

A Recent Experience in Teaching Multi-Agent Systems Using *Jason*

Rafael H. Bordini
Department of Computer Science
University of Durham
Durham DH1 3LE, U.K.
R.Bordini@durham.ac.uk

ABSTRACT

This paper briefly describes a recent experience of using *Jason* for teaching Multi-Agent Systems. *Jason* is a Java-based interpreter for an extended version of AgentSpeak. The basic AgentSpeak programming language for BDI agents is rather simple, making *Jason* a suitable tool for students to put into practice some of the elements of the BDI theory covered in the lectures.

Keywords

Multi-Agent Systems, *Jason*, Undergraduate Teaching

1. INTRODUCTION

Since October 2004, a half-module on Multi-Agent Systems is being taught here at the University of Durham. We use [21] as textbook, but other books (e.g., [18]) are also recommended. The module extends throughout the academic year, with one one-hour lecture per week. There are 19 main lectures plus revision lectures. Currently, there are 25 students attending the course, which is being taught for the first time here. It is a Level 3 course (third, and final, year of study for a first degree in Computer Science in the U.K.), and the students attend an “Introduction to Artificial Intelligence” course in the second year. One of the learning outcomes of the module is for students to acquire some understanding of the agent-oriented programming paradigm.

The experience has been quite interesting in that students seem very interested in the techniques that they are being taught. The lectures are at times quite heavy on theory, but two pieces of coursework are aimed at encouraging students to make practical use of what they have learnt. The first coursework is to develop an agent for the Trading Agent Competition [19] (<http://www.sics.se/tac/>). Students appear to have thoroughly enjoyed themselves in doing this.

The second coursework integrates theory and practice much further, at least in principle. A significant part of the course is spent on BDI agents. The second coursework (worth double the marks of the first one) aims at the development of a (simple) multi-agent system with BDI agents using *Jason* [4] (the students decide what application they will develop, and choosing a suitable application is part of the marking criteria). The system can be designed using the Prometheus methodology [14]; however, as there is only one lecture on Prometheus, this is not compulsory. We believe that this coursework is very important, given the learning outcome of providing students with an understanding of agent-oriented programming; with no hands-on experience of agent-oriented programming, such outcome could hardly be expected to be effectively achieved.

The first students in Durham to attend this module are still working on this coursework, so the demonstrations of the final systems have not yet taken place by the time of writing the camera-ready version of this paper, as it will appear in the TeachMAS proceedings. Therefore, it is not yet possible to determine how successful this first experience will be.

The remainder of this paper gives an overview of *Jason*, the platform we use for the main coursework. *Jason* can be useful for others who teach the BDI architecture in their multi-agent systems courses, hence such overview is included here. A much more detailed overview of *Jason* will be published soon [5]. Below we give some of the pointers to existing literature on *Jason* and AgentSpeak (the language on which it is based).

2. BACKGROUND

2.1 About *Jason*

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*.

Jason is distributed *Open Source* and is kindly hosted by [Sourceforge.net](https://sourceforge.net) at:

[jason.sourceforge.net](https://sourceforge.net)

2.2 Some References

The idea of Agent-Oriented Programming was first discussed by Yoav Shoham [16]. Although agent-oriented programming is still incipient, the whole area of multi-agent systems [21] 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 [15], and subsequently extended and discussed

in a series of papers by Bordini and colleagues (e.g., [7, 1, 6, 12, 13, 2, 9, 3, 8]). AgentSpeak has a neat notation and is a thoughtful (and computationally efficient) extension of logic programming to BDI agents [20, 17]. *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 [10] (see <http://www.lti.pcs.usp.br/saci/>).

The reason why there is still little use of AgentSpeak in the development of practical application is certainly related to the fact that, before the 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. AGENT AND MULTI-AGENT SYSTEM LANGUAGES

3.1 Syntax of *Jason*'s AgentSpeak

The BNF grammar in Figure 1 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.

The main differences to the original AgentSpeak(L) language are as follows. Wherever a predicate¹ was allowed in the original language, here a literal is used instead. This is either a predicate $p(t_1, \dots, t_n)$, $n \geq 0$, or $\sim p(t_1, \dots, t_n)$, where ‘ \sim ’ denotes strong negation. 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 in the introductory chapters of [11]. Terms now can be atoms, structures, variables, and lists as in Prolog, as well as integer or floating point numbers, and strings (enclosed in double quotes).

Also, a major change is that predicates (i.e., atomic formulæ) now can have “annotations”. This is a list of terms enclosed in square brackets immediately following the predicate. Within the belief base, annotations are used to register the sources of information. Two special atoms, **percept** and **self**, 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 [**self**] annotation), unless the predicate 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

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

<u>agent</u>	→ <u>beliefs</u> <u>plans</u>
<u>beliefs</u>	→ (<u>literal</u> ".")*
	<i>N.B.: a semantic error is generated if the literal was not ground.</i>
<u>plans</u>	→ (<u>plan</u>)+
<u>plan</u>	→ [<u>atomic_formula</u> "->"]
	<u>triggering_event</u> ":" <u>context</u> "<->" <u>body</u> "."
<u>triggering_event</u>	→ "+" <u>literal</u>
	"-" <u>literal</u>
	"+" "!" <u>literal</u>
	"-" "!" <u>literal</u>
	"+" "?" <u>literal</u>
	"-" "?" <u>literal</u>
<u>literal</u>	→ "~" <u>atomic_formula</u>
	"~" "(" <u>atomic_formula</u> ")"
	<u>atomic_formula</u>
<u>default_literal</u>	→ "not" <u>literal</u>
	"not" "(" <u>literal</u> ")"
	<u>literal</u>
<u>context</u>	→ "true"
	<u>default_literal</u> ("&" <u>default_literal</u>)*
<u>body</u>	→ "true"
	<u>body_formula</u> (";" <u>body_formula</u>)*
<u>body_formula</u>	→ <u>literal</u>
	"!" <u>literal</u>
	"?" <u>literal</u>
	"+" <u>literal</u>
	"-" <u>literal</u>
<u>atomic_formula</u>	→ <ATOM> ["(" <u>list_of_terms</u> ")"]
	["[" <u>list_of_annotations</u> "]"]
<u>structure</u>	→ <ATOM> "(" <u>list_of_terms</u> ")"
<u>list_of_terms</u>	→ <u>term</u> ("," <u>term</u>)*
<u>list_of_annotations</u>	→ <i>as list_of_terms, but generating a semantic error if not ground;</i>
<u>list</u>	→ "["
	[<u>term</u> (("," <u>term</u>)*
	" " (<u>list</u> <VAR>)
)
] "]"
<u>term</u>	→ <u>structure</u>
	<u>list</u>
	<ATOM>
	<VAR>
	<NUMBER>
	<STRING>

Figure 1: BFN of the Language Interpreted by *Jason*

been communicated by another agent). For more on the annotation of sources of information for beliefs, see [13].

Plans also have labels, as first proposed in [1]. However, a plan label can now be any predicate, including annotations, although we suggest that plan labels are predicates with annotations (if necessary) but arity 0, e.g. `aLabel` or `aLabel[p(0.9)]`. Annotations in predicates 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 a 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 4.1. Also, as the label is part of an instance of a plan in the set of intentions, and the an-

notations can be changed dynamically, this provides all the means necessary for the implementation of efficient intention selection functions, as the one proposed in [1]. 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, as well as negated percepts in the environment (see Section 4.4).

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. This can be used for programmers to determine that an agent should, for example, persist in attempting to achieve a goal if a plan for achieving it has failed. It is also simple to specify a plan which, under specific conditions, chooses to drop the intention altogether (by means of a standard internal action).

Finally, as also introduced in [1], *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 defined by the user in Java. There are several standard internal actions that are distributed with *Jason*, but we do not mention them here.

3.2 Defining and Running Multi-Agent Systems

The BNF grammar in Figure 2 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> "{"
              [ "architecture" ":" <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      → "events" "=" ( "discard" | "requeue" )
              | "intBels" "=" ( "sameFocus" | "newFocus" )
              | "verbose" "=" <NUMBER>

```

Figure 2: BNF of the Multi-Agent System Definition Language

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 `architecture` is used to specify which of the two overall agent architecture available with *Jason*'s

distribution will be used. The options currently available are either “`Centralised`” or “`Saci`”; the latter is the default option and allows agents to run on different machines over a network. This is necessary to determine whether the Java classes defining the overall agent architecture and environment perception used by *Jason* when creating agent instances and the environment will be `CentralisedAgArch` and `CentralisedEnvironment` (for option `Centralised`, or `SaciAgArch` and `SaciEnvironment` (for option `Saci`), which use SACI for running agents and managing agent communication. Users may define other system architectures, although these two should cover for most general purposes.

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 is only available if you use SACI for the underlying architecture. Section 4.4 explains how environments can be easily created in Java.

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 architecture 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 or requeue; the former means that external events for which there are no applicable plans are discarded (a warning is printed out to the console where the SACI or the system was run), whereas the latter is used when such events should be inserted back at the end of the list of events that the agent needs to handle.

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.

verbose: a number between 0 and 6 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 console where SACI or the system was run. 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.

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` 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 the next section.

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. 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 functions*, 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 [7]). The customisation of these aspects of an agent is explained in Section 4.1.

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 belief revision (i.e., updating the agent’s belief base from perception of the environment), 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 4.2.

4.1 Customising Agents

Unless customised agent classes in Java are provided by the user, default functions are used. The options for customisation that users are given by *Jason* are the three selection

functions that were originally defined with AgentSpeak, plus functions for selecting the next message to be processed, and the next action to be executed on the environment among those already chosen to be executed. The user can also define acceptance relations between sender of messages and their contents (representing such things as social power and trust).

Trust functions and the message selection function are discussed in [13], and the three standard selection functions are discussed in all of the AgentSpeak literature. 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 architecture 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 the feedback from the environment is received, that intention can be selected again for execution (by the intention selection function).

4.2 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; 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; 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 [8]).

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. It does NOT guarantee belief consistency. As it will be seen later in Section 4.4, 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 “para-consistent” agent², which can be easily done.

Users can create their own architectures (providing basic mechanisms for running agents, communication with the environment and between agents, etc.). In most cases, these four functions should be kept abstract so that customising at an individual agent basis is possible. Both the **Centralised** and the **Saci** overall agent architectures distributed with **Jason** define the functions mentioned below.

4.3 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 [1], internal actions that start with ‘.’ are part of a standard library of internal actions; these are distributed with **Jason**, in directory `src/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.

When **Jason** is installed, an (empty) subdirectory `ulib` is created in the main **Jason** directory. Users should use this directory to store their own libraries of internal actions they implement in Java. A library is simply a subdirectory inside `ulib`. Recall that all identifiers starting with an uppercase letter in AgentSpeak denote variables, so the name of the library (the directory in `ulib`) *must* start with a lowercase letter. Each action in the new library should be a Java file inside the created directory, the name of the file (and thus of the public class inside it) is the name of the action as it will be called in the AgentSpeak source, plus a `.java` extension. All classes defining internal actions should implement an `execute` method declared with a predefined way.

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

4.4 Creating Environments

Besides programming a set of AgentSpeak agents, and providing the multi-agent system configuration file, in order to have a complete computational system³, the user needs to create an environment. This is done directly in Java,

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

³AgentSpeak 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.

and a class providing an environment typically extends the **Environment** class provided with **Jason**.

The **EnvironmentPerception** interface always provides `getPercepts` and `getNegativePercepts`, regardless of the system architecture. Note that only instances of the class **Term**, which is part of the `json` package, in the form of a structure (i.e., having the same form of a predicate) should be added to the lists returned by these methods! Class **Term** is used instead of **Pred** as annotations are not allowed here, as all percepts will be received by the agents with a [percept] annotation. All positive percepts (what is true of the environment) should be added to the list returned by `getPercepts`, while the one returned by `getNegativePercepts` is only relevant if the application uses open world; in that case the latter should have all predicates that are to be perceived as explicitly false (strong negation) by the agents. It is normally appropriate to use the class constructor to initialise the lists of percepts, and use the `parse` method of the **Term** class, as in the environments available in the `examples` directory of **Jason**’s distribution.

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** (as before, a structure) which represents 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 successful or not. A plan fails if any basic action attempted by the agent fails.

5. FINAL REMARKS

It is not yet possible to tell how much of all the features described above students will be able to make appropriate use of, given that only very short lecturing time has been dedicated to **Jason**. Much progress in the area of Agent-Oriented Programming is expected, and **Jason** being under constant development, it should keep up with such developments. With this, rather sophisticated multi-agent systems could be constructed with it, even at undergraduate level. Plans for the future include making more extensive use of **Jason** (i.e., increasing the number of lecturing hours dedicated to agent-oriented programming), building upon this initial experience. It would be interesting to see the impact that this will have on student’s learning and interest in the area, and in integrating theory and practice in teaching multi-agent systems.

6. ACKNOWLEDGEMENTS

Jason has been developed in collaboration with Jomi F. Hübner and as the references indicate, it draws on work done in collaboration with many colleagues.

7. REFERENCES

- [1] 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, *Proc. of AAMAS-2002, 15–19 July, Bologna, Italy*, pages 1294–1302, ACM Press, 2002.
- [2] 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, *Proc. of AAMAS-2003, Melbourne, Australia, 14–18 July*, pages 409–416, ACM Press, 2003.
- [3] R. H. Bordini, M. Fisher, W. Visser, and M. Wooldridge. Verifiable multi-agent programs. In *Proc. of ProMAS-03 (held with AAMAS-03), 15 July, 2003, Melbourne, Australia*, number 3067 in LNCS, pages 72–89, Springer, 2003.
- [4] 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, release 0.6 edition, February 2008. <http://jason.sourceforge.net/>.
- [5] R. H. Bordini, J. F. Hübner, and R. Vieira. **Jason** and the Golden Fleece of agent-oriented programming. In R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors, *Multi-Agent Programming*, chapter 1. Springer, 2005. To appear in July 2005.
- [6] 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, *Proc. of CLIMA-02, 1st August, Copenhagen, Denmark*, Electronic Notes in Theoretical Computer Science 70(5). Elsevier, 2002.
- [7] 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.
- [8] 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, *Proc. of RASTA'02, 16 July, 2002, Bologna, Italy (held with AAMAS02)*, number 2934 in Lecture Notes in Artificial Intelligence, pages 70–91, Berlin, 2004. Springer-Verlag.
- [9] 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, *Proc. of CAV-2003, Boulder, CO, 8–12 July*, number 2725 in LNCS, pages 110–113, Springer-Verlag, 2003. Tool description.
- [10] J. F. Hübner. *Um Modelo de Reorganização de Sistemas Multiagentes*. PhD thesis, Universidade de São Paulo, Escola Politécnica, 2003.
- [11] 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.
- [12] R. Machado and R. H. Bordini. Running AgentSpeak(L) agents on SIM_AGENT. In J.-J. Meyer and M. Tambe, editors, *Intelligent Agents VIII – Proc. of ATAL-2001, August 1–3, 2001, Seattle, WA*, number 2333 in LNAI, pages 158–174, Springer-Verlag, 2002.
- [13] 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, Proc. of DALI-03 (held with AAMAS-03), 15 July, 2003, Melbourne, Australia*, number 2990 in LNAI, pages 135–154, Springer-Verlag, 2004.
- [14] L. Padgham and M. Winikoff. *Developing Intelligent Agent Systems: A Practical Guide*. John Wiley and Sons, 2004.
- [15] A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In W. Van de Velde and J. Perram, editors, *Proc. MAAMAW'96, 22–25 January, Eindhoven, The Netherlands*, number 1038 in LNAI, pages 42–55, Springer-Verlag, 1996.
- [16] Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60:51–92, 1993.
- [17] 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.
- [18] G. Weiß, editor. *Multiagent Systems—A Modern Approach to Distributed Artificial Intelligence*. MIT Press, Cambridge, MA, 1999.
- [19] M. P. Wellman and P. R. Wurman. A trading agent competition for the research community. In *IJCAI-99 Workshop on Agent-Mediated Electronic Commerce, Stockholm*, 1999.
- [20] M. Wooldridge. *Reasoning about Rational Agents*. The MIT Press, Cambridge, MA, 2000.
- [21] M. Wooldridge. *An Introduction to MultiAgent Systems*. John Wiley & Sons, 2002.