Programmation Orientée Agent

programmer un agent BDI avec JASON

Stéphane Airiau

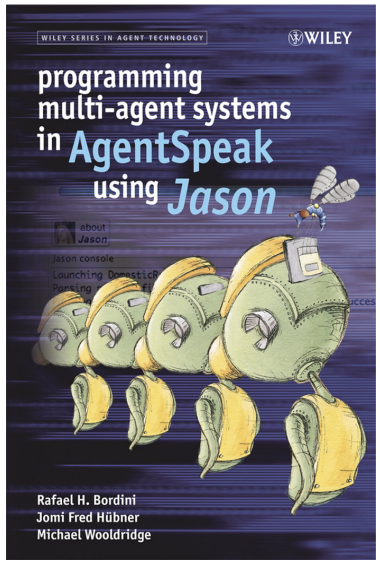
Université Paris-Dauphine

Sources utilisées pour ce cours

Transparents

- Olivier Boissier, ENS Mines Saint Etienne
- Jomi F. Hübner, UFSC/DAS Brazil
- Rafael H. Bordini PUCRS Brazil

Livre :



Programming multi-agent systems in AgentSpeak using Jason
Rafael H. Bordini
Jomi Fred Hübner
Michael Wooldridge
Wiley 2007

Un nouveau type de langage, pour quoi faire ?

- un agent est *autonome* : il ne nécessite pas de contrôle extérieur, il décide à tout moment de ses actions
- un agent est *proactif* : il poursuit ses **buts** au cours du temps, il doit être attentif aux changements de l'environnement pour décider quels buts il peut/doit atteindre. Si un plan échoue, l'agent tentera quelque chose d'autre pour atteindre son but.
- un agent est *réactif* ⇨ Il faut donc porter une importance particulière aux **percepts** (forme perçue d'un stimulus externe) sinon un agent risque de manquer une opportunité.
- un agent est *situé* dans un *environnement* ⇨il faut prendre en compte les **actions** que peut exercer l'agent.
- un agent est *social* et communique avec d'autres agents à l'aide de **messages**.

  de plus, un agent peut appartenir à une *organisation sociale* (*rôle* des agents, *obligations*, *normes*, *groupes*)

Les agents ne sont pas seulement des objets

- un objet ne contrôle pas son comportement
- le modèle objet n'a rien à dire de général à propos de la réactivité, de la pro-activité et de la communication
- les objets « classiques »peuvent utiliser le multi-threads, les SMA sont par définition multi-threads.

Cependant, les agents sont des logiciels, et souvent on peut adapter ou adopter des éléments de génie logiciel orienté objet.

ex : notation UML ⇨notation AUML

Quand utiliser une approche agent ?

- système réactif ⇨ environnements dynamiques, ouverts, incertains, complexes, etc.
- système disitribué ⇨environnements où les données, le contrôle et le traitement sont distribués (trafic aérien, routier, gestion d'eau, d'électricité)
- métaphores naturelles : l'environnement est une société d'agents (e-commerce, systèmes économiques, système d'informations distribuées, organisation)

- système est-il autonome ?
- a-t-il des buts ?
- vu comme un objet, est-il actif ?
- fait-il plusieurs choses en même temps ⇨ interaction
- doit-il adapter son comportement à son environnement ?

Implémentation

Langages pour agents BDI :

- **3APL** (triple A)-PL pour
  Abstract **A**gent **P**rogramming **L**anguage,
  ou **A**rtificial **A**utonomous **A**gents **P**rogramming **L**anguage
- **JACK** : successeur de Procedural Reasoning System (PRS) et Distributed Multi-Agent Reasoning System (dMARS).
- **Jadex** : basé sur la plateforme JADE.
- **JASON** : une extension d'AgentSpeak. Développé en JAVA

méthodologie et aide au développement :

- PDT (Prometheus) ⇨ produit du code en JACK
- TAOM4E (Tropos) ⇨ produit du code en Jadex
- IDK (INGENIAS) ⇨ produit du code en JADE
- agentTool III (O-MaSE) ⇨ produit du code en JADE
- PTK (PASSI) ⇨ produit du code en JADE avec AgentFactory

AgentSpeak

- Proposée à Anand Rao [MAAMAW 1996]
- c'est un langage pour les agents BDI
- notation élégante, basée sur la programmation logique
- sources d'inspiration :
  - PRS (Georgeff & Lansky)
  - dMARS (Kinny)
  - BDI Logics(Rao & Georgeff)
- langage de programmation abstrait dans le but d'obtenir des résultats théoriques

- **Belief :** knowledge about the world and its own internal state
- **Desires (or goals) :** what the agent may decide to work towards achieving, they may be considered as *options* for an agent.
  Desires may be incompatible with one another.
- **Intentions :** how the agents has decided to tackle these goals ⇨ plan instantiated to achieve some goal
  intentions are goals that the agent has *committed* to achieve
- **No planning from first principles :** agents use a *plan library* (library of partially instantiated plans to be used to achieve the goals) ⇨represents the agent's *know how*
- events : happens as a consequence to changes in the agent's belief or goals.

Practical reasoning agents : quickly reason and react to asynchronous events.

Hello World

File "hello.asl"

```
1 started.
2 +started <- .print("Hello World!").
```

- agent's initial belief (and possibly initial goals)
  - ➪ single belief `started`
- agent's plans
  - ➪ whenever the agent believes `started`, print the text `Hello World!`.
    - `+` : when you acquire the belief... trigger of a plan
    - `<-` separator between trigger and body of the plan
    - the body of the plan is to execute an action of printing
    - here, the action is *internal*, it does not change the environment

    internal actions starts with a full stop "`.`"
    actually, it is a pre-defined internal action
    (other examples `.send(...)`, `.broadcast(...)`).

Factorial of 5

```
1  fact(0,1)
2
3  +fact(X,Y)
4     : X < 5
5     <- + fact(X+1, (X+1)*Y).
6
7  +fact(X,Y)
8     : X == 5
9     <- .print("fact 5 == ", Y).
```

- `fact(x,y)` factorial of x is y
- two plans with the same triggering condition, but different contexts
- fill up the belief base with `fact(0,1)`, `fact(1,1)`, ..., `fact(5,120)`!

Factorial of 5

```
1  fact(0,1)
2
3  +fact(X,Y)
4     : X < 5
5     <- + fact(X+1, (X+1)*Y).
6
7  +fact(X,Y)
8     : X == 5
9     <- .print("fact 5 == ", Y).
```

- `fact(x,y)` factorial of x is y
- two plans with the same triggering condition, but different contexts
- fill up the belief base with `fact(0,1)`, `fact(1,1)`, ..., `fact(5,120)`!

Factorial version 2

```
1  !print_fact(5).
2
3  +!print_fact(N)
4    <-!fact(N,F);
5    .print("factorial of ", N, " is ", F).
6
7  +!fact(N,1) : N==0.
8
9  +!fact(N,F) : N>0
8    <- !fact(N-1,F1);
9    F = F1*N.
```

- ! this goal is to be achieved

Jason Agent Programming

- Beliefs
  - annotation
  - strong negation
  - rules
- Goals
- Plans
  - triggering events
  - context
  - body
  - actions
  - achievement goals
  - test goals
  - mental notes
  - internal actions
  - expressions
  - plan labels

Beliefs – Representation

How to represent the agent's *beliefs* ? Use idea from logic programming.

- *atom* : symbol starting with lower case representing a constant.
- *variable* : symbol starting with upper case letter.
  - initially, variables are *free* or *uninstantiated*
  - once *instantiated* or *bound*, they keep the value in their *scope*
  - ⇨ variables are bound to values by *unification*
- a *structure* is used to represent complex data : it is composed by an atom and a number of arguments.
- lists are represented using brackets `[` and `]`. The head of the list can be separated from the rest of the list using the symbol `|`.

```
1 tall(john).
2 likes(john, music).
3 staff("stephane", classes([java,ai,social_choice])).
```

Beliefs – Representation – Annotations

An annotation provides details about a belief :

- can be added by the programmer (exemple : program an `expire` annotation)
- provide information about the source :
  - a belief that is ”sensed” by the agent is called a *percept*

```
1 | colour(box, red) [source(percept)]
```

  - a belief that is acquired through communication

```
2 | likes(john,music) [source(paul)]
```

- a mental note that is added by the programmer
- it is possible to have nested annotations

Beliefs – Negation

Negation is a source of difficulties in logic programming.

- *closed world assumption* : anything that is neither known to be true, or derivable from known facts is assumed to be *false*.

➪ `not` operator : when interpreters fails to derive a fact

➪ ~ operator : the agent believes something to be false.

```
~colour(box, white).
```

the agent believes it is not the case that the box is white

```
1 colour(box,blue) [source(bob)]
2 ~colour(box,white)[source(john)]
3 colour(box, red) [source(percept)]
4 colourblind(bob)[source(self),degOfCert(0.7)]
5 lier(bob)[source(self),degOfCert(0.2)]
```

N.B. `degOfCert` has to be programmed, it is *not* part of the language

Beliefs – Rules

```
conclusion :- conditions
```

```
1 likely_color(C,B)
2   :- colour(C,B) [source(S)] & (S==self | S==percept).
```

Beliefs – Dynamics

- the agent architecture is responsible for the update of the belief

➪ new belief *by perception* :
beliefs annotated with `[source(percept)]` are *automatically* updated accordingly

➪ new belief *by intention* :
operators + or − can be used to add or remove beliefs
annotation with `source(self)`.

```
1 +good(question); //adds good(question)[source(self)]
2 -good(question); //removes good(question)[source(self)]
```

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

```
1 .send(tom, tell, good(question)); // sent by Bob
  // adds good(question) [source(bob)] in Tom's beliefs base
```

Goals

Goals express the properties of the states of the world that the agent wishes to bring about.

Two types of goals :

- **achievement goals** : denoted by `!` operator

```
!own(house)
```

  ⇨ try to achieve a particular goal

- **test goals** : denoted by the `?` operator
  ⇨ retrieve information in the beliefs base (or that can be derived)

```
?salary(S)
```

In the source code, one can add the initial goals of an agent.

Goals – Dynamics 1

Adding a new goal :

- by **intention** :
  the operators `!` and `?` can be used to add a new goal with the annotation `[source(self)]`

```
1 !write(book);
2 // adds an achievement goal !write(book) [source(self)]
3 ?publisher(P);
4 // adds the test goal ?publisher(P)[source(self)]
```

Goals – Dynamics 2

Adding a new goal :

- by **communication**
  when an agent receives an *achieve* or *unachieve* message, a new goal annotated with the sender of the message is automatically added

```
5 .send(tom, achieve,write_review(article)); //sent by Bob
6 // adds a new goal !write_review(article)[source(bob)] for Tom
7 ...
8 .send(tom, unachieve,write_review(article)); //sent by Bob
9 // removes goal !write_review(article)[source(bob)] for Tom
```

  an `askOne` or `askAll` message will add a test goal annotated with the sender of the message

```
5 .send(tom, askOne,author(article), Answer); //sent by Bob
6 // adds a new goal ?author(article)[source(bob)] for Tom
7 // the response of Tom will unify with Answer
```

Plans

```
triggering event : context <- body.
```

- **triggering event** : an event represent a change in the beliefs or in the goals of an agent.
  a change ⇨ can be addition or deletion
  if the triggering event of a plan matches a particular event, we say the plan is *relevant*.
- **context** : the agent commits to a course of action as late as possible.
  for a plan to be executed, the *context* must be a *logical consequence* of the beliefs base.
  A plan which context is a logical consequence of the beliefs is called *applicable*.
- **body** : sequence of goals (considered as sub-goals) and actions

Plans – labels

```
@label triggering event : context <- body.
```

- A plan may have a label, i.e. a name.
- Choose a pertinent name!
- A label may include annotation, which can be used for meta-level reasoning.

Plans – Triggering events

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

- belief addition or deletion may occur after a belief update
- goal addition or deletion may occur due to the execution of plans, as a result of communication
- deletion of goals may be used for handling plan failure.

Plans – Context

A context is a logical sentence must evaluate to `true` given the current belief for the plan to be *applicable*

- use conjunction operator (`&`)
- use disjunction operator (`|`)
- use negation

Syntax	Semantic
`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`

N.B. Not believing that *l* is `false` is not the same as believing it is `true` : the agent may be ignorant about `l`.

When a variable is bound during the evaluation of the context, the variable keeps its value for the rest of the plan.

Plans – Body

The body is a sequence of formulæ, each separated by `;`.
There are 6 types of formulæ :

- actions
- achievement goals
- test goals
- mental notes
- internal actions
- expressions

Plans – Body – Action

- The actions are the actions that are available to the agents
  (i.e. if the agent is a robot, we know the physical actions that the robot can perform (turn, move, use sensor))
- we use a symbolic representation of the action
- ⇨ the interface with the ”real” action is performed using `JAVA`
- the execution of the plan is *suspended* until a feedback from the action is received (telling whether the action was a success or a failure)
- if an action fails, the plan *fails*

Plans – Body – Achievement Goals

- The philosophy of BDI is *not* to plan on first principle!
- The idea is to delay the choice of an action course as late as possible for choose the plan that is most adequate to the current state of the world.
- ⇨ a plan may post a new achievement goal : `!goal`.
- Until this goal is reached, the plan is *suspended*
- Exception with the notation `!!goal` : the plan can continue before `goal` is achieved.

Plans – Body – Test Goals

Beliefs may have changed since the context of the plan has been checked
(e.g. think about receiving a ball)
Using a test goal allows to get up to date information about the current beliefs

Plans – Body – Mental Notes

- **!** belief updates are performed by the architecture
- it may be useful to create beliefs during the plan execution
  ex : for reminding (one self or another one) to do something or that something was done
  ex : `+clean(bathroom, Today, Stephane)`
- one may need to delete these mental notes later!

N.B. There is a $-+$ operator that removes the last instance of a mental note and addd the new instance.

Plans – Body – Internal Action

- internal action have a full stop "`.`" in front of their names.
- internal actions can be programmed in `JAVA`
- Jason defines a set of *standard internal actions* ex : `.broadcast`, `.my_name`, `.send`, actions for manipulating strings, lists, plans, intentions, etc...

Plans – Body – Expressions

- One can use logical sentence in the body.
- ! if they evaluate to `false`, the plan fails.

Plans – Dynamics

The plans that forms 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 message of type
  - `tellHow`
  - `untellHow`

Plans – some examples

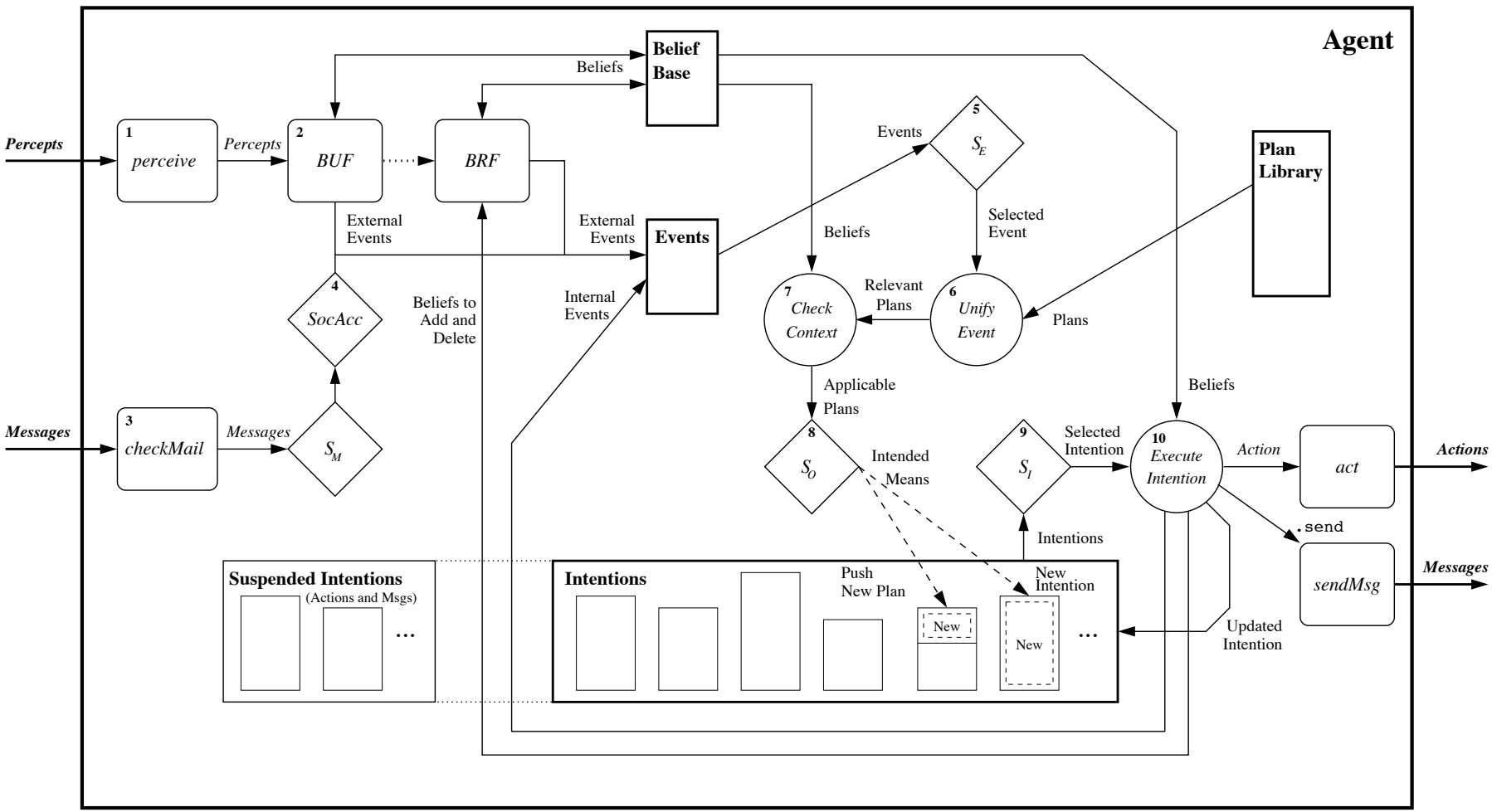
```
1 +!leave(home)
2    : not raining & not ~ raining
3    <- location(window);
4      ?curtain_type(Curtains);
5      open(Curtains);
6      ...
```

```
1 down_the_pub(Pub) [source(Agent)]
2    : good_friend(Agent)
3    <- !location(Pub).
```

Jason Reasoning Cycle

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

Jason Reasoning Cycle



Retrieving relevant plans

Selected event :

```
+colour(box1,blue)[source(percept)], ⊤
```

Plan Library :

```
@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)] :... <- ...
```

Retrieving applicable plans among the relevant ones

Belief base

```
shape(box1,box)[source(percept)].
pos(box1,coord(9,9))[source(percept)].
colour(box1,blue)[source(percept)].
shape(sphere2,sphere)[source(percept)].
pos(sphere2,coord(7,7))[source(bob)].
colour(sphere2,red)[source(percept),source(john)].
```

Applicable plans

```
@p2
   +colour(Object,Colour)
     : shape(Object,box) & not pos(Object,coord(0,0))
     <- ... .

@p3
   +colour(Object,Colour)
     : colour(OtherObj,red)[source(S)] & S=percept &
     shape(OtherObj,Shape) & shape(Object,Shape)
     <- ... .

@p6
   +colour(Object,blue)[source(percept)]
     : colour(OtherObj,red)[source(percept)] &
     shape(OtherObj,sphere)
     <- ... .
```

Selecting one relevant plan / intention

Event
⟨ +b, ⊤⟩

intended means
`+b : true <- !g; a1.`

➪ creation of an event
⟨+!g, [+b : true <- !g; a1.]⟩

➪ find a relevant plan, say `@p2`
`@p2 +!g : true <- a2.`

➪ intended means is pushed on
the top of intention that generated the event
`[ +!g :true<- a2. | +b :true<-!g; a1. ]`

Plan failure

Sources :

- lack of relevant or applicable plans
- failure of a test goal
- action failure

```
!g1. // initial goal
@p1 +!g1 : true <- !g2(X); .print("end g1 ",X).
@p2 +!g2(X) : true <- !g3(X); .print("end g2 ",X)
@p3 +!g3(X) : true <- !g4(X); .print("end g3 ",X).
@p4 +!g4(X) :true <- !g5(X); .print("end g4 ",X).
@p5 +!g5(X) : true <- .fail.

@f1 -!g3(failure) : true <- .print("in g3 failure").
```

Jason Vs JAVA

Consider a very simple robot with two goals :

- when a piece of gold is seen, go to it
- when battery is low, charge

```
1  public class Robot extends Thread {
2     boolean seeGold, lowBattery;
3
4     public void run() {
5       while (true){
6         while (!seeGold) {
7           randomWalk();
8       }
9       while (seeGold) {
10        a = selectDirection();
11        doAction(go(a));
12      }
13    }
14 }
```

Jason Vs JAVA

charge battery behaviour ?

```
1  public class Robot extends Thread {
2     boolean seeGold, lowBattery;
3
4     public void run() {
5       while (true){
6         while (!seeGold) {
7           if(lowBattery)
8             chargeBattery();
9           else
10            randomWalk();
11        }
12        while (seeGold) {
13          if(lowBattery)
14            chargeBattery();
15          else{
16            a = selectDirection();
17            doAction(go(a));
18          }
19        }
20      }
21    }
22 }
```

Jason Vs JAVA

```
1  +see(gold)
2    <- !goto(gold).
3
4  +!goto(gold) : see(gold)
5    <- !select_direction(A);
6      go(A);
7      !goto(gold).
8
9  +battery(low)
10   <- .suspend(goto(gold));
11     !charge;
12     .resume(goto(gold)).
```

Execution and Communication Platform

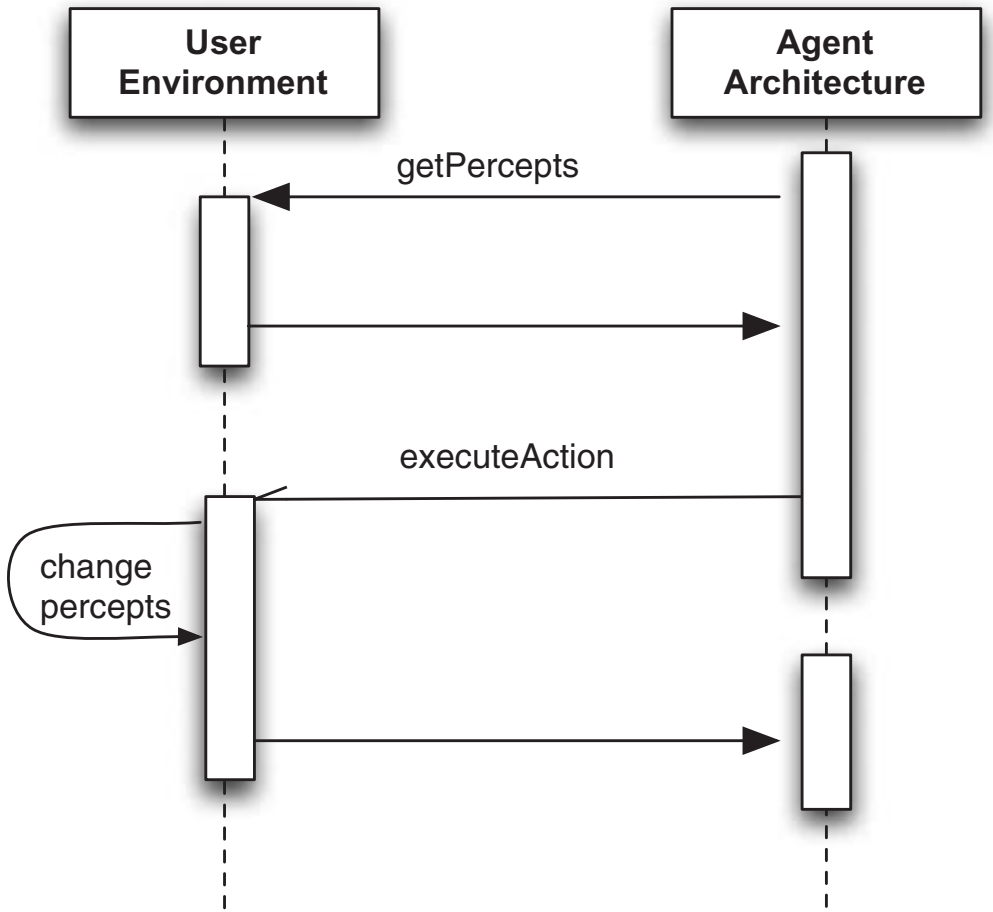
Different execution and communication platforms can be used with Jason :

- **Centralised** : all agents in the same machine, one thread by agent, very fast
- **Centralised (pool)** : all agents in the same machine, fixed number of thread, 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



```
1  import jason.*;
2  public class HouseEnv extends Environment {
3
4    // common literals
5    public static final Literal of = Literal.parseLiteral("open(fridge)");
6    public static final Literal clf = Literal.parseLiteral("close(fridge)");
7
8    public void init(String[] args) {
9      updatePercepts();
10   }
11
12   void updatePercepts() {
13     // get the robot location
14     Location lRobot = model.getAgPos(0);
15     // add agent location to its percepts
16     if (lRobot.equals(model.lFridge)) {
17       addPercept("robot", af);
18     }
19   }
20
21   public boolean executeAction(String ag, Structure action) {
22     boolean result = false;
23     if (action.equals(of)) { // of = open(fridge)
24       result = model.openFridge();
25     } else if (action.equals(clf)) { // clf = close(fridge)
26       result = model.closeFridge();
27     }
28     if (result)
29       updatePercepts();
30     return result;
31   }
32 }
```

Configuration of a Multiagent System

```
1  MAS my_system }
2     infrastucture: Jade // could be Centralised
3
4     environment: robotEnv
5
6    agents: c3po;
7            r2d2 at lucasart.ilm.com;
8            wallE #10; // 10 instances of wallE
9  }
```