Integração da plataforma JaCaMo

Henrique Donâncio N. Rodrigues[1], **Diana Francisca Adamatti**[1]

[1]Centro de Ciências Computacionais – Universidade Federal do Rio Grande (FURG)

henriquedonancio@gmail.com

Data de criação: 11/07/2017
Data da última modificação: 11/07/2017

Jason

A plataforma Jason [Bordini et al. 2007] é um interpretador de uma extensão da linguagem AgentSpeak, conhecida como AgentSpeak(L) [Rao 1996], baseada na arquitetura BDI. Jason foi construído de forma a direcionar o comportamento, sem que o programador tenha de se preocupar com isso.

As origens do modelo de agentes BDI *(Believe - Desire - Intentions)* residem na teoria do raciocínio prático humano desenvolvida pelo filósofo Michael Bratman [Bratman 1987]. O conceito de raciocínio prático envolve (i) decidir que "estado do mundo" se quer alcançar e (ii) "como" ele será alcançado. Em agentes BDI, o processo (i) é chamado deliberação, e o processo (ii) é conhecido como raciocínio meio-fim. As definições formais de deliberação e raciocínio são [Bratman 1987, Woolridge and Wooldridge 2001]:

- Deliberação: O processo de deliberação resulta em um agente adotar uma intenção.

- Raciocínio meio-fim: O raciocínio meio-fim é o processo de decidir como alcançar um fim (ou seja, uma intenção que o agente possui) usando os meios disponíveis (ou seja, as ações que o agente pode executar).

Intenções no contexto do raciocínio prático são usadas para caracterizar ações. Se um agente possui uma intenção ele irá satisfazer as etapas, ou seja, tomar um plano de ações para satisfazer tal intenção. Nesse sentido, intenções tem um papel importante na tomada de ações.

Na arquitetura BDI, as crenças representam o conhecimento que o agente tem sobre o ambiente e os agentes que participam dele, inclusive de si mesmo. Já os desejos representam os estados desejáveis que o sistema poderia apresentar e é um potencial motivador das ações do agente. Por fim, tem se as intenções, que como já dito, determina o processo de raciocínio prático.

A plataforma Jason é baseada na arquitetura BDI, portanto as crenças no Jason têm a função de representação do conhecimento. Um agente nessa arquitetura está constantemente percebendo o ambiente, alterando sua base de conhecimento, raciocinando sobre como agir de modo a atingir seus objetivos, e em seguida, agindo de modo a mudar o ambiente. A parte de raciocínio (prático) do comportamento cíclico do agente na

linguagem AgentSpeak é feita de acordo com os planos que o agente tem em sua biblioteca de planos. Inicialmente, esta biblioteca é formada pelos planos que o programador escreve como um programa AgentSpeak.

a. Crenças

Um agente em AgentSpeak tem uma base de crenças, que em sua forma mais simples é uma coleção de literais, como na programação lógica tradicional. Em linguagens de programação inspiradas por lógica, a informação é representada em forma simbólica por predicados tais como:

(i) *baixo(pedro)*
(ii) *joga(pedro, basquete)*

Em (i) uma propriedade em particular é expressa, no caso a relação entre *Pedro* e o predicado *baixo*. Em (ii) é representado o fato de que uma determinada relação mantém-se entre dois ou mais objetos, no caso para expressar que *Pedro* joga *basquete*. No Jason, crenças podem ser adicionadas/removidas da base de crenças de um agente através de um evento ativador do tipo +*b*, para adição e -*b* para remoção. O evento ativador *?b* serve para buscar uma crença na base do agente, semelhante a procedimentos em Espaço de Tuplas [**?**] em sistemas distribuídos. A diferença está no fato que no Jason a princípio esse espaço de memória é exclusivo do agente e a consulta à base de crenças é determinística seguindo a estrutura de uma *pilha*. A consulta à base de crenças somente cria uma cópia da crença ao termo associado [**?**].

(iii) *?joga(pedro, X)*
(iv) *?joga(_, basquete)*

Na consulta (iii), a variável *X* irá assumir o valor mais alto da pilha da base de crenças, caso exista. Em (iv), o símbolo *"_"* despreza o primeiro predicado, e somente se basea para a consulta no termo *joga(_, X)*, onde X tem como valor o objeto *basquete*, retornando *verdadeiro* caso exista.

b. Objetivos

As crenças expressam propriedades que o agente acredita serem verdadeiras do mundo em que está situado, os objetivos por sua vez expressam as propriedades dos estados do mundo que o agente deseja alcançar. Embora isso não seja imposto pela linguagem AgentSpeak, normalmente, ao representar um objetivo *g* em um programa de agente, significa que o agente está comprometido a agir de modo a mudar o mundo para um estado no qual *g* se torne realmente verdade [Bordini et al. 2007]. No Jason, o evento ativador de um objetivo é denotado pelo operador *"!"*, sendo assim um plano qualquer *g* é ativado pela expressão *"!g"*.

c. Planos

A terceira construção essencial de um programa de um agente seguindo a arquitetura BDI são os planos, que ditam como um objetivo deve ser satisfeito. Alterações em crenças e em objetivos desencadeiam esses planos. Um plano em Jason é constituído das seguintes partes:

(i) evento-ativador : contexto <- corpo

O *eventor-ativador* tem a função de informar para cada um dos planos na biblioteca de planos do agente, quais eventos ele deve ser utilizado de acordo com certas condições dadas pelo *contexto*. Essas condições devem ser verificadas no *contexto* uma vez que o ambiente onde o agente está situado pode ser dinâmico, e alterações nas suas percepções ditam as condições em que ele atua no ambiente. Por fim, tem se o corpo do plano, que é o curso da ação que o agente irá executar. O corpo de um plano pode conter subobjetivos, necessários para a satisfação do objetivo, sendo a execução do corpo de um plano determinística. Um exemplo de um plano pode ser verificado abaixo:

$$+!\text{preparar\_jantar(Amigos)} : \text{Amigos} \;\&\; \text{lugares\_na\_mesa(N)} \;\&\; \text{Amigos} < \text{N} <\text{- ... .}$$

d. Ações Internas

As ações internas são ações executadas fora da "mente" do agente. Uma importante característica das ações internas é que elas são capazes de *mudar o ambiente*. Com as ações internas é possível expandir a programação de linguagem com operações que não estão disponíveis de outra forma.

Exemplos

Quando um agente é criado no Jason, um evento ativador "!start" e um plano "+!start: true <- .print("hello world.")" são criados, como mostra o código abaixo. O contexto com a condição *true* significa que sempre que for possível executar o plano, ele será executado.

```
1
2  /* Initial beliefs and rules */
3
4  /* Initial goals */
5
6  !start.
7
8  /* Plans */
9
10 +!start : true <- .print("hello world.").
```

Exemplo 1: Estruturas condicionais

Vamos trabalhar o "contexto" dos planos. Para estes exemplos crie um projeto com o *infraestructure: Centralised* em *New Project*. Adicione um agente a este projeto com o nome *ag1* e todos os demais campos em branco.

O código abaixo mostra exemplos de planos que possuem um comportamento similar a uma estrutura condicional **IF** em linguagens procedurais. Para este caso, definiu-se uma regra *valor(10)*. Nos planos *+!start* subsequentes, em cada contexto é definida uma condição diferente e a condição que satisfizer a condição irá imprimir uma mensagem através da ação interna *.print*.

```
1  /* Initial beliefs and rules */
2
3  valor(10).
```

```
4
5  /* Initial goals */
6
7  !start.
8
9  /* Plans */
10
11 +!start : valor(X) & X < 0 <- .print("O valor é menor que 0.").
12 +!start : valor(X) & X < 10 <- .print("O valor é menor que 10.").
13 +!start : valor(X) & X < 20 <- .print("O valor é menor que 20.").
```

Se duas ou mais condições são satisfeitas, o Jason irá executar aquela que primeiro satisfaz as condições necessárias, analisando-as de forma sequencial. No código a seguir, por exemplo, o plano ativado é o plano na linha 13.

```
1
2  /* Initial beliefs and rules */
3
4  valor(10).
5
6  /* Initial goals */
7
8  !start.
9
10 /* Plans */
11
12 +!start : valor(X) & X < 0 <- .print("O valor é menor que 0.").
13 +!start : valor(X) & X < 50 <- .print("O valor é menor que 50.").
14 +!start : valor(X) & X < 100 <- .print("O valor é menor que 100.").
```

Exemplo 2: Estruturas de repetição

No Jason, planos podem conter eventos ativadores de outros planos. No código abaixo o plano *+!start* invoca a si mesmo, causando assim uma iteração "infinita".

```
1  contador(0).
2
3  /* Initial goals */
4
5  !start.
6
7  /* Plans */
8
9  +!start : contador(X) & X < 10
10    <- .print("Iteração: ",X);
11    Y = X + 1;
12    -+contador(Y);
13    !start.
14
15 +!start: true.
```

Adicionando a esse plano um *contexto*, pode se obter uma estrutura condicionada, por exemplo, um contador, como mostra o código:

```
1  /* Initial beliefs and rules */
2
3     contador(0).
4
5  /* Initial goals */
6
7  !start.
8
9  /* Plans */
10
11 +!start : contador(X) & X < 10
12    <- .print("Iteração: ",X);
13    Y = X + 1;
14    -+contador(Y);
15    !start.
16
17 +!start: true.
```

É importante notar que no Jason, um evento ativador necessita encontrar um *plano* ao qual o *contexto* seja válido. No caso do código do Exemplo 2 acima , o plano da linha 17 serve para quando o valor X da crença *contador(X)* atinja o valor igual a 10. Na linha 13 o valor dessa crença é incrementado e na linha 14 o valor na crença é atualizado.

Comunicação

Nesse exemplo iremos utilizar a comunicação entre agentes. A comunicação em agentes é importante uma vez que permite a troca de conhecimentos (crenças) entre eles. No Jason, a comunicação é usualmente feita através da ação interna *.send* que envia uma mensagem ao agente destinatário. Basicamente essa primitiva é estruturada da seguinte forma:

.send(destinatário, objetivo, mensagem)

- **Destinatário**: A quem (um agente ou uma lista de agentes) a mensagem se destina;
- **Objetivo (comunicativo)**: Determina o objetivo da comunicação do agente (força locucionaria). No Jason elas podem ser *tell* (adiciona uma crença no agente destinatário), *achieve* (desencadeia um plano no agente destinatário), *askOne* (pergunta algo ao agente destinatário).
- **Mensagem**: Conteúdo da mensagem

Uma ação *.send* no Jason também pode ter parâmetros adicionais, por exemplo, um parâmetro que recebe a resposta das performativas *askOne* e *askAll* e um *timeout*, que define um tempo de espera aguardando a resposta (em milissegundos), como mostra os exemplos abaixo:

- *.send(destinatário, objetivo, mensagem, Resposta)*
- *.send(destinatário, objetivo, mensagem, Resposta, timeout (em milissegundos))*

Vale destacar que o parâmetro “Resposta” é escrito com inicial maiúscula por se tratar de uma variável. Variáveis no Jason tem iniciais ou são letras maiúsculas.

No Jason, os agente podem ter mais de um *evento ativador*. No caso do código abaixo, o agente tem os eventos ativadores *!iniciar* e *!comunicar*. Esses eventos são disparados por ordem sequencial *(1)!iniciar* e *(2)!comunicar*.

Exemplo 3.0: Eventos ativadores

```
/* Initial beliefs and rules */

/* Initial goals */

!iniciar.

!comunicar.

/* Plans */

+!iniciar : true <- .print("Olá mundo!").

+!comunicar : true <- .send(ag2, tell, saudacoes).
```

Exemplo 3.1: Comunicação entre agentes

Abaixo são apresentados dois agentes que se comunicam. O agente ag_ex3_1 envia uma mensagem ao agente ag_ex3_2 e esse responde usando uma performativa *tell*.

```

// Agent ag_ex3_1 in project Exemplo1.mas2j

/* Initial beliefs and rules */

dia(domingo).

/* Initial goals */

!iniciar.

/* Plans */

+!iniciar : dia(domingo)
   <- .send(ag_ex3_2, tell, assistirJogo).

+conviteAceito[source(Ag)]: true
   <- .print("O agente ", Ag, " aceitou meu convite").
```

```

// Agent ag_ex3_2 in project Exemplo1.mas2j

/* Initial beliefs and rules */

/* Initial goals */

/* Plans */

+assistirJogo[source(Ag)]: true
   <- .print("O agente ", Ag, " está me convidando para assistir o jogo
      ");
   .send(Ag, tell, conviteAceito).
```

Com a anotação *source(X)* é possível identificar a fonte que desencadeia aquela

crença. Isso é útil quando o agente possui várias fontes que desencadeiam a ativação de uma mesma crença.

CArtAgO

O arcabouço CArtAgO (Common ARTifact infrastructure for AGents Open environments) [Ricci et al. 2014] é baseado em agentes e artefatos (A & A) [Ricci et al. 2007] para modelar e projetar sistemas multiagentes. Com esta ferramenta pode-se criar artefatos estruturados em espaços abertos onde os agentes podem se unir para trabalhar em conjunto. Os artefatos são recursos e ferramentas, construídos dinamicamente, manipulados e usados por agentes para apoiar/realizar suas atividades individuais e coletivas.

Os principais elementos utilizados no CArtAgO são as *propriedades observáveis*, *operações* e *sinais*. *Propriedades observáveis* são elementos do artefato que o agente passa a observar de forma que qualquer operação sobre esse elemento será percebida. As *operações* são implementadas para fornecer uma interface para utilização dos recursos do artefato pelos agentes, com parâmetros de entrada e retorno *(OpFeedbackParam)*. Finalmente, os sinais *(Signals)* permitem que seja detectado pelo agente uma alteração em sua base de crenças, alterando-a. *Signals* podem ser úteis para desencadear ações a partir de artefatos, ou mesmo adicionar algo ao conjunto de *verdades* que o agente acredita.

Para utilizar CArtAgO e Jason é necessário alguns ajustes ao nosso projeto. O primeiro no ambiente do nosso projeto que agora deverá ser do tipo *c4jason.CartagoEnvironment*. O segundo é adaptar nossos agentes a essas ferramentas incluindo na frente dos seus nomes a sua classe *agentArchClass c4jason.CAgentArch* e o terceiro é incluir os pacotes que fornecem abstrações para trabalhar com essas ferramentas ao nosso projeto. Abaixo é apresentado a declaração da classe principal de um projeto mas2j com dois agentes em um projeto com um ambiente apto a integração Jason-CArtAgO.

```
1
2  MAS exemplo {
3
4     infrastructure: Centralised
5
6     environment: c4jason.CartagoEnvironment
7
8     agents:
9        ag1          agentArchClass c4jason.CAgentArch;
10       ag2          agentArchClass c4jason.CAgentArch;
11
12
13    classpath:
14         "lib/cartago.jar"; // cartago: arcabouço para artefatos
15         "lib/c4jason.jar"; // integração jason-cartago
16
17    aslSourcePath:
18       "src/asl";
19
20 }
```

Os arquivos com extensão *.jar* devem ser incluídos na pasta *lib* do projeto criado ou outra pasta qualquer, desde que devidamente referenciados no “Classpath”. Os arqui-

vos *.jar* usualmente estão disponíveis junto aos arquivos descarregados junto a plataforma JaCaMo [Bordini and Hübner 2014]. Arquivos com extensão *.asl* são agentes Jason.

Abaixo é apresentado como alguns tipos primitivos são mapeados do Jason para CArtAgO e vice-versa [Ricci et al. 2014, Ricci et al. 2010].

CArtAgO para Jason:

- booleano é mapeado para booleano
- int, long, float, double são mapeados para double
- String é mapeada para String
- arrays são mapeados em listas
- Os objetos em geral são mapeados por átomos cobj_XXX que funcionam como referência de objeto

Jason para CArtAgO

- booleano é mapeado para booleano
- um termo numérico é mapeado para o menor tipo de número que seja suficiente para conter os dados
- String é mapeada para um objetos String
- estruturas são mapeadas em objetos String
- listas são mapeadas em arrays
- átomos cobj_XXX referindo-se a objetos são mapeados no objeto referenciado

Criando artefatos CArtAgO

A primeira coisa a se saber é que artefatos CArtAgO são criados a nível de agentes, ou seja, isso significa que os agentes são responsáveis por sua criação. O código abaixo apresenta a criação de um artefato no modelo CArtAgO:

```
1
2  // Agent ag1 in project Exemplo.mas2j
3
4  /* Initial beliefs and rules */
5
6  /* Initial goals */
7
8  !criar_artefato.
9
10 /* Plans */
11
12 +!criar_artefato : true
13    <- makeArtifact("calculadora", "artefatos.Calculadora",[],CALC).
```

A criação de um artefato CArtAgO é feita através de planos, no caso do código acima o plano *+!criar_artefato* utiliza a ação *makeArtifact()* para criar o artefato "calculadora" que está na subpasta "artefatos" do projeto. Para criar o artefato, o agente explora a ação *makeArtifact*, fornecida pelo artefato do espaço de trabalho. Uma lista vazia de parâmetros é especificada, e o *id* do artefato é recuperado, vinculado à variável que nesse caso é a variável CALC [Ricci et al. 2010].

A implementação do artefato "calculadora" em Java está no código abaixo. O código foi extraído dos exemplos disponíveis na página do projeto CArtAgO [Ricci et al. 2010].

```
import cartago.*;

public class Calculadora extends Artifact {

@OPERATION
   void sum(double a, double b, OpFeedbackParam<Double> sum){
   sum.set(a+b);
}

@OPERATION
   void sumAndSub(double a, double b, OpFeedbackParam<Double> sum,
   OpFeedbackParam<Double> sub){
   sum.set(a+b);
   sub.set(a-b);
   }
}
```

Em Java, um artefato no modelo CArtagO herda propriedades da classe *Artifact*. A assinatura @OPERATION serve para o agente executar ações do artefato. *OpFeedbackParam<Tipo>* é um parâmetro de retorno que é atribuído através de *set*. Abaixo tem-se o código completo do agente capaz de criar e usar o artefato "calculadora".

```

/* Initial beliefs and rules */

/* Initial goals */

!criar_artefato.
!usar_artefato.

/* Plans */

+!criar_artefato : true
   <- .print("Criando artefato...");
   makeArtifact("calculadora", "artefatos.Calculadora",[], CALC).

+!usar_artefato : true
   <- sum(4,5,Sum)[artifact_id(CALC)];
   .print("A soma e ",Sum);
   sumAndSub(0.5, 1.5, NewSum, Sub)[artifact_id(CALC)];
   .print("A nova soma e ",NewSum," e a subtracao e ",Sub).
```

O *[artifact_id(ID)]*, serve para identificar em qual artefato será executada a ação.

Observando artefatos CArtAgO

Esse exemplo extraído da página do CArtAgO [Ricci et al. 2010] mostra como definir propriedades observáveis em um artefato. Propriedades observáveis são elementos do artefato que o agente passa a "acompanhar", e quando esse elemento é modificado ou excluído o agente percebe e atualiza suas crenças.

O código completo desse exemplo está disponível em: http://cartago.sourceforge.net/?page_id=69

```
import cartago.*;
```

```
2
3  public class Counter extends Artifact {
4
5    void init(){
6      defineObsProperty("count",0);
7    }
8
9    @OPERATION void inc(){
10     ObsProperty prop = getObsProperty("count");
11     prop.updateValue(prop.intValue()+1);
12     signal("tick");
13   }
14 }
```

No código acima, a propriedade observável "count" é definida (linha 6), com valor 0 associado. Nas linhas 10 e 11 essa propriedade é atualizada quando a ação *inc* é acionada pelo agente. Na linha 12 um *signal* é enviado a todos os agentes que observam a propriedade "count".

```
1  !observe.
2
3  +!observe : true
4    <- ?myTool(C);  // discover the tool
5       focus(C).
6
7  +count(V)
8    <- println("observed new value: ",V).
9
10 +tick [artifact_name(Id,"c0")]
11   <- println("perceived a tick").
12
13 +?myTool(CounterId): true
14   <- lookupArtifact("c0",CounterId).
15
16 -?myTool(CounterId): true
17   <- .wait(10);
18      ?myTool(CounterId).
```

O código acima apresenta o agente que observa o artefato com a propriedade observável "count". O agente busca o artefato e passa a observá-lo através da operação *focus* (linhas 4 e 5 e planos subsequentes).

Esse exemplo apresenta duas formas de interação com o artefato. Uma é através da propriedade observável definida, nesse caso "count" (linha 7) e a outra é através dos *signals*. *Signals* ativam crenças nos agentes, no código acima a crença *+tick*. Todas as vezes que o artefato enviar o *signal* o plano desencadeado pela crença referente ao *signal* enviado será ativada.

MOISE+

O modelo organizacional MOISE+ [Hübner et al. 2010, Hubner et al. 2007, Hübner 2003] foi desenvolvido para modelar a organização de SMA e consiste na especificação de três dimensões: a estrutural, onde definem-se papéis e ligações de heranças e grupos; a funcional, onde é estabelecido um conjunto de planos globais e

missões para que as metas sejam atingidas; e a deôntica, que é a dimensão responsável pela definição de qual papel tem obrigação ou permissão para realizar cada missão.

Assim, MOISE+ é um modelo de organização para sistemas multiagentes baseado em noções como papéis, grupos e missões. Isso permite que o sistema tenha sua organização explícita e que seja usada uma plataforma que faça os agentes cumprirem com suas obrigações da organização [Santos et al. 2014, Santos et al. 2013].

No exemplo abaixo são definidos três papéis, na organização "exemplo": *papel_A*, *papel_B* e *papel_C* . A organização "exemplo" possui um grupo chamado "grupo_exemplo". Para o *papel_A* é necessário o mínimo de 1 participante e um máximo de 10. Para o *papel_B* o mínimo de participantes é 1 e o máximo é 2. Para o *papel_C* é necessário apenas um agente. Essas condições devem ser satisfeitas para que o sistema passe a funcionar.

```
<?xml version="1.0" encoding="UTF-8"?>
<organisational-specification
    id="exemplo"
    os-version="0.8"

    xmlns='http://moise.sourceforge.net/os'
    xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
    xsi:schemaLocation='http://moise.sourceforge.net/os
                        http://moise.sourceforge.net/xml/os.xsd' >

<structural-specification>

<role-definitions>
 <role id="papel_A" />
 <role id="papel_B"/>
 <role id="papel_C"/>
</role-definitions>

<group-specification id="grupo_exemplo">
 <roles>
  <role id="papel_A"          min="1" max="10"/>
  <role id="papel_B"          min="1" max="2"/>
  <role id="papel_C"          min="1" max="1"/>
 </roles>

</group-specification>
</structural-specification>

</organisational-specification>
```

O agente abaixo cria o artefato (ação *makeArtifact*, linha 4) e adota o papel *papel_C* na linha 5, e comunica os demais agentes através de uma ação *.broadcast* (linha 6).

```
!start.

+!start: true
      <-    makeArtifact("grupo_exemplo","ora4mas.nopl.GroupBoard",["
          src/exemplo-os.xml", grupo_exemplo, false, true],GrArtId);
      adoptRole(papel_C)[artifact_id(GroupBoard)];
```

```
6       .broadcast(tell, org).
```

O método *init* do *makeArtifact* tem os seguintes parâmentros:

```
1  public void init(java.lang.String osFile,
2                   java.lang.String grType,
3                   boolean createMonitoring,
4                   boolean hasGUI)
5           throws ParseException,
6                  MoiseException,
7                  cartago.OperationException
```

Parâmetros

- **osFile** - O arquivo de especificação da organização (caminho e nome do arquivo)
- **grType** - O tipo do grupo
- **createMonitoring** - Se um esquema de monitoramento será criado e anexado
- **hasGUI** - Se uma GUI deve ser criada para o artefato

A ação *.broadcast* serve para comunicar aos demais agentes que o artefato "está pronto". Assim, quando ativada a crença *+org*, ele busca o artefato (linha 2), adota um papel (linha 3), e observa o artefato (*focus*, linha 4).

```
1  +org: true
2       <- lookupArtifact("hsj_group", GroupBoard);
3       adoptRole(papel_B)[artifact_id(GroupBoard)];
4       focus(GroupBoard).
```

As demais propriedades organizacionais devem ser explicitadas no XML corresponde da organização.

Os arquivos JAVA entre outros necessários para executar a integração Jason, CArtAgO[1] e MOISE+[2] estão todos presentes na pasta *exemplo_final*. Estes **devem** ser incluídos em qualquer projeto que integre as três ferramentas do JaCaMo seguindo este tutorial.

Códigos tutorial

Os códigos apresentados nesse tutorial estão disponíveis em *https://github.com/hdonancio/jacamo*

Referências

Bordini, R. H. and Hübner, J. F. (2014). Jacamo project. `http://jacamo.sourceforge.net/`.

Bordini, R. H., Hübner, J. F., and Wooldridge, M. (2007). *Programming Multi-Agent Systems in AgentSpeak using Jason*. Wiley, New Jersey.

Bratman, M. (1987). *Intention, plans, and practical reason*. Harvard University Press.

Hübner, J. F. (2003). *Um Modelo de Reorganização de Sistemas Multiagentes*. PhD thesis, Universidade de São Paulo, São Paulo.

[1] Bibliotecas *.jar*

[2] Pasta ora4mas e ora4masMoise

Hübner, J. F., Boissier, O., Kitio, R., and Ricci, A. (2010). Instrumenting multi-agent organisations with organisational artifacts and agents. *Autonomous Agents and Multi-Agent Systems*, 20(3):369–400.

Hubner, J. F., Sichman, J. S., and Boissier, O. (2007). Developing organised multiagent systems using the moise+ model: Programming issues at the system and agent levels. *Int. J. Agent-Oriented Softw. Eng.*, 1:370–395.

Rao, A. S. (1996). AgentSpeak(L): BDI agents speak out in a logical computable language. In *Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, LNCS. Springer, Berlin.

Ricci, A., Santi, A., and Piunti, M. (2014). CArtAgO (common atifact infrastructure for agents open environments). `http://apice.unibo.it/xwiki/bin/view/CARTAGO/`.

Ricci, A., Santi, A., Piunti, M., Viroli, M., Omicini, A., Guidi, M., and Minotti, M. (2010). CArtAgO (common atifact infrastructure for agents open environments). `http://cartago.sourceforge.net`.

Ricci, A., Viroli, M., and Omicini, A. (2007). The a&a programming model and technology for developing agent environments in mas. In *International Workshop on Programming Multi-Agent Systems*, pages 89–106. Springer.

Santos, F., Rodrigues, T., Donancio, H., Santos, I., Adamatti, D. F., Dimuro, G. P., Dimuro, G., and Jerez, E. D. M. (2014). Towards a multi-agent-based tool for the analysis of the social production and management processes in a urban ecosystem: an approach based on the integration of organizational, regulatory, communication and physical artifacts in the JaCaMo framework. In Adamatti, D., Dimuro, G. P., and Coelho, H., editors, *Interdisciplinary Applications of Agent-Based Social Simulation and Modeling*, pages 287–311. IGI Global.

Santos, F. C. P., Rodrigues, H. D. N., Rodrigues, T. F., Dimuro, G., Adamatti, D. F., Dimuro, G. P., and Jerez, E. M. (2013). Integrating cartago artifacts for the simulation of the social production and management of urban ecosystems: the case of san jerónimo vegetable garden of seville, spain.

Woolridge, M. and Wooldridge, M. J. (2001). *Introduction to Multiagent Systems*. John Wiley & Sons, Inc., New York, NY, USA.