A Multi-Agent Framework using Answer Set Programming

Peter Rhys Evans

Batchelor of Science in Computer Science with Honours
The University of Bath
May 2007
This dissertation may be made available for consultation within the University Library and may be photocopied or lent to other libraries for the purposes of consultation.

Signed:
A Multi-Agent Framework using Answer Set Programming

Submitted by: Peter Rhys Evans

COPYRIGHT

Attention is drawn to the fact that copyright of this dissertation rests with its author. The Intellectual Property Rights of the products produced as part of the project belong to the University of Bath (see http://www.bath.ac.uk/ordinances/#intelprop). This copy of the dissertation has been supplied on condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no quotation from the dissertation and no information derived from it may be published without the prior written consent of the author.

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Batchelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Signed:
Abstract

Deductive reasoning is a common way of implementing agents in a multi-agent setting. One of the more popular ways of representing an agents knowledge and reasoning capabilities is by using logic programs. The idea behind this approach is to establish a theory/program that details the best course of action to take in any situation. However a problem with this approach relates to an agents ability to represent the real world accurately and in enough detail. One solution is to use Answer Set Programming. This relatively new logic formalism tries to allow agents knowledge to mirror the real world to a closer degree of accuracy.

This project creates a generic logic based multi-agent system capable of supporting various multi-agent functions, where the agents knowledge and reasoning capabilities are represented by various extensions of answer set programs.
I would like to thank Dr Marina De Vos for her help and guidance during the project, my partner Laura for putting up with me while writing this and Andy for all his help. Finally I would like to thank my family for helping get me here in the first place.
## Contents

1 Introduction

1.1 Aims ................................................................. 1
1.2 Dissertation Outline ........................................ 1

2 Literature Review .................................................. 3

2.1 Introduction ........................................................ 3
2.2 Logic ................................................................. 3
   2.2.1 Types of Logic Systems .................................... 4
2.3 Logic Programming .............................................. 6
   2.3.1 Prolog ........................................................... 6
   2.3.2 Problems with Prolog and Logical Programming .......... 6
   2.3.3 Answer Set Programming ................................... 7
   2.3.4 AnsProlog* ................................................... 7
   2.3.5 Extensions of ASP ........................................... 10
2.4 Agents .............................................................. 11
   2.4.1 Environments ................................................ 11
   2.4.2 Actions ........................................................ 12
   2.4.3 Goals ............................................................ 12
   2.4.4 Multi-Agent Systems ....................................... 13
   2.4.5 Agent Communication ..................................... 13
   2.4.6 Trust ........................................................... 13
   2.4.7 Institutions .................................................. 14
2.5 Tools ............................................................... 15
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>System Structure</td>
<td>27</td>
</tr>
<tr>
<td>5.1</td>
<td>Interaction between Logic and Institutional Agents</td>
<td>38</td>
</tr>
<tr>
<td>5.2</td>
<td>Logic Agent Dataflow</td>
<td>44</td>
</tr>
<tr>
<td>5.3</td>
<td>Ontology Structure</td>
<td>49</td>
</tr>
<tr>
<td>A.1</td>
<td>All agents with same solver</td>
<td>68</td>
</tr>
<tr>
<td>A.2</td>
<td>All agents with different solver</td>
<td>68</td>
</tr>
<tr>
<td>A.3</td>
<td>All agents and an Institutional Agent</td>
<td>69</td>
</tr>
<tr>
<td>A.4</td>
<td>Wrong file-name error message</td>
<td>69</td>
</tr>
<tr>
<td>A.5</td>
<td>Agent on Unix using OCT</td>
<td>70</td>
</tr>
<tr>
<td>A.6</td>
<td>Sniffer Agent Example</td>
<td>71</td>
</tr>
<tr>
<td>A.7</td>
<td>Sniffer Agent Example</td>
<td>72</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

In recent years traditional symbolic AI has been replaced by deductive reasoning agents (De Vos et al. 2005). This has led to the development of theorem proving within multi-agent frameworks. One of the ideas behind this approach is to establish a theory that details the best course of action to take in any situation, using logic programming.

One of the problems with this approach is being able to represent the real world accurately and in enough detail in a logical formalism (De Vos et al. 2006). Answer Set Programming is a new logic formalism that tries to improve upon some of the deficiencies of other logic programming languages. The improvements to ASP help knowledge representation in agents to mirror the real world to a closer degree of accuracy.

In this project we will discuss the foundation of logic, logic programming and the importance of ASP over other logic formalisms. We will also look at multi-agent systems, why logic is so useful in representing their knowledge and reasoning capabilities and appraise the tools currently available to help with their development.

1.1 Aims

The purpose of this project is to design and implement a multi-agent framework where the knowledge and reasoning capabilities of each agent are represented by answer set programs. The framework will allow for many types of answer set solvers and will also work for institutions.

This projects key aim is to create a generic framework that can be easily reused to create a give the agents in the system a specific purpose.

1.2 Dissertation Outline

This dissertation reviews some of the main literature relevant to the topics covered which will identify current technologies that could be used and will help better understand ASP,
Logic Programming and the tools that can be used. It will then outline the development of the system from its requirements, looking at the overall design of the system from those requirements and then detailing how the system was implemented.

The dissertation concludes with the testing of the system, a critical evaluation of the project and covers the potential for expansion and further research.
Chapter 2

Literature Review

2.1 Introduction

The literature review is organised into the following sections:

- Section 2 contains a description of different logic formalisms and how they lead to logic programming.
- Section 3 details the logic programming paradigm, its main problems and introduces ASP as a potential solution to those problems. It also looks at extensions to ASP and potential answer set solvers that can be used.
- Section 4 gives an overview of Agents and Multi-Agent Systems. This will include their beliefs, desires and intentions and how Logic Programming can be applied to aid in agent design.
- Section 5 looks at the tools currently available for developing Multi-Agent Systems, paying particular attention to JADE and Protégé.

2.2 Logic

In order to gain an understanding of why logic is so popular with the agents’ community, we will look at what logic is and some of the main “types” of logic formalisms used in computing.

Logic has a broad definition covering both its philosophical and formal mathematical interpretations. Brain (2004) has given a simplified definition of logic (in its mathematical sense) which shall be referred to in the rest of this paper:

*Logic is formal system for representing and manipulating the relations between concepts.*
CHAPTER 2. LITERATURE REVIEW

Expressing these concepts in this way allows us to reason formally about the relations between them. Reasoning formally allows us to construct valid arguments that can be defended rigorously and mathematically (Huth & Ryan 2004). An example of logical deduction, taken from Bools et al. (2003):

1. A square or cube of a number is a power of that number.
2. A power of a power of a number is a power of that number.
3. Sixty-four is the cube of four and four is a square of two.
4. Therefore, sixty-four is a power of two.

Arguments 1-3 logically imply the conclusion 4, as if the premises are true, then the conclusion must also be true.

2.2.1 Types of Logic Systems

There are many different types of logic systems and most vary quite significantly from each other. Most types of logic have several things in common: a set of literals, that contain the most basic information, a set of connectives, these are operators that connect literals, and a series of function, variables and functors, that add expressiveness to the language (Brain 2004). Other types of logic include Temporal Logic, non-monotonic logic and event calculus. We will now focus on two of the more familiar types of logic systems; Propositional (or Classic) Logic and Predicate Logic.

Propositional Logic

Propositional Logic is probably what most people would associate with the mathematical idea of logic. One of the most basic “building blocks” of formal logic is a proposition. A proposition is a statement that can be assigned a truth value (i.e. it is either true or false). Propositions can be constructed from others by using the logical connectives AND, OR, IMPLIES and NOT, noted as $\land$, $\lor$, $\rightarrow$, $\neg$ (Haggarty 2002).

In order to determine the truth values of the new propositions we need to know what effect each logical connective has on the original propositions. These effects are usually displayed using truth tables, an example of a truth table for the NOT connective, taken from Haggarty (2002), is displayed in Table 2.1.

<table>
<thead>
<tr>
<th>$P$</th>
<th>$\neg P$</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

If $P$ is a proposition then $\neg P$ is its negation, i.e. they always have the opposite truth value. If $P$ and $Q$ are two propositions then $P \land Q$ is the conjunction of the two, i.e. it is only true when both $P$ and $Q$ are true. If $P$ and $Q$ are two propositions then $P \lor Q$ is the disjunction of the two, i.e. it is true when either $P$ or $Q$ or both are true.
The truth table for the implies connective is slightly more complicated, as $P \rightarrow Q$ can also be read as \textit{if} $P$ \textit{then} $Q$ we get the truth table seen in Table 2.2.

<table>
<thead>
<tr>
<th>$P$</th>
<th>$Q$</th>
<th>$P \rightarrow Q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>True</td>
</tr>
</tbody>
</table>

**Predicate Logic**

In propositional logic you can assign truth values to simple statements, such as “he was a doctor”; however there may be a need to refer to a group of objects in a statement that may have differing truth values, such as “some of them were doctors” (Haggarty 2002). In order to deal with this type of statement we need to expand our set of symbols to be able to deal with “some of . . .” and “all of . . .”. For this purpose we introduce the predicate quantifiers \textit{for all}, noted by $\forall$ and \textit{there exists}, noted by $\exists$. When a predicate is quantified it produces a proposition and so a quantified predicate can be assigned a truth value (Huth & Ryan 2004).

We will now look at an example given by [Russell & P (2003)](Author). Given the statement “Everybody loves someone”, we can easily see that this mean that for every person there is someone that person loves, so we can then convert this into the following logic statement.

$$\forall x \exists y \text{Loves}(x, y)$$

The ideas of logic and the way the formalisms can represent data and knowledge in a formal way has led to the development of a novel programming paradigm; logic programming. We will look at logic programming and some of the more prominent languages in the next section.
2.3 Logic Programming

Logic programming is a declarative style of programming based on classic logic. In a more typical programming language the details of a process for finding a solution are described explicitly, programming is very different in logic programming languages. In a logic programming language you define a set of objects, a set of rules for them and a set of goals and constraints that must be satisfied. The program will then compute the solution/solutions to the problem based on these rules without the need to explicitly state the processes by which you achieve the solution.

The way that logic programs reason and can represent data has ensured their widespread usage in the field of artificial intelligence (AI), and consequently agents programming. The most widely used, and probably best known, logic programming language is Prolog. We will now look at Prolog in more detail and discuss its problems.

2.3.1 Prolog

Prolog uses Horn Clauses, which are a restricted set of first order predicate calculus. Horn Clauses take the form:

\[ p_1 \land \ldots \land p_n \land \neg p_{n+1} \land \ldots \land \neg p_m \rightarrow q \]

Where \( p_1, \ldots, p_n, p_{n+1}, \ldots, p_m \) are called the body of the clause, \( q \) is called the head and \( p_1, \ldots, p_n, p_{n+1}, \ldots, p_m \) and \( q \) are predicates (Brakto 2001).

One of Prolog’s more controversial features is the cut operator \(!\). When the interpreter finds a cut operator it will stop looking for possible solutions, i.e. stops back-tracking. Many argue that this is not part of logic and means that Prolog is no longer a true logic programming language.

2.3.2 Problems with Prolog and Logical Programming

There are several problems with Prolog that mean it is not used for knowledge representation in agents (Baral 2003).

- One of the big disadvantages of Prolog is that it uses negation as failure, which gives rise to the Closed World Assumption. The closed world assumption states that not knowing a statement is true is the same as knowing that it is false. This poses a problem as in knowledge representation this is not an appropriate assumption, knowing something is not true is very different to knowing something is false.

- The unification algorithm that Prolog uses to answer queries makes it very easy to write a program that will enter an infinite loop and it will not know that it has entered one.
As mentioned in Section 3.1 the use of the cut operator, as well as the significance of the order in the body, means Prolog uses non-logical operators which many people class as a problem.

As well as problems with Prolog there are also problems with logic programming itself. Although current Logic Programming languages are very good at representing static knowledge, they can have trouble representing knowledge in an open and dynamic environment, such as those seen in Multi-Agent Systems (Alferes et al. 2001). To overcome these problems, the logic must be extended to incorporate knowledge from different sources and knowledge that can change over time.

Answer Set Programming is such an extension which is covered in Section 2.3.3

2.3.3 Answer Set Programming

Answer Set Programming (ASP) is a current field of research in logic programming. ASP is a formal, logical language designed for declarative problem solving, knowledge representation and real world reasoning (De Vos et al. 2005). In ASP a logic program describes the requirements that must be fulfilled by the solution of a problem. The answer sets of the logic program correspond to the solutions of the problem (Heymans et al. 1999).

Most logic programming languages are implemented using subsets of classical logic, however ASP is different. ASP is based on analysis of the constructs that are required in reasoning applications.

ASP overcomes some of the problems associated with Prolog. The semantics of ASP allow for negation as failure and another type of negation: constraint-based negation. Constraint-based negation removes the closed world assumption, seen in Prolog. In this form of negation just because a statement cannot be proved true it is not assumed that it is false. Constraint-based negation also prevents certain combinations of atoms being true at the same time; both of these are beneficial as it allows ASP to reason about incomplete information.

ASP has already been successfully applied to several areas including planning, diagnosis, game theory and Multi-Agent Systems (Heymans et al. 1999).

2.3.4 AnsProlog*

Answer set programming in logic, or AnsProlog for short, is an unambiguous language that can be used to formalise Answer Set Programming (Baral 2003). The * means that there are no restrictions placed on the language.

AnsProlog* has been developed as a logic programming language that focuses purely on answer set semantics.

Below is a definition of the syntax and semantics, and their notations, used in AnsProlog* as obtained from Baral (2003).
Syntax

An AnsProlog* program contains a finite set of rules and is used to express a set of atoms of an answer set framework (ASF).

Definition 1: An ASF consists of:

- Two alphabets, one for the axioms and one for queries
- Two languages defined for the alphabets
- A set of axioms
- A relation between the set of axioms and queries.
- The axiom alphabet is usually referred to as simply the alphabet. (The query alphabet will not be discussed here.)

Definition 2: The alphabet of an ASF contains seven classes of symbols:

- Variables (an arbitrary string of letters and numbers starting with an upper-case letter)
- Constants (also known as object constants)
- Function Symbols (an arbitrary string of letters and numbers starting with a lower-case letter)
- Predicate Symbols
- Connectives (¬, or, ←, \texttt{not}, ‘,’)
- Punctuation Symbols (‘(, ’), ‘.’)
- The Symbol -

Now that we have defined what constitutes an Answer Set Framework we will now look in more detail at the classes of symbols that define the language.

Definition 3: A term is defined as:

- A variable is a term
- A constant is a term
- If $f$ is an n-ary function symbol and $t_1, \ldots, t_n$ are terms then $f(t_1, \ldots, t_n)$ is a term
CHAPTER 2. LITERATURE REVIEW

Definition 4: A term is ground if it contains no variables.

Definition 5: Herbrand Universe and Herbrand Base

- The Herbrand Universe of a language is the set of all ground terms that can be formed in the language.
- An atom is of the form \( p(t_1, \ldots, t_n) \), where \( p \) is a predicate and each \( t \) is a term. Where all \( t \)'s are ground then the atom is ground.
- The Herbrand Base of a language is the set of all ground atoms that can be formed with the predicates from the languages and the terms form the Herbrand Universe.
- A literal is either an atom or one preceded by \( \neg \)

When all atoms are ground the literal is also ground.

Literals are individual pieces of information and are the building blocks to the creation of more complex statements. In order to construct more complicated information we will now look at how to form rules in AnsProlog*.

Definition 6: A rule is of the form:

\[
L_0 \lor \ldots \lor L_k \leftarrow L_{k+1}, \ldots, L_m, \not L_{m+1}, \ldots, \not L_n
\]

Where \( L_i \) is a literal and \( k \geq 0 \) and \( m \geq k \) and \( n \geq m \), or when \( k=0 L_0 = \cdot \).

- A rule is ground if all the literals in the rule are ground.
- The left-hand side of the rule is called the head and the right is called the body of the rule.
- A rule with an empty body and a single item in the head is called a fact.
- When \( k=0 \) and \( L_0 = \cdot \) this is a constraint on the program.

The information covered above is enough to understand the principle concepts of ASP and to start to create and understand programs in the language. However Baral (2003) covers the semantics and syntax for AnsProlog* in much greater detail, if further information on this topic is required see chapter 1 of Baral (2003).

Below is an example of an AnsProlog* program from Baral (2003):

\[
\begin{align*}
\text{fly}(X) & \leftarrow \text{bird}(X), \not \text{ab}(X) \\
\text{ab}(X) & \leftarrow \text{penguin}(X) \\
\text{bird}(X) & \leftarrow \text{penguin}(X) \\
\text{bird}(\text{tweety}) & \leftarrow
\end{align*}
\]
From this program it is obvious that *tweety* can fly but *skippy* cannot. AnsProlog* has many subclasses, these subclasses are generated by imposing restrictions on the rules of the program. One example is AnsProlog$^-$ it is a subclass where the set of rules are restricted so that $L_i$ are atoms and either $k=0$ or $L_0 = \neg$.

**Benefits and Applications**

Due to its implementation AnsProlog*, and its subclasses, have many benefits. It is fully declarative so a program created in AnsProlog* can act as both the specification and the implementation of the program. In comparison to many representations AnsProlog* provides compact representation; i.e. in other representations the problem would take up a much larger space (Baral 2003). AnsProlog* has already been used for a wide number of applications including database querying, planning, scheduling, supply chain management and in solving combinatorial auctions (Baral 2003).

### 2.3.5 Extensions of ASP

**Smodels**

Smodels is an Answer Set Solver, which means it can be used to generate answer sets from a given logic program. Smodels implements the stable model semantics for normal logic programs, which has been extended to deal with ASP. Smodels also claims to have basic semantics that are easily explained to non-experts (Niemela & Simons 1997). Many other languages, such as OCLP, already use Smodels as its answer set solver (De Vos 2004).

**Ordered Choice Logic Programming (OCLP)**

OCLP is an extension of ASP designed to reason about preferences and decisions (De Vos et al. 2005). The language uses choice rules to reason about appropriate decisions and the order is used to represent the agent’s preferences (De Vos & Vermeir 2004). OCLP works by mapping its own rule system into AnsProlog$^-$ rules; see Section 3.4 for more detail on AnsProlog$^-$.  

**OCT**

OCT is a front-end compiler for Smodels, developed at the University of Bath (Brain & De Vos 2003). OCT takes an OCLP and then converts this into a program which can then be read by Smodels which produces the answer sets for the original OCLP program.
DLV

DLV, like Smoodels, is an answer set solver. DLV is based on disjunctive logic and provides a backend for many programs that require knowledge reasoning [Eiter et al. 1998]. According to [Eiter et al. 1998], DLV is a more powerful system for knowledge representation than Smoodels and most problems can be represented in a simpler and more natural way. One of the main uses of ASP is in representing the knowledge and reasoning capabilities of agents. We will look at agents and multi-agent systems in greater detail in the next section.

2.4 Agents

There are no specific definitions of what an agent is or does, only more general definitions of what constitutes an agent. An agent consists of a set of internal states, a set of possible actions it can perform on its environment, a current perception of its environment and a function which decides which action to perform given the current state and the desired state of the agent. Wooldridge & Jennings (1994) outline some other properties that agents must exhibit:

- Autonomy: agents have control over their own actions and do not require human, or any other, intervention.
- Social Ability: agents can communicate with each other through some form of agent-communication language.
- Reactivity: agents must be able to monitor their environment and be able to respond to changes that occur in it.
- Pro-Activeness: agents are able to exhibit intuitive behaviour, through goal-directed actions, and not just react to their environment.

From the properties above it is easy to deduce that each agent must exist within an Environment, of which it is at least partially aware, and have a set of goals which it must achieve.

2.4.1 Environments

An environment is a domain in which an agent exists and which it is at least partially aware. Environment properties can be classified as in Russell & P (2003):

- Accessible v Inaccessible: An accessible environment is one where the agent can obtain complete information about its state. Most real world environments, such as the Internet, are inaccessible.
• **Deterministic v Non-Deterministic**: A deterministic environment is one in which the same action performed by an agent will be guaranteed to have the same effect.

• **Static v Dynamic**: In a static environment the environment can only be changed by actions performed by the agent. However in a dynamic environment there are other processes running, that can change the state of the environment, which are beyond the control of the agent.

• **Discrete v Continuous**: An environment is classed as discrete if there is a finite number of percepts and actions performed in it.

Most environments in the real world are inaccessible, non-deterministic, dynamic and continuous, which makes agents more complex to program.

### 2.4.2 Actions

An action is a process performed by the agent that alters its environment and potentially changes its perception of that environment. An agent possesses a set of actions that are used to help the agent achieve its goal [Corradi (2004)].

The way agents determine which action to perform, like the definition of agents themselves, is very general but [Kowalski & Sadri (1999)] have tried to characterise it in simplified terms:

**Cycle at time T:**

- Observe any inputs and changes to the environment at time T
- Think
- Select Actions
- Perform Actions
- Cycle at time T + n

Where n is the amount of time taken for a single run of the cycle, it can also vary between cycles.

### 2.4.3 Goals

A goal is particular state which is desirable to the agent and which the agent must always try to achieve, for example the goal of a car journey is to reach your destination [Russell & (2003)]. The agent can combine these states with the information of the results of possible actions to determine which of its set of actions can help achieve its goal.

As well as a set of states that would be desirable for the agent to be in, it may also have a set of states it should try to avoid. These are its constraints.

The set of overall goals for an agent can be comprised of a number of states that it must try to achieve and those that it must try to avoid.
2.4.4 Multi-Agent Systems

A multi-agent system (MAS) consists of a set of two or more agents connected by a unidirectional communication channel (De Vos & Vermeir 2001). The agents all act within the same environment but can have a different sphere of influence within that environment, i.e. they are able to influence different parts of the environment. These spheres of influence can coincide and can create dependency relationships between agents. For example, if there are two robots that both can move through a door but because of their size they cannot do this simultaneously (Wooldridge 2002), this is a dependency relationship. There may also be other relationships that link agents, such as hierarchies (Van Nieuwenborgh et al. 2006).

One of the difficulties with the implementation of this type of system is how the individual agents reason and communicate their knowledge and beliefs. For this project each agent contains a logic program representing their personal knowledge and reasoning skills.

2.4.5 Agent Communication

The agents in a MAS communicate by passing information to each other in a predefined way. An agent can choose to accept this information or not, depending on whether it conflicts with its own beliefs and goals. If it does not conflict then the agent will add the information to its knowledge and if it does conflict it will be rejected. The information an agent has is usually filtered before it is shared with the other agents in the MAS (De Vos et al. 2005).

One of the problems associated with communication in a multi-agent framework is how the agents describe the domain, if two agents have different descriptions of the domain it will lead to confusion when communicating. To communicate effectively all agents within a MAS must agree on the same terminology for the domain, this is what an ontology is used for (Wooldridge 2002). An ontology is a formal definition of a body of knowledge and it defines what items exist but it does not determine their properties or interrelationships (Russell & P 2003).

2.4.6 Trust

In a standard MAS all information received by an agent is treated equally with no regard given to the agent that sent it. However this is not always a good thing. Agents may not want to do this as some agents it communicates with may have different levels of knowledge, conflict may arise between information received from two different agents and possibly some agents may want to mislead you. In a trust based MAS each agent will assign a trust based value to every other agent that it can communicate with, these truth values can be updated depending on interactions with the agents. The way agents generate these values varies and has been covered in detail by Castelfranchi & R (1998).

Agents can now base their decision on the level of trust it has in an agent, so information from a more trusted agent will be preferred over that from a less trusted one. Agents may
CHAPTER 2. LITERATURE REVIEW

also choose to accept information from a highly trusted source over its own information, for example a student accepting a teacher’s knowledge is greater than their own (De Vos et al. 2006).

2.4.7 Institutions

Institutions are a mechanism for structuring agent interactions, they can constrain what agents are able to say and what actions they can perform. An institution is captured by its norms, these norms are a set of rules that help to guide and regulate the agents behaviours (Cliffe et al. 2006). These norms can range from the abstract to rules and even protocols. Institutions can enforce obligations upon agents, actions that they must perform in a certain time, or set Permissions so that only certain agents can perform specified actions. If an agent does not meet these obligations then the agent’s action throws a violation and the institution will then sanction that agent.

In order for the institution to observe these event and impose sanctions on the agents the following concepts must be defined, as in the paper by Cliffe et al. (2006):

- Observable Events - Capture physical world events.
- Institutional Events - Events generated by society that only have meaning in the context of that society.
- Institutional State - Evolves over time and is dependent on events generated in the institution, this state can be regarded as a set of Institutional Fluents that may be true at some time.
- Institutional Fluents - Separated into two types: Domain Fluents, which are dependent on the institution, and Normative Fluents.
- Normative Fluents - Are common to all institutions and are separated into three categorise:
  - Institutional Power - This allows the institution to empower an agent to perform certain actions that could bring about changes to fluents in the institutions state. Without this power an agent performing the same action would not have the same effect.
  - Permissions - Is a property given to each action so that it can be performed by agents without introducing a violation.
  - Obligations - States that an event must occur before a certain time or a special violation will occur.

An institution therefore observes the state of the whole Multi-Agent System and monitors the action of all agents, imposing obligations and sanctions on agents to ensure effective interactions between them.

Now that we understand what agents are and how they interact, we will look at tools that are currently available for implementing multi-agent system.
2.5 Tools

There are many tools available to assist in the creation of Multi-Agent Systems, from infrastructures to ontology generators. Below we will focus on two of the more well know tools Jade, which provides a MAS infrastructure, and Protégé which is an ontology generator. We will also look at some of the alternatives and the reasons they will not be used.

2.5.1 JADE

The Java Agent Development framework (JADE) is a framework designed to aid with the development of Multi-Agent Systems, and it is written completely in Java. JADE is an implementation of the FIPA 2000 abstract architecture (JADE 2006). The FIPA abstract architecture specifies, at a high level, the services required by the agents in a multi-agent systems. For a full set of the FIPA abstract architecture specification see FIPA (2000). The architecture outlines three main services that any implementation of the architecture should provide for their agents:

- Directory Facilitator (DF) – This maintains a list of all agents that can provide a service to other agents and what services they provide. When an agent starts it can register itself and any services it is willing to provide with the DF. Any agent looking to acquire services can then interrogate the DF to obtain the name of the agent/agents that can provide the services it needs.
- Agent Management System (AMS) – This maintains a list of the names of all agents that have registered on the agent platform, all agents must register themselves upon start-up. The names of the agents must be unique as they are used for passing messages between agents.
- Agent Communication Channel (ACC) – This is responsible for managing communication between agents on its platform as well as passing messages between different agent platforms. This must implement the Internet Inter-Orb Protocol (IIOP).

As well as implementing the standard FIPA services JADE also provides some other services that are useful to the user:

- Remote Management Agent (RMA) – This provides a GUI which allows the user to interact with the platforms AMS and can then monitor and make changes to the systems state through this agent.
- Dummy Agent – This agent is designed to aid the user with testing of agents, as it allows the user to send arbitrary messages to other agents.
- Sniffer Agent – This agent is also used to aid in testing but, unlike the dummy agent, it does not allow you to send message but instead observes the interactions between other agents.
JADE’s main aim is to make programming a MAS much simpler, its goal is to automate the standard tasks to allow the programmer to concentrate on the specifics of the agent and not to focus on the architecture (JADE 2006a). JADE is also able to handle multiple platforms running at the same time, these platforms can be on the same or different machines. Also as JADE is written in Java it is Operating System independent, so platforms could be run on any machine with Java’s Virtual Machine installed.

However JADE does have disadvantages which largely lie in the implementation and robustness of the messaging system, but this does not concern our project so will not be covered. For a description of these problems see (Corradi 2004). One of the problems with JADE that may concern us is its use of Java. JADE is implemented completely in Java so any agents must also be written in Java or in a language that can be easily translated or has an interface with Java.

### 2.5.2 Protége

Protége is a suite of tools used to construct ontologies for agent communication in Multi-Agent Systems. Protége implements a set of knowledge-modelling structures that aid in the creation, visualisation, and manipulation of ontologies in numerous representation formats (Homepage 2006).

There are many plug-ins available to enhance the functionality of Protége. One of the more useful plug-ins is the Jade Ontology Bean Generator. This bean generator creates java files for use with JADE from the ontology defined in Protége, these java files can then be imported into any agent for direct use in JADE (JADE 2006b).

### 2.5.3 Alternatives

Aside from JADE and the FIPA standards there are other implementation and standards available. Other MAS framework tools exist such as MadAgents (Soares et al. 2006) and AAP (SourceForge.net 2004) which also implement the FIPA abstract architecture but neither have had the success of JADE to date.

As well as other frameworks there are also alternative standards to FIPA, such as KSE (Patil et al. 1992) and OMG (Milojici et al. 1998) but the frameworks that support them have not been as successful as JADE, leaving FIPA as the main standard. The dominance of the FIPA standard and the availability and widespread use of JADE make them the best choice for any project that hopes to develop a multi-agent system.
Chapter 3

Requirements Specification

3.1 Introduction

This chapter tries to identify the key requirements of the system that must be met in order for the system to meet its original aims and be deemed successful, and how these requirements are obtained.

The main focus of this project was to develop a Multi-Agent System using Answer Set Programming. However as this software is mainly used to prototype theoretical ideas the traditional means of requirements analysis, in a software engineering context, are not really relevant to this system. Many of the requirements listed below are a justification of the choice made in implementation and is not a formal set of requirements. Some of the requirements arise from the choice of software used and others from the projects initial brief. The requirements will be split into three sections: Domain Requirements, Functional Requirements and Non-Functional Requirements. Finally there will be a technical specification and a summary of the implementable requirements.

3.2 Domain Requirements

The system must be able to perform the following functions:

1. It must be able to work out an evolutionary fixed-point.

2. The institution must be able to monitor the agents action and update them on its state.

3. Agents must be able to communicate effectively with each using JADE and an ontology.
CHAPTER 3. REQUIREMENTS SPECIFICATION

3.3 Functional Requirements

The system will have to be split into three distinct agents that have very different roles but together make up the system: the Institutional Agent, the Logic Agents and the Monitor Agent.
In order for the system to work correctly it requires the following agents to be set in a JADE container: one Monitor Agent, if needed an Institutional Agent, set with any institution required, and a number of Logic Agents which will act together to solve an ASP problem. Users will have control over what ASP program the Logic Agents use, and the solvers used to compute the answer set. The inputs and outputs of each type of agent are used to determine what is required in the ontology, see 3.4.5.

3.3.1 User Inputs

The system should allow the user to specify the following information to the agents in the system:

1. Specify the number of Logic Agents in the system to the Monitor Agent.
2. Specify whether an Institutional Agent is needed.
3. Specify the location of the Institutional Agents program
4. Add any Logic Agents into the system.
5. Specify the location of the ASP program that each individual Logic Agent will use.
6. Specify the the ASP solver that each individual Logic Agent will use, from a list of predefined solvers.
7. Specify the atoms that each individual Logic Agent can pass on to other agents, i.e. a filter list.

The system should be able to run given the above inputs without any further actions of the part of the user, so all message are sent and protocols adhered to without the knowledge of the user. Users should only be forced to step in if there is a problem.

3.3.2 Monitor Agent

The Monitor Agent is responsible for keeping track of all the logic agents within the system. It sends out a list of agents in the system to all new agents that join and will update old agents when a new agent has joined the system. It also tell the Institutional Agent to begin once a set number of agents has joined the system, if an institution is not used then the Monitor Agent starts the system. The number of agents is set by the initiator of the
CHAPTER 3. REQUIREMENTS SPECIFICATION

monitor agent.
The Monitor Agent must:

1. Be able to monitor the number of agents in the system and when it reaches the set number, specified by the user, it tells the Institutional Agent to start the process. Or it starts a random agent itself if no Institutional Agent is used.

2. Upon receiving an “Agent List” request message it must respond to that agent with a list of all agents in the system.

3. Upon receiving a “New Agent” message, it must update its internal list of agents and send the new agents details to all other agents.

Required Inputs

The Monitor Agent must accept the following inputs:

• An “Agent List” request message from a Logic Agent
• A “New Agent” message from a Logic Agent
• An integer number, which is the total number of agents in the system

Required Outputs

The Monitor Agent must be able to send the following messages:

• An “Agent List” to a requesting agent
• A “New Agent” message to all Logic Agents but the new agent
• A “Start” message to the Institutional Agent, and an agent list.
• A “Start” message to a random Logic Agent.

3.3.3 Institutional Agent

The Institutional Agent will produce an institutional state, based on the current environment, and will send this state to all agents to aid them in selecting the correct action. The agents will then send the chosen action to the institution which then updates its state and continues the cycle. The Institutional Agent begins sending state messages to all agents in the system once it has been told to begin by the Monitor Agent.

The agent will also pick a random agent from the list of all agents to be the first agent in
CHAPTER 3. REQUIREMENTS SPECIFICATION

the cycle.
The Institutional Agent must:

1. Start computing its state when it receives a “Start” message from the Monitor.

2. Send its state to all agents in the system, the agent list will be sent by the monitor along with the start message.

3. Choose a random agent from the list of all agents and send them a message telling them to begin the process.

4. Be able to update its internal state based on the actions received from all other agents and send the newly updated state back to all agents.

Required Inputs

The Institutional Agent must be able to receive the following inputs:

- A “start” message from the Monitor Agent, and an agent list
- The current action of any Logic Agent

Required Outputs

The Institutional Agent must be able to send the following messages:

- A “start agent” message to a random Logic Agent
- An updated Institutional State message to all Logic Agents

3.3.4 Logic Agent

The Logic Agent will perform the majority of the functionality of the system. Each logic agent has an ASP program, which together with the input it receives, is used to generate an answer set. The Logic Agent will also receive information from the Institutional Agent on the institutions state, this will also affect its answer set. This answer set is then filtered, based on the atoms that an agent can share with others, this is defined by the implementer of the agent. The answer set is then sent to the next agent in the chain. It also sends its chosen action to the Institutional Agent so that it can update its state. Agents must also deal with information from other agents that may contradict their own knowledge, it is up to the implementer of the particular agent whether the agents information is believed more strongly than the information received from other agents or vice versa. The agent who is chosen by the Institutional Agent to start the process will run its program with no input from other agents. The agent must also recognise when they have reached a fixed point or
when one is no longer possible. When a fixed is reached for the whole system the agents will stop running.

All Logic Agents must:

1. Inform the Monitor Agent when it starts up and then request a list of other Logic Agents.
2. Work out which agent is the next one in the chain, using the agent list received from the Monitor Agent, and keep this list updated every time it receives a new agent message.
3. Be able to use the Institutional State received from the Institutional Agent to help compute its answer set, and then return any chosen action back to the Institutional Agent.
4. Be able to compute answer sets using the given solver and ASP program.
5. Be able to combine the answer set received from the previous agent with its own knowledge.
6. Be able to come to a conclusion when a contradiction arises between its own information and that received from another agent. This conclusion is based on whether it believes its own information more than the other agents.
7. Be able to filter an answer set so that it only passes on specified atoms to other agents.
8. Be able to pass on its answer set, along with the input it used, its AID and whether it believes it has achieved a fixed-point to the next agent.
9. Be able to compute whether it has achieved a fixed point or not, based on its previous inputs and answer sets.
10. Be able to compute whether all agents now believe they have achieved fixed-points and if they have issue a message to all agents in the system confirming that there is now a fixed-point.
11. Be able to determine if a fixed-point is not achievable and inform all other agents of this, stopping the process.
12. Print out the fixed-point that it has achieved to be seen by the user.

Required Inputs

All Logic Agents must be able to receive the following inputs:

- New Agent message received from the Monitor Agent
• An agent list from the Monitor Agent
• A start message from the Institutional Agent
• The Institutional State from the Institutional Agent
• An answer set from another Logic Agent
• A fixed-point message from another Logic Agent

**Required Outputs**

All Logic Agents must be able to send the following messages:

• An answer set to the next Logic Agent in the chain
• A fixed-point message to other Logic Agents
• The action it performed to the Institutional Agent
• Print a final fixed point answer set to the user
• Send a new agent message to the Monitor Agent
• Send a request for Agent List to the Monitor Agent

### 3.4 Non-Function Requirements

#### 3.4.1 Implementation

As stated in the Literature Review the JADE framework will be used to develop the system. JADE takes care of most of the low level implementation of a MAS, as it handles all messages and coordinates all processes, this allows the agents to be programmed at a higher level by defining behaviours and reactions.

Also Protégé, with JADE bean generator, will be used to create an ontology. The ontology allows the agents to communicate effectively by defining the objects that the agents may pass to each other and how they should be treated. Using the Bean Generator means that the ontology can also be developed at a high level and then converted into Java code and placed into the program. The system will also require several answer set solvers to compute the answer sets of the programs, for this purpose smodels and dlv will be used as they are the most prevalent in this field. Also clasp will be used as an exercise to ensure its interoperability with the JADE framework, along with OCT. As both smodels and clasp require a grounded input lparse will be required to ensure the program can be grounded before being passed to the solvers. The use of these different solvers must be accounted for in the program.

Choosing JADE and Protégé means that Java must be used as a programming language for the agent implementation, the use of JADE and Protégé library’s.
3.4.2 Usability

The system has no user interface other than the command line and JADE’s GUI. It is assumed that all user have a high level of technical expertise and so using a command line interface should not prove to a difficulty. Because of this assumption usability will not be regarded as a high priority for this project.

3.4.3 Robustness and Reliability

As this system is only a prototype the system need not have the high level of robustness required by a complete program, however the system should be able to run without crashing for standard errors. For standard errors some attempt will be made at recovery, but for most errors an error message will be printed to the command line. With the level of expertise expect from the systems users it is reasonable to assume that the users will have the ability to correct errors in the software. Errors could just as easily be caused by problems with the ASP code as they can from the Java code, so a method of dealing with errors in ASP will be required.

3.4.4 Interoperability

Each Logic Agent must be able to interact with all other Logic Agents, the Institutional Agent and the Monitor Agent. In order to compute answer sets the Logic Agents will have to interact with different ASP solvers, either lparse and clasp/smodels, oct or dlv. In order to interact with these programs correctly the Logic Agents must:

- Produce files that are readable ASP programs
- Parse and standardise ASP outputs so that agents are unaware of the different solvers being used.
- Relay error messages from the solvers to the users.

The program will automatically run the solvers and create any files it needs from the given inputs so users will not need to interact with this part of the process. Agents will be able to operate in different containers providing they all contact the same monitor agent, this will not be supported by the system but could easily be achieved by making some alterations to the code of the Logic Agent.

3.4.5 Ontology

An ontology must be create which will define the main objects used by the agents in communications. As already stated the ontology will be developed using Protégé and the JADE Ontology Bean Generator. The ontology must include a class for all of the inputs
and outputs specified for the Monitor Agent, see 3.3.2 and 3.3.2, the Institutional Agent, see 3.3.3 and 3.3.3, and for the Logic Agent, see 3.3.4 and 3.3.4. The classes in the ontology must be defined so that the agents have a means of:

- Passing answer set information between agents. This must include: the agents input, the agents output, the agents AID, whether it was a fixed point for that agent and all of the previous results for this message, i.e. a history list. Both the input and output of the agents will be answer sets.
- Passing a list of agents between the monitor and the other agents in the system.
- Informing all agents that a potential fixed point has been reached, and a response to this fixed point.

The ontology will also define several other classes to allow communication between Monitor Agent, the Institutional Agent and the Logic Agent, but these will only be simple and possibly even empty classes just used to signal event, such as starting the system.

### 3.4.6 Portability and Standards

The agents should be able to run on any machine capable of running JADE and either clasp, smodels, oct or dlv. At present OCT is not supported in MS Windows and Clasp is not supported in UNIX. These constraints will be outlined in the Technical Specification. The agents and ontology must adhere to the standards set out by FIPA for communication.

### 3.5 Technical Specifications

- The system shall be developed using JADE and will adhere to the FIPA specifications.
- The system will be developed using JAVA 1.5, so that it is compatible with the JADE framework and Protégé Bean Generator.
- Smodels, Clasp, OCT and DLV will be used as answer set solvers.
- Both smodels and clasp require the ASP program to be grounded before they can run, lparse will be used to ground the program.
- An ontology is required to represent Answer Sets and message structures.
- The ontology will be developed using Protégé and a suitable content language.
CHAPTER 3. REQUIREMENTS SPECIFICATION

3.6 Assumptions

This system is to be used by people who are already familiar with ASP syntax and have a formal background in logic. Technical expertise is also expected to be well above average. This leads to the assumption that users are capable of compiling source code for the more well-known languages, are able to use a command line interface and can modify the Java program and any ASP programs.

The users will be able to deal with errors produced by the program, most errors will effectively kill an agent and so the user must be able to deal with and correct any errors.

3.7 Summary

The set of requirements discussed in detail above can be summarised as the following objectives which the system must meet in order to be deemed successful:

1. The system must be built using JADE and Prot´eg´e and adhere to the FIPA specifications.
2. The system must be written using Java 1.5.
3. The system must support the ASP solvers: Clasp, Smodels, OCT and DLV.
4. The system must allow for the use of an Institutional Agent to help govern agent behaviour.
5. The system must still be able to function without an institution.
6. The system must allow users to specify if they want to use an institution.
7. The system should be able to find the evolutionary fixed-point of a set of ASP programs, if one exists.
8. The system must allow users to specify the number of logic agents in the system.
9. All Logic Agents should reason with their own program within the institutional boundaries, if an institution is used.
10. The Logic Agents program and solver must be specified by the user.
11. Agents should be able to make conclusions on contradicting information.
12. The agent must make contradiction decisions based on its level of trust in itself or the other agent
13. Agents should be able to filter the information they pass to other agents, the filter must be defined by the person implementing the agent.
14. Logic Agents should be able to communicate with each other by passing answer sets.
Chapter 4

Design

4.1 Introduction

The previous chapter discussed what would be required of the system if it is to be a successful product. This chapter examines those requirements in detail and looks at how they are to be implemented as a full system. Possible problems that may arise and their potential solutions will also be discussed. When looking at each agent a list of required responses will be created, this will be used to help generate the behaviours when converting the design into JADE agents. Although the design will be adhered to as much as possible this is a prototype system and so the product can evolve as needed or is felt appropriate by the programmer.

4.2 Structure of the System

The system is meant to be an agent framework, for multi-agent institutional systems. The internals of the agent are only slightly relevant in the development of the system, providing the agents are capable of handling the expected outputs and inputs of an ASP program the complexity of that program is of no concern to the system.

As already stated the system will use the Multi-Agent Framework structure provided by JADE and will meet the FIPA specifications. The agents will need to use a language provided by JADE and an ontology compatible with it. All agents must extend the JADE Agent class and include the necessary JADE and Protégé libraries. The agents will implement the FIPA Request protocol and will be using JADE behaviours to accept messages, the messages will be differentiated depending on the contents of the message.

In order for the system to work correctly it will require an initial Monitor Agent, an Institutional Agent and a set number of Logic Agents. Once all of the agents are loaded into a JADE container the system will run with no further input from the users. See Figure 4.1 for an outline of the how the agents are linked.
The details of the three individual agents are discussed in Sections 4.3 to 4.6.

Figure 4.1: System Structure

4.3 Monitor Agent

From the requirements the Monitor agent must keep track of all agents in the system, informing them of new agents and then starting the system once there are the correct number of logic agents.

- The Monitor will contain an array of Logic Agents AIDs and send that array to any agent that requests an agent list.

- The Monitor will add the new agent to the array when it receives information on a new Logic Agent, then send that agents details to all other agents, including the Institutional Agent so that their Agent array matches the monitors. This is then used by the agents in determining which agent is next in the cycle.

- The Monitor will have a counter that increases by one each time it receives a new agent message, when this counter reaches the number of agents specified by the user the monitor will send a starter message to the Institutional Agent.

4.3.1 Responses

The monitor agent must respond to the following messages:

- When any agent requests a list of agents the Monitor must send all agents, including the Institutional Agent, its array of agents.
• When a new agent starts the Monitor must have all other agents, including the Institutional Agent, update their array so that it matches the its array.

4.4 Institutional Agent

The requirements state that the Institutional Agent should be capable of determining its own state, updating its state given an action from a Logic Agent, sending its state to all Logic Agents in the system and randomly choosing a Logic Agent to begin the process.

The design criteria for the Institutional Agent are:

• The agent will use an ASP solver to determine its initial state.
• It will then wait for the monitor to send it a start message and an array of all agents.
• It will send the state to all of the agents in the array.
• It will then generate a random number between 0 and the total number of agents and select the agent at that position in the array to begin the process.
• When it then receives an action from a Logic Agent, it will update its ASP program, recalculate the state and send a message to all agents informing them of the new state.

4.4.1 Responses

The Institutional Agent must respond to the following messages:

• When it receives a start message and agent array from the Monitor it must send its current state to all agents in that array.
• When it receives an action message from a Logic Agent it must update its own ASP program, therefore its state, and send the update to all agents.

4.5 Logic Agent

The Logic Agents are the main agents of the process and therefore have far more to do than the other two agents. From the requirements the Logic Agents must be able to:

• Calculate answer sets given an ASP program, and an input answer set, using several different ASP solvers.
• Capable of reasoning with the state of the institution that it is in, and keeping the institution informed of the actions it takes.
• Send and receive those answer sets in such a way that it does not matter what solver was used.
• Establish an evolutionary fixed point, or terminate if one does not exist.
• Filter the answer sets it sends to other agents and be able to resolve any conflicts that arise between the input it receives and its own beliefs.

These requirements can then be converted into the following design criteria which the Logic Agent has to conform to:

• The agent will run external processes from Java that call the solvers as needed, it will also use a file writer to create a file for the inputs so that the inputs can be included with the agents own program.
• As all solvers output the answer sets in different ways some string manipulation tools will be needed to convert all outputs of the solvers to the same form. Different string tools must be written for each solver required.
• Every time the agent creates a new answer set, it adds the previous one it received to an array of previous answer sets, then adds that, its input, plus the answer sets it created and whether it has achieved a fixed point or not to a message. This message will be sent to the next agent in the chain who will perform the same actions.
• When the agent receives the initial agent list from the Monitor Agent it will replace its list with that and then find itself in the list, the next agent in the list will be the agent that it must send all answer sets to. If it is the last agent in the list then it must loop round to the first agent in the list.
• When it receives a new agent update from the Monitor the agent will update the agent list it has and then check to see if that affects the next agent that it sends too.
• Institutional State will be stored as a string by the agent and then added to the input file so that the agent can then reason with the data as well as its own program and an input.
• Although the agents in this system do not determine the actions they will take, the Institutional Agent needs to know the actions taken in order to update its state, for these purposes dummy methods will be used that can easily be adapted at a later date.

4.5.1 Responses

All Logic Agents must respond to the following messages:

• Obviously the agent will need to receive Answer Sets from other agents, this information will then be used to generate a new set of Answer Sets and will be checked
to see if there is a fixed point before the new answer sets are then sent to the next agent.

- The agent should receive an Institutional Data message from the Institutional Agent and be able to update its current copy of the institutions state with the new copy received.

- The agent should receive an array of all the other agents from the Monitor Agent and use this array to work out which agent is next in the cycle, it will be the agent that follows them in the array, or the first agent if they are the last.

- When a New Agent message is received the agent must update its array so that it matches the Monitors array and recalculate who the next agent in the chain is, this will only apply to agents that were last in the array.

- The agent should also receive Fixed Point messages from other Logic Agents that specify whether a fixed point has been achieved or is no longer achievable. If the fixed point is no longer achievable then the agent will stop running, if it is then it will print its fixed point to screen.

- When a start message is received from the Monitor or Institutional Agent, the agent will begin computing its answer sets with an empty input.

Because of the complex nature of the agent, some of the more difficult parts of the agent will be discussed in more detail.

### 4.5.2 Fixed Points

An evolutionary fixed point is achieved when all Logic Agents in the system get the same output from an ASP solver given the same input, i.e. the agent is receiving the same input from the previous agent and sending the same output to the next agent as it did on the last cycle.

In order to establish whether a fixed point has been found or not, the answer sets that are sent between agents will need to store the agents input, the output and their Agent Identifier (AID) in an array, that can be accessed by all agents. This array can then be searched by an agent to see whether they achieved the same output for the current input, if the current and the previous inputs and outputs match then the agent has a possible fixed point, there will also need to be a boolean in the array that states whether the agents thinks that it has previous answer sets were a fixed point. When an agent thinks it has found a fixed point it will scan back through the array, until it finds itself again and check whether all agents believe they have found fixed points. If all the agents have found a fixed point then a evolutionary fixed point for the system has been found and the last agent will send a message to the others telling them a fixed point has been found and all agents will print their fixed points.

However it is not guaranteed that a fixed point will exist so potentially the agents could
continue forever without finding a fixed point. There will need to be a mechanism to prevent this. The system will have two ways to prevent the system cycling indefinitely: the simplest way is to keep a count of the number of times the message will travel through the cycle of agents and end on a predefined number of cycles, this can be specified by the user of the first Logic Agent. If an agent finds a fixed point but the next agent does not then there will obviously be no overall fixed point for the system, if the agent sent the same information out but the agent did not get the same input then clearly there is a conflict and so no fixed point will ever be found. Therefore the agent should terminate the cycle if the previous agent achieved a fixed point but he did not.

4.5.3 Contradiction and Trust

One of the more complicated processes involved in computing the answer set of a particular agent is dealing with conflict between your program and the input being received from another agent. This type of conflict occurs when an agent gets a value from its program that is the opposite to that suggested by another agent. When this problem arises the agent has two choices: trust its own knowledge or believe the other agent. For the purpose of this project there will be the option of both allowing the person implementing the agent to specify whether they wish the agent to believe itself or the other agent.

One of the main problems with contradictions is in determining whether one exists or not, this cannot be done simply by comparing answer sets, as only truth values are shown. For this the agent must determine the atoms at the head of all rules in its program and then using the current input minus the suspected contradiction to work out the potential answer sets. If the potential contradiction is not present in any of the outputted answer sets then there is clearly a contradiction. As if the agent has the atom as the head of a rule but it is not in the answer set then it is not true, but the input says that it is true, so it is up to the agent to decide who to believe.

Contradictions do not apply to OCLP programs, as the input is automatically given a lower priority than the programs components the agent will prefer to believe its data over that of its input.

4.5.4 Filtering

There are situations in which an agent may not wish to pass all of its information on to another agent, it may wish to keep some things private. In order to do this the agent must filter its answer sets so that it can remove anything it wishes to keep private. There are two principle ways this can be done:

1. An agent has a set of atoms that can be communicated to all other agents. The agent calculates its answer sets and then it “filters” the answer set removing any that are not in the set of atoms and then passes the output to the other agents.

2. Also the agent can have two languages, an internal and an output language. The
agent is only allowed to pass on atoms that are in its output language.

For this system the implementer of the agent will specify a set of atoms that the agent can pass on to other agents, all other atoms will be kept private.

4.6 The Ontology

An important part of the system will be the Ontology, which allows the agents to communicate successfully. As already stated the ontology will be developed using Protégé and the JADE Ontology Bean Generator, which will convert the high level ontology design into Java code. The ontology will be defined using the required inputs and outputs of the agents from the requirements section, see Section 3.3 and the Responses sections above. As a template for this ontology the \texttt{SimpleJADEAbstractOntology} will be used, which is an abstract “starting point” for any ontology to be used with JADE. The ontology will need the following classes and slots to allow full communication between all three types of agents:

- An \texttt{Answer Set} class - will contain a string for the Answer Set, one for the input, a copy of the senders AID, whether its a fixed point so far or not, a integer for the total number of cycles this class has gone through, and an array of previous answer sets

- A \texttt{New Agent} class - which will just store the new agents AID

- A class for \texttt{Fixed Points} - which will have a boolean for whether a fixed point has been found

- An empty class that can be sent to start the logic agents

- An \texttt{Agent List} class - that holds an array of AIDs, which will be a list of all agents

- An \texttt{Institutional Data} class - Which will store a string that is the state of the institution.

- An \texttt{Agent Action} class - that store a string representation of the agents action that can be interpreted by the institution

These will be the only pieces of information that the agents will need to pass between each other, this is all that will be required from an ontology.
Chapter 5

Implementation

5.1 Introduction

Using the requirement specifications and the design ideas discussed in previous chapters, this chapter will look at the implementation of the full system. It will detail how the system deviated from the design as it was implemented and it will show the overall class structure of the system. The chapter will also cover the more complex parts of the system in greater depth and how the they have been incorporated into the classes.

5.2 Programming Languages and Tools Used

The program was written entirely in Java 1.5 Standard Edition using the Eclipse IDE (version 3.2), with Protégé being used to generate the Java ontology code. Along with Java, OCT, Smodels, Lparse, Clasp and DLV are used to handle the answer set programs. Also JADE (version 3.3) was used as the Multi-Agent environment.

5.3 Agent Structure

Because of the use of the JADE Framework all of the Agents in the system are extended from the JADE Agent class so that they will run successfully within the framework. Therefore all agents require a setup method that is called when the agent is started, they will also require a set of behaviours to allow them to respond to different events. The setup method for all agents defines its language and ontology, adds the agents behaviour listener class, deals with its input arguments and then registers the agent with the DF. The setup method also has other functions which are unique to the agents and will be covered in more detail for each agent.

In order to create the behaviours effectively a class will be created inside each agent that
extends the SimpleREResponderBehaviour class of JADE, this class requires two methods: prepareResultsNotification, which for all classes will simply return null as everything is handled by the next method, and prepareResponse which will handle all of the incoming messages. SimpleREResponderBehaviour class accepts all FIPA.Request messages with the current agent as the receiver, the message is then passed to prepareResponse which removes the action from the message and then depending on the class of the action will perform a different method. For example if the message sent contains an action that is an AgentList class the agent will update its array, if its a NewAgent class then the agent will add that to its array. All agents will have an IDontUnderstand class which is an extension of the Java exception class is used when dealing with exceptions.

All agents communicate by passing ACL messages so require an ontology and a codec, when the agent is run it sets a global variable for both. The language is obtained by creating a new SLCodec and the ontology by getting an instance of the ontology created using Protégé, see Section 5.8 they are both set while the setup method is being run. To use all of the JADE classes they must be imported into each agents program, along with all of the classes of the ontology. The agents also all have access to a set of helper classes which are defined in Section 5.7.

As well as containing the functionality above each agent will have additional functions called by their setup and prepareReponse methods, this along with their global variables and methods are detailed in the next few sections.

5.4 Monitor Agent

Once the Monitor Agent is created it waits for the Logic Agents to register themselves with it, sending out Agent List (sendAgentList) to all new agents and an update (informAgents) to all other agents when each new agent joins the system. Once the number of agents reaches the number specified by the user the Monitor either starts the agents, (startAgents), or the institution (startInstitution) depending on whether an institution has been specified or not.

5.4.1 Global Variables

The Monitor Agent has these global variable:

- ArrayList(AID) AgentList - Is an up-to-date list of all Logic Agents that have currently registered with the Monitor.
- int noAgents - Is the number of Logic Agents, specified by the user, that will be in the system.
- AID institution - is the AID of the systems Institutional Agent.
5.4.2 Methods

Setup

The setup method of the Monitor Agent only performs the standard tasks listed in Section 5.3 and has no other functionality.

Arguments

1. An integer that defines the number of agents in the system.
2. A boolean that states whether an institution will be used with the system or not.

prepareResponse

The Monitor Agent responds with the following actions to messages who’s action is of the type:

- **NewAgent** - When the monitor receives a NewAgent class it adds the associated agent to its AgentList and calls informAgents.
- **AgentArray** - After receiving an empty AgentArray from an agent the monitor calls sendAgentList.

sendAgentList

This method sends the Monitor’s AgentList array to the specified agent.

Parameters: AID sendTo - The agent the list is being sent to.
Return: NONE
Method: Create an ACL Message, with the receiver set as sendTo, and with the action as an AgentArray set to the monitor current AgentList.

informAgents

This method informs all agents that a new agent has been added to the system.

Parameters: NewAgent newAgent
Return: NONE
Method: For all agents in the system, except the agent that sent the NewAgent message, send them a message with the newAgent action.

startAgents

The monitor randomly picks one of the agents from its AgentList and sends them a start message, this message is only used if there is no institution present.
CHAPTER 5. IMPLEMENTATION

Parameters: NONE
Return: NONE
Method: The monitor generates a random number between zero and the size of the AgentList minus one, it then picks this agent from the array and sends them a message with the Start class as its action.

startInstitution

The monitor tells the institution it can send its information to the agents and start the process.
Parameters NONE
Return: NONE
Method: The monitor sends an AgentList action in a message to the Institutional Agent, as this is the only message it sends to the institution this will also start the process.

5.5 Institutional Agent

Once the institution has been setup, it waits until it receives a message from the monitor with the AgentList. After receiving this message it works out its current state and sends it to all the agents in the agent list (sendInstInfo), it will then start the agents (startAgents). The agents will send their actions back to the Institutional Agent, who will update its program and consequently get a new state. The institution then sends this updated state to the agents who in turn send the actions they perform back to the agent. This cycle repeats until the system stops.

5.5.1 Global Variables

The Institutional Agent has these global variable:

- ArrayList(AID) AgentList - Is an up-to-date list of all logic agents that have currently registered with the Monitor.
- String institutionURL - Stores the location of the institution's program
- FileWriterTools and StringTools helper classes are also stored as global variables.

5.5.2 Methods

Setup

As with the Monitor agent the setup method only deals with the tasks listed in Section 5.3.
CHAPTER 5. IMPLEMENTATION

Arguments

1. A string which is the URL of the institutions ASP program.

prepareResponse

The Institutional Agent responds with the following actions to the messages:

- AgentArray - Updates its own AgentList with the list received from the Monitor and then calls sendInstInfo followed by startAgents.
- AgentsAction - Adds the action to its program using the FileWriteTools method updateIns, see Section 5.7.1 then calls sendInstInfo.

startAgents

This method is the same as that in the Monitor Agent, see Section 5.4.2.

sendInstInfo

This method calculates the institutional state and then sends this state to all agents in the AgentArray.
Parameters: NONE
Return: NONE
Method: The Institutional Agent calculates its state and stores this in an InstitutionalInfo class which is then added to an ACL message and sent to all agents in the AgentList.

5.6 Logic Agent

After the agent has been loaded into jade and has run its setup method, it waits until it either receives a Start message from the monitor/institution or an answer set from another logic agent. It then calculates its answer sets using its program and the answer sets of the previous agent as it input (calculateFixedPoint), if it receives a start message then it calculates the answer sets with no input (calculateAnswerSets). Every time it calculates its answer sets it checks whether the inputs and answer sets match the ones its calculated last time to see if a fixed point has been achieved. If a fixed point has been achieved or cannot be achieved then it will inform all other agents (sendFixedPoint), otherwise it will then send these answer sets to the next agent in the list and makes an action choice which it then sends to the institutional agent (sendInstAction). For an overview of the interaction between the Logic Agents and the Institutional Agent see Figure 5.1 and for a dataflow diagram of a typical Logic Agent see Figure 5.2.
5.6.1 Global Variables

The Logic Agent requires the global variables:

- MonitorAgent - AID that stores the name of the Monitor
- InstitutionalAgent - AID that stores the name of the Institution, if one exists, otherwise null
- ASPSolver - String of the name of the solver to be used to solve answer sets for this agent
- ASPProgram - String that stores the URL of the agent's program
- trustMe - a boolean that stores whether the agent should trust its own beliefs or that of the previous agents, true implies it trusts its own beliefs more
- AgentList - an ArrayList(AID) that stores a list of all other Logic Agents in the system
- NextAgent - an int that stores the position of the next agent in the array
- NoCycles - int that sets the agents to stop after the number of cycles reaches this number
• FixedPointAchievable - a boolean that says whether the agent believes a fixed point is still achievable, if this is false then the agent stops

• FixedPoint - An ArrayList(String) that stores the potential fixed points for this agent

• All three helper classes are also instantiated for use in the agent.

5.6.2 Methods

Setup

In addition to performing the functions specified in Section 5.3, the setup method also registers the Logic Agent with the Monitor (sendConfirmMonitor) and requests a list of all other Logic Agents in the system (getOtherAgents). Then it stores a copy of the agents program for use when calculating answer sets (done using FileWriterTools setprog method).

Arguments

1. A string that sets the ASPSolver for the agent

2. A string that sets the ASPPrograms URL

prepareResponse

The Logic Agent should respond to the message shown with the following actions:

• AnswerSet - Calls calculateFixedPoint with the received AnswerSet class

• InstitutionInfo - Sets the institutionAID, if not already set, and then updates its Institution file, by calling FileWriterTools.writeInst and also setting this files location in asp tools.

• AgentArray - Updates its AgentList with that received from the Monitor, it then cycles through the list until it finds itself and then sets the nextAgent variable to the position of the nextAgent.

• NewAgent - adds the AID from this class to its AgentList and then if the current nextAgent equals 0 it sets nextAgent to the last position in the array

• FixedPoint - Calls checkFixedPoint with the action as an argument

• StartAgent - Calls calculateAnswerSets
CHAPTER 5. IMPLEMENTATION

getMonitor

Returns the Monitor Agents AID
Parameters: NONE
Return: AID MonitorAgent
Method: If monitorAgent is null, then request all of the agents in the system whose Description is MONITOR-AGENT from the DF. Take the first item if more than one is returned and then set the global variable to that AID, and return monitorAgent

sendConfirmMonitor

Sends a NewAgent message with the Logic Agents details to the Monitor Agent.
Parameters: NONE
Method: Creates a NewAgent class, using its AID and send this to the Monitor Agent

getOtherAgents

Requests an array of agents from the Monitor Agent.
Parameters: NONE
Return: NONE
Method: Sends an empty AgentArray class to the Monitor agent, requesting an AgentList.

calculateAnswerSets

Works out the agents answer sets, given no input and sends them to the next agents.
Parameters: NONE
Return: NONE
Method: Calls ASPGenerator, from ASP Tools, using the current ASP solver and programming, it then uses stringTools.justAnswerSets to standardise the answer sets and then passes the resulting ArrayList(String) to sendAnswerSets.

sendAnswerSets

Given an array of answer sets it splits them up into single answer sets and sends them to the next agent in the system.
Parameters: ArrayList(String) answerSets
Return: NONE
Method: It splits each item in the array into a separate answer set class. For each class the input is set to null, the Answer Set is the output and the sender to be the sending agent, with fixed point obviously being false. It then sends this class to the next agent in the cycle.
calculateFixedPoint

Upon receiving a new AnswerSet class from an agent, the agent takes the answer set of the previous agent and uses that as an input to its program to obtain a new answer set. Once a new answer set it obtained the agent calculates whether this input and answer set form a fixed point (checkPrevious). If it does then this sets the value of the fixedPoint field in the AnswerSet class to true, otherwise false. It then checks to see, given a fixed point for this agent, whether there is a fixed point for the whole system. If there is the agent informs all other agents, it also checks if a fixed point is still achievable if there is no fixed point it will also send a message to all agents (sendFixedPoint). If an overall fixed point is not found but one could potentially still exist the agent sends its answer sets to the next agent.

Parameters: AnswerSet as
Return: NONE
Method:

1. Given an Answer Set as input calculate new Answer Set for your program.
2. Check if this new input and answer set form a fixed point from your previous round, this is stored in the previous array of the answer set, by calling checkPrevious
3. If they do form a fixed point then check all previous agents to see if they also have fixed points, by calling isFixedPoint: If Yes, call sendFixedPoint, If No, continue.
4. If you do not have a fixed point check to see if the previous agent found one, if they did then there is no fixed point available, so sendFixedPoint(false).
5. Next check to see if the total number of cycles exceeds the number allowed, if so sendFixedPoint(false) otherwise continue.
6. Now split your ArrayList(String) of answer sets into their strings and send them to the next agent.
7. To create the answer set, set Input to be the Answer Set of the previous agent, Answer Set to be your own answer set, and fixed point to be true if you believe you have a fixed point and false otherwise. Then add the previous AnswerSet received to the previous array and send this to the next agent.

8. It is now complete.

checkPrevious

Searchs through the previous array of the AnswerSet class until it finds the last entry of the current agent and checks to see whether its current input and answer set matches the previous one. If it does then it saves the answer set and returns it’s found a fixed point, if not then it simply returns false.

Parameters: AnswerSet as, ArrayList(String) ansSet
Return: boolean, true if its a fixed point, false otherwise
Method: It searches backwards in the array until it finds the agents previous entry, if the agents previous input and answer sets match the current ones it sets the fixed point variable to the current answer set and return true, if they do not match then it return false.

getPrevious

Is a simple helper class that converts the previous array in an AnswerSet class from a JADE ArrayList to a Java ArrayList.

isFixedPoint

The method checks to see if a fixed point is still achievable by checking to see if the previous agent found a fixed point and this agent has not. It then checks to see given that this agent has a fixed point have all previous agents also found a fixed point, if not then it carries on otherwise it sends a message to all agents.
Parameters: AnswerSet as, boolean fixedPointFound
Return: int, 0 if fixed point is not achivable, 1 if one has been found, -1 otherwise
Method: The method checks the previous answer set and if it found a fixedPoint and this one did not then it calls sendFixedPoint(false) as a fixed point is no longer achievable, and returns 0. It then checks to see for all agents in the system also have found fixed point, if they have it calls sendFixedPoint(true), as the system now has an overall fixed point, and return 1. Otherwise it will return -1 and the system continues sending answer sets.

selectAction and sendInstInfo

These methods are both temporary stub methods, and are used to show that communication is possible between the Logic Agents and the Institutional Agent. The methods do not perform any significant tasks and will not be detailed here, but they could be easily updated to perform any tasks required.

checkFixedPoint

Given a fixed point message from another logic agent the agent either prints out its fixed point if one exists, or stops the agent if a fixed point cannot be achieved for the system.
Parameters: FixedPoint fp
Return: NONE
Method: The agent checks the value of the fixed point message, if its true then it prints out its fixed point to the screen and stops the agent with the doDelete method, otherwise it prints a message “No Fixed Point” and exists with doDelete again.
CHAPTER 5. IMPLEMENTATION

sendFixedPoint

Sends a fixed point message to all other agents, depending on whether the agent believes it is a fixed point or that one is not achievable.
Parameters: boolean fpf
Return: NONE
Method: Sends a FixedPoint message to all the agents in the system, with a true or false value depending on what the agent believes the fixed point to be.

errorExit

Stops the agent running and prints out an error message, used to stop the agent if there is an unrecoverable error
Parameters: String message
Return: NONE
Method: The agent is killed by calling the doDelete method, and the given error message is printed to the screen for the user to deal with.

5.7 Helper Classes

The Institutional Agent and the Logic Agents need to read from and write to file, manipulate strings and use answer set solvers. To help make the system more modular and help reduce code duplication several helper classes were written that contained any code reused by more than one agent.

5.7.1 FileWriter Tools

The FileWriterTools provides a suite of tools used to read/write/manipulate files that all of the agents can use.

Global Variables

FileWriterTools has the global variables:

- ArrayList(String) oldProg, stores a copy of the ASP program of the agent that created the class
- int filesCreated, used as a random number to create new files
- String orderList, only used in OCT programs as the preference list
- Also creates an instance of StringTools
Figure 5.2: Logic Agent Dataflow
Methods

**getOCLP**
Converts the OCLP file for use with the OCT program.
**Parameters:** String fileName
**Return:** String, the OCLP program
**Method:** Copies all lines from the file to a string separated by “newlines” for use with the OCT program.

**getOldProg**
Stores a copy of the given ASP program in its oldProg array
**Parameters:** String aspProgram, String aspSolver
**Return:** NONE
**Method:** Copies all lines from the file to an array in the class, if its OCT then it copies the last line to orderedList, for use later.

**updateIns**
Updates the institutional agents program given an action from a logic agent
**Parameters:** String instituionURL, String action
**Return:** NONE
**Method:** Adds the given action to the end of the given file in ASP format.

**writeInst**
Creates a copy of the institutions state as a file for use by the logic agents
**Parameters:** String program, AID agent
**Return:** String, the name of the created file.
**Method:** Given a string received from the Institutional Agent, this method parses the string and then creates a file of the institutions state that can be used by the agent, and passes the file-name back to the agent.

**appendToFile**
Given an input from a previous agent this creates a file from that input and the agents current program that can be solved by an ASP solver to produce answer sets.
**Parameters:** String fileName, String input, String aspSolver
**Return:** String, the name of the file created is passed back to the agent
**Method:** The method takes the given string input, which is all standardised, and then converts them into ASP rules and adds them to the bottom of the agents previous program, this is created in a separate file, the filename is then passed back to the agent. For OCT programs, the write adds the input to the end of the preference list so that its the least preferable component.

**deleteFile**
Deletes the given file
**Parameters:** String fileName
**Return:** NONE
**Method:** deletes the given file, and prints an error message if it cannot be completed successfully.
5.7.2 String Tools

Provides a set of string manipulation tools used by the Institutional and Logic agents.

Global Variables

StringTools has no GlobalVariables

Methods

**splitAnswerSets**
Takes a string of answer sets, which will have line breaks and converts each line into an entry in an array
Parameters: String answerSets
Return: String[], the string split into lines that are added to the array.
Method: Tokenizes the String using the line break and then adds the tokens to the array.

**removeTrue**
Takes a string of answer sets, the previous input and the current input, it then removes all of the items from the current input that it made true previously, to ensure that items that it made true last time, but maybe untrue this time, are not kept in the system.
Parameters: String previousAS, String previousInput, String currentInput
Return: String, the currentInput minus the atoms the agent made true in the last cycle.
Method: All three strings are tokenized using “,” to give just the atoms. All items that were not in the previous input but are in the output, i.e. the atoms the agent made true previously are then removed from the current input. The current input is then reassembled into answer set form, minus a few atoms, and then this string is returned.

**justAnswerSets**
Takes the output from all of the different solvers, which are all drastically different and then converts them into a standardised answer set, of the form atom1, atom2, atom3, ....
Parameters: String aspOutput, String aspSolver
Return: ArrayList(String), all of the answer sets standard form in an array.
Method: Takes the output from the answer set solvers and tokenises the output. Then depending on the solver performs some string manipulation and throws away all of the information that is not need, keeping just the answer sets. It then searches through every item of the potential answer set removing anything that is not needed, such as “Stable Model:” in Smodels, and creating a string of the atoms separated by a “.”. Once it has searched through an entire line this newly created string is added to the array. Once all of the tokenized items have been checked this array of answer sets is then returned.

**clearAnswerSets**
Takes a list of atoms and converts them into ASP rules
Parameters: String as
Return: String[], of ASP rules from the given atoms
5.7.3 ASP Tools

This class provides the ASP functionality to the agents, it is the only class that directly controls the ASP solvers.

Global Variables

ASP Tools has the following Global Variables:

- **AID me** - AID of the agent that created the object
- **String institutionURL** - Stores the file-name of the institutional program, set and get methods for this variable are included.
- **ArrayList(String) filter** - Stores a copy of the filter atoms in an array list
- **String filter** - The filter in string form that can be set up the implementer of the agent.

Methods

**ASPGenerator**

This method calls the appropriate ASP solver specified by the user, depending on the solver either one or two processes will be called. If clasp or smodels are used then two processes are needed, as they require lparse to be run first. It also controls the information based to the processes whilst they are running. It then converts all of the output into a string to be dealt with by other methods.

Parameters: String `aspProgram`, String `aspSolver`

Return: String, the output of the program in string form

Method: Given the solver the system decides what processes to run and starts the relevant processes, if its smodels or clasp, lparse is run first using the program as its input. The output from the lparse program is stored as a string and then a second process is run, to which the output is passed. If it is DLV or oct just one process is required. All of the output is then added to a string as it is outputted by the processes and this string is the returned. This string will contain all of the answer sets and any other information for the program. Any error messages are outputted to the screen.

**checkContradiction**

Given the current input and the agents answer set program this method checks to see if there are any atoms that are false in the program but true in the input. If there is such an atom, or many, then they are passed to the `dealContradiction` method, along with the
name of the agent, which will then return whether the atom is true or false. This is done for all items in the input.
Return: String, is the currentInput minus any contradicting atoms.
Method:

1. For each atom in the input, the method checks to see if that item is the head of a rule in the current program.
2. If there is no rule the atom is added back to the input.
3. If the atom is the head of a rule then it runs the program with the input minus that atom.
4. If the atom is in the answer set, then there is NO contradiction.
5. If the atom is not in the answer set then clearly it is made not true in the program and so it is false, so we have a contradiction.
6. This is then sent to deal contradiction, which will either return the atom or nothing, depending on who is believed.
7. The method then adds all of the atoms, minus any contradictions, into a string and returns it.

dealContradiction
If a contradiction arises then the method decides whether to believe the agent or to trust the new input, this is based on what the user specified
Parameters: String Contradiction, AID contradictingAgent, boolean trustMe
Return: String, the outcome of the contradiction
Method: Given the agent that believes the contradicting atom to be true and whether the user which to trust the agent or not, the system will either return a letter, if the contradiction is deemed to be true, or nothing if it is deemed false.

filterAnswerSets
Takes an answer set as an input and then removes all atoms that are not in its filter string, as it can only pass on items that are in the string
Parameters: String AnswerSet
Return: String, returns the answer set but filtered
Method: It tokenizes the answer set and if an atom cannot be found in the filter then it is replaced by a space. The answer set is the reassembled without the spaces and this string is returned

getFilter
Converts the answer set filter from a string into an ArrayList(String)
Parameters: NONE
Return: NONE
Method: Takes the Filter global variable and tokenizes it based on “,”, all of the tokens are then added to an ArrayList

5.8 The Ontology

![Ontology Structure](image)

Using the SimpleJADEAbstractOntology template as a starting point in Protégé the ontology includes the following classes and slots:

- **Class: AID and AgentAction** - are both predefined as part of the template ontology.

- **Class: AnswerSet**
  Slots: AS - String that stores the answer set, input - String that stores the input the agent used, sender - AID that is the agents AID, isFixedPoint - boolean that stores whether the agent thinks it was a fixed point or not, TotalCycles - integer that stores the number of cycles this answer set has currently been through, Previous - an Array of AnswerSet that stores all the previous answer sets for checking for fixed points.

- **Class: NewAgent**
  Slots: AgentID - an AID that stores the information of the sender agent

- **Class: FixedPoint**
  Slots: fixedPointAchievable - a boolean that says whether the agent thinks the system has a fixed point or not, sender - an AID that stores the information of the sender agent

- **Class: StarterAgent**
  Slots: start - boolean that’s used to start the process
• Class: AgentArray
  Slots: AgentSet - an array of AIDs

• Class: InstitutionInfo
  Slots: InstitutionalData - a string that contains the institutions state, sender - an
  AID that the agents use to identify the institution

• Class: AgentsAction
  Slots: ActionPerformed - a string that stores what action the agent has performed
  that can then be interpreted by the institution, sender - an AID that is used by the
  institution to identify the agent that performed the action

Once the ontology has been created in Protégé JADE bean generator is then run and the
ontology is translated into Java code for use with the Multi-Agent System.

5.9 Summary

The implementation has not deviated, in the most part, from the initial design (see Section
4). There were no complications in converting the given design into the required Java code.
The descriptions of the more complicated areas of the Logic Agent, see Sections 4.5.2 to
4.5.4 were converted into pseudo-code to confirm the logic before being hard-coded into
the system.
Now that the system has been fully implemented it is important to ensure that the system
is working correctly and so is able to work on standard inputs and successfully deal with
likely errors. The next chapter will look at how the system will be tested and the results
of those test.
Chapter 6

Testing

6.1 Introduction

Now that the system has been produced it is important to minimise the number of errors it contains and to check that it can run for expected input. In order to remove as many errors as possible the system must be thoroughly tested, for this project Black-box testing will be used. Testing will not remove all of the errors in the system but will minimise the number of errors and ensure the system can function properly for expected input. Another vital role of testing is to help determine whether the system has met its original requirements. This chapter contains an overview of the testing strategy used, followed by a discussion of how the system meets the requirements.

6.2 Black-box Testing

Black-box Testing takes an external view of the software, i.e. testing does not require detailed knowledge of the internal structure or system code. Black-box testing involves the creation of a set of test using the initial design requirements, the test designer uses the requirements to obtain a set of valid and invalid inputs for the system and their expected outputs. The tests are conducted by entering these inputs into the system and observing the outputs. If the output matches the expected output the test was successful if they do not match then the test was not successful. To aid in the testing the JADE framework provides two helper agents, the dummy agent and the sniffer agent.

6.2.1 Sniffer and Dummy Agents

The sniffer and dummy agents are provided by the JADE framework for troubleshooting agent communication problems. The sniffer agent provides a GUI for monitor messages sent between agents, so the user can check which messages have been sent between agents
and their contents. The dummy agent allows the user to send messages to any agent in the system and allow the sniffer to monitor those messages.
Sample test outputs and Sniffer Agent screen captures can be found in Appendix A.

6.3 Critique of Testing

The tests proved to be effective and therefore it is expected that the system will work in the majority of cases. Black box testing was performed by analysing the output of the system and by looking at the sniffer agent output, so again the messaging and fixed point are expected to work. The product has been run successfully on two different machine, one running MS Windows XP and the other running Unix. The system ran successfully on both machines, however not all tests were performed on the Unix machine only when programs differed were tests run on both machines. OCT is only available for Unix so all tests involving OCT were performed on the Unix machine, similarly clasp is not available for Unix so all tests were performed on a Windows machine. Overall a basic level of testing was performed on the system which is sufficient for this prototype project to be used in further experimentation.

6.4 Validation of Requirements

It is important to compare the initial requirements to the test results to see how successful the system has been. All of the requirements will now be discussed in detail to see if they have been achieved.

6.4.1 Functional and Domain Requirements

The system must be written using Java 1.5.

This requirement is fully met. All parts of the system are written in Java, except obviously the answer set solvers and their associated programs.

Agents must be able to communicate effectively with each using JADE and an ontology.

This requirement has been fully met too. The agents are able to communicate by passing ACL Messages between each other, the messages contain FIPA performatives and using the SimpleREResponder behaviours of each agent are able to receive the message. The content of each message is set from the classes available from the ontology specified in Protégé. All agents have access to the ontology so are able to encode and decode information from
each other. Using these methods the agents are able to pass messages between each other effectively.

**The system must be built using JADE and Protégé and adhere to the FIPA specifications.**

The system meets this requirement fully. The system has been written to work with JADE and follows FIPA specifications for message sending. The system extends the JADE agent and uses ACL Messages, FIPA performatives and languages provided by JADE. The ontology was created using Protégé and then translated into Java code using the JADE Ontology Bean Generator. The ontology is then fully compatible with all agents and JADE.

**The system must support the ASP solvers: Clasp, Smodels, OCT and DLV.**

This requirements is fully met. The system is able to use all four solvers to compute answer sets. The agent is given a solver and program by the user, the system will then use the solver to compute the answer sets of the given program. In order for Smodels and Clasp to work correctly they require a grounded lparse output as its input. Lparse is called on the input program to created a grounded string which is then passed to Smodels or Clasp. There are several small problems with the solvers, OCT will only work in Unix as there is no other implementation available and Clasp will only work with Windows systems, this is because there is no currently release that is compatible with Unix. The system also assumes that the output of the program will always be the same, as it uses the expect output to convert it into a standardised answer set. If the output of the program differs in anyway then the system may produce different answer sets, however DLV does have some error handling for this.

**The system must allow for the use of an Institutional Agent to help govern agent behaviour.**

This requirement has been met for most cases of the system. The system is able to operate using an institution, which can calculate its own state and then send that state to all Logic Agents in the system. The Logic Agents then use this information along with their own program to compute a set of answer sets. As the agents in the system do not calculate what actions to take this information is not available to be sent to the institution. However the system does have stub methods in place that are able to send the agents actions to the institution, the institution also has a stub method that updates its internal state and then send this to all the agents. Due to a problem with OCLP programs institutions do not work with OCT programs.
The system must still be able to function without an institution.

This requirement has been fully met as the system is able to work with and without and institution. The user must specify when starting up whether they want an institution or not. If an institution is specified but one is not present when all of the Logic Agents have been added, the Monitor assumes that one is no longer required so proceeds as if no institution was specified.

The system must allow users to specify if they want to use an institution.

This requirement is fully met. The monitor agent takes an argument, that is either yes or no. If the argument is yes then an institution is used with the system otherwise the monitor will not except an institution in the system.

All Logic Agents should reason with their own program within the institutional boundaries, if an institution is used.

This requirement has been fully met. Each Logic Agent uses the Institutional Agents state along with its own program and any other input to produce an answer set. If the system involved action selection then the Logic Agent would choice an action based on the answer set generated from all inputs, including the Institutional Agents state. It would also send its action to the Institutional Agent to update its state.

The system should be able to find the evolutionary fixed-point of a set of ASP programs, if one exists.

This requirement has also been fully met. The Answer Set messages sent between Logic agents contain the previous agents input and output, the agents fixed point status and a history of previous answer sets. Each agent upon receiving an answer set message, uses the previous agents output as its input to generate a new set of answer sets. Once it has generated a new set of Answer Sets, the agent compares this input and output to the input and output it achieved in the last cycle. If the two match then the agent has a fixed point, so its sets the value of the fixed point to true. The agent then cycles through the history to see if all other agents have achieved a fixed point or not. If they have then the system has found an evolutionary fixed-point, so the agent send a stop message to all agents, with a true fixed point value. All agents will then stop sending messages and print out their fixed points. If a fixed point for the system has yet to be found the agent will create a new answer set message, with its information and send it to the next agent in the system. Obviously there will be cases where no fixed point will exist so the system must deal with this. Along with checking to see if the all agents have achieved a a fixed point or not, it checks to see if the previous agent has achieved a fixed point and it has not. If the previous agent has a fixed point but the current agent has not achieved one then no fixed point
exists. In this case the agent send a message to all agents telling them to stop as no fixed point is achievable. Finally to prevent the system running forever each agent has a counter that is set by the monitor telling them the maximum number of cycles that an answer set can go through. If an answer set reaches this number then it is assumed no answer set can be found, so the agent that reaches completes this number of cycles will send a message to all other agents informing them that a fixed point cannot be found and to stop calculating answer sets.

The system must allow users to specify the number of logic agents in the system.

This requirement again has been fully met. The Monitor Agent takes two arguments, the first specifies the number of agents that will be present in the system and the second deals with institutions and is covered in a previous requirement. When the user starts-up a Monitor Agent they specify the number of Logic Agents that will be in the system, the monitor will only start the agents when the number of Logic Agents in the system reaches this number. There are a few problems with this method but they will be discussed in the Conclusion, Section 7.2.2.

The Logic Agents ASP program and solver must be specified by the user.

This requirement has been fully met. All Logic Agents take two arguments, the first specifies the Answer Set Solver the agent will use and the second the URL of its program. If the user specifies a solver that the system does not support, currently it only supports Clasp, Smodels, OCT and DLV, then it will automatically default to DLV. The system will compute the answer sets of the program and then standardise them as mentioned above so that each agent can use a different solver and still be able to communicate effectively.

Agents should be able to make conclusions on contradicting information.

This requirement has been fully met. Each logic agent checks any input it receives from another agent against its own information to check that there are no contradictions between the two sets of information. If the agent finds conflicting information, i.e. it believes something to be true but the input makes it false then the agent reasons about the contradicting atoms. The agent will decide who to believe based on the level of trust it has in its own information and the other agents. The agent will make a choice and then either add the atom to its program or remove it, this will only apply to the reasoning for this particular input. The agent follows the same reasoning for all inputs it receives and so if the same input is received again it will always come to the same conclusions.
The agent must make contradiction decisions based on its level of trust in itself or the other agent

This requirement has been mostly achieved. The agent decides whether to believe its own information or that of the other agent depending on whether the implementer has set it to trust its own information over others or not. What was initially suggested, but was unable to be finished due to time constraints, was a system where the agent had a different level of trust in all agents of the system. For example, agent A might believe agent B over itself but not agent C. This level of trust can be left as a future improvement of the system.

Agents should be able to filter the information they pass to other agents, the filter must be defined by the person implementing the agent.

This requirement has been met. All Logic Agents have a list of atoms which they are able to pass to other agents. All of the agents’ answer sets are searched and if they have an atom in their answer set which is not in this list then it is removed from the answer set before it is sent to the other agents. The list of atoms is set by the implementer of each Logic Agent and can only be changed when the agent is compiled.

Logic Agents should be able to communicate with each other by passing answer sets.

This requirement has also been met fully. The Logic Agents communicate by passing ACL Messages between each other, these messages contain answer sets as their action. The answer set class not only contains the agents answer sets, but also the input it used to compute those answer sets (if any), whether a fixed point has been achieved and also a history of previous answer sets. This information can be used by the agent to compute the next answer sets and fixed point information. The agents also pass other messages but communicate in the large part using answer sets.

6.4.2 Non-Functional Requirements

Unlike the functional requirements its difficult to create tests for Non-Functional requirements as they are an emergent property of the system, so these requirements have not been tested. The non-functional requirements have for the most part been met, a selection of the non-functional requirements will be discussed below.

Implementation

Implementation has already been largely discussed in the functional requirements. The implementation meets the requirements outlined in Section 3.4.1 as it is built using JADE and Protégé, and implements the FIPA Standards.
CHAPTER 6. TESTING

Usability

Usability was not a big issue for the project as it was only a prototype project the system uses only the command line and GUI provided by JADE.

Robustness and Reliability

As stated in the initial requirements the system does not need to be highly robust or reliable. The system handles standard errors and is able to run successfully on most common inputs, if there are any errors then it will print this error message to the command line. The system is deemed reliable enough for the purpose of this project. Reliability and Error handling are discussed further in the conclusion, Section 7.2.2.

Interoperability

Using the ontology all agents are able to communicate with each other without any problems, as the ontology is common to all agents. Also the Logic Agents are able to use any answer set solver and convert the output into a standardised answer set which can then be passed to any other agent who can then translate this answer set into an input for their specific answer set program. This interoperability will work for all four of the specified solvers.

Ontology

The ontology was created to define all the classes needed by the system and allows the agents to communicate effectively. It contains all the information that the agents will need to pass between each other.

Portability and Standards

The agents have been tested with both Windows and Unix so the system should be able to work on both systems. The agents and ontology adhere to the standards set out by FIPA.

6.5 Summary

Almost all of the initial functional and non-functional requirements have been met by the final product, so if measured in those terms, the product can be deemed successful. The system meets most of its original design specification and thorough testing has proved that it is functional for a reasonable number of ASP programs. A more detailed discussion of the success of the project is to follow in the Conclusion, section 7.
Chapter 7

Conclusion

7.1 Introduction

The aim of this dissertation was to implement a Multi-Agent framework, where each agent's knowledge and reasoning capabilities are represented by Answer Set Programs. The project should allow for different types of answer set solvers and will also work with institutions. It also aimed to create a generic framework that can be easily reused to create a give the agents in the system a specific purpose.

This chapter will discuss the overall project, how well the aims were met and will suggest further research areas for the project. The chapter begins with an evaluation of the system, discussing the choices made and why, covering known bugs in the system. It will then discuss its more important findings and end with a personal critique of the project.

7.2 System Evaluation

The project began with a review of the current literature on the topics of Agents and Answer Set Programming. In doing this it was found that this is a relatively recent field with the majority of the literature coming from papers published within the last few years. It became apparent from the material that there was a need for a generic framework on which further research could be based. There were very few implementations of some key ideas so it would also be useful to attempt to prove that some of these ideas could be implemented in a usable system.

As the system was to be a prototype system there was no need to perform a user based analysis to gain requirements. It was felt that because of the type of system it would be best to have a broad high-level set of requirements that would cover the main functions required using the literature. At this point the decision was taken to focus on proving the concept instead of making a complete application. This decision has left the project open for a number of further developments which are discussed in Section 7.4. The choice was
also made to make the project as flexible as possible by including a number of possible solvers that could be used and the option of using an institution, this would also help to prove concepts over as wide a range of agent functions as possible. Another decision that had to be made at this stage was which functions should be implemented in the system, one of the key ideas in logic agents was how agents deal with contradiction, as that was to be included it also required at least some ideas of trust. Evolutionary fixed points were included as this would give the system a good stopping point. These decisions were taken as it is believed that these functions would be required in any basic multi-agent framework. Once the requirements were set it was important to translate these into a design, that would define the multi-agent framework and how it would work successfully. The decision was taken not to follow a strict methodology for designing the framework, apart from adhering to FIPA specifications, as this would make it easier to adjust the design if there were any major logical problems when implementing the system. As it turned out this was not the case as the design was successfully translated into the system.

One decision that influenced the design, in fact the whole project, was the use of JADE. The decision to use JADE was taken at the literature review stage. If JADE had not already been chosen as the platform for the agents then the design may have been done differently. However the use of JADE was an advantage. Without JADE the agents would have needed to be more complex to deal with the functionality that is currently provided by JADE and its library also helped by with some of the agents basic function. If the project were to be repeated I would keep the requirement elicitation and design the same. The decisions made were the correct ones for the type of project, deciding to use JADE and sticking to FIPA communication protocols were the correct choices and would be made again.

The next stage of the project was to implement the design. As already discussed the key decisions in the implementation of languages to use and to a certain extent class structure were already decided by the choice of JADE. The programming language used had to be Java to fit in with JADE, and as the JADE abstract agent class was used for the agents it determines some of the methods used. The use of JADE behaviours over Drools was made for simplicity, the behaviour handler was more than adequate for what was needed, so there was no need for something like Drools.

The code has been designed to be as modular as possible to enable it to be fully reusable, and as several methods are used by all agents this also helps to reduce code duplication. However there is still a large amount of functionality left in the Logic agent, if the code were to re-done all of the functionality would be moved out of the agent and placed into other classes so that it could simply be called from the Logic Agent. There are some known problems with the implementation that were not fixed during the testing stages, these problems and an explanation of why they were not removed from the system are covered in Section 7.2.2.

An evaluation of the testing carried out has already been discussed in Section 6.3 and will not be covered again here. Now that the projected has been evaluated in detail, its success or failure will now be discussed.
7.2.1 Project Success

The system can be deemed successful as it meets the original aims of the project and the design requirements obtained from those aims. These have been met as the system is a multi-agent framework where all the knowledge and reasoning capabilities are represented by a user specified answer set program. The system is capable of working with four different answer set solvers, clasp, OCT, smodels and DLV, and is able to work with an institution, except for OCT.
The system is a multi-agent framework that contains a set of generic agents that can be easily adapted to have more advanced/specific functions, so that the system can be used in any type of situation that requires a multi-agent system.

7.2.2 Known Bugs and Issues

There is a known issue within the system involving OCT and institutions. Currently an OCT solver cannot be used with an institution. This is because of the way OCT programs use components to reason. The other solvers use multiple files to reason with institutions, but this cannot be done with an OCT solver. It was decided as this was only a prototype system and due to time constraints a solution to this problem was not worth finding. However a solution should be possible, if the institutional information was simply added to the program instead of adding it as a new file. A decision would also need to be made with respect to the priority the agent gave to the information received from the institutional agent.
Also there can be a problem when specifying the number of Logic Agents in the system. Currently the user has to specify the number of agents when starting the Monitor Agent, however if the number of agents changes or there is a problem when starting the agents, then there is no way of telling the Monitor Agent that there number will be smaller or greater than the number expected. One possible solution is to allow the monitor agent to be updated, or even replaced by a new Monitor. So if a new Monitor is added the old Monitor will pass its information to the new Monitor and update all of the agents information. This is only a minor problem, and again as it is only a prototype system, it was felt that correcting this error would take more time in comparison to the potential improvements it would provide to the system.
Another issue with the system as it is only a proof of concept is that it has limited error handling, it deals with normal errors in the use of the program but all other errors are just printed to screen. In a more advanced system, if it was to be made into a fully usable system for example, this error handling could be expanded. The error handling could be improved to ask users for more input, or the system could try to locate files with more flexibility.
7.3 Achievements

The project has been developed as a generic high-level multi-agent framework. While performing the background research for the literature review, Chapter 2, a number of theoretical concepts were given that had not been put into practice. It was decided that the system should try and prove that some of these key ideas could be implemented in a system. The areas that the system implements are evolutionary fixed points and methods for dealing with contradicting information between agents and simple examples of trust between agents. The project also looks deeper into how filtering can be implemented in multi-agent systems. The project also helps to further the understanding of institutions and their interactions with agents and different answer set solvers. This project adds to the current research in the field by giving future developers a platform from which they can launch their research.

As the project progressed it became clear that there is a huge scope for future research in logic based multi-agent systems. This project provides a much needed framework from which many of these areas can be researched.

7.4 Further Research

There are a number of direction that this project can be taken to further the research:

7.4.1 Trust between Agents

One further development for the system would be to add greater levels of trust between the agents. Currently the agents only have one level of trust either they trust their information over other agents or they trust other agents information over their own. This could be improved so that the agent assigned levels of trust to itself and all other agents, it could update this from what it believed about the agents and what other agents believed about everyone else. The agent could then decide who to believe based on its level of trust in that agent.

7.4.2 Multiple Inputs

As the system stands the agents send answer sets in a cyclic manner, each agent receives information from one agent and then send it to another agent, the agent is chosen from the list provided by the monitor. To increase the flexibility of the system agents could be able to send and receive information form multiple agents, allowing the system to become more powerful.
7.4.3 Specified Research

The main purpose of the system was to be a proof of concept for some key ideas in the field of Multi-Agent Systems. The system is a more generic framework but has been designed in such a way that it allows for the use of the system in more specific areas of research. The basic framework can be used but with more advanced/specific functions added to the agent.

7.4.4 Improved Flexibility

To improve the flexibility of the software more Answer Set solvers could be added, this should be reasonably straightforward for someone with knowledge of Java and JADE. The way the system is structured makes it easy for more solvers to be added. Further improvements could be made so that the system allowed for multiple institutions and to fix the problems with OCT and institutions.

7.4.5 Complete Application

The system produced here, although it is a good foundation, is not a complete usable application. In order to make this into a more complete application it would require a greater level of error handling, that dealt with a greater number of errors and provide more extensive feedback. The user may also require a GUI instead of using the command line and JADE’s own GUI. As the system already implements FIPA specifications and is built using JADE it already has the key components of a complete application.

7.5 Personal Critique

This project was very interesting and a lot has been achieved. A great deal was learnt about multi-agent systems and how they work, and ontology design. The author has also learned a lot about logic programming especially answer set programming and its benefits over Prolog. The author’s programming abilities have been greatly improved in Logic Programming and in software development. The author enjoyed the greater levels of freedom in creating the project, as there were few defined requirements unlike most of the University projects. The project has also been useful in improving the authors general report writing ability and research skills in compiling the Literature Review.

7.6 Overall Summary

Multi-agent systems could potentially be the next big development in Artificial Intelligence. This project provides an insight into the uses of this technology and acts as a proof of
concept for many key ideas. This system has proved that this type of system can be used in a number of areas and it also provides a useful base for further research projects into the area.
Bibliography


Appendix A

Example Output

Some example tests outputs and outputs from the JADE sniffer agents have been included.

A.1 Sample Test Outputs

The system has been thoroughly tested, for more information on testing see Section 6. Some example outputs of those tests are shown in the following pages.

Test A.1 shows an example of all agents with the same solvers, computing the fixed-point of the system and ending when it finds it.

Test A.2 shows an example of all agents with the different ASP solvers, computing the fixed-point of the system and ending when it finds it.

Test A.3 shows an example of all agents with the same solvers but with an institutional agent, computing the fixed-point of the system and ending when it finds it.

Test A.4 shows an example of an error message from the system, with the agent LogB being given an incorrect filename.

Test A.5 shows an example on a Unix system, with the agents using dlv and oct to communicate. The warning is due to Unix being accessed remotely and is not a fault of the system, it does not occur on a normal Unix system.

A.2 Sniffer Agent Outputs

Two screen shots of the sniffer are shown in figure A.6 and A.7. They show messages being passed between all the agents:
APPENDIX A. EXAMPLE OUTPUT

Figure A.1: All agents with same solver

Figure A.2: All agents with different solver
Figure A.3: All agents and an Institutional Agent

Figure A.4: Wrong file-name error message
Figure A.5: Agent on Unix using OCT
Figure A.6: Sniffer Agent Example
Figure A.7: Sniffer Agent Example
Appendix B

Source Code

A copy of the source code for all of the main classes of the program is attached.
B.1 MonitorAgent.java

package lomas;

import jade.content.Concept;
import jade.content.ContentElement;
import jade.content.ContentManager;
import jade.content.lang.sI.SLCodec;
import jade.content.onto.Ontology;
import jade.content.onto.OntologyException;
import jade.content.onto.basic.Action;
import jade.core.AID;
import jade.core.Agent;
import jade.core.behaviours.SenderBehaviour;
import jade.domain.DFService;
import jade.domain.FIPAException;
import jade.domain.FIPAAgentManagement.DFAgentDescription;
import jade.domain.FIPAName.
import java.util.*;
import lomas.ontology.*;

// List of all Logic Agents in the system and the institution
private AID institution;
private ArrayList<AID> agentList = new ArrayList<AID>();
private int numberAgents = 0;

// used to distinguish between the use of an institution and not
private boolean withInst = false;

protected void setup()
{
    // Takes the arguments and determine the number of agents to be used and whether an institution will be used or not
    Object[] args = getArguments();
    String str = args[0].toString().trim();
    numberAgents = Integer.parseInt(str);
    String str2 = args[1].toString().toLowerCase().trim();
    if (str2.equals("yes")) withInst = true;
    doWait(1000);
    System.out.println(getLocalName() + " Starting Monitor Agent");
    // Sets the agents information
    // Not changed from original agent
    language = new SLCodec();
    ontology = LomasOntology.getInstance();
    DFAgentDescription description = new DFAgentDescription();
    description.addLanguages(language.getName());
    description.addOntologies(ontology.getName());
    description.addProtocols(InteractionProtocol.FIPA_REQUEST);
    description.setName(getAID());

    // Some code reused from the Oracle Agent of the CM30174 Coursework Supply Chain Agents
    public class MonitorAgent extends Agent{

        private Codec language;
        private Ontology ontology;

        // Some code reused from the Oracle Agent of the CM30174 Coursework Supply Chain Agents
        protected void setup()
        {
            // Takes the arguments and determine the number of agents to be used and whether an institution will be used or not
            Object[] args = getArguments();
            String str = args[0].toString().trim();
            numberAgents = Integer.parseInt(str);
            String str2 = args[1].toString().toLowerCase().trim();
            if (str2.equals("yes")) withInst = true;
            doWait(1000);
            System.out.println(getLocalName() + " Starting Monitor Agent");
            // Sets the agents information
            // Not changed from original agent
            language = new SLCodec();
            ontology = LomasOntology.getInstance();
            DFAgentDescription description = new DFAgentDescription();
            description.addLanguages(language.getName());
            description.addOntologies(ontology.getName());
            description.addProtocols(InteractionProtocol.FIPA_REQUEST);
            description.setName(getAID());

            ServiceDescription servicedesc = new ServiceDescription();
            servicedesc.setName(getLocalName());
            servicedesc.setType("MONITOR_AGENT");
        }
    }
}
the service has a list of supported languages, ontologies
and protocols for this service.

```
servicedesc.addLanguages(language.getName());
servicedesc.addOntologies(ontology.getName());
servicedesc.addProtocols(InteractionProtocol.FIPA_REQUEST);
```

description.addServices(servicedesc);

```
// register with the DF
try{
    DFService.register(this, description);
}
```catch(FIPAException e){
    System.err.println(getLocalName() + ":error_registering
    with DF, exiting:" + e);
doDelete();
return;
}
addBehaviour(new MonitorResponderBehaviour(this));

```
// Listen to and reponsed to all incoming events
// received by the monitor
class MonitorResponderBehaviour extends
    SimpleAchieveREResponder{
    public MonitorResponderBehaviour(Agent agent){
        // Calls the constructor of SimpleAchieveREResponder
        super(agent, MessageTemplate
            .and(createMessageTemplate(
                InteractionProtocol.FIPA_REQUEST), MessageTemplate
            .and(MessageTemplate.MatchOntology(ontology.getName()),
                MessageTemplate.MatchLanguage(language.getName()))));
    }
    // This extracts the message content and then dispatches the
    // body of the message to a sub_handler.
    public ACLMessage prepareResponse(ACLMessage msg){
        ContentElement content = null;
        // Adds another instance of the behaviour so that we can
        // concurrently deal with another request.
        myAgent.addBehaviour(new
            MonitorResponderBehaviour(myAgent));
        try{
            try{
                content =myAgent.getContentManager().extractContent(msg);
            }
            catch(EncoderException e){
                System.err.println("Error relating to message and
                ontology:":" + e);
                throw new I DontUnderstand("" + msg.toString() + ": Error_
                Parsing the message:" + e);
            throw new IDontUnderstand("" + msg.toString() + ");
        }
        Concept action_content;
        if(content==null || !(content instanceof Action)){
            System.err.println(getLocalName() + ":Message body was
            not an action");
            throw new IDontUnderstand("" + msg.toString() + ");
        action_content = ((Action) content).getAction();
        if(action_content==null ){
            System.err.println("Message body was empty");
            throw new IDontUnderstand("" + msg.toString() + ");
        }if(action_content instanceof NewAgent){
            // If the content of the message is a NEWAgent message
            then add the new
            // agent to the array of
            // agents and inform all other agents
            NewAgent na = (NewAgent) action_content;
            agentList.add(na.getAgentID());
            informAgents(na);
        }else if(action_content instanceof AgentArray){
            sendAgentList(msg.getSender());
            if(agentList.size()==numberAgents){
                // If there is an institution then start that otherwise
                start the agents
                if(withInst) startInstitution();
                else startAgents();
            }
        }else{
            System.err.println("Unknown action body class" +
```
```java
// After a request for all other agents has been received the agent sends
// a reply message which is an array of all logic agents to the requesting agent
private void sendAgentList(AID sendTo) {
    doWait(1000);
    ACLMessage msg = new ACLMessage(ACLMessage.REQUEST);
    msg.setProtocol(InteractionProtocol.FIPAREQUEST);
    msg.setLanguage(language.getName());
    msg.setOntology(ontology.getName());
    msg.setSender(getAID());
    msg.addReceiver(sendTo);
    AgentArray aa = new AgentArray();
    for (int i = 0; i < agentList.size(); i++) { // aa.addAgentSet(agentList.get(i));
        Action action = new Action(sendTo, aa);
        try {
            getContentManager().fillContent(msg, action);
        } catch (OntologyException e) {
            System.err.println("Error with message ontology" + e);
        } catch (Codec.CodecException e) {
            System.err.println("Error with message encoding" + e);
        }
        addBehaviour(new SenderBehaviour(this, msg));
    }
}

// Send the new agents information to all other agents
private void informAgents(NewAgent newAgent) {
    if (!agentList.isEmpty()) { // for(int i = 0; i < agentList.size(); i++) {
        ACLMessage msg = new ACLMessage(ACLMessage.REQUEST);
        msg.setProtocol(InteractionProtocol.FIPAREQUEST);
        msg.setLanguage(language.getName());
        msg.setOntology(ontology.getName());
        if (agentList.get(i).equals(newAgent.getAgentID())) {
            System.err.println("Message created as before with NewAgent information as content");
            ACLMessage msg = new ACLMessage(ACLMessage.REQUEST);
            msg.setProtocol(InteractionProtocol.FIPAREQUEST);
            msg.setLanguage(language.getName());
            msg.setOntology(ontology.getName());
            msg.setSender(getAID());
            msg.addReceiver(agentList.get(i));
        }
        Action action = new Action(agentList.get(i), newAgent);
        try {
            getContentManager().fillContent(msg, action);
        } catch (OntologyException e) {
            System.err.println("Error with message ontology" + e);
        } catch (Codec.CodecException e) {
            System.err.println("Error with message encoding" + e);
        }
        addBehaviour(new SenderBehaviour(this, msg));
    }
}
```

getContentPane().fillContent(msg, action);
} catch (OntologyException e) {
    System.err.println("Error with message ontology"+ e);
}

catch (Codec.CodeException e) {
    System.err.println("Error with message encoding"+ e);
}

addBehaviour(new SenderBehaviour(this, msg));
}

else{
    System.out.println(getLocalName() + ": No Other Agents to send inform");
}

private void startTimeInstitution(){
    getInstitution();
    sendAgentList(institution);
}

private void getInstitution(){
    // if we have not already got a institutional agents then try and find one
    // if we have then return that AID
    if(institution == null){
        // Sends a request to the DF, asking for a list of all agents that met the given description, "AGENT"
        DFAgentDescription searchdesc = new DFAgentDescription();
        ServiceDescription servicedesc = new ServiceDescription(
            ServiceDescription().setServiceType("INSTITUTIONALAGENT"),
            servicedesc.addOntologies(ontology.getName()),
            servicedesc.addLanguages(language.getName()),
            servicedesc.addProtocols(new InteractionProtocol.FIPAREQUEST),
            searchdesc.addServices(servicedesc);
        DFAgentDescription results [];
        try {
            results = DFService.search(this, searchdesc);
        } catch (FIPAException e) {
            System.err.println(getLocalName() + ": Could not search the DF, exiting:" + e);
        }
        results = new DFAgentDescription[0];
    }
}

// If there is more than one monitor just return the first one
if (results.length > 0) institution = results[0].getName();

else{

    System.err.println(getLocalName() + ": DF search returned no results");
    System.out.println(getLocalName() + "Continuing as if no Institution....");
    startAgents();
}

// This class is an exception message generated by the agent
static class IDontUnderstand extends Exception{
    public IDontUnderstand(String msg){
        super(msg);
    }
}

B.2 LogicAgent.java

package lomas;

import jade.content.Concept;
import jade.content.ContentElement;
import jade.content.ContentManager;
import jade.content.lang.SLCodec;
import jade.content.lang.owl.Ontology;
import jade.content.lang.owl.OntologyException;
import jade.content.lang.owl.Action;
import jade.core.Agent;
import jade.core.AID;
import jade.core.behaviours.SenderBehaviour;
import jade.domain.DFService;
import jade.domain.FIPAAgentManagement.DFAgentDescription;
import jade.domain.FIPA.AgentManagement.ServiceDescription;
import jade.domain.FIPA.Names.InteractionProtocol;
import jade.lang.acl.ACLMessage;
import jade.lang.acl.MessageTemplate;
import jade.proto.SimpleAchieveREResponder;
import java.util.*;
import lomas.ontology.*;

This is the main agent type that would reason using answer set programming and transmit it answer sets to other agents.

/*
 * This is the main agent type that would reason using answer set programming and transmit it answer sets to other agents
 */

//Some code reused from the Oracle Agent of the CM30174 Coursework Supply Chain Agents
public class LogicAgent extends Agent {
    // Stores the agents communication information
    private Codec language;
    private Ontology ontology;
    private AID monitorAgent;
    private AID institutionalAgent = null;
    // Stores the ASP details
    private String aspSolver;
    private String aspProgram;
    // If this variable is true the agent will believe itself over other agents
    private boolean trustMe = true;
    // No of cycles the message should make, before it stops sending messages
    private int noCycles = 15;
    // List of all other agents in the system
    private ArrayList<AID> agentList = new ArrayList<AID>();
    private int nextAgent = 0;
    // Fixed Point Details
    private boolean fixedPointAchievable = true;
    private ArrayList<String> fixedPoint = new ArrayList<String>();

    //All classes required by the program are instantiated
    private ASPTools aspTools;
    private FileWriterTools fileTools = new FileWriterTools();
    private StringTools stringTools = new StringTools();

    protected void setup(){
        Object[] args = getArguments();
        aspSolver = args[0].toString().trim().toLowerCase();
        aspProgram = args[1].toString();
        aspTools = new ASPTools(getAID());
        doWait(1000);
        System.out.println(getLocalName() + ": Starting Logic Agent");
        // Sets the agents information
        language = new SLCodec();
        ontology = LomasOntology.getInstance();
        ContentManager manager = getContentManager();
        manager.registerLanguage(language);
        manager.registerOntology(ontology);

        // register service description
        DFAgentDescription description = new DFAgentDescription();
        description.addLanguages(language.getName());
        description.addOntologies(ontology.getName());
        description.addProtocols(InteractionProtocol.FIPA_CONTRACT_NET);
        description.setName(getAID());
        ServiceDescription servicedesc = new ServiceDescription();
        servicedesc.setName(getLocalName());
        servicedesc.setType("AGENT");
        // the service has a list of supported languages, ontologies
        // and protocols for this service.
servicedesc.addLanguages(language.getName());
servicedesc.addOntologies(ontology.getName());
servicedesc.addProtocols(InteractionProtocol.FIPA_REQUEST);
description.addServices(servicedesc);
//register with the DF
try{
  DFService.register(this, description);
} catch(FIPAException e){
  System.err.println(getLocalName() + ": error registering with DF, exiting:" + e);
doDelete();
return;
}
//Find the monitor Agent
AID theMonitor = getMonitorAID();
//If one doesn't exist then exit
if (theMonitor == null) {
  System.err.println(getLocalName() + ": No_monitor_found, exiting");
doDelete();
return;
}
//Tell the monitor of your existence
sendConfirmMonitor();
//Add behaviour that receives messages
addBehaviour(new LogicResponderBehaviour(this));
//Get a list of all other agents
getOtherAgents();
//Stores a copy of all other agents
fileTools.setProg(aspProgram, aspSolver);
private void sendConfirmMonitor(){
  ACLMessage msg = new ACLMessage(ACLMessage.REQUEST);
  msg.setProtocol(InteractionProtocol.FIPA_REQUEST);
  msg.setLanguage(language.getName());
  msg.setOntology(ontology.getName());
  msg.setSender(getAID());
  msg.addReceiver(monitorAgent);
  NewAgent na = new NewAgent();
  na.setAgentID(getAID());
  Action action = new Action(monitorAgent, na);
  try {
    getContentManager().fillContent(msg, action);
  } catch (OntologyException e) {
    System.err.println("Error with message ontology"+ e);
  } catch (Codec.CodecException e) {
    System.err.println("Error with message encoding"+ e);
  }
  addBehaviour(new SenderBehaviour(this, msg));
}
private void calculateAnswerSets(){
  String output = aspTools.aspGenerator(aspSolver, aspProgram);
  //Run the answers through a method to remove any unused information and standardise
  //the output
  ArrayList<String> answerSetList = stringTools.justAnswerSets(output, aspSolver);
  if (output != null){
    //If there is an output then send it to all other agents
    sendAnswerSets(answerSetList, false);
  } else{
    System.out.println(getLocalName() + ": No Answer Sets to send, informing other agents and exiting");
    sendFixedPoint(false);
    doDelete();
  }
  //Takes the given answer sets and send them to all agents registered as logic agents in the system
  private void sendAnswerSets(ArrayList<String> answerSets, boolean again){
    //Code here
for (int j = 0; j < answerSets.size(); j++) {
    ACLMessage msg = new ACLMessage(ACLMessage.REQUEST);
    msg.setProtocol(InteractionProtocol.FIPA_REQUEST);
    msg.setLanguage(language.getName());
    msg.setOntology(ontology.getName());
    msg.setSender(getAID());
    msg.addReceiver(agentList.get(nextAgent));
    AnswerSet asp = new AnswerSet();
    // Adds all of the answer sets to the message
    String filteredAS = aspTools.filterAnswerSets(answerSets.get(j));
    asp.setAS(stringTools.alphabetise(filteredAS));
    asp.setInput("");
    asp.setSender(getAID());
    asp.setIsFixedPoint(false);
    // Sets the number of cycles that cannot be exceeded
    asp.setTotalCycles(agentList.size() * noCycles);
    Action action = new Action(agentList.get(nextAgent), asp);
    try {
        getContentManager().fillContent(msg, action);
    }
    catch (OntologyException e) {
        System.err.println("Error with message ontology" + e);
    }
    catch (Codec.CodecException e) {
        System.err.println("Error with message encoding" + e);
    }
    addBehaviour(new SenderBehaviour(this, msg));
}
private void getOtherAgents() {
    agentList.clear();
    // Sends a request to the monitor, asking for a list of all
    // agents that have registered with it
    ACLMessage msg = new ACLMessage(ACLMessage.REQUEST);
    msg.setProtocol(InteractionProtocol.FIPA_REQUEST);
    msg.setLanguage(language.getName());
    msg.setOntology(ontology.getName());
    msg.setSender(getAID());
    msg.addReceiver(monitorAgent);
    AgentArray aa = new AgentArray();
    aa.addAgentSet(getAID());
    Action action = new Action(monitorAgent, aa);
    try {
        getContentManager().fillContent(msg, action);
    }
    catch (OntologyException e) {
        System.err.println("Error with message ontology" + e);
    }
    catch (Codec.CodecException e) {
        System.err.println("Error with message encoding" + e);
    }
    addBehaviour(new SenderBehaviour(this, msg));
}
private AID getMonitorAID() {
    // If we have not already got a monitor agent then try and find one
    if (monitorAgent == null) {
        // Sends a request to the DF, asking for a list of all
        // agents that met the given description, "AGENT"
        DFAgentDescription searchdesc = new DFAgentDescription();
        ServiceDescription servicedesc = new ServiceDescription();
        servicedesc.setType("MONITORAGENT");
        servicedesc.addOntologies(ontology.getName());
        servicedesc.addLanguages(language.getName());
        servicedesc.addProtocols(InteractionProtocol.FIPA_REQUEST);
        searchdesc.addServices(servicedesc);
        DFAgentDescription results[];
        try {
            results = DFService.search(this, searchdesc);
        }
        catch (FIPAException e) {
            System.err.println(getLocalName() + ": could not search the DF, exiting:" + e);
            return null;
        }
    // If there is more than one monitor just return the first one
    if (results.length > 0) monitorAgent = results[0].getName();
else {
    System.err.println(getLocalName() + ": DF_search
    returned no results" );
    return null;
}

return monitorAgent;

// Listen to and respond to all incoming events
// received by the monitor

class LogicResponderBehaviour extends
SimpleAchieveREResponder{
public LogicResponderBehaviour(Agent agent){
    super(agent, MessageTemplate
        .and(createMessageTemplate(_
            InteractionProtocol.FIPA_REQUEST),
            MessageTemplate.and(_
                MessageTemplate.MatchOntology(ontology.getName()),
                MessageTemplate.MatchLanguage(language.getName()))));
}

    /∗
    * Deals with all messages received that match the
    * msgtemplate
    */
    public ACLMessage prepareResponse(ACLMessage msg){
        //ACLMessage response = msg.createReply();
        ContentElement content = null;
        // Adds instance of the behaviour so that we can
        // concurrently deal with another request
        myAgent.addBehaviour(new
            LogicResponderBehaviour(myAgent));
        try{
            try{
                content =myAgent.getContentManager().extractContent(msg);
            }catch(OntologyException e){
                System.err.println(myAgent.getLocalName() + ": Error
                relating to message and ontology": + e);
                throw new IDontUnderstand("" + msg.toString()+ "");
            }

            Concept action_content;
            if(content==null || !(content instanceof Action)){
                System.err.println(getLocalName() + ": Message.body was
                not an action" );
                throw new IDontUnderstand("" + msg.toString()+ "");
            }
            action_content = ((Action) content).getAction();
            if(action_content==null ){
                System.err.println("Message.body was empty");
                throw new IDontUnderstand("" + msg.toString()+ "");
            }
            if(action_content instanceof lomas.ontology.AnswerSet){
                // If the message content is a set of answer sets
                // then add them to my current knowledge
                AnswerSet abc;
                abc = (AnswerSet) action.content;
                calculateFixedPoint(abc);
                return null;
            }else if(action_content instanceof
                lomas.ontology.InstitutionInfo){
                InstitutionInfo newInt = (InstitutionInfo)action.content;
                institutionalAgent = newInt.getSender();
                if(!newInt.getInstitutionData().equals("")){
                    String URL =
                        fileTools.writeFile(newInt.getInstitutionData(),
                        getAID());
                    aspTools.setInstURL(URL);
                }
            }else if(action_content instanceof
                lomas.ontology.AgentArray){
                // Receives an array from the monitor agent of all the
                // other logic agents in /the system
                // It then assigns the agent after itself in the array to
                // be the next agent
                AgentArray aa = (AgentArray) action.content;
                Iterator it = aa.getAllAgentSet();
                while(it.hasNext()){}
                agentList.add((AID) it.next());
            }
            if (!agentList.isEmpty()){
for (int i = 0; i < agentList.size(); i++) {
    System.out.println(getLocalName() + "\": Adding\"Agent\" +
    agentList.get(i).getLocalName());
    if (agentList.get(i).equals(getAID())) {
        if ((i + 1) < agentList.size()) nextAgent = i + 1;
    } else nextAgent = 0;
} else if (action_content instanceof
    lomas.ontology.NewAgent) {
    // If the message content is a new agent
    // then add this agent to my list of other agents
    NewAgent na = (NewAgent) action_content;
    if (!na.getAgentID().equals(getAID())) {
        if (!agentList.contains(na.getAgentID())) {
            agentList.add(na.getAgentID());
            // If you currently loop back to the first agent then set
            // next agent to be the new one
            if (nextAgent == 0) {
                nextAgent = agentList.size() - 1;
            }
        }
    }
    System.out.println(getLocalName() + ",\"next\"agent\" +
    agentList.get(nextAgent).getLocalName());
    return null;
} else if (action_content instanceof
    lomas.ontology.FixedPoint) {
    // Confirms that a fixed point has been found or cannot
    // be found
    checkFixedPoint((FixedPoint) action_content);
    return null;
} else if (action_content instanceof
    lomas.ontology.StarterAgent) {
    // If this agent is selected to start first then start
    // calculating
    StarterAgent sa = (StarterAgent) action_content;
    System.out.println(getLocalName() + ",\"Starting\"Process\");
    if (sa.getStart() == true) {
        calculateAnswerSets();
    }
    return null;
} else {
    System.err.println("Unknown\"action\"body\"class\" +
    action_content.getClass().toString());
    throw new IDontUnderstand("(" + msg.toString() + ")");
} catch (IDontUnderstand e) {
    System.out.println(e.getMessage().toString());
    return null;
}

public ACLMessage prepareResultNotification(ACLMessage msg1, ACLMessage msg2) {
    // Method does nothing but is required by the class
    return null;
}

private void calculateFixedPoint(AnswerSet as) {
    String newInput = as.getAS();
    // Remove the items this agent made true last time from
    // the received answer set
    ArrayList<AnswerSet> list = getPrevious(as);
    for (int i = list.size() - 1; i >= 0; i--) {
        if (list.get(i).getSender().equals(getAID())) {
            newInput = stringTools.removeTrue(list.get(i).getAS(),
            list.get(i).getInput(), as.getAS());
            break;
        }
    }
    // OCT does not need to check for contradictions
    if (!aspSolver.equals("oct")) newInput =
    aspTools.checkContradictions(newInput,
    aspProgram, aspSolver, fileTools, as.getSender(),
    trustMe);
    String newASPProg = fileTools.appendToFile(aspProgram,
    newInput, aspSolver);
    if (newASPProg.equals(null)) errorExit("New\"ASP\"Program\" could\not\be\created");
String output = aspTools.aspGenerator(aspSolver, newASPProg);
fileTools.deleteFile(newASPProg);
//Run the answers through a method to remove any unused information and standardise
//the output
ArrayList<String> answerSetList = 
   stringTools.justAnswerSets(output, aspSolver);
boolean isFP = checkPrevious(answerSetList, as);  
int check = -1;
if(as.getIsFixedPoint()==true)
   check = isFixedPoint(as, isFP);
//Send information back to Institutional Agent, if there is one
if(institutionalAgent != null)
   selectAction(answerSetList);
if(check == -1){
   //Adds the last received set of answer sets to the previously used answer sets
   //and send the answer sets its worked out to the next agent
   Iterator it = as.getAllPrevious();
   //If the number of answer sets is now greater than the number of cycles then stop
   //and say there is no fixed point
   if(as.getPrevious().size()+1 <= as.getTotalCycles()){
      ACLMessage msg = new ACLMessage(ACLMessage.REQUEST); 
      msg.setProtocol(InteractionProtocol.FIPA_REQUEST);
      msg.setLanguage(language.getName());
      msg.setOntology(ontology.getName());
      msg.setSender(getAID());
      msg.addReceiver(agentList.get(nextAgent));
      //Adds all of the answer sets to the message
      AnswerSet asp = new AnswerSet();
      String alphaInput = stringTools.alphabetise(asp.getAS());
      asp.setInput(alphaInput);
      String alphaAS = stringTools.alphabetise(aspTools.filterAnswerSets(answerSetList.get(j)));
      asp.setAS(alphaAS);
      asp.setSender(getAID());
      asp.setIsFixedPoint(isFP);
      asp.setTotalCycles(as.getTotalCycles());
      ACLMessage msg = new ACLMessage(ACLMessage.REQUEST);
      msg.setProtocol(InteractionProtocol.FIPA_REQUEST);
      msg.setLanguage(language.getName());
      msg.setOntology(ontology.getName());
      msg.setSender(getAID());
      msg.addReceiver(agentList.get(nextAgent));
      //Adds all of the answer sets to the message
      AnswerSet asp = new AnswerSet();
      String alphaInput = stringTools.alphabetise(asp.getAS());
      asp.setInput(alphaInput);
      String alphaAS = stringTools.alphabetise(aspTools.filterAnswerSets(answerSetList.get(j)));
      asp.setAS(alphaAS);
      asp.setSender(getAID());
      asp.setIsFixedPoint(isFP);
      asp.setTotalCycles(as.getTotalCycles());
      while(it.hasNext()){
         asp.addPrevious((AnswerSet) it.next());
      }
      asp.addPrevious((AnswerSet) it.next());
   }}
   asp.addPrevious((AnswerSet) it.next());
   Action action = new Action(agentList.get(nextAgent), as);
   try {
      getContentManager().fillContent(msg, action);
   }
   catch (OntologyException e) {
      System.err.println("Error with message ontology" + e);
   }
   catch (Codec.CodecException e) {
      System.err.println("Error with message_encoding" + e);
   }
   addBehaviour(new SenderBehaviour(this, msg));
}}
else{
   //If the number of answer sets has exceeded your stated amount then stop
   sendFixedPoint(false);
}
}

//This is just a stub method, as this program is not concerned with the
//agents action selection, if the system were to be more detailed
//then this would contain the agents mechanism for determining the
//actions it would take given different answer sets.
//This information is then sent to the institutional agent to make decisions.
private void selectAction(ArrayList<String> AnswerSetList){
   String action = "";
   sendInstAction(action);
}

//Sends the action that the agent performed to the institution
private void sendInstAction(String currentAction){
   ACLMessage msg = new ACLMessage(ACLMessage.REQUEST);
   msg.setProtocol(InteractionProtocol.FIPA_REQUEST);
   msg.setLanguage(language.getName());
   msg.setOntology(ontology.getName());
   msg.setSender(getAID());
   msg.addReceiver(agentList.get(nextAgent));
   //Sends the action that the agent performed to the institution
   ACLMessage msg = new ACLMessage(ACLMessage.REQUEST);
   msg.setProtocol(InteractionProtocol.FIPA_REQUEST);
   msg.setLanguage(language.getName());
   msg.setOntology(ontology.getName());
   msg.setSender(getAID());
   msg.addReceiver(agentList.get(nextAgent));
   //Sends the action that the agent performed to the institution
   ACLMessage msg = new ACLMessage(ACLMessage.REQUEST);
   msg.setProtocol(InteractionProtocol.FIPA_REQUEST);
   msg.setLanguage(language.getName());
   msg.setOntology(ontology.getName());
   msg.setSender(getAID());
   msg.addReceiver(agentList.get(nextAgent));
   //Sends the action that the agent performed to the institution
   ACLMessage msg = new ACLMessage(ACLMessage.REQUEST);
   msg.setProtocol(InteractionProtocol.FIPA_REQUEST);
   msg.setLanguage(language.getName());
   msg.setOntology(ontology.getName());
   msg.setSender(getAID());
   msg.addReceiver(agentList.get(nextAgent));
   //Sends the action that the agent performed to the institution
   ACLMessage msg = new ACLMessage(ACLMessage.REQUEST);
   msg.setProtocol(InteractionProtocol.FIPA_REQUEST);
   msg.setLanguage(language.getName());
   msg.setOntology(ontology.getName());
   msg.setSender(getAID());
   msg.addReceiver(agentList.get(nextAgent));
   //Sends the action that the agent performed to the institution
   ACLMessage msg = new ACLMessage(ACLMessage.REQUEST);
   msg.setProtocol(InteractionProtocol.FIPA_REQUEST);
   msg.setLanguage(language.getName());
   msg.setOntology(ontology.getName());
   msg.setSender(getAID());
   msg.addReceiver(agentList.get(nextAgent));
   //Sends the action that the agent performed to the institution
msg.setLanguage(language.getName());
msg.setOntology(ontology.getName());
msg.setSender(getAID());
msg.addReceiver(institutionalAgent);

AgentsAction aa = new AgentsAction();
aa.setSender(getAID());
aa.setActionPerformed(currentAction);
Action action = new Action(institutionalAgent, aa);

try {
    getContentManager().fillContent(msg, action);
} catch (OntologyException e) {
    System.err.println("Error with message ontology" + e);
} catch (Codec.CodecException e) {
    System.err.println("Error with message encoding" + e);
}

addBehaviour(new SenderBehaviour(this, msg));

// Is a helper function that returns all of the previous answer sets as an array
private ArrayList<AnswerSet> getPrevious(AnswerSet as) {
    ArrayList<AnswerSet> list1 = new ArrayList<AnswerSet>();
    Iterator it = as.getAllPrevious();
    while (it.hasNext()) {
        list1.add((AnswerSet) it.next());
    }
    return list1;
}

// returns true if the current answersets and input match those received last time
private boolean checkPrevious(ArrayList<String> answerSets, AnswerSet as) {
    ArrayList<AnswerSet> list1 = getPrevious(as);
    // Cycles through all of the previous answer sets in reverse until it finds
    // its previous entry
    for (int i = as.getPrevious().size() - 1; i >= 0; i--) {
        if (list1.get(i).getSender().equals(getAID())) {
            if (list1.get(i).getInput().equals(stringTools.alphabetise(as.getAS()))) {
                if (stringTools.alphabetise(aspTools.filterAnswerSets(answerSets.get(0))).equals(.
                    list1.get(i).getAS()))) {
                    fixedPoint.addAll(answerSets);
                    return true;
                }
            }
        }
    }
    return false;
}

// If the agent currently has a fixed point, or the previous agent had a fixed point
// it checks to see if all agents now have fixed point or if a fixed point is no longer possible
private int isFixedPoint(AnswerSet as, boolean fpf) {
    int fixedFound = -1;
    ArrayList<AnswerSet> list1 = getPrevious(as);
    // If the previous one is true and this is false then there is no answer set
    if (list1.get(list1.size() - 1).getIsFixedPoint() == true) {
        if (fpf == false) fixedFound = 0;
    } // if all of the agents in the loop have registered true then we have reached a fixed point
    for (int i = list1.size() - 1; i >= 0; i--) {
        if (list1.get(i).getSender().equals(getAID())) {
            fixedFound = 1;
            break;
        } else if (list1.get(i).getIsFixedPoint() != true) return fixedFound;
    }
    if (fixedFound == 1) sendFixedPoint(true);
    else if (fixedFound == 0) sendFixedPoint(false);
    return fixedFound;
}

// Sends a fixed point message to all agents in the system
private void sendFixedPoint(boolean fpf) {
    for (int i = 0; i < agentList.size(); i++) {
        ACLMessage msg = new ACLMessage(ACLMessage.REQUEST);
        msg.setProtocol(InteractionProtocol.FIPA_REQUEST);
        msg.setLanguage(language.getName());
        msg.setOntology(ontology.getName());
        msg.setSender(getAID());
        msg.addReceiver(agentList.get(j));
    }
}
FixedPoint fp = new FixedPoint();
fp.setFixedPointAcheivable(fp);
fp.setSender(getAID());

Action action = new Action(agentList.get(j), fp);
try {
getFragmentManager().fillContent(msg, action);
} catch (OntologyException e) {
System.err.println("Error with message_ontology" + e);
}

addBehaviour(new SenderBehaviour(this, msg));

private void checkFixedPoint(FixedPoint fp)
{
fixedPointAcheivable = fp.getFixedPointAcheivable();
// If the fixed point can't be achieved then exit the program
if (!fixedPointAcheivable) {
System.out.println(getLocalName() + ": A fixed point cannot be found. For this answer set.");
if (!aspTools.getInstURL().equals("")
    fileTools.deleteFile(_aspTools.getInstURL());
else {
// If you can find a fixed point then print out the fixed point
System.out.println(getLocalName() + ": Fixed Point Answer Sets: ");
for (int i = 0; i < fixedPoint.size(); i++) {
    System.out.println(stringTools.alphaBetaize(fixedPoint.get(i)));
}
if (!aspTools.getInstURL().equals("")
    fileTools.deleteFile(aspTools.getInstURL());
doDelete();
}
}

private void errorExit(String errorMessage)
{
System.out.println("The following error occurred:");
System.out.println(errorMessage + " the agent will now finish");
doDelete();
}

// This class is an exception message generated by the agent
static class IDontUnderstand extends Exception
{
    public IDontUnderstand(String msg)
    {
        super(msg);
    }
}

B.3 InstitutionalAgent.java

package lomas;

import jade.content.Concept;
import jade.content.ContentElement;
import jade.content.ContentManager;
import jade.content.lang.sl.SLCodec;
import jade.content.ontology.Ontology;
import jade.content.ontology.OntologyException;
import jade.content.basic.Action;
import jade.core.AID;
import jade.core.Agent;
import jade.core.behaviours(SenderBehaviour);
import jade.domain.DFService;
import jade.domain.FIPAAgentManagement.DFAgentDescription;
import jade.domain.FIPAInteractionProtocol;
import jade.lang.acl.ACLMessage;
import jade.lang.acl.MessageTemplate;
import jade.proto.SimpleAchieveREResponder;
import java.util.*;
import lomas.MonitorAgent.IDontUnderstand;
import lomas.MonitorAgent.MonitorResponderBehaviour;
import lomas.ontology.*;

/**
 * The institutional agent will send information of the
 * institution to all logic agents in the system.
 */
public class InstitutionalAgent extends Agent{
private Codec language;
private Ontology ontology;
private ArrayList<AID> agentList = new ArrayList<AID>();
private int numberAgents = 0;

// Set of classes required by the program
private ASPTools aspTools;
private StringTools stringTools = new StringTools();
private FileWriterTools fileTools = new FileWriterTools();
private String institutionURL;

protected void setup(){
// Takes the arguments passed to the agent and sets the
// solver and asp program
// based on it
Object[] args = getArguments();
institutionURL = args[0].toString();
doWait(1000);
System.out.println(getLocalName() + " Starting Institutional Agent");
// Sets the agents information
// Not changed from original agent
language = new SLCodec();
ontology = LomasOntology.getInstance();

ContentManager manager = getContentManager();
manager.registerLanguage(language);
manager.registerOntology(ontology);
// register service description

DFAgentDescription description = new
DFAgentDescription();
description.addLanguages(language.getName());
description.addOntologies(ontology.getName());
description.addProtocols(InteractionProtocol.FIPA_REQUEST);
description.setName(getAID());

ServiceDescription servicedesc = new
ServiceDescription();
servicedesc.setName(getLocalName());
servicedesc.setType("INSTITUTIONAL_AGENT");

// the service has a list of supported languages, ontologies
// and protocols for this service.

/service desc.addLanguages(language.getName());
servicedesc.addOntologies(ontology.getName());
servicedesc.addProtocols(InteractionProtocol.FIPA_REQUEST);

description.addServices(servicedesc);

// register with the DF
try{
DFService.register(this, description);
} catch(FIPAException e){
System.err.println(getLocalName() + "; error registering 
with DF, exiting: " + e);
doDelete();
return;
}

aspTools = new ASPTools(getAID());
addBehaviour(new InstitutionResponderBehaviour(this));
}

// Listen to and respond to all incoming events received by the monitor

class InstitutionResponderBehaviour extends
SimpleAchieveREResponder{
public InstitutionResponderBehaviour(Agent agent){
// Calls the constructor of SimpleAchieveREResponder
super(agent, MessageTemplate
and(createMessageTemplate(InteractionProtocol.
FIPA_REQUEST),
MessageTemplate
.and(MessageTemplate.MatchOntology(ontology.getName()),
MessageTemplate.MatchLanguage(language.getName())));
}
/*
 * This extracts the message content and then dispatches the
 * body of the message to a sub-handler.
 */
public ACLMessage prepareResponse(ACLMessage msg) {
  ContentElement content = null;
  //Adds another instance of the behaviour so that we can
  //concurrently deal with another request.
  myAgent.addBehaviour(new
    InstitutionResponderBehaviour(myAgent));
  try {
    try {
      content = myAgent.getContentManager().extractContent(msg);
    } catch (OntologyException e) {
      System.err.println("Error relating to message and
        ontology:" + e);
      throw new IDontUnderstand("(" + msg.toString() + ")");
    } catch (Codec.CodeException e) {
      System.err.println(myAgent.getLocalName() + ": Error
        parsing the message:" + e);
      throw new IDontUnderstand("(" + msg.toString() + ")");
    }
  } catch (IDontUnderstand e) {
    System.out.println(e.getMessage().toString());
  }
  return null;
}
public ACLMessage prepareResultNotification(ACLMessage msg1, ACLMessage msg2) {
  //we always reply completely in the
  //handlePrepareResponse, so there
  //is no need to do anything here.
  return null;
}
private void sendInstInfo() {
  for(int i = 0; i < agentList.size(); i++){
    if(action_content instanceof AgentArray){
      AgentArray temp = (AgentArray)action_content;
      Iterator it = temp.getAllAgentSet();
      while(it.hasNext()) {
        agentList.add(((AID) it.next());
      }
      if(agentList.isEmpty()) System.out.println("There are no
        agents in the system");
      else {
        sendInstInfo();
        startAgents();
      }
    } else if (action_content instanceof AgentsAction){
      //Update the institutions state based on the agents
      //action
      AgentsAction act = (AgentsAction) action_content;
      String newAction = act.getActionPerformed();
      if(!newAction.equals("")) {
        fileTools.updateIns(institutionURL, newAction);
        //Then send an update message to all agents
        sendInstInfo();
      }
    } else {
      System.err.println("Unknown action body class " +
        action_content.getClass().toString());
      throw new IDontUnderstand("(" + msg.toString() + ")");
    }
  }
}
public ACLMessage prepareResultNotification(ACLMessage msg1, ACLMessage msg2) {
  //we always reply completely in the
  //handlePrepareResponse, so there
  //is no need to do anything here.
  return null;
}
//Send the institutional information to all agents
String institution = "";
ACLMessage msg = new ACLMessage(ACLMessage.REQUEST);
msg.setProtocol(InteractionProtocol.FIPAREQUEST);
msg.setLanguage(language.getName());
msg.setOntology(ontology.getName());
msg.setSender(getAID());
msg.addReceiver(agentList.get(i));
InstitutionInfo info = new InstitutionInfo();
info.setSender(getAID());
// gets the answer sets of the program and sends
// them to all agents
ArrayList<String> a = stringTools.justAnswerSets(
    aspTools.aspGenerator("dlv", institutionURL),
    "dlv");
if (a.isEmpty()) System.out.println("There is no institutional state to calculate, sending nothing");
else institution = a.get(0);
info.setInstitutionData(institution);
Action action = new Action(agentList.get(i), info);
try {
    getContentManager().fillContent(msg, action);
} catch (OntologyException e) {
    System.err.println("Error with message: ontology" + e);
}
try {
    getContentManager().fillContent(msg, action);
} catch (Codec.CodecException e) {
    System.err.println("Error with message: encoding" + e);
}
addBehaviour(new SenderBehaviour(this, msg));

B.4 ASPTools.java

package lomas;
import jade.core.AID;
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.util.ArrayList;
import java.util.StringTokenizer;
/
∗
∗ Is a helper class contain tools for use with ASP
∗
∗/
public class ASPTools {

    private void startAgents()
    {
        Random ran = new Random();
        // Picks a random number from the array
        int starter = ran.nextInt(agentList.size() - 1);
        ACLMessage msg = new ACLMessage(ACLMessage.REQUEST);
        msg.setProtocol(InteractionProtocol.FIPAREQUEST);
        msg.setLanguage(language.getName());
        msg.setOntology(ontology.getName());
        msg.setSender(getAID());
        msg.addReceiver(agentList.get(starter));
        StarterAgent start = new StarterAgent();
        start.setStart(true);
        Action action = new Action(agentList.get(start),
                                  start);
        try {
            getContentManager().fillContent(msg, action);
        } catch (OntologyException e) {
            System.err.println("Error with message: ontology" + e);
        }
        try {
            getContentManager().fillContent(msg, action);
        } catch (Codec.CodecException e) {
            System.err.println("Error with message: encoding" + e);
        }
        addBehaviour(new SenderBehaviour(this, msg));
    }

    private void startAgents()
    {
        Random ran = new Random();
        // Picks a random number from the array
        int starter = ran.nextInt(agentList.size() - 1);
        ACLMessage msg = new ACLMessage(ACLMessage.REQUEST);
        msg.setProtocol(InteractionProtocol.FIPAREQUEST);
        msg.setLanguage(language.getName());
        msg.setOntology(ontology.getName());
        msg.setSender(getAID());
        msg.addReceiver(agentList.get(starter));
        StarterAgent start = new StarterAgent();
        start.setStart(true);
        Action action = new Action(agentList.get(start),
                                  start);
        try {
            getContentManager().fillContent(msg, action);
        } catch (OntologyException e) {
            System.err.println("Error with message: ontology" + e);
        }
        try {
            getContentManager().fillContent(msg, action);
        } catch (Codec.CodecException e) {
            System.err.println("Error with message: encoding" + e);
        }
        addBehaviour(new SenderBehaviour(this, msg));
    }

    // Sends a message to a random agent telling them to be the first agent to start the
    // system
    //private void startAgents(){
    //    Random ran = new Random();
    //    // Picks a random number from the array
    //    int starter = ran.nextInt(agentList.size() - 1);
    //    ACLMessage msg = new ACLMessage(ACLMessage.REQUEST);
    //    msg.setProtocol(InteractionProtocol.FIPAREQUEST);
    //    msg.setLanguage(language.getName());
    //    msg.setOntology(ontology.getName());
    //    msg.setSender(getAID());
    //    msg.addReceiver(agentList.get(starter));
    //    StarterAgent start = new StarterAgent();
    //    start.setStart(true);
    //    Action action = new Action(agentList.get(starter),
    //                              start);
    //    try {
    //        getContentManager().fillContent(msg, action);
    //    } catch (OntologyException e) {
    //        System.err.println("Error with message: ontology" + e);
    //    }
    //    try {
    //        getContentManager().fillContent(msg, action);
    //    } catch (Codec.CodecException e) {
    //        System.err.println("Error with message: encoding" + e);
    //    }
    //    addBehaviour(new SenderBehaviour(this, msg));
    //}

    // Is the AID of the current agent private AID me;
    // Institutional Information file

}
private String institutionURL = "";
// The array that stores the output language
private ArrayList<String> filter = new ArrayList<String>();

// List of atoms that the agent can pass to other agents
private String filterList = "a,b,c,d,e,f,g,h,i,cdrom";
// Just creates a copy of string tools & file tools that can be used to generate files
private StringTools stringTools = new StringTools();
private FileWriterTools fileTools = new FileWriterTools();

// Constructor
public ASPTools(AID agent) {
    me = agent;
}

// Helper functions
public void setInstURL(String url) {
    institutionURL = url;
}
public String getInstURL() {
    return institutionURL;
}

/* Receives the location of an ASP program and the solver used to compute the answer sets */
public String aspGenerator(String solver, String programURL) {
    String output = null;
    String processInputLparse = null;
    String cmd = null;
    String progURL = "";
    if (institutionURL.equals("")) progURL = programURL;
    else progURL = programURL + "\n" + institutionURL;
    // Depending on the Solver it selects the correct details
    if (solver.equals("smodels")) {
        cmd = "lparses " + progURL;
        cmd2 = "smodels_o"
    } else if (solver.equals("clasp")) {
        cmd = "lparse_o " + progURL;
        cmd2 = "clasp_0"
    } else if (solver.equals("dlv")) {
        cmd = "dlv-silent " + progURL;
    } else if (solver.equals("oct")) {
        cmd2 = "oct_c_o"
    } else if (solver.equals("oct") || solver.equals("oct") || solver.equals("oct") || solver.equals("oct")) {
        processInputLparse = fileTools.getOCLP(programURL);
    } else {
        System.out.println("Cannot find specified solver, using Default : dlv");
        cmd = "dlv-silent " + progURL;
    }
    Process p, p2;
    if (cmd != null) {
        try {
            // Runs the process specified depending on the solver
            p = Runtime.getRuntime().exec(cmd);
            // Reader the information outputted by the process
            InputStreamReader cin = new InputStreamReader(p.getInputStream());
            BufferedReader in = new BufferedReader(cin);
            InputStreamReader cerr = new InputStreamReader(p.getErrorStream());
            BufferedReader err = new BufferedReader(cerr);
            // Stores all of the information outputted as a string
            StringBuffer testing = new StringBuffer(); // If the solver is dlv then the string is the answer sets
            String line = null;
            while ((line = in.readLine()) != null) {
                testing.append(line + "\n" + "\n" + "\n" + "\n" + "\n" + "\n" + "\n" + "\n" + "\n" + "\n" + "\n" + "\n"
            }
        } // if not the the string is just a grounded program and must be
// passed to the solver
if (cmd2 != null) {
    processInputLparse = testing.toString();
} else {
    output = testing.toString();
} in.close();

String line2 = null;
while ((line2 = err.readLine()) != null) {
    System.out.println(line2);
} err.close();
} catch (IOException e) {
    e.printStackTrace();
}
}

try {
    // if the solver used lpars to ground it then this runs the relevant solver
    // (smodels/clasp) to compute the answer sets
    // All code is the same as above
    if (cmd2 != null) {
        p2 = Runtime.getRuntime().exec(cmd2);
        if (processInputLparse != null) {
            BufferedWriter processInputWriter = new BufferedWriter(new OutputStreamWriter(p2.getOutputStream()));
            processInputWriter.write(processInputLparse);
            processInputWriter.close();
            InputStream pin2 = p2.getInputStream();
            InputStreamReader cin2 = new InputStreamReader(pin2);
            BufferedReader in2 = new BufferedReader(cin2);
            InputStream perr2 = p2.getInputStream();
            InputStreamReader cerr2 = new InputStreamReader(perr2);
            BufferedReader err2 = new BufferedReader(cerr2);

            StringBuffer testing2 = new StringBuffer(); String line3 = null;
            while ((line3 = in2.readLine()) != null) {
                testing2.append(line3 + "\n");
            }

            output = testing2.toString();
in2.close();

            String line4 = null;
            while ((line4 = err2.readLine()) != null) {
                System.out.println(line4);
            } err2.close();
        } // Returns the answer sets as a string
        return output;
    } catch (IOException e) {
        e.printStackTrace();
        return null;
    }
}

// Compares the inputed answer set to the current one to check for any contradictions
// if a contradiction appears it is passed to the dealContradiction method
public String checkContradictions(String currentInput, String aspProg, String aspSolver, FileWriterTools fileWriter, AID senderAgent, boolean trustMe) {
    ArrayList<String> program = fileWriter.oldProg;
    ArrayList<String> newAS = new ArrayList<String>();

    StringTokenizer tokens = new StringTokenizer(currentInput, " , ");
    String[] currInputArray = new String[tokens.countTokens()];
    int i = 0;
    while (tokens.hasMoreTokens()) {
        currInputArray[i] = tokens.nextToken().trim();
        i++;
    }

    for (int i3 = 0; i3 < currInputArray.length; i3++) {
        String ans = currInputArray[i3];
        boolean notConflicting = false;
        boolean check = false;
        for (int i2 = 0; i2 < program.size(); i2++) {
            String checking = "";
            //...
if (program.get(i2).length() > 0) checking =
    program.get(i2).substring(0, 1);
if (ans.equals(checking)) {
    check = true;
    // If an atom is in the program but when the
    // asp program is run without the letter and the
    // atom is not in the answer set then the values
    // contradict
    StringBuffer buffered = new StringBuffer();
    for (int num = 0; num < currInputArray.length; num++) {
        if (!currInputArray[num].equals(currInputArray[i3])) {
            if (buffered.length() <= 0)
                buffered.append(currInputArray[num]);
            else
                buffered.append(",", +
                currInputArray[num]);
        }
    }
    // Generates the answer sets with the input minus the atom
    // being checked
    String newInput = buffered.toString();
    String newASPProg = fileWriter.appendToFile(aspProg, newInput, aspSolver);
    String output = aspGenerator(aspSolver, newASPProg);
    fileWriter.deleteFile(newASPProg);
    ArrayList<String> answerSetList =
        stringTools.justAnswerSets(output, aspSolver);
    // If it is then there is no contradiction, if it is not
    // then there is
    for (int arrayFor = 0; arrayFor < answerSetList.size(); arrayFor++) {
        StringTokenizer tokenedAS = new
            StringTokenizer(answerSetList.get(arrayFor), " ", "");
        while (tokenedAS.hasMoreElements()) {
            if (!tokenedAS.nextToken().trim().equals_
                (currInputArray[i3]))) {
                notConflicting = true;
                break;
            }
        }
        if (!notConflicting) break;
    }
    // If there is a contradiction then deal with it
    If (!notConflicting && check) {
        String conflict = currInputArray[i3];
        if (!conflict.equals("")) {
            String resolvedItem = dealContradiction(conflict, senderAgent, trustMe);
            newAS.add(resolvedItem);
        }
        // Otherwise just add the atom to the array
        else
            newAS.add(currInputArray[i3]);
    } // Recreates the input string to include the
    // contradictions
    StringBuffer buffer = new StringBuffer();
    for (int x = 0; x < newAS.size(); x++) {
        if (!newAS.get(x).equals("")) {
            if (buffer.length() <= 0) buffer.append(newAS.get(x));
            else
                buffer.append(",", + newAS.get(x));
        }
    }
    return buffer.toString();
}
// Selects whether to believe its own information or that
// of another agent
private String dealContradiction(String contradiction, AID contradictingAgent, boolean
    trustMe) {
    // If the user specifies to trust their own data over
    other agents then
    // return the true or false depending on what you say it
    // is otherwise
    // return what the other agent believes
    String resolution;
    if (trustMe) {
        if (!contradictingAgent.equals(me)) resolution = "";
        else resolution = contradiction;
    } else {
        if (!contradictingAgent.equals(me)) resolution = "";
        else resolution = contradiction;
    }
    return resolution;
public String filterAnswerSets(String answerSets) {
    // Split up the answer set into an array and check through the filter list
    // removing any items in the answer set that appear in the filter list
    // as these cannot be sent to the other agents.
    getFilter();
    StringTokenizer tokens = new StringTokenizer(answerSets, ",");
    ArrayList<String> asArray = new ArrayList<String>();
    int i = 0;
    while (tokens.hasMoreTokens()) {
        asArray.add(tokens.nextToken().trim());
        i++;
    }
    // Check all of the atoms in the answer set and compare them
    // to the list of atoms allowed to be sent
    for (int x = 0; x < asArray.size(); x++) {
        String check = asArray.get(x);
        boolean found = false;
        for (int x2 = 0; x2 < filter.size(); x2++) {
            if (check.equals(filter.get(x2))) {
                found = true;
                break;
            }
        }
        if (!found) asArray.set(x, "");
    }
    // Buffer converts the array of items into an answer set
    StringBuffer buffer = new StringBuffer();
    for (int x3 = 0; x3 < asArray.size(); x3++) {
        if (buffer.length() <= 0) buffer.append(asArray.get(x3));
        else buffer.append(" , " + asArray.get(x3));
    }
    return buffer.toString();
}

public void setFilter() {
    StringList tokens = new StringList(filterList, ",");
    while (tokens.hasMoreTokens()) {
        filter.add(tokens.nextToken().trim());
    }
}

B.5 FileWriterTools.java

package lomas;

import jade.core.AID;

import java.io.*;
import java.util.ArrayList;
import java.util.StringTokenizer;

/**
 * Is a helper class contain tools used for accessing files
 */

public class FileWriterTools {

    private StringTools stringTools = new StringTools();
    private int filesCreated = 0;

    public ArrayList<String> oldProg = new ArrayList<String>();

    private String orderList;

    // Stores the old program as an array of strings
    public void setProg(String fileName, String solver) {
        try {
            BufferedReader input = new BufferedReader(new FileReader(fileName));
            String line = null;
        }
while((line = input.readLine()) != null){
    oldProg.add(line);
}
input.close();

//Last line of the program is need to set the ordering later, but only for OCT
if(solver.equals("oct")) orderList =
    oldProg.get((oldProg.size()-1)) + "<input";
} catch (FileNotFoundException e) {
    System.out.println("Program with filename: " + fileName + " was not found.");
e.printStackTrace();
return null;
}

//used by the institutional agent to update its program
public void updateIns(String institutionURL, String newActions){
    try{
        BufferedWriter out = new BufferedWriter(new FileWriter(institutionURL , true));
        StringTokenizer progList = new StringTokenizer(newActions , " , ");
        //using a function in String Tools and copies these to the correct file
        while(progList.hasMoreTokens()){
            out.write(progList.nextToken().trim() + ",");
            out.newLine();
        }
        out.close();
    } catch (IOException e) {
        System.out.println("Error updating institution");
e.printStackTrace();
    }
}

//Gets a copy of the OCLP program for use with the ASP Tools
public String getOCLP(String filename){
    StringBuffer tester = new StringBuffer();
    try{
        BufferedReader input = new BufferedReader(new FileReader(filename));
        String line = null;
        while((line = input.readLine()) != null){
            tester.append(line + "\n"");
        }
        tester.append("\n\n" + fileName + " was not found.");
        e.printStackTrace();
    } catch (IOException e) {
        System.out.println("Problem with_filename: " + fileName + " was not found.");
        e.printStackTrace();
    }
    return tester.toString();
}

//Creates a new file with the information from the institutional agent
public String writeInst(String prog, AID agent){
    try{
        //Creates a new file or overwrites the current one
        String newFileName = agent.getLocalName() + "Inst.lp";
        BufferedWriter out = new BufferedWriter(new FileWriter(newFileName , true));
        StringTokenizer progList = new StringTokenizer(prog , " , ");
        //using a function in String Tools and copies these to the correct file
        while(progList.hasMoreTokens()){
            out.write(progList.nextToken().trim() + ",");
            out.newLine();
        }
        out.close();
    } catch (IOException e) {
        System.out.println("Problem with_filename: " + fileName + " was not found.");
        e.printStackTrace();
    }
    return null;
}
```java
out.close();
return newFileName;
}
catch (IOException e) {
System.out.println("Error creating institutional_index\nfile for " +
agent.getLocalName());
e.printStackTrace();
return null;
}

// Outputs all of the information to the filename given
public String appendToFile(String fileName, String input, String solver) {
try {
    StringTokenizer newFile = new StringTokenizer(fileName, ".");
    String newFileName = newFile.nextToken() + filesCreated + "," +
    newFile.nextToken();
    // Used to generate a random filename so that the original file is not
    overwritten
    filesCreated++;
    // crops the answer sets so that they are all of the same form
    String[] newRules = stringTools.clearAnswerSets(input);
    ArrayList<String> newProg = new ArrayList<String>();
    // OCT programs are treated differently to other asp programs because of their structure
    if (solver.equals("oct")) {
        for (int x = 0; x < (oldProg.size() - 1); x++) {
    newProg.add(oldProg.get(x));
        }
        newProg.add("component\ninput .");
        for (int x = 0; x < newRules.length; x++) {
    if (!newRules[x].equals("")) newProg.add(newRules[x]);
        }
        newProg.add("\n")
        newProg.add("\n\n");
        newProg.add(orderList);
    } else {
    newProg.addAll(oldProg);
    for (int x = 0; x < newRules.length; x++) {
```
//Takes the inputted string and splits if using the new line character
//and then returns the split string as a string array
private String[] splitAnswerSets(String answerSets) {
    StringTokenizer asp = new StringTokenizer(answerSets, "\
    ");
    String[] newASPList = new String[asp.countTokens()];
    int i = 0;
    while (asp.hasMoreTokens()) {
        newASPList[i] = asp.nextToken();
        i++;
    }
    return newASPList;
}

//Removes the items that this agent made true previously, if there were none then just return the current input
public String removeTrue(String previousAS, String previousInput, String currentInput) {
    String newInput = currentInput;
    //Tokenize all of the received strings
    StringTokenizer prevAS = new StringTokenizer(previousAS, ",");
    StringTokenizer prevInput = new StringTokenizer(previousInput, ",");
    StringTokenizer currInput = new StringTokenizer(currentInput, ",");
    //Set up the arrays
    String[] prevASArray = new String[prevAS.countTokens()];
    String[] currInputArray = new String[currInput.countTokens()];
    int i2 = 0;
    //Create an array of all the tokens from the previous answer set
    while (prevAS.hasMoreTokens()) {
        prevASArray[i2] = prevAS.nextToken().trim();
        i2++;
    }
    int i3 = 0;
    //Create a string of the new input
    StringBuffer newInputBuff = new StringBuffer();
    while (currInput.hasMoreTokens()) {
        currInputArray[i3] = currInput.nextToken().trim();
        i3++;
    }
    //Check for all items in the previous input
    //remove them from the previous answer set, leaving you with
    //all of the items made true last time
    int i = 0;
    while (prevInput.hasMoreTokens()) {
        String find = prevInput.nextToken().trim();
        for (int x = 0; x < prevASArray.length; x++) {
            if (find.equals(prevASArray[x])) {
                prevASArray[x] = "";
                break;
            }
        }
        i++;
    }
    //Cycle through all of the items left in the previous answer set and remove them
    //from the current input
    for (int x2 = 0; x2 < prevASArray.length; x2++) {
        String find = prevASArray[x2];
        if (!find.equals("")) {
            for (int x3 = 0; x3 < currInputArray.length; x3++) {
                if (find.equals(currInputArray[x3])) {
                    currInputArray[x3] = "";
                    break;
                }
            }
        }
    }
    //Create an array of all the tokens from the current input
    while (prevAS.hasMoreTokens()) {
        prevASArray[i2] = prevAS.nextToken().trim();
        i2++;
    }
    int i4 = 0;
    //Create a string of the new input
    StringBuffer newInputBuff = new StringBuffer();
    for (int x4 = 0; x4 < currInputArray.length; x4++) {
        if (currInputArray[x4].equals("")) {
            if (newInputBuff.length() <= 0)
                newInputBuff.append(currInputArray[x4]);
        else newInputBuff.append(" , " + currInputArray[x4]);
    }
newInput = newInputBuff.toString();
return newInput;
}
//Depending on the asp solver it adjusts their output so
//that all answersets are of the form
//a, b, c, etc. so that all agents can communicate
//regardless of the solver they are using
public ArrayList<String> justAnswerSets(String asp, String solver){
ArrayList<String> list1 = new ArrayList<String>();
//Splits the answer sets received into individual lines
String[] list2 = splitAnswerSets(asp);
//Cycles through all lines received
for(int x = 0; x<list2.length; x++){
int i = 0;
if(solver.equals("smodels")){
String newRule = "";
String c;
//Only if the first 6 characters of the line are answer
//are we interested in the line
//so if they are not then simply skip to the next one
if(list2[x].length()<6) c = "no";
else c = (list2[x].substring(i, i+6)).trim();
if(c.equals("Answer")){
//If line is answer then look at the next line for the
//answerset
//If using smodels then skip the first 14 chars as they are
//always Stable Model:
int j = 14;
while(j<list2[x+1].length()){
String cl = list2[x+1].substring((j-14)+14, (j-14)+15);
//Skip all none alphabetic characters, all alphabetic
//characters are added together using a ,
if(cl.equals("(") || cl.equals("") || cl.equals("\"") ||
cl.equals("\") || cl.equals("\n") || cl.equals("null")){
}
else{
if(newRule.equals("")) newRule = cl;
else newRule = newRule + "," + cl;
}
++j;
}
}
else if(solver.equals("clasp")){
String newRule = "";
String c;
if(list2[x].length()<6) c = "no";
else c = (list2[x].substring(i, i+6)).trim();
if(c.equals("Answer")){
int j = 0;
//If line is answer then look at the next line for the
//answerset
//If using clasp then skip the first 14 chars as they are
//always answer:
while(j<list2[x+1].length()){
String cl = list2[x+1].substring((j-14)+1, (j-14)+2);
//Skip all none alphabetic characters, all alphabetic
//characters are added together using a ,
if(cl.equals("(") || cl.equals("") || cl.equals("\"") ||
cl.equals("\") || cl.equals("\n") || cl.equals("null")){
}
else{
if(newRule.equals("")) newRule = cl;
else newRule = newRule + "," + cl;
}
++j;
}
}
else if(solver.equals("dlv")){
String newRule = "";
//splits the answer set up and cycles through the tokens
//to ensure
//all unnecessary characters are removed
StringTokenizer tok = new StringTokenizer(list2[x], ",\n");
while(tok.hasMoreTokens()){
}
```java
String check = tok.nextToken().trim();
String newOne = "";
int x1 = 0;
while(x1 < check.length()){
    String c = check.substring(x1, x1+1);
    if(c.equals("{") || c.equals("")) || c.equals(".") ||
    c.equals(",") || c.equals(""") || c.equals(null)){
        }else{
            newOne += c;
        }
        x1++;
    }
    if(newRule.equals("")) newRule = newOne;
    else newRule = newRule + "," + newOne;
    //Add this answer set to the array
    list1.add(newRule);
    }
    //The output of oct is always Stable Model: "answer sets"
    //so if the line is not blank then remove "Stable" and
    //"Model:" and
    //what remains are answer sets seperated by spaces
    else if(solver.equals("oct")){
        String newRule = "";
        StringTokenizer tokens = new StringTokenizer(list2[x], ",");
        String[] a1 = new String[tokens.countTokens()];
        int i = 0;
        while(tokens.hasMoreTokens()){
            a1[i] = tokens.nextToken().trim();
            i++;
        }
        //Calls the sort function of the array class
        Arrays.sort(a1);
        StringBuffer buffer = new StringBuffer();
        for(int x = 0; x < a1.length; x++){
            if(buffer.length()<=0) buffer.append(a1[x]);
            else buffer.append("", + a1[x]);
        }
        return buffer.toString();
    }
```