

Introdução ao Framework Jason: Sistemas Multi- agentes na Prática

Carlos Eduardo Pantoja

Sumário

1. Introdução a Sistemas Multi-Agentes
2. Modelo Belief-Desire-Intention
3. Linguagem de Programação a Agentes Jason
4. Beliefs
5. Goals
6. Plans & Actions
7. Comunicação Entre Agentes
8. O Ambiente dos Agentes
9. Conclusão
10. Referências Bibliográficas

1. Introdução a Sistemas Multi-Agentes

- **Agente**

Conforme [WOOLDRIDGE, 2000], agentes são componentes **autônomos** e **cognitivos**, originados da *inteligência artificial*, situados em um **ambiente** e possuem uma biblioteca de **planos** com possíveis **ações** em resposta aos estímulos **percebidos**, com a finalidade de atingir seus **objetivos** de projeto e modificar o ambiente em que estão inseridos.

Visão Tradicional de um Agente

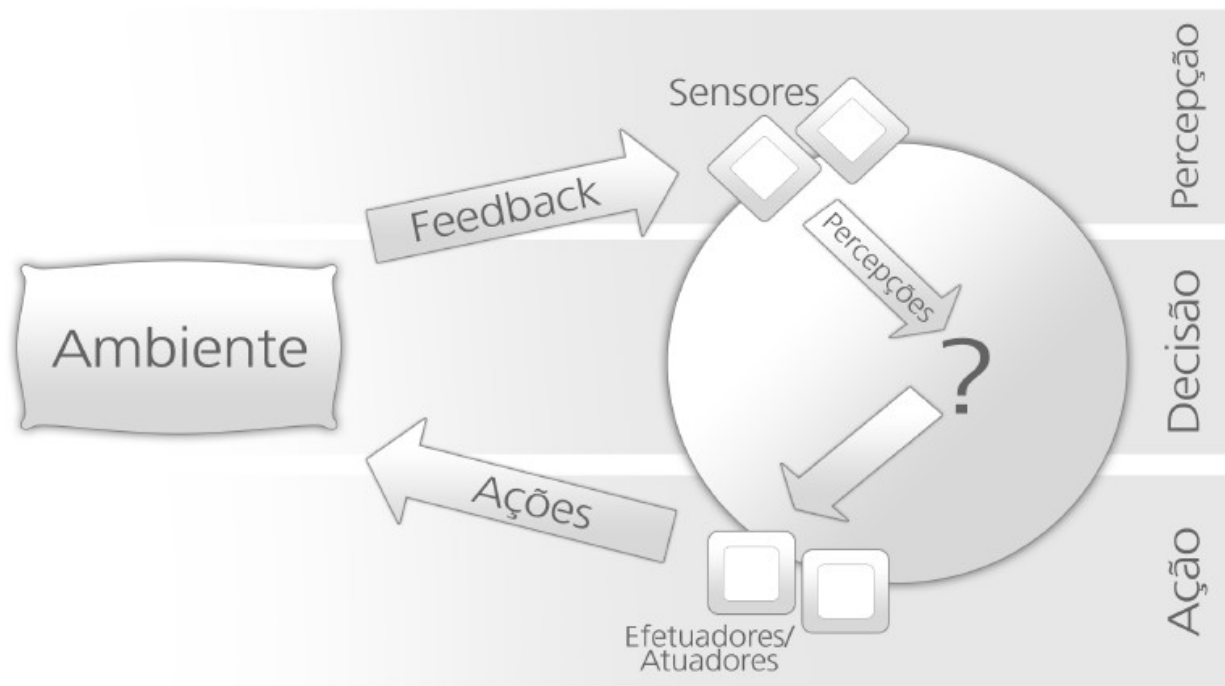


Figura 1. Um agente inteligente.

- **Sistemas Multi-Agentes (SMA)**

Um SMA contém um quantitativo de agentes que se **comunicam** entre si e podem **agir** em determinado **ambiente**. Diferentes agentes possuem esferas de influência onde terão controle sobre o que será **percebido** do ambiente e que podem coincidir em alguns casos.

Os agentes ainda podem estar agrupados em **organizações** com a finalidade de atingir **objetivos** e metas comuns.
[WOOLDRIDGE, 2009].

Visão Tradicional de um SMA

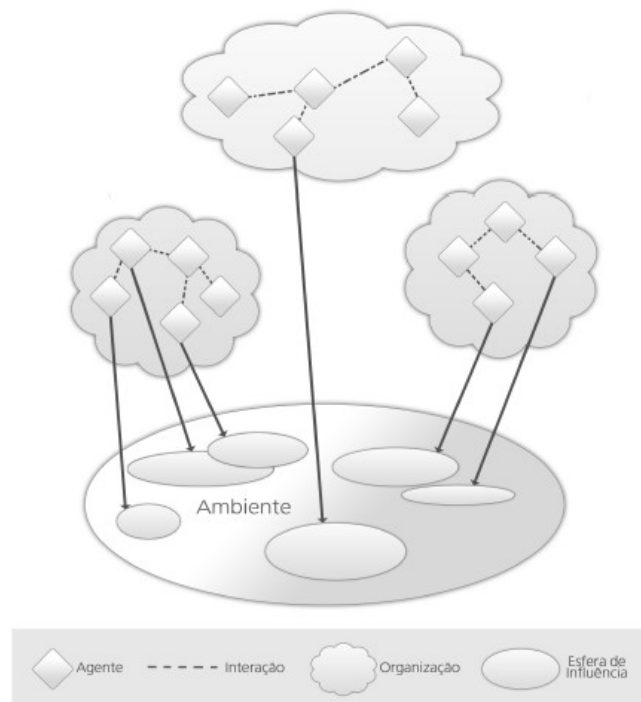


Figura 2. Um Sistema Multi-agente.

Conforme [WOOLDRIDGE, 2009], a abordagem SMA permite a modelagem desde sistemas simples a **complexos** e são usados em uma variedade de aplicações como indústria:

1. Gestão da Informação
2. Internet
3. Transportes
4. Telecomunicações
5. Medicina
6. Robótica
7. Entretenimento

Projetos de **SMA** no **verde** mundo:

1. Intelligent Room (AIRE/MIT)
2. A Plan-Based Command Post for UAVs (AIRE/MIT)
3. Biologically-Inspired Control for Self-Adaptive MAS (Harvard)
4. JaCaMo Project (UFSC/Itália/França/PUC-RS)
5. Bio Simulations (PUC/RJ)

Projetos de **SMA** no **Projeto Turing**:

1. Gerador de Codificação Automática para Jason
2. Ferramenta Gráficas para Metodologias Orientadas a Agentes
3. Plataforma Orientada a Agentes para UAVs.
4. Utilização da Plataforma para Automatização de Hardware

Objetivo Principal

- Criar sistemas multi-agentes utilizando uma **plataforma cognitiva** baseada em Java e **AgentSpeak: Jason Framework**.

Objetivo Secundário

- Permitir à equipe do projeto Turing adquirir conhecimentos para participar do **WESAAC**.

2. Modelo Belief-Desire-Intention (BDI)

O BDI se refere ao uso de programas de computadores com analogias a **crenças** (beliefs), **desejos** (desires) e **intenções** (intentions). A definição de cada uma é descrita como se segue [BORDINI et al., 2007]:

1. Crenças são **informações** que o agente tem sobre o mundo.
2. Desejos são todas as **possibilidades** de estados de negócio que o agente deve querer **atingir**. Porém, ter um desejo não significa que o agente irá atuar sobre ele, mas este é uma potencial influência nas ações do agente.
3. Intenções são todos os estados de negócios em que o agente **decidiu trabalhar**.

A arquitetura BDI permite que programas de computadores possuam **estado mental** [BRATMAN, 1987].

Um agente é considerado **racional** se escolher agir em busca dos seus interesses e baseado nas **crenças** que possuir do mundo [WOOLDRIDGE, 1999].

Existem diversas linguagens e plataformas que implementam o conceito de **BDI**:

1. PRS [BRATMAN, 1987]
2. JAM [HUBER, 1999]
3. dMARS [D'INVERNO et al., 1998]
4. JACK [WINIKOFF, 2005]
5. JASON [BORDINI et al., 2007]
6. JADE/JADEX [BELLIFEMINE et al., 2007]

- **Procedural Reasoning System**

Caso um modelo computacional de agentes precise ser implementado, utiliza-se o modelo de **raciocínio prático** fazendo o uso da **deliberação** e o **raciocínio fim-meio**.

Os planos em PRS contêm os seguintes componentes:

1. Metas (Goals)
2. Contexto (Context)
3. Corpo (Body)

No momento de sua inicialização, um agente PRS terá uma coleção de **planos**, e **crenças iniciais** sobre o ambiente. As crenças são representadas como formulas atômicas da linguagem de primeira ordem e, além disso, um agente também possuirá uma **meta principal** [BORDINI et al., 2007].

A Arquitetura PRS

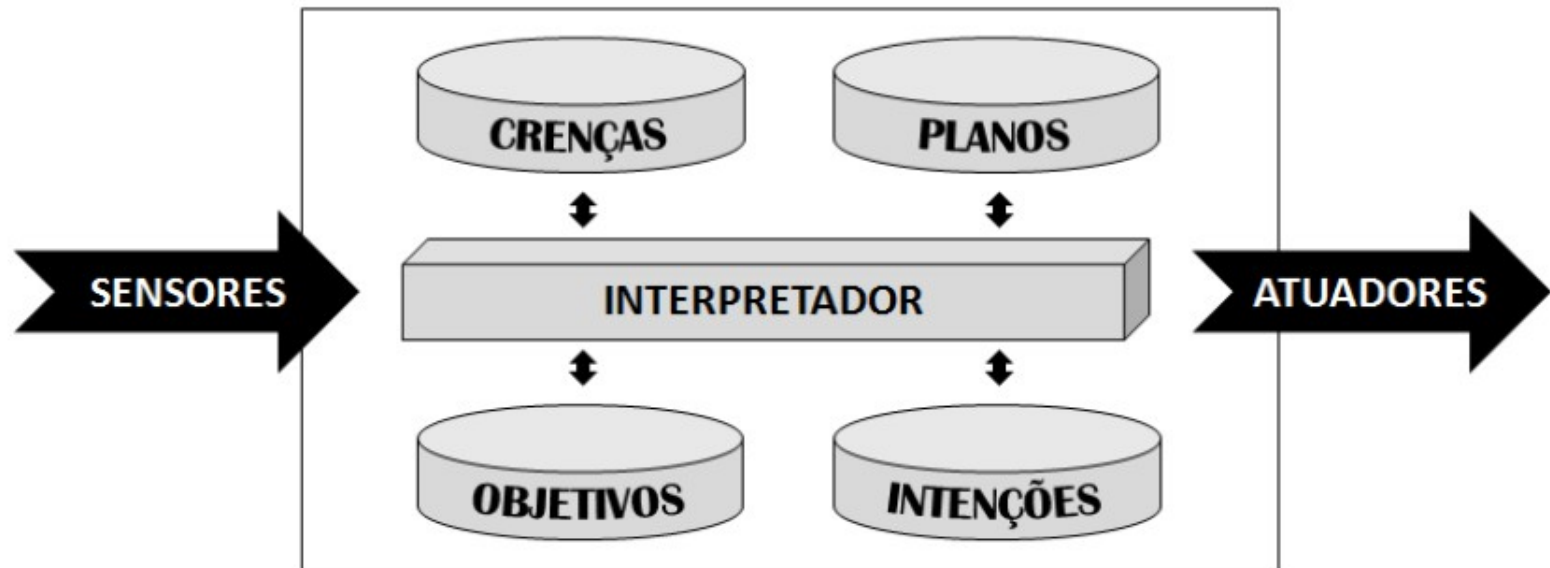


Figura 3. O Procedural Reasoning System.

3. Linguagem de Programação a Agentes Jason

- **Framework Jason**

O JASON é um framework baseado em AgentSpeak e Java que utiliza as principais características do PRS. Em JASON um agente é composto de **crenças, metas, planos** e **ações** e é programado utilizando o **AgentSpeak**.

Os agentes em JASON estão inseridos em um ambiente, que estende a classe Environment, onde as **percepções** e **reações a estímulos** do ambiente são programadas em Java [BORDINI et al., 2007].

- **Instalando o Framework Jason**

1. Instalando o Eclipse (DVD)
2. Configurando o Eclipse/Jason (Manual de Instalação)

- **Criando um Novo Projeto Jason**

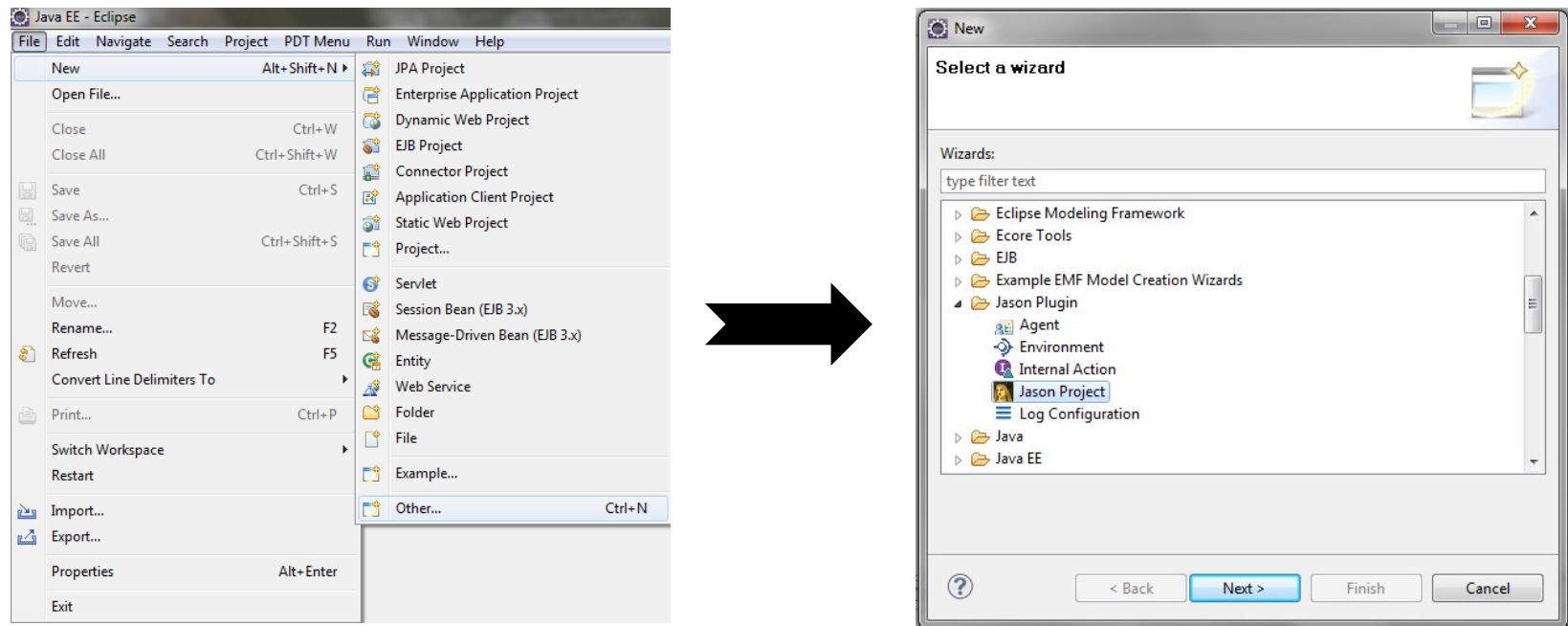
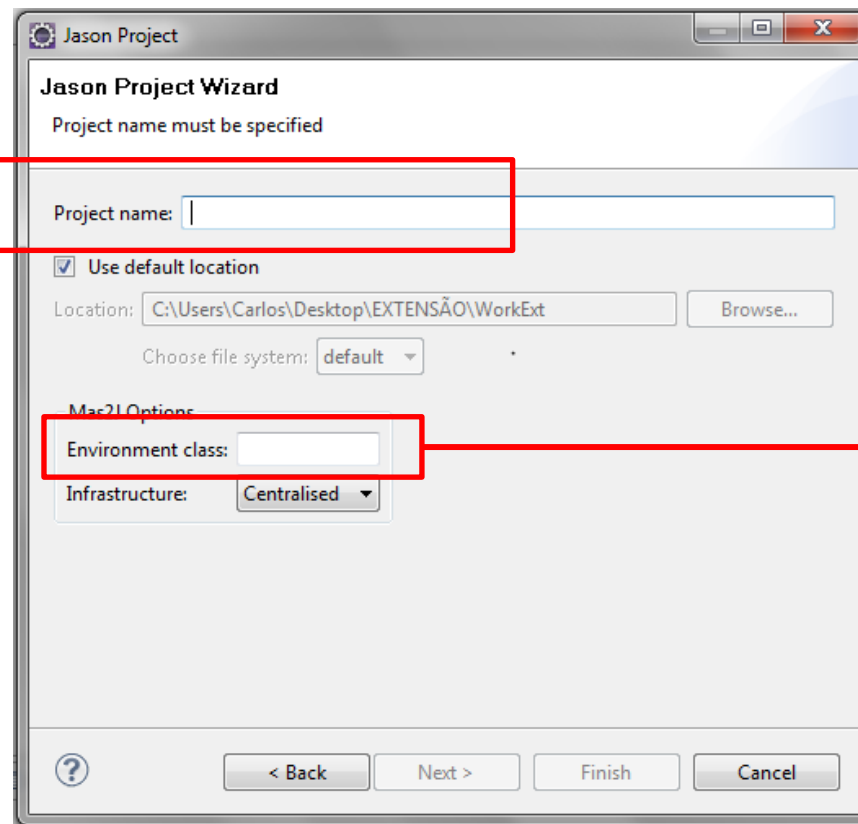


Figura 4. File>New>Other>Jason Project

- **Criando um Novo Projeto Jason**

Colocar o
nome do
projeto



Começar o
projeto
com um
ambiente
definido

Figura 5. O Jason Project.

- **Criando um Novo Projeto Jason**

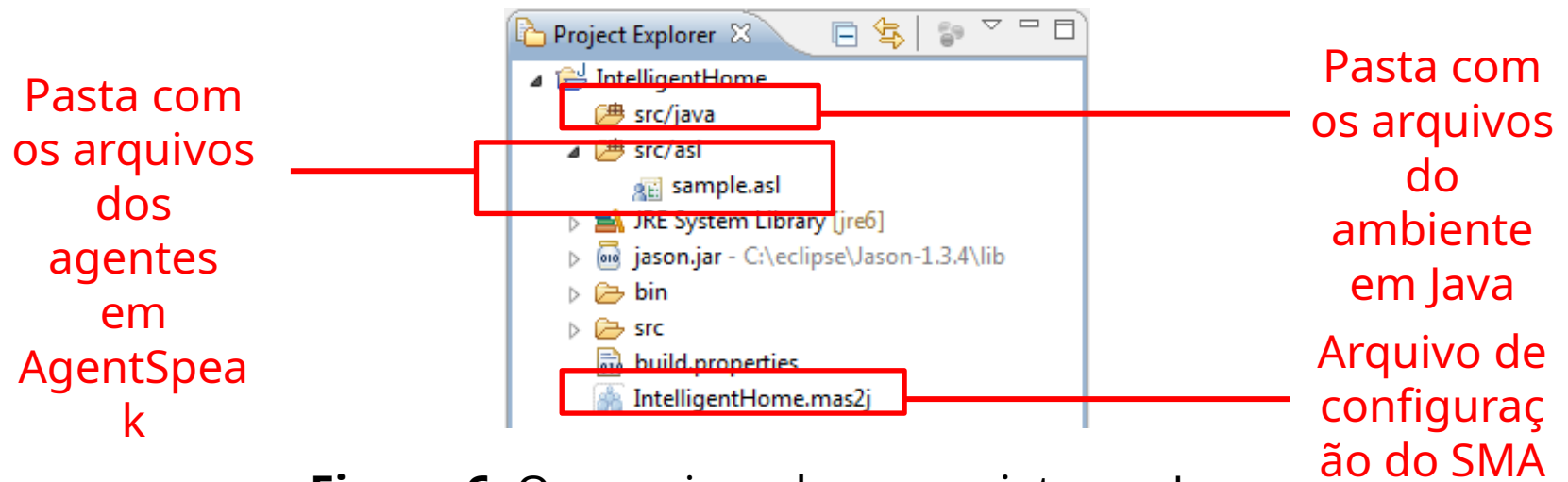


Figura 6. Os arquivos de um projeto em Jason.

- **Inserindo um Novo Agente**

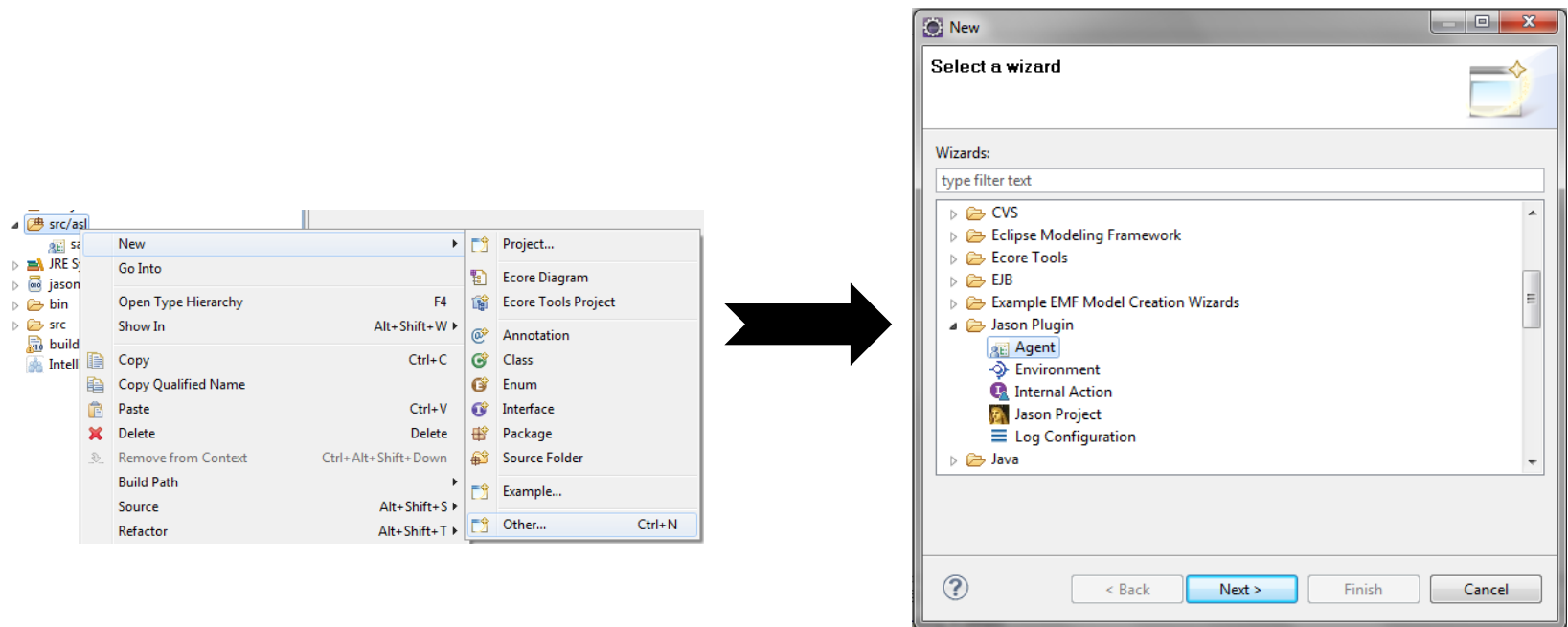


Figura 7. File>New>Other>Agent.

- **Inserindo um Novo Agente**

Digitar o nome do Agente. A primeira letra deve ser MINÚSCULA.

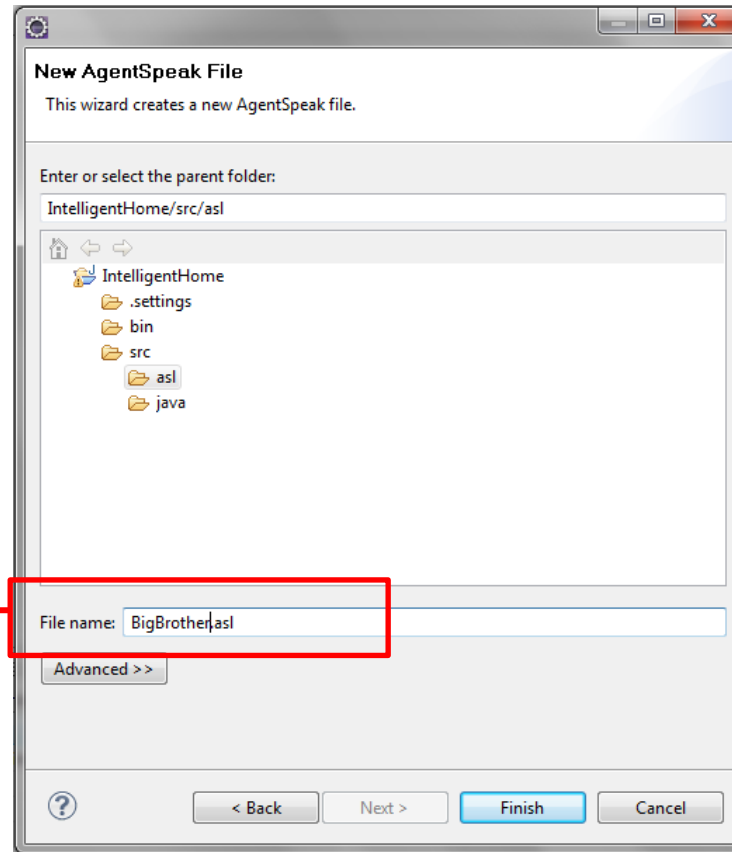


Figura 8. Um novo agent AgentSpeak.

- **Inserindo um Novo Agente**

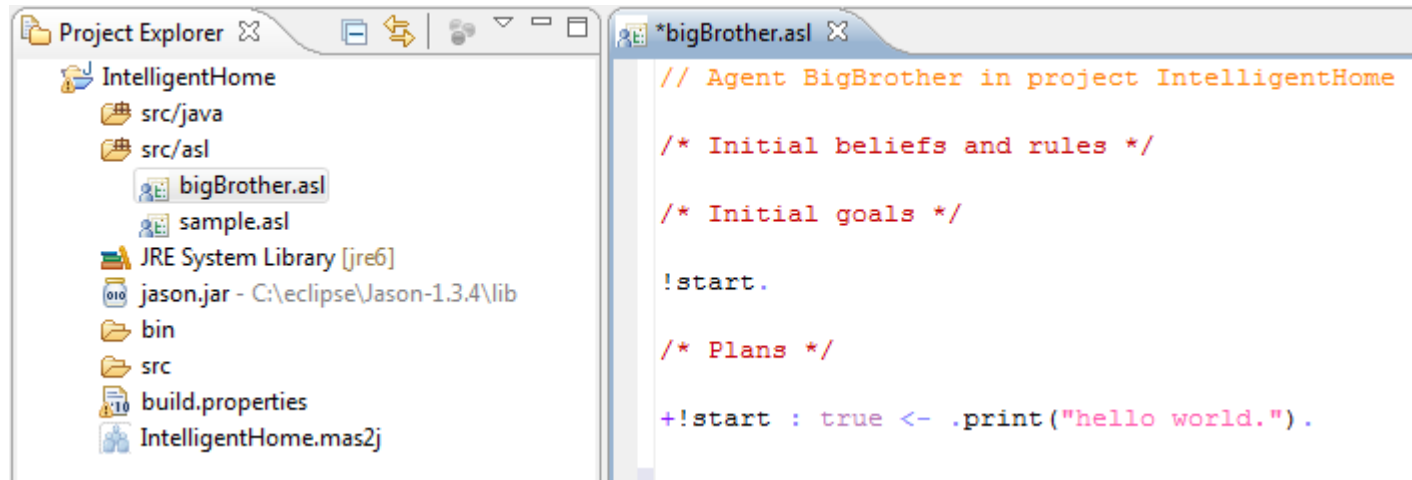
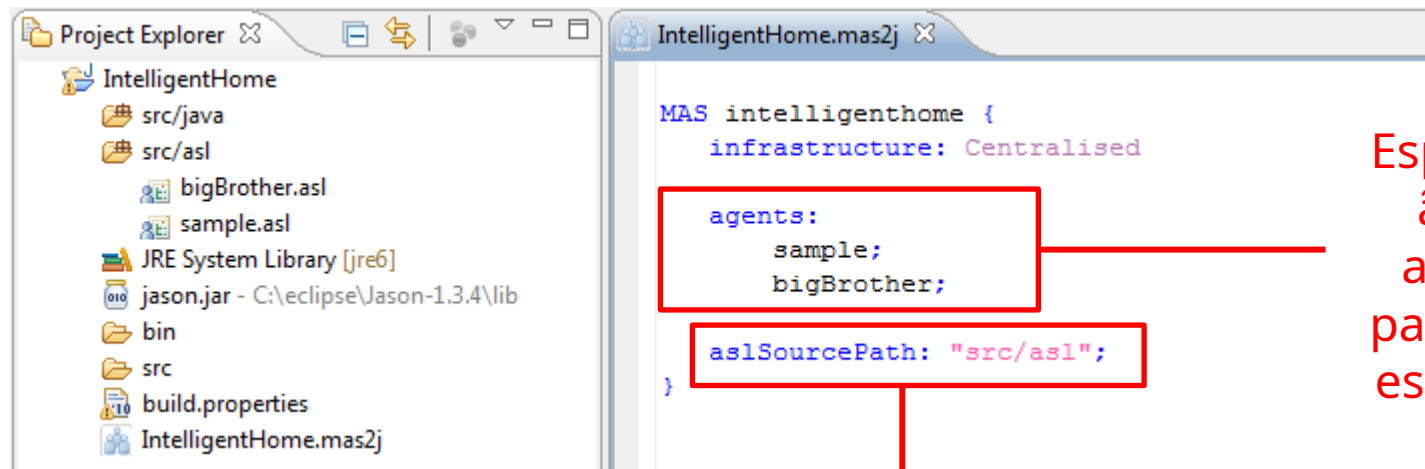


Figura 9. Estrutura inicial de um agente.

- **Configurando o SMA**



Especificação dos agentes participantes do SMA

Figura 10. O arquivo de configuração em Jason.

Especificação do endereço da pasta onde estão localizados os agentes

- **Executando o SMA**

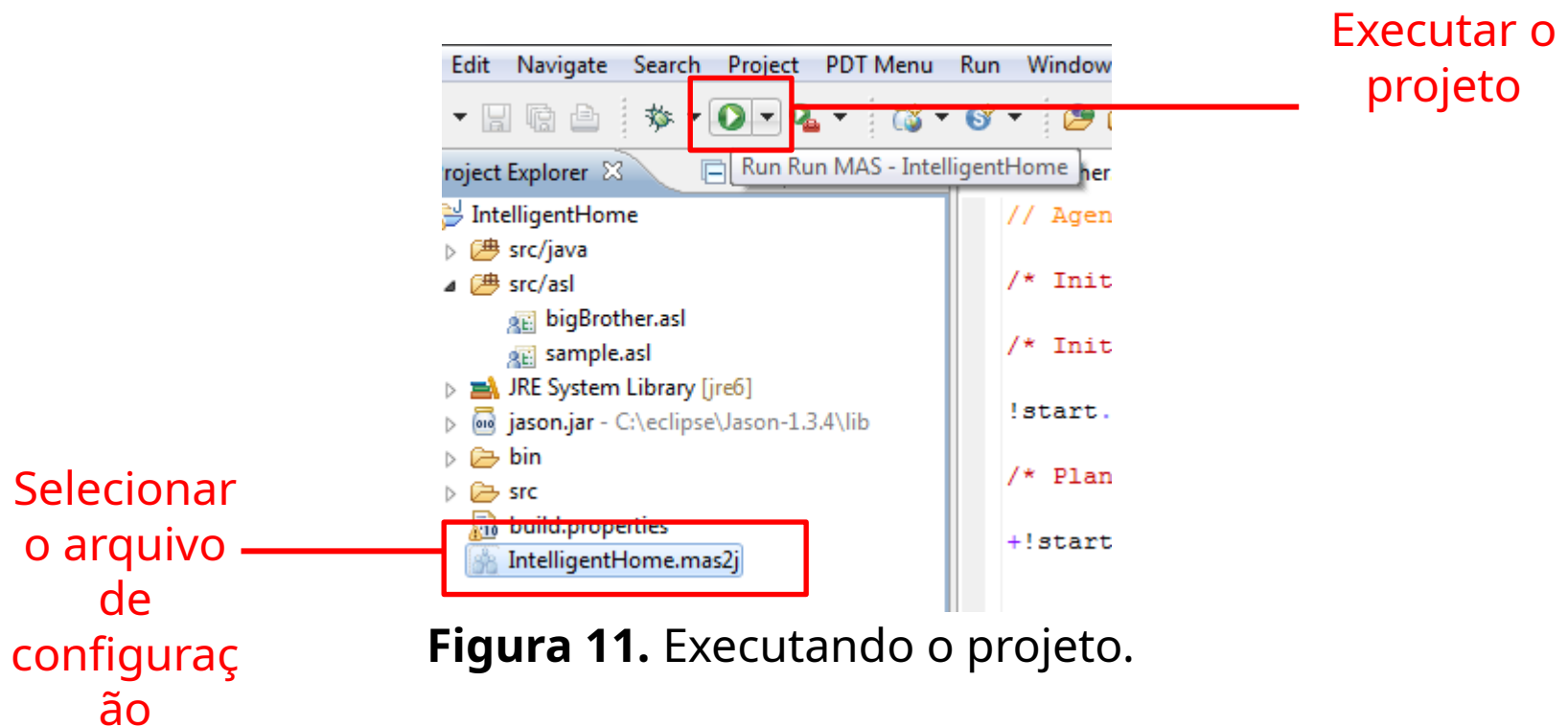


Figura 11. Executando o projeto.

- **Executando o SMA**

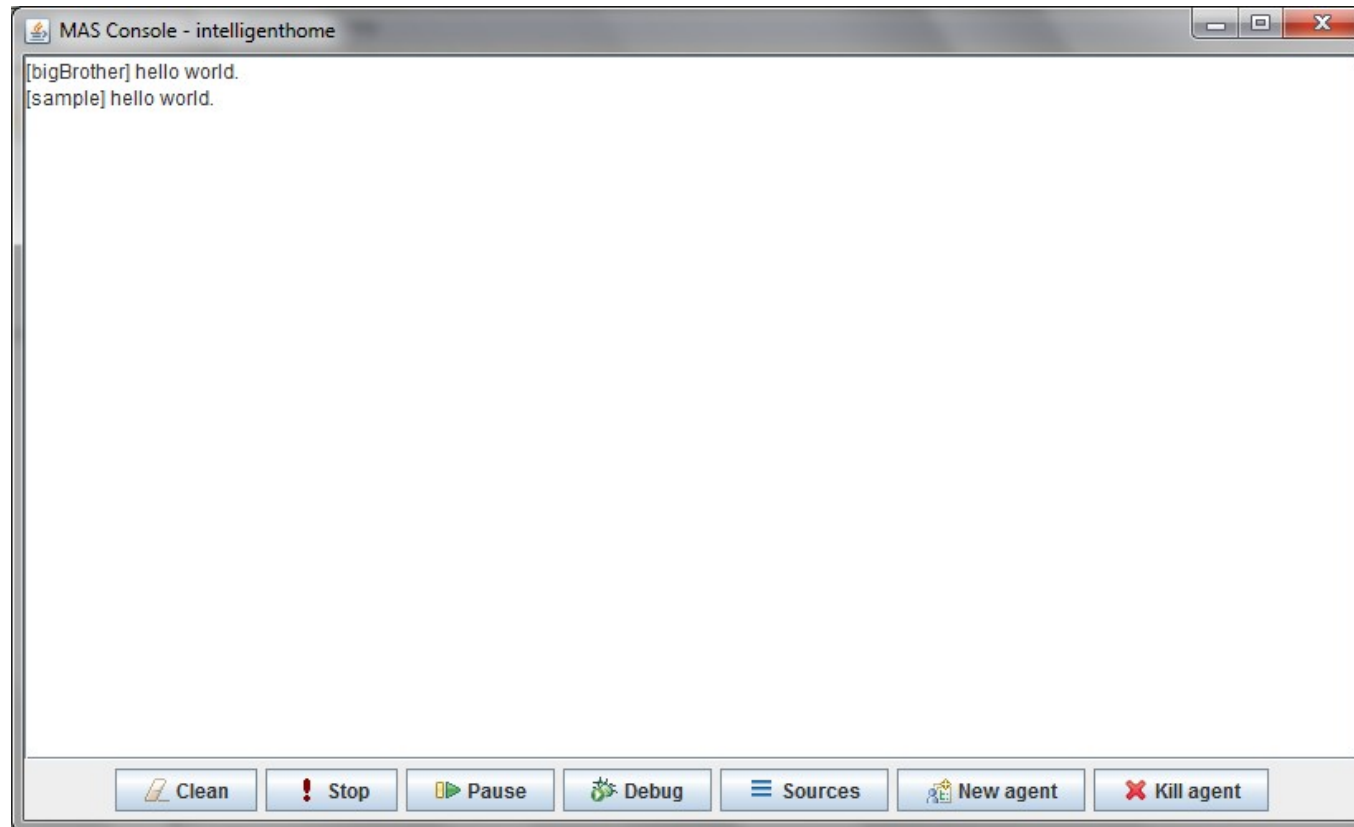


Figura 12. O console do framework Jason.

- **Debug do SMA**

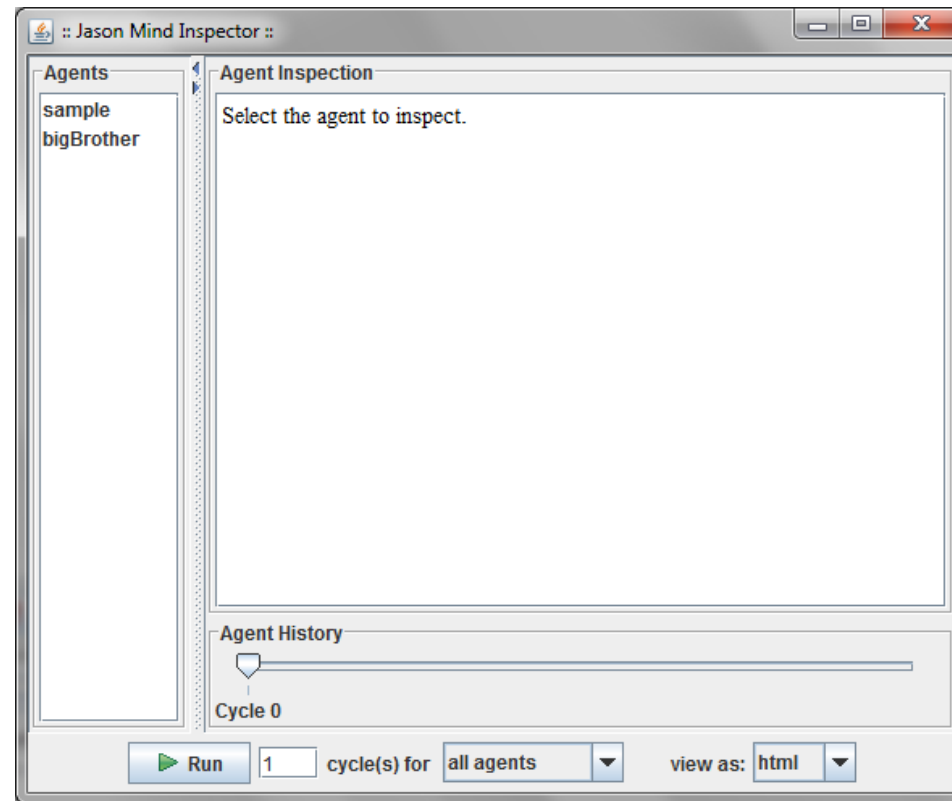


Figura 13. O debug do framework Jason.

4. Beliefs

Em Jason, um agente armazena as **informações** percebidas do ambiente; as informações internas; e informações de comunicação através de crenças.

As **crenças** são armazenadas em uma **Base de Crenças** (Belief Base).

As crenças são representadas como predicados da **lógica tradicional**. Os **predicados** representam propriedades particulares.

- **Tipos**

1. **Percepções do Ambiente (Percepts)**

Informações coletadas pelo agente que são relativas ao sensoramento constante do ambiente.

2. **Notas Mentais (Mental Notes)**

Informações adicionadas na base de crenças pelo próprio agente resultado de coisas que aconteceram no passado, promessas. Esse tipo de informação geralmente é adicionada pela execução de um plano. constante do ambiente.

3. **Comunicação**

Informações obtidas pelo agente através da comunicação com outros agentes.

- **Exemplos: Crenças Iniciais**

```
salario(5000  
  )alto(giba).  
missionStarte  
dCarro(c4,  
kadu).
```

OBS.: Toda **crença inicial** em Jason
deve terminar com .

OBS.: Toda **crença** deve começar com
letra MINÚSCULA.

- **Exemplos: Strong Negation**

~missionStarted.

~alto(giba).

~dia.

OBS.: Toda **strong negation** em
Jason deve começar com ~

- **Exemplos: Crenças Iniciais**

salario(5000
).

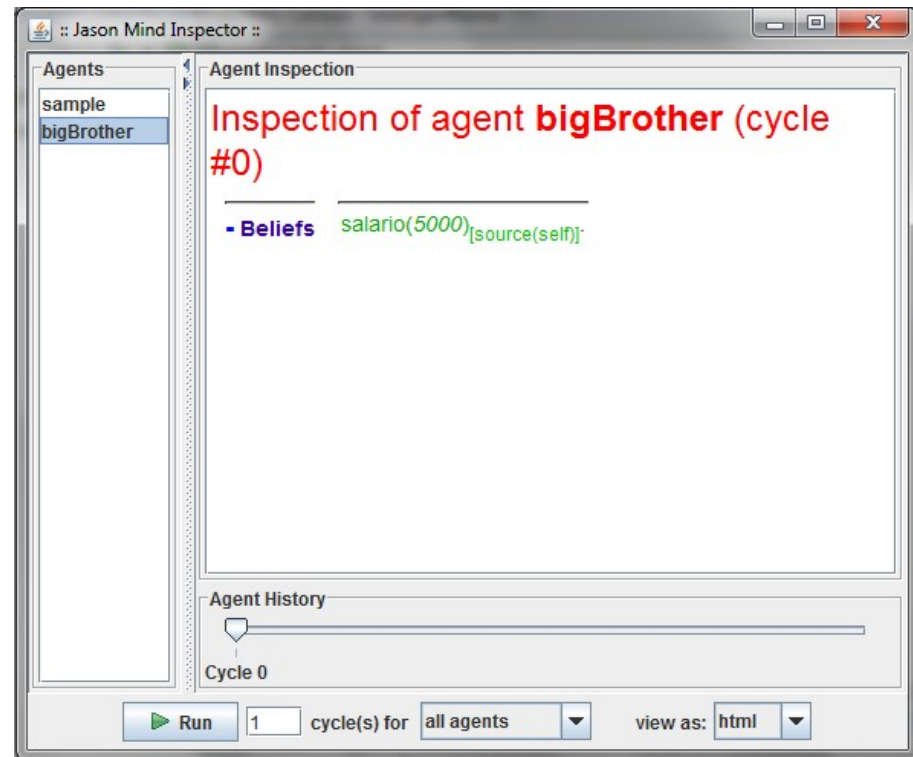


Figura 14. As crenças iniciais do agente bigBrother.

• **Exercício**

- a) Crie um SMA que simule um quarto inteligente. O quarto será controlado por um agente que começará com as seguintes crenças sobre o quarto:
 - É dia;
 - A luz do quarto está apagada;
 - A Kate está no quarto;
 - O Bob não está no quarto;
 - A temperatura atual do quarto;
- b) Adicione ao SMA um novo agente que simule o despertador. Esse novo agente deverá ter 3 crenças, incluindo o horário para despertar da Kate e do Bob.
- c) Analise no Mind Inspector as crenças dos agentes do SMA.

5. Goals

Em Jason, os **goals** (objetivos) representam os estados do mundo em que o agente deseja atingir.

- **Tipos**

1. **Achievement Goals (!)**

É um objetivo para atingir determinado estado desejado pelo agente.

2. **Test Goals (?)**

É um objetivo que tem basicamente a finalidade de resgatar informações da base de crenças do agente.

- **Exemplos: Goals Iniciais**

!start.

!thinking.

!print("Kadu").

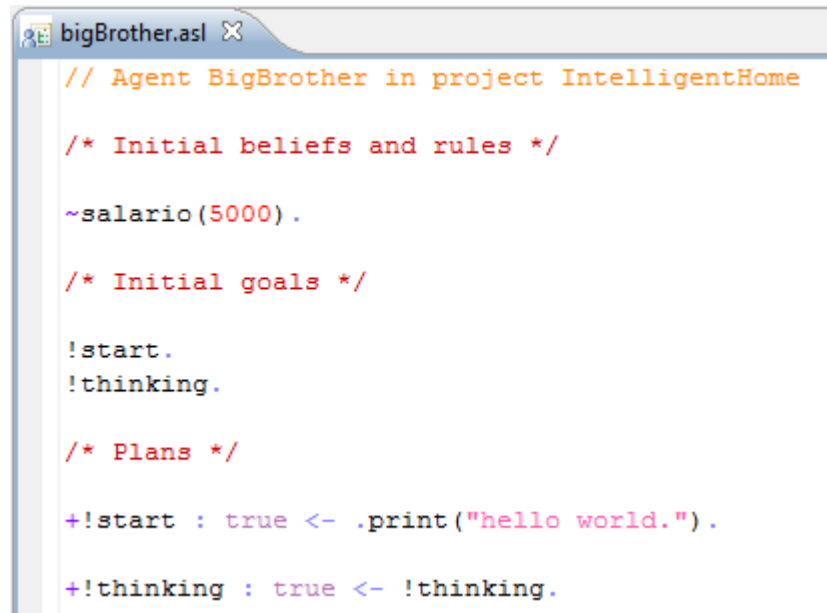
OBS.: Toda **goal inicial** em Jason deve ser um Achievement Goal; começar com **!**; e terminar com **.**

OBS.: Todo **goal** deve começar com letra **MINÚSCULA**.

- **Exemplos: Goals Iniciais**

!start.

!thinking.



```
bigBrother.asl X
// Agent BigBrother in project IntelligentHome

/* Initial beliefs and rules */

~salario(5000) .

/* Initial goals */

!start.
!thinking.

/* Plans */

+!start : true <- .print("hello world.") .

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

Figura 15. Os objetivos iniciais do agente bigBrother.

- **Exemplos: Goals Iniciais**

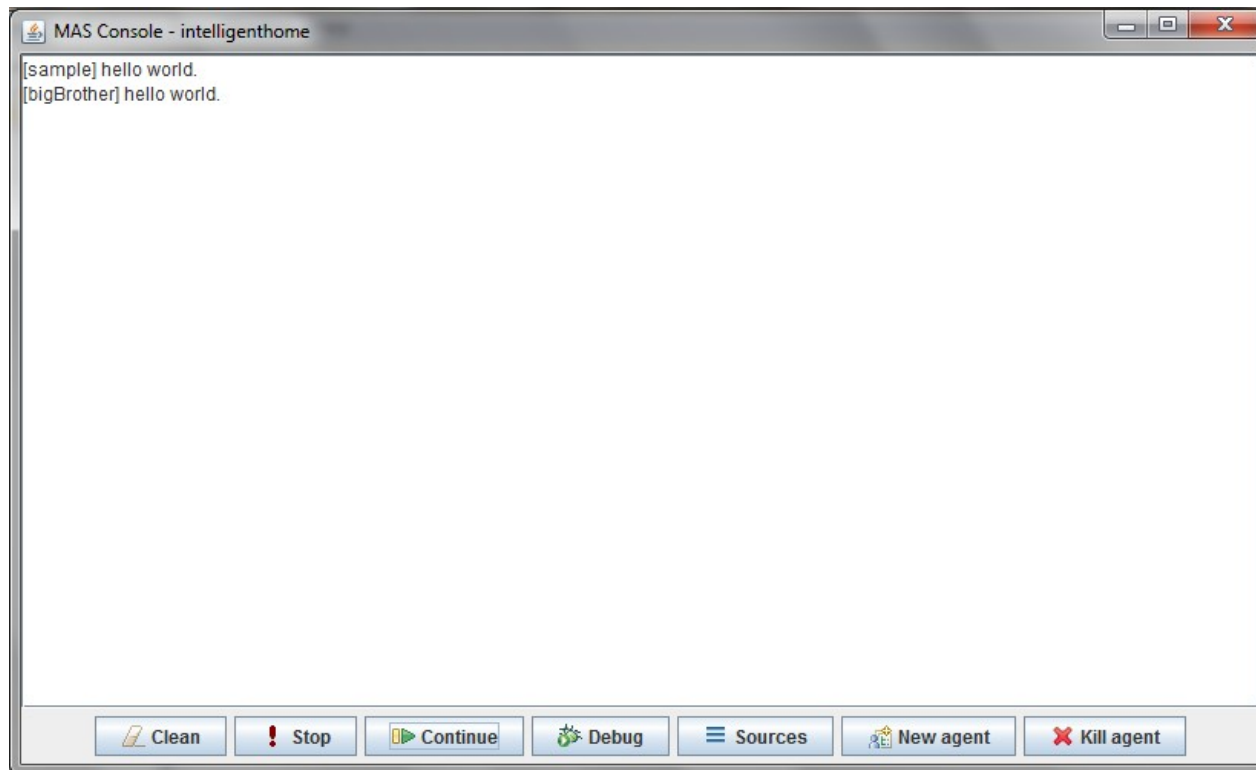


Figura 16. O console exibindo o resultado da execução.

- **Exemplos: Goals Iniciais**

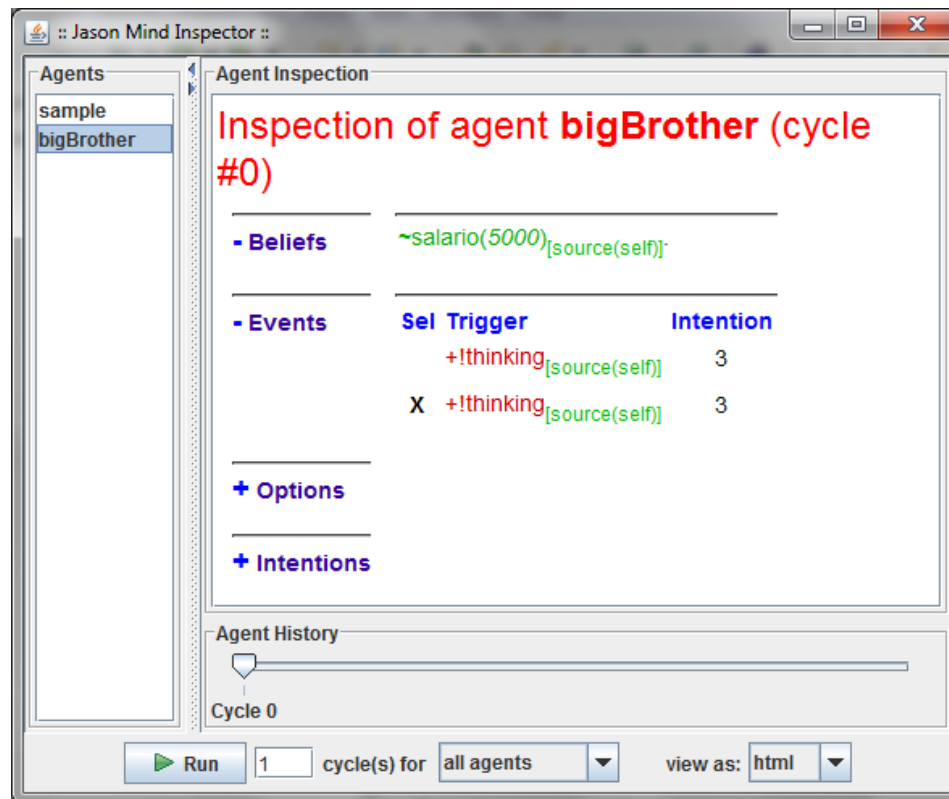


Figura 17. O debug.

• **Exercício**

- a) Adicione ao agente do quarto inteligente os seguintes objetivos iniciais. (para cada objetivo abaixo, gere um plano com a ação `.print("Objetivo")`):
 - Gerenciar a luminosidade;
 - Gerenciar as pessoas que estão no quarto;
- b) Adicione ao agente despertador um objetivo para despertar. Quando o objetivo for executado, este deve imprimir alguma mensagem. Obs.: O despertador deve despertar sem nenhuma condição específica.

6. Plans & Actions

Em Jason, um plano é composto por três partes:

Triggering_event : **context** <- **body**.

```
+!order(Product,Qtd)[source(Ag)] : true <-  
  ?last_order_id(N);  
  OrderId = N + 1;  
  +-last_order_id(OrderId);  
  deliver(Product,Qtd);  
  .send(Ag, tell, delivered(Product,Qtd,OrderId)).
```

• **Formato de um Plano**

1. Triggering Event

Um agente pode ter diversos objetivos. Os planos são ativados baseados nos eventos que podem ser ativados em determinado momento.

2. Context

São as condições para a ativação de um plano dentro vários eventos.

3. Body

É o corpo do plano. Uma sequência de ações a ser executada pelo agente.

- **Tipos de Triggering Events**

1. **Addition**

São ativados quando um plano é transformado de um desejo para uma intenção na mente do agente.

```
!bark.  
  
+!bark : true <-  
  .print("Au Au Au!").
```

- **Tipos de Triggering Events**

2. Deletion

Funciona como um tratamento de erros para planos que não possuem ativação.

```
!bark.
```

```
+!bark : dog(unknown) <-  
  .print("Au Au Au!");
```

```
-!bark <-  
  .print("sniff sniff!");  
  !bark.
```

- **Tipos de Planos**

1. **Achievement Goal**

São objetivos que os agentes se comprometem em atingir.

```
!bark.  
  
+!bark : true <-  
  .print("Au Au Au!").
```


- **Tipos de Planos**

2. Test Goal

São objetivos que recuperam informações da base de crenças.

```
!bark.  
  
+!bark : dog(unknow) <-  
        .print("Au Au Au!").  
  
-!bark <-  
        .print("sniff sniff!");  
        !sniff.  
  
+!sniff <-  
        .print("Is it bob?");  
        ?dog(X);  
        .print(X).  
  
+?dog(X) <-  
        X = bob;  
        +dog(X).
```

- **Tipos de Planos**

3. Belief

São planos ativados quando o agente adiciona ou remove uma crença da sua base de crenças

```
!sniff.
```

```
+!sniff <-  
  .print("Is it bob?");  
+dog(bob).
```

```
+dog(bob) <-  
  .print("sniff sniff!").
```

- **Ações de um Plano**

1. **Achievement e Test Goals**

São as chamadas para execução de um plano.

```
!bark.  
  
+!bark <-  
  .print("sniff!");  
  !sniff;  
  .print("sniff!").  
  
+!sniff <-  
  .print("Is it bob?");  
  ?dog(X);  
  .print(X).  
  
+?dog(X) <-  
  X = bob;  
  +dog(X);  
  .print("I found X").
```

```
!bark.  
  
+!bark <-  
  .print("sniff!");  
  !!sniff;  
  .print("sniff!").  
  
+!sniff <-  
  .print("Is it bob?");  
  ?dog(X);  
  .print(X).  
  
+?dog(X) <-  
  X = bob;  
  +dog(X);  
  .print("I found X").
```

• Ações de um Plano

2. Mental Notes

São ações que adicionam, removem ou atualizam uma crença na base de crença do agente.

```
!sniff.
```

```
+!sniff <-  
  .print("Is it bob?");  
  +dog(bob).  
  
+dog(bob) <-  
  .print("sniff sniff!").
```

```
hungry.  
food(100).  
stomach(0).
```

```
!eat.
```

```
+!eat: hungry & food(F) & stomach(S) & S<=50 <-  
  .print("Eating...");  
  -+food(F-1);  
  -+stomach(S+1);  
  .print(F);  
  !eat.
```

```
+!eat: stomach(S) & S>50 <-  
  .print("I'm Satisfied.");  
  -hungry.
```

• Ações de um Plano

3. Internal Action

São ações pré-definidas executadas dentro do raciocínio do agente.

.print	.max	.create_agent
.send	.nth	.date
.broadcast	.sort	.wait
.drop_all_desire	.substring	.random
s	.drop_all_events	.kill_agent
.my_name	.abolish	.time
.concat	.string	.perceive
.length	.count	.stopMAS
.min		

- **Ações de um Plano**

4. External Action

São ações executadas no ambiente em que o agente estiver inserido.

```
!walk.  
  
+!walk <-  
  .print("Lets sniff the environment!");  
  sniff.
```



- **Ações de um Plano**

5. Expressões

```
!bark.  
  
+!bark <-  
  .print("sniff!");  
  !sniff;  
  .print("sniff!").  
  
+!sniff <-  
  .print("Is it bob?");  
  ?dog(X);  
  .print(X).  
  
+?dog(X) <-  
  X = bob;  
  +dog(X);  
  .print("I found X").
```

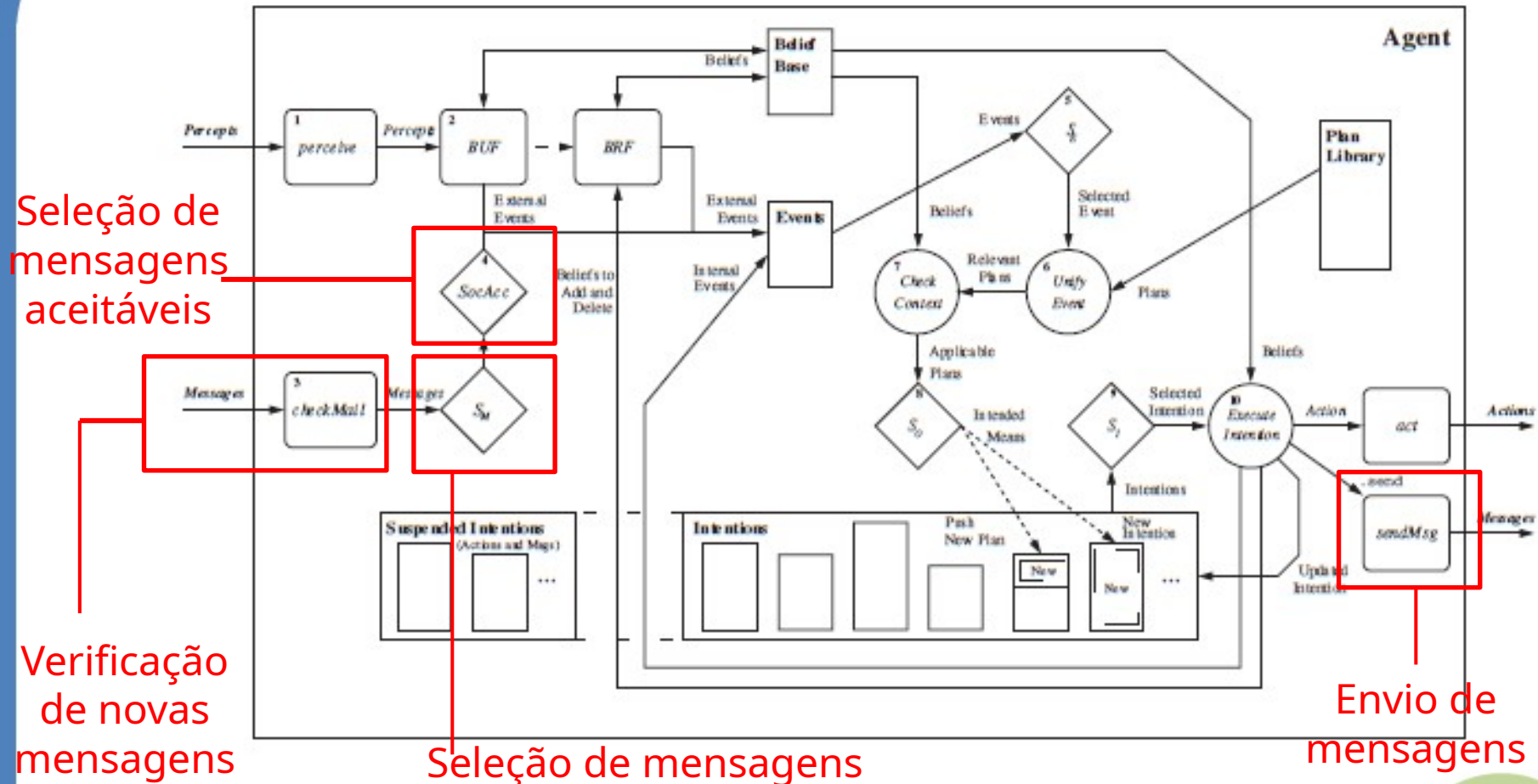

7. Comunicação Entre Agentes

- **Introdução**

No início de cada ciclo de raciocínio, o agente verifica mensagens que ele possa ter recebido de outros agentes

Baseada em *Speech Act* e *KQML*

- Reasoning Cycle



- **Estrutura**

**<sender; illocutionary forces;
content>**

i. Sender

Uma proposição atômica representando o nome do agente que enviou a mensagem.

ii. Illocutionary Forces

São as performativas que denotam as intenções do remetente.

iii. Content

Conteúdo da mensagem enviada.

- **Estrutura no Jason**

.send(receiver, illocutionary forces, propositional
.broadcast(illocutionary forces, propositional content)

- i. **Receiver**

Uma proposição atômica em AgentSpeak representando o nome do agente que enviou a mensagem.

- ii. **Illocutionary Forces**

São as performativas que denotam as intenções do remetente.

- iii. **Propositional Content**

Um termo em AgentSpeak que varia de acordo com as forças ilocucionárias.

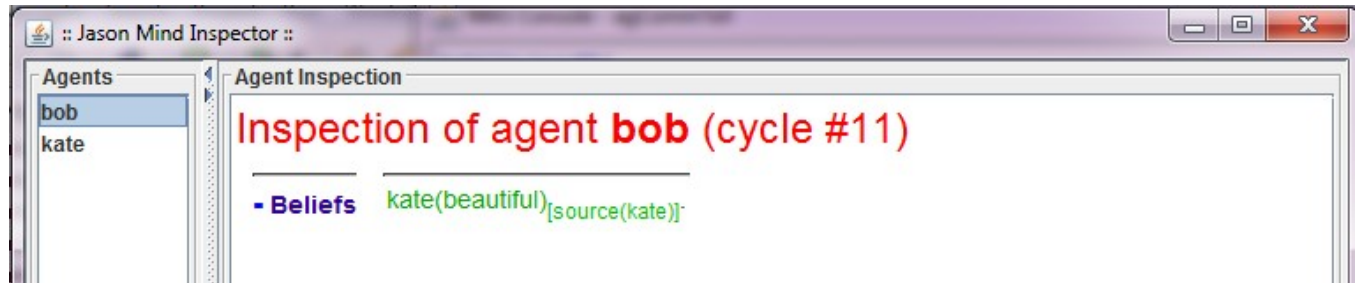
• Performativas Implementadas

1. tell

O agente remetente pretende que o receptor **acredite** que o conteúdo enviado é verdadeiro de acordo com as crenças do remetente.

Agente

```
!talkTo. Kate  
  
+!talkTo : true <-  
  .print("I'm beautiful.");  
  .send(bob, tell, kate(beautiful)).
```



- **Performativas Implementadas**

- 2. **untell**

O agente remetente pretende que o receptor **não acredite** que o conteúdo enviado é verdadeiro de acordo com as crenças do remetente.

Agente
Kate

```
!talkTo.  
  
+!talkTo : true <-  
  .print("Hi Bob, I'm Beautiful!");  
  .send(bob, tell, kate(beautiful)).  
  
+~kate(beautiful) [source(bob)] <-  
  .print("Sorry.");  
  .send(bob, untell, kate(beautiful)).
```

Agente Bob

```
+kate(beautiful) <-  
  +~kate(beautiful);  
  .print("No, You Don't!");  
  .send(kate, tell, ~kate(beautiful)).
```

- **Performativas Implementadas**

- 3. **achieve**

O agente remetente pede que o receptor **tente atingir um objetivo** de estado verdadeiro de acordo com conteúdo enviado.

Agente
Kate

```
!talkTo.  
  
+!talkTo : true <-  
  .print("Please, turn on the lights.");  
  .send(bob, achieve, turn(on)).
```

Agente Bob

```
+!turn(on) <-  
  .print("Lights On.).
```



• Performativas Implementadas

4. unachieve

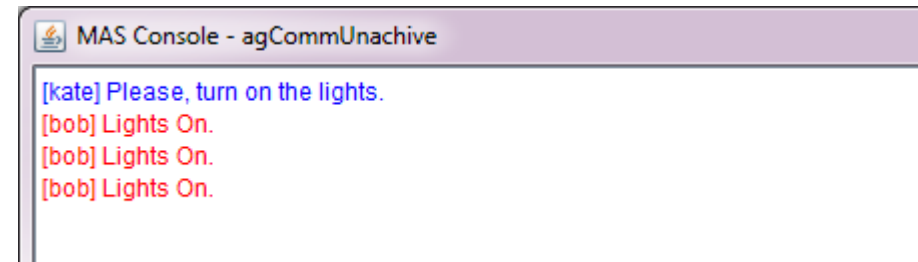
O agente remetente pede que o receptor **deixe de tentar atingir um objetivo** de estado verdadeiro de acordo com conteúdo enviado.

Agente

```
!talkTo.  
  
+!talkTo : true <-  
  .print("Please, turn on the lights.");  
  .send(bob, achieve, turn(on)).  
  
+light(on) <-  
  .send(bob, unachieve, turn(on)).
```

Agente Bob

```
+!turn(on) <-  
  .print("Lights On.");  
  .send(kate, tell, light(on));  
  !turn(on).
```



```
MAS Console - agCommUnachieve  
[kate] Please, turn on the lights.  
[bob] Lights On.  
[bob] Lights On.  
[bob] Lights On.
```

- **Performativas Implementadas**

5. askOne

O agente remetente deseja saber se a resposta do receptor para determinada questão é verdadeira.

Agente

```
!talkTo.  
  
+!talkTo : true'<-  
  .print("What's your name?");  
  .send(bob, askOne, name(Name), Reply);  
  +Reply.
```

Agente Bob

```
name(bob).
```



• Performativas Implementadas

6. askAll

O agente remetente deseja saber todas as repostas do receptor sobre uma questão.

Agente Bob

```
weather(clean).  
weather(sunny).
```

Agente

```
!goToBeach.  
  
+!goToBeach <-  
  !talkTo;  
  !analyze.  
  
+!talkTo : true <-  
  .print("What is the weather prevision?");  
  .send(bob, askAll, weather(Name)).
```



• Performativas Implementadas

7. askHow

O agente remetente deseja saber todas implementações de planos do receptor para determinado plano.

Agente

```
!talkTo.  
!turn(on).  
  
+!talkTo : true <-  
  .print("Please, Can you teach me how to turn on the lights?");  
  .wait(5000);  
  .send(bob, askHow, "+!turn(on)").
```

Agente Bob

```
+!turn(on) <-  
  .print("Lights On.").
```



The screenshot shows a window titled "MAS Console - agCommAskHow". The console output displays the following messages:

```
[kate] Please, Can you teach me how to turn on the lights?  
[kate] I don't know how to turn on the lights.  
[kate] I don't know how to turn on the lights.  
[kate] Lights On.
```

- **Performativas Implementadas**

- 8. tellHow**

O agente remetente informa ao agente receptor a implementação de um plano.

Agente Bob

```
!teach(kate).  
  
+!teach(kate) <-  
  .print("This is how we do it.");  
  .send(kate, tellHow, "+!turn(on) <- .print(\"Lights On.\").");  
  .wait(3000);  
  .send(kate, achieve, turn(on)).  
  
+!turn(on) <-  
  .print("Lights On.");
```



• Performativas Implementadas

9. untellHow

O agente remetente solicita ao agente receptor a remoção da implementação de um plano da biblioteca de planos do receptor.

Agente Bob

```
!teach(kate).

+!teach(kate) <-
  .print("This is how we dance.");
  .wait(2000);
  .send(kate, tellHow, "@d +!dance : true <- .print(\"I'm dancing with myself!\");
                                     .wait(1000); !dance.");
);
.wait(2000);
.send(kate, untellHow, "@d").

@d
+!dance : true <-
  .print("I'm dancing with myself!");
  .wait(1000);
  !dance.
```


• Performativas Implementadas

10. broadcast

Permite o uso de todas as performativas vistas anteriormente. Contudo, não é preciso identificar o agente de destino, visto que ela será enviada a todos os agentes do SMA.

Agente

```
!talkTo.  
  
+!talkTo : true <-  
  .print("I'm beautiful.");  
  .broadcast(tell, kate(beautiful)).
```



- **Por trás do Jason**

1. **Agente**

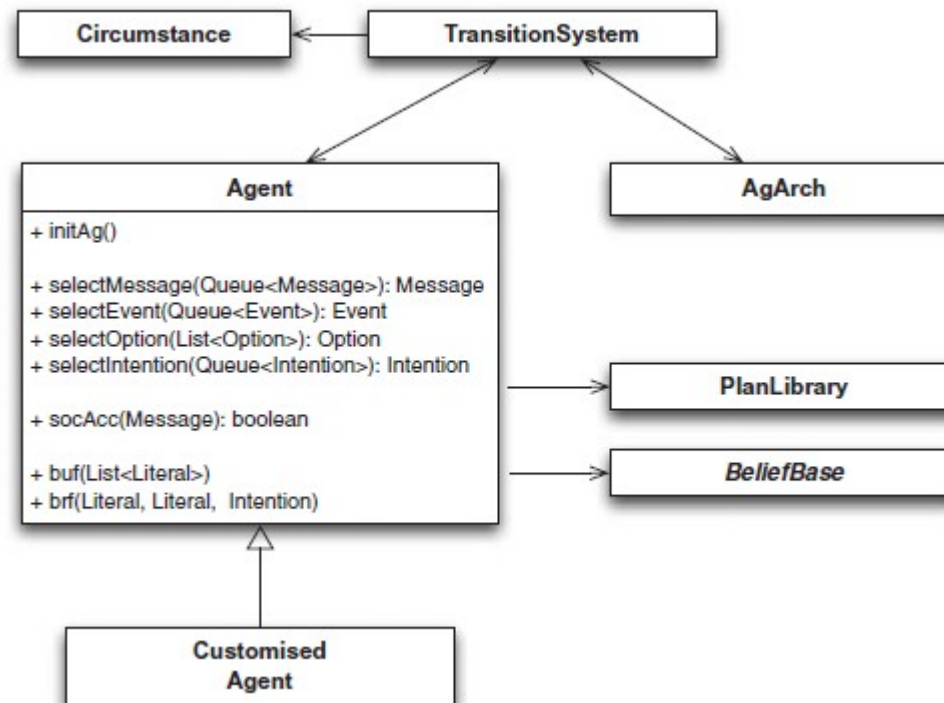


Figura 18. A arquitetura de um agente [Bordini et al., 2007].

- **Por trás do Jason**
- 2. **Arquitetura**

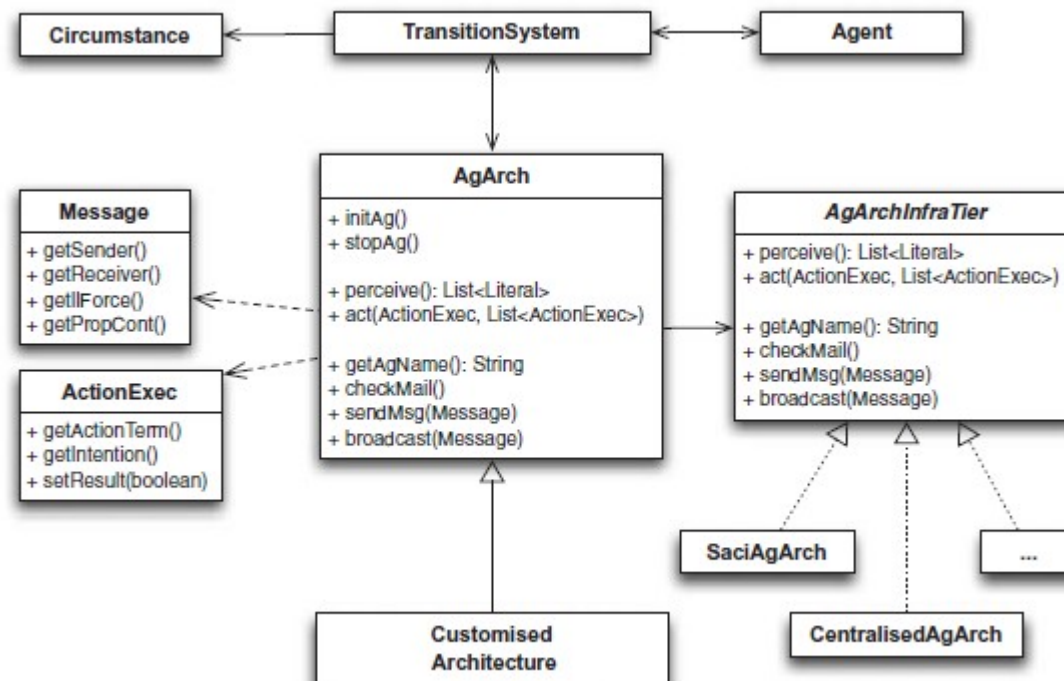


Figura 19. A arquitetura de mensagens [Bordini et al., 2007].

8. O Ambiente dos Agentes

- **Introdução**

Em Jason o ambiente é uma representação simulada por uma classe em Java com métodos padrões para a execução de ações e atualização de crenças em cada ciclo de raciocínio do agente.

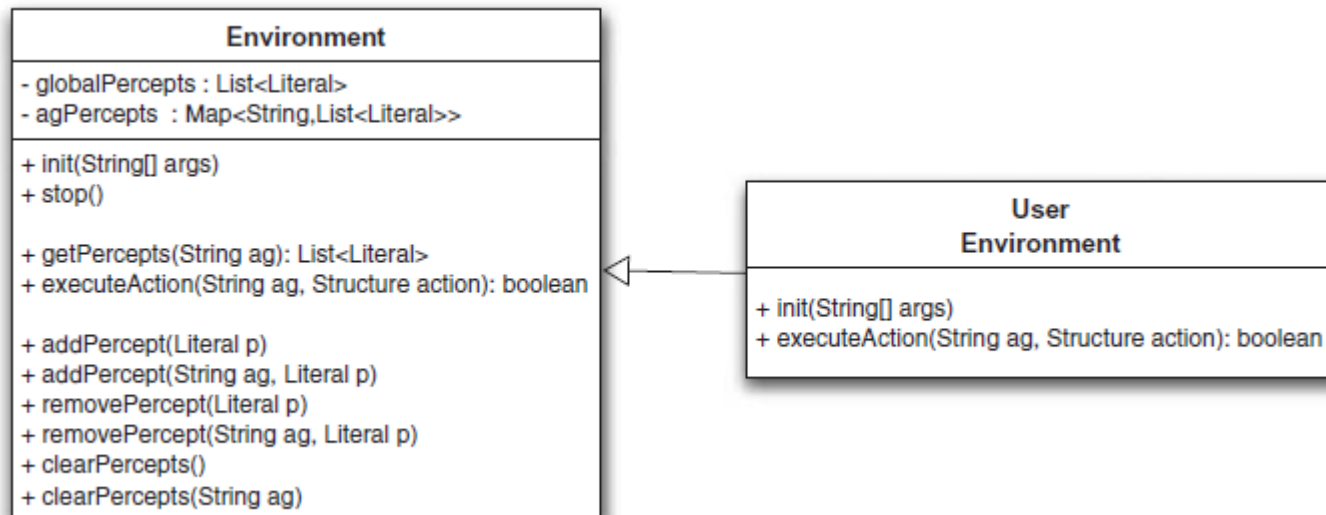
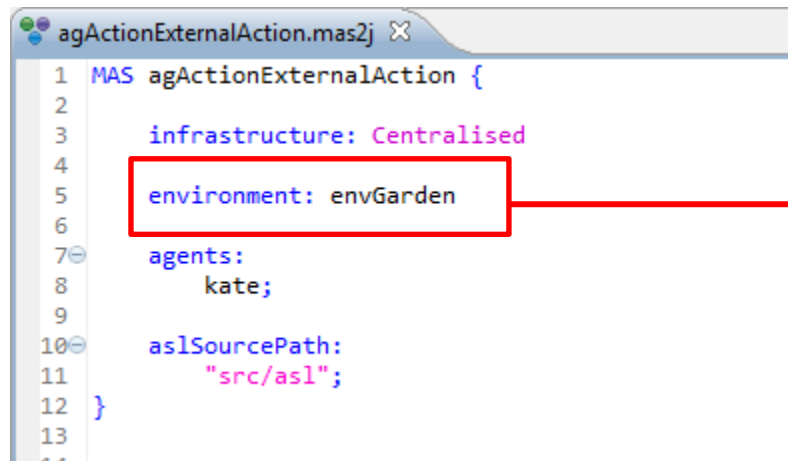


Figura 20. A arquitetura do ambiente [Bordini et al., 2007].

- **Configurando o ambiente do SMA**



```
1 MAS agActionExternalAction {
2
3   infrastructure: Centralised
4   environment: envGarden
5
6   agents:
7     kate;
8
9   aslSourcePath:
10    "src/asl";
11
12 }
13
```

Especificação da classe que representará o ambiente

Figura 21. Configuração da classe que representará o ambiente.

- **Pacote do Ambiente**

Localização
dos
arquivos
do
ambiente

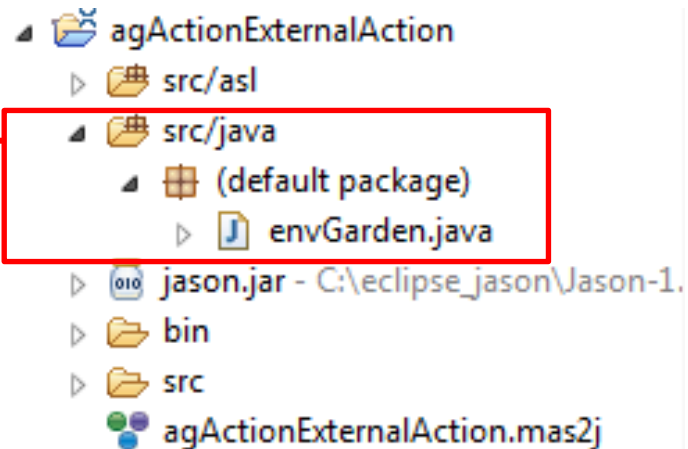



Figura 22. A localização do pacote do ambiente.

- **A classe Environment**



```
1 // Environment code for project agEnvironment
2
3 import jason.asSyntax.*;
4
5
6 public class simulatedEnvironment extends Environment {
7
8     private Logger logger = Logger.getLogger("agEnvironment."+simulatedEnvironment.class.getName());
9
10    /** Called before the MAS execution with the args informed in .mas2j */
11    @Override
12    public void init(String[] args) {
13        super.init(args);
14        addPercept(Literal.parseLiteral("percept(demo)"));
15    }
16
17    @Override
18    public boolean executeAction(String agName, Structure action) {
19        logger.info( "executing: " + action + ", but not implemented! ");
20        return true;
21    }
22
23    /** Called before the end of MAS execution */
24    @Override
25    public void stop() {
26        super.stop();
27    }
28 }
29
```

método que inicia o ambiente

método que simula a execução de uma ação no ambiente

Figura 23. A classe de ambiente auto-gerada pelo Jason.

- **Criando uma Percepção Global**

Criação de uma percepção global como atributo de uma classe em java.

```
9 private Logger logger = Logger.getLogger("agEnvironment."+simulatedEnvironment.class.getName());
10 private int food = 100;
11
12 /** Called before the MAS execution with the args informed in .mas2j */
13 @Override
14 public void init(String[] args) {
15     super.init(args);
16     addPercept(Literal.parseLiteral("food(" + this.food + ")"));
17 }
```

Atualiza (como crença para todos os agentes ao iniciar o MAS) a quantidade de alimento disponível no ambiente.

Figura 24. Criando um percept e atribuindo a crença aos agentes.

- **Programando uma Ação Externa**

```
19 @Override
20 public boolean executeAction(String agName, Structure action) {
21     if (action.getFunctor().equals("eat")) {
22         this.food -= 1;
23         clearPercepts();
24         addPercept(Literal.parseLiteral("food(" + this.food + ")"));
25         logger.info(agName + " ate 1 unit!");
26     }
27     return true;
28 }
29 }
```

Verifica se a ação executada possui alguma programação no método *executeAction*.

Figura 25. Programando a ação externa *eat*.

- **Executando o MAS com o Ambiente**

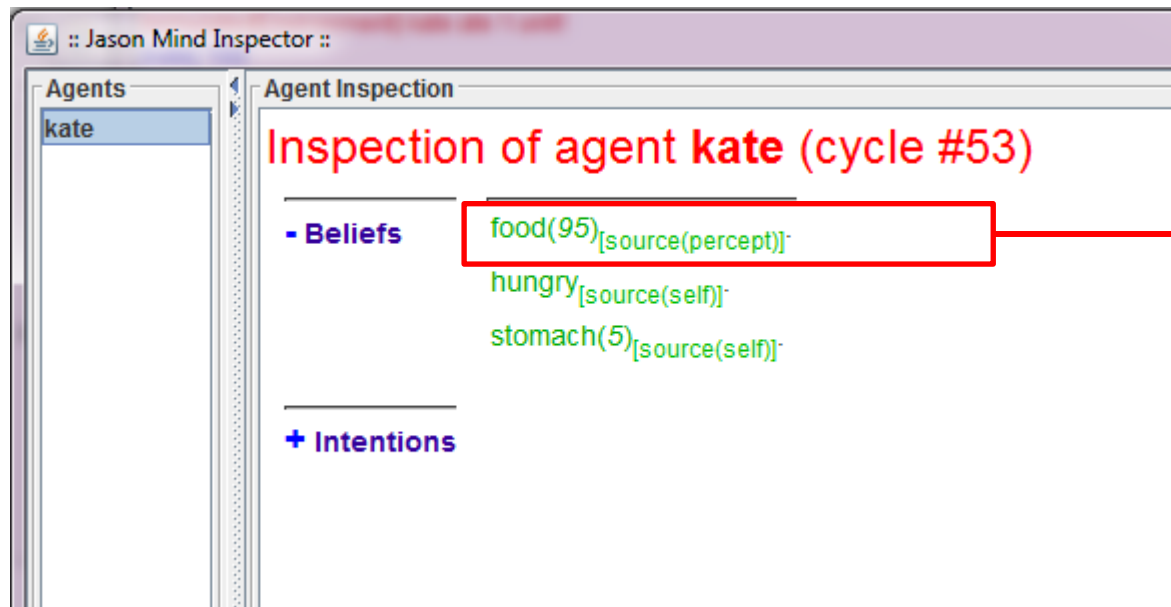
```
hungry.  
stomach(0).
```

```
!eat.
```

```
+!eat: hungry & food(F) & stomach(S) & S<=50 <-  
    .print("Eating...");  
    eat;  
    -+stomach(S+1);  
    .print(F);  
    .wait(500);  
    !eat.
```

```
+!eat: stomach(S) & S>50 <-  
    .print("I'm Satisfied.");  
    -hungry.
```

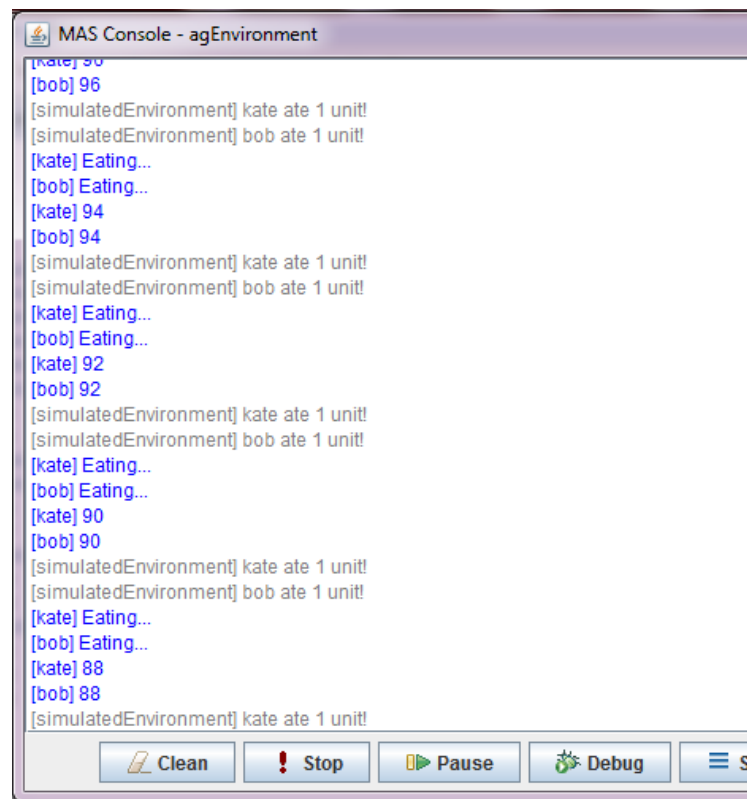
- **Debug**



A percepção global com origem do ambiente (percept).

Figura 26. O percept visto como crença pelo agente.

- **Agentes Competindo por Um Recurso**



```
MAS Console - agEnvironment
[kate] 96
[bob] 96
[simulatedEnvironment] kate ate 1 unit!
[simulatedEnvironment] bob ate 1 unit!
[kate] Eating...
[bob] Eating...
[kate] 94
[bob] 94
[simulatedEnvironment] kate ate 1 unit!
[simulatedEnvironment] bob ate 1 unit!
[kate] Eating...
[bob] Eating...
[kate] 92
[bob] 92
[simulatedEnvironment] kate ate 1 unit!
[simulatedEnvironment] bob ate 1 unit!
[kate] Eating...
[bob] Eating...
[kate] 90
[bob] 90
[simulatedEnvironment] kate ate 1 unit!
[simulatedEnvironment] bob ate 1 unit!
[kate] Eating...
[bob] Eating...
[kate] 88
[bob] 88
[simulatedEnvironment] kate ate 1 unit!
```

Figura 27. Dois agentes competindo pelo recurso *food* no ambiente simulado.

9. Conclusão

Este curso apresentou uma introdução ao **Sistemas Multi-Agentes** usando o Framework Jason.

O curso é **introdutório** contudo é possível criar SMA concisos e completos para uma realidade simulada.

Atualmente é utilizado o **CArtAgO** para a implementação de artefatos de ambiente e o **Moise** para implementação de organizações e papéis.

10. Referências Bibliográficas

Boissier O, Bordini RH, Hübner JF, Ricci A, Santi A. Multi-agent oriented programming with JaCaMo. *Science of Computer Programming*. 2013 Jun;6(1): 747-761.

Bordini RH, Hubner JF, Wooldridge W. *Programming Multi-Agent Systems in AgentSpeak using Jason*. John Wiley and Sons, London, 2007

Bratman, M. Intentions, Plans, and Practical Reason. Harvard University Press, 1987.

Huber MJ. Jam: a bdi-theoretic mobile agent architecture. In Proceedings of the third annual conference on Autonomous Agents, AGENTS '99, pages. 236-243, New York, 1999.

Winikoff M. Jack intelligent agents: An industrial strength platform. Em Bordini R, Dastani M, Dix J, Fallah AS, Weiss G, editors. Multi-Agent Programming, volume 15 of Multiagent Systems, Artificial Societies, and Simulated Organizations, pages. 175-193. Springer US, 2005.

Wooldridge, M. (2000). *Reasoning about rational agents. Intelligent robotics and autonomous agents. MIT Press.*

Wooldridge M. An Introduction to MultiAgent Systems. John Wiley & Sons, 2009.

Zambonelli F, Jennings NR, Omicini A, Wooldridge M. Agent-Oriented Software Engineering for Internet Applications. In: Omicini A, Zambonelli F, Klusch M, Tolksdorf R, editors. Coordination of Internet Agents. Springer Verlag; 2001. p.326-345.

Obrigado.

pantoja@cefet-rj.br

Agradecimentos: