unifier) != null && groundDisjunction.isGround()) {
226                 return unifier;
227             }
228             return null;
229         }
230     }
231
232     private Unifier entails(GreaterEqualsPlausibility greaterEqualsPlausibility,
    Unifier unifier) throws Exception {
233         GreaterEqualsPlausibility groundGreaterEqualsPlausibility =
    greaterEqualsPlausibility.substitute(unifier);
234         if (this.languageContains(groundGreaterEqualsPlausibility)) {
235             HashSet<CompactEpistemicState> relevantEpistemicStates =
    getRelevantEpistemicStates(groundGreaterEqualsPlausibility);
236             for (CompactEpistemicState compactEpistemicState : relevantEpistemicStates)
     {
237                 Unifier unifierValid = compactEpistemicState.entails(
    greaterEqualsPlausibility, unifier);
238                 if (unifierValid != null) {
239                     return unifierValid;
240                 }
241             }
242             return null;
243         } else {
244             if (groundGreaterEqualsPlausibility.isGround() ) {
245                 if (languageContains(groundGreaterEqualsPlausibility.getLeft()) &&
    languageContains(groundGreaterEqualsPlausibility.getRight())) {
246                     double lambdaLeft = 0;
247                     double lambdaRight = 0;
248                     CompactEpistemicState classLeft = null;
249                     CompactEpistemicState classRight = null;
250
251                     StrongNegation negatedLeft = new StrongNegation(
    groundGreaterEqualsPlausibility.getLeft());
252                     StrongNegation negatedRight = new StrongNegation(
    groundGreaterEqualsPlausibility.getRight());
253
254                     for (CompactEpistemicState compactEpistemicState :
    getRelevantEpistemicStates(negatedLeft)) {
255                         classLeft = compactEpistemicState;
256                         lambdaLeft = compactEpistemicState.getLambda(negatedLeft);
257                     }
258
259                     for (CompactEpistemicState compactEpistemicState :
    getRelevantEpistemicStates(negatedRight)) {
260                         classRight = compactEpistemicState;
```

```
261                         lambdaRight = compactEpistemicState.getLambda(negatedRight);
262                     }
263                     if (classLeft instanceof CompactProbabilisticEpistemicState && !(
    classRight instanceof CompactProbabilisticEpistemicState)) {
264                         LOGGER.error("The operands of a qualitative operator must be of
     the same type");
265                     } else if (classLeft instanceof CompactPossibilisticEpistemicState
    && !(classRight instanceof CompactPossibilisticEpistemicState)) {
266                         LOGGER.error("The operands of a qualitative operator must be of
     the same type");
267                     }
268 
269                     if (lambdaLeft <= lambdaRight) {
270                         return unifier;
271                     }
272                     return null;
273 
274                 } else {
275                         LOGGER.error("The operands must be formulas in the language L_G
    ");
276                         return null;
277                 }
278             } else {
279                 LOGGER.error("Formula is not ground.");
280                 return null;
281             }
282         }
283     }
284 
285     private Unifier entails(GreaterThanPlausibility greaterThanPlausibility, Unifier
    unifier) throws Exception {
286         GreaterThanPlausibility groundGreaterThanPlausibility = greaterThanPlausibility
    .substitute(unifier);
287         if (this.languageContains(groundGreaterThanPlausibility)) {
288             HashSet<CompactEpistemicState> relevantEpistemicStates =
    getRelevantEpistemicStates(groundGreaterThanPlausibility);
289             for (CompactEpistemicState compactEpistemicState : relevantEpistemicStates)
     {
290                 Unifier unifierValid = compactEpistemicState.entails(
    greaterThanPlausibility, unifier);
291                 if (unifierValid != null) {
292                     return unifierValid;
293                 }
294             }
295             return null;
296         } else {
297             if (groundGreaterThanPlausibility.isGround() ) {
298                 if (languageContains(groundGreaterThanPlausibility.getLeft()) &&
    languageContains(groundGreaterThanPlausibility.getRight())) {
299                     double lambdaLeft = 0;
300                     double lambdaRight = 0;
301                     CompactEpistemicState classLeft = null;
302                     CompactEpistemicState classRight = null;
303 
```

```
304                     StrongNegation negatedLeft = new StrongNegation(
    groundGreaterThanPlausibility.getLeft());
305                     StrongNegation negatedRight = new StrongNegation(
    groundGreaterThanPlausibility.getRight());
306 
307                     for (CompactEpistemicState compactEpistemicState :
    getRelevantEpistemicStates(negatedLeft)) {
308                         classLeft = compactEpistemicState;
309                         lambdaLeft = compactEpistemicState.getLambda(negatedLeft);
310                     }
311 
312                     for (CompactEpistemicState compactEpistemicState :
    getRelevantEpistemicStates(negatedRight)) {
313                         classRight = compactEpistemicState;
314                         lambdaRight = compactEpistemicState.getLambda(negatedRight);
315                     }
316                     if (classLeft instanceof CompactProbabilisticEpistemicState && !(
    classRight instanceof CompactProbabilisticEpistemicState)) {
317                         LOGGER.error("The operands of a qualitative operator must be of
     the same type");
318                     } else if (classLeft instanceof CompactPossibilisticEpistemicState
    && !(classRight instanceof CompactPossibilisticEpistemicState)) {
319                         LOGGER.error("The operands of a qualitative operator must be of
     the same type");
320                     }
321 
322                     if (lambdaLeft < lambdaRight) {
323                         return unifier;
324                     }
325                     return null;
326                 } else {
327                     //TODO: Add exception
328                     LOGGER.error("The operands must be formulas in the language L_G");
329                     return null;
330                 }
331             } else {
332                 LOGGER.error("Formula is not ground.");
333                 return null;
334             }
335         }
336     }
337 
338     private Unifier entails(Negation negation, Unifier unifier) throws Exception {
339         Negation groundNegation = (Negation) negation.substitute(unifier);
340         if (this.languageContains(groundNegation)) {
341             HashSet<CompactEpistemicState> relevantEpistemicStates =
    getRelevantEpistemicStates(groundNegation);
342             for (CompactEpistemicState compactEpistemicState : relevantEpistemicStates)
     {
343                 Unifier unifierValid = compactEpistemicState.entails(negation, unifier)
    ;
344                 if (unifierValid != null) {
345                     return unifierValid;
346                 }
347             }
```

```
348         }
349         return null;
350     }
351 
352     private Unifier entails(RelationalExpression relationalExpression, Unifier unifier)
     throws Exception {
353         RelationalExpression groundRelationalExpression = (RelationalExpression)
    relationalExpression.substitute(unifier);
354         if (this.languageContains(groundRelationalExpression)) {
355             HashSet<CompactEpistemicState> relevantEpistemicStates =
    getRelevantEpistemicStates(groundRelationalExpression);
356             for (CompactEpistemicState compactEpistemicState : relevantEpistemicStates)
     {
357                 Unifier unifierValid = compactEpistemicState.entails(
    relationalExpression, unifier);
358                 if (unifierValid != null) {
359                     return unifierValid;
360                 }
361             }
362         }
363         return null;
364     }
365 
366     @Override
367     public String toString() {
368         String string = "\nGlobal Uncertain Belief:\n\t";
369         Iterator it = epistemicStates.entrySet().iterator();
370         int i = 1;
371         while (it.hasNext()) {
372             Map.Entry pair = (Map.Entry)it.next();
373             string += "Epistemic State " + i + ": \n\t\tDomain: " + pair.getKey().
    toString() + "\n\t\t" + pair.getValue().toString();
374             if (it.hasNext()) {
375                 string += ", \n\t";
376             }
377             i += 1;
378         }
379         string += "\n\t}";
380         return string;
381     }
382 }
```

A.2 EpistemicState Class

```
1  package main.java.uncertain_agentspeak.uncertainty;
2 
3  public abstract class EpistemicState {
4 
5      private HashSet<BeliefAtom> domain;
6 
7      public void setDomain(HashSet<BeliefAtom> domain) {
8          this.domain = domain;
9      }
10 
```

```
11      public HashSet<BeliefAtom> getDomain() {
12          return domain;
13      }
14
15      public boolean languageContains(LogicalExpression logicalExpression) {
16          for (BeliefAtom beliefAtom : logicalExpression.getBeliefAtoms()) {
17              if (!languageContains(beliefAtom)) {
18                  return false;
19              }
20          }
21          return true;
22      }
23
24      public boolean languageContains(BeliefAtom beliefAtom) {
25          for (BeliefAtom beliefAtomDomain : domain) {
26              Unifier unifier = beliefAtom.getTerm().unify(beliefAtomDomain.getTerm());
27              if (unifier != null) {
28                  return true;
29              }
30          }
31          return false;
32      }
33
34      public HashSet<Term> getUnifiers(BeliefAtom beliefAtom, Variable variable) {
35          HashSet<Term> terms = new HashSet<>();
36          for (BeliefAtom beliefAtomDomain : domain) {
37              Unifier unifier = beliefAtom.getTerm().unify(beliefAtomDomain.getTerm());
38              Term term = unifier.get(variable);
39              if (unifier != null) {
40                  terms.add(term);
41              }
42          }
43          return terms;
44      }
45
46      public abstract double getLambda(LogicalExpression logicalExpression) throws
        Exception;
47
48      public LogicalExpression pare(LogicalExpression logicalExpression) throws Exception
         {
49          if(logicalExpression instanceof Contradiction) {
50              return this.pare((Contradiction) logicalExpression);
51          } else if(logicalExpression instanceof Tautology) {
52              return this.pare((Tautology) logicalExpression);
53          } else if(logicalExpression instanceof BeliefAtom) {
54              return this.pare((BeliefAtom) logicalExpression);
55          } else if(logicalExpression instanceof BeliefLiteral) {
56              return this.pare((BeliefLiteral) logicalExpression);
57          } else if(logicalExpression instanceof Conjunction) {
58              return this.pare((Conjunction) logicalExpression);
59          } else if(logicalExpression instanceof Disjunction) {
60              return this.pare((Disjunction) logicalExpression);
61          } else if(logicalExpression instanceof GreaterEqualsPlausibility) {
62              return this.pare((GreaterEqualsPlausibility) logicalExpression);
63          } else if(logicalExpression instanceof GreaterThanPlausibility) {
```

```
64              return this.pare((GreaterThanPlausibility) logicalExpression);
65          } else if(logicalExpression instanceof StrongNegation) {
66              return this.pare((StrongNegation) logicalExpression);
67          } else if(logicalExpression instanceof NegationAsFailure) {
68              return this.pare((NegationAsFailure) logicalExpression);
69          } else if(logicalExpression instanceof Equal) {
70              return this.pare((Equal)logicalExpression);
71          } else if(logicalExpression instanceof NotEqual) {
72              return this.pare((NotEqual)logicalExpression);
73          } else {
74              throw new UnsupportedOperationException("Formula not normalised");
75          }
76      }
77
78      public Conjunction pare(Conjunction conjunction) throws Exception {
79          return new Conjunction(this.pare(conjunction.getLeft()), this.pare(conjunction.
    getRight()));
80      }
81
82      public Disjunction pare(Disjunction disjunction) throws Exception {
83          return new Disjunction(this.pare(disjunction.getLeft()), this.pare(disjunction.
    getRight()));
84      }
85
86      public Primitive pare(GreaterEqualsPlausibility greaterEqualsPlausibility) throws
    Exception {
87          double left = this.getLambda(new StrongNegation(greaterEqualsPlausibility.
    getLeft()));
88          double right = this.getLambda(new StrongNegation(greaterEqualsPlausibility.
    getRight()));
89          if (left <= right) {
90              return new Tautology();
91          } else {
92              return new Contradiction();
93          }
94      }
95
96      public Primitive pare(GreaterThanPlausibility greaterThanPlausibility) throws
    Exception {
97          double left = this.getLambda(new StrongNegation(greaterThanPlausibility.getLeft
    ()));
98          double right = this.getLambda(new StrongNegation(greaterThanPlausibility.
    getRight()));
99          if (left < right) {
100             return new Tautology();
101         } else {
102             return new Contradiction();
103         }
104     }
105
106     public StrongNegation pare(StrongNegation strongNegation) throws Exception {
107         return new StrongNegation(this.pare(strongNegation.getTerm()));
108     }
109
110     public Primitive pare(NegationAsFailure negationAsFailure) throws Exception {
```

```
111         if (this.getLambda(new StrongNegation(negationAsFailure.getTerm())) >= this.
    getLambda(negationAsFailure.getTerm())) {
112             return new Tautology();
113         } else {
114             return new Contradiction();
115         }
116     }
117 
118     public Primitive pare(Equal equal) throws Exception {
119         if (equal.getLeft().equals(equal.getRight())) {
120             return new Tautology();
121         } else {
122             return new Contradiction();
123         }
124     }
125 
126     public Primitive pare(NotEqual notEqual) throws Exception {
127         if (!notEqual.getLeft().equals(notEqual.getRight())) {
128             return new Tautology();
129         } else {
130             return new Contradiction();
131         }
132     }
133 
134     public BeliefAtom pare(BeliefAtom beliefAtom) throws Exception {
135         return beliefAtom;
136     }
137 
138     public BeliefLiteral pare(BeliefLiteral beliefLiteral) throws Exception {
139         return beliefLiteral;
140     }
141 
142     public Contradiction pare(Contradiction contradiction) throws Exception {
143         return contradiction;
144     }
145 
146     public Tautology pare(Tautology tautology) throws Exception {
147         return tautology;
148     }
149 
150     public Unifier entails(LogicalExpression logicalExpression) throws Exception {
151         return entails(logicalExpression, new Unifier());
152     }
153 
154     public Unifier entails(LogicalExpression logicalExpression, Unifier unifier) throws
     Exception {
155         LogicalExpression groundLogicalExpression = logicalExpression.substitute(
    unifier);
156         if (!domain.containsAll(groundLogicalExpression.getBeliefAtoms())) {
157             return null;
158         }
159         LogicalExpression pare = pare(groundLogicalExpression).convertToNNF();
160         LogicalExpression pareNegation = pare(new StrongNegation(pare)).convertToNNF();
161         double pareLambda = getLambda(pare);
162         double pareLambdaNegation = getLambda(pareNegation);
```

```
163         if (pareLambda > pareLambdaNegation) {
164             return unifier;
165         } else {
166             return null;
167         }
168     }
169 
170 }
```

A.3 CompactEpistemicState Class

```
1  package main.java.uncertain_agentspeak.uncertainty.epistemic_states;
2  
3  public class CompactEpistemicState extends EpistemicState {
4  
5      private HashMap<BeliefAtom, Weight> weightedBeliefBase;
6      private double totalWeight;
7  
8      private final Weight initialWeight = new Weight(0,0);
9  
10     public CompactEpistemicState(HashSet<BeliefAtom> domain) throws NotGroundException
       {
11         for (BeliefAtom d : domain) {
12             if (!d.isGround()) {
13                 throw new NotGroundException("The belief atoms in the domain must be
       ground");
14             }
15         }
16         super.setDomain(domain);
17         this.weightedBeliefBase = new HashMap<>();
18         this.totalWeight = 0;
19     }
20 
21     public HashMap<BeliefAtom, Weight> getWeightedBeliefBase() {
22         return weightedBeliefBase;
23     }
24 
25     public Weight getInitialWeight() {
26         return initialWeight;
27     }
28 
29     public double getMinWeight() {
30         return Double.NEGATIVE_INFINITY;
31     }
32 
33     public double getMaxWeight() {
34         return this.totalWeight;
35     }
36 
37     public void revise(BeliefLiteral beliefLiteral, double weight) throws Exception {
38         BeliefAtom beliefAtom = beliefLiteral.getBeliefAtom();
39 
40         if (!this.getDomain().contains(beliefAtom)) {
41             throw new Exception("Belief atom not in domain: " + beliefAtom.toString());
```

```
42          }
43
44          if (weightedBeliefBase.containsKey(beliefAtom)) {
45              Weight oldWeight = weightedBeliefBase.get(beliefAtom);
46              totalWeight -= oldWeight.max();
47              if (beliefLiteral.isPositive()) {
48                  oldWeight.addPositive(weight);
49              } else {
50                  oldWeight.addNegative(weight);
51              }
52              if (oldWeight.equals(this.getInitialWeight())) {
53                  weightedBeliefBase.remove(beliefAtom);
54              } else {
55                  weightedBeliefBase.put(beliefAtom, oldWeight);
56                  totalWeight += oldWeight.max();
57              }
58          } else {
59              Weight newWeight = getInitialWeight().copy();
60              if (beliefLiteral.isPositive()) {
61                  newWeight.addPositive(weight);
62              } else {
63                  newWeight.addNegative(weight);
64              }
65              weightedBeliefBase.put(beliefAtom, newWeight);
66              totalWeight += newWeight.max();
67          }
68      }
69
70      protected Weight getWeight(BeliefAtom beliefAtom) throws Exception {
71          if (super.getDomain().contains(beliefAtom)) {
72              if (this.weightedBeliefBase.containsKey(beliefAtom)) {
73                  return this.weightedBeliefBase.get(beliefAtom);
74              } else {
75                  return getInitialWeight().copy();
76              }
77          } else {
78              throw new Exception("Belief atom not contained in domain");
79          }
80      }
81
82      protected double getWeight(BeliefLiteral beliefLiteral) throws Exception {
83          if (beliefLiteral.isPositive()) {
84              return getWeight((PositiveLiteral) beliefLiteral);
85          } else {
86              return getWeight((NegativeLiteral) beliefLiteral);
87          }
88      }
89
90      protected double getWeight(PositiveLiteral positiveLiteral) throws Exception {
91          BeliefAtom beliefAtom = positiveLiteral.getBeliefAtom();
92          if (getDomain().contains(beliefAtom)) {
93              if (this.weightedBeliefBase.containsKey(beliefAtom)) {
94                  return this.weightedBeliefBase.get(positiveLiteral.getBeliefAtom()).
    getPositive();
95              } else {
```

```
96                  return this.getInitialWeight().getPositive();
97              }
98          } else {
99              throw new Exception("Belief atom not contained in domain");
100         }
101     }
102
103     protected double getWeight(NegativeLiteral negativeLiteral) throws Exception {
104         BeliefAtom beliefAtom = negativeLiteral.getBeliefAtom();
105         if (super.getDomain().contains(beliefAtom)) {
106             if (this.weightedBeliefBase.containsKey(beliefAtom)) {
107                 return this.weightedBeliefBase.get(negativeLiteral.getBeliefAtom()).
    getNegative();
108             } else {
109                 return this.getInitialWeight().getNegative();
110             }
111         } else {
112             throw new Exception("Belief atom not contained in domain");
113         }
114     }
115
116
117     public double getLambda(LogicalExpression logicalExpression) throws Exception {
118         if (!logicalExpression.isGround()) {
119             throw new Exception("Formula is not ground: " + logicalExpression);
120         }
121         LogicalExpression formula = this.pare(logicalExpression);
122         if (!formula.inNNF()) {
123             formula = formula.convertToNNF(false);
124         }
125         return this.getLambda(formula, new HashSet<>());
126     }
127
128     private double getLambda(LogicalExpression logicalExpression, HashSet<BeliefLiteral
    > boundedLiterals) throws Exception {
129         if (logicalExpression instanceof BeliefLiteral){
130             return this.getLambda((BeliefLiteral) logicalExpression, boundedLiterals);
131         } else if (logicalExpression instanceof Conjunction) {
132             return this.getLambda((Conjunction) logicalExpression, boundedLiterals);
133         } else if (logicalExpression instanceof Disjunction) {
134             return this.getLambda((Disjunction) logicalExpression, boundedLiterals);
135         } else if (logicalExpression instanceof Contradiction) {
136             return this.getLambda((Contradiction) logicalExpression, boundedLiterals);
137         } else if (logicalExpression instanceof Tautology) {
138             return this.getLambda((Tautology) logicalExpression, boundedLiterals);
139         } else if (logicalExpression instanceof NegationAsFailure) {
140             return this.getLambda((NegationAsFailure) logicalExpression,
    boundedLiterals);
141         } else if (logicalExpression instanceof GreaterEqualsPlausibility) {
142             return this.getLambda((GreaterEqualsPlausibility) logicalExpression,
    boundedLiterals);
143         } else if (logicalExpression instanceof GreaterThanPlausibility) {
144             return this.getLambda((GreaterThanPlausibility) logicalExpression,
    boundedLiterals);
145         } else if (logicalExpression instanceof Equal) {
```

```
146             return this.getLambda((Equal) logicalExpression, boundedLiterals);
147         } else if (logicalExpression instanceof NotEqual) {
148             return this.getLambda((NotEqual) logicalExpression, boundedLiterals);
149         } else {
150             throw new Exception("Formula not normalised");
151         }
152     }
153 
154     public double getLambda(BeliefLiteral beliefLiteral, HashSet<BeliefLiteral>
    boundedLiterals) throws Exception {
155         HashSet<BeliefLiteral> copyBoundedLiterals = (HashSet<BeliefLiteral>)
    boundedLiterals.clone();
156         copyBoundedLiterals.add(beliefLiteral);
157         double sum = 0;
158         for (BeliefLiteral boundedLiteral : boundedLiterals){
159             if (copyBoundedLiterals.contains(boundedLiteral.negation())) {
160                 return this.getLambda(new Contradiction(), copyBoundedLiterals);
161             } else {
162                 Weight weight = this.getWeight(boundedLiteral.getBeliefAtom());
163                 if (boundedLiteral.isPositive()) {
164                     sum += Math.abs(weight.getPositive() - weight.max());
165                 } else {
166                     sum += Math.abs(weight.getNegative() - weight.max());
167                 }
168             }
169         }
170         return this.getMaxWeight() - sum;
171     }
172 
173     public double getLambda(Contradiction contradiction, HashSet<BeliefLiteral>
    boundedLiterals) {
174         return this.getMinWeight();
175     }
176 
177     public double getLambda(Tautology tautology, HashSet<BeliefLiteral> boundedLiterals
    ) {
178         return this.getMaxWeight();
179     }
180 
181     public double getLambda(Conjunction conjunction, HashSet<BeliefLiteral>
    boundedLiterals) throws Exception {
182         HashSet<BeliefLiteral> copyBoundedLiterals = (HashSet<BeliefLiteral>)
    boundedLiterals.clone();
183         if (conjunction.getLeft().isConjunctive() && conjunction.getRight().
    isDisjunctive()) {
184             copyBoundedLiterals.addAll(conjunction.getLeft().getBeliefLiterals());
185             return this.getLambda(conjunction.getRight(), copyBoundedLiterals);
186         } else if (conjunction.getLeft().isDisjunctive() && conjunction.getRight().
    isConjunctive()) {
187             copyBoundedLiterals.addAll(conjunction.getRight().getBeliefLiterals());
188             return this.getLambda(conjunction.getLeft(), copyBoundedLiterals);
189         } else {
190             throw new Exception("Formula not in language " + this);
191         }
192     }
```

```
193
194     public double getLambda(Disjunction disjunction, HashSet<BeliefLiteral>
        boundedLiterals) throws Exception {
195         return Math.max(getLambda(disjunction.getLeft(), boundedLiterals),getLambda(
        disjunction.getRight(), boundedLiterals));
196     }
197
198     private double getLambda(NegationAsFailure negationAsFailure, HashSet<BeliefLiteral
        > boundedLiterals) throws Exception {
199         return this.getLambda(this.pare(negationAsFailure), boundedLiterals);
200     }
201
202     private double getLambda(GreaterEqualsPlausibility greaterEqualsPlausibility,
        HashSet<BeliefLiteral> boundedLiterals) throws Exception {
203         return this.getLambda(this.pare(greaterEqualsPlausibility), boundedLiterals);
204     }
205
206     private double getLambda(GreaterThanPlausibility greaterThanPlausibility, HashSet<
        BeliefLiteral> boundedLiterals) throws Exception {
207         return this.getLambda(this.pare(greaterThanPlausibility), boundedLiterals);
208     }
209
210     private double getLambda(Equal equal, HashSet<BeliefLiteral> boundedLiterals)
        throws Exception {
211         return this.getLambda(this.pare(equal), boundedLiterals);
212     }
213
214     private double getLambda(NotEqual notEqual, HashSet<BeliefLiteral> boundedLiterals)
         throws Exception {
215         return this.getLambda(this.pare(notEqual), boundedLiterals);
216     }
217
218
219     @Override
220     public String toString() {
221         String string = "{";
222         Iterator it = weightedBeliefBase.entrySet().iterator();
223         while (it.hasNext()) {
224             Map.Entry pair = (Map.Entry)it.next();
225             string += "[" + pair.getKey().toString() + ", " + pair.getValue().toString
        () + "]";
226             if (it.hasNext()) {
227                 string += ", ";
228             }
229             it.remove(); // avoids a ConcurrentModificationException
230         }
231         string += "}";
232         return string;
233     }
234 }
```

A.4 CompactProbabilisticEpistemicState Class

```
1 package main.java.uncertain_agentspeak.uncertainty.epistemic_states.
```

```
    compact_epistemic_states;

public class CompactProbabilisticEpistemicState extends CompactEpistemicState {

    public CompactProbabilisticEpistemicState(HashSet<BeliefAtom> atoms) throws
    Exception {
        super(atoms);
    }

    public double getMinWeight() {
        return 0;
    }

    public double getMaxWeight() {
        return 1;
    }

    @Override
    public Weight getInitialWeight() {
        return new Weight(0.5, 0.5);
    }

    @Override
    public void revise(BeliefLiteral beliefLiteral, double weight) throws Exception {
        BeliefAtom beliefAtom = beliefLiteral.getBeliefAtom();

        if (!super.getDomain().contains(beliefAtom)) {
            throw new Exception("Belief atom is not in domain: " + beliefAtom.toString
    ());
        }

        /** If belief atom in domain revise it's corresponding weight, otherwise revise
     initial probabilistic weight
         * **/
        Weight w;
        if (this.getWeightedBeliefBase().containsKey(beliefAtom)) {
            w = this.getWeightedBeliefBase().get(beliefAtom);
        } else {
            w = this.getInitialWeight().copy();
        }

        /** Check if the belief literal is positive or negative and revise accordingly
         * **/
        if (beliefLiteral.isPositive()) {
            w.setPositive(weight);
            w.setNegative(this.getMaxWeight() - weight);
        } else {
            w.setNegative(weight);
            w.setPositive(this.getMaxWeight() - weight);
        }

        /** If the atom is contained in the weighted belief base and it's revised
    weight is equal to the initial
         * probabilistic weight then remove it from the weighted belief base, otherwise
     put the belief atom and
```

```
51          * its associated weight into the weighted belief base.
52          * **/
53         if (this.getWeightedBeliefBase().containsKey(beliefAtom) && w.equals(this.
    getInitialWeight())) {
54             this.getWeightedBeliefBase().remove(beliefAtom);
55         } else {
56             this.getWeightedBeliefBase().put(beliefAtom, w);
57         }
58 
59     }
60 
61     public double getProbability(BeliefLiteral beliefLiteral) throws Exception {
62         return this.getWeight(beliefLiteral);
63     }
64 
65     @Override
66     public double getLambda(BeliefLiteral beliefLiteral, HashSet<BeliefLiteral>
    beliefLiterals) throws Exception {
67         return this.getProbability(beliefLiteral);
68     }
69 
70     @Override
71     public double getLambda(Contradiction contradiction, HashSet<BeliefLiteral>
    boundedLiterals) {
72         return this.getMinWeight();
73     }
74 
75     @Override
76     public double getLambda(Tautology tautology, HashSet<BeliefLiteral> boundedLiterals
    ) {
77         return this.getMaxWeight();
78     }
79 
80     @Override
81     public double getLambda(Conjunction conjunction, HashSet<BeliefLiteral>
    boundedLiterals) throws Exception {
82         SATsolver satSolver = new SATsolver();
83         if (satSolver.solve(conjunction)) {
84             return this.getLambda(conjunction.getLeft()) * this.getLambda(conjunction.
    getRight());
85         } else {
86             return this.getMinWeight();
87         }
88 
89     }
90 
91     @Override
92     public double getLambda(Disjunction disjunction, HashSet<BeliefLiteral>
    boundedLiterals) throws Exception {
93         return this.getLambda(disjunction.getLeft()) + this.getLambda(disjunction.
    getRight()) - this.getLambda(new Conjunction(disjunction.getLeft(), disjunction.
    getRight()));
94     }
95 
96     @Override
```

```
97      public String toString() {
98          String string = "{ ";
99          Iterator it = this.getWeightedBeliefBase().entrySet().iterator();
100         while (it.hasNext()) {
101             Map.Entry pair = (Map.Entry)it.next();
102             string += "\n\t\t\tProb( " + pair.getKey().toString() + ", " + pair.
    getValue().toString() + " )";
103             if (it.hasNext()) {
104                 string += ", ";
105             }
106         }
107         string += " \n\t\t}";
108         return string;
109     }
110 }
```

A.5 CompactPossibilisticEpistemicState Class

```
1  package main.java.uncertain_agentspeak.uncertainty.epistemic_states.
       compact_epistemic_states;
2
3  public class CompactPossibilisticEpistemicState extends CompactEpistemicState {
4
5      public CompactPossibilisticEpistemicState(HashSet<BeliefAtom> atoms) throws
       Exception {
6          super(atoms);
7      }
8
9      @Override
10     public HashSet<BeliefAtom> getDomain() {
11         return super.getDomain();
12     }
13
14     @Override
15     public HashMap<BeliefAtom, Weight> getWeightedBeliefBase() {
16         return super.getWeightedBeliefBase();
17     }
18
19     @Override
20     public Weight getInitialWeight() {
21         return new Weight(1, 1);
22     }
23
24     @Override
25     public double getMinWeight() {
26         return 0;
27     }
28
29     @Override
30     public double getMaxWeight() {
31         return 1;
32     }
33
34     @Override
```

```
35      /** Revise as N(beliefLiteral) >= weight **/
36      public void revise(BeliefLiteral beliefLiteral, double weight) throws Exception {
37          BeliefAtom beliefAtom = beliefLiteral.getBeliefAtom();
38
39          if (!this.getDomain().contains(beliefAtom)) {
40              throw new Exception("Belief atom is not in domain");
41          }
42
43          for (BeliefAtom beliefAtomDomain : this.getDomain()) {
44
45              /** If belief atom in domain revise it's corresponding weight, otherwise
    revise initial probabilistic weight
46               * **/
47              Weight w;
48              if (this.getWeightedBeliefBase().containsKey(beliefAtomDomain)) {
49                  w = this.getWeightedBeliefBase().get(beliefAtomDomain);
50              } else {
51                  w = this.getInitialWeight().copy();
52              }
53
54              /** Check if the belief literal equals the domain belief atom, if it is
    positive or negative and then
55               * revise accordingly
56               * **/
57              double alpha;
58              Weight oldWeight = getPossibilityMeasure(beliefAtom);
59              if (beliefLiteral.isPositive()) {
60                  alpha = Math.max(oldWeight.getPositive(), 1 - weight);
61              } else {
62                  alpha = Math.max(1 - weight, oldWeight.getNegative());
63              }
64              if (beliefLiteral.getBeliefAtom().equals(beliefAtomDomain)) {
65                  if (beliefLiteral.isPositive()) {
66                      w.setNegative(Math.min(w.getNegative(), Math.min(1-weight, alpha)))
    ;
67                      w.setPositive(Math.min(w.getPositive(), alpha));
68                  } else {
69                      w.setNegative(Math.min(w.getNegative(), alpha));
70                      w.setPositive(Math.min(w.getPositive(),Math.min(1-weight, alpha)));
71                  }
72              } else {
73
74                  w.setPositive(Math.min(w.getPositive(),alpha));
75                  w.setNegative(Math.min(w.getNegative(),alpha));
76              }
77
78
79              /** If the atom is contained in the weighted belief base and it's revised
    weight is equal to the initial
80               * possibilistic weight then remove it from the weighted belief base,
    otherwise put the belief atom and
81               * its associated weight into the weighted belief base.
82               * **/
83              if (w.equals(this.getInitialWeight())) {
84                  this.getWeightedBeliefBase().remove(beliefAtomDomain);
```

```
85              } else {
86                  this.getWeightedBeliefBase().put(beliefAtomDomain, w);
87              }
88          }
89      }
90
91      public Weight getPossibilityMeasure(BeliefAtom beliefAtom) throws Exception {
92          return this.getWeight(beliefAtom);
93      }
94
95      public double getPossibilityMeasure(BeliefLiteral beliefLiteral) throws Exception {
96          return this.getWeight(beliefLiteral);
97      }
98
99      @Override
100     public double getLambda(BeliefLiteral beliefLiteral, HashSet<BeliefLiteral>
    boundedLiterals) throws Exception {
101         HashSet<BeliefLiteral> boundedLiteralsCopy = (HashSet<BeliefLiteral>)
    boundedLiterals.clone();
102         boundedLiteralsCopy.add(beliefLiteral);
103         for (BeliefLiteral boundedLiteral : boundedLiterals) {
104             if (boundedLiteralsCopy.contains(boundedLiteral.negation())) {
105                 return getLambda(new Contradiction(), boundedLiteralsCopy);
106             }
107         }
108         return getPossibilityMeasure(beliefLiteral);
109     }
110
111     @Override
112     public String toString() {
113         String string = "{ ";
114         Iterator it = this.getWeightedBeliefBase().entrySet().iterator();
115         while (it.hasNext()) {
116             Map.Entry pair = (Map.Entry)it.next();
117             string += "\n\t\t\tPi( " + pair.getKey().toString() + ", " + pair.getValue
    ().toString() + " )";
118             if (it.hasNext()) {
119                 string += ", ";
120             }
121         }
122         string += " \n\t\t}";
123         return string;
124     }
125
126 }
```

APPENDIX B

Logical Expression Class Diagrams

This appendix provides the class diagrams for the different types of logical expressions. All of the classes shown here are extensions of the logical expression class as shown in Figure 3.6.

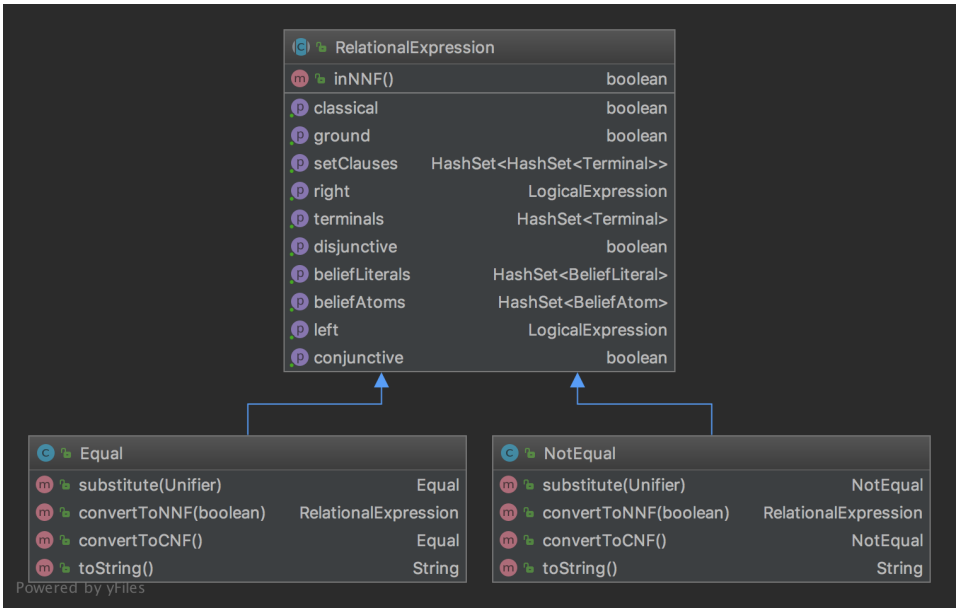


Figure B.1: Relational expression class diagrams

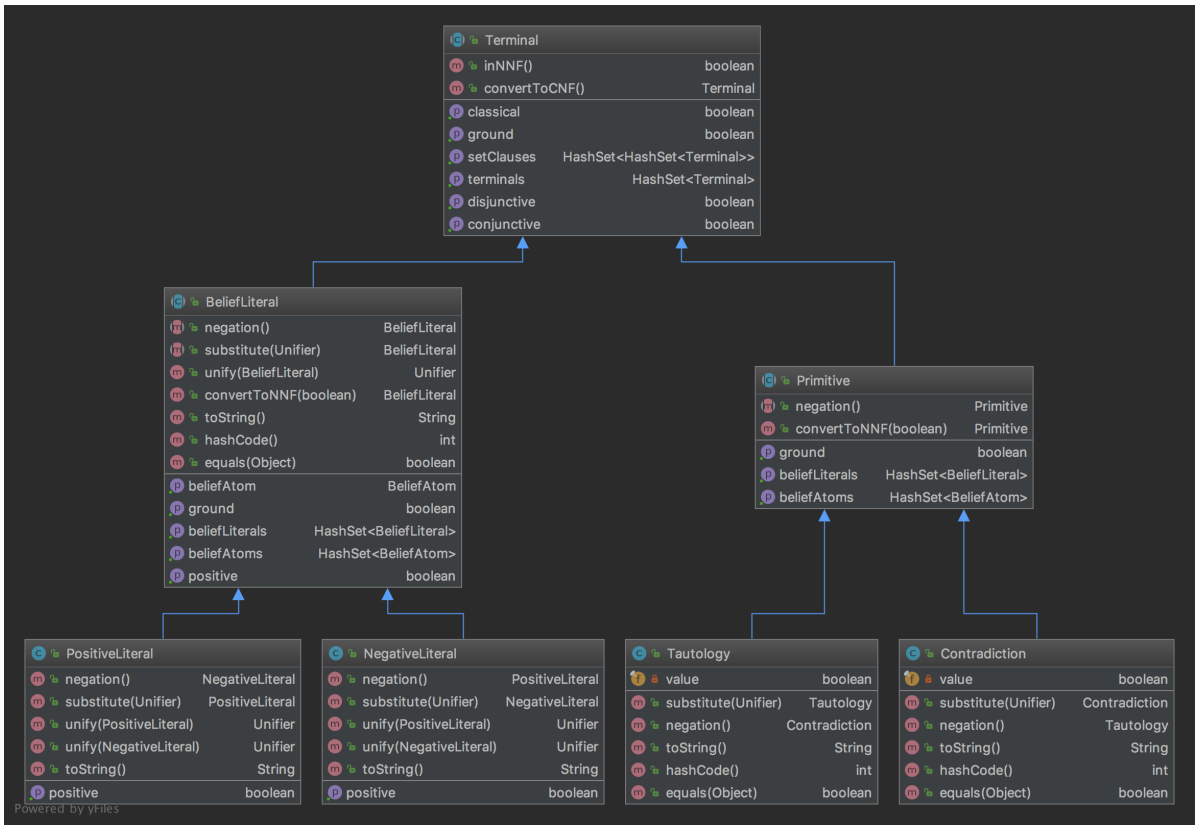


Figure B.2: Terminal class diagrams

Figure B.3: Operator class diagrams

BIBLIOGRAPHY

Alchourron, Carlos E., Peter Gardenfors, and David Makinson (1985). "On the Logic of Theory Change: Partial Meet Contraction and Revision Functions". In: *J. Symbolic Logic* 50.2, pp. 510–530.

Ana, Casali, Godo Lluis, and Sierra Carles (2005). "Graded BDI Models for Agent Architectures". In: *Computational Logic in Multi-Agent Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 126–143.

Andrew, Alex M. (2001). "REASONING ABOUT RATIONAL AGENTS, by Michael Wooldridge, MIT Press, Cambridge, Mass., 200, Xi+227Pp., ISBN 0-262-23213 (Hardback &Pound;23.50)". In: *Robotica* 19.4, pp. 459–462.

Ash, R.B. (2012). *Basic Probability Theory*. Dover Books on Mathematics Series. Dover Publications, Incorporated.

Baader, Franz and Wayne Snyder (1999). *Unification Theory*.

Bauters, Kim et al. (2017). "Managing different sources of uncertainty in a BDI framework in a principled way with tractable fragments". In: *Journal of Artificial Intelligence Research* 58, pp. 731–775.

Benferhat, Salem, Didier Dubois, and Henri Prade (1998). "Practical Handling of Exception-Tainted Rules and Independence Information in Possibilistic Logic". In: *Applied Intelligence* 9.2, pp. 101–127.

Bordini, Rafael H. and Jomi F. Hübner (2006). "BDI agent programming in AgentSpeak using Jason". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 3900 LNAI, pp. 143–164.

Bordini, Rafael H and Jomi F Hübner (2007). "A Java-based interpreter for an extended version of AgentSpeak". In:

Bordini, Rafael H., Jomi Fred Hübner, and Michael Wooldridge (2007). *Programming Multi-Agent Systems in AgentSpeak using Jason*, pp. 1–273.

Bordini, Rafael H. and Álvaro F. Moreira (2002). "Proving the Asymmetry Thesis Principles for a BDI Agent-Oriented Programming Language". In: *Electronic Notes in Theoretical Computer Science* 70.5. CLIMA'2002, Computational Logic in Multi-Agent Systems (FLoC Satellite Event), pp. 108–125.

Bordini, Rafael H. and Álvaro F. Moreira (2004). “Proving BDI Properties of Agent-Oriented Programming Languages: The asymmetry thesis principles in AgentSpeak(L)”. In: *Annals of Mathematics and Artificial Intelligence* 42.1, pp. 197–226.

Bratman, Michael (1987). *Intention, Plans, and Practical Reason*. Center for the Study of Language and Information.

Casali, Ana, Lluís Godo, and Carles Sierra (2011). “A graded BDI agent model to represent and reason about preferences”. In: *Artificial Intelligence* 175.7. Representing, Processing, and Learning Preferences: Theoretical and Practical Challenges, pp. 1468–1478.

Clement, Daniel (1987). *The Intentional Stance*.

Cohen, Philip R. and Hector J. Levesque (1990). “Intention is choice with commitment”. In: *Artificial Intelligence* 42.2, pp. 213–261.

Dastani, Mehdi (2008). “2APL: A Practical Agent Programming Language”. In: *Autonomous Agents and Multi-Agent Systems* 16.3, pp. 214–248.

Dubois, Didier, Serafin Moral, and Henri Prade (1998). “Belief Change Rules in Ordinal and Numerical Uncertainty Theories”. In: *Belief Change*. Ed. by Didier Dubois and Henri Prade. Dordrecht: Springer Netherlands, pp. 311–392.

Dubois, Didier and Henri Prade (2015). “Possibility Theory and Its Applications: Where Do We Stand?” In: 18.

Edwin, Jaynes (2003). *Probability Theory: the Logic of Science*. Cambridge university press.

Fernández, Víctor et al. (2010). “Analysis of Jason’s performance for crowd simulations”. In:

Georgeff, Michael et al. (1999). “The Belief-Desire-Intention Model of Agency”. In: *Intelligent Agents V: Agents Theories, Architectures, and Languages*. Ed. by Jörg P. Müller, Anand S. Rao, and Munindar P. Singh. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 1–10.

Herzig, Andreas et al. (2017). “BDI Logics for BDI Architectures: Old Problems, New Perspectives”. In: *KI - Künstliche Intelligenz* 31.1, pp. 73–83.

Hübner, Jomi and Jaime Sichman (2009). *SACI: Simple Agent Communication Infrastructure*.

Ingrand, Francois F., Michael P. Georgeff, and Anand S. Rao (1992). “An Architecture for Real-Time Reasoning and System Control”. In: *IEEE Expert: Intelligent Systems and Their Applications* 7.6, pp. 34–44.

Jennings, Nicholas R., Katia Sycara, and Michael Wooldridge (1998). “A Roadmap of Agent Research and Development”. In: *Autonomous Agents and Multi-Agent Systems* 1.1, pp. 7–38.

Kern-Isberner, Gabriele and Thomas Lukasiewicz (2017). “Special Issue on Challenges for Reasoning under Uncertainty, Inconsistency, Vagueness, and Preferences”. In: *KI - Künstliche Intelligenz* 31.1, pp. 5–8.

Kravari, Kalliopi and Nick Bassiliades (2015a). “A Survey of Agent Platforms”. In: *Journal of Artificial Societies and Social Simulation* 18.1.

— (2015b). “A Survey of Agent Platforms”. In: *Journal of Artificial Societies and Social Simulation* 18.1, p. 11.

Kwisthout, Johan and Mehdi Dastani (2006). “Modelling Uncertainty in Agent Programming”. In: *Proceedings of the Third International Conference on Declarative Agent Languages and Technologies*. DALT'05. Utrecht, The Netherlands: Springer-Verlag, pp. 17–32.

Ma, Jianbing and Weiru Liu (2011a). “A framework for managing uncertain inputs: An axiomization of rewarding”. In: *International Journal of Approximate Reasoning* 52.7, pp. 917–934.

— (2011b). “A framework for managing uncertain inputs: An axiomization of rewarding”. In: *International Journal of Approximate Reasoning* 52.7. Selected Papers - Uncertain Reasoning Track - FLAIRS 2009, pp. 917–934.

Machado, Rodrigo and Rafael Bordini (2003a). *Running AgentSpeak(L) Agents on SIM AGENT*.

— (2003b). *Running AgentSpeak(L) Agents on SIM AGENT*.

Mascardi, Viviana, Daniela Demergasso, and Davide Ancona (2005). *Languages for Programming BDI-style Agents: an Overview*.

Moreira, Alvaro F. and Rafael H. Bordini (2002). *An Operational Semantics for a BDI Agent-Oriented Programming Language*.

Moreira, Álvaro F., Renata Vieira, and Rafael H. Bordini (2004). “Extending the Operational Semantics of a BDI Agent-Oriented Programming Language for Introducing Speech-Act Based Communication”. In: *Declarative Agent Languages and Technologies*. Ed. by João Leite et al. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 135–154.

Nair, Ranjit and Milind Tambe (2005). “Hybrid BDI-POMDP Framework for Multiagent Teaming”. In: *J. Artif. Int. Res.* 23.1, pp. 367–420.

Nebel, B. (1995). “Based Revision Operations and Schemes: Semantics, Representation, and Complexity”. In: *Proceedings of the ISSEK94 Workshop on Mathematical and Statistical Methods in Artificial Intelligence*. Ed. by G. Della Riccia, R. Kruse, and R. Viertl. Vienna: Springer Vienna, pp. 157–170.

Parr, Terence (2013). *The Definitive ANTLR 4 Reference*. 2nd. Pragmatic Bookshelf.

Rao, Anand S. (1996). “AgentSpeak(L): BDI agents speak out in a logical computable language”. In: L, pp. 42–55.

Rao, Anand S. and Michael P. Georgeff (1991). “Modeling Rational Agents Within a BDI-architecture”. In: *Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning*. KR'91. Cambridge, MA, USA: Morgan Kaufmann Publishers Inc., pp. 473–484.

Spohn, Wolfgang (1988). “Ordinal Conditional Functions: A Dynamic Theory of Epistemic States”. In: *Causation in Decision, Belief Change, and Statistics: Proceedings of the Irvine Conference on Probability and Causation*. Ed. by William L. Harper and Brian Skyrms. Dordrecht: Springer Netherlands, pp. 105–134.

TILAB (2009). *JADE (Java Agent DEvelopment Framework)*. `http://jade.tilab.com/`.

Williams, Mary-Anne (1995). “Iterated Theory Base Change: A Computational Model”. In: *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*. IJCAI’95. Montreal, Quebec, Canada: Morgan Kaufmann Publishers Inc., pp. 1541–1547.

Zadeh, L. A. (1999). “Fuzzy Sets As a Basis for a Theory of Possibility”. In: *Fuzzy Sets Syst.* 100, pp. 9–34.