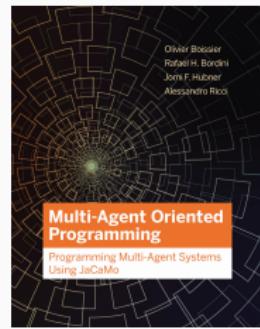


Agent Dimension

Multi-Agent Oriented Programming Tutorial

EASSS 2023 — Prague



2023-07-13

Plan:

slot 1h00:

- 40 min for introduction to knowledge based agent in JaCaMo
- 20 min for practical part with execution of simple application and exercises

slot 1h30

- 20 min for communication
- 30 min for fundamentals (BDI)
- 40 min for exercises



Agent



2023-07-13

└ Agent



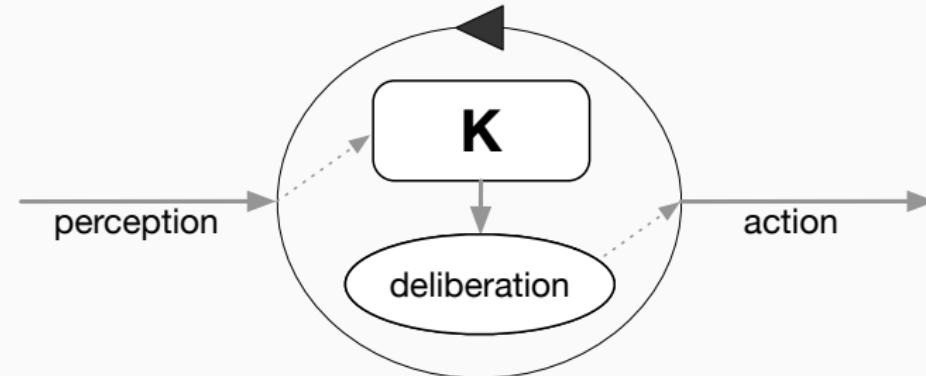
The goal of this part is to introduce agent oriented programming

So it is about programming and about agent

What is an agent?

- it is not like a conventional program (that starts and ends)
- it is continuously running (like a server)
- continuously perceiving (“inputs”) the environment (sensors, messages, user commands, ...)
- the “output” is continuous acting
- output is not data (as a procedural program) neither knowledge (as an inference engine)
- it changes a lot!
- it is about how to program to act instead of programming to change data, or to infer something
- it is about programming an agent and not a computer or a “mind”!

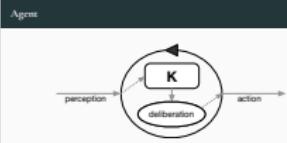
Agent



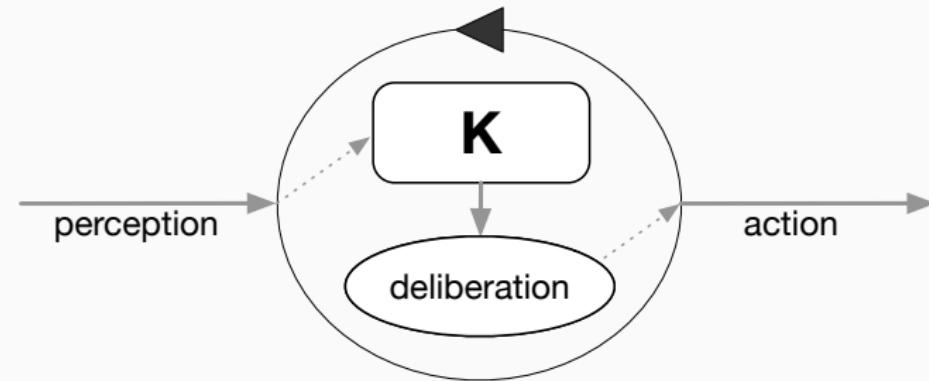
2023-07-13

└ Agent

- considering an AI context, we take a **symbolic approach**:
- the agent has **knowledge**
- its behaviour is based on knowledge — “The Knowledge Level” [Newell, 1982]
- the developer defines that K (it can be learnt, but not the focus today)
- what is K? information, rules, plans, goals, ...
- in our case (agents), the focus is on K elements directed to actions



Agent



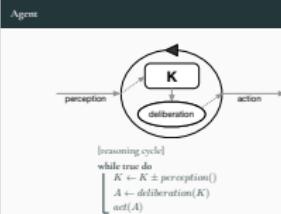
[reasoning cycle]

while true do

```
 $K \leftarrow K \pm perception()$ 
 $A \leftarrow deliberation(K)$ 
 $act(A)$ 
```

2023-07-13

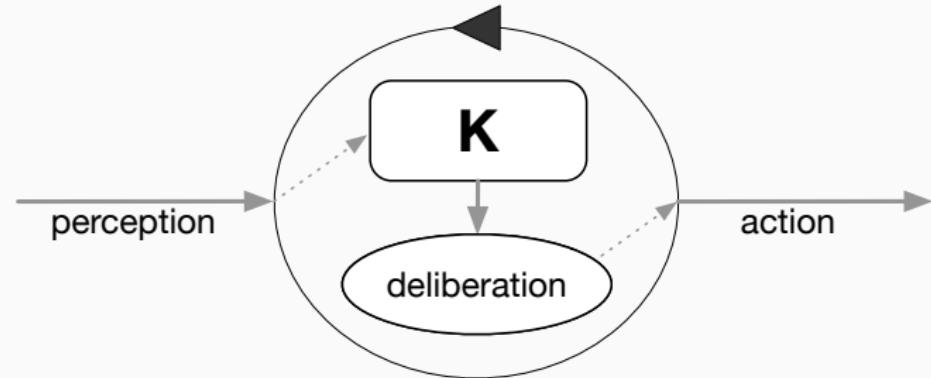
Agent



- what is the engine?
- it is a continuous process that
 - perceives
 - decides an actions to achieve a goal
 - does the actions
- the agent has **autonomy** the choose actions
- an agent decides what to do!
- part of the task that usually a programmer does (ordering the actions) is done by the agent.
- to program an agent is to define K (and not to write an algorithm)

Let's move to a more practical perspective to consolidate the basic concepts (we latter return to the conceptual background)

Agent



to program an agent is to define K

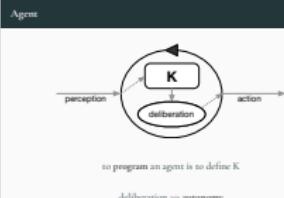
deliberation \leadsto autonomy

2023-07-13

Agent

- what is the engine?
- it is a continuous process that
 - perceives
 - decides an actions to achieve a goal
 - does the actions
- the agent has **autonomy** the choose actions
- an agent decides what to do!
- part of the task that usually a programmer does (ordering the actions) is done by the agent.
- to program an agent is to define K (and not to write an algorithm)

Let's move to a more practical perspective to consolidate the basic concepts (we latter return to the conceptual background)



Beliefs : information about the environment, other agents, itself, application, ...

```
temperature(20).
happy(bob).
```

Goals : the agent objectives

```
!temperature(20).
!happy(bob).
```

Plans :

Agent Knowledge (in JaCaMo)

Beliefs : information about the environment, other agents, itself, application,

```
temperature(20).
happy(bob).
```

Goals : the agent objectives

```
!temperature(20).
!happy(bob).
```

Plans :

2023-07-13

Agent Knowledge (in JaCaMo)

JaCaMo is a framework with languages that allows us to implement systems based on K agents and ...

so elements of agent knowledge in JaCaMo (beliefs quite usual, novelty are goals and plans and they are “interpreted”)

- Syntax inspired by Prolog: predicate(arguments)
- Plans = know how
- informal semantics: <if this happens> <- <do this>
- event oriented



Beliefs : information about the environment, other agents, itself, application, ...
 temperature(20).
 happy(bob).

Goals : the agent objectives
 !temperature(20).
 !happy(bob).

Plans

Agent Knowledge (in JaCaMo)

Beliefs : information about the environment, other agents, itself, application,

```
temperature(20).
happy(bob).
```

Goals : the agent objectives

```
!temperature(20).
!happy(bob).
```

Plans :

2023-07-13

Agent Knowledge (in JaCaMo)

JaCaMo is a framework with languages that allows us to implement systems based on K agents and ...
 so elements of agent knowledge in JaCaMo (beliefs quite usual, novelty are goals and plans and they are “interpreted”)

- Syntax inspired by Prolog: predicate(arguments)
- Plans = know how
- informal semantics: <if this happens> <- <do this>
- event oriented



```

Beliefs : information about the environment, other agents, itself,
application, ...
temperature(20).
happy(bob).

Goals : the agent objectives
!temperature(20).
!happy(bob).

Plans : specifies how goals can be achieved by actions
+!temperature(20) <- startCooling.
+!happy(bob) <- kiss(bob).

```

Agent Knowledge (in JaCaMo)

Beliefs : information about the environment, other agents, itself, application,

```

temperature(20).
happy(bob).

```

Goals : the agent objectives

```

!temperature(20).
!happy(bob).

```

Plans : specifies how goals can be achieved by **actions**

```

+!temperature(20) <- startCooling.
+!happy(bob) <- kiss(bob).

```

2023-07-13

Agent Knowledge (in JaCaMo)

JaCaMo is a framework with languages that allows us to implement systems based on K agents and ...

so elements of agent knowledge in JaCaMo (beliefs quite usual, novelty are goals and plans and they are “interpreted”)

- Syntax inspired by Prolog: predicate(arguments) - Plans = know how
- informal semantics: <if this happens> <- <do this>
- event oriented



```

Beliefs : information about the environment, other agents, itself,
application, ...
temperature(20).
happy(bob).

Goals : the agent objectives
!temperature(20).
!happy(bob).

Plans : specifies how goals can be achieved by actions
+!temperature(20) <- startCooling.
+!happy(bob) <- kiss(bob).

specifies reactions to mental state changes
+temperature(10) <- !temperature(20).
-happy(bob) <- !happy(bob).

```

Agent Knowledge (in JaCaMo)

Beliefs : information about the environment, other agents, itself, application,

```

temperature(20).
happy(bob).

```

Goals : the agent objectives

```

!temperature(20).
!happy(bob).

```

Plans : specifies how goals can be achieved by **actions**

```

+!temperature(20) <- startCooling.
+!happy(bob) <- kiss(bob).

```

specifies **reactions** to mental state changes

```

+temperature(10) <- !temperature(20).
-happy(bob) <- !happy(bob).

```

2023-07-13

Agent Knowledge (in JaCaMo)

JaCaMo is a framework with languages that allows us to implement systems based on K agents and ...

so elements of agent knowledge in JaCaMo (beliefs quite usual, novelty are goals and plans and they are “interpreted”)

- Syntax inspired by Prolog: predicate(arguments) - Plans = know how
- informal semantics: <if this happens> <- <do this>
- event oriented



Knowledge Sources

Beliefs, goals, and plans are provided by

- perception: in the case of beliefs
- developers: initial mental state of the agent
- other agents: by communication
- the agent itself: by reasoning or learning

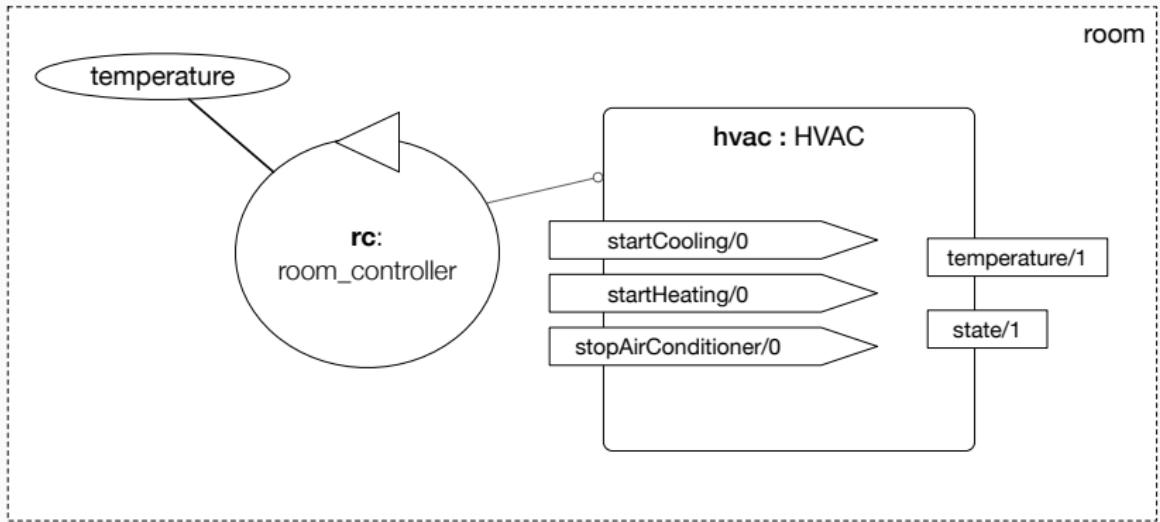
2023-07-13

└ Knowledge Sources

Beliefs, goals, and plans are provided by

- perception: in the case of beliefs
- developers: initial mental state of the agent
- other agents: by communication
- the agent itself: by reasoning or learning

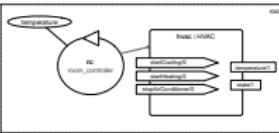
Smart Room Scenario — initial implementation



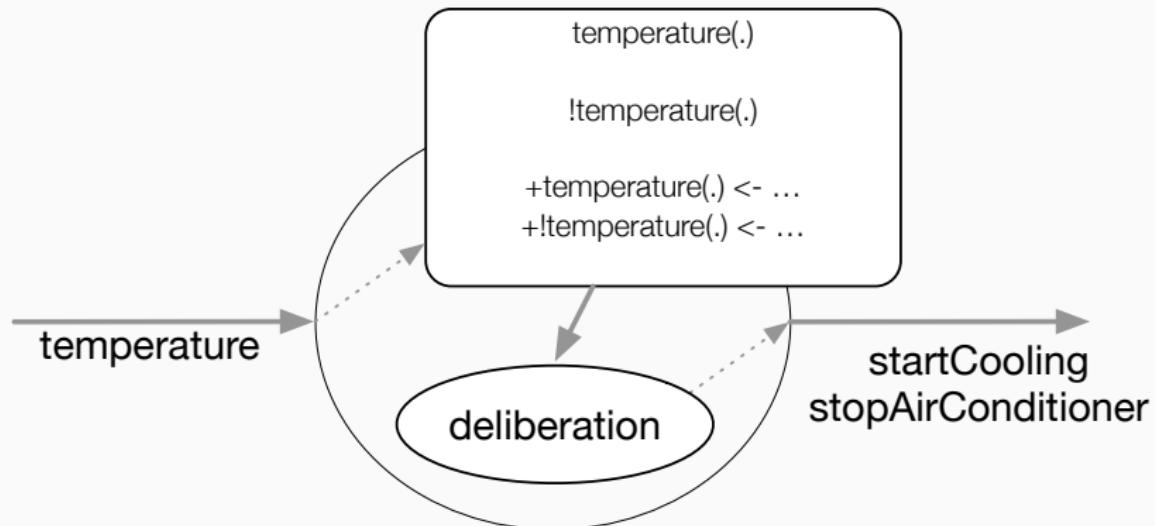
2023-07-13

Smart Room Scenario — initial implementation

- HVAC provides perception of its state and the current temperature
 - exposes 3 actions for the agent
- (details of how to program this artifact will be presented later)



Agent Programming (in JaCaMo)

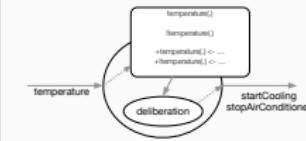


2023-07-13

Agent Programming (in JaCaMo)

Let's focus on programming that agent

- the perception of the current temperature is mapped to a belief like **temperature(30)**
- the objective to maintain some temperature is mapped to a belief like **!temperature(30)**
- the agent has plans to react to changes in the current temperature and the creation of new goals to maintain some temperature



```
+temperature(30) <- !temperature(20).  
+!temperature(20) <- startCooling.
```

Agent Programming (in JaCaMo)

```
+temperature(30) <- !temperature(20).  
+!temperature(20) <- startCooling.
```

2023-07-13

Agent Programming (in JaCaMo)

(agents are programmed in JaCaMo using the Jason language)

- these 2 lines are a complete Jason program, a program with 2 plans
- beliefs are added by perception
- (read the plans): in the event of a new belief `temperature(30)`, react to it creating the goal `!temperature(20)`
- in the event of a new a goal `!temperature(20)`, react to it by doing `startCooling`
- the program has no begin/end, declarative approach (K is declared)
- set of reactive “rules” (implemented by the plans)
- which are the problems of this implementation?
(implement, run, and see!)



```
+temperature(30) <- !temperature(20).  
+temperature(20) <- stopAirConditioner.  
+!temperature(20) <- startCooling.
```

Agent Programming (in JaCaMo)

```
+temperature(30) <- !temperature(20).  
+temperature(20) <- stopAirConditioner.  
+!temperature(20) <- startCooling.
```

2023-07-13

Agent Programming (in JaCaMo)

(improved version with stopAirConditioner, that stops
(image the agent behaviour) - which are the problems of this implementation?

```
// initial belief, given by the developer
preference(20).

// reaction to changes in the temperature
+temperature(T) : preference(P) & math.abs(P-T) > 2
  <- !temperature(P).
+temperature(T) : preference(T)
  <- stopAirConditioner.

// plans to achieve some temperature
+!temperature(P) : temperature(T) & T > P
  <- startCooling.
```

2023-07-13

Agent Programming (in JaCaMo)

What is new:

- new belief, not perceived, but defined in the initial code of the agent
- variable with upper case first letter
- plans have context, used for the agent to select the most appropriated
- the evaluation of the context is like a query to the belief base, and it may assign values to variables

Agent behaviour:

- any change in temperature produces actions to start cooling, is it ok?
- what if the preference changes?

```
Agms Programming in JaCaMo

// initial belief, given by the developer
preference(20).

// reaction to changes in the temperature
+temperature(T) : preference(P) & math.abs(P-T) > 2
  <- !temperature(P).
+temperature(T) : preference(T)
  <- stopAirConditioner.

// plans to achieve some temperature
+!temperature(P) : temperature(T) & T > P
  <- startCooling.
```

```
// initial belief, given by the developer
preference(20).

// initial goal, given by the developer
!keep_temperature.

// maintenance the goal pattern
+!keep_temperature
  : temperature(T) & preference(P) & T > P
  <- startCooling;
  !keep_temperature.

+!keep_temperature
  : temperature(T) & preference(P) & T <= P
  <- stopAirConditioner;
  !keep_temperature.
```

2023-07-13

Agent Programming (in JaCaMo)

maintenance goal — long term goals

agent is not reacting to changes in beliefs anymore, it has a "forever" goal that, based on the circumstances, select a proper plan of actions

- does it reacts to changes in the preference?
- pro-activity

```
Agms Programming (in JaCaMo)

// initial belief, given by the developer
preference(20).

// initial goal, given by the developer
!keep_temperature.

// maintenance the goal pattern
+keep_temperature
  : temperature(T) & preference(P) & T > P
  <- startCooling;
  keep_temperature.

+keep_temperature
  : temperature(T) & preference(P) & T <= P
  <- stopAirConditioner;
  keep_temperature.
```

Main Features

- **reactivity**: even when achieving some goals
- **pro-activity**: new goals can be created
- **long-term goals**: agents are committed to achieve goals
- **context awareness**: plans are selected based on the circumstances
- **transparency**: we can trace back the reasons for an action
- **sound theoretical background for agent architectures**:
 - practical reasoning [Bratman, 1987]
 - intentions [Cohen and Levesque, 1987]
 - BDI [Rao and Georgeff, 1995]
 - ...

2023-07-13

└ Main Features

because:

- agents have a reasoning cycle
- based on knowledge
- reasoning about what to do (practical reasoning) (detailed later in the course)

Are usual languages (Java, Python, Prolog, ...) appropriate to implement agents?
Can we use them? Sure we can. But they will give us a lot of work to code agents.

- **reactivity**: even when achieving some goals
 - pro activity: new goals can be created
 - long-term goals: agents are committed to achieve goals
- **context awareness**: plans are selected based on the circumstances
- **transparency**: we can trace back the reasons for an action
- **sound theoretical background for agent architectures**
 - practical reasoning [Bratman, 1987]
 - intentions [Cohen and Levesque, 1987]
 - BDI [Rao and Georgeff, 1995]
 - ...

Main Features

- **reactivity**: even when achieving some goals
- **pro-activity**: new goals can be created
- **long-term goals**: agents are committed to achieve goals
- **context awareness**: plans are selected based on the circumstances
- **transparency**: we can trace back the reasons for an action
- **sound theoretical background for agent architectures**:
 - practical reasoning [Bratman, 1987]
 - intentions [Cohen and Levesque, 1987]
 - BDI [Rao and Georgeff, 1995]
 - ...

2023-07-13

└ Main Features

because:

- agents have a reasoning cycle
- based on knowledge
- reasoning about what to do (practical reasoning) (detailed later in the course)

Are usual languages (Java, Python, Prolog, ...) appropriate to implement agents?
Can we use them? Sure we can. But they will give us a lot of work to code agents.

- reactivity: even when achieving some goals
- pro-activity: new goals can be created
- long-term goals: agents are committed to achieve goals
- context awareness: plans are selected based on the circumstances
- transparency: we can trace back the reasons for an action
- sound theoretical background for agent architectures:
 - practical reasoning [Bratman, 1987]
 - intentions [Cohen and Levesque, 1987]
 - BDI [Rao and Georgeff, 1995]
 - ...

- **reactivity**: even when achieving some goals
- **pro-activity**: new goals can be created
- **long-term goals**: agents are committed to achieve goals
- **context awareness**: plans are selected based on the circumstances
- **transparency**: we can trace back the reasons for an action
- **sound theoretical background for agent architectures**:
 - practical reasoning [Bratman, 1987]
 - intentions [Cohen and Levesque, 1987]
 - BDI [Rao and Georgeff, 1995]
 - ...

Main Features

2023-07-13

└ Main Features

because:

- agents have a reasoning cycle
- based on knowledge
- reasoning about what to do (practical reasoning) (detailed later in the course)

Are usual languages (Java, Python, Prolog, ...) appropriate to implement agents?

Can we use them? Sure we can. But they will give us a lot of work to code agents.

- **reactivity**: even when achieving some goals
 - **pro-activity**: new goals can be created
 - **long-term goals**: agents are committed to achieve goals
 - **context awareness**: plans are selected based on the circumstances
- transparency: we can trace back the reasons for an action
 - sound theoretical background for agent architectures:
 - practical reasoning [Bratman, 1987]
 - intentions [Cohen and Levesque, 1987]
 - BDI [Rao and Georgeff, 1995]
 - ...

Main Features

2023-07-13

└ Main Features

because:

- agents have a reasoning cycle
- based on knowledge
- reasoning about what to do (practical reasoning) (detailed later in the course)

Are usual languages (Java, Python, Prolog, ...) appropriate to implement agents?

Can we use them? Sure we can. But they will give us a lot of work to code agents.

- **reactivity**: even when achieving some goals
- **pro-activity**: new goals can be created
- **long-term goals**: agents are committed to achieve goals
- **context awareness**: plans are selected based on the circumstances
- **transparency**: we can trace back the reasons for an action
- sound theoretical background for agent architectures:
 - practical reasoning [Bratman, 1987]
 - intentions [Cohen and Levesque, 1987]
 - BDI [Rao and Georgeff, 1995]
 - ...

Main Features

2023-07-13

└ Main Features

because:

- agents have a reasoning cycle
- based on knowledge
- reasoning about what to do (practical reasoning) (detailed later in the course)

Are usual languages (Java, Python, Prolog, ...) appropriate to implement agents?

Can we use them? Sure we can. But they will give us a lot of work to code agents.

Main Features

- **reactivity**: even when achieving some goals
- **pro-activity**: new goals can be created
- **long-term goals**: agents are committed to achieve goals
- **context awareness**: plans are selected based on the circumstances
- **transparency**: we can trace back the reasons for an action
- **sound theoretical background** for agent architectures:
 - practical reasoning [Bratman, 1987]
 - intentions [Cohen and Levesque, 1987]
 - BDI [Rao and Georgeff, 1995]
 - ...

2023-07-13

└ Main Features

because:

- agents have a reasoning cycle
- based on knowledge
- reasoning about what to do (practical reasoning) (detailed later in the course)

Are usual languages (Java, Python, Prolog, ...) appropriate to implement agents?

Can we use them? Sure we can. But they will give us a lot of work to code agents.

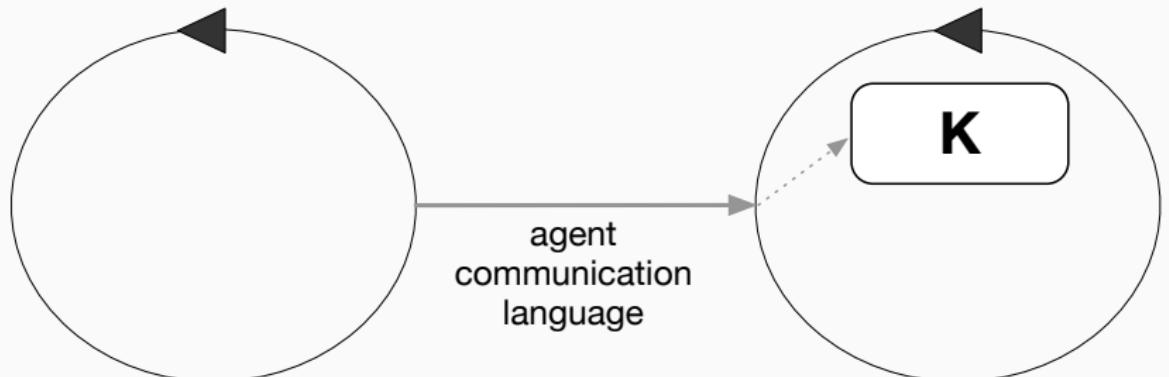
- reactivity: even when achieving some goals
- pro-activity: new goals can be created
- long-term goals: agents are committed to achieve goals
- context awareness: plans are selected based on the circumstances
- transparency: we can trace back the reasons for an action
- sound theoretical background for agent architectures:
 - practical reasoning [Bratman, 1987]
 - intentions [Cohen and Levesque, 1987]
 - BDI [Rao and Georgeff, 1995]
 - ...

Agent Interaction (communication)

2023-07-13

Agent Interaction
(communication)

Agent–Agent Communication

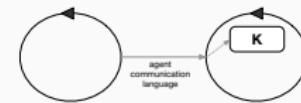


2023-07-13

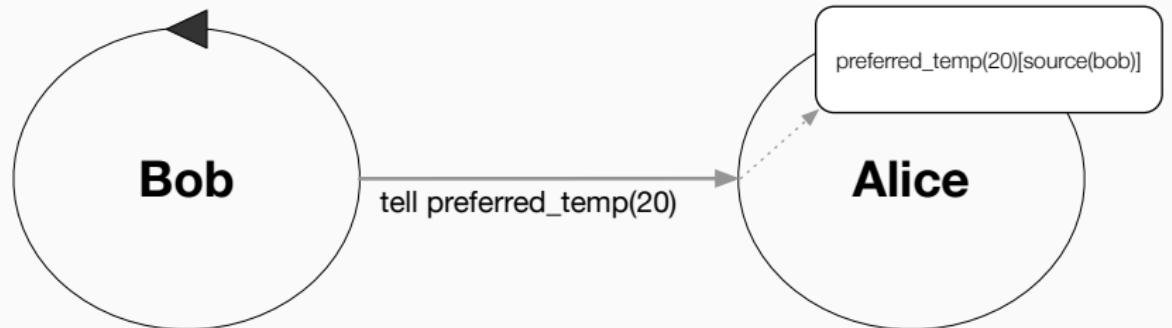
Agent–Agent Communication

Agent Communication Language

- the language to communicate is not the same as the language to program agents, since they have different purposes
- works at the Knowledge level (again!)
- when sending a message, the sender intends to change the mind of receiver (mentalistic view)
- K is transmitted (thing I know that, know how to, I wish, ...) so send beliefs, desires, plans, ...
- used to build negotiation, coordination, information share



Semantic of messages

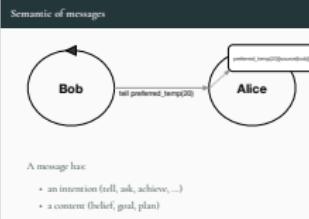


A message has:

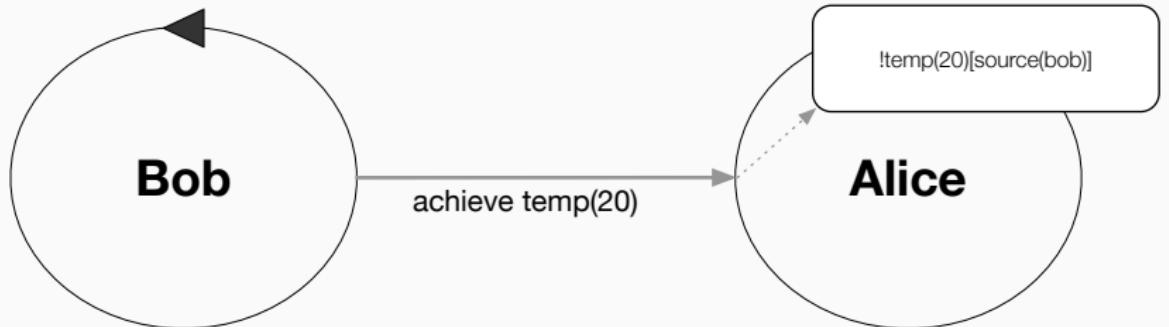
- an intention (tell, ask, achieve, ...)
- a content (belief, goal, plan)

2023-07-13

└ Semantic of messages



Semantic of messages

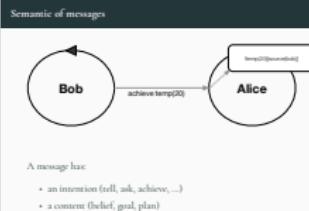


A message has:

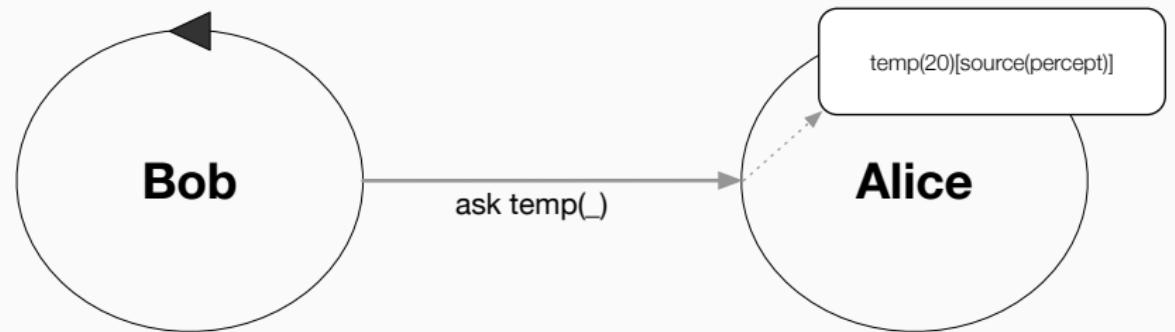
- an intention (tell, ask, achieve, ...)
- a content (belief, goal, plan)

2023-07-13

└ Semantic of messages



Semantic of messages

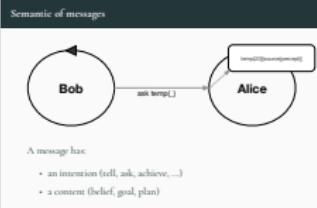


A message has:

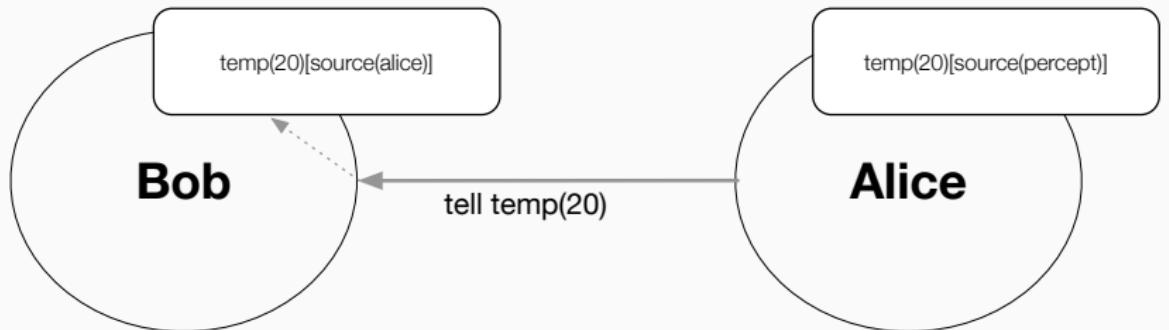
- an intention (tell, ask, achieve, ...)
- a content (belief, goal, plan)

2023-07-13

└ Semantic of messages



Semantic of messages

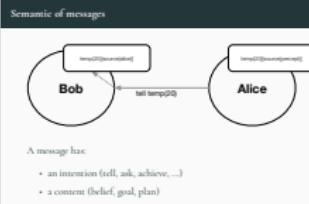


A message has:

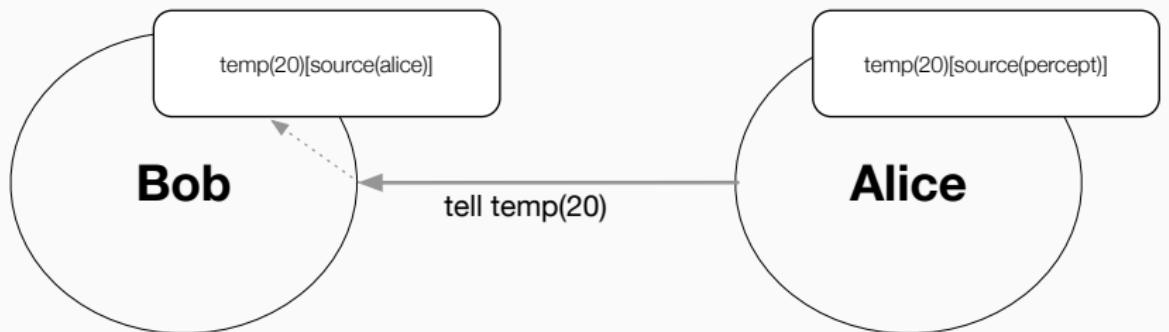
- an intention (tell, ask, achieve, ...)
- a content (belief, goal, plan)

2023-07-13

└ Semantic of messages



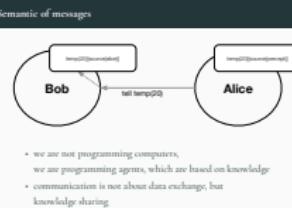
Semantic of messages



- we are not programming computers,
we are programming agents, which are based on knowledge
- communication is not about data exchange, but
knowledge sharing

2023-07-13

Semantic of messages



Sender: `.send(bob,tell,happy(alice))`

- receiver: agent unique name
- performative: tell, achieve, askOne, askHow, ...
- content: a literal

Receiver

- nothing is needed

Properties

- distributed & support for decentralized
- (usually) asynchronous
- KQML vs FIPA-ACL
- not reduced to method invocation

2023-07-13

└ JaCaMo implementation

no code in the receiver, the semantics of the ACL is implemented on the interpreter!

distributed means several machines

decentralised means no central control

KQML and FIPA-ACL are initiatives to standardise ACL

KQML was the standard when Jason was first developed

JaCaMo implementation	
Sender: <code>.send(bob,tell,happy(alice))</code>	
• receiver: agent unique name	
• performative: tell, achieve, askOne, askHow, ...	
• content: a literal	
Receiver	
• nothing is needed	
Properties	
• distributed & support for decentralized	
• (usually) asynchronous	
• KQML vs FIPA-ACL	
• not reduced to method invocation	

- **tell** and **untell**: change beliefs of receiver
- **achieve** and **unachieve**: change goals of receiver
- **askOne** and **askAll**: ask for beliefs of the receiver
- **askHow**, **tellHow**, and **untellHow**: exchange plans with other agent
- **signal**: add an event in the receiver

2023-07-13

└ JaCaMo Performatives

Theoretical background is speech acts [Austin, 1975, Searle and Vanderveken, 1985]: to say is to act; to power of word.

synchronous cases:

.send(a,askOne,v(X),A)

it blocks the intention until an answer is received, the answer is assigned to A

signal is quite recent in JaCaMo (Jason)

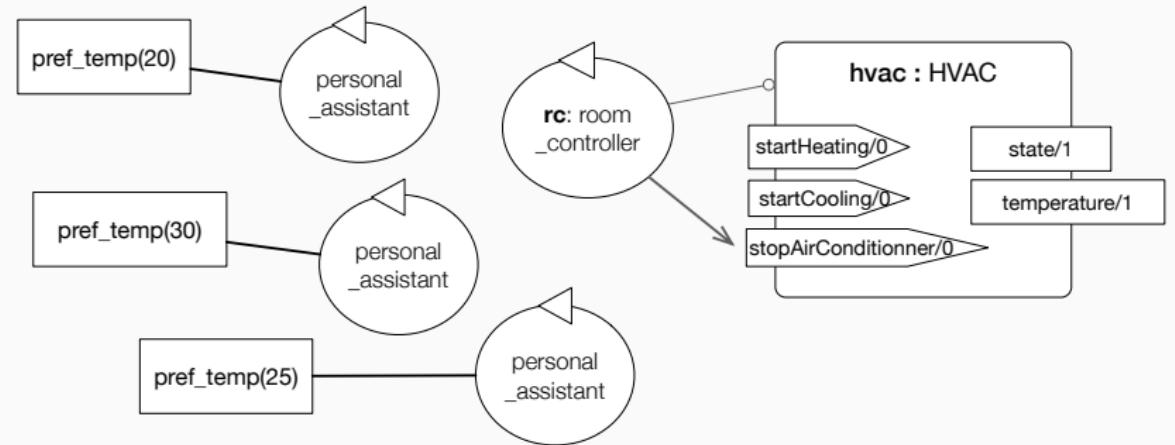
e.g. .send(bob,signal,hello)

- **tell** and **untell**: change beliefs of receiver
- **achieve** and **unachieve**: change goals of receiver
- **askOne** and **askAll**: ask for beliefs of the receiver
- **askHow**, **tellHow**, and **untellHow**: exchange plans with other agent
- **signal**: add an event in the receiver

Smart Room Scenario

many users

The system have to consider the preference of temperature of many users and use a voting strategy to define the target temperature

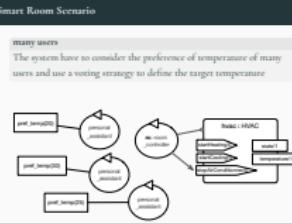


2023-07-13

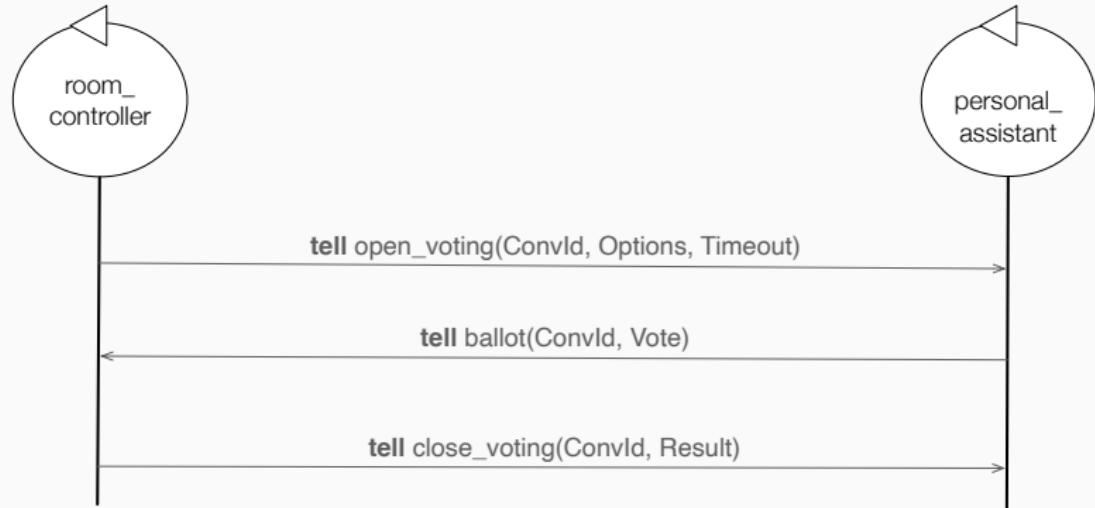
Smart Room Scenario

decentralised solution

we will solve it using agent communication



Interaction Protocols ~~~ coordination



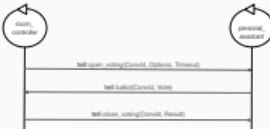
2023-07-13

Interaction Protocols ~~~ coordination

decentralised solution requires coordination (of actions)

coordination of actions, order actions

here, order of communicative actions



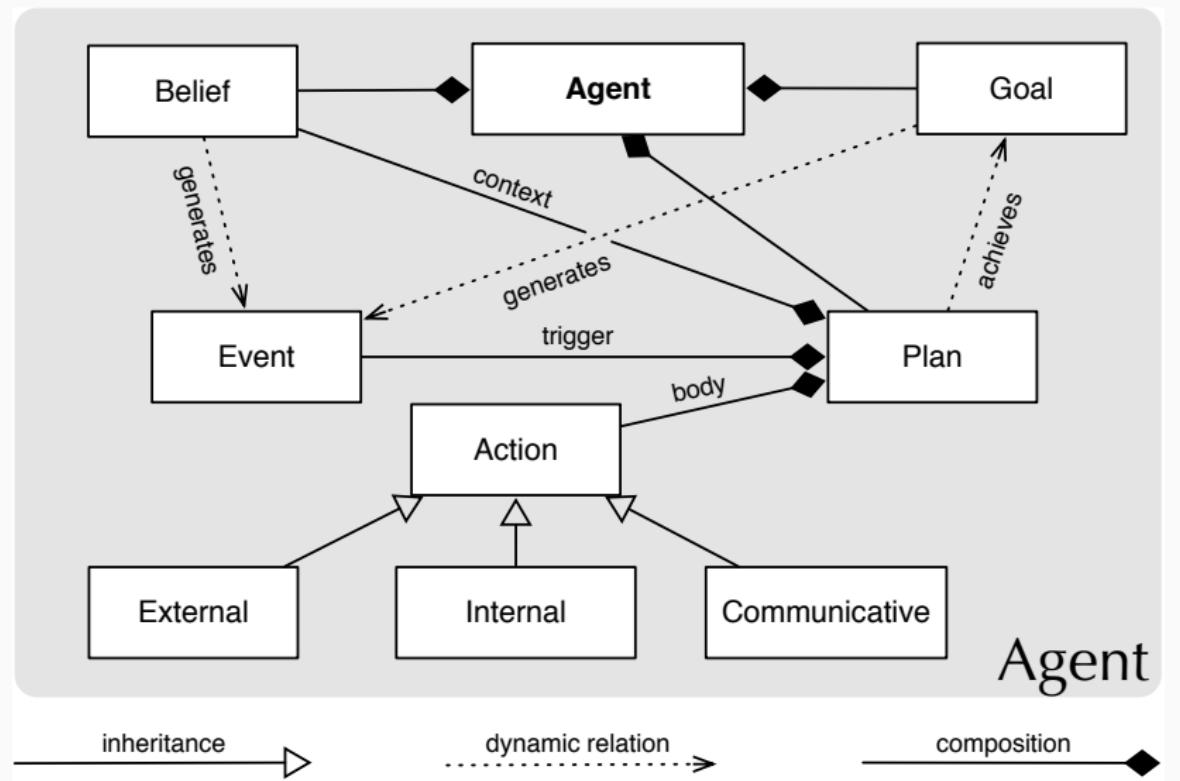
Protocol Implementation

2023-07-13

└ Protocol Implementation

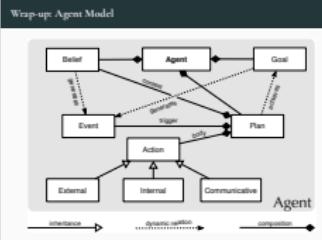
live coding

Wrap-up: Agent Model



2023-07-13

└ Wrap-up: Agent Model



Wrap-up: Agent Programming

- **AgentSpeak**
 - Logic + BDI
 - Agent programming language
- *Jason*
 - AgentSpeak interpreter
 - Implements the operational semantics of AgentSpeak
 - Speech-act based communication
 - Highly customisable
 - Useful tools
 - Open source

2023-07-13

└ Wrap-up: Agent Programming

- **AgentSpeak**
 - Logic + BDI
 - Agent programming language
- *Jason*
 - AgentSpeak interpreter
 - Implements the operational semantics of AgentSpeak
 - Speech-act based communication
 - Highly customisable
 - Useful tools
 - Open source

Fundamentals

2023-07-13

I do not plan to present all the following slides, I will select them regarding the interests of the audience

Literature

Books: [Bordini et al., 2005], [Bordini et al., 2009]

Proceedings: EMAS, ProMAS, DALT, LADS, AGERE, ...

Surveys: [Bordini et al., 2006], [Fisher et al., 2007] ...

Languages of historical importance: Agent0 [Shoham, 1993],

AgentSpeak(L) [Rao, 1996], MetateM [Fisher, 2005],

3APL [Hindriks et al., 1997],

Golog [Giacomo et al., 2000]

Other prominent languages:

Jason [Bordini et al., 2007], Jadex [Pokahr et al., 2005],

2APL [Dastani, 2008], GOAL [Hindriks, 2009],

JACK [Winikoff, 2005],

ASTRA, SARTL

But many others languages and platforms...

2023-07-13

└ Fundamentals
 └ Literature

Hard work have been done on this approach already.

Books: [Bordini et al., 2005], [Bordini et al., 2009]
 Proceedings: EMAS, ProMAS, DALT, LADS, AGERE, ...
 Surveys: [Bordini et al., 2006], [Fisher et al., 2007] ...
 Languages of historical importance: Agent0 [Shoham, 1993],
AgentSpeak(L) [Rao, 1996], MetateM [Fisher, 2005],
 3APL [Hindriks et al., 1997],
 Golog [Giacomo et al., 2000]
 Other prominent languages:
Jason [Bordini et al., 2007], Jadex [Pokahr et al., 2005],
 2APL [Dastani, 2008], GOAL [Hindriks, 2009],
 JACK [Winikoff, 2005],
 ASTRA, SARTL
 But many others languages and platforms...

Some Languages and Platforms

Jason (Hübner, Bordini, ...); 3APL and 2APL (Dastani, van Riemsdijk, Meyer, Hindriks, ...); Jadex (Braubach, Pokahr); MetateM (Fisher, Guidini, Hirsch, ...); ConGoLog (Lesperance, Levesque, ... / Boutilier – DTGolog); Teamcore/ MTDP (Milind Tambe, ...); IMPACT (Subrahmanian, Kraus, Dix, Eiter); CLAIM (Amal El Fallah-Seghrouchni, ...); GOAL (Hindriks); BRAHMS (Sierhuis, ...); SemantiCore (Blois, ...); STAPLE (Kumar, Cohen, Huber); Go! (Clark, McCabe); Bach (John Lloyd, ...); MINERVA (Leite, ...); SOCS (Torroni, Stathis, Toni, ...); FLUX (Thielscher); JIAC (Hirsch, ...); JADE (Agostino Poggi, ...); JACK (AOS); Agentis (Agentis Software); Jackdaw (Calico Jack); ASTRA (Rem Collier); SARL (Stephane Galland, Sebastian Rodriguez); *simpAL*, ALOO (Ricci, ...);

• • •

Jason (Hübner, Bordini, ...); 3APL and 2APL (Dastani, van Riemsdijk, Meyer, Hindriks, ...); Jadex (Braubach, Pokahr); MetateM (Fisher, Guidini, Hirsch, ...); ConGoLog (Lesperance, Levesque, ... / Boutilier – DTGolog); Teamcore/ MTDP (Milind Tambe, ...); IMPACT (Subrahmanian, Kraus, Dix, Eiter); CLAIM (Amal El Fallah-Seghrouchni, ...); GOAL (Hindriks); BRAHMS (Sierhuis, ...); SemantiCore (Blois, ...); STAPLE (Kumar, Cohen, Huber); Go! (Clark, McCabe); Bach (John Lloyd, ...); MINERVA (Leite, ...); SOCS (Torroni, Stathis, Toni, ...); FLUX (Thielscher); JIAC (Hirsch, ...); JADE (Agostino Poggi, ...); JACK (AOS); Agentis (Agentis Software); Jackdaw (Calico Jack); ASTRA (Rem Collier); SARL (Stephane Galland, Sebastian Rodriguez); *simpAL*, ALOO (Ricci, ...);

...

2023-07-13

Fundamentals

Some Languages and Platforms

some proposals are libraries/packages for existing languages
others are new languages

many agent languages have efficient and stable interpreters



Agent Oriented Programming — Inspiration

- Use of **mentalistic** notions and a **societal** view of computation
[Shoham, 1993]
- Heavily influenced by the **BDI** architecture and reactive planning systems [Bratman et al., 1988]

2023-07-13

└ Fundamentals

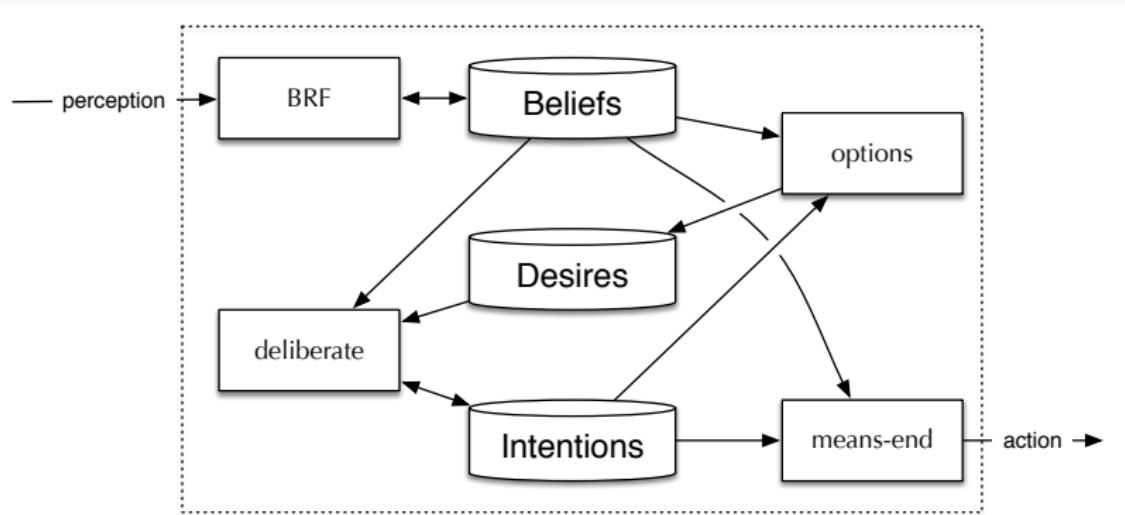
└ Agent Oriented Programming — Inspiration

I do recommend to read foundational papers like these from a Philosopher (trying to solve the problem for humans)

• Use of **mentalistic** notions and a **societal** view of computation
[Shoham, 1993]

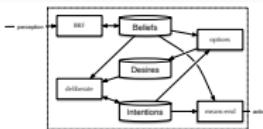
• Heavily influenced by the **BDI** architecture and reactive planning systems [Bratman et al., 1988]

BDI architecture



2023-07-13

└ Fundamentals
└ BDI architecture



mentalistic view (the behavior of the agent is explained in terms of its mental state: B, D, and I):

- B: beliefs (information) the agent has about its environment (updated by perception)
- D: what the agent wishes
- I: desires the agent has *committed* to (based on the current beliefs and other intentions)

Two main processes: deliberate: Desire → Intention means-ends: Intention → Actions

BDI explains the actions of the agent! (because the agent intents to, desires to, and believes it is feasible)

So next slides will highlight properties of the commitment (serious commitment but not too much)

BDI reasoning cycle [Wooldridge, 2009]

while true do

```
B ← brf(B, perception())           // belief revision
D ← options(B, I)                  // desire revision
I ← deliberate(B, D, I)            // get intentions
π ← meansend(B, I, A)              // gets a plan
while π ≠ ∅ do
    execute( head(π) )
    π ← tail(π)
```

2023-07-13

Fundamentals

BDI reasoning cycle [Wooldridge, 2009]

intentions are desire + commitment.

types of commitments: over commitment, Singel-Minded, there are good bibliography on that.

BDI reasoning cycle [Wooldridge, 2009]

```
while true do
    B ← brf(B, perception())
    D ← options(B, I)
    I ← deliberate(B, D, I)
    π ← meansend(B, I, A)
    while π ≠ ∅ do
        execute(head(π))
        π ← tail(π)
```

```
// belief revision
// desire revision
// get intentions
// gets a plan
```

BDI reasoning cycle [Wooldridge, 2009]

while true do

```
B ← brf(B, perception())           // belief revision
D ← options(B, I)                  // desire revision
I ← deliberate(B, D, I)            // get intentions
π ← meansend(B, I, A)              // gets a plan
while π ≠ ∅ do
    execute( head(π) )
    π ← tail(π)
```

fine for pro-activity, but not for reactivity (over commitment)

2023-07-13

Fundamentals

BDI reasoning cycle [Wooldridge, 2009]

BDI reasoning cycle [Wooldridge, 2009]

```
while true do
    B ← brf(B, perception())
    D ← options(B, I)
    I ← deliberate(B, D, I)
    π ← meansend(B, I, A)
    while π ≠ ∅ do
        execute(head(π))
        π ← tail(π)
```

fine for pro-activity, but not for reactivity (over commitment)

intentions are desire + commitment.

types of commitments: over commitment, Singel-Minded, there are good bibliography on that.

BDI reasoning cycle [Wooldridge, 2009]

while true do

```
B ← brf(B, perception())           // belief revision
D ← options(B, I)                  // desire revision
I ← deliberate(B, D, I)            // get intentions
π ← meansend(B, I, A)              // gets a plan

while π ≠ ∅ do
    execute( head(π) )
    π ← tail(π)
    B ← brf(B, perception())
    if ¬sound(π, I, B) then
        π ← meansend(B, I, A)
```

revise commitment to plan – re-planning for context adaptation

2023-07-13

Fundamentals

BDI reasoning cycle [Wooldridge, 2009]

intentions are desire + commitment.

types of commitments: over commitment, Singel-Minded, there are good bibliography on that.

BDI reasoning cycle [Wooldridge, 2009]

```
while true do
    B ← brf(B, perception())
    D ← options(B, I)
    I ← deliberate(B, D, I)
    π ← meansend(B, I, A)
    while π ≠ ∅ do
        execute( head(π) )
        π ← tail(π)
        B ← brf(B, perception())
        if ¬sound(π, I, B) then
            π ← meansend(B, I, A)

revise commitment to plan – re-planning for context adaptation
```



BDI reasoning cycle [Wooldridge, 2009]

while true do

```
B ← brf(B, perception())           // belief revision
D ← options(B, I)                   // desire revision
I ← deliberate(B, D, I)             // get intentions
π ← meansend(B, I, A)               // gets a plan
while π ≠ ∅ and ¬succeeded(I, B) and ¬impossible(I, B) do
    execute( head(π) )
    π ← tail(π)
    B ← brf(B, perception())
    if ¬sound(π, I, B) then
        π ← meansend(B, I, A)
```

revise commitment to intentions – Single-Minded Commitment

2023-07-13

Fundamentals

BDI reasoning cycle [Wooldridge, 2009]

intentions are desire + commitment.

types of commitments: over commitment, Singel-Minded, there are good bibliography on that.

BDI reasoning cycle [Wooldridge, 2009]

```
while true do
    B ← brf(B, perception())           // belief revision
    D ← options(B, I)                 // desire revision
    I ← deliberate(B, D, I)            // get intentions
    π ← meansend(B, I, A)              // gets a plan
    while π ≠ ∅ and ¬succeeded(I, B) and ¬impossible(I, B) do
        execute( head(π) )
        π ← tail(π)
        B ← brf(B, perception())
        if ¬sound(π, I, B) then
            π ← meansend(B, I, A)
    end
    revise commitment to intentions – Single-Minded Commitment
```



BDI reasoning cycle [Wooldridge, 2009]

while true do

```
B ← brf(B, perception())           // belief revision
D ← options(B, I)                   // desire revision
I ← deliberate(B, D, I)             // get intentions
π ← meansend(B, I, A)               // gets a plan
while π ≠ ∅ and ¬succeeded(I, B) and ¬impossible(I, B) do
    execute( head(π) )
    π ← tail(π)
    B ← brf(B, perception())
    if reconsider(I, B) then
        D ← options(B, I)
        I ← deliberation(B, D, I)
        if ¬sound(π, I, B) then
            π ← meansend(B, I, A)
```

2023-07-13

Fundamentals

BDI reasoning cycle [Wooldridge, 2009]

intentions are desire + commitment.

types of commitments: over commitment, Singel-Minded, there are good bibliography on that.

BDI reasoning cycle [Wooldridge, 2009]

```
while true do
    B ← brf(B, perception())           // belief revision
    D ← options(B, I)                 // desire revision
    I ← deliberate(B, D, I)             // get intentions
    π ← meansend(B, I, A)               // gets a plan
    while π ≠ ∅ and ¬succeeded(I, B) and ¬impossible(I, B) do
        execute( head(π) )
        π ← tail(π)
    B ← brf(B, perception())
    if reconsider(I, B) then
        D ← options(B, I)
        I ← deliberation(B, D, I)
        if ¬sound(π, I, B) then
            π ← meansend(B, I, A)
    reconsider the intentions (not always!)
```

- Intentions pose problems for the agents: they need to determine a way to achieve them (planning and acting)
- Intentions provide a “screen of admissibility” for adopting new intentions
- Agents keep tracking their success of attempting to achieve their intentions
- Agents should not spend all their time revising intentions (losing pro-activity and reactivity)

2023-07-13

Fundamentals

Intentions

- Intentions pose problems for the agents: they need to determine a way to achieve them (planning and acting)
- Intentions provide a “screen of admissibility” for adopting new intentions
- Agents keep tracking their success of attempting to achieve their intentions
- Agents should not spend all their time revising intentions (losing pro-activity and reactivity)

(BDI & Jason) Hello World – agent bob

```
happy(bob).                                // B
!say(hello).                               // D
+!say(X) : happy(bob)                     // I
<- .print(X).
```

2023-07-13

Fundamentals

(BDI & Jason) Hello World – agent bob

- how does it look like? (comparing with other languages like C, Java,)
- jason uses procedural goals (goals to do) and not declarative goals (goals to be), as 2APL. it comes from the original PRS inspiration where we are specifying behaviour instead of (env) states [Georgeff and Lansky, 1987]
- plans are not prolog (theoretical reasoning), they are for practical reasoning.
- the language gives constructors to program BDI with the required features shown in the Woodridge algorithm

```
happy(bob).                                // B
!say(hello).                               // D
+!say(X) : happy(bob)
<- .print(X).
```



```
happy(bob).                                // B
!say(hello).                               // D
+!say(X) : happy(bob)                      // I
<- .print(X).
```

beliefs

- prolog like (FOL)

2023-07-13

Fundamentals

(BDI & Jason) Hello World – agent bob

```
happy(bob).                                // B
!say(hello).                               // D
+!say(X) : happy(bob)
<- .print(X).                            // I
```

- how does it look like? (comparing with other languages like C, Java,)
- jason uses procedural goals (goals to do) and not declarative goals (goals to be), as 2APL. it comes from the original PRS inspiration where we are specifying behaviour instead of (env) states [Georgeff and Lansky, 1987]
- plans are not prolog (theoretical reasoning), they are for practical reasoning.
- the language gives constructors to program BDI with the required features shown in the Woodridge algorithm



(BDI & Jason) Hello World – agent bob

happy(bob).

!say(hello).

+!say(*X*) : happy(bob)

<- .print(*X*).

desires

- prolog like
- with ! prefix

// B

// D

// I

2023-07-13

Fundamentals

(BDI & Jason) Hello World – agent bob

- how does it look like? (comparing with other languages like C, Java,)
- jason uses procedural goals (goals to do) and not declarative goals (goals to be), as 2APL. it comes from the original PRS inspiration where we are specifying behaviour instead of (env) states [Georgeff and Lansky, 1987]
- plans are not prolog (theoretical reasoning), they are for practical reasoning.
- the language gives constructors to program BDI with the required features shown in the Woodridge algorithm

```
happy(bob).
!say(hello).
+!say(X) : happy(bob)
<- .print(X).
```

// B

// D

// I



(BDI & Jason) Hello World – agent bob

`happy(bob).`

`!say(hello).`

`+!say(X) : happy(bob)`

`<- .print(X).`

plans

- define when a desire becomes an intention
~~ deliberate
- how it is satisfied
- are used for practical reasoning
~~ means-end

2023-07-13

Fundamentals

(BDI & Jason) Hello World – agent bob

- how does it look like? (comparing with other languages like C, Java,)
- jason uses procedural goals (goals to do) and not declarative goals (goals to be), as 2APL. it comes from the original PRS inspiration where we are specifying behaviour instead of (env) states [Georgeff and Lansky, 1987]
- plans are not prolog (theoretical reasoning), they are for practical reasoning.
- the language gives constructors to program BDI with the required features shown in the Woodridge algorithm

```
happy(bob).  
!say(hello).  
+!say(X) : happy(bob)  
<- .print(X).
```

// 0

// 0

// 1



BDI Hello World — desires from perception (*options*)

```
+happy(bob) <- !say(hello).
```

```
+!say(X) : not today(monday)  
<- .print(X).
```

2023-07-13

Fundamentals

BDI Hello World — desires from perception (*options*)

desire via perception, the agent starts believing it is happy and then creates a new desire

```
+happy(bob) <- !say(hello).  
+!say(X) : not today(monday)  
<- .print(X).
```



```
+happy(bob) [source(A)]  
  : someone_who_knows_me_very_well(A)  
  <- !say(hello).  
  
+!say(X) : not today(monday) <- .print(X).
```

2023-07-13

└ Fundamentals
 └ BDI Hello World — source of beliefs

considers the source of the belief

```
+happy(bob) [source(A)]  
  : someone_who_knows_me_very_well(A)  
  <- !say(hello).  
  
+!say(X) : not today(monday) <- .print(X).
```

```
+happy(H) [source(A)]
```

```
: sincere(A) & .my_name(H)  
<- !say(hello).
```

```
+happy(H)
```

```
: not .my_name(H)  
<- !say(i_envy(H)).
```

```
+!say(X) : not today(monday) <- .print(X).
```

2023-07-13

Fundamentals

BDI Hello World — plan selection

```
+happy(H) [source(A)]  
: sincere(A) & .my_name(H)  
<- !say(hello).
```



```
+happy(H)  
: not .my_name(H)  
<- !say(i_envy(H)).
```



```
+!say(X) : not today(monday) <- .print(X).
```

the agent selects the plan that is more suitable for the current circumstance. plan context is used for that.

whenever (trigger event)

- I start to believe that H is happy, from a source A and (context)

- I belief that this source is sincere

- and H is “me” (I am happy because A said so)
then (body)

- create a new desire to say hello

whenever I have the desire to say something, and it is not monday, commit to that desire and use the body of the plan to fullfil it.

BDI Hello World — intention revision

```
+happy(H) [source(A)]
```

```
: sincere(A) & .my_name(H)  
<- !say(hello).
```

```
+happy(H)
```

```
: not .my_name(H)  
<- !say(i_envy(H)).
```

```
+!say(X) : not today(monday) <- .print(X); !say(X).
```

2023-07-13

Fundamentals

BDI Hello World — intention revision

```
+happy(H) [source(A)]  
: sincere(A) & .my_name(H)  
<- !say(hello).
```



```
+happy(H)  
: not .my_name(H)  
<- !say(i_envy(H)).
```



```
+say(X) : not today(monday) <- .print(X); !say(X).
```

BDI Hello World — intention revision

+happy(H) [source(A)]

```
: sincere( $A$ ) & .my_name( $H$ )
<- !say(hello).
```

+happy(H)

```
: not .my_name( $H$ )
<- !say(i_envy( $H$ )).
```

+!say(X) : not today(monday) <- .print(X); !say(X).

-happy(H)

```
: .my_name( $H$ )
<- .drop_intention(say(hello)).
```

2023-07-13

Fundamentals

BDI Hello World — intention revision

```
+happy( $H$ ) [source( $A$ )]
: sincere( $A$ ) & .my_name( $H$ )
<- !say(hello).

+happy( $H$ )
: not .my_name( $H$ )
<- !say(i_envy( $H$ )).

+!say( $X$ ) : not today(monday) <- .print( $X$ ); !say( $X$ ).

-happy( $H$ )
: .my_name( $H$ )
<- .drop_intention(say(hello)).
```

features

```
+happy(H) [source(A)]
  : sincere(A) & .my_name(H)
```

```
<- !say(hello).
```

```
+happy(H)
```

```
  : not .my_name(H)
  <- !say(i_envy(H)).
```

```
+!say(X) : not today(monday)
```

```
-happy(H)
```

```
  : .my_name(H)
  <- .drop_intention(say(hello)).
```

- we can have several intentions based on the same plans
~~> running concurrently
- long term goals running
~~> reaction meanwhile
~~> not overcommitted
- plan selection based on circumstance
- sequence of actions (partially)
computed by the interpreter
~~> programmer declares plans

2023-07-13

Fundamentals

└ BDI Hello World — intention revision

```
+happy(H) [source(A)]
  : sincere(A) & .my_name(H)
  <- !say(hello).

+happy(H)
  : not .my_name(H)
  <- !say(i_envy(H)).

+say(X) : not today(monday) <- .print(X); !say(X).

-happy(H)
  : .my_name(H)
  <- .drop_intention(say(hello)).
```

Jason

2023-07-13

Jason

Jason

AgentSpeak: The foundational language for *Jason*

- Programming language for BDI agents
- Originally proposed by Rao [Rao, 1996]
- 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

2023-07-13

└ *Jason*

 └ AgentSpeak: The foundational language for *Jason*

- Programming language for BDI agents
- Originally proposed by Rao [Rao, 1996]
- 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 Jomi F. Hübner, Rafael H. Bordini, and others

2023-07-13

└ Jason
 └ Jason: A practical implementation of AgentSpeak

agent dimension in JaCaMo

- 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 Jomi F. Hübner, Rafael H. Bordini, and others



Main Language Constructs

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

2023-07-13

Jason
└ Main Language Constructs

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

Beliefs — Representation

Syntax

Beliefs are represented by annotated literals of first order logic

functor(*term*₁, ..., *term*_{*n*}) [*annot*₁, ..., *annot*_{*m*}]

Example (belief base of agent Tom)

```
red(box1) [source(percept)] .  
friend(bob,alice) [source(bob)] .  
liar(alice) [source(self),source(bob)] .  
~liar(bob) [source(self)] .
```

2023-07-13

Jason

Beliefs — Representation

Syntax

Beliefs are represented by annotated literals of first order logic

functor(*term*₁, ..., *term*_{*n*}) [*annot*₁, ..., *annot*_{*m*}]

Example (belief base of agent Tom)

`red(box1) [source(percept)] .`

`friend(bob,alice) [source(bob)] .`

`liar(alice) [source(self),source(bob)] .`

`~liar(bob) [source(self)] .`

annotations is a set of terms with special unification — not available in Prolog

by perception

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

by intention

the **plan operators** + and - can be used to add and remove beliefs annotated with `source(self)` (**mental notes**)

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

2023-07-13

Jason
└ Beliefs — Dynamics

Beliefs — Dynamics i

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

by intention
the plan operators + and - can be used to add and remove beliefs annotated with `source(self)` (mental notes)

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

2023-07-13

Jason

Beliefs — Dynamics

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
```

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).

2023-07-13

└ Jason
 └ Goals — Representation

jason uses procedural goals (goals to do) and not declarative goals (goals to be, as in planning, 2APL, ...).

it comes from the original PRS inspiration where we are specifying behaviour instead of (env) states [Georgeff and Lansky, 1987]

PRS also proposes maintenance goal, that is available in Jason(ER)

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).

by intention
 the plan 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);
...
```

Goals — Dynamics i

by intention

the plan 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);
```

...

2023-07-13

Jason
 └─Goals — Dynamics



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,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
```

2023-07-13

Jason
Goals — Dynamics

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
```

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,published(P),Answer); // sent by Bob
// adds new goal ?publisher(P)[source(bob)] for Tom
// the response of Tom unifies with Answer
```

2023-07-13

Jason
└ Goals — Dynamics

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 unifies with Answer
```

Triggering Events — Representation

- Events happen as consequence to changes in the agent's beliefs or goals
- An agent reacts to events by executing **plans**
- Types of **plan triggering events**

+b (belief addition)

-b (belief deletion)

+!g (achievement-goal addition)

-!g (achievement-goal deletion)

+?g (test-goal addition)

-?g (test-goal deletion)

2023-07-13

Jason

Triggering Events — Representation

- Events happen as consequence to changes in the agent's beliefs or goals
- An agent reacts to events by executing plans
- Types of plan triggering events
 - +b** (belief addition)
 - b** (belief deletion)
 - +!g** (achievement-goal addition)
 - !g** (achievement-goal deletion)
 - +?g** (test-goal addition)
 - ?g** (test-goal deletion)

Plans — Representation

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

2023-07-13

Jason
└ Plans — Representation

Plans — Representation

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

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)

2023-07-13

Jason

Plans — Operators for Plan Context

Boolean operators	Arithmetic operators
& (and)	+ (sum)
 (or)	- (subtraction)
not (not)	* (multiplication)
= (unification)	/ (divide)
>, >= (relational)	div (divide – integer)
<, <= (relational)	mod (remainder)
== (equals)	** (power)
\ == (different)	



Plans — Operators for Plan Body

```
+rain : time_to_leave(T) & clock.now(H) & H >= T
  <- !g1;          // new sub-goal
  !!g2;           // new goal
  ?b(X);          // new test goal
  +b1(T-H);       // add mental note
  -b2(T-H);       // remove mental note
  -+b3(T*H);      // update mental note
  jia.get(X);     // internal action
  X > 10;          // constraint to carry on
  close(door);    // external action
  !g3[hard_deadline(3000)]. // goal with deadline
```

2023-07-13

Jason

Plans — Operators for Plan Body

Plans — Operators for Plan Body

```
+rain : time_to_leave(T) & clock.now(H) & H >= T
  <- !g1;          // new sub-goal
  !!g2;           // new goal
  ?b(X);          // new test goal
  +b1(T-H);       // add mental note
  -b2(T-H);       // remove mental note
  -+b3(T*H);      // update mental note
  jia.get(X);     // internal action
  X > 10;          // constraint to carry on
  close(door);    // external action
  !g3[hard_deadline(3000)]. // goal with deadline
```

The plans that form the plan library of the agent come from

- initial plans defined by the programmer
- plans added dynamically and intentionally by
 - `.add_plan`
 - `.remove_plan`
- plans received from
 - `tellHow` messages
 - `untellHow`

2023-07-13

Jason
└ Plans — Dynamics

The plans that form the plan library of the agent come from

- initial plans defined by the programmer
- plans added dynamically and intentionally by
 - `.add_plan`
 - `.remove_plan`
- plans received from
 - `tellHow` messages
 - `untellHow`

A note about “Control”

Agents can control (manipulate) their own (and influence the others)

- beliefs
- goals
- plan

By doing so they control their behaviour

The developer provides initial values of these elements and thus also influence the behaviour of the agent

2023-07-13

Jason

└ A note about “Control”

Agents can control (manipulate) their own (and influence the others)

- beliefs
- goals
- plan

By doing so they control their behaviour

The developer provides initial values of these elements and thus also influence the behaviour of the agent

Reasoning Cycle

2023-07-13

Jason

Reasoning Cycle

Runtime Structures for the Reasoning Cycle

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 consequence to changes in the
agent's beliefs or goals

Intentions: plans instantiated to achieve some goal

2023-07-13

Jason

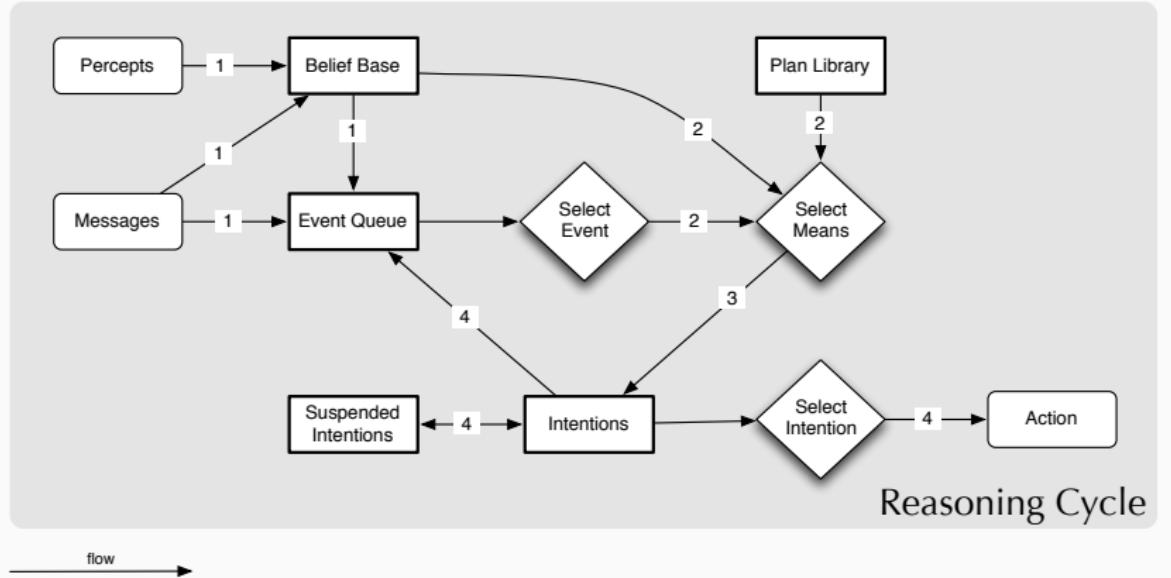
Runtime Structures for the Reasoning Cycle

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 consequence to changes in the agent's beliefs or goals

The former three come from the agent program (syntax), the latter two exist at runtime to support the interpretation (semantics) of Jason



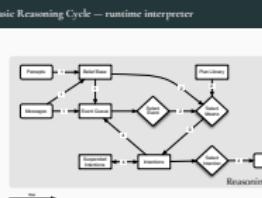
Basic Reasoning Cycle — runtime interpreter



2023-07-13

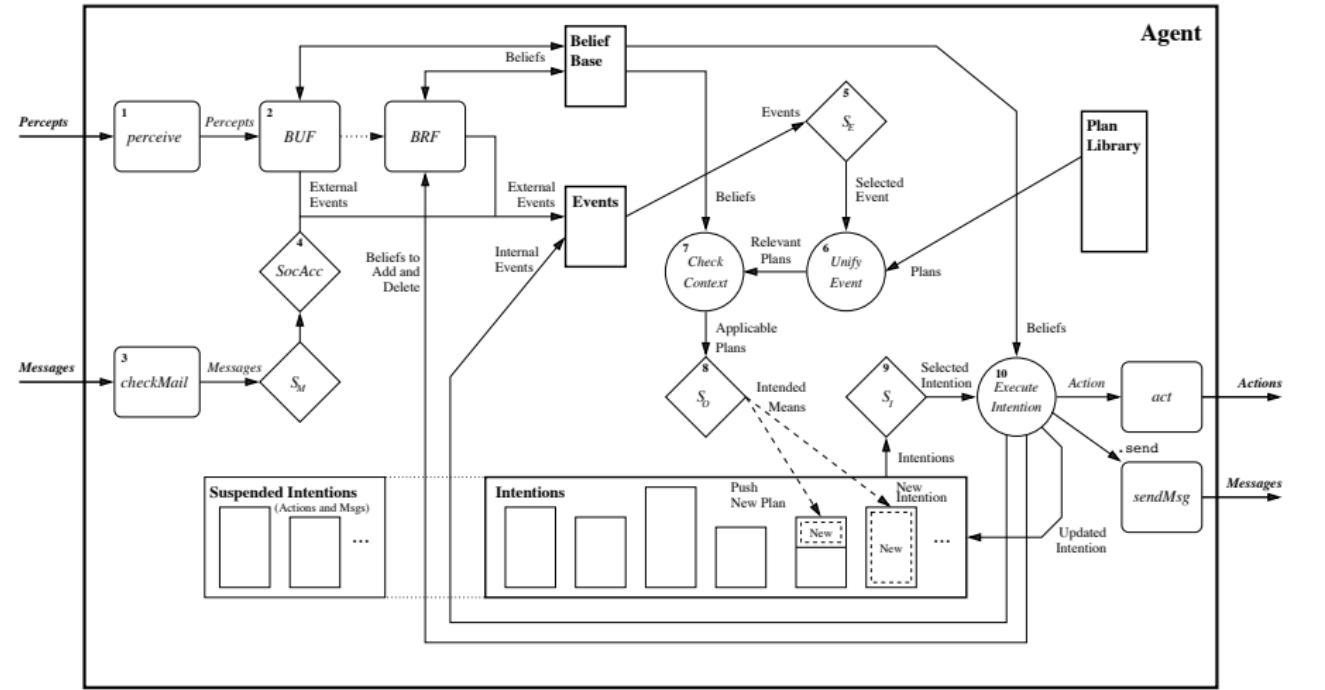
Jason

Basic Reasoning Cycle — runtime interpreter



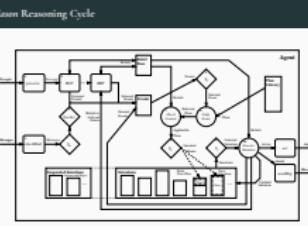
- 1 perceive the environment and update belief base
- 1 process new messages
- 1 select event
- 2 select relevant plans
- 2 select applicable plans
- 3 create/update intention
- 4 select intention to execute
- 4 execute one step of the selected intention

Jason Reasoning Cycle

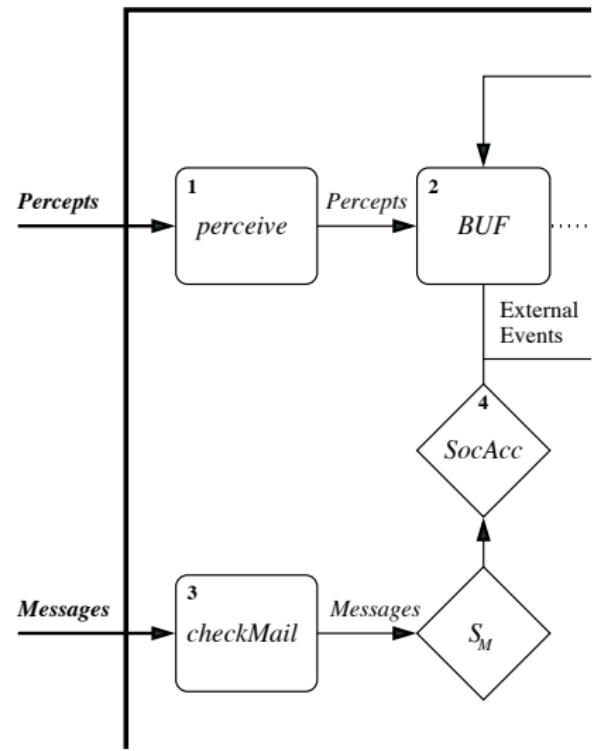


2023-07-13

Jason
└ Jason Reasoning Cycle



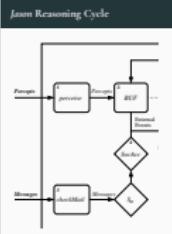
Jason Reasoning Cycle



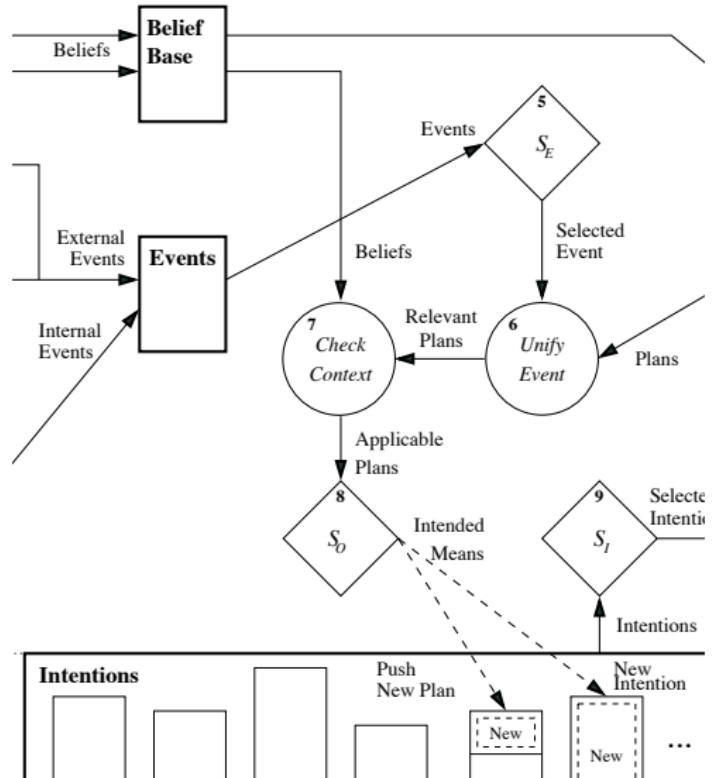
- machine perception
- belief revision
- knowledge representation
- communication, argumentation
- trust
- social power

2023-07-13

Jason
└ Jason Reasoning Cycle



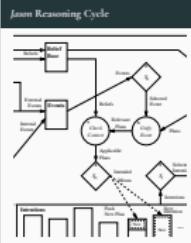
Jason Reasoning Cycle

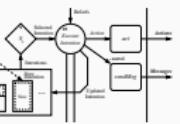


- planning
- reasoning
- decision theoretic techniques
- learning (reinforcement)

2023-07-13

└ Jason
 └ Jason Reasoning Cycle



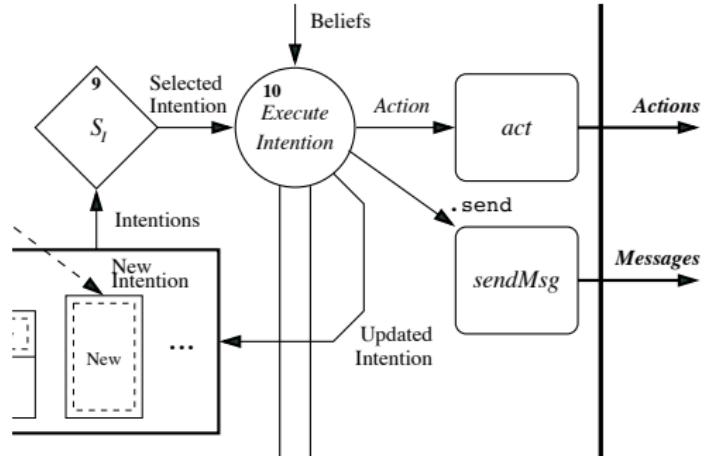


Jason
└ Jason Reasoning Cycle

2023-07-13

- intention reconsideration
- scheduling
- action theories

Jason Reasoning Cycle



Other Features

2023-07-13

Jason

Other Features

Failure Handling: Contingency Plans

Example (an agent blindly committed to g)

```
+!g : g. // g is a declarative goal
```

```
+!g : ... <- a1; ?g.
```

```
+!g : ... <- a2; ?g.
```

```
+!g : ... <- a3; ?g.
```

```
+!g <- !g. // keep trying
```

```
-!g <- !g. // in case of some failure
```

```
+g <- .succeed_goal(g).
```

2023-07-13

Jason

Failure Handling: Contingency Plans

Failure Handling: Contingency Plans

Example (an agent blindly committed to g)

```
+!g : g. // g is a declarative goal
```

```
+!g : ... <- a1; ?g.
```

```
+!g : ... <- a2; ?g.
```

```
+!g : ... <- a3; ?g.
```

```
+!g <- !g. // keep trying
```

```
-!g <- !g. // in case of some failure
```

```
+g <- .succeed_goal(g).
```

Failure Handling: Contingency Plans

Example (single minded commitment)

```
+!g : g. // g is a declarative goal
```

```
+!g : ... <- a1; ?g.
```

```
+!g : ... <- a2; ?g.
```

```
+!g : ... <- a3; ?g.
```

```
+!g <- !g. // keep trying
```

```
-!g <- !g. // in case of some failure
```

```
+g <- .succeed_goal(g).
```

```
+f : .super_goal(g, SG) <- .fail_goal(SG).
```

f is the drop condition for goal g

2023-07-13

Jason

Failure Handling: Contingency Plans

Example (single minded commitment)

```
+!g : g. // g is a declarative goal
```

```
+!g : ... <- a1; ?g.
```

```
+!g : ... <- a2; ?g.
```

```
+!g : ... <- a3; ?g.
```

```
+!g <- !g. // keep trying
```

```
-!g <- !g. // in case of some failure
```

```
+g <- .succeed_goal(g).
```

```
+f : .super_goal(g, SG) <- .fail_goal(SG).
```

f is the drop condition for goal g

Compiler pre-processing – directives

Example (single minded commitment)

```
{ begin smc(g,f) }
    +!g : ... <- a1.
    +!g : ... <- a2.
    +!g : ... <- a3.
{ end }
```

2023-07-13

└ Jason

└ Compiler pre-processing – directives

Example (single minded commitment)

```
( begin smc(g,f) )
    +!g : ... <- a1.
    +!g : ... <- a2.
    +!g : ... <- a3.
( end )
```

Meta Programming

Example (an agent that asks for plans *on demand*)

```
- !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

2023-07-13

Jason

└ Meta Programming

Example (an agent that asks for plans *on demand*)

```
- !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

Other Language Features: Strong Negation

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

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

2023-07-13

Jason

└─ Other Language Features: Strong Negation

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

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

Prolog-like Rules in the Belief Base

```
tall(X) :- woman(X) & height(X, H) & H > 1.70.  
tall(X) :- man(X) & height(X, H) & H > 1.80.
```

2023-07-13

Jason

└ Prolog-like Rules in the Belief Base

```
tall(X) :- woman(X) & height(X, H) & H > 1.70.  
tall(X) :- man(X) & height(X, H) & H > 1.80.
```

Internal Actions

- Unlike actions, internal actions do not change the environment
- They are 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(...)`

2023-07-13

Jason
└ Internal Actions

- Unlike actions, internal actions do not change the environment
- They are 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

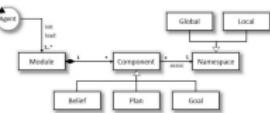
- Standard (pre-defined) internal actions have an empty library name
 - `.print(term1, term2, ...)`
 - `.union(list1, list2, list3)`
 - `.my_name(var)`
 - `.send(ag, perf, literal)`
 - `.intend(literal)`
 - `.drop_intention(literal)`
- Many others available for: printing, sorting, list/string operations, manipulating the beliefs/annotations/plan library, creating agents, waiting/generating events, etc.

2023-07-13

Jason
└ Standard Internal Actions

Standard Internal Actions

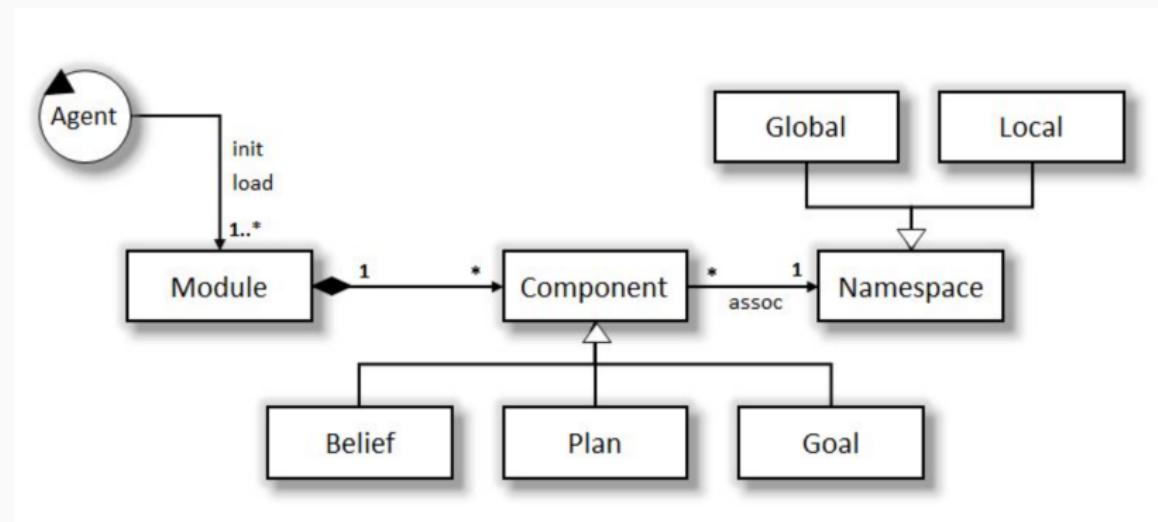
- Standard (pre-defined) internal actions have an empty library name
 - `.print(term1, term2, ...)`
 - `.union(list1, list2, list3)`
 - `.my_name(var)`
 - `.send(ag, perf, literal)`
 - `.intend(literal)`
 - `.drop_intention(literal)`
- Many others available for: printing, sorting, list/string operations, manipulating the beliefs/annotations/plan library, creating agents, waiting/generating events, etc.



Namespaces & Modularity

2023-07-13

Jason
Namespaces & Modularity



Namespaces & Modularity

Inspection of agent alice

- Beliefs

```
{include("initiator.asl", pc)}
{include("initiator.asl", tv)}

!pc:::startCNP(fix(pc)).
!tv:::startCNP(fix(tv)).

+pc:::winner(X)
<- .print(X).
```

tv::

```
introduction(participant)[source(comparison)]
propose(11.075337225252543)[source(self)]
propose(12.043311087442898)[source(self)]
propose(12.81277904935436)[source(self)]
winner(company_A1)[source(self)].
```

#8priv::

```
state(finished)[source(self)].
```

pc::

```
introduction(participant)[source(comparison)]
propose(11.389500048463455)[source(self)]
propose(11.392553683771682)[source(self)]
propose(12.348901000262853)[source(self)]
winner(company_A2)[source(self)].
```

2023-07-13

Jason

Namespaces & Modularity

```
(include("initiator.asl", pc)
(include("initiator.asl", tv))
!pc:::startCNP(fix(pc)).
!tv:::startCNP(fix(tv)).

+pc:::winner(X)
<- .print(X).
```

Concurrent Plans

```
+!ga <- ...; !gb; ...
```

```
+!gb <- ...; !g1 |&| !g2; a1; ...
```

```
+!ga <- ...; !gb; ...
```

```
+!gb <- ...; !g1 ||| !g2; a1; ...
```

```
+!g <- x; (a;b) |&| (c;d) ||| (e;f); y.
```

2023-07-13

Jason

Concurrent Plans

```
+!ga <- ...; !gb; ...
```

```
+!gb <- ...; !g1 |&| !g2; a1; ...
```

```
+!ga <- ...; !gb; ...
```

```
+!gb <- ...; !g1 ||| !g2; a1; ...
```

```
+!g <- x; (a;b) |&| (c;d) ||| (e;f); y.
```

Jason(ER) — motivation

```
+cfp(Id,Task) [source(A)] // answer to Call For Proposal
  : price(Task,Offer) & not my_offer(Task)
  <- +offered(Task);
    .send(A,tell,propose(Id,Offer)).  

+cfp(Id,_) [source(A)]
  <- .send(A,tell,refuse(Id)).  

+accept_proposal(Id) : my_offer(Task)
  <- !do(Task);
    -my_offer(Task).  

+reject_proposal(Id) : my_offer(Task)
  <- -my_offer(Task).
```

2023-07-13

└ Jason
 └ Jason(ER) — motivation

- what is the goal related to the action .send(A,tell,propose(Id,Offer))
- when executing actions for goal “do”, if we ask “why” we can track back to accepted_proposal, but not to the cfp or even some implicit goal that is “participate in CNP”
- some jason intentions have no explicit goal
- no explicit causal link among plans

```
cfp(Id,Task) [source(A)] // answer to Call For Proposal
  : price(Task,Offer) & not my_offer(Task)
  <- +offered(Task);
    .send(A,tell,propose(Id,Offer));
+cfp(Id,_) [source(A)]
  <- .send(A,tell,refuse(Id)).  

+accept_proposal(Id) : my_offer(Task)
  <- !do(Task);
    -my_offer(Task).  

+reject_proposal(Id) : my_offer(Task)
  <- -my_offer(Task).
```

Scope & sub-plans & goal conditions

```
+!g(X) : c <: gc <- a1; !g1.  
{
```

```
+e : c1 <- a2(X).
```

```
+!g1 ....
```

```
}
```

2023-07-13

Jason

Scope & sub-plans & goal conditions

```
+!g(X) : c <: gc <- a1; !g1.  
(  
  +e : c1 <- a2(X).  
  +!g1 ....  
)
```

- main objective: all behaviour is the result of an (explicit) goal (not a Jason intention, that can have no explicit goal)
- new syntax: goal condition after <: and sub-plans enclosed by { and }
- relevant event are defined by the current intentions
- event +e is relevant only the agent intends g
- relevant plans are defined by the scope of some goal
- plan for g1 is visible only in scope of g
- g is dropped only when gc is true: maintenance goal
- variables have a broader scope (X is visible in sub-plans)

Example of a participant in a CNP

```
+!participate_cnp <: false. {
    +cfp(Id,Task)[source(A)] // answer to Call For Proposal
        : price(Task,Offer) & not my_offer(Task,_)
        <: false
        <- +my_offer(Task, Offer); .send(A,tell,propose(Id,Offer))
    {
        +accept_proposal(Id) <- !do(Task); -my_offer(Task,_);

        +reject_proposal(Id) <- -my_offer(Task,_); .done.
    }
    +cfp(Id,_)[source(A)] <- .send(A,tell,refuse(Id));
}

+!do(T) <- ...
```

2023-07-13

Jason

Example of a participant in a CNP

- the intention for +participate_cfp never finishes
- e-plan for +cfp is triggered only if the agent has goal participate_cnp
- the progression of the intention due to +cfp is finished only by .done, since the goal condition ('false') will never hold
- the e-plans enclosed by { and } are relevant only while the progression for +cfp is “running”
- consider that some progress in the intention is created from cfp(10,"banana"), only events accept_proposal(10) and reject_proposal(10) are relevant to trigger the subplans.
- enforce that every behaviour is due to a goal

Example of a participant in a CNP

```
+!participate_cnp <: false. (
    +cfp(Id,Task)[source(A)] // answer to Call For Proposal
        : price(Task,Offer) & not my_offer(Task,_)
        <: false
        <- my_offer(Task, Offer); .send(A,tell,propose(Id,Offer));
        +accept_proposal(Id) <- !do(Task); -my_offer(Task,_);
        +reject_proposal(Id) <- -my_offer(Task,_); .done.
    )
    +cfp(Id,_)[source(A)] <- .send(A,tell,refuse(Id));
)

+!do(T) <- ...
```

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 available with *Jason*: persistent belief base (in text files, in data bases, ...)

2023-07-13

└ Jason
 └ Jason Customisations

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 available with *Jason*: persistent belief base (in text files, in data bases, ...)

Consider a very simple robot with two goals:

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

2023-07-13

└ Comparison with other paradigms
 └ Jason × Java

Consider a very simple robot with two goals:

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

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

Java code – go to gold

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

2023-07-13

└ Comparison with other paradigms
 └ Java code – go to gold

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

Java code – charge battery

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

2023-07-13

└ Comparison with other paradigms
 └ Java code – charge battery

Jason code

```
direction(gold)    :- see(gold).
```

```
direction(random) :- not see(gold).
```

```
+!find(gold)           // long term goal
```

```
<- ?direction(A);
```

```
go(A);
```

```
!find(gold).
```

```
+battery(low)         // reactivity
```

```
<- !charge.
```

```
^!charge[state(executing)] // goal meta-events
```

```
<- .suspend(find(gold)).
```

```
^!charge[state(finished)]
```

```
<- .resume(find(gold)).
```



2023-07-13

└ Comparison with other paradigms
 └ Jason code

```
Jason code
```

```
direction(gold)    :- see(gold).  
direction(random) :- not see(gold).
```

```
+!find(gold)           // long term goal
```

```
<- ?direction(A);
```

```
go(A);
```

```
!find(gold).
```

```
+battery(low)          // reactivity
```

```
<- !charge.
```

```
^!charge[state(executing)] // goal meta-events
```

```
<- .suspend(find(gold)).
```

```
^!charge[state(finished)]
```

```
<- .resume(find(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

2023-07-13

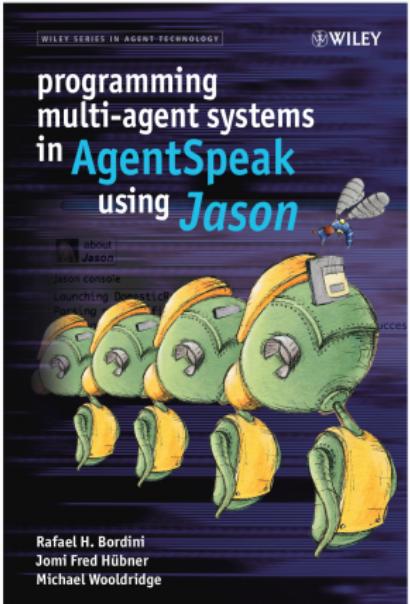
└ Comparison with other paradigms └ *Jason* × Prolog

- 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



Further Resources

- <http://jason.sourceforge.net>
- R.H. Bordini, J.F. Hübner, and
M. Wooldridge
**Programming Multi-Agent Systems in
AgentSpeak using Jason**
John Wiley & Sons, 2007.



2023-07-13

- └ Comparison with other paradigms
 - └ Further Resources

Besides the JaCaMo book (which has chapters dedicated to the agent dimension), the Jason book is all focused on this dimension

Bibliography i

 Austin, J. L. (1975).

How to do things with Words.

Harvard University Press, Cambridge, 2 edition.

 Bordini, R. H., Braubach, L., Dastani, M., Fallah-Seghrouchni, A. E., Gómez-Sanz, J. J., Leite, J., O'Hare, G. M. P., Pokahr, A., and Ricci, A. (2006).

A survey of programming languages and platforms for multi-agent systems.

Informatica (Slovenia), 30(1):33–44.

2023-07-13

└ Comparison with other paradigms
 └ Bibliography

 Austin, J. L. (1975).
How to do things with Words.

Harvard University Press, Cambridge, 2 edition.

 Bordini, R. H., Braubach, L., Dastani, M., Fallah-Seghrouchni, A. E., Gómez-Sanz, J. J., Leite, J., O'Hare, G. M. P., Pokahr, A., and Ricci, A. (2006).
A survey of programming languages and platforms for multi-agent systems.
Informatica (Slovenia), 30(1):33–44.

Bibliography ii

-  Bordini, R. H., Dastani, M., Dix, J., and Fallah-Seghrouchni, A. E., editors (2005).

Multi-Agent Programming: Languages, Platforms and Applications,
*volume 15 of Multiagent Systems, Artificial Societies, and Simulated
Organizations.*

Springer.

-  Bordini, R. H., Dastani, M., Dix, J., and Fallah-Seghrouchni, A. E., editors (2009).

Multi-Agent Programming: Languages, Tools and Applications.

Springer.

-  Bordini, R. H., Hübner, J. F., and Wooldridge, M. (2007).
- Programming Multi-Agent Systems in AgentSpeak Using Jason.*
Wiley Series in Agent Technology. John Wiley & Sons.

2023-07-13

└ Comparison with other paradigms
 └ Bibliography

- Bibliography ii
-  Bordini, R. H., Dastani, M., Dix, J., and Fallah-Seghrouchni, A. E., editors (2005).
Multi-Agent Programming: Languages, Platforms and Applications,
*volume 15 of Multiagent Systems, Artificial Societies, and Simulated
Organizations.*
Springer.
-  Bordini, R. H., Dastani, M., Dix, J., and Fallah-Seghrouchni, A. E., editors (2009).
Multi-Agent Programming: Languages, Tools and Applications.
Springer.
-  Bordini, R. H., Hübner, J. F., and Wooldridge, M. (2007).
Programming Multi-Agent Systems in AgentSpeak Using Jason.
Wiley Series in Agent Technology. John Wiley & Sons.

Bibliography iii

-  Bratman, M. E. (1987).
Intention, Plans, and Practical Reason.
Harvard University Press, Cambridge.
-  Bratman, M. E., Israel, D. J., and Pollack, M. E. (1988).
Plans and resource-bounded practical reasoning.
Computational Intelligence, 4:349–355.
-  Cohen, P. R. and Levesque, H. J. (1987).
Intention = choice + commitment.
In *Proceedings of the 6th National Conference on Artificial Intelligence*, pages 410–415. Morgan Kaufmann.

2023-07-13

- └ Comparison with other paradigms
 - └ Bibliography

- Bratman, M. E. (1987).
Intention, Plans, and Practical Reason.
Harvard University Press, Cambridge.
- Bratman, M. E., Israel, D. J., and Pollack, M. E. (1988).
Plans and resource-bounded practical reasoning.
Computational Intelligence, 4:349–355.
- Cohen, P. R. and Levesque, H. J. (1987).
Intention = choice + commitment.
In *Proceedings of the 6th National Conference on Artificial Intelligence*, pages 410–415. Morgan Kaufmann.

Bibliography iv

 Dastani, M. (2008).

2apl: a practical agent programming language.

Autonomous Agents and Multi-Agent Systems, 16(3):214–248.

 Fisher, M. (2005).

Metatem: The story so far.

In *PROMAS*, pages 3–22.

 Fisher, M., Bordini, R. H., Hirsch, B., and Torroni, P. (2007).

Computational logics and agents: A road map of current technologies and future trends.

Computational Intelligence, 23(1):61–91.

2023-07-13

└ Comparison with other paradigms
 └ Bibliography

- Dastani, M. (2008).
 2apl: a practical agent programming language.
 Autonomous Agents and Multi-Agent Systems, 16(3):214–248.
- Fisher, M. (2005).
 Metatem: The story so far.
 In *PROMAS*, pages 3–22.
- Fisher, M., Bordini, R. H., Hirsch, B., and Torroni, P. (2007).
 Computational logics and agents: A road map of current technologies and future trends.
 Computational Intelligence, 23(1):61–91.

Bibliography v

 Georgeff, M. P. and Lansky, A. L. (1987).

Reactive reasoning and planning.

In *Proc. of the Sixth National Conference on Artificial Intelligence (AAAI 87)*, pages 677–682.

 Giacomo, G. D., Lespérance, Y., and Levesque, H. J. (2000).

Congolog, a concurrent programming language based on the situation calculus.

Artif. Intell., 121(1-2):109–169.

 Hindriks, K. V. (2009).

Programming rational agents in GOAL.

In [Bordini et al., 2009], pages 119–157.

2023-07-13

└ Comparison with other paradigms
 └ Bibliography

 Georgeff, M. P. and Lansky, A. L. (1987).

Reactive reasoning and planning.

In *Proc. of the Sixth National Conference on Artificial Intelligence (AAAI 87)*, pages 677–682.

 Giacomo, G. D., Lespérance, Y., and Levesque, H. J. (2000).

Congolog, a concurrent programming language based on the situation calculus.

Artif. Intell., 121(1-2):109–169.

 Hindriks, K. V. (2009).

Programming rational agents in GOAL.

In [Bordini et al., 2009], pages 119–157.

Bibliography vi

 Hindriks, K. V., de Boer, F. S., van der Hoek, W., and Meyer, J.-J. C. (1997).

Formal semantics for an abstract agent programming language.

In Singh, M. P., Rao, A. S., and Wooldridge, M., editors, *ATAL*, volume 1365 of *Lecture Notes in Computer Science*, pages 215–229. Springer.

 Newell, A. (1982).

The knowledge level.

Artificial Intelligence, 18:87–127.

 Pokahr, A., Braubach, L., and Lamersdorf, W. (2005).

Jadex: A bdi reasoning engine.

In [Bordini et al., 2005], pages 149–174.

2023-07-13

└ Comparison with other paradigms
 └ Bibliography

- © Hindriks, K. V., de Boer, F. S., van der Hoek, W., and Meyer, J.-J. C. (1997).
Formal semantics for an abstract agent programming language. In Singh, M. P., Rao, A. S., and Wooldridge, M., editors, *ATAL*, volume 1365 of *Lecture Notes in Computer Science*, pages 215–229. Springer.
- © Newell, A. (1982).
The knowledge level. *Artificial Intelligence*, 18:87–127.
- © Pokahr, A., Braubach, L., and Lamersdorf, W. (2005).
Jadex: A bdi reasoning engine. In [Bordini et al., 2005], pages 149–174.

Bibliography vii

 Rao, A. S. (1996).

Agentspeak(l): Bdi agents speak out in a logical computable language.

In de Velde, W. V. and Perram, J. W., editors, *MAAMAW*, volume 1038 of *Lecture Notes in Computer Science*, pages 42–55. Springer.

 Rao, A. S. and Georgeff, M. P. (1995).

BDI agents: from theory to practice.

In Lesser, V., editor, *Proceedings of the First International Conference on MultiAgent Systems (ICMAS'95)*, pages 312–319. AAAI Pess.

 Searle, J. R. and Vanderveken, D. (1985).

Foundations of illocutionary logic.

Cambridge University Press, Cambridge.

2023-07-13

└ Comparison with other paradigms
 └ Bibliography

- ─ Rao, A. S. (1996).
Agentspeak(l): Bdi agents speak out in a logical computable language.
In de Velde, W. V. and Perram, J. W., editors, *MAAMAW*, volume 1038 of *Lecture Notes in Computer Science*, pages 42–55. Springer.
- ─ Rao, A. S. and Georgeff, M. P. (1995).
BDI agents: from theory to practice.
In Lesser, V., editor, *Proceedings of the First International Conference on MultiAgent Systems (ICMAS'95)*, pages 312–319. AAAI Pess.
- ─ Searle, J. R. and Vanderveken, D. (1985).
Foundations of illocutionary logic.
Cambridge University Press, Cambridge.

Bibliography viii

 Shoham, Y. (1993).

Agent-oriented programming.

Artif. Intell., 60(1):51–92.

 Winikoff, M. (2005).

Jack intelligent agents: An industrial strength platform.

In [Bordini et al., 2005], pages 175–193.

 Wooldridge, M. (2009).

An Introduction to MultiAgent Systems.

John Wiley and Sons, 2nd edition.

2023-07-13

└ Comparison with other paradigms
 └ Bibliography

 Shoham, Y. (1993).
Agent-oriented programming.
Artif. Intell., 60(1):51–92.

 Winikoff, M. (2005).
Jack intelligent agents: An industrial strength platform.
In [Bordini et al., 2005], pages 175–193.

 Wooldridge, M. (2009).
An Introduction to MultiAgent Systems.
John Wiley and Sons, 2nd edition.