Multi-Agent Oriented Programming
– Agent-Oriented Programming –

The Jason Agent Programming Language

O. Boissier, R.H. Bordini, J.F. Hübner, A. Ricci

May 2014

Outline

- Origins and Fundamentals
- Features
- Use of Jason within a Multi-Agent System
- Current Shortfalls and Future Trends

Agent Oriented Programming

Use of mentalistic notions and a societal view of computation

Various sophisticated abstractions
- Agent: Belief, Goal, Intention, Plan (this course)
- Organisation: Group, Role, Norm (see next course)
- Interaction: Speech Acts, Interaction protocols (this course)
- Environment: Artifacts, Percepts, Actions (see next course)

- Reacting to events × long-term goals
- Course of actions depends on circumstance
- Plan failure (dynamic environments)
- Rational behaviour
- Social ability
- Combination of theoretical and practical reasoning
AgentSpeak
the foundational language for Jason

- Originally proposed by Rao (1996)
- Programming language for BDI agents
- Elegant notation, based on logic programming
- Inspired by PRS (Georgeff & Lansky), dMARS (Kinny), and BDI Logics (Rao & Georgeff)
- Abstract programming language aimed at theoretical results

Jason
a practical implementation of AgentSpeak

- Jason implements the operational semantics of a variant of AgentSpeak
- Has various extensions aimed at a more practical programming language (e.g. definition of the MAS, communication, ...)
- Highly customised to simplify extension and experimentation
- Developed by Rafael H. Bordini and Jomi F. Hübner

Basics

- As in Prolog, any symbol (i.e. a sequence of characters) starting with a lowercase letter is called an atom
- An atom is used to represent particular individuals or objects
- A symbol starting with an uppercase letter is interpreted as a logical variable
- Initially variables are free or uninstantiated and once instantiated or bound to a particular value, they maintain that value throughout their scope (plan)
- Variables are bound to values by unification; a formula is called ground when it has no more uninstantiated variables.
Main Language Constructs and Runtime Structures

- **Beliefs**: represent the information available to an agent (e.g. about the environment or other agents)
- **Goals**: represent states of affairs the agent wants to bring about
- **Plans**: are recipes for action, representing the agent’s know-how
- **Events**: happen as a consequence to changes in the agent’s beliefs or goals
- **Intentions**: plans instantiated to achieve some goal

Jason basic reasoning cycle

- perceives the environment and update belief base
- processes new messages
- selects event
- selects *relevant* plans
- selects *applicable* plans
- creates/updates intention
- selects intention to execute

Main Architectural Components

- **Belief base**: where beliefs are stored
- **Set of events**: to keep track of events the agent will have to handle
- **Plan library**: stores all the plans currently known by the agent
- **Set of Intentions**: each intention keeps track of the goals the agent is committed to and the courses of action it chose in order to achieve the goals for one of various foci of attention the agent might have
**Beliefs – Representation**

**Syntax**

Beliefs are represented by annotated literals of first order logic:

\[
\text{functor}(\text{term}_1, \ldots, \text{term}_n)[\text{annot}_1, \ldots, \text{annot}_m]
\]

**Example (belief base of agent Tom)**

- `red(box1)[source(percept)]`
- `friend(bob, alice)[source(bob)]`
- `lier(alice)[source(self), source(bob)]`
- `\sim lier(bob)[source(self)]`

**Beliefs – Dynamics I**

**by perception**

Beliefs annotated with `source(percept)` are automatically updated accordingly to the perception of the agent.

**by intention**

The operators `+` and `-` can be used to add and remove beliefs annotated with `source(self)`.

- `+lier(alice); // adds lier(alice)[source(self)]`
- `-lier(john); // removes lier(john)[source(self)]`
- `+-lier(john); // updates lier(john)[source(self)]`

**Goals – Representation**

**Types of goals**

- Achievement goal: `goal to do`
- Test goal: `goal to know`

**Syntax**

Goals have the same syntax as beliefs, but are prefixed by `!` (achievement goal) or `?` (test goal).

**Example (initial goal of agent Tom)**

- `!write(book)`
Goals – Dynamics I

by intention
the operators \(!\) and \(?\) can be used to add a new goal annotated with `source(self)`

```
... // adds new achievement goal \(!\text{write(book)}\)[source(self)]
!\text{write(book)};

// adds new test goal \(?\text{publisher(P)}\)[source(self)]
?\text{publisher(P)};
...```

Goals – Dynamics II

by communication – achievement goal
when an agent receives an `achieve` message, the content is a new achievement goal annotated with the sender of the message

```
.send(tom,achieve,\text{write(book)}) ; // sent by Bob
// adds new goal \text{write(book)}[source(bob)] for Tom
...
.send(tom,unachieve,\text{write(book)}) ; // sent by Bob
// removes goal \text{write(book)}[source(bob)] for Tom
```

Goals – Dynamics III

by communication – test goal
when an agent receives an `askOne` or `askAll` message, the content is a new test goal annotated with the sender of the message

```
.send(tom,askOne,\text{published(P)},\text{Answer}) ; // sent by Bob
// adds new goal \text{?publisher(P)}[source(bob)] for Tom
// the response of Tom will unify with Answer
```
An AgentSpeak plan has the following general structure:

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

where:
- the *triggering event* denotes the events that the plan is meant to handle (cf. events description)
- the *context* represents the circumstances in which the plan can be used
- the *body* is the course of action to be used to handle the event if the context is believed to be true at the time a plan is being chosen to handle the event

**Plans – Operators for Plan Context**

**Boolean operators**
- & (and)
- | (or)
- not (not)
- = (unification)
- >, >= (relational)
- <, <= (relational)
- == (equals)
- \\= (different)

**Arithmetic operators**
- + (sum)
- - (subtraction)
- * (multiply)
- / (divide)
- div (divide – integer)
- mod (remainder)
- ** (power)

**Plans – Operators for Plan Body**

A plan body may contain:
- Belief operators (+, -, ++)
- Goal operators (!, ?, !!)
- Actions (internal/external) and Constraints

**Example (plan's body)**

```
+beer : time_to_leave(T) & clock.now(H) & H >= T
<= !g1;  // new sub-goal suspending plan execution
  !g2;    // new goal not suspending plan execution
  +b1(T-H); // adds new belief
  -+b2(T*H); // updates belief
  ?b(X);  // new test goal
  X > 10; // constraint to carry on
  close(door). // external action
```

**Plans – Example**

```
+green_patch(Rock) [source(percept)]
  : not battery_charge(low)
  <- ?location(Rock,Coordinates);
  !at(Coordinates);
  !examine(Rock).
+!at(Coords)
  : not at(Coords) & safe_path(Coords)
  <- move_towards(Coords);
  !at(Coords).
+!at(Coords)
  : not at(Coords) & not safe_path(Coords)
  <- ...+
  !at(Coords) : at(Coords).
```
The plans that form the plan library of the agent comes from:
- initial plans defined by the programmer
- plans added dynamically and intentionally by
  - .add_plan
  - .remove_plan
- plans received from messages of type:
  - tellHow
  - untellHow
- messages

perceives the environment and update belief base
- processes new messages
- selects event
- selects relevant plans
- selects applicable plans
- creates/updates intention
- selects intention to execute

Consider a very simple robot with two goals:
- when a piece of gold is seen, go to it
- when battery is low, charge

Example (Java code – go to gold)
```java
public class Robot extends Thread {
    boolean seeGold, lowBattery;
    public void run() {
        while (true) {
            while (! seeGold) {
                while (true) {
                    a = selectDirection();
                    doAction(go(a));
                }
            }
        }
    }
}
```
Example (Java code – charge battery)

```java
public class Robot extends Thread {
    boolean seeGold, lowBattery;
    public void run() {
        while (true) {
            while (!seeGold)
                if (lowBattery) charge();
            while (seeGold) {
                a = selectDirection ();
                if (lowBattery) charge();
                doAction(go(a));
                if (lowBattery) charge();
            }
        }
    }
}
```

(note where the test for low battery have to be done)

Example (Jason code)

```prolog
+see(gold)
  <- !goto(gold).
+!goto(gold) : see(gold) // long term goal
  <- !select_direction(A);
  go(A);
  !goto(gold).
+battery(low) // reactivity
  <- .suspend(goto(gold));
  !charge;
  .resume(goto(gold)).
```

With the Jason extensions, nice separation of theoretical and practical reasoning

- BDI architecture allows
  - long-term goals (goal-based behaviour)
  - reacting to changes in a dynamic environment
  - handling multiple foci of attention (concurrency)

- Acting on an environment and a higher-level conception of a distributed system
Negation

Negation as failure

- **not**: formula is true if the interpreter fails to derive it
- **Closed world assumption**: 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.

Strong negation

- **~**: used to express that an agent *explicitly* believes something to be false.

Example

```prolog
+!leave(home) : ~raining
<-> open(curtains); ...

+!leave(home) : not raining & not ~raining
<-> .send(mum,askOne,raining,Answer,3000); ...
```

Prolog-like Rules in the Belief Base

Rules

Rules can be used to simplify certain tasks, i.e. making certain conditions used in plans more succinct. Their syntax is similar to the one used for plans.

Example

```prolog
likely_color(Obj,C) :-
  colour(Obj,C)[degOfCert(D1)] &
  not (colour(Obj,_) [degOfCert(D2)] & D2 > D1) &
  not ~colour(Obj,B).
```

Plan Annotations

- Like beliefs, plans can also have annotations, which go in the plan label
- Annotations contain meta-level information for the plan, which selection functions can take into consideration
- The annotations in an intended plan instance can be changed dynamically (e.g. to change intention priorities)
- There are some pre-defined plan annotations, e.g. to force a breakpoint at that plan or to make the whole plan execute atomically

Example (an annotated plan)

```prolog
@myPlan[chance_of_success(0.3), usual_payoff(0.9), any_other_property]
+!g(X) : c(t) <- a(X).
```
Example (an agent blindly committed to $g$)

+!g : g.

+!g : ... <- ... ?g.

-!g : true <- !g.

Internal Actions

- Unlike actions, internal actions do not change the environment
- Code to be executed as part of the agent reasoning cycle
- AgentSpeak is meant as a high-level language for the agent's practical reasoning and internal actions can be used for invoking legacy code elegantly
- Internal actions can be defined by the user in Java

Standard Internal Actions

- Standard (pre-defined) internal actions have an empty library name
  - .print(term1, term2,...)
  - .union(list1, list2, list3)
  - .my_name(var)
  - .send(ag, perf, literal)
  - .intend(literal)
  - .drop_intention(literal)

- Many others available for: printing, sorting, list/string operations, manipulating the beliefs/annotations/plan library, creating agents, waiting/generating events, etc.
Jason Customisations

- **Agent class customisation:**
  selectMessage, selectEvent, selectOption, selectIntention, buf, brf, ...

- **Agent architecture customisation:**
  perceive, act, sendMsg, checkMail, ...

- **Belief base customisation:**
  add, remove, contains, ...
  - Example: persistent belief base
    (in text files, in databases, ...)

---

**Execution & Communication Platform**

Different execution and communication platforms can be used with Jason:

**Centralised:** all agents in the same machine, one thread by agent, very fast

**Centralised (pool):** all agents in the same machine, fixed number of threads, allows thousands of agents

**Jade:** distributed agents, FIPA-ACL

**Saci:** distributed agents, KQML

... others defined by the user (e.g. AgentScape)

---

**Definition of a Simulated Environment**

- There will normally be an environment where the agents are situated
- The agent architecture needs to be customised to get perceptions and to act on such environment
- We often want a simulated environment (e.g. to test a MAS application)
- This is done in Java by extending Jason's Environment class
Interaction with the Environment Simulator

Example of an Environment Class

```java
1 import jason.*;
2 import ...;
3 public class robotEnv extends Environment {
4     ....
5     public robotEnv() {
6         Literal gp =
7             Literal.parseLiteral("green_patch(souffle)");
8         addPercept(gp);
9     }
10
11     public boolean executeAction(String ag, Structure action)
12         if (action.equals(...)) {
13             addPercept(ag,
14                 Literal.parseLiteral("location(souffle,c(3,4))"))
15         }
16         ...
17     return true;
18 }
```

MAS Configuration Language I

- Simple way of defining a multi-agent system

Example (MAS that uses JADE as infrastructure)

```plaintext
MAS my_system {
    infrastructure: Jade
    environment: robotEnv
    agents:
        c3po;
        r2d2 at jason.sourceforge.net;
        bob #10; // 10 instances of bob
    classpath: ".../lib/graph.jar";
}
```

MAS Configuration Language II

- Configuration of event handling, frequency of perception, user-defined settings, customisations, etc.

Example (MAS with customised agent)

```plaintext
MAS custom {
    agents: bob [verbose=2,paramters="sys.properties"]
        agentClass MyAg
        agentArchClass MyAgArch
        beliefBaseClass jason.bb.JDBCPersistentBB{
            "org.hsqldb.jdbcDriver",
            "jdbc:hsqldb:bookstore",
            ...
)
```
 Origins and Fundamentals

 Features

 Use of Jason within a Multi-Agent System

 Current Shortfalls and Future Trends

 Perspectives: Some Past and Future Projects

 Summary
Some Related Projects I

- **Speech-act** based communication
  Joint work with Renata Vieira, Álvaro Moreira, and Mike Wooldridge
- **Cooperative** plan exchange
  Joint work with Viviana Mascardi, Davide Ancona
- **Plan Patterns** for Declarative Goals
  Joint work with M.Wooldridge
- **Planning** (Felipe Meneguzzi and Colleagues)
- **Web and Mobile Applications** (Alessandro Ricci and Colleagues)
- **Belief Revision**
  Joint work with Natasha Alechina, Brian Logan, Mark Jago

Some Related Projects II

- **Ontological** Reasoning
  - Joint work with Renata Vieira, Álvaro Moreira
  - **JASDL**: joint work with Tom Klapiscak
  - Goal-Plan Tree Problem (Thangarajah et al.)
    Joint work with Tricia Shaw
  - Trust reasoning (ForTrust project)
  - Agent verification and model checking
    Joint project with M.Fisher, M.Wooldridge, W.Visser, L.Dennis, B.Farwer

Some Related Projects III

- **Environments, Organisation and Norms**
  - Normative environments
    Joint work with A.C.Rocha Costa and F.Okuyama
  - MADeM integration (Francisco Grimaldo Moreno)
  - Normative integration (Felipe Meneguzzi)
  - **CArtAgO** integration
  - .../OSE integration

- More on [jason.sourceforge.net, related projects](#)

Some Trends for Jason I

- **Modularity and encapsulation**
  - Capabilities (JACK, Jadex, ...)
  - Roles (Dastani et al.)
  - Mini-agents (?)

- Recently done: **meta-events**

- To appear soon: annotations for **declarative goals**, improvement in plan failure handling, etc.

- **Debugging** is hard, despite mind inspector, etc.

- Further work on combining with environments and organisations
**Summary**

- **AgentSpeak**
  - Logic + BDI
  - Agent programming
- **Jason**
  - AgentSpeak interpreter
  - implements the operational semantics of AgentSpeak
  - speech act based
  - highly customisable
  - useful tools
  - open source
  - open issues

**More information**

- [http://jason.sourceforge.net](http://jason.sourceforge.net)

**Bibliography**