Programming Multi-Agent Systems in AgentSpeak Using *Jason*

Published by John Wiley & Sons Ltd., October 2007.

http://jason.sf.net/jBook/

AgentSpeak

- Originally proposed by Rao [MAAMAW 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

Syntax of AgentSpeak

- The main language constructs of AgentSpeak are:
  - Beliefs
  - Goals
  - Plans
- The architecture of an AgentSpeak agent has five main components:
  - Belief Base
  - Plan Library
  - Set of Events
  - Set of Intentions

Syntax of AgentSpeak (Beliefs and Goals)

- Beliefs represent the information available to an agent (e.g., about the environment or other agents)

```
publisher(wiley)
```

- Goals represent states of affairs the agent wants to bring about (come to believe, when goals are used declaratively)
  - Achievement goals:

```
!write(book)
```

Or attempts to retrieve information from the belief base

  - Test goals:

```
?publisher(P)
```

Syntax of AgentSpeak (Events and Plans)

- An agent reacts to events by executing plans
- Events happen as a consequence to changes in the agent’s beliefs or goals
- Plans are recipes for action, representing the agent’s know-how
- 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;
  - the context represent 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 true at the time a plan is being chosen to handle the event.

Syntax of AgentSpeak (Plans Cont.)

- AgentSpeak triggering events:
    - `+b` (belief addition)
    - `-b` (belief deletion)
    - `+!g` (achievement-goal addition)
    - `-!g` (achievement-goal deletion)
    - `+?g` (test-goal addition)
    - `-?g` (test-goal deletion)
- The context is logical expression, typically a conjunction of literals to be checked whether they follow from the current state of the belief base
- The body is a sequence of actions and (sub) goals to achieve.
- NB: This is the original AgentSpeak syntax; ***Jason*** allows other things in the context and body of plans.

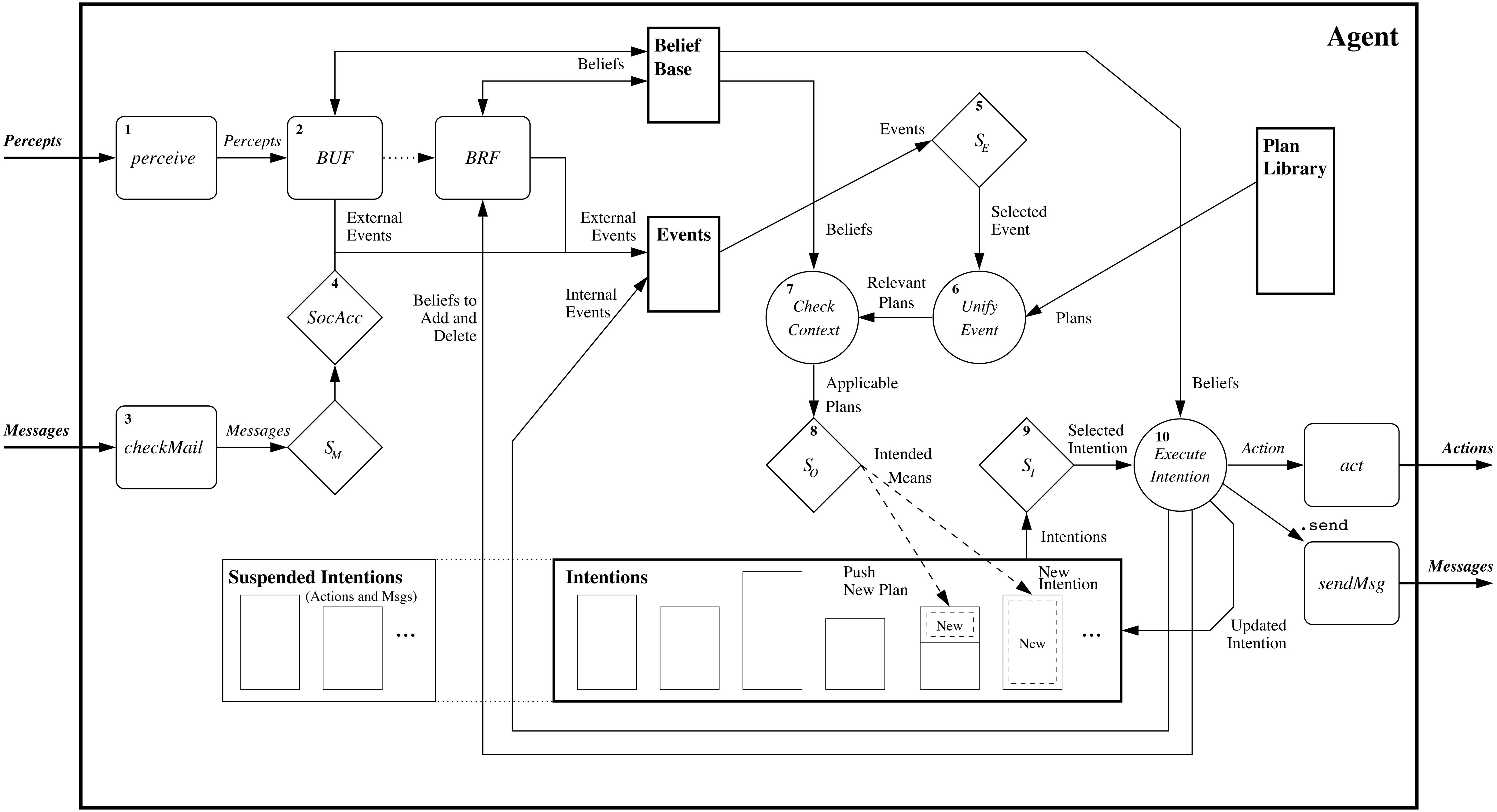
AgentSpeak Plans

```
+green_patch(Rock)
  :  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) ...
```

Jason

- ***Jason*** implements the operational semantics of a variant of AgentSpeak
- Various extensions aimed at a more practical programming language
- Platform for developing multi-agent systems
- Developed by Jomi F. Hübner and Rafael H. Bordini
- We'll look at the ***Jason*** additions to AgentSpeak and its features

Jason Reasoning Cycle



Reasoning Cycle (Steps)

1. Perceiving the Environment
2. Updating the Belief Base
3. Receiving Communication from Other Agents
4. Selecting ‘Socially Acceptable’ Messages
5. Selecting an Event

Reasoning Cycle (Steps)

6. Retrieving all Relevant Plans

7. Determining the Applicable Plans

8. Selecting one Applicable Plan

9. Selecting an Intention for Further Execution

10. Executing one step of an Intention

10. IntentionExecution

a. Environment actions

b. Achievement goals

c. Test goals

d. Mental notes

e. Internal actions

f. Expressions

Belief Annotations

- Annotated predicate:

  $\texttt{ps}(t_1, ..., t_n)[a_1, ..., a_m]$

  where a_i are first order terms

- All predicates in the belief base have a special annotation $\texttt{source}(s_i)$ where $s_i \in \{\texttt{self}, \texttt{percept}\} \cup \texttt{AgId}$

Example of Annotations

- An agent’s belief base with a user-defined `doc` annotation (degree of certainty)

```
blue(box1)[source(ag1)].
red(box1)[source(percept)].
colourblind(ag1)[source(self),doc(0.7)].
lier(ag1)[source(self),doc(0.2)].
```

Plan Annotations

- Plan labels also can have annotations (e.g., to specify meta-leval information)
- Selection functions (Java) can use such information in plan/intention selection
- Possible to change those annotations dynamically (e.g., to update priorities)
- Annotations go in the plan label

Annotated Plan Example

```
@aPlan[
   chance_of_success(0.3),
   usual_payoff(0.9),
   any_other_property]
+!g(X)
   :  c(t)
   <- a(X).
```

Strong Negation

- The operator ‘~’ is used for strong negation

```
+!leave(home)
   :  not raining & not ~raining
   <- open(curtains); ...

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

Belief-Base Rules

- Prolog-like rules in the belief base

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

Handling Plan Failure

- Goal-deletion events were syntactically defined, but no semantics

- We use them for a plan failure handling mechanism (probably not what they were meant for)

- Handling plan failures is very important as agents are situated in dynamic environments

- A form of “contingency plan”, possibly to “clean up” before attempting another plan

Contingency Plan Example

- To create an agent that is blindly committed to goal g:

```
+!g : g     <- true.
+!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

- Internal actions can be used for invoking legacy code elegantly

Internal Actions (Cont.)

- Libraries of user-defined interal actions

```
lib_name.action_name(...)
```

- Predefined internal actions have an empty library name

- Internal action for communication

`.send(r,`*`ilf`*`,pc)` where *`ilf`* ∈
`{tell,untell,achieve,unachieve,`
` askOne,askAll,askHow,`
` tellHow,untellHow}`

Internal Actions (Cont.)

- Examples of BDI-related internal actions:

```
.desire(literal)
.intend(literal)
.drop_desires(literal)
.drop_intentions(literal)
```

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

A ***Jason*** Plan

```
+green_patch(Rock)

  :  ~battery_charge(low)
     & .desire(at(_))

  <- .drop_desires(at(_));
     dip.get_coords(Rock, Coords);
     !at(Coords);
     !examine(Rock).
```

AgentSpeak X Prolog

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

- BDI arcthicture 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

- Direct integration with Java

MAS Configuration File

- Simple way of defining a multi-agent system

```
MAS my_system {
   infrastructure: Jade
   environment: MyEnv
   ExecuctionControl: ...
   agents: ag1; ag2; ag3;
}
```

MAS Definition (Cont.)

- Infrastructure options: `Centralised`, `Saci`, `Jade`
- Easy to define the host where agents and the environment will run
- If the file name with the code is unusual

```
agents:
   ag1 at host1.dur.ac.uk;
```

```
agents: ag1 file1;
```

MAS Definition (Cont.)

- Multiple instances of an agent

```
agents: ag1 #10;
```

- Interpreter configuration

```
agents: ag1 [conf=option];
```

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

Agent Customisation

- Users can customise the Agent class to define the selection functions, social relations for communication, and belief update and revision
    - selectMessage()
    - selectEvent()
    - selectOption()
    - selectIntention()
    - socAcc()
    - buf()
    - brf()

Overall Agent Architecture

- Users customise the AgentArch class to change the way the agent interacts with the infrastrcuture: perception, action, and communication

- Helps switching between simulation for testing and real deployment

    - perceive()

    - act()

    - sendMsg()

    - broadcast()

    - checkMail()

Belief Base Customisation

- Logical belief base might not be appropriate for large applications
- Jason has an alternative belief base combined with a database
- Users can create other customisations
    - add()
    - remove()
    - contains()
    - getRelevant()

Customised MAS

```
MAS Custom {

 agents:

  a1 agentClass MyAg

     agentArchClass MyAgArch

     beliefBaseClass Jason.bb.JDBCPersistentBB(
     "org.hsqldb.jdbcDriver",
     "jdbc:hsqldb:bookstore",
     ...
     "[count_exec(1,tablece)]");
}
```

Environments

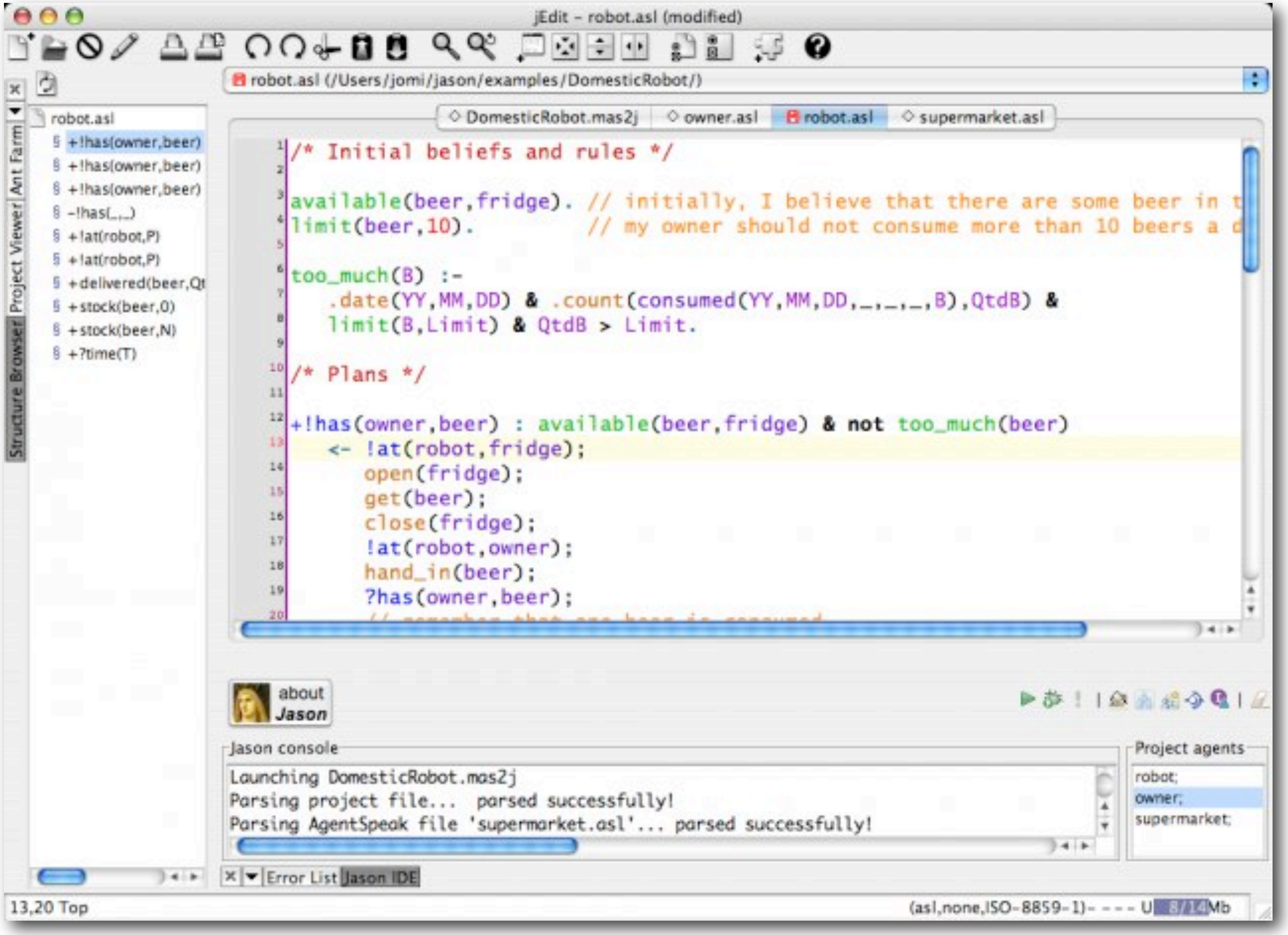
- In actual deployment, there will normally be an environment where the agents are situated

- As discussed earlier, the AgentArchitecture needs to be customised to get perceptions and 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 and using methods such as addPercept(String Agent, Literal Percept)

Jason for jEdit



```
+!has(owner,beer)
+!has(owner,beer)
+!has(owner,beer)
-!has(_,_)
+!at(robot,P)
+!at(robot,P)
+delivered(beer,Qt
+stock(beer,0)
+stock(beer,N)
+?time(T)
```

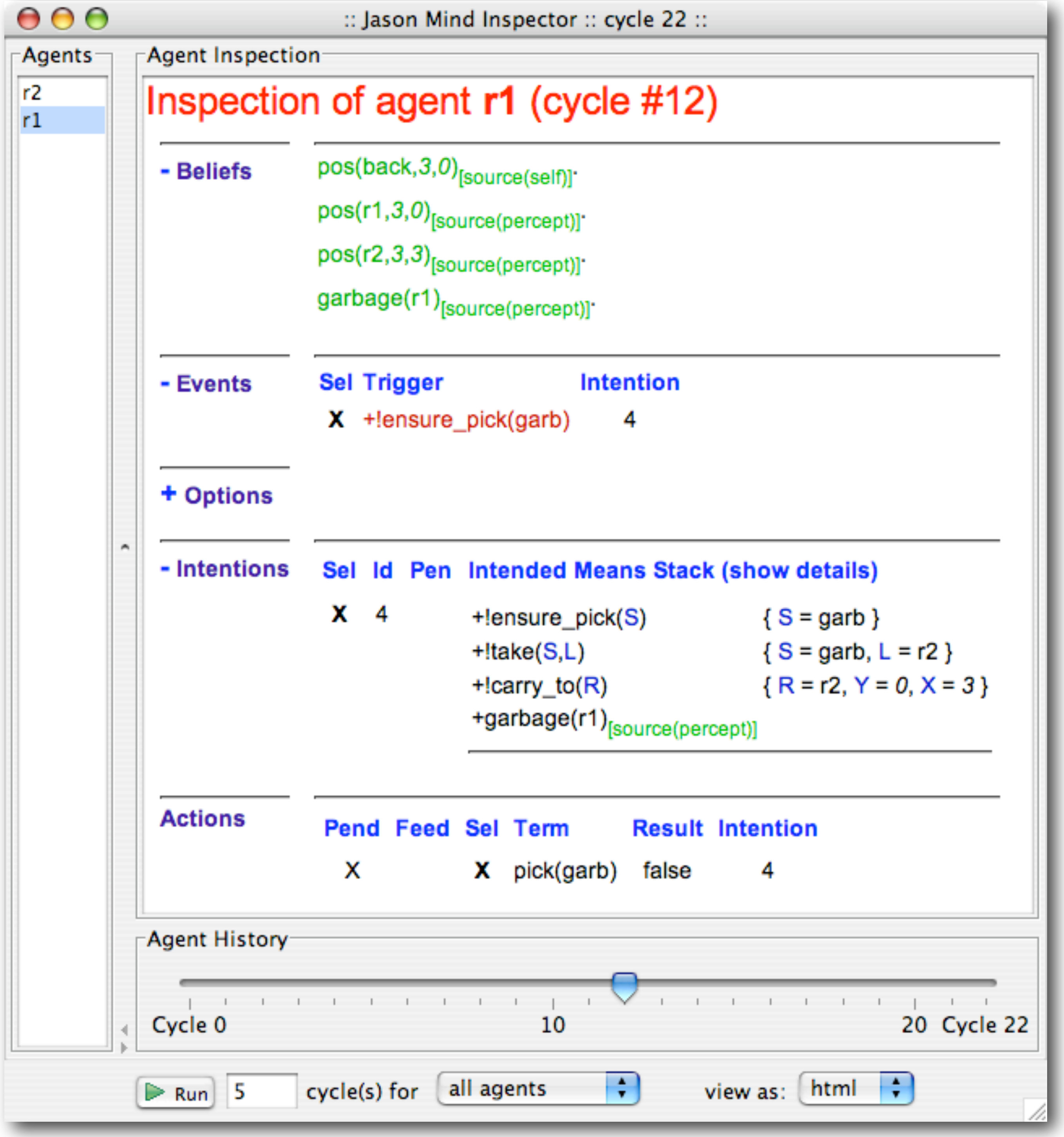
```
/* Initial beliefs and rules */

available(beer,fridge). // initially, I believe that there are some beer in t
limit(beer,10).         // my owner should not consume more than 10 beers a d

too_much(B) :-
   .date(YY,MM,DD) & .count(consumed(YY,MM,DD,_,_,_,B),QtdB) &
   limit(B,Limit) & QtdB > Limit.

/* Plans */

+!has(owner,beer) : available(beer,fridge) & not too_much(beer)
   <- !at(robot,fridge);
      open(fridge);
      get(beer);
      close(fridge);
      !at(robot,owner);
      hand_in(beer);
      ?has(owner,beer);
```

```
Launching DomesticRobot.mas2j
Parsing project file...  parsed successfully!
Parsing AgentSpeak file 'supermarket.asl'... parsed successfully!
```

```
robot;
owner;
supermarket;
```

```
13,20 Top                                        (asl,none,ISO-8859-1)- - - - U 8/14Mb
```

Jason's Mind Inspector



:: Jason Mind Inspector :: cycle 22 ::

Agents

r2
r1

Agent Inspection

Inspection of agent **r1** (cycle #12)

- Beliefs

pos(back,*3*,*0*)$_{[source(self)]}$.
pos(r1,*3*,*0*)$_{[source(percept)]}$.
pos(r2,*3*,*3*)$_{[source(percept)]}$.
garbage(r1)$_{[source(percept)]}$.

- Events

Sel	Trigger	Intention
X	+!ensure_pick(garb)	4

+ Options

- Intentions

Sel	Id	Pen	Intended Means Stack (show details)	

X	4		+!ensure_pick(S)
			+!take(S,L)
			+!carry_to(R)
			+garbage(r1)$_{[source(percept)]}$
				{ S = garb }
				{ S = garb, L = r2 }
				{ R = r2, Y = *0*, X = *3* }

Actions

Pend	Feed	Sel	Term	Result	Intention
X		**X**	pick(garb)	false	4

Agent History

5 cycle(s) for all agents

view as: html

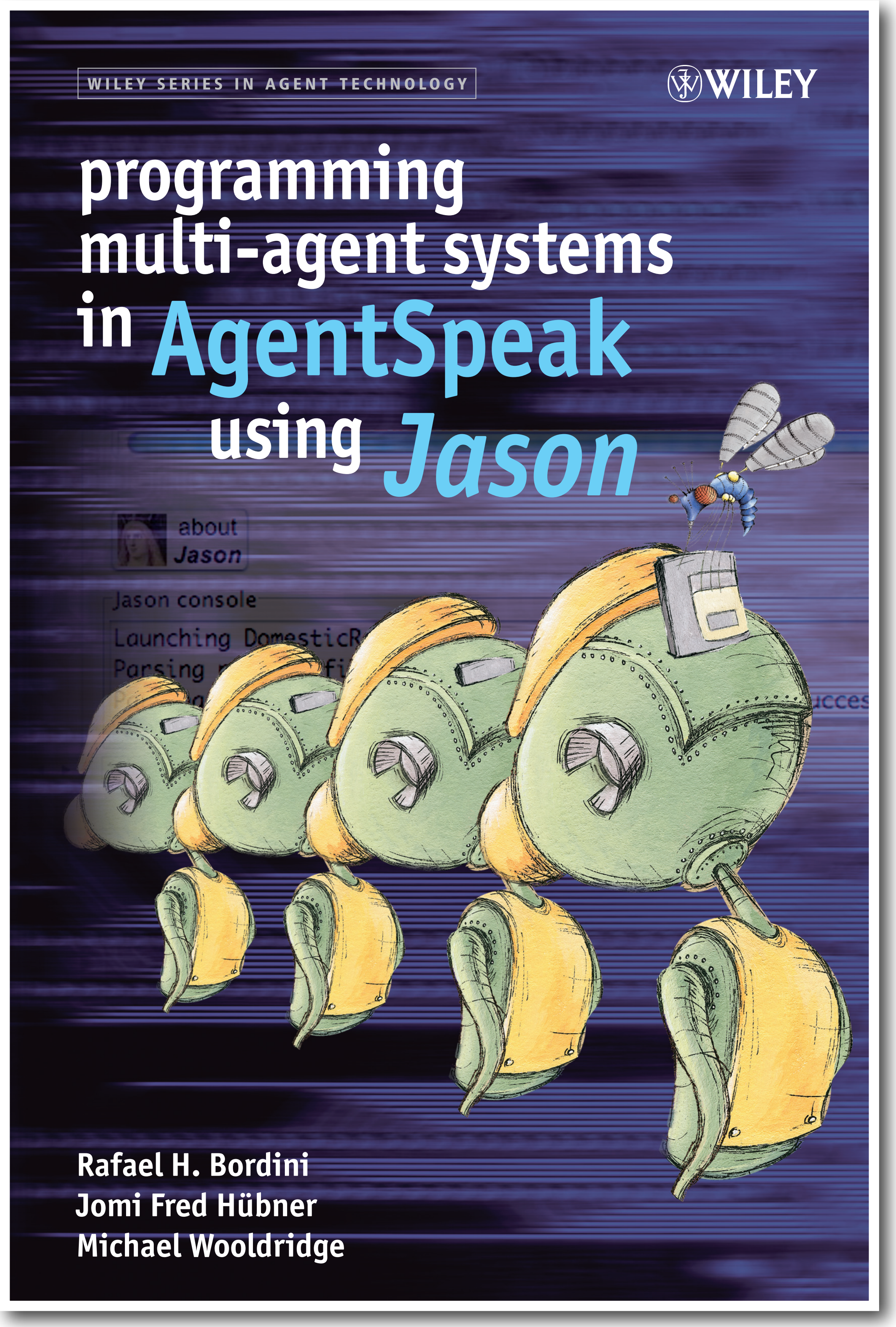
Jason is available
Open Source
under GNU LGPL at:

http://jason.sf.net

(kindly hosted by
SourceForge)

Jason
by *Gustave Moreau* (1865)

Oil on canvas, 204 x 115.5 cm.
Musée d'Orsay, Paris.
© Photo RMN. Photograph by
Hervé Lewandowski.



WILEY SERIES IN AGENT TECHNOLOGY

WILEY

programming multi-agent systems in AgentSpeak using *Jason*

Rafael H. Bordini
Jomi Fred Hübner
Michael Wooldridge