Programação orientada a agentes

Maiquel de Brito

[1]UFSC-Blumenau-Brasil

SEPEX 2020

Outubro 2020

Outline

Contexto

Beer robot

Considere-se a existência de um robô cuja única missão é levar cerveja até seu proprietário quando este lhe solicita.

- Deve-se trazer a cerveja favorita do proprietário
- Na falta da cerveja favorita, deve-se trazer próxima na ordem de preferências
- O proprietário pode mudar suas preferências ao longo do tempo (mas ele não sabe programar o robô)
- Não há cerveja na geladeira
- A geladeira mudou de lugar
- O robô mudou de proprietário
- etc, etc, etc...

Como programar este robô?

Contexto

Beer robot: como programar este robô?

Programas de computador não são escritos com objetivos de alto nível

i.e. dar um objetivo e deixar a máquina decidir como atingi-lo

Programas de computador são listas detalhadas de instruções

Todas as possíveis situações devem ser previstas e tratadas detalhadamente

Programas de computador dificilmente adaptam sua execução a outros programas ou aos usuários

A comunicação entre programa consiste em selecionar opções pré-definidas (ex. menus)

Contexto

A Programação Orientada a Agentes (na abordagem deste curso)

- Programa-se o *know how* do agente

  i.e. programa-se a forma como o agente "faz coisas"

  ex.: ir até a geladeira, descobrir a cerveja favorita, comprar cerveja...

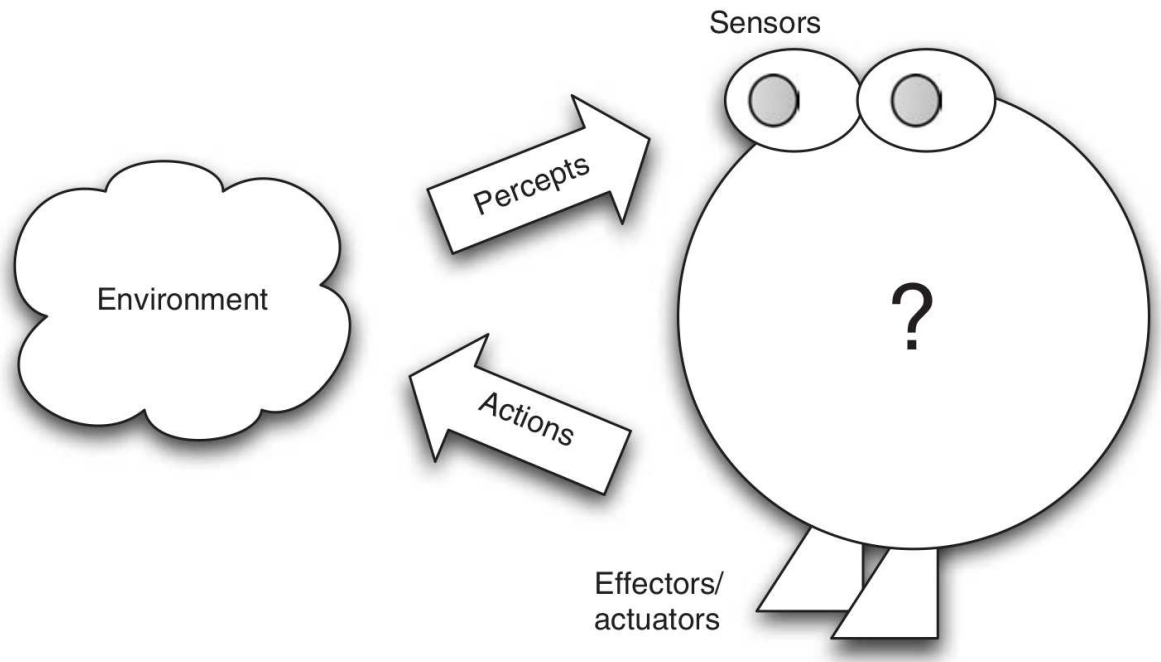
- O agente "decide" quais *coisas* irá fazer

  ex.: levar a única cerveja disponível ou comprar a cerveja favorita?

- Agentes são
  - Autonomos i.e. decidem seu curso de ação sem a intervenção de um operador
  - Reativos i.e. adaptam-se rapidamente a novas circunstâncias do sistema
  - Proativos i.e. tomam a iniciativa de tentar atingir objetivos
  - Socialmente capacitados i.e. interagem com outros agentes e até mesmo com humanos

O que é um agente?

"... é um sistema situado em um ambiente do qual tem percepções e no qual atua de forma autônoma para satisfazer os objetivos para os quais foi projetado." [Russel e Norvig, 1995, Wooldridge, 2009]



Abstrações em computação

Abstrações de alto nível para pensar em soluções computacionais:

- Funções e procedimentos

  programação estruturada
- Classes e objetos

  programação orientada a objetos
- Estados mentais

  programação orientada a agentes

A arquitetura BDI

Beliefs, **D**esires, and **I**ntentions

ou crenças, desejos e intenções

Beliefs

Informações que o agente possui sobre o mundo em que atua

Desires

Estados do mundo que o agente gostaria de atingir (*opções*)

Intentions

Estados do mundo que o agente decidiu atuar para atingir

A arquitetura BDI

Exemplo: veículo autônomo

Beliefs: posição atual, velocidade atual, energia disponível

Desires: chegar rapidamente ao destino, manter a energia acima de um limite

Intentions: se a energia baixar do limite: adquirir energia

Exemplo: *autonomous trader*

Beliefs: patrimônio atual, valor dos bens em oferta

Desires: comprar bens, manter o patrimônio

Intentions: se valor dos bens em oferta é baixo: comprar bens

A arquitetura BDI

Como um agente vai de suas BDI a ações?

Raciocínio prático

Decidir entre opções (com base no conhecimento que se possui)
que podem até ser conflitantes (ex.: chegar rápido ao destino vs. manter energia)

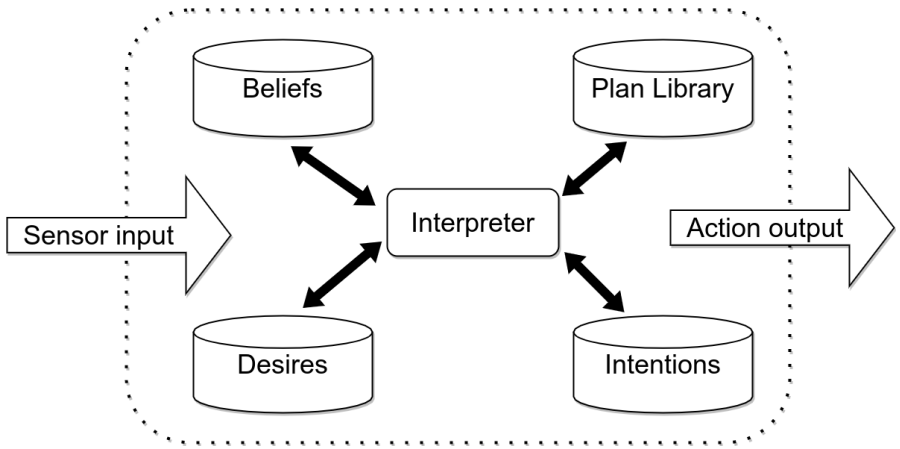
Raciocínio prático envolve dois processos

1. *Deliberação*: decisão sobre quais intenções adotar
2. *Raciocínio Meios-Fins*: decisão sobre a sequência de ações a seguir para satisfazer as intenções adotada

A arquitetura BDI

Como se comporta um programa de computador BDI?

1. Percebe o mundo e atualiza crenças
2. Avalia seus desejos e decide qual intenção atingir
3. Através de raciocínio meios-fins, define um plano para satisfazer a intenção
4. Executa o plano



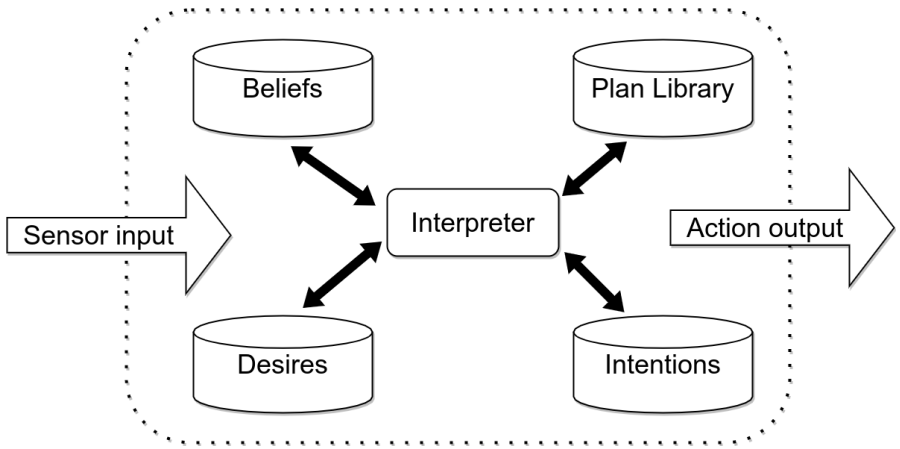
Adaptado de [Bordini et al., 2007]

Como implementar BDI?

Como implementar programas com o comportamento do BDI?

É necessário implementar:

1. mecanismos de percepção
2. representação de crenças, desejos, planos
3. revisão de crenças
4. geração de intenções
5. deliberação
6. raciocínio meios fins
7. mecanismos de comunicação
8. mecanismos de execução de planos
9. etc, etc, etc...
10. comportamento do agente



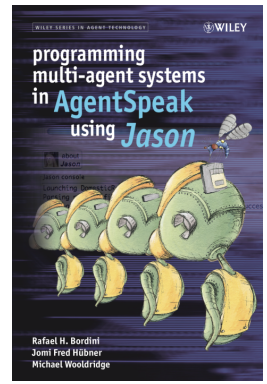
Adaptado de [Bordini et al., 2007]

Programação orientada a agentes: linguagens e interpretadores implementam os itens 1 a 9 acima. O programador preocupa-se com o item 10.

A linguagem Jason

Linguagem para programação de agentes BDI

`http://jason.sf.net`



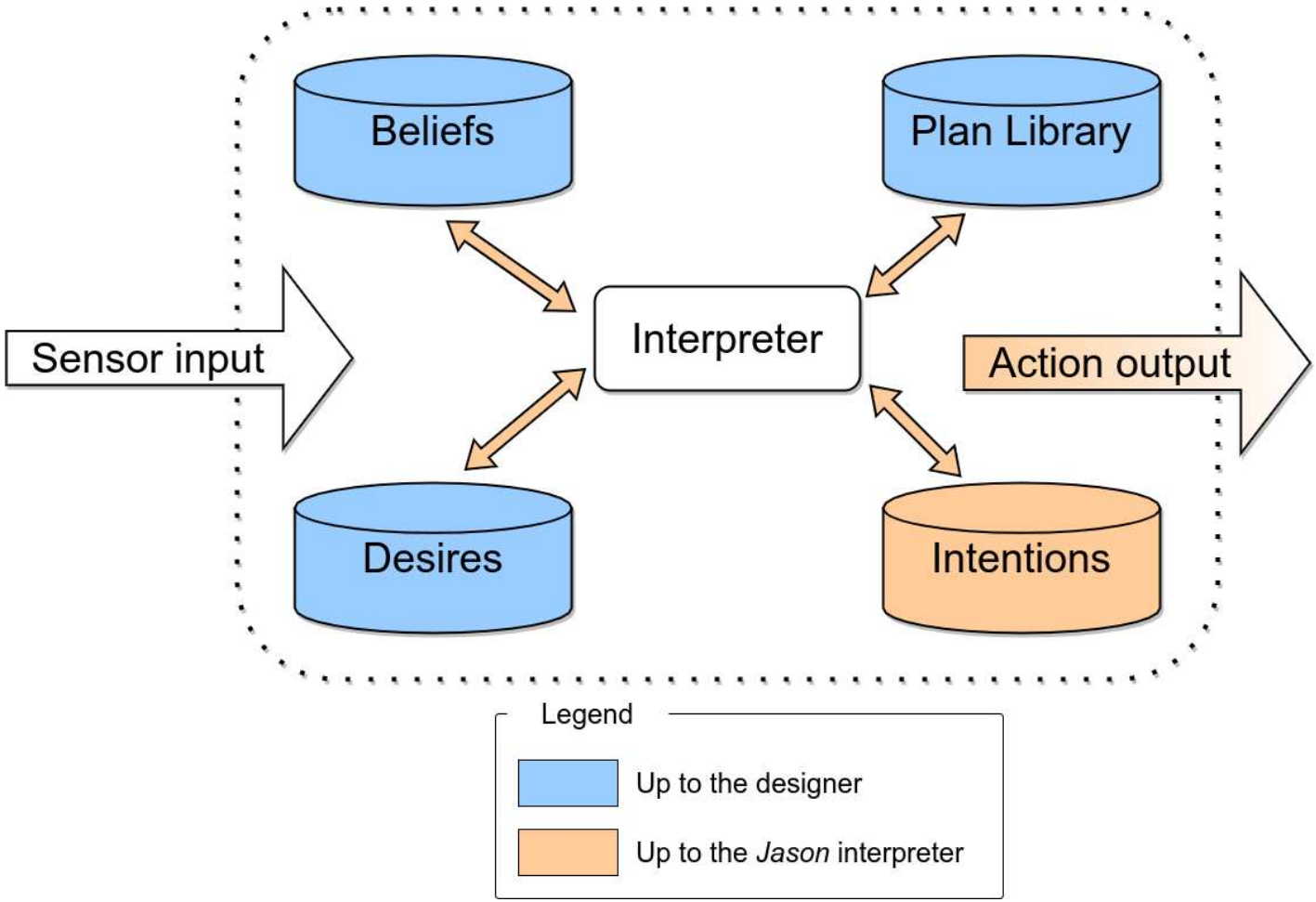
Programação de agentes com construtores BDI de alto nível

em Jason, programa-se crenças, desejos e intenções
em vez de programar variáveis, funções, procedimentos, métodos, objetos, ...

Execução de agentes na forma de um ciclo de raciocínio

um agente Jason permanece em execução contínua, observando o mundo, atualizando suas crenças
avaliando os desejos que podem ser satisfeitos, selecionando planos para executá-los,
colocando os planos em execução e reiniciando o ciclo

A linguagem Jason



Hello World

O primeiro programa *Jason*:

```
!start. //initial desire

//plan to satisfy the desire
+!start
    <- .print("hello world.").
```

Greet

O primeiro programa *Jason*: download

pequena alteração: trocar `start` por `greet`

```
!greet. //initial desire (or goal)

//plan to satisfy the desire
+!greet
    <- .print("hello world.").
```

Analisando o programa acima...

O programador do agente especificou que:

- agente tem um desejo inicial
- tem um plano para satisfazer o desejo

O interpretador Jason:

- Identifica o desejo inicial
- Seleciona um plano
- Transforma o plano em intenção
- Coloca o plano em execução

Hello World - Adicionando crenças

```
current_day(friday). //belief

!greet. //initial desire

//plan to satisfy the desire
+!greet : current_day(friday)
    <- .print("hello world.");
       print("Today is friday").
```

download

Hello World - Adicionando crenças

Exercício

```
current_day(friday). //belief

!greet. //initial desire

//plan to satisfy the desire
+!greet : current_day(friday)
    <- .print("hello world.");
       print("Today is friday").
```

1. Responder:
   O que acontece se removermos a crença `current_day(friday)`?
2. Criar planos para satisfazer o desejo `greet` quando não é sexta-feira.

Beliefs

```
//beliefs
successor(monday,tuesday).
successor(tuesday,wednesday).
successor(wednesday,thursday).
successor(thursday,friday).
successor(friday,saturday).
successor(saturday,sunday).
successor(sunday, monday).

current_day(friday).


!greet. //initial desire


//plan to satisfy the desire
+!greet : current_day(friday) & successor(friday,S)
    <- .print("hello world.");
       print("Today is friday").
       print("Tomorrow is ", S).
```

download

Um pouco de programação lógica I

Átomo (*Atom*):

- sequência de caracteres começando com letra minúscula
- representa indivíduos ou objetos
  ex.: `friday`, `john`, `ufsc`

Variável (*Variable*):

- sequência de caracteres começando com letra maiúscula
  ex.: `DayOfWeek`, `Person`, `Ufsc`
- podem ser ligadas (*bound*) a um valor átomo
  ex. : `DayOfWeek/monday`, `Person/john`, `University/ufsc`

Um pouco de programação lógica II

Predicado:

- átomo (ou *functor* seguido por argumentos entre parênteses
- representa propriedades ou relações entre objetos
  ex.: `current_day(monday)` — propriedade
  `successor(sunday, monday)` — relação
- o número de argumentos indica a *aridade* do predicado

  `current_day(monday)` — aridade 1

  `successor(sunday, monday)` — aridade 2

  `raining` — aridade zero

Um pouco de programação lógica III

Negação

- *Negação por falha*: agente não pode concluir que algum fato é verdade.
  ex.: `not vida_inteligente(jupiter)` indica que o agente não acredita em vida_inteligente(jupiter).

- *Negação forte* (ou *strong negation*): o agente acredita que algum fato não é verdade
  ex.: `~is_round(earth)` indica que o agente acredita que is_round(earth) é falso

Um pouco de programação lógica III

Fórmula:

um ou mais predicados (e suas negações, tanto forte quanto por falha)
combinados com

- operadores lógicos

  `&` (*e*), `|` (*ou*),
- variáveis
- operadores aritméticos

  $+$ (soma), $-$ (subtração), $*$ (multiplicação), $/$ (divisão), $**$ (potenciação),

  div (divisão inteira), mod (resto da divisão)
- operadores relacionais

  $==$ (igual), $\ ==$ ($\neq$),$>$,$<$, $<=$ ($\leq$), $>=$, ($\geq$)

Um pouco de programação lógica IV

Fórmulas são avaliadas com respeito às crenças do agente

Crenças

```
//beliefs
successor(friday,saturday).
successor(saturday,sunday).
current_day(friday).
```

Fórmulas

`current_day(friday)` - verdadeiro
`current_day(monday)` - falso
`current_day(X)` - verdadeiro: `(X/friday,X /monday)`
`current_day(friday)&successor(friday,saturday)` - verdadeiro
`current_day(friday)&successor(friday,monday)` - falso
`current_day(friday)&successor(friday,S)` - verdadeiro: `(S/saturday)`
`current_day(friday)&~successor(friday,saturday)` - falso

Rules

Permitem ao agente concluir novos fatos a partir de suas crenças.

```
//beliefs
successor(monday,tuesday).              successor(friday,saturday).
successor(tuesday,wednesday).           successor(saturday,sunday).
successor(wednesday,thursday).          successor(sunday, monday).
successor(thursday,friday).

current_day(friday).


//Rule to check whether Yesterday was predecessor of Today
predecessor(Today,Yesterday) :- successor(Yesterday, Today).

!greet. //initial desire

//plan to satisfy the desire
+!greet : current_day(friday) & successor(friday,S) &
          predecessor(friday, Yesterday)
    <- .print("hello world.");
       .print("Today is friday").
       .print("Tomorrow is ", S);
       .print("Yesterday was ", Yesterday).
```

download

Rules

Permitem ao agente concluir novos fatos a partir de suas crenças.

```
//beliefs
successor(monday,tuesday).              successor(friday,saturday).
successor(tuesday,wednesday).           successor(saturday,sunday).
successor(wednesday,thursday).          successor(sunday, monday).
successor(thursday,friday).

current_day(friday).


//Rule to check whether Yesterday was predecessor of Today
predecessor(Today,Yesterday) :- successor(Yesterday, Today).

!greet. //initial desire

//plan to satisfy the desire
+!greet : current_day(friday) & successor(friday,S) &
          predecessor(friday, Yesterday)
    <- .print("hello world.");
       .print("Today is friday").
       .print("Tomorrow is ", S);
       .print("Yesterday was ", Yesterday).
```

Exercício: estender o programa acima para que o agente:

1. informe o dia de anteontem
2. informe se amanhã será final de semana (*saturday* ou *sunday*)

De onde vêm as crenças? I

Podem ser codificadas diretamente no agente.

```
current_day(friday). //belief

!greet. //initial desire

//plan to satisfy the desire
+!greet : current_day(Today)
    <- print("Today is ", Today).
```

Pode adicionadas novas pelo próprio agente.

```
current_day(friday). //belief

!greet. //initial desire

//plan to satisfy the desire
+!greet : current_day(Today) &
           predecessor(Today, Yesterday)
    <- print("Today is ", Today).
      -last_greeting(Yesterday); //add belief
      +last_greeting(Today); //remove belief
      -+last_greeting(Today). //update belief
```

Crenças adicionadas pelo próprio agente possuem a anotação `source(self)`
Ex: `current_day(friday)[source(self)]`

De onde vêm as crenças? II

Pela percepção do agente sobre o ambiente

- Crenças provenientes de percepção possuem a anotação `source(percept)`
- Crenças são atualizadas automaticamente conforme as percepções mudam

```
current_day(sunday).

!greet.

+!greet : current_day(MyDay)[source(self)] &
          current_day(CalendarDay)[source(percept)]
   <- .print("hello world.");
      .print("I thought it was ", MyDay);
      .print("The calendar informs it is ", CalendarDay).
```

download

De onde vêm as crenças? III

Pela comunicação com outros agentes

será visto mais tarde...

Goals

Goals representam, em *Jason*, a noção de *desejo* do BDI.

Representam estados de mundo que o agente gostaria de atingir.

Tipos de goals:

- *achievement goal*: objetivo de fazer

  sintaxe: `!g.` : o agente deseja atingir *g*

- *achievement goal*: objetivo de saber

  sintaxe: `?g.` : o agente deseja saber *g*

Exemplos

```
!greet. //achievement goal

?is_holiday. //test goal
```

De onde vêm os goals? I

Podem ser goals iniciais do agente

```
current_day(friday). //belief

!greet. //initial desire

//plan to satisfy the desire
+!greet : current_day(Today)
    <- print("Today is ", Today).
```

Podem ser adicionadas durante a execução de planos

```
current_day(friday). //belief

!greet. //initial desire

//plan to satisfy the desire
+!greet : current_day(Today)
    <- print("Today is ", Today);
       //add test [sub]goal
       ?raining(Today);
       //add achievement [sub]goal
       !check_temperature.
```

De onde vêm os Goals? II

Pela comunicação com outros agentes

será visto mais tarde...

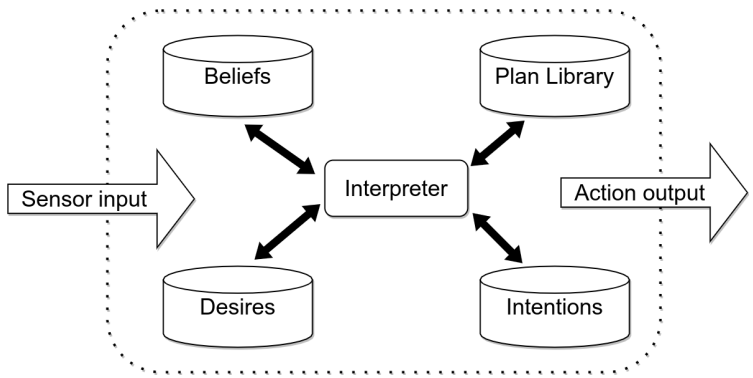
O ciclo de raciocínio BDI

Mudança nas crenças ou nos desejos:

- o agente revê suas intenções:
  - algumas intenções podem não ser mais factíveis
  - novas intenções podem surgir em função das novas circunstâncias

em Jason:

- a adição ou remoção de crenças dispara avaliação de *planos*
- planos factíveis são colocados em execução e se transformam em *intenções*



Fonte: [Bordini et al., 2007]

Planos em Jason

`triggering_event` : context <– body.

`triggering_event`: evento que determina a seleção do plano

`context`: circunstâncias em que o plano será selecionado

`body`: curso de ação a ser tomado se o plano for selecionado

Triggering events

`+b` (adição de crença)
`-b` (remoção de crença)
`+!g` (adição de ach. goal)
`-!g` (remoção de ach. goal)
`+?g` (adição de test goal)
`-?g` (remoção de test goal)

Planos em Jason – Exemplo

```
current_day(friday).
current_temperature(25).

!greet. //initial goal

/* Plan 1 */
+!greet : current_day(monday)
   <- .print("It is not a good day").

/* Plan 2 */
+!greet : current_day(tuesday)|current_day(wednesday)|
          current_day(thursday)
   <- .print("It is a boring day").


/* Plan 3 */
+!greet : (current_day(friday)|current_day(saturday)) &
          current_temperature(X) & X >= 22 & X <= 26
   <- .print("hello world.");
      .print("It is a good day!").


/* Plan 4 */
+!greet
   <- .print("hello world.").
```

download

Planos em Jason – Exemplo

```
current_day(friday).
current_temperature(25).

!greet. //initial goal

/* Plan 1 */
+!greet : current_day(monday)
   <- .print("It is not a good day").

/* Plan 2 */
+!greet : current_day(tuesday)|current_day(wednesday)|
          current_day(thursday)
   <- .print("It is boring day").


/* Plan 3 */
+!greet : (current_day(friday)|current_day(saturday)) &
          current_temperature(X) & X >= 22 & X <= 26
   <- .print("hello world.");
      .print("It is a good day!").


/* Plan 4 */
+!greet
   <- .print("hello world.").
```

⇝ O que o agente fará se for domingo (i.e. `current_day(sunday)`)?

⇝ O que o agente fará se for sexta-feira e a temperatura for 30°?

Planos em Jason – Exemplo

```
current_day(friday).
current_temperature(25).


/* Plan 1 */
+current_day(monday)
   <- .print("It is not a good day").

/* Plan 2 */
+current_day(Day) : Day==tuesday | Day==wednesday | Day==thursday
   <- .print("It is boring day").

/* Plan 3 */
+current_day(Day) : (Day == friday | Day == saturday) &
                    current_temperature(X) & X >= 22 & X <= 26
   <- .print("It is a good day!").

/* Plan 4 */
+current_day(_) <- .print("hello world.").
```

download

Planos em Jason

O que pode constar no *body* de um plano?

```
+chuva : hora_saida(T) & hora_atual(H) & H>=T
    <- !g1; //novo subgoal
       !!g2; //novo goal
       ?b(X); //novo test goal
       +b1(T-H) //adiciona uma crença
       -b2(T+H) //remove uma crença
       -+b3(T*H) //atualiza uma crença (remove e adiciona)
       jia.get(X) //internal action
       fechar(janela) //ação externa (ambiente)
       .print(``Partiu'') internal action
```

Planos *vs.* ações dos agentes

É nos **planos** que se especifica as **ações** que os agentes executam.

Mas...
Onde essas ações são implementadas?

Há dois tipos de ações:

- Ações internas (*internal actions*)
- Ações ações externas (executadas sobre o ambiente)

Ações externas

Executadas sobre elementos externos à implementação do agente

i.e. sobre o *ambiente*

Provocam efeitos sobre o *ambiente*.

Exemplo: disponível para download aqui

demo_MAS.mas2j

```
MAS demo_MAS {
   infrastructure: Centralised

   //DemoEnv: classe java que implementa o ambiente
   environment: DemoEnv

   agents:
      bob;
}
```

agent code

```
!increment.

+!increment
   <- inc; //implementado em DemoEnv
      .wait(500);
      !increment.
```

Ações externas

Ambiente em Jason

O ambiente é acessado através de programação Java

Ações são disponibilizadas pelo ambiente aos agentes

A execução dessas ações produz efeitos no ambiente.

Formas de implementar ambientes para Jason:

- Jason API: estender a classe `Environment` (documentação aqui)
- framework CArtAgO (informações aqui e aqui)

Internal actions

Internal actions são ações que

- fazem parte da implementação do agente
- agente é capaz de executar em qualquer cenário, aplicação, ambiente etc

Internal actions são implementadas em Java

- Novas *internal actions* podem ser implementadas
  e adicionadas ao agente
- O nome da ação é precedido por "."
  ex. `.print(‘‘Hello’’)`
- Há um conjunto padrão de *internal actions* disponível pelo Jason
  e estão disponiveis a todos os agentes Jason
- Mais informações aqui

Exemplo (download aqui)

```
+!start <-
    .print("hello world.");
    .random(R); .print(R);
    .date(Y,M,D);
    .print(Y,-",M,-",D);
    .time(H,MN,S);
    .print(H,-",MN,-",S).
```

Goals, planos e paralelismo

Um agente pode ter múltiplos *goals* que podem disparar planos que executam simultaneamente.

```
multiplier_2(0).
multiplier_3(0).

!print_multiples_of_2.
!print_multiples_of_3.

+!print_multiples_of_2 : multiplier_2(N)
   <- .print("2 x ", N, "= ", 2*N);
      -+multiplier_2(N+1);
      .wait(1000);
      !print_multiples_of_2.

+!print_multiples_of_3 : multiplier_3(N)
   <- .print("3 x ", N, "= ", 3*N);
      -+multiplier_3(N+1);
      .wait(800);
      !print_multiples_of_3.
```

download

Exercício

Disponível em

`https://maiquelb.github.io/sepex2020/exercises/vaccum/vaccum.html`

Interação entre agentes

A solução de alguns problemas pode requerer que agentes interajam uns com os outros

tipos de interação: colaboração, competição, negociação etc

Agentes podem interagir comunicando-se uns com os outros

Jason possui recursos para que agentes comuniquem-se uns com os outros

A comunicação entre os agentes pode afetar suas crenças e goals.

Comunicação entre agentes em Jason

Para enviar mensagens, utiliza-se a *internal action* `.send`:

```
.send(receiver,illocutionary_force,propositional_content)
```

`receiver`: nome do agente que deve receber a mensagem

`propositional_content`: conteúdo da mensagem

`illocutionary_force`: *performativa* que indica a intenção do agente ao enviar a mensagem

Comunicação entre agentes

Algumas performativas explicações considerando que o agente s envia mensagem para agente r

`tell`: s quer que r acredite que o conteúdo da mensagem é verdade

`untell`: s quer que r não acredite que o conteúdo da mensagem é verdade

`achieve`: s quer que r atinja um estado de mundo em que o conteúdo da mensagem seja verdadeiro (i.e. s delega um goal a r)

`unachieve`: s quer que r desista de atingir um estado de mundo em que o conteúdo da mensagem seja verdadeiro (i.e. s remove um objetivo r)

`askOne`: s saber se há uma crença de r que faça o conteúdo da mensagem ser verdadeiro

`askAll`: s saber se todas as crenças de r que façam o conteúdo da mensagem ser verdadeiro

`tellHow`: s envia um plano a r

`untellHow`: s pede que r remova o plano enviado de sua biblioteca de planos

`askHow`: s pede que r envie os planos aplicáveis ao evento enviado na mensagem

Comunicação entre agentes - Exemplo

Um sistema multiagente conta, inicialmente, apenas com o agente *Bob*. Todos os outros agentes que entram no sistema enviam uma mensagem a *Bob* dizendo "Hello".

Bob deseja saudar todos os participantes do sistema em sua língua nativa. Para isso, sempre que conhece um novo agente, pergunta qual é a sua nacionalidade.

O agente *Tom* quer fazer o mesmo, mas lhe falta conhecimento para isso. Por isso, pergunta a *Bob* (i) as origens de todos os demais agentes, (ii) as linguas faladas nos países de origem e (iii) como é a saudação naquelas línguas.

Mas, além disso, *Tom* não sabe como usar todo esse conhecimento para saudar os agentes. Por isso, solicita que *Bob* o informe como fazer isso. *Bob* lhe envia seus planos para executar saudações e, assim, *Tom* consegue enviar saudações aos demais agentes em suas línguas de origem.

download

Exercício

Domestic robot:

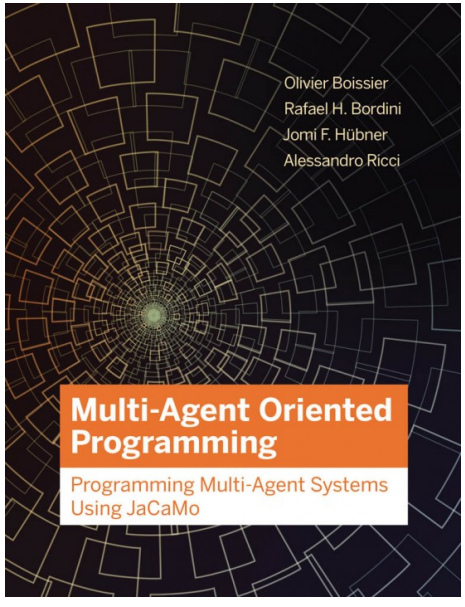
`https://maiquelb.github.io/sepex2020/exercises/domestic_robot/domestic_robot.html`

Avançando em programação orientada a agentes

O framework JaCaMo

Conjunto de ferramentas integradas para o desenvolvimento de sistemas multiagente

- Jason: programação de agentes
- $\mathcal{M}$oise: coordenação dos agentes
- CArtAgO: programação do ambiente



Avançando em programação orientada a agentes

Multi-Agent Programming Contest

Competição anual de programação de agentes

Avançando em programação orientada a agentes

Para mais ideias, respostas, projetos, novas perguntas...

maiquel.b@ufsc.br

References I


Michael Wooldridge
An Introduction to MultiAgent Systems.. 2 ed. Willey, 2009.


Rafael H. Bordini, Jomi F. Hübner, Michael Wooldridge
Programming Multi-Agent Systems in AgentSpeak using Jason.. 1 ed. Willey, 2007.


Stuart Russel and Peter Norvig.
Artificial Intelligence: A Modern Approach. 1 ed. Prentice Hall, 1995.