The *Jason* Agent Programming Language

Chapter 3

Fabrizio Natale

Agent programming

- First, let's make a distinction between:
  - An *agent program;*
  - An *agent architecture*.

Agent programming

- An *agent architecture* is the software framework within which an agent program runs
- An *agent program* directs the agent behaviour

Much of what the agent effectively does is determined by the architecture itself, without the programmer having to worry about it

BDI agents

- BDI agents are defined as *reactive planning systems* :
- Systems that are not meant to compute the value of a function and terminate
- But rather designed to be permanently running, reacting to some form of ‘events’ from the environment
- Plans are courses of action that agents commit to execute so as to handle such events.

Jason - Main categories

- The language interpreted by Jason is an extension of AgentSpeak, which is based on the BDI architecture.
- The BDI architecture components are:
  - **Belief base**: the interpreter will perceive the environment, constantly, and update the belief base accordingly;
  - **Goals** (or *Desires*) achieved by the execution of
  - **Plans** (or *Intentions*).

Jason – Beliefs

- Any agent has a *belief base* which in its simplest form is a collection of literals.
- Information is represented in symbolic form by *predicates* such as:
- **tall(alessandro)** which expresses a particular property of an object or individual;
- **likes(marco, chocolate)** which represents a certain relationship between two objects.

Jason – Beliefs

- A literal is such a predicate or its negation
- When a formula such as **likes(marco, chocolate)** appears in an agent's belief, that is only meant to express the fact that the agent currently *believes* that to be true.

Jason – Basics of logic programming

- Any symbol starting with a lowercase letter is called an *atom* which is used to represent particular individuals or objects. E.g. **faber** is an atom.
- A symbol starting with an uppercase is interpreted as a *logical variable*.
- E.g. **Person** is a variable.
- Numbers and strings are also classified as *constants*.

Jason – Basics of logic programming

- Initially variables are *free* or *uninstantiated* and once *instantiated* or *bound* to a value they maintain that value throughout their *scope*.
- Variables can be bound to a constant (i.e. an atom, a number, or a string), or to a more complex data type called *structure*.
- A structure can be defined as follows: **staff("Faber", 23, student)**

Jason – Basics of logic programming

- That structure can represent a student in a university.
- Structures start with an atom (called the *functor*) and are followed by a number of terms (called *arguments*) separated by commas and enclosed in parentheses.
- Predicates represent a logical proposition and are called *facts*.

Jason – Basics of logic programming

- The number of arguments of a predicate/ structure is important, and is called its *arity*.
- A particular structure is referred to by its functor and its arity: from the previous example, the structure is referred to by "**staff/3**" and it must have always exactly three terms as arguments (otherwise is a different structure)

Jason – Basics of logic programming

- A special type of structure are lists. They are represented as [item1, item2].
- A special operator ("|") can be used to separate the first item in a list from the list of all remaining items in it. E.g. [H|T] matched with [1,2,3] follows that H is instantiated with 1 and T with the list [2,3].

Jason – Annotations

- *Annotations* are complex terms that provide details that are strongly associated with one particular belief.
- open(school)[expires(august)]
- Means that the agent believes that school is open, but as soon as august comes in, that should be no longer believed to hold.
- We could write also open_until(school,august)

Jason – Annotations

- There are two advantages of the way annotations are used in Jason:
  - Elegant notation;
  - Easy management of the belief base.

Jason – Annotations

- It is also possible to use nested annotations:
- loves(maria,bob)[source(john)[source(maria)]
- There are belief annotations which do have specific meaning for the interpreter. One of those is the **source** annotation.
- It is used to record what was the source of the information leading to a particular belief.

Jason – Annotations

- There are three different types of information source for agents in multi-agent systems:
  - **Perceptual information**: an agent acquires certain beliefs as a consequence of the sensing of its environment;
  - **Communication**: an agent receives information from other agents and it is useful to know exactly which agent provided each information;
  - **Mental notes**: these beliefs are added to the belief base by the agent itself as part of an executing plan. They can be useful to remind of things that happened in the past, or things the agent has done or promised.

Jason – Annotations

- For the first type of information the interpreter automatically adds an annotation **source(percept).**
- For mental notes, we'll have **source(self).**
- Finally, any other source of information annotated by interpreter will be the name of the agent which sent a communication message such as: **likes(marco, chocolate) [source(marco)]**

Jason – Negations

- The closed world assumption may be understood as follows:
  - *Anything that is neither known to be true, nor derivable from the known facts using the rules in the program, is assumed to be false.*

Jason – Negations

- The "**not**" operator is used to mean that the negation of a formula is true if the interpreter *fails* to derive the formula using the facts and rules in the program.
- There's another type of negation, denoted by the "**~**" operator and called *strong negation;* is used to express that an agent *explicitly believes something to be false*.

Jason – Negations

- The belief **colour(box1,white)** means that the agent believes the colour of box1 is white;
- whereas the belief **~colour(box1,white)** means that the agent believes that it is not the case that the colour of box1 is white.

Jason – Negations

- Example:

```
colour(box1,blue)[source(bob)]
~colour(box1,white)[source(john)]
colour(box1,red)[source(percept)]
colourblind(bob)[source(self),degOfCert(0.7)]
liar(bob)[source(self),degOfCert(0.2)]
```

Jason – Rules

- Consider the following rules:

```
likely_colour(C,B) :- colour(C,B)[source(S)] & (S == self |
S==percept)
likely_colour(C,B) :- colour(C,B)[degOfCert(D1)] & not
(colour(_,B)[degOfCert(D2)] & D2 > D1) & not ~ colour(C,B)
```

- To the left of the ":-" operator, can be only one literal, which is the conclusion to be made if the condition to the right is satisfied.

Jason – Goals

- Whereas beliefs express properties that are believed to be true of the world in which the agent is *situated*, goals express the properties of the states of the world that the agent *wishes to bring about*.
- Normally, a *goal* is represented as the commitment of the agent to act so as to change the world to a state in which the agent will believe that the goal is indeed true.
- E.g. own(house)

Jason – Goals

- There are two types of goal in AgentSpeak: *achievements goals* and *test goals*.
- The first ones are denoted by the "!" operator, while the others by the "?" operator.
- The previous example shows a particular use of an achievement goal. A test goal is used, instead, simply to retrieve information that is available in the agent's belief base.

Jason – Goals

- When we write **?bank_balance(BB)**, that is typically because we want the logical variable **BB** to be instantiated with the specific amount of money the agent currently *believes* its bank balance is.
- Test goals can also lead to the execution of plans in certain circumstances.

Jason – Plans

- An AgentSpeak plan has three distinct parts: the *triggering event*, the *context*, and the *body*.
- The triggering event and the context are called the *head* of the plan.
- The three plan parts are syntactically separated by ":" and "<-" as follows:

```
triggering_event : context <- body
```

Jason – Plans: Triggering event

- There are two important aspects of agent behaviour: reactiveness and pro-activeness.
- Goals determine the agent's pro-active behaviour.
- While acting so as to achieve their goals, agents need to be attentive to changes in their environment.

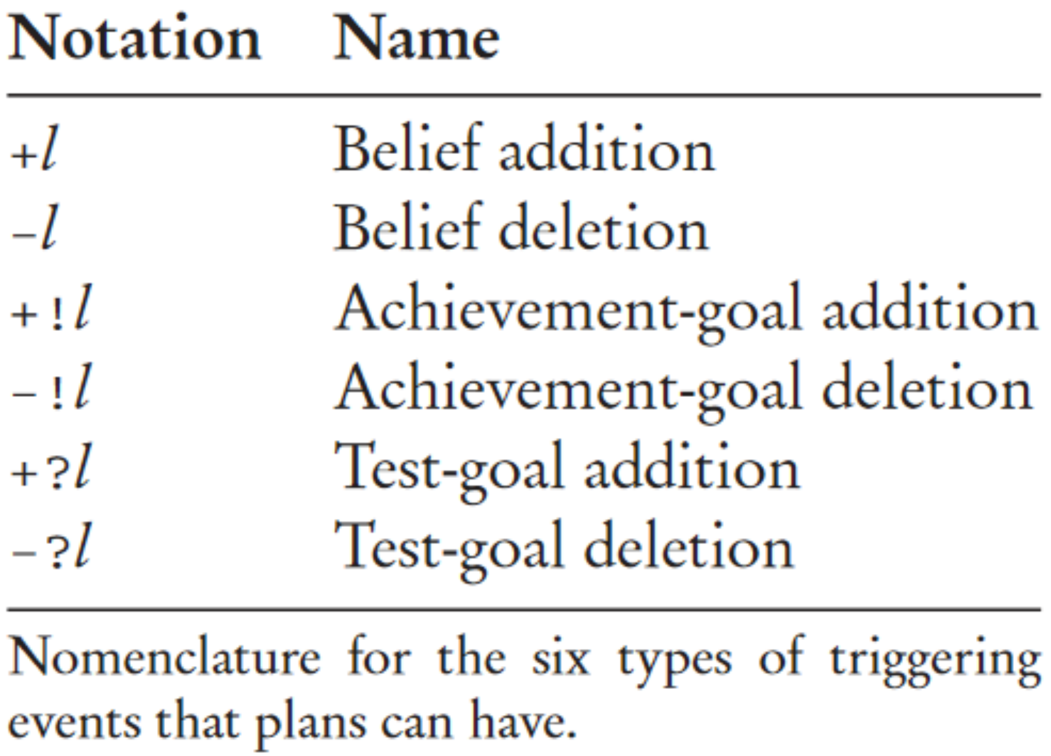
Jason – Plans: Triggering event

- Changes in the environment can also mean that there are new opportunities for the agent to do thing, perhaps considering adopting new goals that they previously did not have or indeed dropping existing goals.
- Two types of changes in an agent's mental attitudes (*changes in beliefs* and *changes in the agent's goals*) create the events upon which agents will act. Such changes can be of two types: *addition* and *deletion*.

Jason – Plans: Triggering event

- **The triggering event part of a plan exists precisely to tell the agent, which are the specific events for which the plan is to be used.**
- If the triggering event of a plan matches a particular event, we say that the plan is *relevant* for that particular event.

Jason – Plans: Triggering event



Notation	Name
$+l$	Belief addition
$-l$	Belief deletion
$+!l$	Achievement-goal addition
$-!l$	Achievement-goal deletion
$+?l$	Test-goal addition
$-?l$	Test-goal deletion

Nomenclature for the six types of triggering events that plans can have.

Fabrizio Natale

Jason – Plans: Triggering event

- Events for belief additions and deletions happen when the agent updates its beliefs according to its perception of the environment obtained at every reasoning cycle.
- Events due to the agent having new goals happen mostly as a consequence of the execution of other plans, but also as a consequence of agent communication.
- The goal deletion types of events are used for handling plan failure.

Jason – Plans: Triggering event

- Test goals are normally used to retrieve simple information from the belief base.

  For example, I normally remember how many packets of pasta I have in the larder.
- When I need that information, I just retrieve it from memory.
- However, if I happen not to have that information to hand, I may need to perform an action specifically to find out this information.
- The action might be to ask someone else, or to go to the larder myself.

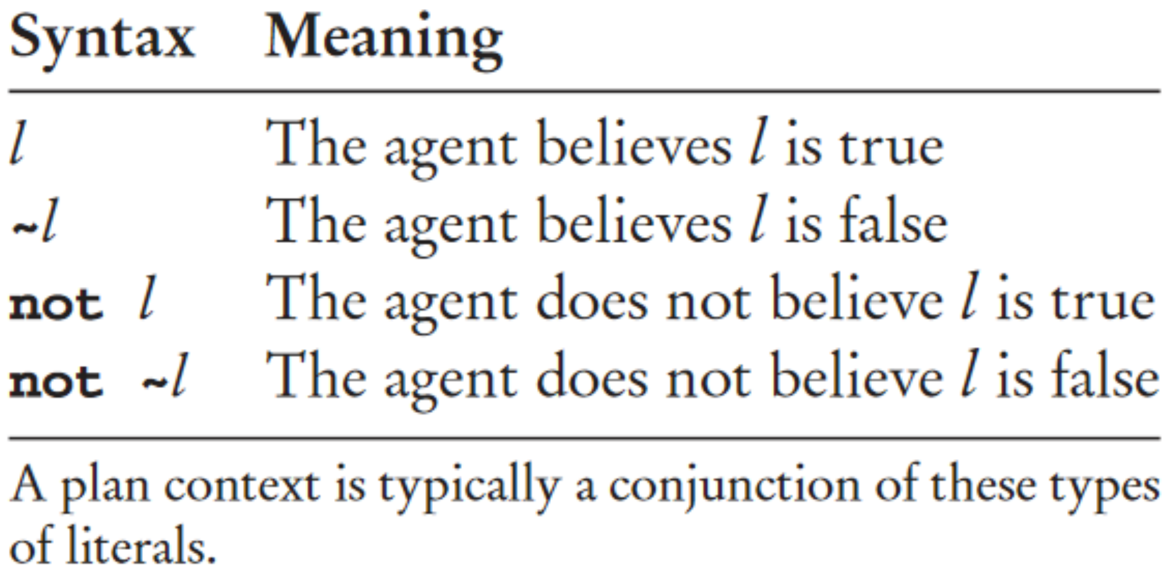
Jason – Plans: Context

- The *context* of a plan also relates to an important aspect of reactive planning systems.
- The context of a plan is used precisely for checking the current situation so as to determine whether a particular plan is likely to succeed in handling the event given the latest information the agent has about its environment.
- A plan is only chosen for execution if its context is a *logical consequence* of the agent's beliefs.

Jason – Plans: Context

- Contexts are used to define when a plan should be considered *applicable*.
- It is often the case that a context is simply a conjunction of *default literals* and relational expressions.
- Default literals are literals which may have another type of negation ("**not**" operator)
- Logical expressions can appear by combining literals with operators **and** ("&") and **or** ("|").

Jason – Plans: Context



Syntax	Meaning
l	The agent believes *l* is true
~*l*	The agent believes *l* is false
not *l*	The agent does not believe *l* is true
not ~*l*	The agent does not believe *l* is false

A plan context is typically a conjunction of these types of literals.

Jason – Plans: Context

- Examples of plan contexts could be as follows:

```
+!prepare(Something) : number_of_people(N) &
stock(Something, S) & S > N <- …

+!buy(Something) : not ~legal(Something) &
price(Something, P) & bank_balance(B) & B > P <-
…
```

Jason – Plans: Body

- The body of a plan is simply a sequence of formulæ determining a course of action, one that will, hopefully, succeed in handling the event that triggered the plan.
- Another important construct appearing in plan bodies is that of a goal this allows us to say what are the (sub)goals that the agent should adopt and that need to be achieved in order for that plan to handle an event successfully.
- We can refer to the term *subgoals*, given that the plan where they appear can itself be a plan to achieve a particular goal – recall that the triggering events allow us to write plans to be executed when the agent has a new goal to achieve.

Jason – Plans: Body

- The body of a plan defines a course of action for the agent to take when an event that matches the plan’s triggering event has happened and the context of the plan is true in accordance with the agent’s beliefs (and the plan is chosen for execution).
- The course of action is represented by a sequence of formulæ, each separated from the other by ‘;’.

Jason – Plans: Body

- There are six different types of formulæ that can appear in a plan:
  - actions;
  - achievements goals;
  - test goals;
  - mental notes;
  - internal actions;
  - expressions;

Jason – Plans label

- We can give a specific label to a plan.
- It is sometimes necessary to be able to refer to a particular plan that the agent is running.
- The general notation for plans with a specific label is as follows:

```
@label t_event: context <- body
```