A Multi-Agent Platform using Ordered Choice Logic Programming

Marina De Vos*, Tom Crick, Julian Padget, Martin Brain,
Owen Cliffe, and Jonathan Needham

Department of Computer Science
University of Bath
Bath BA2 7AY, UK
Email: {mdv,tc,jap,mjb,occ,jmn20}@cs.bath.ac.uk

Abstract. Multi-agent systems (MAS) can take many forms depending on the characteristics of the agents populating them. Amongst the more demanding properties with respect to the design and implementation is how these agents may individually reason and communicate about their knowledge and beliefs, with a view to cooperation and collaboration. In this paper we present a deductive reasoning multi-agent platform using an extension of answer set programming (ASP). We show that it is capable of dealing with the specification and implementation of the system's architecture, communication and the individual reasoning capacities of the agents. Agents are represented as Ordered Choice Logic Programs (OCLP) for modelling their knowledge and reasoning capacities, with communication between the agents regulated by uni-directional channels transporting information based on their answer sets. For the implementation of our system we combine the extensibility of the JADE framework with the efficiency of the OCT front-end to the Smodels answer set solver. The power of this approach is demonstrated by a multi-agent system reasoning about equilibria of extensive games with perfect information.

1 Introduction

The emergence of deductive reasoning agents over traditional symbolic AI has led to the development of theorem proving within an agent framework. The principal idea is to use logic formalisms to encode a theory stating the best action to perform in any given situation. Unfortunately, the problems of translating and representing the real world in an accurate and adequate symbolic description are still largely unsolved (although there has been some encouraging work in the area, for example, the OpenCyc project[1].

Answer set programming (ASP) [2] is a formal, logical language designed for declarative problem solving, knowledge representation and real world reasoning. It represents a modern approach to logic programming based on analysis of which constructs are needed in reasoning applications rather than implementing subsets of classical or other abstract logical systems.

* This work was partially supported by the European Fifth Framework Programme under the grant IST-2001-37004 (WASP).

M. Baldoni, U. Endriss, A. Omicini, and P. Torroni (eds.)
DALT 2005, Third International Workshop, Utrecht, July 25th, 2005
http://www.doc.ic.ac.uk/~ue/DALT-2005/

In this paper, we present a formalism for multi-agent systems in which the agent use answer set programming for representing their reasoning capabilities. Such systems are useful for modelling decision-problems; not just the solutions of the problem at hand but also the evolution of the beliefs of and the interactions between the agents can be characterised.

A system of logic programming agents consists of a set of agents connected by means of uni-directional communication channels. To model a single agent's reasoning, we use Ordered Choice Logic Programs [3], an extension of answer set programming that provides for explicit representation of preferences between rules and dynamic choice between alternatives in a decision. Agents use information received from other agents as a guidance for the reasoning. As long as the information received does not conflict with the decision(s) the agent has to make (i.e. its goal(s)) this new knowledge will be accepted. If conflict arises, the agent will always prefer its own alternatives over the ones presented by other agents. In this sense, one could say that our agents act strategically and rational in the game theoretic sense ([4]). The knowledge an agent possesses is captured by its answer set semantics. Part of this knowledge is shared with agents listening on the outgoing channels. This is regulated by filters that for each connecting agent decide what sort of information this agent is allowed to receive. The semantics of the whole system corresponds to a stable situation where no agent, with stable input, needs to change its output.

Our implementation uses the JADE platform[5] to provide basic communications and agent services, an ontology developed in Protégé[6] and OCT [3, 7] to model, maintain and reason about the agent's beliefs and knowledge.

The rest of the paper is organised as follows: Section 2 explains the advantages of using logic programming for modelling agent behaviour and details why we believe ASP is better suited for this than Prolog, the traditional choice for logic programming. A brief overview of OCLP is provided in Section 3. Sections 4 and 5 present the theoretical multi-agent framework, LAIMAS, and its implementation, JOF. Game theory, one of the possible application areas of LAIMAS/JOF is detailed in Section 6. This paper ends with a discussion of future developments in ASP that are beneficial to the agent community and the future work on the presented system.

2 Why Answer Set Programming?

A consistent theme in agent research is the desire to have agents behave in an 'intelligent' manner. One component of this is the ability to reason: perform logical inference, handle combinatorial constraints and being able to handle complex, logical queries over a large search domain. These actions are simple to express and implement in declarative logic programming formalisms. By using these tools, the developer of an agent system can focus on the reasoning and 'thought' processes of the agent rather than being bogged down in how to implement them. Using existing logical formalisms rather than an ad-hoc systems also brings a greater degree of robustness and certainty to the agent's reasoning, i.e. because it is possible or easier to prove and verify the behaviour of participating agents. Finally, the availability of a number of powerful and mature implementations contributes to reduced development time.

One common question asked of researchers working on non-monotonic logic programming systems such as ASP is: "Prolog has been around for many years and is a mature technology, why not just use that?". We argue that Prolog has a number of limitations at the conceptual and design levels that make it unsuitable for many knowledge representation and 'real world' reasoning tasks.

Negation tends to create problems in logic programming languages and Prolog is no exception. A variety of different mechanisms for computing when the negation of a predicate is true and a variety of different intuitions of what this means have been proposed[8]. The most common approach is to compute negation as failure, i.e. $not(p)$ is true if p cannot be proved using the current program, and to characterise this as classical negation, i.e. every proposition is either true or false and cannot be both. This combination creates a problem when using Prolog to model real world reasoning, which is referred to as the "closed world assumption". By equating negation as failure with classical negation anything that cannot be proved to be true is known to be false, essentially assuming that everything that is known about the world is contained in the program.

In some tasks this is acceptable, as the program contains all complete information about the situation in models, whilst in the context of agents, requiring perfect information is an unreasonable and unnecessarily restrictive constraint. For example an agent participating in an auction needs to be able to differentiate between not knowing whether to bid and knowing not to bid!

In contrast, the semantics of ASP naturally gives rise to two different methods of calculating negation, negation as failure and constraint based negation. Negation as failure, (i.e. we cannot prove p to be true) is characterised as epistemic negation[1] (i.e. we do not know p to be true). Constraint-based negation introduces constraints that prevent certain combinations of atoms from being simultaneously true in any answer set. This is characterised as classical negation as it is possible to prevent a and $\neg a$ both being simultaneously true, a requisite condition for capturing classical negation. This is a significant advantage in some reasoning tasks as it allows reasoning about incomplete information. More importantly, the closed world assumption is not present in ASP, as negation as failure is not associated with classical negation.

In the extension of ASP that we propose to use for modelling agents, we only allow implicit negation coming from the modelling decisions (i.e. you have decide upon exactly one alternative when forced to make a choice). However, as described in [9], it is perfectly possible to embed any form of negation (classical negation or negation as failure) using the order mechanism. In other words, the ordering or rules replaces use of negation, making OCLP a suitable tool to model the exact type of negation one wishes to use. In terms of complexity, having both concepts (negations and order) is no different to having just one.

One key difference between Prolog and ASP is that the semantics of ASP clearly give rise to multiple possible world views[2] in which the program is consistent. The number and composition of these varies with the program. Attempting to model the same ideas in Prolog can lead to confusion as the multiple possible views may manifest themselves differently, depending on the query asked. In ASP terms, Prolog would an-

[1] For a full discussion of such issues, see [2].

[2] The exact formal declarative meaning of answer sets is still under debate[8].

swer a query on a as true if there is at least one answer set in which a is true. However, there is no notion of 'in which answer set is this true'. Thus, a subsequent query on b might also return true, but without another query it would not be possible to infer if a and b could be simultaneously true.

3 Ordered Choice Logic Programming

OCLP ([3]) was developed as an extension of ASP to reason about decisions and preferences. This formalism allows programmers to explicitly express decisions (in the form of exclusive choices between multiple alternatives) and situation dependent preferences. We explain the basics[3] of OCLP by means of a simplified grid-computing situation:

Example 1. Suppose a grid-computing agent capable of performing two tasks which are mutually exclusive because of available resources. Your agent has permissions to use three clusters ($cluster_1$,$cluster_2$, $cluster_3$). Because of prices, computing power and reachability, the agent prefers $cluster_3$ over $cluster_2$ and $cluster_2$ over $cluster_1$. However, in order to perform task two, she needs access to both a database and a mainframe. The former can only be provided by $cluster_1$, while the latter is only available from $cluster_2$. The above information leads to two correct behaviours of our agent:

- performing task one, the agent uses $cluster_3$
- executing task two, the agent requires access to clusters 1 and 2 in order to use the required database and mainframe.

To model such behaviour, we represent the agent as an answer set program capable of representing preference-based choices between various alternatives. The preference is established between components, groups of rules. Components are linked to each other by means of strict order denoting the preference relation between them. Information flows from less specific components to the more preferred ones until a conflict among alternatives arises, in which case the most specific one will be favoured. Alternatives are modelled by means of choice rules (rules that imply exclusive disjunction). In other words, an OCLP P is a pair $\langle \mathcal{C}, \prec \rangle$ where $\mathcal{C}$ is a collection components, containing a finite number of (choice) rules, and strict order relation $\prec$ on $\mathcal{C}$. We will use $\preccurlyeq$ to denote the reflexive closure of $\prec$. For the examples, we represent an OCLP as a directed acyclic graph (DAG), in which the nodes are the components and the arcs represent the relation "$\prec$". For two components $C1$ and C_2, $C_1 \prec C_2$ is represented by an arrow from going from C_1 to C_2, indicating that information from C_1 takes precedence over C_2.

Example 2. The agent mentioned in Example 1 can be easily modelled using an OCLP, as shown in Figure 1. The rules in components P_1, P_2 and P_3 express the preferences between the clusters when only computing power is required. The rules in P_4 indicate the grid computing problem and the required choice between the possible alternatives. In component P_5, the first rule states the goals of the agent in terms of the tasks. The third and the fourth rules specify the resources, apart from computing power, needed for a certain tasks. The last two rules express the availability of the extra resources in a cluster.

[3] Full details can be found in [3].

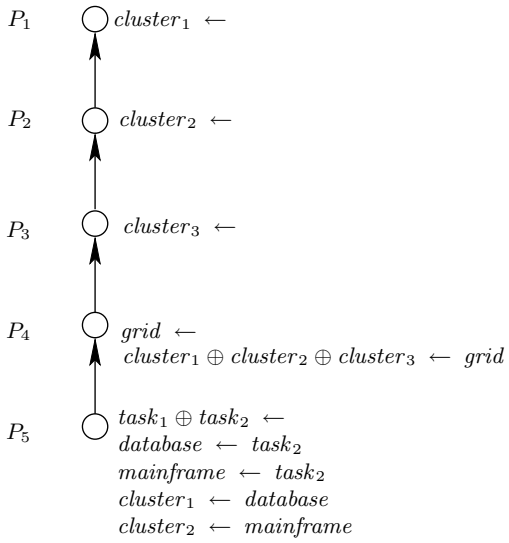

Fig. 1. The GridComputing Agent from Example 2

The semantics of an OCLP is determined by means of *interpretations*, i.e. sets of atoms that are assumed to be true (with atoms not in the interpretation being unknown).

Given an interpretation, we call a rule $A \leftarrow B$ *applicable* when the precondition B, the *body*, is true (all the elements in the body are part of the interpretation). A rule is *applied* when it is applicable and when the consequence A, called the *head*, contains exactly one true atom. The latter condition is reasonable as rules with more than one element in their head represent decisions where only one alternative may be chosen.

Using interpretations we can reason about the different alternatives. Two atoms are considered to be alternatives with respect to an interpretation if and only if a choice between them is forced, i.e. there exists a more specific and applicable choice rule with a head containing (at least) the two atoms. So given an atom a in a component C, we can define the *alternatives* of a in that component C with respect to an interpretation I, written as $\Omega_C^I(a)$, as those atoms that appear together with a in the head of a more specific applicable choice rule.

Example 3. Reconsider Example 2. Let I and J be the following interpretations:

– $I = \{cluster_2, task_1\}$ and $J = \{grid, cluster_1, task_1\}$.

The alternatives for $cluster_2$ in P_2 w.r.t. J are $\Omega_{P_2}^J(cluster_2) = \{cluster_1, cluster_2\}$. W.r.t. I, we obtain $\Omega_{P_2}^I(cluster_2) = \emptyset$, since the choice rule in P_4 is not applicable. When we take P_5 instead of P_2, we obtain w.r.t. J: $\Omega_{P_5}^J(cluster_2) = \emptyset$, since the applicable choice rule is in a less preferred component which makes it irrelevant to the decision process in the current component.

Unfortunately, interpretations are too general to convey the intended meaning of a program, as they do not take the information in the rules into account. Therefore, models

are introduced. The model semantics for choice logic programs, the language used in the components [10], and for ASP is fairly simple: an interpretation is a model if and only if every rule is either not applicable (i.e. the body is false) or applied (i.e. the body is true and the head contains exactly one head atom). For OCLP, taking individual rules into account is not sufficient: the semantics must also consider the preference relation. In cases where two or more alternatives of a decision are triggered, the semantics requires a mechanism to deal with it. When the considered alternatives are all more specific than the decision itself, the decision should be ignored. When this is not the case, the most specific one should be decided upon. If this is impossible, because they are unrelated or equally specific, an arbitrary choice is justified. This selection mechanism is referred to as defeating.

Example 4. Reconsider Example 2. The rule $cluster_1 \leftarrow$ is defeated w.r.t. interpretation $I = \{cluster_2, grid\}$ because of the applicable rule $cluster_2 \leftarrow$.
W.r.t. interpretation $J = \{cluster_1, cluster_2, grid, database, mainframe\}$ we have that the rule $cluster_1 \oplus cluster_2 \oplus cluster_3 \leftarrow grid$ is defeated by the combination of the rules $cluster_1 \leftarrow database$ and $cluster_2 \leftarrow mainframe$.

So we can define a *model* for an OCLP as an interpretation that leaves every rule either not applicable, applied or defeated. Unfortunately, in order to obtain the true meaning of a program, the model semantics tends to be too crude.

For traditional ASP, the Gelfond-Lifschitz transformation or reduct [11] was introduced to remove models containing unsupported assumptions. Interpretations that are the minimal model of their Geldond-Lifschitz transformation are called answer sets, as they represent the true meaning of the program or alternatively the answer to the problem encoded by the program. Therefore, algorithms and their implementations for obtaining the answer sets of a program are often referred to as answer set solvers.

For OCLP, a reduct transformation to obtain the answer sets of our programs is also required. The transformed logic program can be obtained by adding together all components of the OCLP. Then, all defeated rules are taken away, together with all false head atoms of real choice rule (i.e. more than one head atom). The remaining rules with multiple head atoms are transformed into constraints, assuring that only one of them can become true whenever the body is satisfied[4].

Example 5. Reconsider the OCLP from Example 2. This program has two answer sets:
- $M_1 = \{grid, cluster_1, task_2\ database, mainframe, cluster_2\}$ and
- $M_2 = \{cluster_3, grid, task_1\}$,

which matches our agent's intended behaviour.

In [3] it was shown that a bi-directional polynomial mapping exists between ordered choice logic programs and extended logic programs with respect to their answer set semantics. The mapping from OCLPs to normal logic programs is possible by introducing two new atoms for each rule in order to indicate that a rule is applied and defeated. For each rule in the OCLP, we generate a set of rules (the number of rules is equal to the

[4] The relation between those two definitions of reduct clearly convey that order and negation are interchangeable and explains why negation can be easily embedded in OCLP [9].

number of head atoms) in the logic program that become applicable when the original rule is made applied. A rule is also created for each possible situation in which the original rule could become defeated. Finally add one rule that should be fired if the rule is applied and not defeated. Constraints to make sure that the head elements cannot be true at the same time is are not necessary.

The reverse mapping is established by creating an OCLP with three components placed in a linear formation. The least preferred one establishes negation as failure. The middle component contains all the rules from the original program. The most preferred makes sure that for each pair $a, \text{not } a$ only one can be true at any time, without given a a chance to be true without reason in the middle component

These polynomial mappings demonstrate that the complexity of both systems is identical (more information on the complexity aspects can be found in [2]). Having a polynomial mapping to a traditional logic program makes it possible implement a front end to one of the existing answer set solvers like Smodels ([12]) or DLV ([13]). OCT[3] is such a front-end. We will in our agents to compute the knowledge of individual agents and the beliefs they hold about the knowledge of the others.

4 The LAIMAS Framework

In this section we consider systems of communicating agents, where each agent is represented by an OCLP that contains knowledge and beliefs about itself, its goals, the environment and other agents and the mechanism to reason about this.

We assume that agents are fully aware of the agents they can communicate with, i.e. the communication structure is fixed, and that they can communicate by means of passing sets of atoms over uni-directional channels.

A *LAIMA* system is a triple $\langle \mathbf{A}, C, F \rangle$ containing a set of A of agents, a set S of atoms representing the language of the system and a relation C as a subset of $A \times A \times S$ representing the communication channels between the agents and the filter they use when passing information. The filter tells the listening agent which sort of information they can expect, if any. Since this is public information, we have opted for mentioning the information that could be passed in favour of the information that is kept secret. Furthermore, with each agent a we associate an OCLP F_a.

In examples we use the more intuitive representation of a graph. The set S is formed by all the atoms appearing in the OCLP associated with the agents. The filter is shown next to the arrow when leaving the transmitting agent. In order not to clutter the image, we assume that if no filter is present the agent could potentially transmit every atom of S.

Example 6. The system in Figure 2 displays a multi-agent system where seven agents "cooperate" to solve a murder case. Two witnesses, agents $Witness_1$ and $Witness_2$ provide information to the Inspector agent is called to a scene to establish a murder took place. Both witnesses will only pass on information relevant to the crime scene. Information from the witnesses is passed to the $Officer$ agent for further processing. Depending on information provided she can decide to question the three suspect agents. If questioned and when no alibi can be provided, the Officer can accuse the suspect. Of

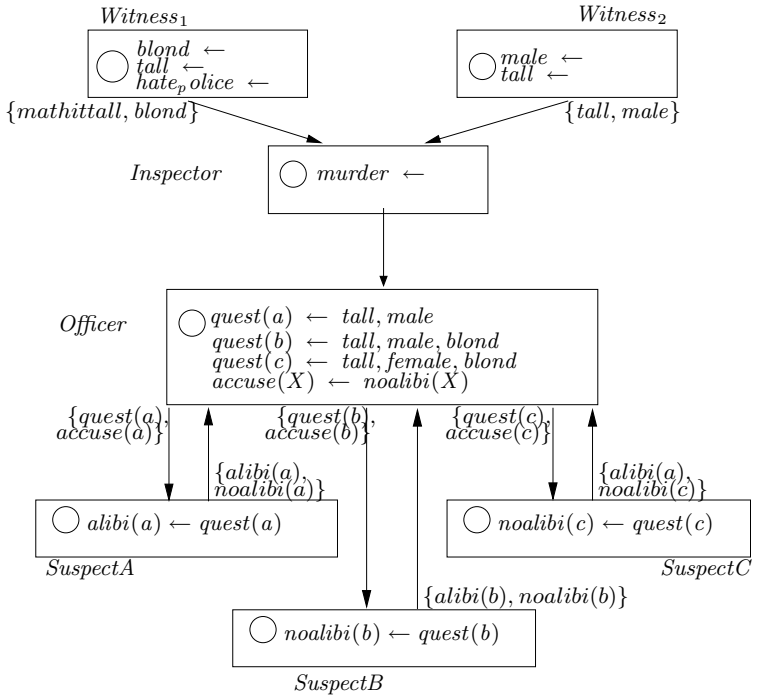

Fig. 2. The Murder LAIMAS of Example 6

course, the Officer will only pass on selective information about the case to the Suspects. The suspects from the side have no intention to tell more than if they have an alibi or not.

For OCLP we defined the notion of interpretation to give a meaning to all the atoms in our program. For our LAIMAS systems we have a number of agents that can hold sets of beliefs which do not necessarily have to be the same. It is perfectly acceptable for two agents (e.g. humans) to agree to disagree. To allow this in our system we need *interpretations* to be functions taking an agent as input and an interpretation of the agent's updated version as output.

Example 7. Consider the Murder LAIMAS of Example 6. Then, the function I with:

- $I(Witness1) = \{hate_p olice, blond, tall\}$,
- $I(Witness2) = \{male, tall\}$,
- $I(Inspector) = \{murder, male, blond, tall\}$,
- $I(Officer) = \{murder, blond, tall, male, quest(a), quest(b), accuse(b), alibi(a), noalibi(b)\}$
- $I(SuspectA) = \{quest(a), alibi(a)\}$
- $I(SuspectB) = \{quest(b), noalibi(b)\}$
- $I(SuspectC) = \{\}$

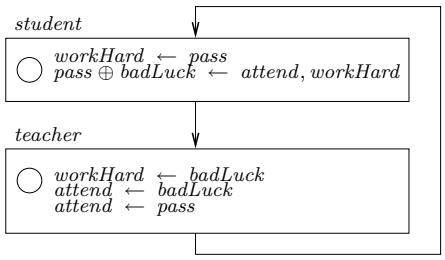

Fig. 3. The Exam LAIMAS of Example 9

is an interpretation for this system.

In the current setting, an agent's output is not only depend on the agent's current beliefs, defined by an interpretation, but also on the recipient of this information. The information sent is determined by the intersection of the agent's belief and the filter used for communication the agent he is sending information to, i.e. $Out_I^b(a) = I(a) \cap F$ with $(a, b, F) \in C$, On the other hand, an agent receives as input the output of all agents connected to its incoming channels, i.e. $In_I(b) = cons(\cup_{(a,b)\in C} Out_I^b(a))$.

An agent reasons on the basis of positive information received from other agents (its input) and its own program that may be used to draw further conclusions, possibly contradicting incoming information. In this framework we made the decision that an agent will attach a higher preference to their own rules rather than to suggestions coming from outside. This can be conveniently modelled by extending an agent's ordered program with an extra "top" component containing the information gathered from its colleagues. This new OCLP is referred to as the updated version of the agent. This way, the OCLP semantics will automatically allow for defeat of incoming information that does not fit an agent's own program, i.e. contradicting the agent's goals.
When modelling rational agents, we assume that agents would only forward information of which they can be absolutely sure. Therefore, it is only sensible to request from agents being modelled as OCLP, to communicate in terms of answer sets.

When an interpretation produces an answer set for each agent's updated version, we call it a *model* for the LAIMAS system. Given that one agent can create multiple answer sets of its updated version, LAIMAS generates an extra source of non-determinism.

Example 8. Consider the Murder LAIMAS of Example 6 and the interpretation I from Example 7. Then I is a model for this LAIMAS and corresponds with the intuition of the problem.

The model semantics provides an intuitive semantics for systems without cycles. Having cycles allow assumptions made by one agent to be enforced by another agent.

Example 9. Consider the LAIMAS in Figure 3 where a student and a teacher discuss issues about the relationship between attending the lecture, working hard, passing the unit or having bad luck. This systems has three models:

– $M(teacher) = M(student) = \emptyset$

– $N(teacher) = N(student) = \{attend, workHard, pass\}$
– $S(teacher) = S(student) = \{attend, workHard, badLuck\}$

Nothing in the OCLP program suggest that an exam has taken place or that the agents are not just hypothesising. Therefore, the only sensible situation is represented by interpretation M.

To avoid such self-sustaining propagation of assumptions, we require that the semantics is the result of a fixpoint procedure which mimics the evolution of the belief set of the agents over time. Initially, all agents start off with empty input from the other agents (agents do not have any dynamic beliefs regarding the other agents). At any stage agents receive the input generated by the previous cycle to update their current belief set. This evolution stops when a fixpoint (i.e. no agent changes the beliefs they had from the previous cycle) is reached. This final interpretation is then called an *answer set*.

More formally, a sequence $I_0, \ldots, I_n$ of interpretations is an *evolution* of a LAIMAS F if for all agents a and $i > 0$ we have that $I_{i+1}(a)$ is an answer set of the updated program of a with the input of the current interpretation, i.e. $In_{I_i}(a)$.
An interpretation I is an *evolutionary fixpoint* of F w.r.t. an interpretation I_0 iff there exists an evolution $I_0, \ldots$ and an integer $i \in \mathbb{N}$ such that $I_j = I_i = I$ for all $j > i$.
An *answer set* of F is an evolutionary fixpoint of F w.r.t. the empty interpretation $I_\emptyset$ (which associates the empty set with each agent).

Thus, in an evolution, the agents evolve as more information becomes available: at each phase of the evolution, an agent updates its program to reflect input from the last phase and computes a new set of beliefs. An evolution thus corresponds to the way decision-makers try to get a feeling about the other participants. The process of reaching a fixpoint boils down to trying to get an answer to the question "if I do this, how would the other agents react", while trying to establish a stable compromise. Note that the notion of evolution is nondeterministic since an agent may have several local models. For a fixpoint, it suffices that each agent can maintain the same set of beliefs as in the previous stage.

Example 10. The LAIMAS of Example 9 has exactly one answer set, namely M, just as we hoped. At iteration 1, both agents will receive no input resulting in producing both empty answer sets as output. This makes that the input in iteration 2 is also empty for both, obviously resulting the same output. Since both iterations are exactly the same, we have reached a fixpoint.

The LAIMAS of Example 6 has also one answer set: the model mentioned in Example 7. The evolution producing this answer set is slightly more interesting. We leave it to reader to construct the evolution in detail. In the first iteration, the witnesses produce their facts which are received by the inspector in the second iteration. In second iteration both witnesses and inspector complete their belief set which is reported to the Officer for use in the third iteration. In this iteration, the Officer will decide which suspects to question. This is done in the fourth iteration. In the fifth iteration, the Officer will solve the crime. SuspectB will be notified of being accused during the sixth iteration.

5 JOF: The LAIMAS Implementation

The theoretical multi-agent architecture LAIMAS described in the previous section has been implemented as the JADE OCLP Framework (JOF). We have opted for using JADE for the MAS architecture, Protégé [6] for the ontology and OCT [3] as the answer set solver.

The choice of working within the JADE framework[5] and the use of the relevant FIPA[14] specifications allows for an architecture based on behaviours. The agents can be implemented at a high-level by defining their behaviours and reactions to events.

An ontology is required for the grounding of the concepts and relationships within JOF and to smooth out the communication between the various agents in the system and to allow easy expansion to different types agents.

JOF is based on a simple client-server model, using fat clients. The implementation has two main agents or roles: the Queen and JOF agents. Using the GAIA methodology of defining roles by four attributes (responsibilities, permissions, activities and protocols), we have defined the collection of behaviours for our two main agents. The Queen contains the graphical user interface for the application and is designed to be the interface between the human user and the MAS. When initialised, the Queen agent will simply start up the user interface, awaiting user input. The Queen is capable of ordering agents to perform a cycle, update and retrieve their information. It also a maintains list of agents that can be edited by the user, along with other administrative actions. The JOF agents are designed to also run autonomously. This distinction is important, as it allows the option of running JOF without the need for a human supervisor. Agents are then able to subscribe to the Queen or to each other, and receive periodic information (i.e. the answer sets).

It is possible to run JOF in two different modes: retained and runtime. Retained mode allows the user to add, remove or modify any of the agents in the JOF environment. All agents running in retained mode are controlled by the Queen agent. Runtime mode provides the autonomous functionality and events to fire upon relevant stages in the OCLP agent life-cycle but also gives means for a developer to alter all of the relevant information of the local agents.

The JOF ontology gives a basis for the representation of different objects and functionality in the JOF environment. For example, the ontology provides for the fundamental of notions like ‘rule’ or ‘OCLP program’. There are three main sections that we can identify in the ontology: OCLP programs, message passing and the knowledge base. The ontology itself is declared hierarchically using the BeanGenerator within Protégé and also encapsulates a number of message classes, such as the answer set broadcast and the Queen’s information request and retrieval messages.

One of the key implementation details is the interaction with OCT to provide the answer sets. In order to interact correctly with OCT, JOF is required to produce valid OCT input, parse OCT result output and deal with the OCT error messages when they occur. Therefore, valid input would be an OCLP program as a string, satisfying OCT syntax. Output would be an answer set broadcast. It is possible to either synchronously run OCT and wait for its output, or as a thread which fires an event when finished.

JOF agents need a way of knowing when it is time to process their inputs, which is the motivation for the Call For Answer Sets (CFAS) protocol. This controls the life-

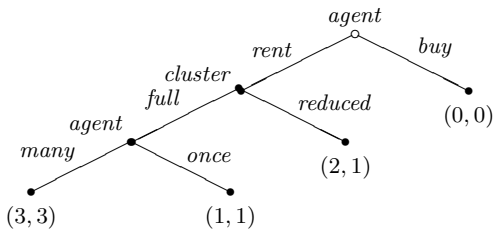

Fig. 4. The Cluster use Game of Example 11.

cycle of the JOF agents and is sent with a given deadline, after which the agents that not receive input will carry on using the empty set as input. Once all responses have been received, the solution to the updated OCLP is computed immediately.

JOF is not only implemented with LAIMA systems in mind, but is not limited to it. JOF can be seen as an open society where agents come and go and communication is established as necessary. The architecture also allows for different types of agents to join provided they communicate using the ontology developed.

6 Playing Games

In this section, we demonstrate that LAIMAS and its implementation JOF are an elegant and intuitive mechanism for reasoning about games and their equilibria. Due to page limitations, we limit the scope to extensive games with perfect information [4] and their Nash equilibria.

An extensive game is a detailed description of a sequential structure representing the decision problems encountered by agents (*players*) in strategic decision making (agents are capable of reasoning about their actions in a rational manner). The agents in the game are perfectly informed of all events that have previously occurred. Thus, they can decide upon their action(s) using information about the actions which have already taken place. This is done by means of passing *histories* of previous actions to the deciding agents. *Terminal histories* are obtained when all the agents/players have made their decision(s). Players have a preference for certain outcomes over others. Often, preferences are indirectly modelled using the concept of *payoff* where players are assumed to prefer outcomes where they receive a higher payoff. Therefore, an extensive game with perfect information is 4-tuple, denoted $\langle N, H, P, (\succcurlyeq_i)_{i \in N} \rangle$, containing the players N of the game, the histories H, a player function P telling who's turn it is after a certain history and a preference relation $\leq_i$ for each player i over the set of terminal histories.

For examples, we use a more convenient representation: a tree. The small circle at the top represents the initial history. Each path starting at the top represents a history. The terminal histories are the paths ending in the leafs. The numbers next to nodes represent the players while the labels of the arcs represent an action. The number below the terminal histories are payoffs representing the players' preferences (The first number is the payoff of the first player, the second number is the payoff of the second player, etc).

Example 11. An agent responsible for the large chunks of data processing contemplates using of an existing cluster (and paying rent for it) or buying a new cluster. The cluster supplier has the option to offer his client the full power of the cluster or not. Given full power, the agent can decide to use this service just once or many times. The game depicted in Figure 4 models an individual's predicament.

A *strategy* of a player in an extensive game is a plan that specifies the actions chosen by the player for every history after which it is her turn to move. A *strategy profile* contains a strategy for each player.

The first solution concept for an extensive game with perfect information ignores the sequential structure of the game; it treats the strategies as choices that are made once and for all before the actual game starts. A strategy profile is a *Nash equilibrium* if no player can unilaterally improve upon his choice. Put in another way, given the other players' strategies, the strategy stated for the player is the best this player can do.

Example 12. The game of Example 11 has two equilibria:
- $\{\{rent, many\}, \{full\}\}$, and
- $\{\{rent, once\}, \{reduced\}\}$

Given an extensive game with perfect information, a polynomial mapping to a LAIMAS exists such that the Nash equilibria of the former can be retrieved as the answer sets of the latter. The evolution leading to those answer sets models exactly how players in a game reason to find the answer sets. So LAIMAS are not only useful for calculation the equilibria but they also provide a mechanism to monitor the change of the players' beliefs.

An agent is created for each player in the game. The OCLP of such an agents contains as many component as the represented player has payoffs (step 1). The order among the components follows the expected payoff, higher payoffs correspond to more specific components (step 2). The various actions a player can choose from a certain stage of the game are turned into a choice rule which is placed in the most specific component of the agent modelling the player making the decision (step 3). Since Nash equilibria do not take into account the sequential structure of the game, players have to decide upon their strategy before starting the game, leaving them, for each decision, to reason about both past and future. This is reflected in the rules (step 4): each non-choice rule is made out of a terminal history (path from top to bottom in the tree) where the head represents the action taken by the player/agent, when considering the past and future created by the other players according to this history. The component of the rule corresponds to the payoff the deciding player would receive in case the history was actually followed.

Example 13. If we apply this technique to the game in Example 11 we obtain the multi-agent system depicted in Figure 5. The two answer sets of the system match the two Nash equilibria (Example 12) perfectly.

7 Conclusions and Directions for Future Research

The deductive multi-agent architecture described in this paper uses OCLP to represent the knowledge and reasoning capacities of the agents involved. However, the architec-

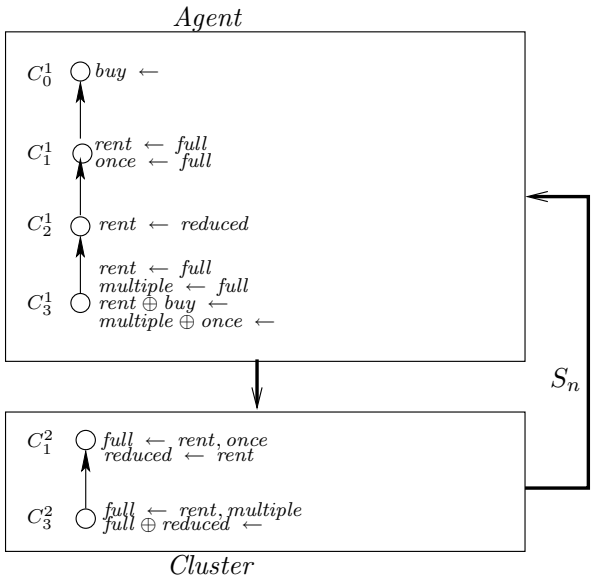

Fig. 5. The LAIMAS corresponding to the Cluster Game (Example 13)

ture and its behaviour are designed to be able to deal with any extension of traditional answer set programs. [2] gives an overview of some the language extensions currently available.

There is an emerging view [9, 15–18] that ASP, with or without preferences or other language constructs, is a very promising technology for building MAS. It satisfies the perspective that agent programming should be about describing *what* the goals are, not *how* they should be achieved— i.e. "post-declarative" programming [19]. Furthermore, having the same language for specification and implementation, makes verification trivial. Thanks to its formal grounding of all its language constructs, it becomes easy to verify the soundness and completeness of your language and implementation.

The Dali project [20] is a complete multi-agent platform entirely writen in Prolog. The EU-funded research project SOCS ([21, 22]) has constructed a multi-agent architecture based solely on logic programming components, each responsible for a part of the reasoning of the agent. Using ASP, it would be possible to use the same language for all components, avoiding duplication of domain description and knowledge. This technique is used in [16] for diagnosis and planning for the propulsion system of the NASA Space Shuttle.

Another multi-agent system is the Minerva architecture [18]. They build their agents out of subagents that work on a common knowledge base written as a MDLP (Multi-dimentional Logic Program) which is an extension of Dynamic Logic Programming. It can be shown that MDLP can easily be translated into OCLP such that their stable models match our answer sets. The advantage of using OCLP is that you have a far

more flexible defeating structure. One is not restricted to decision comprising only two alternatives. Furthermore, decision are dynamic in our system. On the other hand, the Minerva does not restrict itself to modelling the beliefs of agents, but allows for full BDI-agents that can plan towards a certain goal. It would be interesting to see what results we obtain when OCLP approach was incorporated into Minerva and what results we would obtain by incorporating Minerva into LAIMAS.

The flexibility on the specification side of ASP is counter-balanced by the technology of current (grounding) answer set solvers, which mean that a major part of the solution space is instantiated and analysed before the any computation of a solution begins. The advantage of this is that the computation is staged [23] into a relatively expensive part that can be done off-line and a relatively cheap part that is executed on-line in response to queries. For example, [16] reports that the space shuttle engine control program is 21 pages, of which 18 are declarations and 3 are rules describing the actual reasoning. Grounding takes about 80% of the time, while the 20% is used for the actual computation of the answer set and answering queries. There is a parallel between the way answer set solvers compute the whole solution space and model-checking.

In our framework, an agent is given an initial set of beliefs and a set of reasoning capacities. Each time the agent receives information from the outside world, it will update its knowledge/beliefs base and computes the answer sets that go with it. This means that for even the smallest change the whole program has to be recomputed (including the expensive grounding process). To provide some additional flexibility, a recent development [24] describes a technique for *incremental* answer set solving. With an incremental solver that permits the assertion of new rules at run-time we are in a position to move from purely reactive agents to deliberative agents that can support belief-desire-intention style models of mentality. We aim to replace the fixed solver (OCT) we have built into JADE and described here with the incremental one (IDEAS) shortly.

The current implementation of grounding in ASP limits its use to finite domains. Clearly, a fixed solution space is quite limiting in some domains, but reassuring in others. OCLP significantly simplifies the development of such systems by constructing a finite domain. To go beyond the grounding solver is currently one of the key challenges, together with the more practical aspects of ASP like updates, methodology, and debugging.

The advantage of an ungrounded solver is that the initialisation overheads are significantly reduced, compared to a grounding solver, since the only task is the grounding and removal of redundant rules. However, the disadvantage is that the finite space for checking (discussed above) is lost: the price of flexibility is a loss in verifiability. We are currently working on a lazy grounding solver, as a result of which we believe we will be in a position to offer a very high-level programming language for the construction of practical reasoning agents. Thus, not only will it be possible to extend the knowledge base of the agent dynamically, but it will also enable the solution of problems over infinite domains, like time (discreet or continuous), numbers.

It seems likely that the next few years will see significant developments in answer set solver technology, deploying both incremental and ungrounded solvers. We foresee future improvements on the framework itself. Currently, all our agents take input only from other agents. At present, agents communicate positive information, but it would

very beneficial if they could communicate negative information too. Unfortunately, this may lead to contradictions, but various strategies to deal with it come to mind, such as removing conflicting information, introducing trust between agents or allowing another sort of non-determinism by attempting to initiate a cycle for both possibilities. Furthermore, the LAIMAS agents are very generous in updating the other agents about their knowledge and beliefs. In reality, it would be most likely better for the agents to only share part of their knowledge, to make sure that the others cannot take advantage of it. With the current implementation, all these changes can easily be made by simply adding an agent or changing the JADE wrapper around OCT.

The LAIMA system and its answer set semantics was developed using OCLP as a way to model the agents. However, the system could also be used for other agent characterisations. In the paper, we assumed that agents will always prefer their own beliefs over the information passed on by other. This does not need to be the case, neither would all agents react in the same way. An extension we currently looking at, is the used of trust levels to allow agents to distinguish between various sources and deal with it appropriately. In this system we assumed that the environment would be represented as an agent. Normally information passed from the environment should be considered more accurate than an agent's current beliefs. Using these trust levels and OCLP, it would be easy to model this behaviour: each input is assigned its own component with an ordering depending on the trust level of the supplier.

Another problem for the moment is feeding back failure into the system. When an agent fails to produce an answer set for its updated version, communication will stop at this agent, without warning the others. From both a theoretical and implementation point of view this raises interesting possibilities

References

1. Opencyc:: (`http://www.cyc.com.opencyc`)
2. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge Press (2003)
3. Brain, M., De Vos, M.: Implementing OCLP as a front-end for Answer Set Solvers: From Theory to Practice. In: ASP03: Answer Set Programming: Advances in Theory and Implementation, Ceur-WS (2003) online CEUR-WS.org/Vol-78/asp03-final-brain.ps.
4. Osborne, M.J., Rubinstein, A.: A Course in Game Theory. Third edn. The MIT Press, Cambridge, Massachusets, London, Engeland (1996)
5. Jade:: (`http://jade.tilab.com/`)
6. Protégé: (`http://protege.stanford.edu/`)
7. De Vos, M.: Implementing Ordered Choice Logic Programming using Answer Set Solvers. In: Third International Symposium on Foundations of Information and Knowledge Systems (FoIKS'04). Volume 2942., Vienna, Austria, Springer Verlag (2004) 59–77
8. Denecker, M.: What's in a Model? Epistemological Analysis of Logic Programming, Ceur-WS (2003) online CEUR-WS.org/Vol-78/.
9. De Vos, M., Vermeir, D.: Extending Answer Sets for Logic Programming Agents. Annals of Mathematics and Artifical Intelligence **42** (2004) 103–139 Special Issue on Computational Logic in Multi-Agent Systems.
10. De Vos, M., Vermeir, D.: On the Role of Negation in Choice Logic Programs. In Gelfond, M., Leone, N., Pfeifer, G., eds.: Logic Programming and Non-Monotonic Reasoning Conference

(LPNMR'99). Volume 1730 of Lecture Notes in Artificial Intelligence., El Paso, Texas, USA, Springer Verslag (1999) 236–246
11. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Proc. of fifth logic programming symposium, MIT PRESS (1988) 1070–1080
12. Niemelä, I., Simons, P.: Smodels: An implementation of the stable model and well-founded semantics for normal LP. In Dix, J., Furbach, U., Nerode, A., eds.: Proceedings of the 4th International Conference on Logic Programing and Nonmonotonic Reasoning. Volume 1265 of LNAI., Berlin, Springer (1997) 420–429
13. Eiter, T., Leone, N., Mateis, C., Pfeifer, G., Scarcello, F.: The KR system `dlv`: Progress report, comparisons and benchmarks. In Cohn, A.G., Schubert, L., Shapiro, S.C., eds.: KR'98: Principles of Knowledge Representation and Reasoning. Morgan Kaufmann, San Francisco, California (1998) 406–417
14. FIPA:: (`http://www.fipa.org/`)
15. Dix, J., Eiter, T., Fink, M., Polleres, A., Zhang, Y.: Monitoring Agents using Declarative Planning. Fundamenta Informaticae **57** (2003) 345–370 Short version appeared in Gnther/Kruse/Neumann (Eds.), Proceedings of KI 03, LNAI 2821, 2003.
16. Nogueira, M., Balduccini, M., Gelfond, M., Watson, R., Barry, M.: A A-Prolog Decision Support System for the Space Shuttle. In: Answer Set Programming: Towards Efficient and Scalable Knowledge Represenation and Reasoning. American Association for Artificial Intelligence Press, Stanford (Palo Alto), California, US (2001)
17. Gelfond, M.: Answer set programming and the design of deliberative agents. In: Procs. of 20th International Conference on Logic Programming. Number 3132 in Lecture Notes in Artificial Intelligence (LNCS) (2004) 19–26
18. Leite, J.A., Alferes, J.J., Pereira, L.M.: Minerva - a dynamic logic programming agent architecture. In Meyer, J.J., Tambe, M., eds.: Intelligent Agents VIII. Number 2002 in LNAI, Springer-Verlag (2002) 141–157
19. Wooldridge, M.J., Jennings, N.R.: Agent theories, architectures and languages: a survey. In Wooldridge, M.J., Jennings, N.R., eds.: Intelligent Agents; ECAI-94 Workshop on Agent Theories, Architectures, and Languages (Amsterdam 1994). Volume 890 of LNCS (LNAI)., Berlin: Springer (1994) 1–39
20. Costantini, S., Tocchio, A.: A Logic Programming Language for Multi-agent Systems. In: Logics in Artificial Intelligence, Proceedings of the 8th European Conference, Jelia 2002. Volume 2424 of Lecture Notes in Artificial Intelligence., Cosenza, Italy, Springer-Verlag, Germany (2002)
21. Kakas, A., Mancarella, P., Sadri, F., Stathis, K., Toni, F.: Declarative agent control. In Leite, J., Torroni, P., eds.: 5th Workshop on Computational Logic in Multi-Agent Systems (CLIMA V). (2004)
22. SOCS:: (`http://lia.deis.unibo.it/research/socs/`)
23. Jørring, U., Scherlis, W.: Compilers and staging transformations. In: Proceedings of 13th ACM Symposium on Principles of Programming Languages, New York, ACM (1986) 86–96
24. Brain, M.J.: Undergraduate dissertation: Incremental answer set programming. Technical Report 2004–05, University of Bath, U.K., Bath (2004)