Jason

A Java-based interpreter for an extended version of AgentSpeak

developed by

Rafael H. Bordini[1] and *Jomi F. Hübner*[2]

The work leading to ***Jason*** received many contributions, in particular from:

Michael Fisher, *Joyce Martins*, *Álvaro F. Moreira*, *Renata Vieira*, *Willem Visser*, *Michael Wooldridge*,

and many others; see the Acknowledgements section for a full list.

[1]Department of Computer Science
University of Durham
Durham DH1 3LE, U.K.
R.Bordini@durham.ac.uk

[2]Depto. de Sistemas e Computação
Universidade Regional de Blumenau
Blumenau, SC 89035-160, Brazil
jomi@inf.furb.br

Release Version 0.9.5, February 2007.

Jason
by *Gustave Moreau* (1865)

Oil on canvas, 204 x 115.5 cm, Musée d'Orsay, Paris.

In the painting, there are two quotations from book VII of P. Ovidius Naso's *Metamorphoses*: "Nempe tenens, quod amo, gremioque in Iasonis haerens per freta longa ferar: nihil illum amplexa verebor" (verses 66-67), and "heros Aesonius potitur spolioque superbus muneris auctorem secum, spolia altera, portans" (verses 156-157). Quoted from URL <`http://www.perseus.tufts.edu/cgi-bin/ptext?doc=Perseus%3Atext%3A1999.02.0029&query=head%3D%237`>, where English translations can be found.

Typeset in Bordini fonts using LaTeX.

Preface

One of the best known approaches to the development of cognitive agents is the BDI (Beliefs-Desires-Intentions) architecture. In the area of agent-oriented programming languages in particular, AgentSpeak(L) has been one of the most influential abstract languages based on the BDI architecture. The type of agents specified with AgentSpeak(L) are sometimes referred to as reactive planning systems. To the best of our knowledge, ***Jason*** is the first fully-fledged interpreter for a much improved version of AgentSpeak, including also speech-act based inter-agent communication. Using SACI, a ***Jason*** multi-agent system can be distributed over a network effortlessly. Various *ad hoc* implementations of BDI systems exist, but one important characteristic of AgentSpeak is its theoretical foundation; work on formal verification of AgentSpeak systems is also underway (references are given throughout this document). Another important characteristic of ***Jason*** in comparison with other BDI agent systems is that it is implemented in Java (thus multi-platform) and is available *Open Source*, and is distributed under GNU LGPL.

Besides interpreting the original AgentSpeak(L) language, ***Jason*** also features:

- strong negation, so both closed-world assumption and open-world are available;
- handling of plan failures;
- speech-act based inter-agent communication (and belief annotations on information sources);
- annotations on plan labels, which can be used by elaborate (e.g., decision theoretic) selection functions;
- support for developing Environments (which are not normally to be programmed in AgentSpeak; in this case they are programmed in Java);
- the possibility to run a multi-agent system distributed over a network (using SACI);
- fully customisable (in Java) selection functions, trust functions, and overall agent architecture (perception, belief-revision, inter-agent communication, and acting);
- a library of essential "internal actions";
- straightforward extensibility by user-defined internal actions, which are programmed in Java.

Besides, it is an implementation of the operational semantics, formally given to the AgentSpeak(L) language and most of the extensions available in ***Jason***.

This manual is still quite preliminary, and written in the form of something like a tutorial. It does cover most of the features of the version of AgentSpeak available in the interpreter provided with ***Jason***, but some of them are only just mentioned, and the reader is referred to research papers where the ideas are explained in more details. We decided to release Jason even though documentation was still preliminary, and having done very little testing on the interpreter, because many colleagues were in need of an AgentSpeak interpreter. Future releases of ***Jason*** will have better documentation, and the interpreter should be better tested (in the long run, maybe even model checked, we hope!).

1 Introduction

The idea of Agent-Oriented Programming was first discussed by Yoav Shoham [24]. Although agent-oriented programming is still incipient, the whole area of multi-agent systems [30] has received a great deal of attention from computer scientists in general in the last few years. Researchers in the area of multi-agent systems think that the technologies emerging from the area are likely to influence the design of computational system in very challenging areas of application, where such systems are situated within very dynamic, unpredictable environments. From that perspective, agent-oriented programming is likely to become a new major approach to the development of computational system in the near future.

The language interpreted by ***Jason*** is an extension of AgentSpeak(L), an abstract agent language originally devised by Rao [22], and subsequently extended and discussed in a series of papers by Bordini and colleagues (e.g., [8, 3, 7, 18, 20, 5, 10, 6, 9]). AgentSpeak has a neat notation and is a thoughtful (and computationally efficient) extension of logic programming to BDI agents [29, 25]. ***Jason*** is a fully-fledged interpreter for AgentSpeak with many extensions making up for a very expressive programming language for cognitive agents. Also, an interesting feature available with ***Jason*** is that a multi-agent system can be easily configured to run on various hosts. This is accomplished by the use of SACI, an agent communication infra-structure implemented by Jomi Hübner [15] (see `http://www.lti.pcs.usp.br/saci/`).

This document is intended as ***Jason***'s manual, but is structured more in the form of a (still rather concise) tutorial. The next chapter gives an introduction the basics of AgentSpeak(L). Chapters 3 and 4 give the syntax allowed in agent and multi-agent systems specifications, respectively. In Chapter 5, we discuss various aspects of agents, environment, and customising agents (using Java code). Chapter 6 provides brief instructions on how to install ***Jason***.

2 An Introduction to AgentSpeak(L)

2.1 Basic Notions

The AgentSpeak(L) programming language was introduced in [22]. It is a natural extension of logic programming for the BDI agent architecture, and provides an elegant abstract framework for programming BDI agents. The BDI architecture is, in turn, the predominant approach to the implementation of "intelligent" or "rational" agents [29]. An AgentSpeak(L) agent is created by the specification of a set of base beliefs and a set of plans. A *belief atom* is simply a first-order predicate in the usual notation, and belief atoms or their negations are termed *belief literals*. An *initial set of beliefs* is just a collection of ground belief atoms.

AgentSpeak(L) distinguishes two types of goals: *achievement goals* and *test goals*. Achievement and test goals are predicates (as for beliefs) prefixed with operators '`!`' and '`?`' respectively. Achievement goals state that the agent wants to achieve a state of the world where the associated predicate is true. (In practice, these initiate the execution of *subplans*.) A *test goal* returns a unification for the associated predicate with one of the agent's beliefs; they fail otherwise. A *triggering event* defines which events may initiate the execution of a plan. An *event* can be internal, when a subgoal needs to be achieved, or external, when generated from belief updates as a result of perceiving the environment. There are two types of triggering events: those related to the *addition* ('+') and *deletion* ('-') of mental attitudes (beliefs or goals).

Plans refer to the *basic actions* that an agent is able to perform on its environment. Such actions are also defined as first-order predicates, but with special predicate symbols (called action symbols) used to distinguish them from other predicates. A plan is formed by a *triggering event* (denoting the purpose for that plan), followed by a conjunction of belief literals representing a *context*. The context must be a logical consequence of that agent's current beliefs for the plan to be *applicable*. The remainder of the plan is a sequence of basic actions or (sub)goals that the agent has to achieve (or test) when the plan, if applicable, is chosen for execution.

```
+concert(A,V) : likes(A)
   ← !book_tickets(A,V).

+!book_tickets(A,V) : ¬busy(phone)
   ← call(V);
      ...;
      !choose_seats(A,V).
```

Figure 2.1: Examples of AgentSpeak(L) Plans

Figure 2.1 shows some examples of AgentSpeak(L) plans. They tell us that, when a concert is announced for artist `A` at venue `V` (so that, from perception of the environment, a belief `concert(A,V)` is *added*), then if this agent in fact likes artist `A`, then it will have the new goal of booking tickets for that concert. The second plan tells us that whenever this agent adopts the goal of booking tickets for `A`'s performance at `V`, if it is the case that the telephone is not busy, then it can execute a plan consisting of performing the basic action `call(V)` (assuming that making a phone call is an atomic action that the agent can perform) followed by a certain protocol for booking tickets (indicated by '...'), which in this case ends with the execution of a plan for choosing the seats for such performance at that particular venue.

$$
\begin{array}{lcll}
ag & ::= & bs \quad ps & \\
bs & ::= & at_1\texttt{.} \quad \ldots \quad at_n\texttt{.} & (n \geq 0) \\
at & ::= & \mathsf{P}(t_1, \ldots, t_n) & (n \geq 0) \\
ps & ::= & p_1 \quad \ldots \quad p_n & (n \geq 1) \\
p & ::= & te \texttt{ : } ct \texttt{ <- } h \texttt{ .} & \\
te & ::= & \texttt{+}at \;\mid\; \texttt{-}at \;\mid\; \texttt{+}g \;\mid\; \texttt{-}g & \\
ct & ::= & \texttt{true} \;\mid\; l_1 \texttt{ \& } \ldots \texttt{ \& } l_n & (n \geq 1) \\
h & ::= & \texttt{true} \;\mid\; f_1 \texttt{ ; } \ldots \texttt{ ; } f_n & (n \geq 1) \\
l & ::= & at \;\mid\; \texttt{not } at & \\
f & ::= & \mathsf{A}(t_1, \ldots, t_n) \;\mid\; g \;\mid\; u & (n \geq 0) \\
g & ::= & \texttt{!}at \;\mid\; \texttt{?}at & \\
u & ::= & \texttt{+}at \;\mid\; \texttt{-}at &
\end{array}
$$

Figure 2.2: The Concrete Syntax of AgentSpeak(L).

2.2 AgentSpeak(L) Syntax

An AgentSpeak(L) agent specification ag is given by the grammar[1] in Figure 2.2. In AgentSpeak(L), an agent is simply specified by a set bs of beliefs (the agent's initial belief base) and a set ps of plans (the agent's plan library). The atomic formulæ at of the language are predicates where P is a predicate symbol, A is an action symbol, and $t_1, \ldots, t_n$ are standard terms of first order logic. Note that at in bs must be ground (i.e., variables are not allowed).

A plan in AgentSpeak(L) is given by p above, where te is the *triggering event*, ct is the plan's context, and h is sequence of actions, goals, or belief updates; $te : ct$ is referred as the *head* of the plan, and h is its *body*. Then the set of plans of an agent is given by ps as a list of plans. Each plan has in its head a formula ct that specifies the conditions under which the plan can be executed. The formula ct must be a logical consequence of the agent's beliefs if the plan is to be considered applicable.

A triggering event te can then be the addition or the deletion of a belief from an agent's belief base ($+at$ and $-at$, respectively), or the addition or the deletion of a goal ($+g$ and $-g$, respectively). A sequence h of actions, goals, and belief updates defines the body of a plan. We assume the agent has at its disposal a set of *actions* and we use a as a metavariable ranging over them. They are given as normal predicates except that an action symbol A is used instead of a predicate symbol. Goals g can be either *achievement goals* ($!at$) or *test goals* ($?at$). Finally, $+at$ and $-at$ (in the body of a plan) represent operations for updating (u) the belief base by, respectively, adding and removing at.

2.3 Informal Semantics

The AgentSpeak(L) interpreter also manages a set of *events* and a set of *intentions*, and its functioning requires three *selection functions*. The event selection function ($\mathcal{S}_\mathcal{E}$) selects a single event from the set of events; another selection function ($\mathcal{S}_\mathcal{O}$) selects an "option" (i.e., an applicable plan) from a set of applicable plans; and a third selection function ($\mathcal{S}_\mathcal{I}$) selects one particular intention from the set of intentions. The selection functions are supposed to be agent-specific, in the sense that they should make selections based on an agent's characteristics (though previous work on AgentSpeak(L) did not elaborate on how designers specify such functions[2]). Therefore, we here leave the selection functions undefined, hence the choices made by them are supposed to be non-deterministic.

Intentions are particular courses of actions to which an agent has committed in order to handle certain events. Each intention is a stack of partially instantiated plans. *Events*, which may start off the execution of plans that have relevant triggering events, can be *external*, when originating from perception of the agent's environment (i.e., addition and deletion of beliefs based on perception are external events); or *internal*, when generated from the agent's own execution of a plan (i.e., a subgoal in a plan generates an event of type "addition of achievement goal"). In the latter case, the event is accompanied with the intention which generated it (as the plan chosen for that event will be pushed

[1] Note that this is just an introduction on the basics of the language. The grammar for AgentSpeak with all the extensions supported in ***Jason*** will be given in Chapter 3. Also, the only addition to Rao's original language used here are addition and deletion of beliefs in the body of plans.

[2] Our extension to AgentSpeak(L) in [3] deals precisely with the automatic generation of efficient intention selection functions. The extended language allows one to express relations between plans, as well as quantitative criteria for their execution. We then use decision-theoretic task scheduling to guide the choices made by the intention selection function.

on top of that intention). External events create new intentions, representing separate focuses of attention for the agent's acting on the environment.

We next give some more details on the functioning of an AgentSpeak(L) interpreter, which is clearly depicted in Figure 2.3 (reproduced from [18]). The pictorial description of such interpreter, given in Figure 2.3, greatly facilitates the understanding of the interpreter for AgentSpeak(L) proposed by Rao. In the figure, sets (of beliefs, events, plans, and intentions) are represented as rectangles. Diamonds represent selection (of one element from a set). Circles represent some of the processing involved in the interpretation of AgentSpeak(L) programs.

At every interpretation cycle of an agent program, AgentSpeak(L) updates a list of events, which may be generated from perception of the environment, or from the execution of intentions (when subgoals are specified in the body of plans). It is assumed that beliefs are updated from perception and whenever there are changes in the agent's beliefs, this implies the insertion of an event in the set of events. This belief revision function is not part of the AgentSpeak(L) interpreter, but rather a necessary component of the agent architecture.

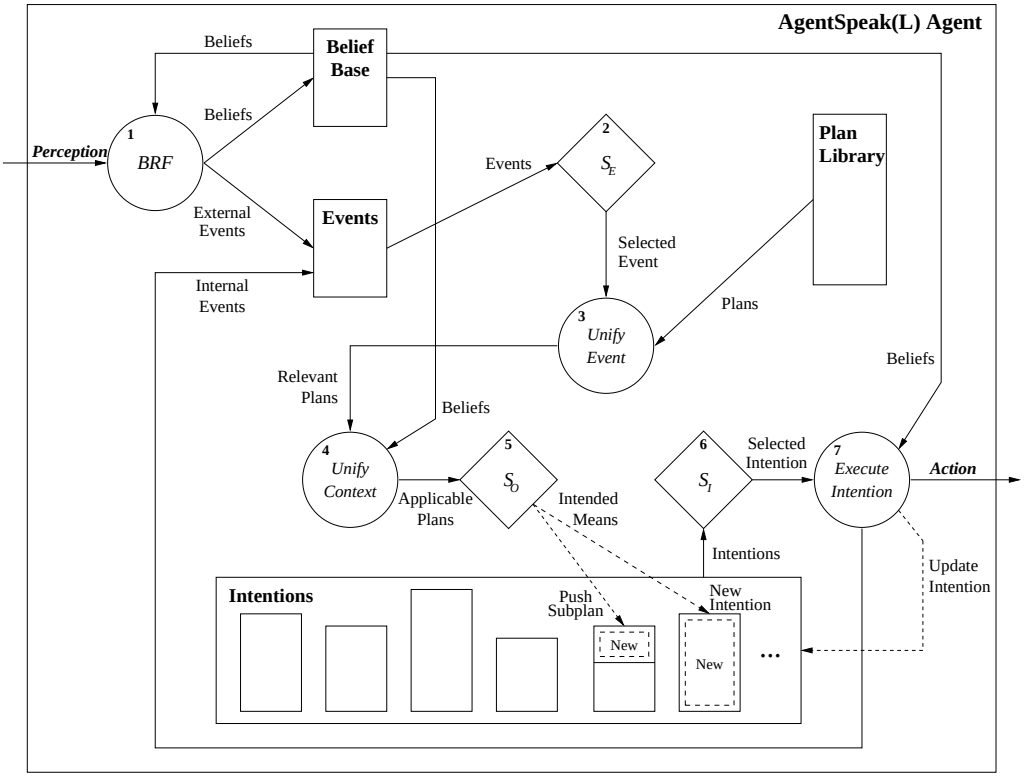

Figure 2.3: An Interpretation Cycle of an AgentSpeak(L) Program [18].

After $\mathcal{S}_\mathcal{E}$ has selected an event, AgentSpeak(L) has to unify that event with triggering events in the heads of plans. This generates a set of all *relevant plans*. By checking whether the context part of the plans in that set follow from the agent's beliefs, AgentSpeak(L) determines a set of *applicable plans* (plans that can actually be used at that moment for handling the chosen event). Then $\mathcal{S}_\mathcal{O}$ chooses a single applicable plan from that set, which becomes the *intended means* for handling that event, and either pushes that plan on the top of an existing intention (if the event was an internal one), or creates a new intention in the set of intentions (if the event was external, i.e., generated from perception of the environment).

All that remains to be done at this stage is to select a single intention to be executed in that cycle. The $\mathcal{S}_\mathcal{I}$ function selects one of the agent's intentions (i.e., one of the independent stacks of partially instantiated plans within the set of intentions). On the top of that intention there is a plan, and the formula in the beginning of its body is taken for execution. This implies that either a basic action is performed by the agent on its environment, an internal event is generated (in case the selected formula is an achievement goal), or a test goal is performed (which means that the set of beliefs has to be checked).

If the intention is to perform a basic action or a test goal, the set of intentions needs to be updated. In the case of a test goal, the belief base will be searched for a belief atom that unifies with the predicate in the test goal. If that search succeeds, further variable instantiation will occur in the partially instantiated plan which contained that test goal (and the test goal itself is removed from the

intention from which it was taken). In the case where a basic action is selected, the necessary updating of the set of intentions is simply to remove that action from the intention (the interpreter informs to the architecture component responsible for the agent effectors what action is required). When all formulæ in the body of a plan have been removed (i.e., have been executed), the whole plan is removed from the intention, and so is the achievement goal that generated it (if that was the case). This ends a cycle of execution, and AgentSpeak(L) starts all over again, checking the state of the environment after agents have acted upon it, generating the relevant events, and so forth.

2.4 Simple Examples

This section show two simple examples of programs written in AgentSpeak(L). The purpose here is just to give examples of the use of the basic constructs of the language. The applications are very simple, so as to be easily understood, but BDI programming is particularly interesting for complex system in dynamic environments. The applications shown here are therefore not illustrative of typical BDI oriented programming, but rather an attempt to show in practice the various constructs of the language. Both examples (given in the next two section) were first used in papers that introduced model-checking techniques for AgentSpeak. The application in Section 2.4.1 was first presented in [6], and the one in Section 2.4.2 in [5]. Both of these applications are available as runnable examples in ***Jason***'s distribution (note that the examples available in the distribution might be slightly improved from the ones below, as we use them to test new features introduced in ***Jason***).

2.4.1 Collecting Garbage

The scenario used here involves two robots that are collecting garbage on planet Mars, and was first used in [6]. Robot `r1` searches for pieces of garbage and when one is found, the robot picks it up, take it to the location of `r2`, drops the garbage there, and return to the location where it found the garbage and continues its search from that position. Robot `r2` is situated at an incinerator; whenever garbage is taken to its location by `r1`, `r2` just puts it in the incinerator. One or two pieces of garbage are randomly scattered on the grid. Another source of non-determinism is a certain imprecision of the robot's arm that grabs the pieces of garbage. The action of picking up garbage may fail, but it is assumed that the mechanism is good enough so that it never fails more than twice; that is, in the worst case robot `r1` has to attempt to pick up a piece of garbage three times, but by then `r1` will definitely have grabbed it). The AgentSpeak(L) code for `r1` is given in below; we have annotated each plan with a label, so that we can refer to them in the text below.

Agent `r1`

Beliefs

```
pos(r2,2,2).
checking(slots).
```

Plans

```
+pos(r1,X1,Y1) : checking(slots) & not garbage(r1)                    (p1)
   <- next(slot).

+garbage(r1) : checking(slots)                                        (p2)
   <- !stop(check);
      !take(garb,r2);
      !continue(check).

+!stop(check) : true                                                  (p3)
   <- ?pos(r1,X1,Y1);
      +pos(back,X1,Y1);
      -checking(slots).

+!take(S,L) : true                                                    (p4)
   <- !ensure_pick(S);
      !go(L);
      drop(S).

+!ensure_pick(S) : garbage(r1)                                        (p5)
   <- pick(garb);
      !ensure_pick(S).
```

```
+!ensure_pick(S) : true <- true.                                    (p6)

+!continue(check) : true                                            (p7)
   <- !go(back);
      -pos(back,X1,Y1);
      +checking(slots);
      next(slot).

+!go(L) : pos(L,Xl,Yl) & pos(r1,Xl,Yl)                              (p8)
   <- true.

+!go(L) : true                                                      (p9)
   <- ?pos(L,Xl,Yl);
      moveTowards(Xl,Yl);
      !go(L).
```

The only initial beliefs this agent needs is on the position of agent `r2` on the "grid" into which the territory is divided, and that, to start with, what it is doing is to check all the slots in the territory grid for garbage. All the plans are explained below.

Plan `p1` is used when the agent perceives that it is in a new position and its currently checking for garbage. If there is no garbage perceived in that slot, all it has to do is a basic action `next(slot)` which moves the robot to the next slot in the grid, except the one where the incinerator is: garbage in that position is dealt with by robot `r2`. Note that this is a basic action from the point of view of the agent's reasoning: it is assumed that the robot has the mechanisms to move itself to a next position on the territory grid, automatically skipping the incinerator's position.

The environment provides percepts stating whether a piece of garbage is present in `r1` or `r2`'s position. When `r1` perceives garbage in its position, a belief `garbage(r1)` is added to its belief base, hence plan `p2` can be used (if the agent is still in its operation mode where it is checking for garbage, rather then moving them to `r2` or coming back). The task of dealing with a perceived piece of garbage is accomplished in three parts. It involves achieving subgoals which: (i) make sure that the robot will stop checking for garbage in a consistent manner (e.g., remembering where it was so that the task of continuing the search can be resumed); (ii) actually taking the garbage (`garb`) to `r2`; and (iii) resuming the task of checking the slots for garbage. Each of these goals are achieved by the following three plans respectively.

When the agent intends to achieve subgoal (i) above, plan `p3` is its only option, and it is always applicable (i.e., it has an empty context). The agent retrieves from its belief base its own current position (which will be in its belief base from perception of the environment). It then makes a note to itself of where is the position it will need to go back to when it is to resume searching for garbage. This is done by adding a belief to its belief base: `+pos(back,X1,Y1)`. It also removes from its belief base the information that it is presently searching for garbage, as this is no longer true (it will soon intend to take the garbage to `r2` and then go back).

Subgoal (ii) is achieved by plan `p4`, which states that in order for the robot to take garbage to `r2`, it should pick it up, then achieve the subgoal of going to the position where `r2` is, and when `r1` is there, it can finally drop the garbage. Note that `pick(garb)` and `drop(garb)` are again basic actions, i.e., things that are assumed the robot can "physically" perform in the environment, by means of its hardware apparatus.

Plans `p5` and `p6` together ensure that the robot will keep trying to pick up a piece of garbage until it can no longer perceive garbage on the grid (i.e., the grabbing mechanism succeeded). Recall that the grabbing mechanism is imprecise, so the robot may need to try a few times before it succeeds.

Plan `p7` is used for the agent to continue the task of checking the grid slots for garbage. The agent needs to achieve the subgoal of going back to its previous position (`!go(back)`), and once there, it can remove the note it has made of that position, remember itself that it is now again checking the slots for garbage, and then proceed to the next slot.

The last two plans are used for achieving the goal of going to a specific position on the grid where `L` is located. Plan `p9` retrieves the belief the agent has about the location of reference `L`, then move itself towards those coordinates `moveTowards(Xl,Yl)`, and keep going towards `L` (this is a recursive goal). Plan `p8` provides the end of the recursion, saying that there is nothing else to do in order to achieve the goal of going towards `L`, if agent `r1` is already at that position.

Agent `r2` is defined by the very simple AgentSpeak(L) code below. All it does is to burn the pieces of garbage (`burn(garb)`) when it senses that there is garbage on its location (`+garbage(r2)`).

Agent r2
```
+garbage(r2) : true
   <- burn(garb).
```

All it does is to burn the pieces of garbage (`burn(garb)`) when it senses that there is garbage on its location. A belief `+garbage(r2)` is added to the agent's belief base by belief revision (from the perception of the environment).

2.4.2 An Abstract Auction Model

In this section, we describe a simplified auction scenario, first used in [5]. The simple environment announces 10 auctions and simply states which agent is the winner in each one (the one with the highest bid). There are three agents participating in these auctions, with three simplified bidding strategies.

Agent ag1
```
+auction(N) : true
   <- place_bid(N,6).
```

Agent ag1 is a very simple agent which bids 6 whenever the environment announces a new auction.

Agent ag2
```
myself(ag2).
bid(ag2,4).
ally(ag3).

+auction(N) : myself(I) & ally(A) & not alliance(A,I)
   <- ?bid(I,B); place_bid(N,B).

+auction(N) : alliance(A,I)
   <- place_bid(N,0).

+alliance(A,I) : myself(I) & ally(A)
   <- ?bid(I,B);
      .send(A,tell,bid(I,B));
      .send(A,tell,alliance(A,I)).
```

Agent ag2 bids 4, unless it has agreed on an alliance with ag3, in which case it bids 0. When ag2 receives a message from ag3 proposing an alliance, a belief `alliance(ag3,ag2)` is added to ag2's belief base (the default trust function is used). That is a triggering event to the last plan, which informs ag3 of how much ag2 was bidding, and confirms that ag2 agrees to form an alliance with ag3.

Agent ag3
```
myself(ag3).
bid(ag3,3).
ally(ag2).
threshold(3).

+auction(N) : threshold(T) & N < T
   <- !bid_normally(N).

+auction(N) : myself(I) & winner(I)
              & ally(A) & not alliance(I,A)
   <- !bid_normally(N).

+auction(N) : myself(I) & not winner(I)
              & ally(A) & not alliance(I,A)
   <- !alliance(I,A);
      !bid_normally(N).

+auction(N) : alliance(I,A)
   <- ?bid(I,B); ?bid(A,C);
      D = B+C; place_bid(N,D).
```

```
+!bid_normally(N) : true
   <- ?bid(I,B); place_bid(N,B).

+!alliance(I,A) : true
   <- .send(A,tell,alliance(I,A)).
```

Agent ag3 tries to win the first T auctions, where T is a threshold stored in its belief base. If it is does not win any auctions up to that point, it will try to achieve an alliance with ag2 (by sending the appropriate message to it). When ag2 confirms that it agrees to form an alliance, then ag3 starts bidding, on behalf of them both, with the sum of their usual bids.

2.5 Recent Research on AgentSpeak

Since Rao's original proposal [22], a number of authors have investigated a range of different aspects of AgentSpeak(L). In [12], a complete abstract interpreter for AgentSpeak(L) was formally specified using the Z specification language. Most of the elements in that formalisation had already appeared in [11]; this highlights the fact that AgentSpeak(L) is strongly based on the experiences with the BDI-inspired dMARS [16] system, a successor of PRS [13].

Some extensions to AgentSpeak(L) were proposed in [3], and an interpreter for the extended language was introduced. The extensions aim at providing a more practical programming language; the extended language also allows the specification of relations between plans and quantitative criteria for their execution. The interpreter then uses decision-theoretic task scheduling for automatically guiding the choices made by an agent's intention selection function.

In [19], an operational semantics for AgentSpeak(L) was given following Plotkin's structural approach; this is a more familiar notation than Z for giving semantics to programming languages. Later, that operational semantics was used in the specification of a framework for carrying out proofs of BDI properties of AgentSpeak(L) [7]. The particular combination of asymmetry thesis principles [23] satisfied by any AgentSpeak(L) agent was shown in [7], and later in [8], where detailed proofs were included. This is relevant in assuring the rationality of agents programmed in AgentSpeak(L) based on the principles of rationality from the point of view of the BDI theory. In [20], the operational semantics of AgentSpeak(L) was extended to give semantics to speech-acts based inter-agent communication.

Recently, a toolkit called CASP was introduced which allows the use of model checking techniques for the automatic verification of AgentSpeak(L)-like programs [10]. Those tools translate a simplified version of AgentSpeak(L) into the input language of existing model checkers for linear temporal logic. The translation to Promela was presented in [5], and the translation to Java in [6]. The actual model checking is then done by Spin [14] in the first case, and JPF2 [28] in the second. This work on model checking AgentSpeak(L) programs uses the definitions of the BDI modalities in terms of the operational semantics of AgentSpeak(L), as defined in [8] to show precisely how the BDI specifications to be verified against systems of multiple AgentSpeak(L) agents are interpreted.

At the moment, few applications have been developed with AgentSpeak(L), given that only recently it has been implemented in practice, and is still in need of a great deal of experimentation in order to make it a powerful programming language. In fact, the same applies to other agent oriented programming languages; although work in the area has been going on since the early 90's, a large number of issues still remain to be clarified and properly sorted out. Among the existing application developed with AgentSpeak(L) we can mention a simulation of social aspects of urban growth [9], and a storehouse robot in a virtual reality environment [26]. The former was developed as a test case for the MAS-SOC platform for agent-based social simulation, which is based on AgentSpeak(L) for the implementation of individual agents. The latter was used as a test case for a two-layered architecture which uses an AgentSpeak(L) interpreter at one level, and has an articulated system for modelling 3D characters in virtual environments at the other level. Such architecture is aimed at allowing the use of sophisticated autonomous agents in virtual reality systems or other applications based on 3D animations.

The reason for the little use of AgentSpeak in the development of practical application is certainly related to the fact that, before this release of ***Jason***, there was no fully-fledged interpreter for AgentSpeak available. Also, being based on Java and available *Open Source* makes ***Jason*** particularly interesting for the development of multi-agent system for large scale applications.

3 *Jason*'s AgentSpeak Language

3.1 Syntax

The BNF grammar below gives the AgentSpeak syntax that is accepted by ***Jason***. Below, <ATOM> is an identifier beginning with an lowercase letter or '.', <VAR> (i.e., a variable) is an identifier beginning with an uppercase letter, <NUMBER> is any integer or floating-point number, and <STRING> is any string enclosed in double quote characters as usual.

```
agent             → ( init_bels | init_goals )* plans
init_bels         → beliefs rules
beliefs           → ( literal "." )*
rules             → ( literal ":-" log_expr ".")*
init_goals        → ( "!" literal "." )*
plans             → ( plan )*
plan              → [ "@" atomic_formula ] triggering_event [ ":" context ] [ "<-" body ] "."
triggering_event  → ( "+" | "-" ) [ "!" | "?" ] literal
literal           → [ "~" ] atomic_formula
context           → log_expr  |  "true"
log_expr          → simple_log_expr
                  |  "not" log_expr
                  |  log_expr "&" log_expr
                  |  log_expr "|" log_expr
                  |  "(" log_expr ")"
simple_log_expr   → ( literal | rel_expr | <VAR> )
body              → body_formula ( ";" body_forumula )*
                  |  "true"
body_formula      → ( "!" | "?" | "+" | "-" | "-+" ) literal
                  |  atomic_formula
                  |  <VAR>
                  |  rel_expr
atomic_formula    → ( <ATOM> | <VAR> ) [ "(" list_of_terms ")" ] [ "[" list_of_terms "]" ]
list_of_terms     → term ( "," term )*
term              → literal
                  |  list
                  |  arithm_expr
                  |  <VAR>
                  |  <STRING>
list              → "[" [ term ( "," term )* [ "|" ( list | <VAR> ) ] ] "]"

rel_expr          → rel_term ("<"|"<="|">"|">="|"=="|"\\=="|"=") rel_term
rel_term          → (literal|arithm_expr)
arithm_expr       → arithm_term [ ( "+" | "-" ) arithm_expr ]
arithm_term       → arithm_factor [ ( "*" | "/" | "div" | "mod" ) arithm_term ]
arithm_factor     → arithm_simple [ "**" arithm_factor ]
arithm_simple     → <NUMBER>
                  |  <VAR>
                  |  "-" arithm_simple
                  |  "(" arithm_expr ")"
```

N.B.: This grammar is a slightly more readable version then the one actually used in the parser. The BNF for the grammar used in the parsers can be found in file `AS2JavaParser.html` located in the `doc` folder of the distribution.

Also, in the production rule for beliefs, note that a semantic error is generated if the literal was not ground.

The main differences to the original AgentSpeak(L) language are as follows. Wherever an atomic formulæ[1] was allowed in the original language, here a literal is used instead. This is either an atomic formulæ `p(t1,...,tn)`, $n \geq 0$, or `~p(t1,...,tn)`, where '`~`' denotes strong negation[2]. Default negation is used in the context of plans, and is denoted by '`not`' preceding a literal. The context is therefore a conjunction of default literals. For more details on the concepts of strong and default negation, plenty of references can be found, e.g., in the introductory chapters of [17]. Terms now can be variables, lists (with Prolog syntax), as well as integer or floating point numbers, and strings (enclosed in double quotes as usual); further, any atomic formulæ can be treated as a term, and (bound) variables can be treated as literals (this became particularly important for introducing communication, but can be useful for various things). Infix relational operators, as in Prolog, are allowed in plan contexts. Also as in Prolog, `_` is used as anonymous variable.

Also, a major change is that atomic formulæ now can have "annotations". This is a list of terms enclosed in square brackets immediately following the formula. Within the belief base, annotations are used, e.g., to register the sources of information. A term `source(`s`)` is used in the annotations for that purpose; s can be an agent's name (to denote the agent that communicated that information), or two special atoms, `percept` and `self`, that are used to denote that a belief arose from perception of the environment, or from the agent explicitly adding a belief to its own belief base from the execution of a plan body, respectively. The initial beliefs that are part of the source code of an AgentSpeak agent are assumed to be internal beliefs (i.e., as if they had a `[source(self)]` annotation), unless the belief has any explicit annotation given by the user (this could be useful if the programmer wants the agent to have an initial belief about the environment or as if it had been communicated by another agent). Fore more on the annotation of sources of information for beliefs, see [21].

Clearly, because of annotations, unification becomes slightly more complicated. When attempting to unify an atomic formula at_1 with annotations $s_{11}, \ldots, s_{1n}$ against another atomic formula $at_2[s_{21}, \ldots, s_{2m}]$, not only must at_1 unify with at_2 in the usual ways (cf. unification in Prolog), but it also must be the case that $\{s_{11}, \ldots, s_{1n}\} \subseteq \{s_{21}, \ldots, s_{2m}\}$. Suppose for example that at_1 appears in a plan context and at_2 is in the belief base. The intuition is that not only should at_1 unify with at_2, but also that all specified sources of information for at_1 should be corroborated by at_2. So, for example, `p(X)[ag1]` follows from `{p(t)[ag1,ag2]}`, but `p(X)[ag1,ag2]` does *not* follow from `{p(t)[ag1]}`. More concretely, if a plan requires, for being applicable, that a drowning person was explicitly perceived rather than communicated by another agent (which can be represented by `drowning(Person)[source(percept)]`), that follows from a belief `drowning(man)[source(percept),source(passerby)]` (i.e., that this was both perceived and communicated by a passerby). On the other hand, if the required context was that two independent sources provided the information, say `cheating(Person)[source(witness1),source(witness2)]`, this cannot be inferred from a belief `cheating(husband)[source(witness1)]`. Fore more on unification in the presence of annotations, see [27].

More generally, instead of an agent being a set of initial beliefs and a set of plans, agents now can also have initial goals, which should appear between the initial beliefs and the plans. Initial goals are like beliefs except that they are prefixed with "`!`", and they do not need to be ground. Another substantial change in comparison to the original, abstract AgentSpeak(L) language, is that the belief base now can have Prolog-like rules, and the same syntax used in the body of the rules can also be used in plan contexts. The syntax is slightly different from Prolog, for example "`c(X) :- a(X) & not(b(X) | c(X)).`".

The "`+`" and "`-`" operators, in plan bodies, are used for addition and deletion of beliefs that work as "mental notes" of the agent itself (and receive the "`self`" annotation as described above). The operator "`-+`" adds a belief after removing (the first) existing occurrence of that belief in the belief base. For example, "`-+a(X+1)`" first removes `a(_)` from, and adds `a(X+1)` to, the belief base; this is quite useful in practice (e.g., for counter-like beliefs).

Variables now can be used in most places where an atomic formula is expected, including triggering events, context and body of plans. An abstract example of the various places where a variable can be used is:

```
+P[source(A),hinders(G)] : .desire(G) <- !~P; .send(A,tell,~P).
```

[1] Recall that actions are special atomic formulæ with an action symbol rather than a predicate symbol. What we say next only applies to usual predicates, not actions.

[2] Note that for an agent that uses Closed-World Assumption, all the user has to do is not to use literals with strong negation anywhere in the program, nor negated percepts in the environment (see "Creating Environments" under Section 5.4).

which means that whenever any new belief P is informed to our agent by an agent A, and that belief P is annotated with the fact that it is known to hinder a goal that this agent currently has, it tries to achieve a state of affairs in which the negation of that belief P is true, but at least it informs A that now ~P is true instead. In the context or body, one can also use a variable in place of an atomic formula but keep the annotations explicit, as in the following examples where unification is to be attempted:

```
X[a,b,c] = p[a,b,c,d] (unifies and X is p)
p[a,b] = X[a,b,c]     (unifies and X is p)
X[a,b] = p[a]         (does not unify)
p[a,b] = X[a]         (does not unify)
```

In fact, annotations can be treated as an ordinary lists, so it is possible to use constructs such as:

```
p(t)[a,b,c] = p(t)[b|R]        (unifies and R is [a,c])
```

As in Prolog, the =.. operator can be used to (de)construct literals, the syntax being <literal> =.. <list>, where <list> is [<functor>, <list of terms>, <list of annots>], for example:

```
p(t1,t2)[a1,a2] =.. L         (L is [p,[t1,t2],[a1,a2]])
X =.. [p,[t1,t2],[a1,a2]]     (X is p(t1,t2)[a1,a2])
```

Note that "true" in the context or body of a plan (to denote empty context or body) can now be omitted, so

```
+e : c <- true.
+e : true <- !g.
+!e : true <- true.
```

can be written as

```
+e : c.
+e <- !g.
+!e.
```

Further, plans have labels, as first proposed in [3]. However, a plan label can now be any atomic formula, including annotations, although we suggest that plan labels use annotations (if necessary) but have a predicate symbol of arity 0, as in `aLabel` or `anotherLabel[chanceSuccess(0.7), expectedPayoff(0.9)]`. Annotations in formulæ used as plan labels can be used for the implementation of sophisticated applicable plan (i.e., option) selection functions. Although this is not yet provided with the current distribution of ***Jason***, it is straightforward for the user to define, e.g., decision-theoretic selection functions; that is, functions which use something like expected utilities annotated in the plan labels to choose among alternative plans. The customisation of selection functions is done in Java (by choosing a plan from a list received as parameter by the selection functions), and is explained in Section 5.1. Also, as the label is part of an instance of a plan in the set of intentions, and the annotations can be changed dynamically, this provides all the means necessary for the implementation of efficient intention selection functions, as the one proposed in [3]. However, this also is not yet available as part of ***Jason***'s distribution, but can be set up by users with some customisation.

Note that for an agent that uses Closed-World Assumption, all the user has to do is not to use literals with strong negation anywhere in the program, nor negated percepts in the environment (see Section 5.4).

Events for handling plan failure are already available in ***Jason***, although they are not formalised in the semantics yet. If an action fails or there is no applicable plan for a subgoal in the plan being executed to handle an internal event with a goal addition +!g, then the whole failed plan is removed from the top of the intention and an internal event for -!g associated with that same intention is generated. If the programmer provided a plan that has a triggering event matching -!g and is applicable, such plan will be pushed on top of the intention, so the programmer can specify in the body of such plan how that particular failure is to be handled. If no such plan is available, the whole intention is discarded and a warning is printed out to the console. Effectively, this provides a means for programmers to "clean up" after a failed plan and before "backtracking" (that is, to make up for actions that had already been executed but left things in an inappropriate state for next attempts to achieve the goal). For example, for an agent that persist on a goal !g for as long as there are applicable plans for +!g, suffices it to include a plan `-!g : true <- !g.` in the plan library. Note that the body can be empty as a goal is only removed from the body of a plan when the intended means chosen for that goal finishes successfully. It is also simple to specify a plan which, under specific

condition, chooses to drop the intention altogether (by means of a standard internal action mentioned below).

Finally, as also introduced in [3], *internal actions* can be used both in the context and body of plans. Any action symbol starting with '`.`', or having a '`.`' anywhere, denotes an internal action. These are user-defined actions which are run internally by the agent. We call them "internal" to make a clear distinction with actions that appear in the body of a plan and which denote the actions an agent can perform in order to change the shared environment (by means of its "effectors"). In ***Jason***, internal actions are coded in Java, as explained in Section 5.3, or indeed other programming language through the use of JNI (Java Native Interface), and they can be organised in libraries of actions for specific purposes (the string to the left of '`.`' is the name of the library; standard internal actions have an empty library name). The next section mentions some of the standard internal actions that are distributed with ***Jason***.

3.2 Standard Internal Actions

We no longer include these in the manual as they are well documented in the API. To see a list of all available standard internal actions, open file `doc/api/index.html` of the ***Jason*** distribution using a web browser, then click on the `jason.stdlib` pacakge.

3.3 Standard Plan Annotations

As mentioned above, annotations in plan labels can be used to associate meta-level information about the plans, so that the user can write selection functions that access such information for choosing among various relevant/applicable plans or indeed intentions. ***Jason*** provides two pre-defined annotations which, when place in the annotations of a plan's label, affect the way that plan is interpreted by Jason. These pre-defined plan label annotations are:

`atomic`: if an instance of a plan with an `atomic` annotation is chosen for execution by the intention selection function, this intention will be selected for execution in the subsequent reasoning cycles *until that plan is finished* — note that this overrides any decisions of the intention selection function in subsequent cycles (in fact, the interpreter does not even call the intention selection function after an `atomic` plan began to be executed). This is quite handy in cases where the programmer needs to guarantee that no other intention of that agent will be executed inbetween the execution of the formulæ in the body of that plan.

`breakpoint`: this is very handy in debugging: if the `debug` mode is begin used (see Section 4.2), as soon as any of the agents start executing an intention with an instance of a plan that has a `breakpoint` annotation, the execution stops and the control goes back to the user, who can then use the "step" and "run" buttons to carry on the execution.

`all_unifs`: is used to include all possible unifications that make a plan applicable in the set of applicable plans. Normally, for one given plan, only the first unification found is included in the set of applicable plans. In normal circumstances, the applicable-plan selection function is used to choose between *different* plans, all of which are applicable. If you have created a plan for which you want the applicable-plan selection function to consider which is the best unification to be used as intended means for the given event, then you should included this special annotation in the plan label.

4 Defining and Running Multi-Agent Systems

In this chapter, we explain how the user can define a system of multiple AgentSpeak agents to be run with ***Jason***. A multi-agent system should have an environment where all AgentSpeak agents will be situated, and a set of instances of AgentSpeak agents. The environment should be programmed in Java[1], as explained in the next chapter. The configuration of the whole multi-agent system is given by a very simple text file, as described next.

4.1 Syntax

The BNF grammar below gives the syntax that can be used in the configuration file of a multi-agent system. Configuration files must have a name with extension `.mas2j`. Below, `<NUMBER>` is used for integer numbers, `<ASID>` are AgentSpeak identifiers, which must start with a lowercase letter, `<ID>` is any identifier (as usual), and `<PATH>` is as required for defining file pathnames in ordinary operating systems.

```
mas             → "MAS" <ID> "{"
                      [ infrastructure ]
                      [ environment ]
                      [ exec_control ]
                      agents
                  "}"
infrastructure  → "infrastructure" ":" <ID>
environment     → "environment" ":" <ID> [ "at" <ID> ]
exec_control    → "executionControl" ":" <ID> [ "at" <ID> ]
agents          → "agents" ":" ( agent )+
agent           → <ASID>
                  [ filename ]
                  [ options ]
                  [ "agentArchClass" <ID> ]
                  [ "beliefBaseClass" <ID> ]
                  [ "agentClass" <ID> ]
                  [ "#" <NUMBER> ]
                  [ "at" <ID> ]
                  ";"
filename        → [ <PATH> ] <ID>
options         → "[" option ( "," option )* "]"
option          → "events" "=" ( "discard" | "requeue" | "retrieve" )
                | "intBels" "=" ( "sameFocus" | "newFocus" )
                | "nrcbp" "=" <NUMBER>
                | "verbose" "=" <NUMBER>
                | <ID> "=" ( <ID> | <STRING> | <NUMBER> )
```

The `<ID>` used after the keyword `MAS` is the name of the society; this is used, among other things, in the name of the scripts that are automatically generated to help the user compile and run the system, as described in the next two sections. The keyword `infrastructure` is used to specify which of the underlying infrastructures for running a multi-agent system will be used. The options currently available in ***Jason***'s distribution are either "`Centralised`" or "`Saci`": the former is the default option, and the latter should be used if you need (some) agents to run on different machines over a network.

[1] In the MAS-SOC social simulation tool, environments can be defined in ELMS, by means of a graphical interface, where also AgentSpeak agents can be defined. MAS-SOC should be made available soon, and is described in [9].

Users may define other system infrastructures, although these two should cover for most general purposes.

Next an `environment` can be specified. This is simply the name of Java class that was used for programming the environment, normally. Although not indicated explicitly in the grammar, it is possible to specify arguments (as in a Java method invocation), in which case the environment implementation will receive the data in the init method, which is useful for initialisation of the simulated environment for a given multi-agent systems configuration (see the ***Jason***'s FAQ for more details on environment with parameters). Note that an optional host name where the environment will run can be specified. This is only available if you use SACI as infrastructure. Section 5.4 explains how environments can be easily created in Java. Then there is an optional `executionControl` definition, which allows user-defined management of agents' execution. Again what needs to be specified here is the name of a Java class that implements such control; ***Jason***'s FAQ explains how this is done (see `doc/faq.html` of the distribution). **Note that having an environment is optional; this might be useful e.g. for deploying a *Jason* system in a real-world environment (of course this will require customising the perception and acting functions).**

The keyword `agents` is used for defining the set of agents that will take part in the multi-agent system. An agent is specified first by its symbolic name given as an AgentSpeak term (i.e., an identifier starting with a lowercase letter); this is the name that agents will use to refer to other agents in the society (e.g. for inter-agent communication). Then, an optional filename can be given (it may include a full path, if it is not in the same directory as the `.mas2j` file) where the AgentSpeak source code for that agent is given; by default ***Jason*** assumes that the AgentSpeak source code is in file `<name>.asl`, where `<name>` is the agent's symbolic name. There is also an optional list of settings for the AgentSpeak interpreter available with ***Jason*** (these are explain below). An optional number of instances of agents using that same source code can be specified by a number preceded by `#`; if this is present, that specified number of "clones" will be created in the multi-agent system. In case more than one instance of that agent is requested, the actual name of the agent will be the symbolic name concatenated with an index indicating the instance number (starting from 1). As for the `environment` keyword, an agent definition may end with the name of a host where the agent will run (preceded by "`at`"). As before, this is only available if the SACI-based infrastructure was chosen.

The following settings are available for the AgentSpeak interpreter available in ***Jason*** (they are followed by '=' and then one of the associated keywords, where an underline denotes the option used by default):

`events`: options are either `discard`, `requeue`, or `retrieve`; the `discard` option means that external events for which there are no applicable plans are discarded (a warning is printed out to the console where SACI or the system was run), whereas the `requeue` option is used when such events should be inserted back at the end of the list of events that the agent needs to handle. An option `retrieve` is also available; when this option is selected, the user-defined `selectOption` function is called even if the set of relevant/applicable plans is empty. This can be used, for example, for allowing agents to request plan from other agents who may have the necessary know-how that the agent currently lacks, as proposed in [1].

`intBels`: options are either `sameFocus` or `newFocus`; when internal beliefs are added or removed explicitly within the body of a plan, if the associated event is a triggering event for a plan, the intended means resulting from the applicable plan chosen for that event can be pushed on top of the intention (i.e., focus of attention) which generated the event, or it could be treated as an external event (as the addition or deletions of belief from perception of the environment), creating a new focus of attention. Because this was not considered in the original version of the language, and it seems to us that both options can be useful, depending on the domain area, we left this as an option for the user. For example, by using `newFocus` the user can create, as a consequence of a single external event, different intentions that will be competing for the agent's attention.

`nrcbp`: **n**umber of **r**asoning **c**ycles **b**efore **p**erception. Normally, the AgentSpeak(L) interepreter assumes that belief revision happens before every reasoning cycle. When the environment of one particular application is not very dynamic, users may want to prevent the agents from spending time checking if there are new percepts to be "retrieved" from the environment. This configuration parameter allows the user to set the number of reasoning cycles the agent will carry out before the next time it does perception and belief revision (the default is, as in the original conception of AgentSpeak(L), 1). The parameter could, obviously, be an artificially high number so as to prevent perception and belief revision ever happening "spontaneously". If users set a high number for this parameter, it is likely they will need to code their plans to actively

do perception and belief update, as happens in various other agent oriented programming languages (note that in AgentSpeak(L) an agent will soon have incorrect information about the world as, unlike other agent programming languages, you can use internal belief changes to update the agent's perception, which is quite coherent in our point of view). The internal action `.perceive()` may be used to force the agent to do perception.

`verbose`: a number between 0 and 2 should be specified. The higher the number, the more information about that agent (or agents if the number of instances is greater than 1) is printed out in the ***Jason*** console. The default is in fact 1, not 0; verbose 1 prints out only the actions that agents perform in the environment and the messages exchanged between them.

user settings: Users can create their own settings in the agent declaration, for example:

```
... agents: ag1 [verbose=2,file="an.xml",value=45];
```

These extra parameters are stored in the Settings class and can be consulted within the programmer's classes by getUserParameter method, for example:

```
ts.getSettings().getUserParameter("file");
```

Finally, user-defined overall agent architectures and other user-defined functions to be used by the AgentSpeak interpreter for each particular agent can be specified with the keywords `agentArchClass`, `beliefBaseClass`, and `agentClass`. ***Jason*** provides great flexibility by allowing users to easily redefining the default functions used in the interpreter. The way this can be done is explained in Chapter 5.

Next, we mention the scripts that should be run for generating the Java code for the agents, compiling them, and running the multi-agent system as specified in the `.mas2j` configuration file.

4.2 Compiling and Running a Multi-Agent System

Jason comes with an IDE which is a jEdit plugin (`www.jedit.org`). All you have to do is run `jason.sh` or, under MS-Windows (if you must!) `jason.bat` — these scripts are included in the `bin` directory, see Chapter 6. The IDE provides editing facilities for the MAS configuration file as well as AgentSpeak code for the individual agents, and it also controls the execution of the MAS (all required compilation is done automatically by the IDE). The graphical interface should be very intuitive to use, so we do not include instructions on how to use the IDE here. We have kept below the original description of what goes on "behind the scene", as some users may prefer to do certain tasks by hand in particular circumstances.

Jason has three different execution modes, two of which are also available through the GUI. Further, the execution mode can be customised; ***Jason***'s FAQ explains how to do that: see the `faq.html` in the `doc` directory of ***Jason***'s distribution. The GUI allows the system to be run as usual, but also step-by-step; in the latter mode, called the debugging mode, the user has to click a button to allow all agents to perform the next reasoning cycle. Running the system in debugging mode automatically opens another tool, which is called the "mind inspector". With this tool, the user can observe the changes in the agents' mental attitudes at every reasoning cycle (and this also applies to agents distributed over a network). The interface has two buttons: "step" and "run", the latter being particular useful with `breakpoint` plan annotations (see Section 3.3).

We now described how the IDE actually runs a system. Once a multi-agent system is configured as explained in the previous section, the first thing to be done is to run the script `mas2j.sh` which is located in the `bin` directory of ***Jason***'s distribution. It takes as parameter the name of the file where the multi-agent system configuration can be found (recall that the file should have a `.mas2j` extension).

Any sophisticated agent will need customisation of selection functions and possibly overall architecture (which will be explained in the next chapter). In case the user needs to change the Java code of those classes, or the Java code for the environment, those classes need to be compiled again, an Ant script `build.xml` is create by `mas2j.sh` to compile and run the project.

N.B.: The AgentSpeak code is only read at run time, which means that if you change the AgentSpeak code but do not change the multi-agent system configuration or agent customisations, you do *not* need to run `mas2j.sh` again. Just run the system with the script provided for this, as mentioned above.

5 Agents, Overall Agent Architectures, and Environments

From the point of view of an (extended) AgentSpeak interpreter, and agent is a set of beliefs, a set of plans, some user-defined *selection functions* and *trust function* (a "socially acceptable" relation for received messages), the Belief Update Function (which updates the agent's belief base from perception of the environment, and can be customised by the user) and the Belief Revision Function (which is used, e.g., when the agent's beliefs are changed from the execution of one of its plans, and should be costumised to include algorithms found in the belief revision literature[1]) , and a "circumstance" which includes the pending events, intentions, and various other structures that are necessary during the interpretation of an AgentSpeak agent (formally given in [8]). The customisation of these aspects of an agent is explained in Section 5.1.

However, for an agent to work effectively in a multi-agent system, the AgentSpeak interpreter must be only the reasoning module within an "overall agent architecture" (we call it like this to avoid confusion with the fact that BDI is the agent architecture, but this is just the cognitive part of the agent, so to speak). Such overall agent architecture provides perception (which models the agent's "sensors"), acting (modelling the agent's "effectors"), and how the agent receives messages from other agents. These aspects can also be customised for each agent individually, as explained in Section 5.2.

The belief base itself can be customised, which can be quite important in large-scale applications. There are two customisations available with the ***Jason*** distribution: one which stores the beliefs in a text file (so as to persist the state of an agent's belief base) and another which stores some of the beliefs in a relational data base. This latter customisation can be used the access any relational data base (via JDBC). The AgentSpeak code remains the same regardless of the belief base customisation. We here do not explain how other customisations can be made, but in the `examples` folder of the ***Jason*** distribution you will find the the "persistentBelBase" example which illustrates how this can be done.

Section 5.3 gives instructions on how to define new internal actions. Finally, in section 5.4 we briefly explain how environments must be defined (this works for both the available infrastructures, `Saci` and `Centralised`). While the description for this in the manual is very brief, looking at the environments available in the examples distributed with ***Jason*** should make it clear how this should be done.

5.1 Customising Agents

Unless customised agent classes in Java are provided by the user, the following default functions are used. They show all the options for customisation that users are given by ***Jason***.

```
// add code to the methods below to change this agent's
// message acceptance relation and selection functions

public void buf(List percepts) {
    // this functions needs to update the belief base
    // with the given "percepts" and include in the set
    // of events all changes that were actually carried out
}

public List[] brf(Literal add, Literal del, Intention i) {

/*
```

[1]Note that the default belief update and revision methods that come with ***Jason*** do not guarantee consistency of the belief base; they simple perform all changes requested even if that results in a direct contradiction.

```
This method should revise the belief base with a literal to be added
(if any), a literal to be deleted (if any), and the Intention structure
that required the belief change. In the return, List[0] has the list
of actual additions to the belief base, and List[1] has the list of
actual deletions; the returned lists are used by the interpreter to
generate the appropriate internal events.
*/

   // the specific belief revision code goes here
}

public boolean socAcc(Message m) {
    // docile and benevolent agent
    return(true);
}

public Event selectEvent(Queue<Event> events) {
    // make sure the selected Event is removed from events queue
     return events.poll();
}

public Option selectOption(List<Option> options) {
    // make sure the selected Option is removed from options
    // normally there is no need to change this
    return((Option)optList.remove(0));
}

public Intention selectIntention(Queue<Intention> intentions) {
    // make sure the selected Intention is removed from 'intentions' AND
    // make sure no intention will "starve"!!!
    return intentions.poll();
}

public Message selectMessage(Queue<Message> messages) {
    // make sure the selected Message is removed from 'messages'
    return messages.poll();
}
```

It is important to emphasise that the belief update function provided with *Jason* simply updates the belief base and generates the external events (i.e., additions and deletion of beliefs from the belief base) in accordance with current percepts. It does NOT guarantee belief consistency. As it will be seen later in Section 5.4, percepts are actually sent from the environment, and they should be lists of terms stating everything that is true (and explicitly false too, if closed-world assumption is not used). It is up to the programmer of the environment to make sure that contradictions do not appear in the percepts. Also, if AgentSpeak programmers use addition of internal beliefs in the body of plans, it is their responsibility to ensure consistency. In fact, the user may, in rare occasions, be interested in modelling a "paraconsistent" agent[2], which can be easily done.

Trust functions and the message selection function are discussed in [20], and the three first selection functions are discussed in all of the AgentSpeak literature (see Chapter 2). By changing the message selection function, the user can determine that the agent will give preference to messages from certain agents, or certain types of messages, when various messages have been received during one reasoning cycle. The last selection function was deemed interesting during the development of ***Jason***. In the system infrastructure that is based on SACI, the agent can go ahead with its reasoning after requesting an action to be executed in the environment (that is, asynchronous communication is used between the agent and the environment, which is the process that effectively executes the external action). This means that we need, in the agent's circumstance, another structure which stores the feedback from the environment after the action has been executed (this tells the agent whether the attempted action execution in the environment succeeded or not). In case more than one action feedback is received from the environment during a reasoning cycle, the action selection function is used to choose which action feedback will be chosen first to be considered in the next reasoning cycle. Notice that, as with internal events, waiting for an action feedback makes an intention to become suspended. Only after

[2] The `SimpleOpenWorld` example distributed with ***Jason*** gives an abstract situation which includes this possibility.

the feedback from the environment is received, that intention can be selected again for execution (by the intention selection function).

Suppose, for example, that a user wants to redefine the event selection function only. All that needs to be done is to create the following java file called, say, `MyAgClass.java`:

```
import java.util.*;
import jason.asSemantics.*;

/** example of agent function overriding */
public class MyAgClass extends Agent {

    public Event selectEvent(Queue<Event> events) {
        System.out.println("Selecting an event from "+events);
        return events.poll();
    }

}
```

and then specify `agentClass MyAgClass` in the agent's entry in the `.mas2j` configuration file.

5.2 Customising Agent Architectures

Similarly, the user can customise the functions defining the overall agent architecture. These functions handle the way the agent will receive percepts from the environment; how the agent gets messages sent from other agents (for speech-act based inter-agent communication); and how the agent acts on the environment (for the basic, i.e. external, actions that appear in the body of plans) – normally this is provided by the environment implementation, so this function only has to pass the action selected by the agent on to the environment.

For the perception function, it may be interesting to use the function defined in ***Jason***'s distribution and after it has received the current percepts, and then process further the list of percepts, in order to simulate faulty perception, for example. Unless, of course, the environment has been modelled so as to send different percepts to different agents, according to their perception abilities (so to speak) within the given multi-agent system (as with ELMS environments, see [9]).

The methods of the agent architecture will rely on one of the existing multi-agent infrastructure/middleware. Files `SaciAgArch.java` and `CentralisedAgArch.java` in directory `src/jason/infra` have specific implementations for these four methods, according the the underlying infrastructure (based on SACI or using threads in the local host, respectively). Looking at those files may be helpful for programmers intending to use other multi-agent infrastructures (e.g., JADE [2]), but this is not yet documented.

The following code shows the options available to the user for customisation:

```
// add code to the methods below to change this agent's
// perception, mail checking, and acting functions

public List<Literal> perceive() {
}
public void checkMail() {
}

// Executes the action <i>action</i> and, when finished,
// add it back in feedback actions.
public void act(ActionExec action, List<ActionExec> feedback) {
}
```

To customise any of the functions mentioned above, create a Java file named, e.g., `MyAgArch.java` as follows:

```
import jason.architecture.*;

/** example of agent architecture's functions overriding */
public class MyAgArch extends AgArch {
```

```
    public List<Literal> perceive() {
        List<Literal> p = super.perceive();
        // change list p by adding/removing literals to
        // simulate faulty perception, for example
        return p;
    }

}
```

and then add `agentArchClass MyAgArch` to the agent's entry in the configuration file.

5.3 Defining New Internal Actions

An important construct for allowing AgentSpeak agents to remain at the right level of abstraction, is that of internal actions. As suggested in [3], internal actions that start with '.' are part of a standard library of internal actions; these are distributed with ***Jason***, in directory `src/jason/stdlib`. Internal actions defined by users should be organised in specific libraries, as described below. In the AgentSpeak code, the action is accessed by the name of the library, followed by '.', followed by the name of the action.

Libraries are defined as Java packages and each action in the user library should be a Java class within that package, the name of the package and class are the names of the library and action as it will be used in the AgentSpeak programs. Recall that all identifiers starting with an uppercase letter in AgentSpeak denote variables, so the name of the library *must* start with a lowercase letter. All classes defining internal actions should look something like:

```
package <LibraryName>;

import jason.asSyntax.*;
import jason.asSemantics.*;

public class <ActionName> extends DefaultInternalAction {

    public Object execute( TransitionSystem ts, Unifier un,
                           Term[] args ) throws Exception {
        ...
    }
}
```

It is important that the class has an execute method declared *exactly* as above. Each agent creates an instance of the class the first time that internal action is executed; this means that an internal action can keep a state *within the agent*.

As expected, `<LibraryName>` in the example above is the name of the library (the newly created directory inside `ulibs`), and `<ActionName>` is the name of the particular action being defined in this Java file. The internal action's arguments are passed as an array of Termss. Note that this is the third argument of the execute method. The first argument is the transition system (as defined by the operational semantics of AgentSpeak), which contains all information about the current state of the agent being interpreted. The second is the unifying function currently determined by the execution of the plan, or the checking of whether the plan is applicable[3]; the unifying functions is important in case the value binded to AgentSpeak variables need to be used in the implementation of the action.

This makes ***Jason***'s AgentSpeak extensible in a very straightforward way. Looking at the examples in `stdlib` should make it fairly easy for users to implement their own internal actions.

5.4 Creating Environments

Besides programming a set of AgentSpeak agents, and providing the multi-agent system configuration file, in order to have a complete computational system[4], the user needs to create an environment. This is done directly in Java, and a class providing an environment, typically looks like:

[3] This depends on whether the internal action being run appears int the body or the context of a plan.

[4] AgentSpeak can in principle be used for agents or robots that act on real environments. Models of environments are of course necessary if the whole system is an application aimed at running on (a network of) computers.

```
import java.util.*;
import jason.asSyntax.*;
import jason.environment.*;

public class <EnvironmentName> extends Environment {

    // any class members you may need

    public <EnvironmentName>() {
        // setting initial percepts ...
        addPercept(Literal.parseLiteral("p(a)"));

        // if you are using open-world...
        addPercept(Literal.parseLiteral("~q(b)"));

        // if this is to be perceived only by agent ag1
        addPercept("ag1", Literal.parseLiteral("p(a)"));

        ...
    }


    public boolean executeAction(String ag, Structure act) {
        ...
    }

    // any other methods you may need
}
```

where <EnvironmentName> is the name of the environment class specified in the multi-agent system configuration file.

Both positive and negative percepts (for creating an open-world multi-agent system) are included in the same list (negative percepts are literals with the strong negation operator '~'). It is normally appropriate to use the class constructor to initialise the lists of percepts, and use the parseLiteral method of the Literal class, as in the environments available in the `examples` directory of ***Jason***'s distribution. Further, the Environment class supports individualised perception, greatly facilitating the task for the user to give certain percepts only to certain agents. The following methods can be used in the user's environment:

`addPercept(L)`: add literal `L` to the global list of percepts; that is, *all agents* will perceive `L`;

`addPercept(A,L)`: add literal `L` to the list of percepts that are exclusive to agent `A`; that is, only `A` will be able to perceive `L`;

`removePercept(L)`: remove literal `L` from the global list of percepts;

`removePercept(A,L)`: remove literal `L` from the list of percepts that are exclusive to agent `A`;

`clearPercepts()`: delete all percepts in the global list of percepts;

`clearPercepts(A)`: delete all percepts in the list of percepts that are exclusive to agent `A`.

Note that only instances of the class Literal, which is part of the jason package, should be added to the lists of percepts using these methods! Normally, you should not add any annotations here, as all percepts will be received by the agents with a [source(percept)] annotation. Also, the access to the lists of percepts are automatically synchronised, but depending on how you model your environment you may need further synchronisation in the executeAction() method.

Most of the code for building environments should be (referenced at) the body of the method executeAction which must be declared as described above. Whenever an agent tries to execute a basic action (those which are supposed to change the state of the environment), the name of the agent and a Term representing the chosen action are sent as parameter to this method. So the code for this method needs to check the Term (having the form of a Prolog "structure") which represents the action (and any required parameters) being executed, and check which is the agent attempting to execute the action, then do whatever is necessary in that particular model of an environment – normally, this means changing the percepts, i.e., what is true or false of the environment is changed according to the actions being performed. Note that the execution of an action needs to return a boolean value,

stating whether the agent's attempt at performing that action on the environment was successful or not. A plan fails if any basic action attempted by the agent fails.

Just a few more comments on environments with individualised perception, i.e., the fact that in programming the environment you can determine what subset of the environment properties will be perceptible to individual agents. Recall that within and agent's overall architecture you can further customise what beliefs the agent will actually aquire from what it perceives. Intuitively, the environment properties available to an agent from the environment definition itself are associated to what is actually perceptible at all in the environment (for example, if something is behind my office wall, I cannot see it). The customisation at the agent overall architecture level should be used for simulating faulty perception (i.e., even though something is perceptible for that agent in that environment, it may still not include some of those properties in its belief revision process).

6 Installation

Jason is distributed *Open Source*[1] under GNU LGPL, and is kindly hosted by SourceForge (`http://sourceforge.net`). To install ***Jason***, first download the latest distribution from `http://jason.sourceforge.net`, then uncompress the downloaded file (in Linux, `tar xzf <filename>` should do). It is probably a good idea to include ***Jason***'s `bin` directory in your `$PATH` environment variable as well. Assuming you are already able to compile and run Java programs in your machine, that is all you have to do to run your AgentSpeak multi-agent system with ***Jason***.

It should be helpful having a look at the `examples` directory of ***Jason***'s distribution. They have the same examples (including the ones in Section 2.4) using the `Centralised` and `Saci` infrastructure, respectively.

If you choose to change the jason Java package, you will have to install Apache Ant (`http://ant.apache.org/`), then run `ant compile` at the main directory of the distribution. **Remember that by GNU LGPL you are only allowed to change copies of the original files which are given *different names* from the official distribution.**

[1] Check the Open Source Initiative website at `http://www.opensource.org/`.

7 Conclusion

Jason was implemented in Java by Rafael H. Bordini and Jomi F. Hübner, with contributions from various colleagues. Research in the area of agent-oriented programming languages is still incipient, so we expect much progress from research in the area. We hope to incorporate new techniques arising from research in the area into future releases of ***Jason***.

Note that ***Jason*** is distributed completely on an "as is" basis. Because it has been developed during our spare time, we cannot guarantee much support. However, if you have questions or bug reports, you are welcome to use the mailing lists at SourceForge:

- `jason-announcement@lists.sourceforge.net` (where we announce new releases and important news about ***Jason***)
- `jason-users@lists.sourceforge.net` (for questions about using ***Jason***)
- `jason-bugs@lists.sourceforge.net` (to report bugs)

We particularly welcome comments and experiences on building (large) applications with ***Jason***, and issues with the language that users are face with during their experimentation with AgentSpeak.

Acknowledgements

As seen from the various references throughout this document, the research on AgentSpeak has been carried out with the help of many colleagues. We are grateful for the many contributions received over the last few years from: Davide Ancona, Marcelo G. de Azambuja, Deniel M. Basso, Ana L.C. Bazzan, Antônio Carlos da Rocha Costa, Guilherme Drehmer, Michael Fisher, Rafael de O. Jannone, Romulo Krafta, Victor Lesser, Rodrigo Machado, Joyce Martins, Viviana Mascardi, Álvaro F. Moreira, Fabio Y. Okuyama, Denise de Oliveira, Carmen Pardavila, Marios Richards, Maíra R. Rodrigues, Rosa M. Vicari, Renata Vieira, Willem Visser, Michael Wooldridge.

In the past, some research projects which contributed to the development of AgentSpeak were financed by the Brazilian agencies CNPq and FAPERGS. The initial releases of ***Jason*** were developed while Rafael Bordini was at the University of Liverpool. The *Model Checking for Mobility* project at the University of Liverpool (coordinated by Michael Fisher), for which the initial AgentSpeak model in Java was developed, is supported by Marie Curie Fellowships of the European Community programme *Improving Human Potential* under contract number HPMF-CT-2001-00065.

Bibliography

[1] Davide Ancona, Viviana Mascardi, Jomi F. Hübner, and Rafael H. Bordini. Coo-AgentSpeak: Cooperation in AgentSpeak through plan exchange. In Nicholas R. Jennings, Carles Sierra, Liz Sonenberg, and Milind Tambe, editors, *Proceedings of the Third International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-2004), New York, NY, 19-23 July*, pages 698-705, New York, NY, 2004. ACM Press.

[2] Fabio Bellifemine, Federico Bergenti, Giovanni Caire, and Agostino Poggi. JADE — a java agent development framework. In Bordini et al. [4], chapter 5, pages 125-147.

[3] Rafael H. Bordini, Ana L. C. Bazzan, Rafael O. Jannone, Daniel M. Basso, Rosa M. Vicari, and Victor R. Lesser. AgentSpeak(XL): Efficient intention selection in BDI agents via decision-theoretic task scheduling. In Cristiano Castelfranchi and W. Lewis Johnson, editors, *Proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-2002), 15-19 July, Bologna, Italy*, pages 1294-1302, New York, NY, 2002. ACM Press.

[4] Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni, editors. *Multi-Agent Programming: Languages, Platforms and Applications*. Number 15 in Multiagent Systems, Artificial Societies, and Simulated Organizations. Springer-Verlag, 2005.

[5] Rafael H. Bordini, Michael Fisher, Carmen Pardavila, and Michael Wooldridge. Model checking AgentSpeak. In Jeffrey S. Rosenschein, Tuomas Sandholm, Michael Wooldridge, and Makoto Yokoo, editors, *Proceedings of the Second International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-2003), Melbourne, Australia, 14-18 July*, pages 409-416, New York, NY, 2003. ACM Press.

[6] Rafael H. Bordini, Michael Fisher, Willem Visser, and Michael Wooldridge. Verifiable multi-agent programs. In *Proceedings of the First International Workshop on Programming Multiagent Systems: languages, frameworks, techniques and tools (ProMAS-03), held with AAMAS-03, 15 July, 2003, Melbourne, Australia*, 2003. To appear as a volume in Springer's LNAI series.

[7] Rafael H. Bordini and Álvaro F. Moreira. Proving the asymmetry thesis principles for a BDI agent-oriented programming language. In Jürgen Dix, João Alexandre Leite, and Ken Satoh, editors, *Proceedings of the Third International Workshop on Computational Logic in Multi-Agent Systems (CLIMA-02), 1st August, Copenhagen, Denmark*, Electronic Notes in Theoretical Computer Science 70(5). Elsevier, 2002. URL: ‹http://www.elsevier.nl/locate/entcs/volume70.html›. CLIMA-02 was held as part of FLoC-02. This paper was originally published in Datalogiske Skrifter number 93, Roskilde University, Denmark, pages 94-108.

[8] Rafael H. Bordini and Álvaro F. Moreira. Proving BDI properties of agent-oriented programming languages: The asymmetry thesis principles in AgentSpeak(L). *Annals of Mathematics and Artificial Intelligence*, 42(1-3):197-226, September 2004. Special Issue on Computational Logic in Multi-Agent Systems.

[9] Rafael H. Bordini, Fabio Y. Okuyama, Denise de Oliveira, Guilherme Drehmer, and Romulo C. Krafta. The MAS-SOC approach to multi-agent based simulation. In Gabriela Lindemann, Daniel Moldt, and Mario Paolucci, editors, *Proceedings of the First International Workshop on Regulated Agent-Based Social Systems: Theories and Applications (RASTA'02), 16 July, 2002, Bologna, Italy (held with AAMAS02) — Revised Selected and Invited Papers*, number 2934 in Lecture Notes in Artificial Intelligence, pages 70-91, Berlin, 2004. Springer-Verlag.

[10] Rafael H. Bordini, Willem Visser, Michael Fisher, Carmen Pardavila, and Michael Wooldridge. Model checking multi-agent programs with CASP. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *Proceedgins of the Fifteenth Conference on Computer-Aided Verification (CAV-2003), Boulder,*

CO, 8-12 July, number 2725 in Lecture Notes in Computer Science, pages 110-113, Berlin, 2003. Springer-Verlag. Tool description.

[11] Mark d'Inverno, David Kinny, Michael Luck, and Michael Wooldridge. A formal specification of dMARS. In Munindar P. Singh, Anand S. Rao, and Michael Wooldridge, editors, *Intelligent Agents IV—Proceedings of the Fourth International Workshop on Agent Theories, Architectures, and Languages (ATAL-97), Providence, RI, 24-26 July, 1997*, number 1365 in Lecture Notes in Artificial Intelligence, pages 155-176. Springer-Verlag, Berlin, 1998.

[12] Mark d'Inverno and Michael Luck. Engineering AgentSpeak(L): A formal computational model. *Journal of Logic and Computation*, 8(3):1-27, 1998.

[13] Michael P. Georgeff and A. L. Lansky. Reactive reasoning and planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI'87), 13-17 July, 1987, Seattle, WA*, pages 677-682, Manlo Park, CA, 1987. AAAI Press / MIT Press.

[14] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[15] Jomi Fred Hübner. *Um Modelo de Reorganização de Sistemas Multiagentes*. PhD thesis, Universidade de São Paulo, Escola Politécnica, 2003.

[16] David Kinny. The distributed multi-agent reasoning system architecture and language specification. Technical report, Australian Artificial Intelligence Institute, Melbourne, Australia, 1993.

[17] João Alexandre Leite. *Evolving Knowledge Bases: Specification and Semantics*, volume 81 of *Frontiers in Artificial Intelligence and Applications, Dissertations in Artificial Intelligence*. IOS Press/Ohmsha, Amsterdam, 2003.

[18] Rodrigo Machado and Rafael H. Bordini. Running AgentSpeak(L) agents on SIM_AGENT. In John-Jules Meyer and Milind Tambe, editors, *Intelligent Agents VIII - Proceedings of the Eighth International Workshop on Agent Theories, Architectures, and Languages (ATAL-2001), August 1-3, 2001, Seattle, WA*, number 2333 in Lecture Notes in Artificial Intelligence, pages 158-174, Berlin, 2002. Springer-Verlag.

[19] Álvaro F. Moreira and Rafael H. Bordini. An operational semantics for a BDI agent-oriented programming language. In John-Jule Ch. Meyer and Michael J. Wooldridge, editors, *Proceedings of the Workshop on Logics for Agent-Based Systems (LABS-02), held in conjunction with the Eighth International Conference on Principles of Knowledge Representation and Reasoning (KR2002), April 22-25, Toulouse, France*, pages 45-59, 2002.

[20] Álvaro F. Moreira, Renata Vieira, and Rafael H. Bordini. Extending the operational semantics of a BDI agent-oriented programming language for introducing speech-act based communication. In *Proceedings of the Workshop on Declarative Agent Languages and Technologies (DALT-03), held with AAMAS-03, 15 July, 2003, Melbourne, Australia*, 2003. To appear as a volume in Springer's LNAI series.

[21] Álvaro F. Moreira, Renata Vieira, and Rafael H. Bordini. Extending the operational semantics of a BDI agent-oriented programming language for introducing speech-act based communication. In João Leite, Andrea Omicini, Leon Sterling, and Paolo Torroni, editors, *Declarative Agent Languages and Technologies, Proceedings of the First International Workshop (DALT-03), held with AAMAS-03, 15 July, 2003, Melbourne, Australia (Revised Selected and Invited Papers)*, number 2990 in Lecture Notes in Artificial Intelligence, pages 135-154, Berlin, 2004. Springer-Verlag.

[22] Anand S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In Walter Van de Velde and John Perram, editors, *Proceedings of the Seventh Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'96), 22-25 January, Eindhoven, The Netherlands*, number 1038 in Lecture Notes in Artificial Intelligence, pages 42-55, London, 1996. Springer-Verlag.

[23] Anand S. Rao and Michael P. Georgeff. Decision procedures for BDI logics. *Journal of Logic and Computation*, 8(3):293-343, 1998.

[24] Yoav Shoham. Agent-oriented programming. *Artificial Intelligence*, 60:51-92, 1993.

[25] Munindar P. Singh, Anand S. Rao, and Michael P. Georgeff. Formal methods in DAI: Logic-based representation and reasoning. In Gerhard Weiß, editor, *Multiagent Systems—A Modern Approach to Distributed Artificial Intelligence*, chapter 8, pages 331-376. MIT Press, Cambridge, MA, 1999.

[26] Jorge A. R. Torres, Luciana P. Nedel, and Rafael H. Bordini. Using the BDI architecture to produce autonomous characters in virtual worlds. In Thomas Rist, Ruth Aylett, Daniel Ballin, and Jeff Rickel, editors, *Proceedings of the Fourth International Conference on Interactive Virtual Agents (IVA 2003), Irsee, Germany, 15-17 September*, number 2792 in Lecture Notes in Artificial Intelligence, pages 197-201, Heidelberg, 2003. Springer-Verlag. Short paper.

[27] Renata Vieira, Alvaro Moreira, Michael Wooldridge, and Rafael H. Bordini. On the formal semantics of speech-act based communication in an agent-oriented programming language. *Submitted article, to appear*, 2005.

[28] Willem Visser, Klaus Havelund, Guillaume Brat, and SeungJoon Park. Model checking programs. In *Proceedings of the Fifteenth International Conference on Automated Software Engineering (ASE'00), 11-15 September, Grenoble, France*, pages 3-12. IEEE Computer Society, 2000.

[29] Michael Wooldridge. *Reasoning about Rational Agents*. The MIT Press, Cambridge, MA, 2000.

[30] Michael Wooldridge. *An Introduction to MultiAgent Systems*. John Wiley & Sons, 2002.