Outline

1. Jason & AgentSpeak Language
   - Goals
   - Events
   - Plans
   - Triggering events
   - Context
   - Body

2. The Jason Interpreter
   - The Jason reasoning cycle
   - Failure management

3. Features
   - Strong negation
   - Rules
   - Failure Handling
   - Internal actions
   - Customisations
AgentSpeak [Rao, 1996] is an agent programming language whose semantics implements the functionalities of the Procedural Reasoning System.

- Based on the BDI architecture (as is the PRS).
- AgentSpeak agents are reactive (in that they do not terminate) planning agents.
- The Jason agent programming system [Bordini et al., 2007] uses an extension of the AgentSpeak language.
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

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

Intentions: plans instantiated to achieve some goal
Beliefs representation

**Syntax**

Beliefs are represented by annotated literals of first order logic

\[ \text{functor}(term_1, \ldots, term_n)[annot_1, \ldots, annot_m] \]

**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 describe what the agent knows about its environment.
In their simplest form, logic programming facts.

Property
\texttt{tall(john)}.

Relationship
\texttt{likes(john,music)}.

These atoms are meant to express the agent’s beliefs, not the absolute truth!
The interpreter can be programmed to use annotations.

Main use: remembering a belief’s source.
During the agent’s execution, beliefs can be updated with information coming from the following sources:

- **perception**: resulting from a continual sensing of the environment;
- **communication**: resulting from another agent’s message;
- **mental notes**: added by the agent to its own belief base during the execution of a plan (analogous to *asserts* in Prolog).
Changes in the belief base 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)]
by communication

when an agent receives a *tell* message, the content is a new belief annotated with the sender of the message

```
.send(tom,tell,lier(alice)); // sent by bob
// adds lier(alice)[source(bob)] in Tom’s BB
...
.send(tom,untell,lier(alice)); // sent by bob
// removes lier(alice)[source(bob)] from Tom’s BB
```
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.*

*not* operator (negation as failure): is true if the interpreter fails to derive its argument.

```plaintext
not likes(john, music).
```
Negation as failure is simplistic in open environments.

We cannot realistically assume that everything that we don’t know positively is false!

Better solution is to distinguish:
- what the agent believes to be true,
- what the agent believes to be false, and
- what the agent believes nothing about.

〜 operator: true if the agent has an explicit belief that its argument is false.
Belief types

colour(box1,blue)[source(bob)].
colour(box1,white)[source(john)].
colour(box1,red)[source(percept)].
colourblind(bob)[source(self),degOfCert(0.7)].
lier(bob)[source(self),degOfCert(0.2)].
Rules

- Used for theoretical reasoning: deriving knowledge from existing knowledge.
- Analogous to Prolog clauses (slightly different syntax).

**Rule syntax**

<head> :- <body>.

- head is one atom;
- body can contain disjunctions (symbol: |) and conjunctions (symbol: &).
Example

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).
Goals

Types
- 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).
by intention

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

... 

// adds new achievement goal !write(book)[source(self)]
!write(book);

// adds new test goal ?publisher(P)[source(self)]
?publisher(P);

...
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, write(book)); // sent by Bob
// adds new goal write(book)[source(bob)] for Tom
...
.send(tom, unachieve, write(book)); // sent by Bob
// removes goal write(book)[source(bob)] for Tom
```
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, published(P), Answer); // sent by Bob
// adds new goal ?publisher(P)[source(bob)] for Tom
// the response of Tom will unify with Answer
```
Events

- Events happen as a consequence to changes in the agent’s beliefs or goals

- Types of events
  - $+b$ (belief addition)
  - $-b$ (belief deletion)
  - $+!g$ (achievement-goal addition)
  - $-!g$ (achievement-goal deletion)
  - $+?g$ (test-goal addition)
  - $?g$ (test-goal deletion)

- An agent reacts to events by executing plans
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
  - `tellHow` messages
  - `untellHow`
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
For an agent it is necessary to balance reactivity and proactiveness.

Goals support proactiveness.

For reactivity, the agent needs to be aware of the changes that occur in the agent itself and in the environment.

Events are generated in case of changes (addition, deletion) in
- the agent’s beliefs;
- the agent’s goals.
A plan’s triggering part defines in which circumstances a plan should be considered.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>+\l</td>
<td>Belief addition</td>
</tr>
<tr>
<td>-\l</td>
<td>Belief deletion</td>
</tr>
<tr>
<td>+!\l</td>
<td>Achievement goal addition</td>
</tr>
<tr>
<td>-!\l</td>
<td>Achievement goal deletion</td>
</tr>
<tr>
<td>+?\l</td>
<td>Test goal addition</td>
</tr>
<tr>
<td>-?\l</td>
<td>Test goal deletion</td>
</tr>
</tbody>
</table>

**Table:** Notation for types of triggering events
A plan’s context defines in what circumstance a plan is *applicable*.

Atoms and relational expressions combined with the operators:

- **not** (negation as failure)
- **&** (conjunction)
- **|** (disjunction)
## Types of literals in a plan context

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>/</td>
<td>The agent believes / is true</td>
</tr>
<tr>
<td>~/</td>
<td>The agent believes / is false</td>
</tr>
<tr>
<td>not /</td>
<td>The agent does not believe / is true</td>
</tr>
<tr>
<td>not ~/</td>
<td>The agent does not believe / is false</td>
</tr>
</tbody>
</table>

**Table:** A plan context is typically a conjunction of these types of literals.
Examples

Trigger and context

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

Trigger and context with negation

+!buy(Something)
  : not ~legal(Something) & price(Something, P)
  & bank_balance(B) & B > P
  <- ... .
Internal actions

- Executed internally rather than managed by the interpreter.
- E.g., to execute Java code on partial results.
- If they have side effects they are not always appropriate in the context (which should be used to check if a plan is applicable).
Rules in the belief base can be used to make plan contexts more compact.

**Context with defined atoms**

If the belief base contains a rule

```
can_afford(Something)
    :- price(Something,P) & bank_balance(B) & B > P.
```

we can write

```
+!buy(Something)
    : not ~legal(Something) & can_afford(Something)
<- ... .
```
A plan’s body

- Represents the course of action to be executed.
- Can contain six types of formulæ:
  - actions
  - achievement goals
  - test goals
  - mental notes
  - internal actions
  - expressions
- Formulae are separated by semicolons (;).
The actions that the agent can perform are known beforehand and represented symbolically.

In a plan’s body, a *ground predicate* represents an action.

**Actions**

- right
- left
- rotate(left_arm, 45)
- rotate_right_arm(90)

Internal actions: run internally within the agent, rather than change the environment.
Achievement goals

- Complex plans require more than simple actions to be executed.
- It is sometimes necessary to achieve intermediate goals for the plan to be completed.
- Therefore a plan’s body can also contain achievement goals.
- Two categories w.r.t. suspension:

**Achievement goals in plan’s body**

- with !at(home);call(john), the action call(john) is executed only after the agent arrived at home
- with !!at(home);call(john), the action call(john) is executed as soon as the agent’s goal to be at home is set
Test goals

- Used to retrieve information during plan execution.
- In a plan’s body, test goals are identified by the `?` operator.

**Test goal in plan’s body**

`?coords(Target,X,Y)` lets the agent know the current coordinates of a moving target.

**Difference with test goals in context:**
- If a test goal fails in the plan’s context, the plan is not executed;
- If a test goal fails in the plan’s body, the plan fails.
Mental notes

- Beliefs added (+ operator) to the belief base during the execution of a plan.
- Have a source(self) annotation.

Mental note

+ moved(lawn, Today)
(Today must have been instantiated earlier)

Beliefs can also be removed (- operator).

Belief removal

- current_targets(_)

- + some_predicate(arg) is an abbreviation for
  - some_predicate(_); + some_predicate(arg)
Internal actions

- Executed internally, rather than to change the environment.
- Identified by the presence of a . (also used to separate library name from action name) in their name.

Internal action

The agent might be using a path-finding library pf, which makes available an internal action get_path to find the shortest path from its current position to a point with coordinates X and Y:

```
 pf.get_path(X,Y,P)
```

Standard internal action, with empty library name. E.g.,

- .send
- .print
Most Prolog built-in expressions are available.

- \( T1 == T2 \) iff \( T1 \) and \( T2 \) are identical terms
- \( T1 \neq T2 \) iff \( T1 \) and \( T2 \) are not identical terms
- \( T1 = T2 \) (tries to) unify \( T1 \) and \( T2 \)
- =.. behaves slightly differently to support annotations:
  \[ p(b,c)[a1,a2] =.. [p, [b, c], [a1, a2]] \]
Plan examples

Plan with test goal

+!leave(home)
  : not raining and not ~raining
<- !location(window);
  ?curtain_type(Curtains);
  open(Curtains);
  ...

Plan with internal action

+!leave(home)
  : not raining and not ~raining
<- .send(mum,askIf,raining);
  ...

AgentSpeak in Jason
Plan labels

- Labels allow plans to be referenced
- Identified by the @ operator
- Labels can be simple constants or terms, including annotations.

Plan label

@shopping(1) [chance_of_success(0.7), source(ag1), usual_payoff(0.9), expires(autumn)]
+need(Something)
  : can_afford(Something)
  <- !buy(Something)
Plans
Operators for plan’s context

Boolean operators

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

Arithmetic operators

+  (sum)
-  (subtraction)
*  (multiply)
/  (divide)
div (divide – integer)
mod (remainder)
** (power)
A plan’s body may contain:

- Goal operators (!, ?, !!)
- Belief operators (+, -, -+)
- Actions and Constraints

Example (plan’s body)

```
+beer : time_to_leave(T) & clock.now(H) & H >= T

<- !g1; // new sub-goal
  !!g2; // new goal
  +b1(T-H); // add new belief
  -+b2(T*H); // update belief
  ?b(X); // new test goal
  X > 10; // constraint to carry on
  close(door). // external action
```
+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).
Figure: The Jason reasoning cycle.
Symbols

**Rectangles**: components of the *agent state*:
- belief base
- set of events
- plan library
- set of intentions

**Rounded boxes, diamonds, circles**: functions used in the reasoning cycle.
The initial beliefs initialize the belief base.
This causes belief additions, which are added to the set of events.
So are initial goals.
Plans form the initial plan library.
The set of intentions is empty.
Step 1: Perceiving the environment

- Necessary to update beliefs according to the current state of the environment.
- Requires a component that can perceive the environment in symbolic form as a list of literals.
- Each literal is a percept.
- Implemented by the perceive method.
- The environment can be simulated by a Java class.
Beliefs are updated by a belief update function.

Implemented by the (customizable) `buf` method.

If \( p \) is the set of current percepts and \( b \) is the set of literals in the belief base obtained by the previous perception:

1. Each literal \( l \) in \( p \) not currently in \( b \) is added to \( b \);
2. Each literal \( l \) in \( b \) no longer in \( p \) is deleted from \( b \).

Each change generates an external event.
Step 3: Receiving communication from other agents

- The agents checks for messages it may have received in its “mailbox”.
- Implemented by the `checkMail` method.
- Only one message is processed at a reasoning cycle.
- Such message is returned, among all received and not yet processed, by the `message selection function`.
- Implemented by methods that can be overridden.
Step 4: Selecting “socially acceptable” messages

- The agent selects the messages that can be accepted according to some “social rules”.
- E.g., which agents can provide information, or delegate goals.
- Acceptable messages are selected by a social acceptance function.
- Implemented by the customizable SocAcc method.
Step 5: Selecting an event

- Agents operate handling events.
- One event is handled at each cycle.
- Selected by a customizable event selection function, which can define priorities among events.
- By default, FIFO policy.
Step 6: Retrieving all relevant plans

- The *relevant* plans (i.e., those that can handle the event) are selected.
- The selected plans are those whose triggering event can be *unified* with the selected events.
- Unification as in Logic Programming.
Step 6: Retrieving all relevant plans

Example

Relevant plans

Out of the following plans

@p1  +position(Object,Coords) :...<-... .
@p2  +colour(Object,Colour) :...<-... .
@p3  +colour(Object,Colour) :...<-... .
@p4  +colour(Object,red) :...<-... .
@p5  +colour(Object,Colour)[source(self)] :...<-... .
@p6  +colour(Object,blue)[source(percept)] :...<-... .

the ones that are relevant for the event
⟨+colour(box1,blue)[source(percept)], ⊤⟩
are p2, p3 and p6.
Out of all the relevant plans, only some are applicable.
Applicability depends on the plan’s context.
Step 7: Determining the applicable plans

Example

<table>
<thead>
<tr>
<th>Plans</th>
</tr>
</thead>
<tbody>
<tr>
<td>@p2  +colour(Object,Colour) : shape(Object,box) &amp; not pos (Object,coord(0,0)) &lt;- ... .</td>
</tr>
<tr>
<td>@p3  +colour(Object,Colour) : colour(OtherObj,red)[source(S)] &amp; S==percept &amp; shape(OtherObj,Shape) &amp; shape(Object,Shape) &lt;- ... .</td>
</tr>
<tr>
<td>@p6  +colour(Object,blue)[source(percept)] : colour(OtherObj,red)[source(percept)] &amp; shape(OtherObj,sphere) &lt;- ... .</td>
</tr>
</tbody>
</table>
Step 7: Determining the applicable plans

Example

Current belief base

\begin{align*}
\text{shape(box1,box)} & \text{[source(percept)]}. \\
\text{pos(box1,coord(9,9))} & \text{[source(percept)]}. \\
\text{colour(box1,blue)} & \text{[source(percept)]}. \\
\text{shape(sphere2,sphere)} & \text{[source(percept)]}. \\
\text{pos(sphere2,coord(7,7))} & \text{[source(bob)]}. \\
\text{colour(sphere2,red)} & \text{[source(percept),source(john)]}. \\
\end{align*}

p2 and p6 are applicable.
Step 8: Selecting one applicable plan

- Out of the applicable plans, the agent chooses one to commit to.
- The course of action corresponding to chosen plan becomes an *intended means*.
- The intended means is added to the agent’s *set of intentions*.
- The choice is defined by a customizable *option selection function* (or *applicable plan selection function*).
- Intentions are implemented as stacks.
- External events generate new intentions, internal events push the new intended means onto the intention that generated the event.
Step 9: Selecting an intention for further execution

- At each cycle, there will be, in general, more than one intended means to choose from.
- At most one can be executed.
- This is selected by a customizable intention selection function.
- Default policy: round-robin.
Step 10: Executing one step of an intention

Environment actions

- The action is performed by the agent’s *effectors*.
- The effector will return a feedback to the interpreter about the action execution.
- Action execution can take time: the interpreter *suspends* the intention, because the plan’s body must be executed sequentially.
- Meanwhile, it can use its time for other tasks.
Step 10: Executing one step of an intention

Achievement goals

- The plan execution cannot continue until the goal is achieved.
- A goal addition event is generated.
- The achievement goal is not removed from the intention’s body until it is actually achieved, so as to exploit variable unification during goal achievement.
Step 10: Executing one step of an intention

Test goals

- Used to check if a property is believed by the agent, or to retrieve information.
- If the goal succeeds, it can be removed.
- Variable instantiations will stay.
If the formula is a belief to be added or removed, the interpreter passes the request to the brf method.

The corresponding event is generated (which requires handling by the interpreter).

By default, the interpreter adds a source(self) annotation to the event.
Step 10: Executing one step of an intention
Internal actions

- Consist of Java/legacy code to be executed.
- Must be encoded in a boolean method, so as to provide feedback about the action’s success.
- Can cause variable instantiation.
Expressions are simply executed.
Failure causes the whole plan’s failure.
Unless the information to be checked is only available during plan execution, expressions are better placed in the context.
Final stage before restarting the cycle

- Suspended intentions, waiting for feedback, are placed back in the intention set if the feedback has become available.
- Finished plans are removed from the top of their intention stack.
- Finished intention stacks are removed from the set of intentions.
Failure

Causes for failure:
- Lack of relevant or applicable plan for an achievement goal
- Failure of a test goal
- Action failure

In case of failure, the whole intention is dropped, unless the failed plan was triggered by a goal addition event.
When an added (achievement or test) goal $g$ fails, the interpreter generates a $\neg !g$ event.

Plans to handle such events can be seen as contingency plans for $g$.

Purpose:
- how to backtrack actions executed on the environment?
- “cleanup” to be performed before (possibly) backtracking the execution.
1. Jason & AgentSpeak Language

2. The Jason Interpreter

3. Features
   - Strong negation
   - Rules
   - Failure Handling
   - Internal actions
   - Customisations
Example

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

+!leave(home)
  :  not raining & not ~raining
  <- .send(mum,askOne,raining,Answer,3000); ...
```
likely_color(Obj,C) :-
    colour(Obj,C)[degOfCert(D1)] &
    not (colour(Obj,_) [degOfCert(D2)] & D2 > D1) &
    not ~colour(C,B).
Example (an agent blindly committed to g)

+!g : g.

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

-!g : true <- !g.
Example (an agent that asks for plans on demand)

```agentSpeak
- !G[error(no_relevant)] : teacher(T)
  <- .send(T, askHow, { +!G }, Plans);
    .add_plan(Plans);
  !G.
```

*in the event of a failure to achieve any goal G due to no relevant plan, asks a teacher for plans to achieve G and then try G again*

- The failure event is annotated with the error type, line, source, ... *error(no_relevant)* means no plan in the agent’s plan library to achieve G
- `{ +!G }` is the syntax to enclose triggers/plans as terms
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:

```
libname.action_name(...)
```
Standard Internal actions

Standard (pre-defined) internal actions have an empty library name

- `.print(term_1, term_2, ...)`
- `.union(list_1, list_2, list_3)`
- `.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.
Possible customisations in *Jason*

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

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

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