BDI Agent Programming in AgentSpeak Using *Jason*

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

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

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

Abstract. This paper is based on the tutorial given as part of the tutorial programme of CLIMA-VI. The tutorial aimed at giving an overview of the various features available in ***Jason***, a multi-agent systems development platform that is based on an interpreter for an extended version of AgentSpeak. The BDI architecture is the best known and most studied architecture for cognitive agents, and AgentSpeak is an elegant, logic-based programming language inspired by the BDI architecture.

1 Introduction

The BDI agent architecture [27, 33, 29] has been a central theme in the multi-agent systems literature since the early 1990's. After a period of relative decline, it seems BDI agents are back in vogue, with various conference papers referring again to elements of the BDI theory. Arguably, that theory provides the grounding for some of the essential features of autonomous agents and multi-agent systems, so it will always have an important role to play in the research in this area. Besides, the software industry is beginning to use technologies that clearly derived from the academic work on BDI-based systems.

AgentSpeak is an elegant agent-oriented programming language based on logic programming, and inspired by the work on the BDI architecture [27] and BDI logics [28] as well as on practical implementations of BDI systems such as PRS [16] and dMARS [17]. However, in its original definition [26], AgentSpeak was just an abstract programming language. For these reasons, our effort in developing ***Jason*** was very much directed towards using AgentSpeak as the basis, but also providing various extensions that are required for the practical development of multi-agent systems.

The elegance of the AgentSpeak core of the language interpreted by ***Jason*** makes it an interesting tool both for teaching multi-agent systems as well as the practical development of multi-agent systems (in particular in association with existing agent-oriented software engineering methodologies for BDI-like systems). ***Jason*** is implemented in Java and is available Open Source at `http://jason.sourceforge.net`. Some of the features available in ***Jason*** are:

- speech-act based inter-agent communication (and annotation of beliefs with information sources);
- annotations on plan labels, which can be used by elaborate (e.g., decision theoretic) selection functions;
- the possibility to run a multi-agent system distributed over a network (using SACI, or some other agent middleware);
- fully customisable (in Java) selection functions, trust functions, and overall agent architecture (perception, belief-revision, inter-agent communication, and acting);
- straightforward extensibility (and use of legacy code) by means of user-defined "internal actions";
- clear notion of *multi-agent environments*, which can be implemented in Java (this can be a simulation of a real environment, e.g., for testing purposes before the system is actually deployed).

This paper is based on a CLIMA-VI tutorial which aimed at giving an overview of the various features available in ***Jason***. It is intended for a general audience although some parts might be clearer for readers familiar with agent-oriented programming. To keep the paper at a reasonable size, we only describe the main features of ***Jason***, so that readers can assess whether ***Jason*** might be of interest, rather than aiming at a didactic presentation. For the interested reader, we give here plenty of references to other papers and documentation where more detail and examples can be found; a general reference giving a longer overview is [9], and see [8] for details. The paper is organised as follows. Section 2 presents the *language* interpreted by ***Jason***, and its informal semantics is given in Section 3. Some other features of the language related to multi-agent communication and interaction are discussed in Section 4. We then present the main feature of the *platform* which facilitate the development of multi-agent systems in Section 5. Section 6 discusses various issues (such as formal verification) and we then make some final remarks.

2 ***Jason*** Extension of the AgentSpeak Language

The AgentSpeak(L) programming language was introduced in [26]. 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 [33].

An AgentSpeak agent is defined by a set of *beliefs* giving the initial state of the agent's *belief base*, which is a set of ground (first-order) atomic formulæ, and a set of plans which form its *plan library*. Before explaining exactly how a plan is written, we need to introduce the notions of goals and triggering events. AgentSpeak distinguishes two types of *goals*: achievement goals and test goals. Achievement goals are formed by an atomic formulæ prefixed with the '`!`' operator, while test goals are prefixed with the '`?`' operator. An *achievement goal* states that the agent wants to achieve a state of the world where the associated atomic formulæ is true. A *test goal* states that the agent wants to test whether the associated atomic formulæ is (or can be unified with) one of its beliefs.

An AgentSpeak agent is a reactive planning system. The events it reacts to are related either to changes in beliefs due to perception of the environment, or to changes in the agent's goals that originate from the execution of plans triggered by previous events. A *triggering event* defines which events can initiate the execution of a particular plan. Plans are written by the programmer so that they are triggered by the *addition* ('`+`') or *deletion* ('`-`') of beliefs or goals (the "mental attitudes" of AgentSpeak agents).

An AgentSpeak plan has a *head* (the expression to the left of the arrow), which is formed from a triggering event (specifying the events for which that plan is *relevant*), and a conjunction of belief literals representing a *context*. The conjunction of literals in the context must be a logical consequence of that agent's current beliefs if the plan is to be considered *applicable* at that moment in time (only applicable plans can be chosen for execution). A plan also has a *body*, which is a sequence of basic actions or (sub)goals that the agent has to achieve (or test) when the plan is triggered. Plan bodies include *basic actions* — such actions represent atomic operations the agent can perform so as to change the environment. Such actions are also written as atomic formulæ, but using a set of *action symbols* rather than predicate symbols.

Figure 1 give examples of three AgentSpeak plans, illustrating a scenario in which a robot is instructed to be especially attentive to "green patches" on rocks it observers while roving on Mars. The first plan says that whenever the rover perceives a green patch on a certain rock (a belief addition), it should try and examine that particular rock; however note that this plan can only be used (i.e., it is only applicable) in case the batteries are not too low. In order to examine the rock, it has to retrieve, from its own belief base, the coordinates it has associated with that rock (this is the test goal in the beginning of the plan's body), then achieve the goal of traversing to those coordinates and, once there, examining the rock. Recall that each of these achievement goals will trigger the execution of some other plan.

The two other plans (note the last one is only an excerpt) provide alternative courses of actions that the Mars exploration rover has to take according to what it believes about the environment when the rover has to achieve a new goal of traversing towards some given coordinates. If the rover believes that there is a safe path in that direction, then all it has to do is to take the action of moving towards those coordinates (this is a basic action via which the rover can effect changes in its environment). The alternative plan is not shown here; it should provide alternative means for the agent to reach the rock but avoiding unsafe paths.

```
+green_patch(Rock)
   : not battery_charge(low)
     <- ?location(Rock,Coordinates);
        !traverse(Coordinates);
        !examine(Rock).

+!traverse(Coords)
   : safe_path(Coords)
     <- move_towards(Coords).

+!traverse(Coords) :
   : not safe_path(Coords)
     <- ...
```

Fig. 1. Examples of AgentSpeak Plans for a Mars Rover

The main differences between the language interpreted by ***Jason*** and the original AgentSpeak(L) language described above 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 $\texttt{p(t}_1\texttt{,}\ldots\texttt{,t}_n\texttt{)}$, $n \geq 0$, or $\texttt{\textasciitilde p(t}_1\texttt{,}\ldots\texttt{,t}_n\texttt{)}$, 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 [18]. 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, 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)]` annota-

[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).

tion), 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].

Plans also 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 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. 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.

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`, it suffices to include a plan `-!g : true <- !g.` in the plan library. It is also simple to specify a plan which, under specific condition, chooses to drop the intention altogether (by means of a pre-defined internal action).

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 (in the usual jargon of the area, by means of its "effectors"). In ***Jason***, internal actions are coded in Java, or in indeed other programming languages 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).

There are several standard internal actions that are distributed with ***Jason***, but we do not mention all them here (see [8] for a complete list). To give an example, ***Jason*** has an internal action that implements KQML-like inter-agent communication. The usage is: `.send(+receiver, +illocutionary_force, +prop_content)` where each parameter is as follows. The `receiver` is simply referred to using the name given to agents in the multi-agent system (see Section 5). The `illocutionary_forces` available so far are: `tell`, `untell`, `achieve`, `unachieve`, `tellHow`, `untellHow`, `askIf`, `askOne`, `askAll`, and `askHow`. The effects of receiving messages with each of these types of illocutionary acts are explained in Section 4. Finally, the message's propositional content `prop_content` is a literal.

Another important class of standard internal actions are related to querying about the agent's current desires and intentions as well as forcing itself to drop desires or intentions. The notion of desire and intention used is exactly as formalised for AgentSpeak agents in [11]. The standard AgentSpeak language has provision for beliefs to be queried (in plan contexts and by test goals) and since our earlier extensions beliefs can be added or deleted from plan bodies. However, an equally important feature, as far as the generic BDI architecture is concerned, is for an agent to be able to check current desires/intentions and drop them under certain circumstances. In ***Jason***, this can be done by the use of certain special standard internal actions.

3 Informal Semantics

As we mentioned in the introduction, one of the important characteristics of ***Jason*** is that it implements the operational semantics of an extension of AgentSpeak. Having formal semantics also allowed us to give precise definitions for practical notions of beliefs, desires, and intentions in relation to running AgentSpeak agents, which in turn underlies the work on formal verification of AgentSpeak programs, as discussed later in this section. The formal semantics, using structural operational semantics [24] (a widely-used notation for giving semantics to programming languages) was given then improved and extended in a series of papers [20, 10, 11, 21, 31]. In particular, [31] presents a revised version of the semantics and include some of the extensions we have proposed to AgentSpeak, including rules for the interpretation of speech-act based communication. Due to space limitation, in this paper we will only provide the main intuitions behind the interpretation of AgentSpeak programs.

Besides the belief base and the plan library, the AgentSpeak 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 did not elaborate on how designers specify such functions[3]). 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 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 interpreter, which is clearly depicted in Figure 2 (reproduced from [19]). Note, however, that this is a depiction of the essential aspects of the interpreter for the original (abstract) definition of AgentSpeak; it does *not* include the extensions implemented in ***Jason***. 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 programs.

At every interpretation cycle of an agent program, the interpreter 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 interpreter, but rather a necessary component of the agent architecture.

After $\mathcal{S}_\mathcal{E}$ has selected an event, the interpreter has to unify that event with triggering events in the heads of plans. This generates the set of all *relevant plans* for that event. By checking whether the context part of the plans in that set follows from the agent's beliefs, the set of *applicable plans* is determined — these are the 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

[3] Our extension of AgentSpeak 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.

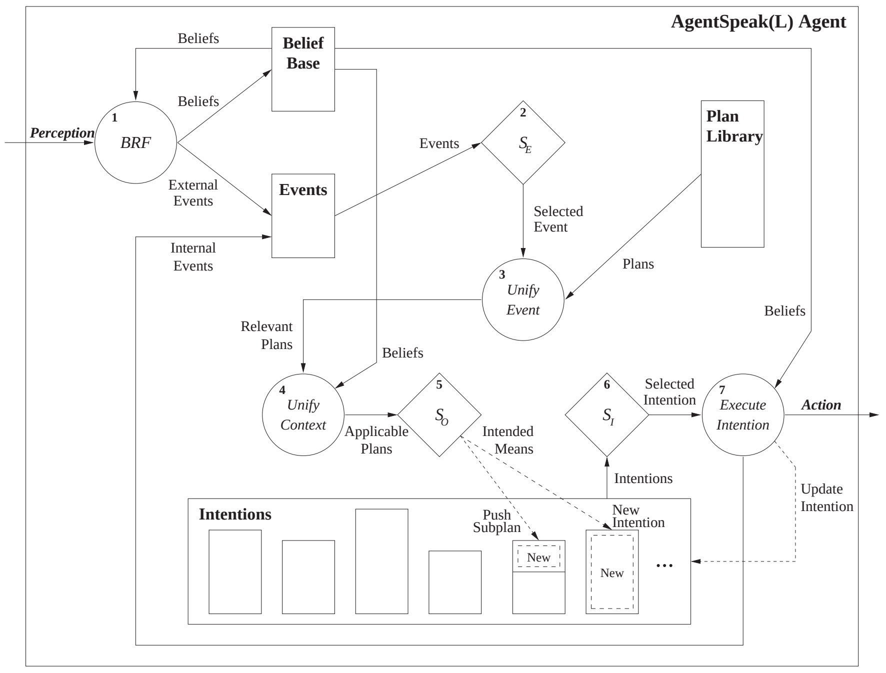


Fig. 2. An Interpretation Cycle of an AgentSpeak Program [19].

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 $\mathcal{S_I}$ 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 atomic formula 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 everything is repeated all over again, initially checking the state of the environment after agents have acted upon it, then generating the relevant events, and so forth.

4 Other Features of the Language

Agent Communication in ***Jason***

The performatives that are currently available for agent communication in ***Jason*** are largely inspired by KQML. We also include some new performatives, related to plan exchange rather than communication about propositions. The available performatives are briefly described below, where s denotes the agent that sends the message, and r denotes the agent that receives the message. Note that `tell` and `untell` can be used either for an agent to pro-actively send information to another agent, or as replies to previous `ask` messages.

`tell`: s intends r to believe (that s believes) the sentence in the message's content to be true;
`untell`: s intends r not to believe (that s believes) the sentence in the message's content to be true;
`achieve`: s requests that r try to achieve a state of the world where the message content is true;
`unachieve`: s requests that r try to drop the intention of achieving a state of the world where the message content is true;
`tellHow`: s informs r of a plan;
`untellHow`: s requests that r disregard a certain plan (i.e., delete that plan from its plan library);
`askIf`: s wants to know if the content of the message is true for r;
`askAll`: s wants all of r's answers to a question;
`askHow`: s wants all of r's plans for a triggering event;

A mechanism for receiving and sending messages asynchronously is used. Messages are stored in a mail box and one of them is processed by the agent at the beginning of a reasoning cycle. The particular message to be handled at the beginning of the reasoning cycle is determined by a selection function, which can be customised by the programmer, as three selection functions that are originally part of the AgentSpeak interpreter.

Further, in processing messages we consider a "given" function, in the same way that the selection functions are assumed as given in an agent's specification. This function defines a set of *socially acceptable* messages. For example, the receiving agent may want to consider whether the sending agent is even allowed to communicated with it (e.g., to avoid agents being attacked by malicious communicating agents). For a message with illocutionary force `achieve`, the agent will have to check, for example, whether the sending agent has sufficient social power over itself, or whether it wishes to act altruistically towards that agent and then do whatever it is being asked.

Note that notions of trust can also be programmed into the agent by considering the annotation of the sources of information during the agent's practical reasoning. When applied to `tell` messages, the function only determines if the message is to be processed at all. When the source is "trusted" (in this limited sense used here), the information source for a belief acquired from communication is annotated with that belief in the belief base, enabling further consideration on degrees of trust during the agent's reasoning.

When the function for checking message acceptance is applied to an `achieve` message, it should be programmed to return true if, e.g., the agent has a subordination relation towards the sending agent. However this "power/subordination" relation should not be interpreted with particular social or psychological nuances: the programmer defines this function so as to account for all possible reasons for an agent to do something for another agent (from actual subordination to true altruism). Similar interpretations for the result of this function when applied to other types of messages (e.g., `askIf`) can be derived easily. For more elaborate conceptions of trust and power, see [14].

In order to endow AgentSpeak agents with the capability of processing communication messages, we annotate, for each belief, what is its source. This annotation mechanism provides a very elegant notation for making explicit the sources of an agent's belief. It has advantages in terms of expressive power and readability, besides allowing the use of such explicit information in an agent's reasoning (i.e., in selecting plans for achieving goals).

Belief sources can be annotated so as to identify which was the agent in the society that previously sent the information in a message, as well as to denote internal beliefs or percepts (i.e., in case the belief was acquired through perception of the environment). By using this information source annotation mechanism, we also clarify some practical problems in the implementation of AgentSpeak interpreters relating to internal beliefs (the ones added during the execution of a plan). In the interpreter reported in [3], we dealt with the problem by creating a separate belief base where the internal beliefs were included or removed.

Due to space restriction, we do not discuss the interpretation of received messages with each of the available illocutionary forces. This is presented both formally and informally in [31].

Cooperation in AgentSpeak

Coo-BDI (Cooperative BDI, [1]) extends traditional BDI agent-oriented programming languages in many respects: the introduction of *cooperation* among agents for the retrieval of external plans for a given triggering event; the extension of plans with *access specifiers*; the extension of *intentions* to take into account the external plan retrieval mechanism; and the modification of the interpreter to cope with all these issues.

The *cooperation strategy* of an agent *Ag* includes the set of agents with which it is expected to cooperate, the plan retrieval policy, and the plan acquisition policy. The cooperation strategy may evolve during time, allowing greater flexibility and autonomy to the agents, and is modelled by three functions:

– `trusted`(*Te*,*TrustedAgentSet*), where *Te* is a (not necessarily ground) triggering event and *TrustedAgentSet* is the set of agents that *Ag* will contact in order to obtain plans relevant for *Te*.
– `retrievalPolicy`(*Te*,*Retrieval*), where *Retrieval* may assume the values `always` and `noLocal`, meaning that relevant plans for the trigger *Te* must be retrieved from other agents in any case, or only when no local relevant plans are available, respectively.
– `acquisitionPolicy`(*Te*,*Acquisition*), where *Acquisition* may assume the values `discard`, `add` and `replace` meaning that, when a relevant plan for *Te* is retrieved from a trusted agent, it must be used and discarded, or added to the plan library, or used to update the plan library by replacing all the plans triggered by *Te*.

Plans. Besides the standard components which constitute BDI plans, in this extension plans also have a *source* which determines the first owner of the plan, and an *access specifier* which determines the set of agents with which the plan can be shared. The source may assume two values: `self` (the agent possesses the plan) and *Ag* (the agent was originally from *Ag*). The access specifier may assume three values: `private` (the plan cannot be shared), `public` (the plan can be shared with any agent) and `only`*(TrustedAgentSet)* (the plan can be shared only with the agents contained in *TrustedAgentSet*).

The Coo-AgentSpeak mechanism to be available in ***Jason*** soon will allow users to define cooperation strategies in the Coo-BDI style, and takes care of all other issues such as sending the appropriate requests for plans, suspending intentions that are waiting for plans to be retrieved from other agents, etc. The Coo-AgentSpeak mechanism is described in detail in [1].

One final characteristic of ***Jason*** that is relevant here is the configuration option on what to do in case there is no applicable plan for a relevant event. If an event is relevant, it means that there are plans in the agent's plan library for handling that particular event (representing that handling that event is normally a desire of that agent). If it happens that none of those plans are applicable at a certain time, this can be a problem as the agent does not know how to handle the situation at that time. Ancona and Mascardi [1] discussed how this problem is handled in various agent-oriented programming languages. In ***Jason***, a configuration option is given to users, which can be set in the file where the various agents and the environment composing a multi-agent system are specified. The option allows the user to state, for events which have relevant but not applicable plans, whether the interpreter should discard that event altogether (`events=discard`) or insert the event back at the end of the event queue (`events=requeue`). Because of ***Jason***'s customisation mechanisms, the only modification that were required for ***Jason*** to cope with Coo-AgentSpeak

```
mas         → "MAS" <ID> "{"
                 [ "infrastructure" ":" <ID> ]
                 [ environment ]
                 agents
              "}"
environment → "environment" ":" <ID> [ "at" <ID> ]
agents      → "agents" ":" ( agent ";" )+
agent       → <ASID>
              [ filename ]
              [ options ]
              [ "agentArchClass" <ID> ]
              [ "agentClass" <ID> ]
              [ "#" <NUMBER> ]
              [ "at" <ID> ]
filename    → [ <PATH> ] <ID>
options     → "[" option ( "," option )* "]"
option      → <ID> "=" ( <ID> | <NUMBER> | <STRING> )
```

Fig. 3. EBNF of the Language for Configuring Multi-Agent Systems.

was a third configuration option that is available to the users — no changes to the interpreter itself was required. When Coo-AgentSpeak is to be used, the option `events=retrieve` must be used in the configuration file. This makes ***Jason*** call the user-defined `selectOption` function *even when no applicable plans exist for an event*. This way, part of the Coo-BDI approach can be implemented by providing a special `selectOption` function which takes care of retrieving plans externally, whenever appropriate.

5 Main Features of the *Jason* Platform

Configuring Multi-Agent Systems

The configuration of a complete multi-agent system is given by a very simple text file. The EBNF grammar in Figure 3 gives the syntax that can be used in the configuration file. In this grammar, `<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 as usual in ordinary operating systems.

The `<ID>` used after the keyword `MAS` is the name of the society. The keyword `infrastructure` is used to specify which of the two infrastructures available in ***Jason***'s distribution will be used. The options currently available are either "`Centralised`" or "`Saci`"; the latter option allows agents to run on different machines over a network. It is important to note that the user's environment and customisation classes remain the same with both infrastructures.

Next an `environment` needs to be referenced. This is simply the name of Java class that was used for programming the environment. Note that an optional host name where the environment will run can be specified. This only works if the SACI option is used for the underlying system infrastructure.

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 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 in ***Jason*** (these are explained 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(s) will run (preceded by "`at`"). As before, this only works if the SACI-based infrastructure was chosen.

The user can change the initial settings of the AgentSpeak interpreter available in ***Jason***, or pass on settings to the agent classes by enclosing in square brackets certain configuration statements. These have the form of a keyword, followed by '`=`' and then the value (possibly predefined keywords) attributed to them; see [8] for further details. Finally, user-defined overall agent architecture and other user-defined functions to be used by the AgentSpeak interpreter for each particular agent can be specified with the keywords `agentArchClass` and `agentClass`.

Creating Environments

Jason agents can be situated in real or simulated environments. In the former case, the user would have to customise the "overall agent architecture", as described in the next part of this section; in the latter case, the user must provide an implementation of the simulated environment. This is done directly in a Java class that extends the ***Jason*** base Environment class. A general example of an environment class is shown in Figure 4.

All percepts (i.e., everything that is perceptible in the environment) should be determined using the addPercept method; the argument is a literal, so strong negation can be used in applications where there is open-world assumption. It is possible to send individualised perception; that is, in programming the environment the developer can determine what subset of the environment properties will be perceptible to individual agents. Recall that within an 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

```
public class myEnv extends Environment {

   public myEnv() {
      // environment initialisations
   }

   public String getPos(String ag) {
      // some code that returns the agent position
   }

   public boolean executeAction(String ag, Term action) {
      if (action.equals(...)) {
         // execute the action
      }
      ...
      removePercept(ag); // remove all percepts of agent ag
      addPercept(ag,Literal.parseLiteral("pos(r1," + getPos(ag) + ")"));
      addPercept(p); // add p as a percept to all agents
      return true;
   }
}
```

Fig. 4. Example of an Environment Class.

office's walls, 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, because it failed to perceive it). Determination of an agent's individual perception within the environment is done by using the "`addPercept(agentName, percept)`" method, where `agentName` is a string and `percept` is a literal.

Most of the code for building environments should be (referenced) in 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 (which has the form of a Prolog structure) representing the action (and any 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 executed or not. A plan fails if any basic action attempted by the agent fails.

Customising Agents

Certain aspects of the cognitive functioning of an agent can be customised by the user overriding methods of the Agent class (see Figure 5). The three first selection functions are discussed extensively in the AgentSpeak literature (see Section 3 and Figure 2). The social acceptance function (`socAcc`, which is related to pragmatics, e.g., trust and power social relations) and the message selection function are discussed in [31] and Section 4. 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. While basic actions are being executed by the environment, before the (boolean) feedback from the environment is available the intention to which that action belongs must be suspended; the last internal function allows customisation of priorities to be given when more than one intention can be resumed because feedback from the environment became available during the last reasoning cycle.

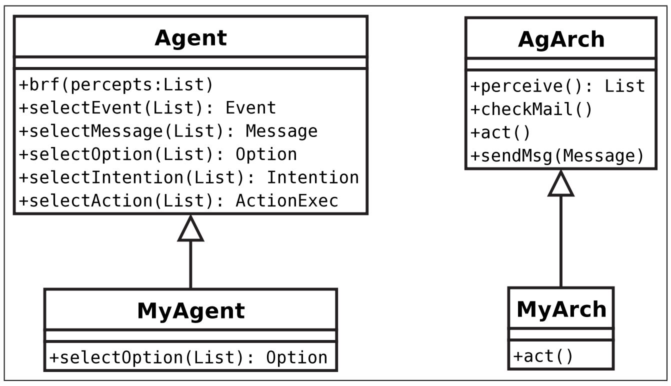

Agent
+brf(percepts:List)
+selectEvent(List): Event
+selectMessage(List): Message
+selectOption(List): Option
+selectIntention(List): Intention
+selectAction(List): ActionExec
MyAgent
+selectOption(List): Option
AgArch
+perceive(): List
+checkMail()
+act()
+sendMsg(Message)
MyArch
+act()

Fig. 5. Agent Customisation.

Similarly, the user can customise the functions defining the overall agent architecture (see Figure 5, AgArch class). These functions handle: (i) the way the agent will perceive the environment; (ii) the way it will update its belief base given the current perception of the environment, i.e., the so called belief revision function (BRF) in the AgentSpeak literature; (iii) how the agent gets messages sent from other agents (for speech-act based inter-agent communication); and (iv) how the agent acts on the environment (for the basic 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, but clearly for multi-agent systems situated in a real-world environment this might be more complicated, having to interface with, e.g., available process control hardware.

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, then process further the list of percepts, in order to simulate faulty perception, for example. This is on top of the environment being 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 [12]).

It is important to emphasise that the belief revision 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. In particular, it does not guarantee belief consistency. As 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 dropped), 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 might be interested in modelling a "paraconsistent" agent, which can be done easily.

An important construct for allowing AgentSpeak agents to remain at the right level of abstraction is that of internal actions, which allows for straightforward extensibility and use of legacy code. As suggested in [3], internal actions that start with '`.`' are part of a standard library of internal actions that are distributed with ***Jason***. Internal actions defined by users should be organised in specific libraries, which provides an interesting way of organising such code, which is normally useful for a range of different systems. In the AgentSpeak program, 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, the name of the package and class are the names of the library and action as it will be used in the AgentSpeak programs.

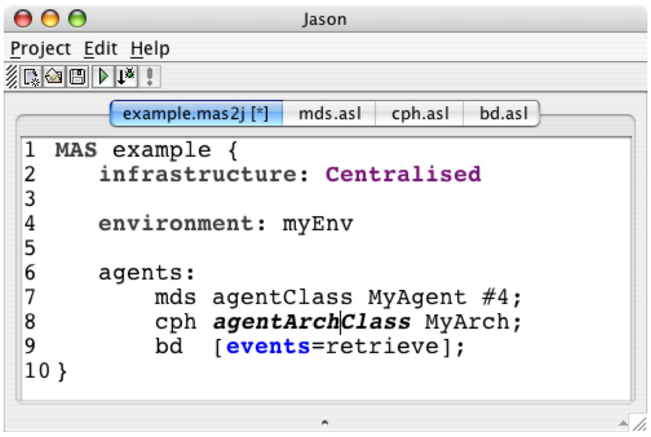
Available Tools and Documentation

Jason is distributed with an Integrated Development Environment (IDE) which provides a GUI for editing a MAS configuration file as well as AgentSpeak code for the individual agents. Figure 6 shows a screenshot of the ***Jason*** IDE, when the user is editing the multi-agent systems configuration file; the AgentSpeak code of each agent can also be edited (with syntax highlight) from the GUI.

Through the IDE, it is also possible to control the execution of a MAS, and to distribute agents over a network in a very simple way. There are three execution modes:

Asynchronous: in which all agents run asynchronously. An agent goes to its next reasoning cycle as soon as it has finished its current cycle. This is the default execution mode.

Synchronous: in which each agent performs a single reasoning cycle in every "global execution step". That is, when an agent finishes a reasoning cycle, it



```
MAS example {
    infrastructure: Centralised

    environment: myEnv

    agents:
        mds agentClass MyAgent #4;
        cph agentArchClass MyArch;
        bd  [events=retrieve];
}
```

Fig. 6. ***Jason*** IDE.

informs ***Jason***'s execution controller, and waits for a "carry on" signal. The ***Jason*** controller waits until all agents have finished their current reasoning cycle and then sends the "carry on" signal to them.

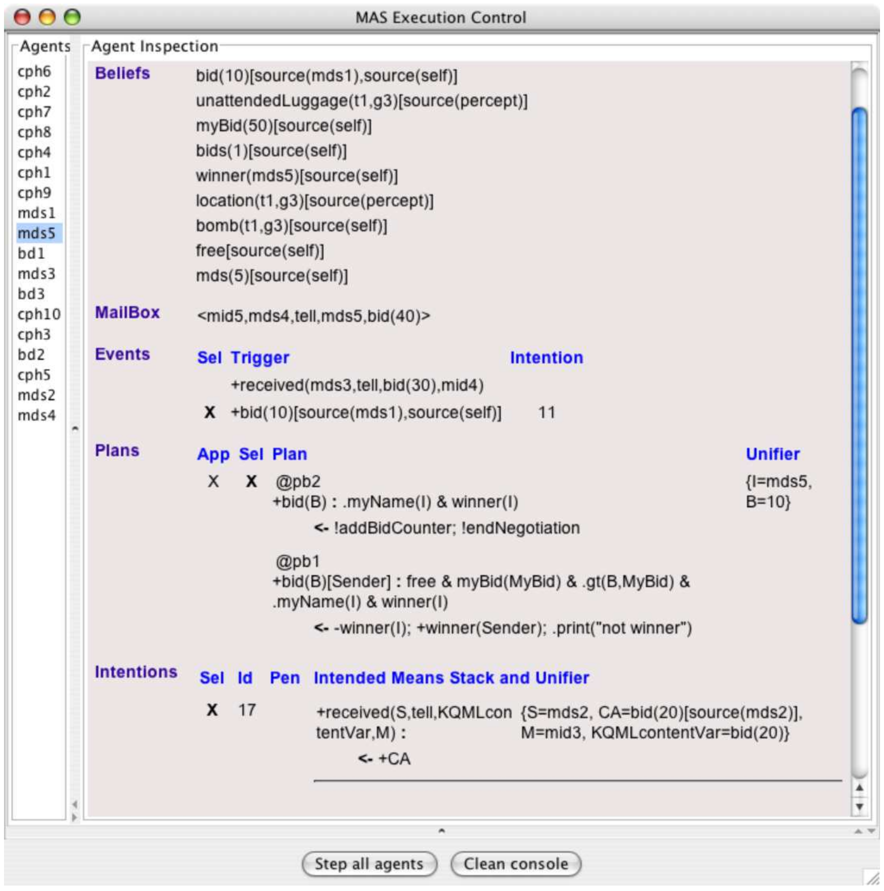
Debugging: this execution mode is similar to the synchronous mode; however, the ***Jason*** controller also waits until the user clicks on a "Step" button in the GUI before sending the "carry on" signal to the agents.

There is another tool provided as part of the IDE which allows the user to inspect agents' internal states when the system is running in debugging mode. This is very useful for debugging MAS, as it allows "inspection of agents' minds" across a distributed system. The tool is called "mind inspector", and is shown in Figure 7.

Jason's distribution comes with documentation which is also available online at `http://jason.sourceforge.net/Jason.pdf`. The documentation has something of the form of a tutorial on AgentSpeak, followed by a description of the features and usage of the platform. Although it covers all of the currently available features of ***Jason***, we still plan to improve substantially the documentation, in particular because the language is at times still quite academic. Another planned improvement in the available documentation, in the relatively short term, is to include material (such as slides and practical exercises) for teaching Agent-Oriented Programming with ***Jason***.

6 Discussion

One of the reasons for the growing success of agent-based technology is that it has been shown to be quite useful for the development of various types of applications, including air-traffic control, autonomous spacecraft control, health care, and industrial systems control, to name just a few. Clearly, these are application



```
cph6   Beliefs     bid(10)[source(mds1),source(self)]
cph2               unattendedLuggage(t1,g3)[source(percept)]
cph7               myBid(50)[source(self)]
cph8               bids(1)[source(self)]
cph4               winner(mds5)[source(self)]
cph1               location(t1,g3)[source(percept)]
cph9               bomb(t1,g3)[source(self)]
mds1               free[source(self)]
mds5               mds(5)[source(self)]
bd1
mds3   MailBox     <mid5,mds4,tell,mds5,bid(40)>
bd3
cph10  Events      Sel Trigger                                Intention
cph3                   +received(mds3,tell,bid(30),mid4)
bd2                X   +bid(10)[source(mds1),source(self)]     11
cph5
mds2   Plans       App Sel Plan                                              Unifier
mds4                X   X  @pb2                                              {I=mds5,
                           +bid(B) : .myName(I) & winner(I)                  B=10}
                                <- !addBidCounter; !endNegotiation

                           @pb1
                           +bid(B)[Sender] : free & myBid(MyBid) & .gt(B,MyBid) &
                           .myName(I) & winner(I)
                                <- -winner(I); +winner(Sender); .print("not winner")

       Intentions  Sel Id  Pen  Intended Means Stack and Unifier
                    X  17       +received(S,tell,KQMLcon  {S=mds2, CA=bid(20)[source(mds2)],
                                tentVar,M) :              M=mid3, KQMLcontentVar=bid(20)}
                                     <- +CA
```

Fig. 7. ***Jason***'s Mind Inspector.

areas for which *dependable systems* are in demand. Consequently, formal verification techniques tailored specifically for multi-agent systems is also an area that is attracting much research attention and is likely to have a major impact in the uptake of agent technology. One of the advantages of the approach to programming multi-agent systems resulting from the research reviewed in this paper is precisely the fact that it is amenable to formal verification. In particular, model checking techniques (and state-space reduction techniques to be used in combination with model checking) for AgentSpeak have been developed [6, 7, 5, 13].

Although very little has been considered so far in regards to using agent-oriented software engineering methodologies for the development of designs for systems to be implemented in ***Jason***, existing methodologies that specifically concern BDI agents, such as Prometheus [23], should be perfectly suitable for that purpose. In that book, the authors show an example of the use of JACK (see [32]) for the implementation, but they explicitly say that any platform that provides the basic concepts of reactive planning systems (such as goals and plans) would be most useful in the sense of providing all the required constructs to support the implementation of designs developed in accordance to the Prometheus methodology. Because AgentSpeak code is considerably more readable than other languages such as JACK and Jadex (see [25]), it is arguable that ***Jason*** will provide at least a much more clear way of implementing such designs. However, being an industrial platform, JACK has, currently, far better supporting tools and documentation, but on the other hand, ***Jason*** is *open source*, whereas JACK is not.

A construct that has an important impact in maintaining the right level of abstraction in AgentSpeak code even for sophisticated systems is that of internal actions (described earlier in Section 2). Internal actions necessarily have a boolean value returned, so they are declaratively represented within a logic program in AgentSpeak — in effect, we can keep the agent program as a high-level representation of the agent's reasoning, yet allowing it to be arbitrarily sophisticated by the use of existing software implemented in Java, or indeed any programming language through the use of JNI. Thus, the way in which integration with traditional object-oriented programming and use of legacy code is accomplished in ***Jason*** is far more elegant than with other agent programming languages (again, such as JACK and Jadex).

As ***Jason*** is implemented in Java, there is no issue with portability, but very little consideration has been given so far to standards compliance and interoperability. However, components of the platform can be easily changed by the user. For example, at the moment there are two infrastructures available in ***Jason***'s distribution: a centralised one (which means that the whole system runs in a single machine) and another which uses SACI for distribution. It should be reasonably simple to produce another infrastructure which uses, e.g., JADE (see [2]) for FIPA-compliant distribution and management of agents in a multi-agent system.

As yet, ***Jason*** has been used only for a couple of application described below, and also for simple student projects in academia. However, due to its AgentSpeak basis, it is clearly suited to a large range of applications for which it is known that BDI systems are appropriate; various applications of PRS [16] and dMARS [17] for example have appeared in the literature [34, Chapter 11].

Although we aim to use it for a wide range of applications in the future, in particular Semantic Web and Grid-based applications, one particular area of application in which we have great interest is Social Simulation [15]. In fact, ***Jason*** is being used as part of a large project to produce a platform tailored particularly to Social Simulation. The platform is called MAS-SOC and is described in [12]; it includes a high-level language called ELMS [22] for defining multi-agent environments. This approach was used to develop a simple social simulation on social aspects of urban growth. Another area of application that has been initially explored is the use of AgentSpeak for defining the behaviour of animated characters for computer animation (or virtual reality) [30].

7 Final Remarks

Jason is being actively improved and extended. The long term objective is to have a platform which makes available important technologies resulting from research in the area of Multi-Agent Systems, but doing this in a sensible way so as to avoid the language becoming cumbersome and, most importantly, having formal semantics for most, if not all, of the essential features available in ***Jason***. There are ongoing projects to extend ***Jason*** with organisations, given that social structure is an essential aspect of developing complex multi-agent systems, and with ontological descriptions underlying the belief base, thus facilitating the use of ***Jason*** for Semantic Web and Grid-based applications. We aim to contribute, for example, to the area of e-Social Science, developing large-scale Grid-based social simulations using ***Jason***.

Acknowledgments

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, Viviana Mascardi, Victor Lesser, Rodrigo Machado, Joyce Martins, Álvaro F. Moreira, Fabio Y. Okuyama, Denise de Oliveira, Carmen Pardavila, Marios Richards, Maíra R. Rodrigues, Rosa M. Vicari, Willem Visser, Michael Wooldridge.

Rafael Bordini gratefully acknowledges the support of The Nuffield Foundation (grant number NAL/01065/G).

References

1. D. Ancona, V. Mascardi, J. F. Hübner, and R. H. Bordini. Coo-AgentSpeak: Cooperation in AgentSpeak through plan exchange. In N. R. Jennings, C. Sierra, L. Sonenberg, and M. 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. F. Bellifemine, F. Bergenti, G. Caire, and A. Poggi. JADE — a java agent development framework. In Bordini et al. [4], chapter 5, pages 125–147.
3. R. H. Bordini, A. L. C. Bazzan, R. O. Jannone, D. M. Basso, R. M. Vicari, and V. R. Lesser. AgentSpeak(XL): Efficient intention selection in BDI agents via decision-theoretic task scheduling. In C. Castelfranchi and W. L. 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. R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors. *Multi-Agent Programming: Languages, Platforms and Applications.* Number 15 in Multi-agent Systems, Artificial Societies, and Simulated Organizations. Springer-Verlag, 2005.

5. R. H. Bordini, M. Fisher, C. Pardavila, and M. Wooldridge. Model checking AgentSpeak. In J. S. Rosenschein, T. Sandholm, M. Wooldridge, and M. 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. R. H. Bordini, M. Fisher, W. Visser, and M. Wooldridge. Model checking rational agents. *IEEE Intelligent Systems*, 19(5):46–52, September/October 2004.
7. R. H. Bordini, M. Fisher, W. Visser, and M. Wooldridge. State-space reduction techniques in agent verification. In N. R. Jennings, C. Sierra, L. Sonenberg, and M. 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 896–903, New York, NY, 2004. ACM Press.
8. R. H. Bordini, J. F. Hübner, et al. ***Jason**: A Java-based agentSpeak interpreter used with saci for multi-agent distribution over the net*, manual, version 0.6 edition, Feb 2005. `http://jason.sourceforge.net/`.
9. R. H. Bordini, J. F. Hübner, and R. Vieira. ***Jason*** and the golden fleece of agent-oriented programming. In Bordini et al. [4], chapter 1, pages 3–37.
10. R. H. Bordini and Á. F. Moreira. Proving the asymmetry thesis principles for a BDI agent-oriented programming language. In J. Dix, J. A. Leite, and K. Satoh, editors, *Proceedings of the Third International Workshop on Computational Logic in Multi-Agent Systems (CLIMA-02), 1st August, Copenhagen, Denmark, held as part of FLoC-02*, Electronic Notes in Theoretical Computer Science 70(5). Elsevier, 2002. URL: <http://www.elsevier.nl/locate/entcs/volume70.html>.
11. R. H. Bordini and Á. 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, Sept. 2004. Special Issue on Computational Logic in Multi-Agent Systems.
12. R. H. Bordini, F. Y. Okuyama, D. de Oliveira, G. Drehmer, and R. C. Krafta. The MAS-SOC approach to multi-agent based simulation. In G. Lindemann, D. Moldt, and M. 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.
13. R. H. Bordini, W. Visser, M. Fisher, C. Pardavila, and M. Wooldridge. Model checking multi-agent programs with CASP. In W. A. Hunt Jr. and F. 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.
14. C. Castelfranchi and R. Falcone. Principles of trust for MAS: Cognitive anatomy, social importance, and quantification. In Y. Demazeau, editor, *Proceedings of the Third International Conference on Multi-Agent Systems (ICMAS'98), Agents' World, 4–7 July, Paris*, pages 72–79, Washington, 1998. IEEE Computer Society Press.
15. J. Doran and N. Gilbert. Simulating societies: An introduction. In N. Gilbert and J. Doran, editors, *Simulating Society: The Computer Simulation of Social Phenomena*, chapter 1, pages 1–18. UCL Press, London, 1994.
16. M. 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.

17. D. Kinny. The distributed multi-agent reasoning system architecture and language specification. Technical report, Australian Artificial Intelligence Institute, Melbourne, Australia, 1993.
18. J. A. 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.
19. R. Machado and R. H. Bordini. Running AgentSpeak(L) agents on SIM_AGENT. In J.-J. Meyer and M. 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.
20. Á. F. Moreira and R. H. Bordini. An operational semantics for a BDI agent-oriented programming language. In J.-J. C. Meyer and M. 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.
21. Á. F. Moreira, R. Vieira, and R. H. Bordini. Extending the operational semantics of a BDI agent-oriented programming language for introducing speech-act based communication. In J. Leite, A. Omicini, L. Sterling, and P. 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. F. Y. Okuyama, R. H. Bordini, and A. C. da Rocha Costa. ELMS: an environment description language for multi-agent simulations. In D. Weyns, H. van Dyke Parunak, F. Michel, T. Holvoet, and J. Ferber, editors, *Environments for Multiagent Systems, State-of-the-art and Research Challenges. Proceedings of the First International Workshop on Environments for Multiagent Systems (E4MAS), held with AAMAS-04, 19th of July*, number 3374 in Lecture Notes in Artificial Intelligence, pages 91–108, Berlin, 2005. Springer-Verlag.
23. L. Padgham and M. Winikoff, editors. *Developing Intelligent Agent Systems: A Practical Guide*. John Wiley and Sons, 2004.
24. G. D. Plotkin. A structural approach to operational semantics. Technical report, Computer Science Department, Aarhus University, Aarhus, 1981.
25. A. Pokahr, L. Braubach, and W. Lamersdorf. Jadex: A BDI reasoning engine. In Bordini et al. [4], chapter 6, pages 149–174.
26. A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In W. Van de Velde and J. 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.
27. A. S. Rao and M. P. Georgeff. BDI agents: From theory to practice. In V. Lesser and L. Gasser, editors, *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS'95), 12–14 June, San Francisco, CA*, pages 312–319, Menlo Park, CA, 1995. AAAI Press / MIT Press.
28. A. S. Rao and M. P. Georgeff. Decision procedures for BDI logics. *Journal of Logic and Computation*, 8(3):293–343, 1998.

29. M. P. Singh, A. S. Rao, and M. P. Georgeff. Formal methods in DAI: Logic-based representation and reasoning. In G. Weiß, editor, *Multiagent Systems—A Modern Approach to Distributed Artificial Intelligence*, chapter 8, pages 331–376. MIT Press, Cambridge, MA, 1999.
30. J. A. Torres, L. P. Nedel, and R. H. Bordini. Autonomous agents with multiple foci of attention in virtual environments. In *Proceedings of 17th International Conference on Computer Animation and Social Agents (CASA 2004), Geneva, Switzerland, 7–9 July*, pages 189–196, 2004.
31. R. Vieira, A. Moreira, M. Wooldridge, and R. H. Bordini. On the formal semantics of speech-act based communication in an agent-oriented programming language. *Submitted article, to appear*, 2005.
32. M. Winikoff. JACK$^{\text{TM}}$ intelligent agents: An industrial strength platform. In Bordini et al. [4], chapter 7, pages 175–193.
33. M. Wooldridge. *Reasoning about Rational Agents.* The MIT Press, Cambridge, MA, 2000.
34. M. Wooldridge. *An Introduction to MultiAgent Systems.* John Wiley & Sons, 2002.