Chapter 1

JASON AND THE GOLDEN FLEECE OF AGENT-ORIENTED PROGRAMMING

Rafael H. Bordini,[1] Jomi F. Hübner,[2] and Renata Vieira[3]

[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

[3] *Programa Interdisciplinar de Pós-Graduação em Computação Aplicada,*
Universidade do Vale do Rio dos Sinos, São Leopoldo, RS 93022-000, Brazil
renata@exatas.unisinos.br

Abstract This chapter describes ***Jason***, an interpreter written in Java for an extended version of AgentSpeak, a logic-based agent-oriented programming language that is suitable for the implementation of reactive planning systems according to the BDI architecture. We describe both the language and the various features and tools available in the platform.

Keywords: Logic-Based Agent Programming, Beliefs-Desires-Intentions, Operational Semantics, Speech Acts, Plan Exchange, Java-based Extensibility/Customisation.

Now was remaining as the last conclusion of this game,
By force of chaunted herbes to make the watchfull Dragon sleepe
Within whose eyes came never winke: who had in charge to keepe
The goodly tree upon the which the golden fleeces hung.
...
The dreadfull Dragon by and by (whose eyes before that day
Wist never erst what sleeping ment) did fall so fast asleepe
That Jason safely tooke the fleece of golde that he did keepe.

P. Ovidius Naso, Metamorphoses (ed. Arthur Golding), Book VII.

1.1 Motivation

Research on Multi-Agent Systems (MAS) has led to a variety of techniques that promise to allow the development of complex distributed systems. The importance of this is that such systems would be able to work in environments that are traditionally thought to be too unpredictable for computer programs to handle. With more than a decade of work on Agent-Oriented Programming (AOP) — since the seminal paper by Y. Shoham [206] — it has become clear that the task of putting together the technology emerging from MAS research in a way that allow the practical development of real-world MAS is comparable, in mythological terms, to the task of retrieving the Golden Fleece from the distant kingdom of Colchis, where it hang on a tree guarded by a sleepless dragon. Of course, this is not a task for ***Jason*** alone, but for the greatest heros of the time, who became known as the Argonauts (a selection of "whom" is described throughout this book).

The work described here is the result of an attempt to revive one of the most elegant programming languages that appeared in the literature; the language was called AgentSpeak(L), and was introduced by A. Rao in [180]. AgentSpeak(L) is a logic-based agent-oriented programming language, which is aimed at the implementation of reactive planning systems (such as PRS [98]) but also benefited from the experience with more clear notions of Beliefs-Desires-Intentions (BDI) as put forward in the work on the BDI agent architecture [181, 182] and BDI logics [183, 237]. However, AgentSpeak(L) was not but an abstract agent programming language. The work we have done, together with various colleagues, was both on extending AgentSpeak so that it became a practical programming language (in a way to allow full integration with what we consider the most important MAS techniques) and providing operational semantics (a standard formalism for semantics of programming languages) for AgentSpeak and most of the proposed extensions.[1] The driving force of all work reported here is to have a programming language for MAS which is practical (in the sense of allowing the development of real-world applications), yet elegant and with a rigorous formal basis.

Jason is the interpreter for our extended version of AgentSpeak(L), which allows agents to be distributed over the net through the use of SACI [115]. ***Jason*** is available *Open Source* under GNU LGPL at `http://jason.sourceforge.net` [22]. It implements the operational semantics of AgentSpeak(L) originally given in [24, 152] and improved in [229]. It also implements the extension of that operational semantics to account for speech-act based communication among AgentSpeak agents, first proposed

[1] We shall use AgentSpeak throughout this chapter, as a general reference to either AgentSpeak(L) as proposed by Rao or the various existing extensions.

in [153] and then extended in [229] (see Section 1.2.4). Another important extension is on allowing plan exchange [4] (see Section 1.2.4).

Some of the features available in ***Jason*** are:

- speech-act based inter-agent communication (and belief annotation on 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);
- 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";
- a clear notion of a *multi-agent environment*, which is implemented in Java (this can be a simulation of a real environment, e.g. for testing purposes before the system is actually deployed).

Interestingly, most of the advanced features are available as optional, customisable mechanisms. Thus, because the AgentSpeak(L) core that is interpreted by ***Jason*** is very simple and elegant, yet having all the main elements for expressing reactive planning system with BDI notions, we think that ***Jason*** is also ideal for teaching AOP for under- and post-graduate studies.

An important strand of work related to AgentSpeak that adds to making ***Jason*** a promising platform is the work on formal verification of MAS systems implemented in AgentSpeak by means of model checking techniques (this is discussed in Section 1.2.2); this work in fact draws on there being precise definitions of the BDI notions in terms of states of AgentSpeak agents, as mentioned above. Before we start describing ***Jason*** in more detail, we will introduce a scenario that will be used to give examples throughout this chapter. Although not all parts of the scenario are used in the examples given here, we introduce the whole scenario as we think it contains most of the important aspects of environments for which multi-agent systems are appropriate, and may therefore be useful more generally than its use in this chapter.

Scenario for a Running Example: The Airport Chronicle

The year is 2070 *ad*. Airports have changed a lot since the beginning of the century, but terrorist attacks are hardly a thing of the past. Anti-terror technology has improved substantially, arguably to compensate for the sheer

irrationality of mankind when it comes to resolve issues such as economic greed, religious fanaticism, and group favouritism, all of which remain with us from evolutionary times when they may have been useful.

Airports are now completely staffed by robots, specially London Heathrow, where different robot models are employed for various specific tasks. In particular, security is now completely under the control of specialised robots: due to a legacy from XX and early XXI century, Heathrow is still number one... terrorist threat target, that is. The majority of the staff, however, is formed by CPH903 robots. These are **c**ute, **p**olite, **h**andy robots who welcome people into the airport, give them a "hand" with pieces of luggage (e.g. lifting them to place on a trolley), and, of course, provide any information (in natural language, also using multi-media presentations whenever useful) that costumers may need.

Most of the security-related tasks are carried out by model MDS79 robots. The **m**ulti-**d**evice **s**ecurity robots are very expensive pieces of equipment, as they are endowed with all that technology can provide, in 2070, for bomb detection. They use advanced versions of the technology in use by the beginning of the century: x-ray, metal detectors, and computed tomography for detecting explosive devices, ion trap mobility spectrometry (ITMS) for detecting traces of explosives, as well as equipment for detecting radioactive materials (gamma ray and neutrons) used in "dirty bombs".

These days at Heathrow, check-in and security checks are no longer centralised, being carried out directly at the boarding gates. Thus, there are one or two replicas of robot model MDS79 at each departure gate. When unattended luggage is reported, all staff in the vicinity are informed of its location through a wireless local area network to which they all are connected. The robots then start a process of negotiation (with a very tight deadline for a final decision) in order to reach an agreement on which of them will be relocated to handle the unattended luggage report.

All staff robots know that, normally, one MDS79 and one CPH903 robot can cooperate to ensure that reported unattended luggage has been cleared away. The way they actually do it is as follow. The MDS79 robot replica uses all of its devices to check whether there is a bomb in the unattended luggage. If there is any chance of there being a bomb in the luggage, the MDS79 robot sends a high priority message to the bomb-disarming team of robots. (Obviously, robots communicate using speech-act based languages, such as those used for agent communication since the end of last century.) Only three of these very specialised robots are operational for all Heathrow terminals at the moment. Once these robots are called in, the MDS79 and CPH903 robots that had been relocated can go back to their normal duties. The bomb-disarming robots decide whether to set off a security alert to evacuate the airport, or alternatively they attempt to disarm the bomb or move it

to a safe area, if they can ensure such courses of action would pose no threat to the population.

In case the MDS79 robot detects no signs of a bomb in the unattended luggage, the job is passed on to the accompanying CPH903 robot. Luggage these days usually come with a magnetic ID tag that records the details of the passenger who owns it. Replicas of robot CPH903 are endowed with a tag reader and, remember, they are heavily built so as to be able to carry pieces of luggage (unlike MDS79). Besides, MDS79 are expensive and much in demand, so they should not be relocated to carry the piece of luggage after it has been cleared. So, in case the luggage is cleared, it is the CPH903 robot's task to take the unattended luggage to the gate where the passenger is (details of flights and passengers are accessed through the wireless network) if the passenger is known to be already there, or to the lost luggage centre, in case the precise location of the passenger in the airport cannot be determined (which is rather unusual these days).

Thus, all staff robots have, as part of their knowledge representation, that normally an MDS79 robot and a CPH903 robot can cooperate to eventually bring about a state of affairs where the unattended luggage has been cleared away. When unattended luggage is reported, they negotiate (for a very limited period of time, after which a quick overriding decision based simply on distance to the unattended luggage is used) so as to determine the best group of robots to be relocated to sort out the incident. Ideally, the MDS79 robot to be relocated will be currently at a gate where two MDS79 robots are available, to avoid excessive delays in boarding at that gate. Robots of type CPH903 are easy to relocate as they exist in large numbers and do not normally execute critical tasks.

An important aspect to consider is that the whole negotiation process, under normal circumstances, is about the specific MDS79 robot to be relocated, and the choice of one CPH903 robot to help out. However, other more difficult situations may arise under unpredicted circumstances. For example, on the 9th of May 2070, at Heathrow, an unattended piece of luggage was reported near gate 54. It turned out that the robot with ID S39 (an MDS79 replica) was helping out another MDS79 in charge of gate 56 close by. After briefly considering the situation, S39 volunteered to check out the reported unattended luggage, and so did H124 (a CPH903 replica). However, while running a self check, S39 realised that its internal ITMS equipment had just been damaged, which it reported to other robots involved in the negotiation.

In the light of that recent information, negotiation was resumed among the involved robots, to try and define an alternative course of action. Another MDS79 robot could have been relocated, which would have led to delays at one of the nearby gates (gate 52), as that MDS79 robot was alone taking care of security at that gate. Based on an argument put forward by S39,

the agreed course of action was that another (suitably positioned) CPH903 robot would be relocated to take (from a storage facility in that terminal) a handheld ITMS device, while S39 and H124 made their way to the location of the unattended luggage. Any of the three relocated robots can actually operate the portable ITMS device, so together they were able to bring about a state of affairs where the unattended luggage has been cleared away.

1.2 Language

The AgentSpeak(L) programming language was introduced in [180]. 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 [237].

An AgentSpeak(L) 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(L) 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(L) 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

```
skill(plasticBomb).
skill(bioBomb).
~skill(nuclearBomb).

safetyArea(field1).

@p1
+bomb(Terminal, Gate, BombType) :  skill(BombType)
   <-  !go(Terminal, Gate);
       disarm(BombType).

@p2
+bomb(Terminal, Gate, BombType) :  ~skill(BombType)
   <-  !moveSafeArea(Terminal, Gate, BombType).

@p3
+bomb(Terminal, Gate, BombType) :  not skill(BombType) &
                                   not ~skill(BombType)
   <-  .broadcast(tell, alter).

@p4
+!moveSafeArea(T,G,Bomb) :  true
   <-  ?safeArea(Place);
       !discoverFreeCPH(FreeCPH);
       .send(FreeCPH, achieve,
             carryToSafePlace(T,G,Place,Bomb)).
⋮
```

Figure 1.1. Examples of AgentSpeak Plans for a Bomb-Disarming Robot.

as atomic formulæ, but using a set of *action symbols* rather than predicate symbols.

Figure 1.1 shows an example of AgentSpeak code for the initial beliefs and plans of a bomb-disarming agent described in Section 1.1. Initially, the agent believes it is skilled on disarming plastic and biological bombs, but not skilled in nuclear bombs; it knows that "field1" is a safe area to leave a bomb that it cannot disarm. When this agent receives a message from an MDS79 robot saying that a biological bomb is at terminal `t1`, gate `g43`, a new event `+bomb(t1, g43, bioBomb)` is created. A bomb-disarming agent has three *relevant* plans for this event (identified by the labels `p1`, `p2`, `p3`), given that the event matches the triggering event of those three plans. However, only the context of the first plan is satisfied (`skill(bioBomb)`), so that the plan is *applicable*. In plans `p1`–`p3`, the context is used to decide whether

to attempt to disarm a bomb (in case the agent is skilled in disarming that type of bomb), to move it to a safe area (in case it is not skilled), or to set off a security alert (if it is not sure it is sufficiently skilled). As only the first plan is applicable, an intention based on it is created and the plan starts to be executed. It adds a sub-goal `!go(t1, g43)` (the plans for achieving this goal are not included here) and performs a basic action `disarm(BombType)`. In plan `p4`, we have an example of a test goal whereby the agent consults its own beliefs about where to take the bomb (`?safeArea(Place)`), and an example of an internal action used to send a message (`.send(...)`). The details of the AgentSpeak code in Figure 1.1 will be explained in the next sections.

1.2.1 Specifications and Syntactical Aspects

The BNF grammar in Figure 1.2 gives the AgentSpeak syntax as accepted by ***Jason*** . Below, `<ATOM>` is an identifier beginning with a 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 an atomic formulæ[2] was allowed in the original language, here a literal is used instead. This is either an atomic formulæ `p(`t_1`,...,`t_n`)`, $n \geq 0$, or `~p(`t_1`,...,`t_n`)`, where '`~`' denotes strong negation[3]. 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 [135]. 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

[2]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.

[3]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 1.3.1).

```
agent             → beliefs plans
beliefs           → ( literal "." )*
                    N.B.: a semantic error is generated if the
                    literal was not ground.
plans             → ( plan )+
plan              → [ "@" atomic_formula ]
                    triggering_event ":" context "<-" body "."
triggering_event  → "+"     literal
                  | "-"     literal
                  | "+" "!" literal
                  | "-" "!" literal
                  | "+" "?" literal
                  | "-" "?" literal
literal           → atomic_formula
                  | "~" atomic_formula
                  | <VAR>
default_literal   → literal
                  | "not" literal
                  | "not" "(" literal ")"
                  | term ("<"|"<="|">"|">="|"=="|"\\=="|"=") term
                  | literal ("=="|"\\=="|"=") literal
context           → "true"
                  | default_literal ( "&" default_literal )*
body              → "true"
                  | body_formula ( ";" body_formula )*
body_formula      → literal
                  | "!" literal
                  | "?" literal
                  | "+" literal
                  | "-" literal
atomic_formula    → <ATOM>["("list_of_terms")"] ["["list_of_terms"]"]
list_of_terms     → term ( "," term )*
term              → atomic_formula
                  | list
                  | <VAR>
                  | <NUMBER>
                  | <STRING>
list              → "["
                    [ term ( ( "," term )*
                           | "|" ( list | <VAR> )
                           )
                    ] "]"
```

Figure 1.2. BNF of the AgentSpeak Extension Interpreted by ***Jason***.

that purpose; *s* can be an agent's name (to denote the agent that communicated that information), or two special atoms, `percept` and `self`, that are used to denote that a belief arose from perception of the environment, or from the agent explicitly adding a belief to its own belief base from the execution of a plan body, respectively. The initial beliefs that are part of the source code of an AgentSpeak agent are assumed to be internal beliefs (i.e., as if they had a `[source(self)]` annotation), unless the belief has any

explicit annotation given by the user (this could be useful if the programmer wants the agent to have an initial belief about the environment or as if it had been communicated by another agent). Fore more on the annotation of sources of information for beliefs, see [153].

Plans also have labels, as first proposed in [18]. However, a plan label can now be any atomic formula, including annotations, although we suggest that plan labels use annotations (if necessary) but have a predicate symbol of arity 0, as in `aLabel` or `anotherLabel[chanceSuccess(0.7), expectedPayoff(0.9)]`. Annotations in formulæ used as plan labels can be used for the implementation of sophisticated applicable plan (i.e., option) selection functions. Although this is not yet provided with the current distribution of ***Jason***, it is straightforward for the user to define, e.g., decision-theoretic selection functions; that is, functions which use something like expected utilities annotated in the plan labels to choose among alternative plans. The customisation of selection functions is done in Java (by choosing a plan from a list received as parameter by the selection functions), and is explained in Section 1.3.1. Also, as the label is part of an instance of a plan in the set of intentions, and the annotations can be changed dynamically, this provides all the means necessary for the implementation of efficient intention selection functions, as the one proposed in [18]. 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`, suffices it to include a plan `-!g : true <- !g.` in the plan library. Note that the body can be empty as a goal is only removed from the body of a plan when the intended means chosen for that goal finishes successfully. It is also simple to specify a plan which, under specific condition, chooses to drop the intention altogether (by means of a standard internal action mentioned below).

Finally, as also introduced in [18], *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 [22] for a complete list). As an example (see Figure 1.1, plan `p4`), ***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 1.3.1). The `illocutionary_force`s 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 1.2.4. Finally, the `prop_content` is a literal (see `literal` in the grammar above).

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 [24]. 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 easily by the use of certain special standard internal actions, as internal actions are essentially Java code that can also be referred from plans.

1.2.2 Semantics and Verification

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 [169] (a widely-used notation for giving semantics to programming languages) was given then improved and extended in a series of papers [23, 24, 152, 153, 229]. In particular, [229] 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.

However, for space limitation, we are not able to include a complete formal account of the semantics of AgentSpeak here. In this section we will just provide the main intuitions behind the interpretation of AgentSpeak programs, and after that we will give examples of the rules that are part of the formal semantics.

Informal Semantics

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(L) did not elaborate on how designers specify such functions[4]). 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 cre-

[4]Our extension to AgentSpeak(L) in [18] 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.

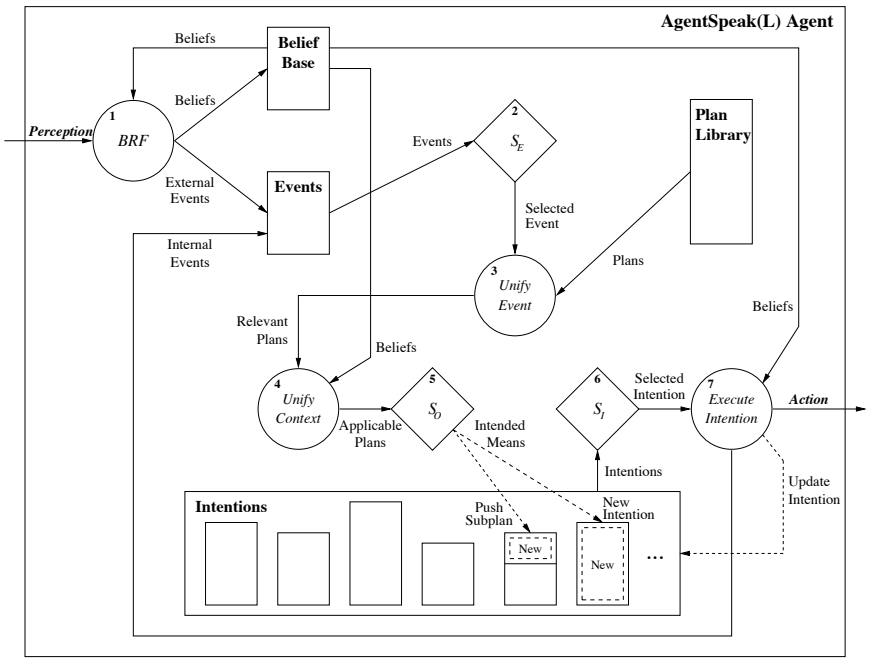

Figure 1.3. An Interpretation Cycle of an AgentSpeak Program [143].

ate 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 1.3 (reproduced from [143]). 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 plan on the top of an existing intention (if the event was an internal one), or creates a new intention in the set of intentions (if the event was external, i.e., generated from perception of the environment).

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

If the intention is to perform a basic action or a test goal, the set of intentions needs to be updated. In the case of a test goal, the belief base will be searched for a belief atom that unifies with the 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.

Formal Semantics

We emphasise again that the purpose of this section is to give a general idea of the style used for giving semantics to the language interpreted by ***Jason***. For a complete account of the formal semantics, we refer the interested reader to [229].

We have defined the formal semantics of AgentSpeak using operational semantics, a widely used method for giving semantics to programming languages and studying their properties [169]. The operational semantics is given by a set of rules that define a transition relation between configurations $\langle ag, C, M, T, s\rangle$ where:

- An agent program *ag* is, as defined above, a set of beliefs and a set of plans.
- An agent's circumstance C is a tuple $\langle I, E, A\rangle$ where:
  - I is a set of *intentions* $\{i, i', \ldots\}$. Each intention i is a stack of partially instantiated plans.
  - E is a set of *events* $\{(te, i), (te', i'), \ldots\}$. Each event is a pair (te, i), where te is a triggering event and i is an intention (a stack of plans in case of an internal event or $\top$ representing an external event).

    When the belief revision function, which is not part of the AgentSpeak interpreter but rather of the general architecture of the agent, updates the belief base, the associated events (i.e., additions and deletions of beliefs) are included in this set. These are called *external* events; internal ones are generated by additions or deletions in the agent's goals.
  - A is a set of *actions* to be performed in the environment. An action expression included in this set tells other architecture components to actually perform the respective action on the environment, thus changing it.
- M is a tuple $\langle In, Out, SI\rangle$ whose components register the following aspects of communicating agents:
  - In is the mail inbox: the system includes all messages addressed to this agent in this set. Elements of this set have the form $\langle mid, id, ilf, cnt\rangle$, where *mid* is a message identifier, *id* identifies the sender of the message, *ilf* the illocutionary force of the message, and *cnt* its content (which can be an AgentSpeak atomic formula, a set of AgentSpeak atomic formulæ, or a set of AgentSpeak plans, depending on the illocutionary force of the message).
  - Out is where the agent posts all messages it wishes to send to other agents; the underlying multi-agent system mechanism makes sure that messages included in this set are sent to the agent addressed in the message. Messages here have exactly the same

format as above, except that now *id* refers to the agent to which the message is to be sent.

- *SI* is used to keep track of intentions that were suspended due to the processing of communication messages; this is explained in more detail in the next section, but the intuition is: intentions associated to illocutionary forces that require a reply from the interlocutor are suspended, and they are only resumed when such reply has been received.

- T is the tuple $\langle R, Ap, \iota, \varepsilon, \rho \rangle$, used to keep temporary information that is required in subsequent stages within a single reasoning cycle; its components are:

  - R for the set of *relevant plans* (for the event being handled).
  - Ap for the set of *applicable plans* (the relevant plans whose context are true).
  - ι, ε, and ρ keep record of a particular intention, event and applicable plan (respectively) being considered along the execution of an agent.

- The current step s within an agent's reasoning cycle is symbolically annotated by $s \in \{\mathsf{ProcMsg}, \mathsf{SelEv}, \mathsf{RelPl}, \mathsf{ApplPl}, \mathsf{SelAppl}, \mathsf{AddIM}, \mathsf{SelInt}, \mathsf{ExecInt}, \mathsf{ClrInt}\}$, which stand for: processing a message from the agent's mail inbox, selecting an event from the set of events, retrieving all relevant plans, checking which of those are applicable, selecting one particular applicable plan (the intended means), adding the new intended means to the set of intentions, selecting an intention, executing the select intention, and clearing an intention or intended means that may have finished in the previous step.

Formally, all the selection functions an agent uses are also part of its configuration, (as is the social acceptance function that we mention below). However, as they are fixed, i.e., defined by the agent's designer when configuring the interpreter, we avoid including them in the configuration, for the sake of readability.

In order to keep the semantic rules clear, we adopt the following notations:

- If C is an AgentSpeak agent circumstance, we write C_E to make reference to the component E of C. Similarly for all the other components of a configuration.

- We write $T_\iota = \_$ (the underline symbol) to indicate that there is no intention being considered in the agent's execution. Similarly for T_ρ and T_ε.
- We write $i[p]$ to denote an intention i that has plan p on its top.

We now present a selection of the rules which define the operational semantics of the reasoning cycle of AgentSpeak. In the general case, an agent's initial configuration is $\langle ag, C, M, T, \mathsf{ProcMsg}\rangle$, where ag is as given by the agent program, and all components of C, M, and T are empty.

Updating the Set of Intentions: At the stage of the reasoning cycle where a relevant and applicable plan has been found for an event, the interpreter can then update the set of intentions. Events can be classified as external or internal (depending on whether they were generated from the agent's perception, or whether they were generated by the previous execution of other plans, respectively). Rule **ExtEv** says that if the event ε is external (which is indicated by T in the intention associated to ε) a new intention is created and its single plan is the plan p annotated in the ρ component. If the event is internal, rule **IntEv** says that the plan in ρ should be put on top of the intention associated with the event.

$$\frac{T_\varepsilon = \langle te, \mathsf{T}\rangle \qquad T_\rho = (p, \theta)}{\langle ag, C, M, T, \mathsf{AddIM}\rangle \longrightarrow \langle ag, C', M, T, \mathsf{SelInt}\rangle} \quad \textbf{(ExtEv)}$$
$$\textit{where:} \quad C'_I = C_I \cup \{\ [p\theta]\ \}$$

$$\frac{T_\varepsilon = \langle te, i\rangle \quad T_\rho = (p, \theta)}{\langle ag, C, M, T, \mathsf{AddIM}\rangle \longrightarrow \langle ag, C', M, T, \mathsf{SelInt}\rangle} \quad \textbf{(IntEv)}$$
$$\textit{where:} \quad C'_I = C_I \cup \{\ (i[p])\theta\ \}$$

Note that, in rule **IntEv**, the whole intention i that generated the internal event needs to be inserted back in C_I, with p as its top. This issue is related to suspended intentions, see rule **Achieve**.

Intention Selection: Rule $\mathbf{IntSel_1}$ uses an agent-specific function ($\mathcal{S}_\mathcal{I}$) that selects an intention (i.e., a stack of plans) for processing, while rule $\mathbf{IntSel_2}$ takes care of the situation where the set of intentions is empty (in which case, the reasoning cycle is simply restarted).

$$\frac{C_I \neq \{\} \qquad \mathcal{S}_\mathcal{I}(C_I) = i}{\langle ag, C, M, T, \mathsf{SelInt}\rangle \longrightarrow \langle ag, C, M, T', \mathsf{ExecInt}\rangle} \quad \textbf{(IntSel}_\mathbf{1}\textbf{)}$$
$$\textit{where:} \quad T'_\iota = i$$

$$\frac{C_I = \{\}}{\langle ag, C, M, T, \mathsf{SelInt}\rangle \longrightarrow \langle ag, C, M, T, \mathsf{ProcMsg}\rangle} \quad (\textbf{IntSel}_2)$$

Executing a Plan Body: Below we show part of the group of rules that define the effects of executing the body of a plan. The plan being executed is always the one on top of the intention that has been previously selected. Observe that all the rules in this group discard the intention ι; another intention can then be eventually selected.
Achievement Goals: this rule registers a new internal event in the set of events E. This event will, eventually, be selected and handled at another reasoning cycle.

$$\frac{T_\iota = i[head \leftarrow !at\,;h]}{\langle ag, C, M, T, \mathsf{ExecInt}\rangle \longrightarrow \langle ag, C', M, T, \mathsf{ProcMsg}\rangle} \quad (\textbf{Achieve})$$

$$\textit{where:} \quad \begin{array}{rcl} C'_E & = & C_E \cup \{\langle +!at, i[head \leftarrow h]\rangle\} \\ C'_I & = & C_I \setminus \{T_\iota\} \end{array}$$

Note how the intention that generated the internal event is removed from the set of intentions C_I. This denotes the idea of *suspended intentions* (see [23] for details).
Updating Beliefs: rule **AddBel** below simply adds a new event to the set of events E. The formula $+b$ is removed from the body of the plan and the set of intentions is updated properly. There is also a **DelBel** rule, for deleting beliefs, which works similarly. In both rules, the set of beliefs of the agent should be modified in a way that either the ground atomic formula b (with annotation "`source(self)`") is included in the new set of beliefs (rule **AddBel**) or it is removed from there (rule **DelBel**).

$$\frac{T_\iota = i[head \leftarrow +b\,;h]}{\langle ag, C, M, T, \mathsf{ExecInt}\rangle \longrightarrow \langle ag', C', M, T, s\rangle} \quad (\textbf{AddBel})$$

$$\textit{where:} \quad \begin{array}{rcl} ag'_{bs} & = & ag_{bs} + b[\texttt{source(self)}] \\ C'_E & = & C_E \cup \{\langle +b[\texttt{source(self)}], \mathsf{T}\rangle\} \\ C'_I & = & (C_I \setminus \{T_\iota\}) \cup \{i[head \leftarrow h]\} \\ s & = & \begin{cases} \mathsf{ClrInt} & \text{if } h = \top \\ \mathsf{ProcMsg} & \text{otherwise} \end{cases} \end{array}$$

Verification

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 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 on which this project is based 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 [19–21, 26].

1.2.3 Software Engineering Issues

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 with ***Jason***, existing methodologies that consider BDI agents, such as Prometheus [164], should be perfectly suitable for that purpose. In the book the authors show an example of the use of JACK (see Chapter 7) 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 Chapter 6), 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. 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 1.2.1). Internal actions necessarily have a boolean value returned, so they are declaratively represented within a logic program in AgentSpeak— in effect, we maintain the agent program as 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 many 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).

1.2.4 Other Features of the Language

Communication in AgentSpeak

The performatives that are currently available for communication in AgentSpeak are largely inspired by KQML performatives. 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* informs *r* about the sentence in the message (i.e., the message content) for *r* to include the sentence in its knowledge base;

`untell`: *s* informs *r* about the sentence in the message (i.e., the message content) for *r* to exclude the sentence from its knowledge base;

`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;

`tell-how`: *s* informs *r* of a plan;

`untell-how`: *s* requests that *r* disregard a certain plan (i.e., eliminate that plan from its plan library);

`ask-if`: *s* wants to know if the content of the message is true for *r*;

`ask-all`: *s* wants all of *r*'s answers to a question;

`ask-how`: *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 to.

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 easily be derived. For more elaborate conceptions of trust and power see [42]).

As a simple example of how the user can customise this power relation in ***Jason***, we may consider that a CPH903 robot always does what an MDS79 robot asks. The following agent customisation class implements that (see Figure 1.7):

```
package cph;
import jason.asSemantics.Agent;

public class CPHAgent extends Agent {

  public boolean socAcc(Message m) {
    if (m.getSender().startsWith("mds") &&
        m.getIlForce().equals("achieve")) {
      return true;
    } else {
      return false;
    }
  }
}
```

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). For example, the triggering event of MDS79's plan `pb1`, seen later in Figure 1.8, uses this annotation to identify the sender of the bid.

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 [18], we dealt with the problem by creating a separate belief base where the internal beliefs are included or removed.

For space restrictions, we do not discuss the interpretation of received messages with each of the available illocutionary forces. This is presented both formally and informally in [229].

Cooperation in AgentSpeak

Coo-BDI (Cooperative BDI, [4]) 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 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 [4].

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 [4] 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 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.

1.3 Platform

1.3.1 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. Figure 1.4 shows an example of this configuration file for the Heathrow scenario. Briefly, the environment is implemented in a class named HeathrowEnv; the system has three types of agents: five instances of MDS79, ten CPH903, and three bomb-disarmers; MDS79 agents have a customised agent class and CPH903 have customised agent and agent architecture classes.

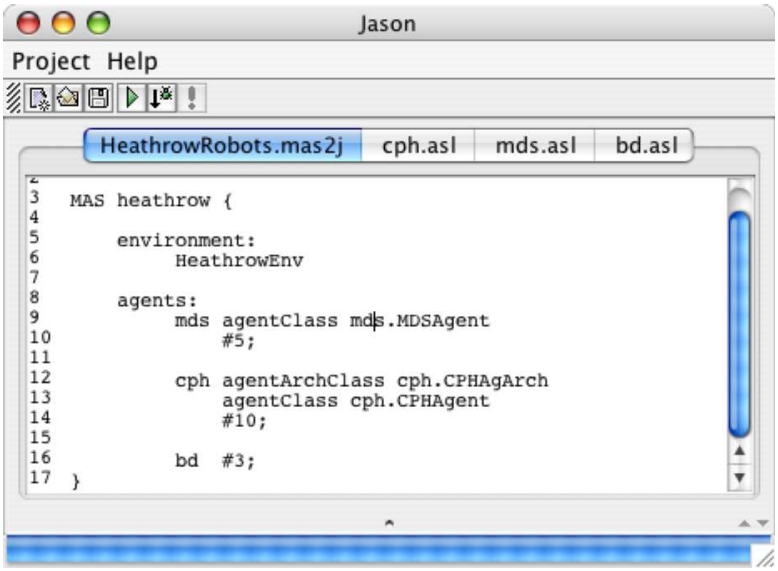

Figure 1.4. ***Jason*** IDE.

The BNF grammar in Figure 1.5 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 `architecture` is used to specify which of the two overall agent architectures available with ***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

```
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" | "retrieve")
            | "intBels" "=" ( "sameFocus" | "newFocus" )
            | "verbose" "=" <NUMBER>
```

Figure 1.5. BNF of the Language for Configuring Multi-Agent Systems.

note that the user's environment and customisation classes remain the same with both (system) architectures.

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 architecture.

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 lower-case 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 with ***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 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`, `requeue`, or `retrieve`; the `discard` option means that external events for which there are no applicable plans are discarded, whereas the `requeue` option is used when such events should be inserted back at the end of the list of events that the agent needs to handle. When option `retrieve` is selected, the user-defined `selectOption` function is called even if the set of relevant/applicable plans is empty. This can be used, for example, for allowing agents to request plan from other agents who may have the necessary know-how that the agent currently lacks, as mentioned in Section 1.2.4 and described in detail in [4].

`intBels`: options are either `sameFocus` or `newFocus`. When internal beliefs are added or removed explicitly within the body of a plan, the associated event is a triggering event for a plan, the intended means resulting from the applicable plan chosen for that event is pushed on top of the intention (i.e., the focus of attention) which generated the event, if the `sameFocus` option is used). If the `newFocus` option is used, the event is treated as external (i.e., as the addition or deletion of belief from perception of the environment), creating a new focus of 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 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`.

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 Section 1.3.1.0; in the latter case, the user must provide

```
public class HeathrowEnv extends Environment {

  Map agsLocation = new HashMap();

  public List getPercepts(String agName) {
     if ( ... an unattended luggage was found ... ) {
         // all agents will perceive the fact that
         // there is unattendedLuggage
         getPercepts().add( Term.parse("unattendedLuggage") );
     }

     if (agName.startsWith("mds")) {
         // mds robots will also perceive their location
         List customPerception = new ArrayList();
         customPerception.addAll(getPercepts());
         customPerception.add(agsLocation.get(agName));
         return customPerception;
     } else {
         return getPercepts();
     }
  }

  public boolean executeAction(String ag, Term action) {
     if (action.hasFunctor("disarm")) {
        ... the code that implements the disarm action
        ... on the environment goes here
     } else if (action.hasFunctor("move")) {
        ... the code for changing the agents' location and
        ... updating the agsLocation map goes here
     }
     return true;
  }
}
```

Figure 1.6. Simulated Environment of the Airport Scenario.

an implementation of the simulated environment. This is done directly in a Java class that extends the ***Jason*** base Environment class. For example, a very simple simulated version of the environment for the Heathrow airport scenario is shown in Figure 1.6.

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 assumption; in that case the latter should have all atomic formulæ that are to be perceived as explicitly false (strong negation) by the agents. It is possible to send indi-

vidualised perception, i.e., 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 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). Customisation of agent's individual perception within the environment is done by overriding the "`getPercepts(agName)`" method; the default methods simply provide all current environment properties as percepts to all agents. In the example above, only MDS79 robots will percept their location at the airport.

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 (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 successful 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 1.7). The three first selection functions are discussed extensively in the AgentSpeak literature (see Section 1.2.2 and Figure 1.3). 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 [229] and Section 1.2.4. 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

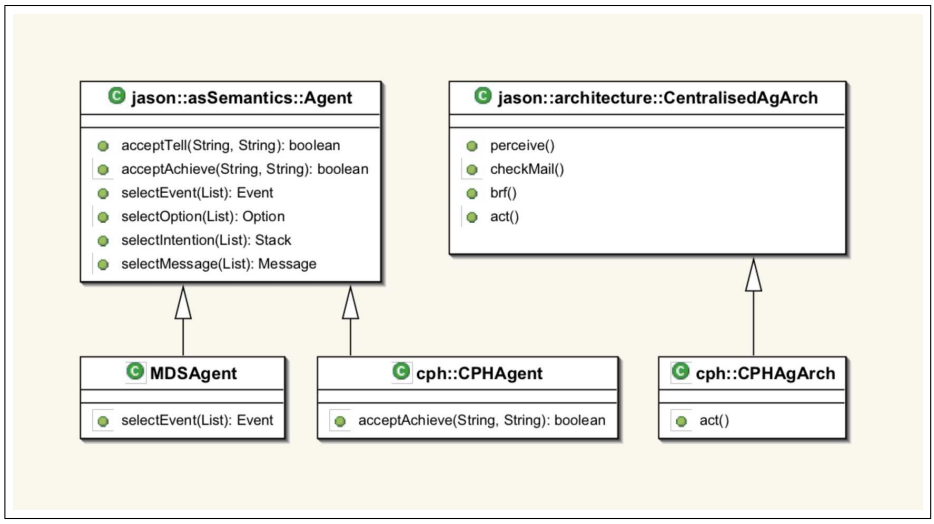

Figure 1.7. Customising Agents for the Airport Scenario.

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.

For example, in the Heathrow scenario, MDS79 robots must give priority to events related to unattended luggage. A customised MDS79 agent class which overrides the select event method can implement this priority, as follows:

```
public class MDSAgent extends Agent {

  public Event selectEvent(List evList) {
    Iterator i = evList.iterator();
    while (i.hasNext()) {
       Event e = (Event)i.next();
       if (e.getTrigger().getFunctor().equals(
                                        "unattendedLuggage")) {
          i.remove();
          return e;
       }
    }
    return super.selectEvent(evList);
}
```

Similarly, the user can customise the functions defining the overall agent architecture (see Figure 1.7, 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, 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, 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 [25]).

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 may, in rare occasions, be interested in modelling a "paraconsistent" agent, which can be easily done.

Suppose, for example, that in no circumstance a CPH903 robot is allowed to disarm a bomb. To prevent them from performing this action, even if they have decided to do so (e.g., they could be infected by a software virus), the developer could override the act method in the CPH903's customised AgArch class and ensure that the selected action is not `disarm` before allowing it to be executed in the environment:

```
public class CPHAgArch extends CentralisedAgArch {
  public void act() {
     // get the current action to be performed
     Term action = fTS.getC().getAction().getActionTerm();
```

```
    if ( !action.getFunctor().equals("disarm") ) {
      // ask the environment to execute the action
      fEnv.executeAction(getName(), action))
      ...
    }
}
```

Defining New Internal Actions

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 [18], 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.

As an example, when an unattended luggage is perceived by the MDS79 robots they send bids to each other that represent how suitable they are for coping with the new situation (see Figure 1.8, plan `pn2`). The robot with the highest bid will be relocated to handle the unattended luggage. Now, suppose a complex formula is used to calculate the initial bid and further checks and calculations are requested to adjust the bid; clearly imperative languages are normally more suitable for implementing this kind of algorithms. The user can thus use the following Java class to implement this algorithm, and refer to it from within the AgentSpeak code as `mds.calculateMyBid(Bid)`:

```
package mds;

import ...

public class calculateMyBid implements InternalAction {

  public boolean execute(TransitionSystem ts, Unifier un,
                         Term[] args) throws Exception {
    int bid = ... a complex formula ...;
    ... plus complex algorithm and calculations
        for adjusting the agent's bid ...

    un.unifies(args[0], Term.parse(""+bid));
    return true;
}}
```

It is important that the class has an execute method declared *exactly* as above, since ***Jason*** uses class introspection to call it. The internal action's arguments are passed as an array of Strings. Note that this is the third argument of the execute method. The first argument is the transition system (as defined by the operational semantics of AgentSpeak), which contains all information about the current state of the agent being interpreted. The second is the unifying function currently determined by the execution of the plan, or the checking of whether the plan is applicable[5]; the unifying functions is important in case the value bound to AgentSpeak variables need to be used in the implementation of the action.

1.3.2 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. 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 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 1.9.

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

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

```
free.  // I'm not currently handling unattended luggage

+unattendedLuggage(Terminal,Gate) :  true
   <-  !negotiate.

@pn1
+!negotiate :  not free
   <-  .broadcast(tell, bid(0)).

@pn2
+!negotiate :  free
   <-  .myName(I); // Jason internal action
       +winner(I); // belief I am the negotiation winner
       +bidsCount(0);
       mds.calculateMyBid(Bid); // user internal action
       +myBid(Bid);
       .broadcast(tell, bid(Bid)).

@pb1 // for a bid better than mine
+bid(B)[source(Sender)] :
                  myBid(MyBid) & MyBid < B &
                  .myName(I) & winner(I)
   <-  -winner(I);
       +winner(Sender).

@pb2 // for other bids (and I'm still the winner)
+bid(B) : .myName(I) & winner(I)
   <-  mds.addBidCounter;
       !endNegotiation.

@pend1 // all bids was received
+!endNegotiation :  bidsCount(N) & numberOfMDS(M) & N >= M
   <-  -free; // I'm no longer free
       !checkUnattendedLuggage.

@pend2 // void plan for endNegotiation not to fail
+!endNegotiation :  true <- true.
⋮
```

Figure 1.8. Example of AgentSpeak Plans for a Security Robot.

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

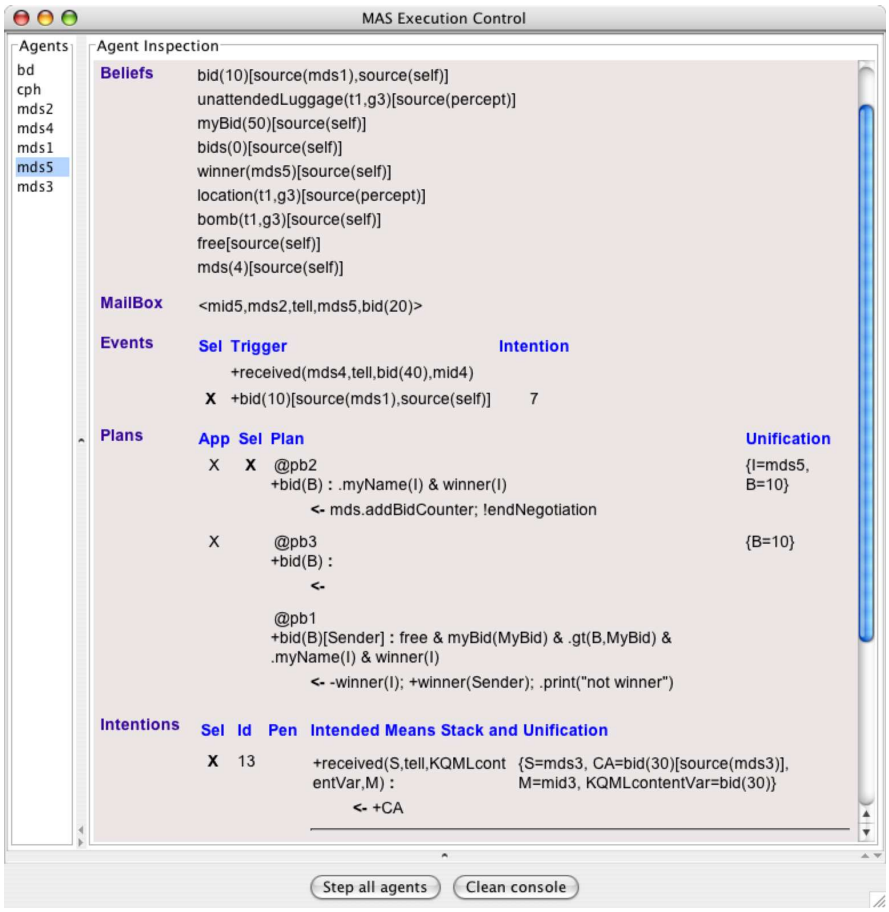

Figure 1.9. ***Jason***'s Mind Inspector.

and practical exercises) for teaching Agent-Oriented Programming with ***Jason***. Because of the elegance and simplicity of the core agent language interpreted by ***Jason***, at the same time having all the important elements for the implementation of BDI-based reactive planning systems, we think ***Jason*** can become an important tool for teaching multi-agent systems.

1.3.3 Standards Compliance, Interoperability, and Portability

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 "system architec-

tures" available with ***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 straightforward to produce another system architecture which uses, e.g., JADE (see Chapter 5) for FIPA-compliant distribution and management of agents in a multi-agent system.

1.3.4 Applications Supported by the Language and the Platform

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 [98] and dMARS [126] for example have appeared in the literature [238, 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 [74]. 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 [25]; it includes a high-level language called ELMS [162] for defining multi-agent environments. This approach was used to develop a simple social simulation on social aspects of urban growth is also mentioned (the simulation was briefly presented in [131]). 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) [223].

1.4 Final Remarks

Jason is constantly being 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 careful way so as to avoid the language becoming cumbersome and, most importantly, having formal semantics of most of the essential, if not all, features available in ***Jason***. We have 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. In particular, we aim to contribute 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, Álvaro F. Moreira, Fabio Y. Okuyama, Denise de Oliveira, Carmen Pardavila, Marios Richards, Maíra R. Rodrigues, Rosa M. Vicari, Willem Visser, Michael Wooldridge.