

**UNIVERSIDADE FEDERAL DE SANTA CATARINA  
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA**

Jéssica Pauli de Castro Bonson

**APLICAÇÃO DE AGENTES & ARTEFATOS PARA  
ACESSAR A WEB SEMÂNTICA:  
DESENVOLVIMENTO DE UMA FERRAMENTA DE  
AUTORIA DE METADADOS DE OBJETOS DE  
APRENDIZAGEM**

Florianópolis

2013



Jéssica Pauli de Castro Bonson

**APLICAÇÃO DE AGENTES & ARTEFATOS PARA  
ACESSAR A WEB SEMÂNTICA: DESENVOLVIMENTO  
DE UMA FERRAMENTA DE AUTORIA DE  
METADADOS DE OBJETOS DE APRENDIZAGEM**

Trabalho de Conclusão de Curso sub-  
metido à Ciências da Computação para  
a obtenção do Grau de Bacharel.  
Orientador: Prof. Elder R. Santos

Florianópolis

2013



## LISTA DE FIGURAS

Figura 1	Modelo básico de um agente.....	16
Figura 2	Camadas de abstração de um SMA.....	18
Figura 3	O modelo A&A expressado em uma notação similar à UML.....	23
Figura 4	Uma representação metafórica de um SMA de acordo com o modelo A&A.....	24
Figura 5	Representação abstrata de um artefato.....	24
Figura 6	Exemplos de interações dos agentes com uma área de trabalho.....	27
Figura 7	Utilização de um artefato.....	27
Figura 8	Foco em um artefato.....	27
Figura 9	Representação de como a integração entre as plataformas de agentes e o Cartago funciona.....	34
Figura 10	Exemplo de ontologia OWL desenvolvida no Protégé... ..	42
Figura 11	Visão geral da ferramenta.....	53
Figura 12	Visão geral do sistema multi-agente.....	54
Figura 13	Screenshot da categoria General.....	55
Figura 14	Screenshots das categoria LifeCycle, Technical e Rights.....	55
Figura 15	Screenshot da aba de ajuda.....	55
Figura 16	Diagrama de fluxo da ferramenta de autoria.....	56
Figura 17	Saída da ferramenta com o campo Keywords preenchido com a palavra “bison” para o campo Title.....	69
Figura 18	Agente deliberativo Aspirador no Mundo do Vácuo. ...	90
Figura 19	Diagrama do processo de raciocínio em uma arquitetura BDI genérica.....	91
Figura 20	Os três tipos de fluxo de controle e de informação que uma arquitetura em camadas pode ter.....	93
Figura 21	Representação abstrata do Artefato Contador.....	107
Figura 22	Representação abstrata do Artefato Relógio.....	110
Figura 23	Representação abstrata do Artefato Sincronizador.....	112
Figura 24	Representação abstrata do Artefato TupleSpace.....	115
Figura 25	Representação abstrata do Artefato MarsEnv.....	127



## LISTA DE TABELAS

Tabela 1	Resultado da consulta SPARQL “Quais são os gases nobres?” .....	44
Tabela 2	Comparação resumida entre ferramentas de autoria. . . .	48
Tabela 3	Resultados por abordagem para a palavra-chave “bison”	71
Tabela 4	Resultados por abordagem para a palavra-chave “tuberculosis” .....	71
Tabela 5	Qualidade dos resultados para algumas palavras-chaves	72





## LISTA DE ABREVIATURAS E SIGLAS

IA	Inteligência Artificial.....	11
SGBD	Sistema de Gerenciamento de Banco de Dados .....	12
SMA	Sistema Multi-Agente.....	17
A&A	Agentes & Artefatos .....	22
AOSE	Agent-Oriented Software Engineering .....	22
EOP	Environment Oriented Programming.....	23
UML	Unified Modelling Language.....	24
GUI	Graphical User Interface .....	32
MABS	Multi-Agent Based Simulation .....	33
SOS	Self-Organizing Systems.....	34
Cartago	Common Artifact infrastructure for Agent Open environ- ment .....	34
RBAC	Role-Based Access Control.....	34
RDF	Resource Description Framework.....	42
IRI	Internationalized Resource Identifier .....	42
OWL	Web Ontology Language .....	43
URI	Universal Resource Identifiers .....	43
SPARQL	SPARQL Protocol and RDF Query Language .....	44
SAIL	Storage and Inference Layer .....	47
OA	Objeto de Aprendizagem.....	49
DC	Dublin Core .....	49
LOM	Learning Object Metadata.....	49
Scorm	Sharable Content Object Reference Model .....	49
LMS	Learning Management System .....	51
CARLOS	Collaborative Authoring of Reusable Learning ObjectS.	51
OBAA	Projeto Objetos de Aprendizagem baseados em Agentes	52
NLP	Processamento de Linguagem Natural.....	59
BDI	Belief-Desire-Intention .....	94
PRS	Procedural Reasoning System.....	99
dMARS	Distributed Multi-Agent Reasoning System .....	99
IRMA	Intelligent Resource-bounded Machine Architecture ....	100
JIAC	Java-based Intelligent Agent Componentware.....	105

JADE	Java Agent DEvelopment Framework .....	105
ACL	Agent Communication Language .....	105

## SUMÁRIO

<b>1 INTRODUÇÃO</b>	11
1.1 OBJETIVOS	12
1.2 MOTIVAÇÃO	13
<b>2 FUNDAMENTAÇÃO TEÓRICA</b>	15
2.1 AGENTES E SISTEMAS MULTI-AGENTE	15
2.1.1 Agentes	15
2.1.2 Sistemas Multi-Agente	17
2.1.3 Jason	19
2.2 AGENTES & ARTEFATOS	22
2.2.1 Introdução	22
2.2.2 Artefatos	25
2.2.3 Funcionalidades dos Artefatos	30
2.2.4 Características do modelo de Agentes & Artefatos	32
2.2.5 Framework de A&A Cartago	34
2.3 ONTOLOGIAS E REPOSITÓRIOS SEMÂNTICOS	40
2.3.1 Web Semântica	40
2.3.2 Ontologias	41
2.3.3 RDF e OWL / OWL 2	42
2.3.4 Repositórios Semânticos	46
2.3.5 DBPedia	48
2.4 FERRAMENTAS DE AUTORIA DE OBJETOS DE APREN- DIZAGEM	48
2.4.1 Introdução	48
2.4.2 Exemplos de Ferramentas de Autoria	50
2.4.3 Trabalho Relacionado: Ferramenta CARLOS	51
2.4.4 Requisitos da Ferramenta de Autoria deste trabalho	52
2.5 TRABALHOS RELACIONADOS COM ACESSO DE AGEN- TES A WEB SEMÂNTICA	52
<b>3 DESENVOLVIMENTO</b>	55
3.1 VISÃO GERAL	55
3.2 FERRAMENTA DE AUTORIA DE METADADOS	56
3.3 MODELO DE SUGESTÕES	60
3.3.1 Abordagens de Busca Semântica	60
3.3.2 SMA	61
3.3.3 Prefixos	62
3.3.4 Filtros	63
3.3.5 Abordagem por Indivíduo	63

<b>3.3.6 Abordagem por Classes</b> .....	67
<b>3.3.7 Abordagem por Categorias</b> .....	69
<b>3.3.8 Processamento dos resultados</b> .....	70
<b>3.4 ANÁLISE DOS RESULTADOS DAS ABORDAGENS</b> .....	73
<b>4 CONCLUSÕES</b> .....	77
<b>5 AGRADECIMENTOS</b> .....	81
<b>REFERÊNCIAS</b> .....	83
<b>APÊNDICE A – Background de Sistemas Multi-Agentes</b>	93
<b>APÊNDICE B – Informações adicionais sobre Agentes</b>	
<b>&amp; Artefatos</b> .....	109
<b>APÊNDICE C – Informações adicionais sobre o Desen-</b>	
<b>volvimento</b> .....	141

## 1 INTRODUÇÃO

Este trabalho visa analisar e aplicar a tecnologia de Agentes & Artefatos (A&A) para o desenvolvimento de uma ferramenta de autoria de metadados utilizando um repositório semântico.

A&A baseia-se no conceito de agentes e sistemas multi-agente, uma sub-área da Inteligência Artificial (IA). Segundo (RUSSEL; NORVIG, 2003), a IA pode ser brevemente definida como um campo que busca compreender e construir entidades inteligentes. O conceito de agentes em IA não possui uma única definição, inicialmente esperava-se que os agentes conseguissem demonstrar uma inteligência humana completa, mas atualmente se considera mais factível que os agentes demonstrem apenas alguns aspectos de um comportamento inteligente (WOOLDRIDGE; JENNINGS, 1995b). Uma definição possível para agentes é que se trata de um sistema computacional situado em algum ambiente e capaz de realizar ações autônomas nesse ambiente visando atingir os seus objetivos de projeto (TWEEDALEA et al., 2006).

Sistemas Multi-Agente (SMA) consistem de um grupo fracamente acoplado de agentes autônomos que interagem entre si buscando resolver um problema (JENNINGS; SYCARA; WOOLDRIDGE, 1996). Algumas das características de um SMA é que os agentes não tem conhecimento ou capacidade para resolverem o problema de forma individual, os dados são descentralizados, não há um controle central e a comunicação é assíncrona (JENNINGS; SYCARA; WOOLDRIDGE, 1996).

O modelo de A&A surgiu da necessidade (RICCI; VIROLI; OMICINI, 2010, 2008c, 2008a; WEYNS; OMICINI; ODELL, 2006) de desenvolver o ambiente como uma entidade de primeira-classe, e assim permitir que ele seja uma parte do sistema que pode ser modelada e programada para encapsular serviços e funcionalidades que serão utilizadas pelos agentes em tempo de execução. Em A&A são utilizados artefatos para se modelar o ambiente, (RICCI; PIUNTI; VIROLI, 2010) apresenta dois pontos de vista diferentes sobre esta entidade: Do ponto de vista dos agentes artefatos são entidades que modelam ferramentas e recursos, e que podem ser dinamicamente instanciadas, observadas, compartilhadas, agrupadas e utilizadas para auxiliar as atividades individuais ou coletivas dos agentes. Do ponto de vista dos programadores do SMA, artefatos são abstrações para modelar e programar ambientes funcionais que os agentes podem utilizar, incluindo funcionalidades que afetam a interação entre os agentes e que permitem que eles interajam com o ambiente externo.

Outros dois conceitos revelantes para este trabalho são o de ferramenta de autoria e de repositório semântico. Mais especificamente, é desenvolvida uma ferramenta de autoria de metadados de objetos de aprendizagem. (DUVAL et al., 2002) define metadados como dados estruturados sobre outros dados, normalmente legíveis por máquinas, que fornecem informações adicionais e contexto para estes dados. De acordo com a IEEE (IEEE, 2002) um objeto de aprendizagem (OA) é qualquer entidade que pode ser utilizada, reutilizada ou referenciada durante o aprendizado apoiado por computador, como um texto, um vídeo ou um aplicativo. Segundo a W3C (RICHARDS et al., 2012) uma ferramenta de autoria é qualquer aplicação que pode ser utilizada para criar ou modificar conteúdo Web que será usado por outros autores ou usuários finais. Para (XAVIER; GLUZ, 2009) as ferramentas de autoria de objetos de aprendizagem devem possuir ambas ou uma das seguintes características: Oferecer um ambiente para a autoria de conteúdo digital e permitir a geração de objetos em conformidade com padrões de metadados. A ferramenta de autoria que será desenvolvida neste trabalho possui apenas a segunda característica, sendo uma ferramenta de autoria de metadados de aprendizagem. Foi decidido desenvolver a ferramenta utilizando o modelo de A&A por ela se beneficiar do potencial de reutilização e serviços fornecidos pelo modelo, como o acesso a Web Semântica.

Em relação a repositórios semânticos, (OPENRDF, 2012b) os definem como ferramentas similares a um Sistema de Gerenciamento de Banco de Dados (SGBD), que podem ser usadas para armazenar, administrar e realizar consultas sobre dados estruturados de acordo com o padrão RDF. (ONTOTEXT, 2012) cita que as principais diferenças são que estes repositórios utilizam ontologias como esquemas semânticos, permitindo a realização de inferências sobre os dados, e que trabalham com modelos de dados genéricos e flexíveis (grafos), possibilitando que facilmente interpretem e adotem novas ontologias e esquemas de metadados. Por conta destas diferenças, estes repositórios integram facilmente dados diversos, e possuem uma maior expressividade e capacidade de representação de conhecimento.

## 1.1 OBJETIVOS

**Objetivo Geral :** Analisar e aplicar a abordagem de Agentes & Artefatos para o desenvolvimento de um sistema de autoria de metadados de objetos de aprendizagem utilizando um repositório semântico.

### Objetivos Específicos:

- Especificar os requisitos da ferramenta de autoria de metadados.
- Compreender a tecnologia de A&A.
- Compreender o armazenamento de dados em repositórios semânticos.
- Modelar o repositório semântico com A&A para poder ser utilizado pelos agentes.
- Modelar ou utilizar uma ontologia que será usada pelos agentes para acessar os objetos de aprendizagem do repositório semântico.
- Implementar o sistema de autoria de metadados com A&A.
- Contextualizar a contribuição do sistema desenvolvido nas áreas de SMA e e-learning.

## 1.2 MOTIVAÇÃO

A principal motivação deste trabalho é realizar um estudo de caso para o modelo de A&A. Por ser um modelo recente poucos projetos foram desenvolvidos utilizando-o e pretende-se informar quais foram os resultados positivos e negativos de utilizá-lo para modelar o sistema, além de possíveis sugestões em relação ao framework Cartago, que implementa o modelo de A&A.

Outra motivação deste trabalho está relacionada à ferramenta de autoria de metadados que será desenvolvida. Até onde se pesquisou, só foi encontrada uma única ferramenta de autoria de objetos de aprendizagem (PADRON, 2003) que utiliza sistemas multi-agente, sendo que ela é focada no desenvolvimento de objetos de aprendizagem, não no preenchimento dos metadados segundo algum padrão, como a que será desenvolvida neste trabalho.





## 2 FUNDAMENTAÇÃO TEÓRICA

### 2.1 AGENTES E SISTEMAS MULTI-AGENTE

#### 2.1.1 Agentes

Os agentes surgiram da necessidade para um número cada vez maior de aplicações de sistemas que consigam decidir por eles mesmos o que eles precisam fazer para satisfazer seus objetivos. Essa necessidade surge para ambientes que mudam rapidamente, são imprevisíveis ou abertos. Exemplos de aplicações deste tipo citadas por (WOOLDRIDGE, 2002) são um robô explorando um ambiente desconhecido em outro planeta e um agente que busca informações na Internet de acordo com uma consulta.

Não há uma definição universal do termo agente, isso ainda é tema de debates e um dos poucos consensos é de que a noção de autonomia é essencial. Autonomia é descrita por (WOOLDRIDGE, 2002) como a habilidade dos agentes agirem sem a intervenção de humanos ou outros sistemas, tendo controle sobre o seu estado interno e seu comportamento. Abaixo seguem definições de agentes segundo diferentes autores.

(WOOLDRIDGE, 2002) descreve que um agente é um sistema computacional que está situado em algum ambiente e é capaz de realizar ações autônomas nesse ambiente visando cumprir seus objetivos de modelagem. Adicionalmente, o autor considera que um agente que seja também inteligente é capaz de realizar ações reativas, um comportamento pró-ativo e habilidades sociais, onde pró-atividade é capacidade do agente ter uma atitude direcionada a cumprir seus objetivos. (RUSSEL; NORVIG, 2003) possui uma definição mais genérica, na qual um agente é qualquer coisa que pode perceber um ambiente através de sensores e pode atuar sobre ele com atuadores.

Para (FRANKLIN; GRAESSER, 1996) um agente autônomo é um sistema situado dentro de uma parte de um ambiente, que percebe este ambiente e atua sobre ele ao longo do tempo de acordo com sua agenda e de forma a afetar o que o agente irá perceber no futuro. (WOOLDRIDGE; JENNINGS, 1995a) distingue o termo agente em dois usos comuns, um fraco e um forte. Na noção fraca um agente é um hardware ou um sistema baseado em software que possui as seguintes propriedades: autonomia, os agentes operam sem uma intervenção direta humana ou de

outros sistemas; habilidade social, o agente interage com outros agentes ou humanos através de alguma linguagem de comunicação para agentes; reatividade, os agentes percebem o ambiente e reagem rapidamente às mudanças que ocorrem nele; e pró-atividade, além de reagirem ao ambiente, os agentes também possuem um comportamento direcionado a objetivos.

A noção forte de agentes descrita por (WOOLDRIDGE; JENNINGS, 1995a) sugere que, além de ter as propriedades estabelecidas pela noção fraca, um agente é modelado e implementado utilizando conceitos humanos, como noções mentais (conhecimento, crença, intenção, obrigação), e as vezes também noções emocionais. Outra característica citada é a racionalidade, segundo este autor um agente é racional se ele agir de forma a atingir seus objetivos, e não agir de alguma maneira que possa impedi-lo de atingi-los.

Na Figura 1 é mostrado um modelo básico do funcionamento de um agente. Um agente possui um conjunto de sensores que permitem entender informações sensoriais do ambiente na forma de percepções e um repertório de ações que podem ser executadas para modificar ou inspecionar o ambiente através de atuadores. Uma ação possui pré-condições e pós-condições, ou seja, condições que ele assume serem verdadeiras e efeitos no ambiente após a execução da ação, sendo que estas ações podem falhar. Dentro do agente ocorre o processo de decisão sobre qual ação efetuar, o modo como este processo é realizado varia de acordo com a arquitetura de agente utilizada.

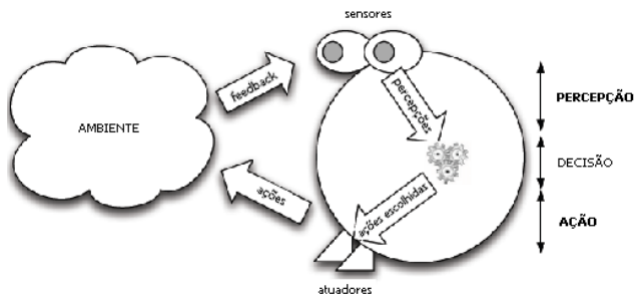


Figura 1 – Modelo básico de um agente (BOISSIER et al., 2011)

Um exemplo de um agente pela noção fraca seria um termostato, com ações para alterar a temperatura do ambiente e sensores para captar essa temperatura, porém não possuindo habilidade social. Outro exemplo seria um robô que explora uma área desconhecida a procura de

um objeto, com atuadores para se locomover, pegar e largar o objeto, e sensores para detectar o objeto procurado ou obstáculos no caminho, este agente pode ser forte ou fraco, dependendo de como for implementado. E por fim um agente jogador de xadrez, com atuadores para movimentar suas peças no jogo e sensores para perceber o estado do jogo, que seria um exemplo da noção forte, pois armazenaria as regras do jogo como um conhecimento.

Neste trabalho é utilizada a teoria de agentes BDI (Belief-Desire-Intention) e a linguagem de agentes Jason. A linguagem Jason foi escolhida por fornecer um suporte direto ao modelo de A&A. Mais informações sobre teorias, arquiteturas e linguagens de agentes podem ser consultadas no Apêndice A.

### 2.1.2 Sistemas Multi-Agente

(JENNINGS; SYCARA; WOOLDRIDGE, 1996) define Sistema Multi-Agente (SMA) como um sistema composto por uma coleção de agentes autônomos que buscam resolver um problema. Os autores também citam algumas características de SMAs:

- cada agente possui informações e capacidades limitadas para resolver o problema, possui um ponto de vista limitado;
- não há um sistema de controle global;
- os dados são descentralizados;
- a computação é assíncrona.

De acordo com (JENNINGS; SYCARA; WOOLDRIDGE, 1996) SMAs são adequados para representar problemas que possuem múltiplas formas de serem resolvidos, múltiplas perspectivas e/ou múltiplas entidades. Isso porque SMAs podem resolver problemas de forma distribuída, concorrente e utilizando diversos tipos de interação entre os agentes. Estas interações incluem: cooperação (agentes trabalhando juntos buscando cumprir um mesmo objetivo), coordenação (organizar a resolução do problema de forma a explorar atividades benéficas, e evitar atividades maléficas para essa resolução) e negociação (chegar a um acordo aceitável para todas as partes envolvidas). A flexibilidade e a natureza de alto nível destas interações provêm o poder fundamental deste paradigma. Outras vantagens relevantes de SMAs incluem:

robustez, eficiência, capacidade para interoperações entre sistemas legados já existentes, e a habilidade para resolver problemas em que os dados, o controle e o conhecimento estejam distribuídos.

Um exemplo de SMA citado por (JENNINGS; SYCARA; WOOLDRIDGE, 1996) é um sistema de controle de tráfego aéreo. Cada agente controla um avião e tem como objetivo chegar ao seu destino com baixo consumo de combustível mantendo uma distância segura dos outros aviões. Agentes envolvidos em uma situação de conflito negociam com um dos agentes conflitantes para tentar resolvê-lo. Os agentes também planejam sua rota, de forma a melhor atingir seu objetivo.

Os autores também citam algumas áreas em que SMAs já foram utilizados com sucesso: manufatura, controle de processo, sistema de telecomunicação, controle de tráfego aéreo, administração de tráfego e de transporte, filtragem e coleta de informações, comércio eletrônico, administração de processos de negócio, entretenimento e cuidado médico.

A Figura 2 apresenta as camadas de abstração de um ambiente sob um ponto de vista no qual um sistema multi-agente é dividido nos seguintes níveis: organizacional/social, agentes e ambiente.

O nível organizacional e social é composto de estruturas e regras, é nele que as interações sociais acontecem, e elas seguem as estruturas e regras estabelecidas neste nível. Um agente pode ter diferentes papéis, as regras do nível organizacional definem restrições em cada papel, quais normas ele deve respeitar e quais responsabilidades ou missões ele possui. As estruturas definem a topologia das interações e das relações sobre o controle da atividade.

O nível dos agentes é composto por agentes individuais, autônomos e situados. Estes agentes percebem e agem sobre o nível de ambiente, que é definido como um conjunto de recursos e serviços que os agentes podem acessar e controlar. O ambiente pode ser endógeno, dentro do próprio SMA, ou exógeno, externo ao SMA.

(WOOLDRIDGE; JENNINGS, 1995a) descreve um exemplo interessante para a utilidade de um SMA: O principal sistema de controle aéreo do país Rutitania para de funcionar de repente, devido as terríveis condições climáticas. Mas felizmente os sistemas de controle aéreo computadorizados dos países vizinhos negociam entre si para rastrear e tratar os vôos afetados, e uma situação possivelmente desastrosa passa sem maiores incidentes.

No Apêndice A.4 são descritos os frameworks JIAC e Jade, escolhidos devido a sua ampla utilização (KÜSTER et al., 2012; HOCH et al.,

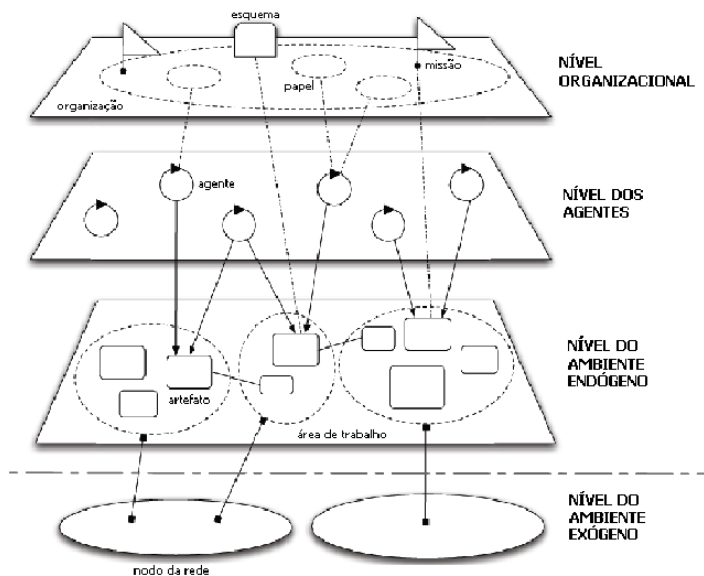


Figura 2 – Camadas de abstração de um SMA, retirado e traduzido de (BOISSIER et al., 2011).

2011; GRUNEWALD et al., 2011; SPANOUDAKIS; MORAITIS, 2007; BELLIFEMINE; CAIRE; GREENWOOD, 2007).

### 2.1.3 Jason

Nesta seção serão dadas mais informações sobre a linguagem de programação Jason (BORDINI; HUBNER, 2007). Como o Jason é baseado em uma versão expandida do AgentSpeak, primeiro será descrito como os agentes desta linguagem são programados.

Um agente AgentSpeak é criado através da especificação de um conjunto de crenças e um conjunto de planos. As crenças são representadas por predicados lógicos de primeira-ordem. Existem dois tipos de objetivos nesta linguagem: objetivos de realização, prefixados com !, e objetivos de teste, prefixados com ?. Objetivos de realização determinam que o agente quer atingir um estado de mundo em que este objetivo esteja concluído, e para isso iniciam a execução de planos. Objetivos de teste retornam a unificação do predicado associado com uma

das crenças dos agentes, e falham caso não consigam unificar. Eventos de gatilho definem quais eventos podem iniciar a execução de um plano, um evento pode ser interno (cumprir um objetivo) ou externo (atualização de crenças ou percepção do ambiente). Existem dois tipos de eventos de gatilho: adição, prefixados com +, e deleção atitudes mentais, prefixado com -. Por fim, um plano é ativado por um evento de gatilho em um determinado contexto (predicado lógico), e é composto por uma sequência de ações básicas ou objetivos.

Abaixo é mostrado um exemplo de trecho de código de um agente AgentSpeak. O primeiro plano (+concerto) é ativado quando o agente percebe que haverá um concerto do artista A no local V, e que ele também gosta deste artista A. Ao ser ativado, o plano cria o objetivo de reservar ingressos para esse concerto, ativando o próximo plano. O segundo plano (+!reservar\_ingressos) é ativado caso ele acredite que o telefone não esteja ocupado, e realiza uma sequência de ações que inclui ligar para o local V e criar o objetivo de escolher os assentos no local.

---

```

1 +concerto(A,V) : gosta(A)
    <- !reservar_ingressos(A,V).
3
4 +!reservar_ingressos(A,V) : not ocupado(telefone)
5    <- ligar(V);
6        ...;
7        !escolher_assentos(A,V).

```

---

As principais diferenças entre AgentSpeak e Jason são:

- negação forte;
- manipulação de falhas de planos;
- comunicação entre agentes baseada na teoria de atos de fala, junto com anotações nas crenças sobre a origem da informação;
- anotações nos rótulos dos planos, permitindo o uso de funções de seleção mais elaboradas;
- suporte para o desenvolvimento de ambientes;
- suporte para sistemas multi-agentes distribuídos;
- funções de seleção personalizáveis, funções de confiança e arquitetura personalizável.
- uma biblioteca de ações internas essenciais;

- extensibilidade através de ações internas definidas pelo(a) usuário(a);
- um agente pode possuir conjunto de de objetivos iniciais;
- base de crenças pode possuir regras semelhantes ao Prolog;
- operador  $+-$ , que adiciona uma crença depois de remover a primeira ocorrência desta crença.

Esta seção dará enfoque ao desenvolvimento de ambientes em Jason. Os ambientes nesta linguagem são classes Java, e a declaração de qual ambiente um SMA irá usar ocorre no arquivo `.mas2j`, nesse arquivo também podem ser incluídos parâmetros para o ambiente e a identificação do nodo da rede em que este ambiente está localizado. Um modelo de ambiente em Java é mostrado abaixo, as duas estruturas principais são o construtor e o método `executeAction`.

O construtor estabelece as percepções iniciais do SMA, as percepções podem ser globais ou individuais. Os métodos de administração de percepções são: `addPercept` (adiciona percepção), `removePercept` (retira percepção) e `clearPercept` (deleta todas as percepções).

O método `executeAction` contém o código principal do ambiente. Dada uma ação e seus parâmetros e o agente que a executa este método realiza os procedimentos que forem necessários neste modelo de ambiente particular, normalmente isso inclui atualizar as percepções do ambiente. O retorno da execução da ação deve retornar um valor booleano, que indica se a ação falhou ou foi concluída com sucesso.

---

```

1  import java.util.*;
   import jason.asSyntax.*;
3  import jason.environment.*;

5  public class <AmbienteNome> extends Environment {

7      // membros de classe quaisquer que possam ser
       necessários

9      public <AmbienteNome> () {
          // adiciona as percepções iniciais ao ambiente
11         addPercept(Literal.parseLiteral("p(a)"));
           // ....
13     }

15     public boolean executeAction(String ag, Structure act) {
           // ...
17     {

19         // quaisquer outros métodos que possam ser necessários
   }

```

---

## 2.2 AGENTES & ARTEFATOS

### 2.2.1 Introdução

Agentes & Artefatos (A&A) é um modelo para desenvolvimento de sistemas multi-agente no qual o ambiente é programado como uma entidade de primeira-classe, ou seja, o ambiente é uma parte do sistema que pode ser modelada e programada, junto com os agentes, para encapsular funcionalidades que serão utilizadas pelos agentes em tempo de execução (RICCI; PIUNTI; VIROLI, 2010). Este modelo possui fundamentação em diversas áreas como a computação, a psicologia, a sociologia e as ciências cognitivas, sendo a Teoria da Atividade (Activity Theory) a principal base teórica (OMICINI; RICCI; VIROLI, 2008a).

A Teoria da Atividade descreve como as atividades humanas coletivas se desenvolvem e se transformam, tendo o conceito de atividade como a unidade fundamental. Segundo esta teoria, qualquer atividade realizada dentro de uma organização só pode ser compreendida completamente se forem analisadas as ferramentas ou artefatos que possibilitaram as ações e mediaram as interações entre os indivíduos entre si e entre o ambiente. Os artefatos, físicos ou psicológicos, seriam a parte do ambiente que pode ser planejada e controlada para auxiliar as atividades dos indivíduos, alguns exemplos seriam relógios, agendas, linguagens e processos operacionais (OMICINI; RICCI; VIROLI, 2008a; RICCI; OMICINI; DENTI, 2003).

De acordo com (RICCI; PIUNTI; VIROLI, 2010) atualmente há duas perspectivas principais que podem ser adotadas para se definir o ambiente em um SMA. A primeira é a da IA clássica, descrita em (RUSSEL; NORVIG, 2003), na qual a noção de ambiente é utilizada para identificar o mundo externo, onde os agentes atuam e recebem percepções. A outra perspectiva se baseia em trabalhos recentes (WEYNS; OMICINI; ODELL, 2006; WEYNS; PARUNAK, 2007a) em Engenharia de Software Orientada para Agentes (Agent-Oriented Software Engineering, AOSE) que aponta que o ambiente pode ser modelado como uma entidade de primeira classe e que, como é o facilitador e mediador das interações entre os agentes, é também um lugar adequado para se encapsular serviços e funcionalidades que auxiliem as atividades dos agentes. Segundo esta



última idéia, o ambiente pode ser construído de forma a melhorar todo o desenvolvimento do sistema, permitindo um maior controle sobre a topologia e funcionalidades do SMA. Conforme (WEYNS; PARUNAK, 2007b) a definição de ambiente nessa nova perspectiva é a de uma abstração de primeira-classe que provê condições para que os agentes existam e que media tanto a interação entre agentes como o acesso a recursos.

(BOISSIER et al., 2011) relata outros motivos para a Programação Orientada a Ambientes (Environment Oriented Programming, EOP) e os categoriza nos níveis básico e avançado. No nível básico, esta programação serve para criar ambientes de teste para ambientes reais ou externos e para facilitar a interface/interação com softwares de ambiente já existentes. No nível avançado ela introduz a noção de externalização, que permite encapsular e modularizar de forma uniforme as funcionalidades do SMA fora do agente, geralmente relacionadas com interação, coordenação, organização ou segurança.

(OMICINI; RICCI; VIROLI, 2008b; RICCI; PIUNTI; VIROLI, 2010)descreve que em A&A os SMAs são construídos baseando-se em duas abstrações principais: agentes e artefatos. Os agentes são as entidades autônomas e pró-ativas que encapsulam o controle e a lógica de suas ações e buscam cumprir seus objetivos, sendo a abstração básica para modelar e programar as partes autônomas do SMA. Enquanto artefatos representam as entidades passivas e reativas, que podem ser usadas para modelar e programar funcionalidades que podem ser acessadas, usadas e possivelmente adaptadas para auxiliar as atividades dos agentes.

Outras abstrações presentes neste modelo são as áreas de trabalho (workspaces) e o ambiente (environment). Uma área de trabalho representa um lugar no qual ocorre uma ou múltiplas atividades envolvendo um conjunto de agentes e artefatos. Elas servem para definir a topologia do ambiente e podem estar espalhadas entre vários nodos da rede. Uma área de trabalho é um conjunto dinâmico de artefatos, sendo que um artefato só pode estar em uma área por vez, mas um agente pode estar em várias áreas de trabalho ao mesmo tempo. Quando um agente é criado ele automaticamente passa a participar de uma área de trabalho padrão. Além disso, elas permitem criar políticas para o acesso e execução de ações dos agentes. Um ambiente é um conjunto dinâmico de agentes e artefatos, e pode conter diversas áreas de trabalho.

A Figura 3 apresenta uma relação UML entre as entidades de um SMA no modelo A&A, sendo que os artefatos, os agentes, a área de trabalho e o ambiente já foram citados e as entidades manual, operação,

eventos observáveis e propriedades observáveis, todas relacionadas com os artefatos, serão explicadas em mais detalhes nas seções a seguir. Na Figura 3 também são mostradas algumas das ações que um agente pode efetuar sobre um artefato e sobre uma área de trabalho, que também serão explicadas em mais detalhes adiante.

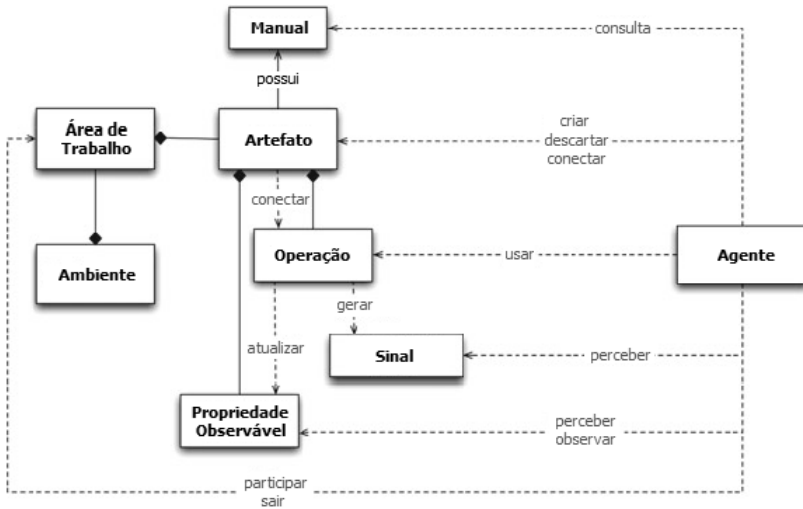


Figura 3 – O modelo A&A expressado em uma notação similar à UML (Unified Modelling Language), retirado e traduzido de (RICCI; PIUNTI; VIROLI, 2010).

A Figura 4 mostra uma representação de um SMA no modelo A&A. Neste exemplo o ambiente possui apenas uma área de trabalho, a padaria, que os agentes podem acessar dinamicamente através da porta. Nela situam-se diversos agentes, os padeiros, e diversos artefatos que eles utilizam. O artefato “canal de comunicação” media a interação entre os agentes e o ambiente externo, os artefatos “quadro-negro”, “relógio” e “agenda de tarefas” auxiliam os agentes na execução do trabalho, enquanto o artefato “bolo” é o próprio alvo do trabalho dos agentes. O artefato “arquivo” guarda informações que podem ser úteis aos agentes, e em um SMA real provavelmente seria a abstração de um repositório ou de um banco de dados.

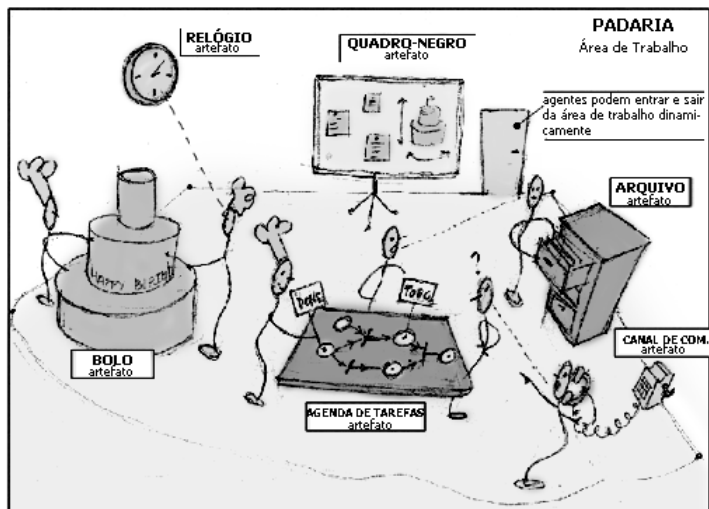


Figura 4 – Uma representação metafórica de um SMA de acordo com o modelo A&A, retirado e adaptado de (RICCI et al., 2009).

## 2.2.2 Artefatos

Artefatos são entidades computacionais passivas do SMA. Do ponto de vista da modelagem e programação de SMA os artefatos são o módulo básico para se estruturar e organizar o ambiente, provendo um modelo computacional de propósito geral para modelar as funcionalidades disponíveis para os agentes. Do ponto de vista dos agentes os artefatos estruturam, de uma maneira funcional, o mundo no qual eles estão situados e que eles podem criar, utilizar, compartilhar e perceber em tempo de execução (RICCI; PIUNTI; VIROLI, 2010).

Na Figura 5 é mostrada uma representação de um artefato. Para que um artefato permita que suas funcionalidades sejam utilizadas pelos agentes ele provê um conjunto de operações (operations), agrupadas em uma interface de utilização (usage interface), e um conjunto de propriedades observáveis (observable properties).

Uma operação representa um processo computacional executado dentro do artefato, que pode ser ativado por um agente ou por outro artefato. Quando um agente executa uma operação de um artefato a sua atividade ou plano atual é suspenso até ocorrer um evento indicando

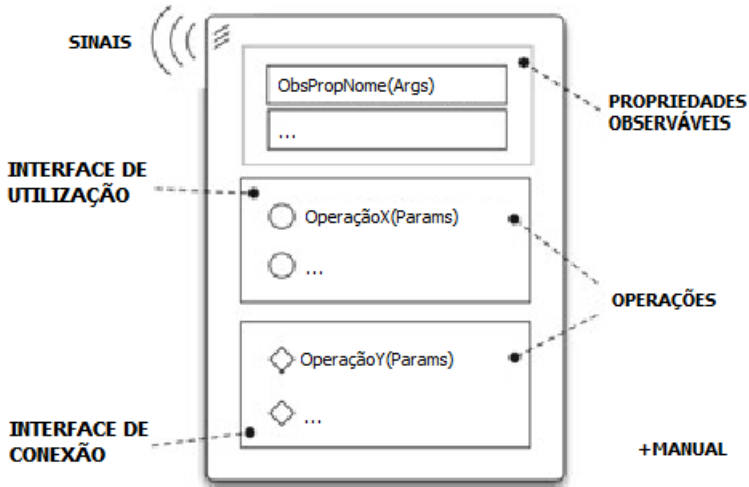


Figura 5 – Representação abstrata de um artefato, retirado e traduzido de (RICCI; PIUNTI; VIROLI, 2010).

que a operação foi concluída. É importante ressaltar que o agente não é bloqueado, apenas uma atividade é suspensa, enquanto isso o ciclo do agente continua executando novas atividades ou planos. Do ponto de vista dos agentes, as operações dos artefatos representam ações externas providas pelo ambiente. Isso implica que o repertório de ações de um agente é dinâmico, porque o conjunto de artefatos do ambiente muda dinamicamente, e além disso o próprio conjunto de operações na interface de utilização de um artefato também é dinâmico, pois pode ser alterado junto com mudanças no estado do artefato (RICCI; PIUNTI; VIROLI, 2010).

Outras características relevantes das operações são que o processo computacional pode ser dividido em várias etapas atômicas. Cada etapa pode possuir uma guarda (guard), que faz com que aquela etapa seja ativada somente quando sua guarda, uma relação booleana, for cumprida. Somente uma etapa é efetuada por vez dentro do artefato, para prevenir inconsistências. Se um agente tentar efetuar uma operação e houver uma etapa de outra operação sendo processada, a operação será efetuada assim que a etapa em execução for concluída (RICCI; PIUNTI; VIROLI, 2010).

Propriedades observáveis representam variáveis de estado do ar-

tefato, que podem ser percebidas pelos agentes e podem se alterar dinamicamente, como resultado de alguma operação. Outro resultado que pode ser gerado pelos artefatos são os sinais (signals) que servem para representar quais eventos não-persistentes ocorreram dentro do artefato. Um artefato também pode possuir variáveis de estado ocultas, usadas pelas suas próprias operações. Em arquiteturas BDI percepções relacionadas aos valores de propriedades observáveis de um artefato que um agente esteja focando podem ser modeladas diretamente dentro do agente como crenças.

Outra interface que está representada na Figura 5 é a interface de conexão (link interface), ela possui operações que podem ser acionadas por outros artefatos, contanto que tenham sido previamente conectados entre si por um agente. Assim é possível criar artefatos complexos através de uma composição de artefatos mais simples, essa composição pode ser feita inclusive entre artefatos de diferentes áreas de trabalho, possibilitando que os artefatos compostos sejam distribuídos pela rede. Essa interface é análoga ao que existe nos artefatos usados por humanos, por exemplo conectando um computador a uma impressora.

Além das características citadas, um artefato também possui um manual com informações sobre quais são e como utilizar suas funcionalidades. Este documento deve ser legível para os agentes. A utilidade dos manuais é facilitar a existência de sistemas abertos e do uso cognitivo de artefatos, no qual os agentes dinamicamente decidem quais artefatos usar de acordo com seus objetivos, e descobrem como utilizá-los. Este é um trabalho em andamento e os pesquisadores da área ainda estão definindo como o manual será feito, uma das opções citada por (RICCI; PIUNTI; VIROLI, 2010; RICCI et al., 2009) é utilizar a Web Semântica como referência para criar linguagens e ontologias que possibilitem uma semântica compartilhada da descrição de um artefato.

Por fim, um agente possui um conjunto de ações que pode efetuar sobre um artefato, (RICCI; PIUNTI; VIROLI, 2010) as categoriza em três grupos: (i) ações para criar/encontrar/descartar artefatos; (ii) ações para usar artefatos, executar operações e observar propriedades e sinais; (iii) ações para conectar/desconectar artefatos. Também há ações para entrar e sair de uma área de trabalho. As Figuras 6, 7 e 8, retiradas e traduzidas de (RICCI et al., 2009), ilustram algumas das ações citadas em mais detalhes, no contexto da plataforma Cartago.

(RICCI; PIUNTI; VIROLI, 2010) descreve que para auxiliar as ações dos agentes no ambiente cada área de trabalho possui um conjunto de artefatos pré-definidos que provêm funcionalidades essenciais. Existem quatro artefatos padrão:

- workspace (área de trabalho): Provê funcionalidades para criar, descartar, procurar, conectar e focar (observar) artefatos da área de trabalho. Também provê operações para definir papéis e políticas de acesso e de execução de operações dos artefatos.
- node (nodo): Provê funcionalidades para criar novas áreas de trabalho e entrar em áreas de trabalho locais ou remotas.
- blackboard (quadro negro): Provê um espaço de tuplas que os agentes podem utilizar para se comunicar e coordenar. Um exemplo utilizando este artefato para implementar o Jantar dos Filósofos é mostrado no final do capítulo.
- console: Provê funcionalidades para imprimir mensagens na saída padrão.

A Figura 6 apresenta as dinâmicas de uma área de trabalho. Um agente passa a participar da área de trabalho “a-areadetrabalho”, e então usa o artefato padrão workspace para criar o artefato meuArt. Após isso o artefato está pronto para ser utilizado tanto pelo agente que o criou como pelos outros agentes que estão usando a área de trabalho a-areadetrabalho.

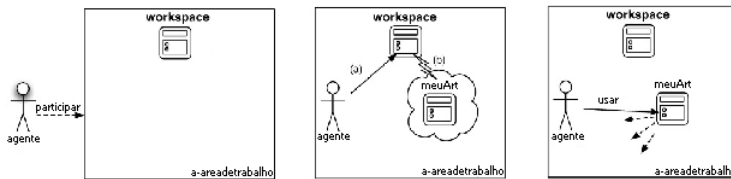


Figura 6 – Exemplos de interações dos agentes com uma área de trabalho, retirada e adaptada de (RICCI et al., 2009).

Na Figura 7 pode-se observar um agente utilizando um artefato. O agente seleciona a operação *minhaOp* na interface de utilização do artefato, desencadeando o início de uma nova instância da operação *minhaOp* dentro do artefato. A execução da operação eventualmente gerará sinais para os agentes que o estejam focando, além de possivelmente alterar as propriedades observáveis do artefato.

A Figura 8 demonstra um agente focando um artefato, assim ele passa a perceber todas as mudanças de valor das propriedades observáveis e os sinais que são gerados por este artefato como percepções.

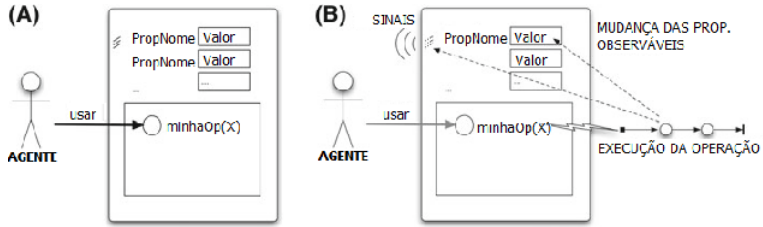


Figura 7 – Utilização de um artefato, retirada e traduzida de (RICCI; PIUNTI; VIROLI, 2010).

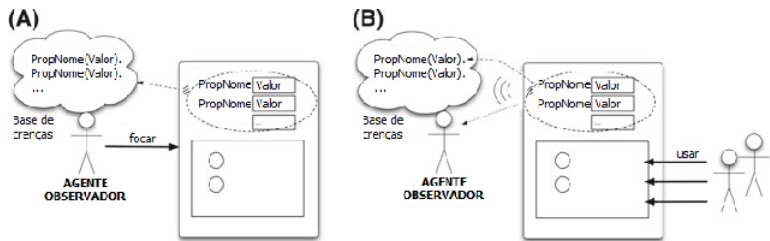


Figura 8 – Foco em um artefato, retirada e traduzida de (RICCI; PIUNTI; VIROLI, 2010).

Para uma melhor compreensão do modelo dos artefatos, em (RICCI et al., 2009) é citado como exemplo um artefato simples: uma máquina de café. O conjunto de botões da máquina de café representam as operações disponíveis; a tela e as luzes representam as propriedades observáveis e os sinais, como um som indicando que o café está pronto, seriam sinais gerados pelo artefato.

Em (RICCI; VIROLI; OMICINI, 2008b) é citado um outro exemplo, mais complexo, do modelo de A&A utilizando o framework Cartago, que será explicado em mais detalhes em 2.2.5. O exemplo é um Mundo de Quartos (Rooms World) composto por quartos e corredores, o objetivo dos agentes é manter estes quartos limpos, sendo que eles ficam sujos depois de uma quantidade arbitrária de tempo. Há artefatos disponíveis para auxiliá-los, como uma checklist na frente de cada quarto e um relógio que conta o tempo simulado. Há dois tipos de agentes: os normais, que apenas vão olhando os quartos aleatoriamente a procura de sujeira; e os que utilizam os artefatos, utilizando a checklist de cada quarto e o relógio para anotar o horário em que o quarto foi limpo, e

assim poder decidir limpar um quarto ou não de acordo com quanto tempo passou desde a última limpeza. Neste exemplo, os agentes que utilizavam os artefatos obtiveram uma melhor performance do que os normais, pois não perdiam recursos procurando sujeira em quartos recentemente limpos.

### 2.2.3 Funcionalidades dos Artefatos

Artefatos podem ter várias funções, (RICCI; PIUNTI; VIROLI, 2010) cita as seguintes: coordenação de agentes, modelagem de mecanismos sociais/organizacionais e modelagem de recursos.

Coordenação de agentes: Como os artefatos são entidades compartilhadas e podem ser usadas concorrentemente pelos agentes, eles podem facilmente ser utilizados como coordenadores. Um artefato coordenador pode encapsular o estado e as regras que definem a política de coordenação do SMA. Isso serve para manter o funcionamento da coordenação fora do conhecimento dos agentes e permitir que as regras da coordenação sejam alteradas em tempo de execução, através de adaptações realizadas pelos agentes que tiverem permissão para isso. Alguns exemplos de artefatos coordenadores são: quadros-negros (blackboards), buffers limitados (bounded-buffers), barreiras (barriers), agendas (task schedulers), relógios, tabuleiros de jogos, entre outros.

Quadro-negros podem ser definidos como uma memória de compartilhamento global, onde existem informações, conhecimentos e requisições que podem ser lidos e escritos pelos agentes (FARACO, 1998). Quadro-negros frequentemente são implementados como espaços de tuplas, um repositório de tuplas que pode ser acessado pelos agentes concorrentemente e onde eles podem inserir (out), ler (rd) e remover (in) conjuntos de dados estruturados chamados tuplas (RICCI; PIUNTI; VIROLI, 2010). O artefato padrão blackboard funciona desta maneira e é implementado como um espaço de tuplas, um exemplo o utilizando é mostrado no Apêndice B.2. Buffers limitados são utilizados para sincronização em interações produtor/consumidor. Barreiras fazem com que uma determinada ação só possa ser efetuada após uma certa condição de sincronização ter sido cumprida.

(RICCI; PIUNTI; VIROLI, 2010) cita que na programação de SMAs sem artefatos um quadro-negro seria implementado como um agente, o que seria uma quebra de conceitos já que um quadro negro tipicamente não é modelado para ser autônomo e pró-ativo, mas sim para ser



usado por outros agentes para comunicação e coordenação. Adotando a programação de ambiente o quadro-negro é implementado simplesmente como um recurso do ambiente, um artefato, que é acessado pelos agentes em termos de ações e percepções.

(RICCI; PIUNTI; VIROLI, 2010) também descreve um exemplo citando as vantagens de se modelar uma barreira como um artefato em uma situação em que para se sincronizar as atividades de  $N$  agentes é necessário que antes de fazer a atividade T2, todos tenham que ter feito a atividade T1. Utilizando uma solução baseada somente na transmissão de mensagens cada agente teria que passar  $N-1$  mensagens para os outros agentes, para assim todos saberem quando os outros concluíram T1, essa solução levaria a uma troca de  $N(N-1)$  mensagens. Utilizando uma solução com um agente mediador funcionando como um coordenador, o número de mensagens é reduzido para  $2N$ . Porém se for utilizado um artefato para implementar uma barreira de sincronização que forneça uma operação `sincronizar()`, de forma que cada artefato só precise chamar uma vez esta operação, são requisitadas apenas  $N$  ações para sincronizar. Um exemplo utilizando um artefato sincronizador é descrito em B.2.

Mecanismos sociais/organizacionais: No método tradicional de construção de SMAs estes mecanismos costumam ser modelados como uma camada extra de agentes mediadores em que qualquer ação de algum dos agentes participantes que precise ser regida por uma lei deve ser realizada como uma ação comunicativa com os agentes mediadores. A programação de ambiente permite uma perspectiva diferente, no qual os artefatos modelam os mecanismos organizacionais e as regras de interação.

Como um exemplo simples, (RICCI; PIUNTI; VIROLI, 2010) cita desenvolver um Jogo da Velha como um SMA. Os agentes seriam utilizados diretamente para implementar os jogadores, e o tabuleiro, que define e aplica as regras do jogo, seria modelado como um artefato utilizado pelos agentes jogadores. O tabuleiro definiria que ações um agente pode fazer dependendo do estágio do jogo (como “novo jogo” para iniciar uma nova partida e “movimento” para jogar durante ela). O artefato também apresentaria o estado do jogo através de suas propriedades observáveis, como “turno” (vez de qual jogador), “estado” (estado do jogo: jogando ou terminado) e “pos(X,Y,S)” (para armazenar o conteúdo das células). Um agente observando o artefato efetuaria um movimento no tabuleiro assim que percebe-se que é o seu turno. (RICCI; PIUNTI; VIROLI, 2010) mostra um exemplo com o código do artefato citado e

um exemplo abstrato do código dos agentes que o utilizariam.

**Modelagem de recursos:** Um artefato pode ser usado para criar uma camada de abstração para modelar qualquer tipo de recurso que não seja um agente, tanto recursos internos (como base de conhecimento, agenda pessoal, calculadora, GUI (Interface Gráfica com o Usuário), biblioteca...) ou encapsular recursos externos (como Web Services, banco de dados, sistemas legados, bibliotecas...). Segundo (RICCI; PIUNTI; VIROLI, 2010) alguns benefícios de modelar os recursos como artefatos são: (i) extensão do repertório de ações dos agentes sem ter que estender a linguagem ou arquitetura do agente; (ii) reduz a carga computacional dos agentes, que não precisam se preocupar com detalhes dos recursos, pois são executados dentro dos artefatos; (iii) aumenta a reusabilidade dos recursos; (iv) recursos podem ser criados, modificados ou descartados dinamicamente em tempo de execução.

#### **2.2.4 Características do modelo de Agentes & Artefatos**

Algumas características relevantes dos artefatos são: maleabilidade (podem ser alterados, criados ou descartados pelos agentes dinamicamente), inspectabilidade (através das propriedades observáveis e dos sinais), predictabilidade (o artefato fará apenas as operações que for solicitado, e em etapas definidas) e distributividade (artefatos podem ser distribuídos por diferentes áreas de trabalho, que podem estar distribuídas por diferentes nodos da rede). Além disso, os artefatos permitem que as ações sejam distribuídas entre agentes e também ao longo do tempo (RICCI; PIUNTI; VIROLI, 2010; RICCI; VIROLI; OMICINI, 2008b).

Uma característica dos artefatos que deve ser frisada é que eles são computacionalmente mais leves que os agentes, afinal eles são apenas entidades passivas e reativas. Por isso características do ambiente que não necessitam da complexidade de um agente para funcionarem sendo modeladas como artefatos diminuem a carga computacional do sistema, além de ser uma forma mais intuitiva de modelagem.

Em relação ao modelo de A&A como um todo, (RICCI; PIUNTI; VIROLI, 2010; RICCI et al., 2009) citam algumas características, apresentadas a seguir, desejáveis que um modelo para programação de SMAs deve ter. O modelo de A&A atende estas características, especialmente na sua implementação com o framework Cartago.

- **Abstração:** O modelo deve preservar a abstração utilizada no

nível de agentes e seus conceitos devem ser efetivos e genéricos o suficiente para capturar as principais propriedades de um ambiente.

- Modularidade: O modelo deve adotar conceitos para modularizar o ambiente, evitando a visão monolítica e centralizada que é geralmente utilizada para se modelar SMAs.
- Ortogonalidade: O modelo deve ser tão independente quanto possível dos modelos, arquiteturas e linguagens adotadas na programação de agentes, e assim ser capaz de suportar naturalmente sistemas heterogêneos.
- Extensibilidade dinâmica: O modelo de programação deve suportar a construção, alteração e extensão dinâmica de partes do ambiente, visando ser possível utilizá-lo em sistemas abertos.
- Reusabilidade: O modelo deve promover o reuso de partes do ambiente em diferentes contextos de aplicação.

Além disso, de acordo com (RICCI et al., 2009) o modelo de A&A provê suporte direto e é especialmente vantajoso para SMAs que sejam:

- Distribuídos: Os agentes podem participar e trabalhar simultaneamente em múltiplas áreas de trabalho, possivelmente situadas em diferentes nodos da rede. Embora artefatos simples não possam ser distribuídos por diferentes áreas de trabalho, é possível distribuir artefatos compostos.
- Heterogêneos: O modelo de A&A e sua implementação na plataforma Cartago são ortogonais ao modelo utilizado para programar os agentes, isso permite que agentes de diferentes tipos trabalhem juntos em um mesmo ambiente, compartilhando artefatos.
- Abertos: Agentes podem entrar e sair de áreas de trabalho dinamicamente, e podem alterar o conjunto de artefatos das áreas de trabalho dinamicamente. Restrições quanto a isso podem ser feitas utilizando uma política de papéis, explicada em mais detalhes na seção 2.2.5

Por fim, (OMICINI; RICCI; VIROLI, 2008a) cita algumas áreas com aplicações possíveis para o modelo de A&A : AOSE (Engenharia de Software Orientada a Agentes), MABS (Multi-Agent Based Simulation, Simulações Baseadas em Multi-Agentes), SOS (Self-Organizing Systems, Sistemas

Auto-Organizadores) e como base para a criação de linguagens e infraestruturas para SMAs. Em MABS e SOS artefatos servem para modelar as partes do sistema que são melhor representadas como entidades passivas, distribuídas e orientadas a função.

### 2.2.5 Framework de A&A Cartago

Há dois frameworks principais para se trabalhar como o modelo A&A: simpA e Cartago. simpA é descrito separadamente no Apêndice B.1, pois o framework usado neste trabalho é o Cartago, devido a ele já possuir uma integração com a linguagem Jason. A diferença principal entre eles é que o simpA é uma linguagem desenvolvida do zero, enquanto o Cartago foi desenvolvido para ser usado de forma ortogonal sobre algum framework de agentes já existente, em especial, o Jason.

Segundo (RICCI et al., 2009) Cartago (Common Artifact infrastructure for Agent Open environment, Infraestrutura Comum de Artefatos para Ambientes Abertos de Agentes) é uma plataforma e infraestrutura que provê um modelo de programação de propósito geral para construir modelos computacionais compartilhados (áreas de trabalho) em que agentes, possivelmente heterogêneos, possam trabalhar juntos. A plataforma é baseada no modelo A&A, assim os ambientes do Cartago são modelados e construídos em termos de um conjunto de artefatos, reunidos em áreas de trabalho. Cartago é código aberto e é implementado em cima da plataforma Java.

Em detalhes, a plataforma inclui:

- Uma API baseada em Java para programar e definir novos tipos de artefatos em termos de classes Java e tipos básicos de dados.
- Uma API que permite que os agentes utilizem o ambiente do Cartago, fornecendo um conjunto de ações para criar e interagir com os artefatos, e para administrar e participar de áreas de trabalho.
- Um ambiente runtime e ferramentas relacionadas que suportam a distribuição e administração do ambiente, e a administração do ciclo de vida das áreas de trabalho e dos artefatos.

Para definir e administrar os aspectos de segurança das áreas de trabalho em um SMA, Cartago utiliza um modelo RBAC (Role-Based Access Control, Controle de Acesso Baseado em Papéis). Isso é feito através da definição de papéis e para cada papel definir políticas especificando quais ações em quais artefatos um agente neste papel

pode ou não efetuar. Também pode-se controlar quais agentes podem acessar uma área de trabalho. Como o modelo RBAC é implementado utilizando o artefato padrão workspace, o conjunto de papéis e políticas pode ser inspecionado e alterado dinamicamente.

Cartago é ortogonal quanto aos modelos e plataformas específicas utilizadas para se construir os agentes que utilizam as áreas de trabalho. A intenção é permitir que até mesmo agentes de modelos heterogêneos possam trabalhar em um mesmo ambiente. Para essa integração ser possível foram criados dois conceitos: corpo do agente (agent body) e mente do agente (agent mind). O corpo do agente é a parte do agente que está situada na área de trabalho, contém atuadores (effectors) para agir sobre os artefatos e sensores (sensors) para perceber os eventos observáveis gerados na área de trabalho. A mente do agente é executada externamente, pela plataforma usada para construir o agente.

De um ponto de vista arquitetural são introduzidas pontes (bridges) específicas para cada plataforma de agentes para conectar corpos e mentes, encapsulando o corpo da mente do agente, e assim permitindo que ela controle o corpo e perceba os estímulos coletados pelos sensores. Atualmente existem pontes para as plataformas Jason, Jadex e simpA, mais informações sobre estas pontes estão disponíveis em (RICCI et al., 2009; RICCI; PIUNTI; VIROLI, 2010), e sobre a ponte com Jadex em (RICCI et al., 2008). Como Jason é a plataforma de agentes que foi primeiramente integrada com o Cartago, e a que possui mais exemplos de código, ela foi escolhida como a plataforma de agentes deste trabalho.

Uma representação de como esta integração funciona aparece na Figura 9. Nela é mostrada uma aplicação SMA e uma camada de plataformas de execução que dão suporte a essa aplicação. No nível da aplicação são mostrados agentes interagindo entre si (setas pontilhadas), utilizando artefatos (setas contínuas) e recebendo percepções dos artefatos (setas tracejadas). Também são mostrados quatro artefatos distribuídos em duas áreas de trabalho, sendo que o artefato “KB compartilhada” (knowledge base, base de conhecimento) está em ambas. No nível das plataformas é mostrado como a aplicação é suportada pelos middlewares, cada mente de agente é modelada em alguma plataforma de agente (Jason, Jadex...) , os artefatos e áreas de trabalho são executados pelo Cartago, enquanto as relações entre eles são modeladas como corpos de agentes, permitindo o recebimento de percepções e a execução de ações. As setas ligando os corpos com as mentes dos agentes representam as pontes.

A integração amplia o repertório de ações dos agentes, nativamente provido pela sua plataforma original, com um novo conjunto

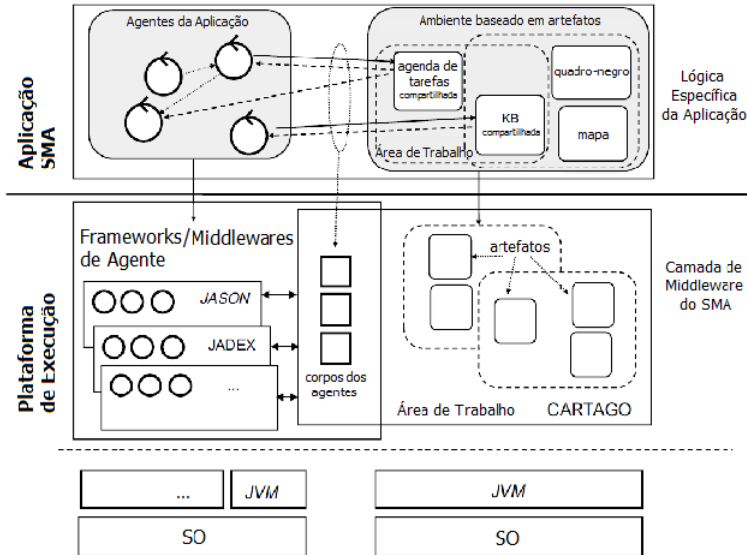


Figura 9 – Representação de como a integração entre as plataformas de agentes e o Cartago funciona, retirada de (RICCI et al., 2009).

de ações para tornar possível que utilizem o ambiente baseado em artefatos. Abaixo serão explicadas as novas ações, mais explicações encontram-se na seção 2.2.2 e na referência (RICCI; PIUNTI; VIROLI, 2010). A sintaxe utilizada é `Name(Params):Feedback` para definir a assinatura das ações, que inclui o nome da ação, parâmetros e opcionalmente o feedback da ação.

- Criar, entrar e sair de áreas de trabalho locais ou remotas:
  - `createWorkspace(AreaDeTrabNome)` - Cria uma nova área de trabalho com o nome `AreaDeTrabNome`.
  - `joinWorkspace(AreaDeTrabNome):AreaDeTrabId` - Participa da área de trabalho `AreaDeTrabNome`, retorna o identificador dela.
  - `joinRemoteWorkspace(AreaDeTrabNome,NodoId):AreaDeTrabId` - Participa da área de trabalho remota `AreaDeTrabNome` localizada no nodo `Nodoid`, retorna o identificador dela.
  - `quitWorkspace()` - Sair de uma área de trabalho.

- Criar, procurar e descartar artefatos:
  - `makeArtifact(ArtNome, ArtTipoNome, ParametrosIniciais): ArtId` - Cria um novo artefato chamado `ArtNome` do tipo `ArtTipoNome` dentro da área de trabalho. Artefatos podem possuir um mesmo nome lógico em áreas de trabalho diferente, por isso é retornado um `ArtId` como um identificador único do artefato no SMA. `ArtTipoNome` é o nome da classe Java da qual este artefato será instanciado.
  - `disposeArtifact(ArtId)` - Remove um artefato da área de trabalho.
  - `lookupArtifact(ArtName): ArtId` - Localiza um artefato dado seu nome lógico e retorna o identificador deste artefato.
  - `lookupArtifactByType(ArtTipoNome): ArtId` - Retorna o conjunto de artefatos que são instâncias do tipo `ArtTipoNome`.
- Usar artefatos:
  - `OpNome(Parametros)` - Dada uma operação que exista em algum artefato da área de trabalho em que o agente está, esta operação é executada. A ação completa com sucesso caso a operação complete com sucesso. A ação falha se a operação falhar ou se o artefato não possuir esta operação no momento. Para se obter um retorno da operação é necessário usar um parâmetro do tipo `OpFeedbackParam` na operação do artefato, um exemplo utilizando este parâmetro é mostrado em (RICCI; PIUNTI; VIROLI, 2010), ele também está disponível no site do Cartago. Caso uma área de trabalho possua múltiplos artefatos com o mesmo nome de operação pode-se utilizar a anotação `[artifact_name(IdArtefato, "NomeArtefato")]` para especificar de qual artefato é a operação.
- Observar artefatos:
  - `focus(ArtId, Filter)` - Observa continuamente os eventos gerados pelo artefato identificado por `ArtId`. Pode ser definido um filtro para selecionar apenas os eventos que o agente estiver interessado. Para agentes BDI, as propriedades observáveis são mapeadas diretamente como crenças na base de crenças. Um agente pode focar múltiplos artefatos ao mesmo tempo. Sempre que um propriedade observável for

alterada ou um sinal for gerado um evento observável será enviado para todos os agentes que estejam observando o artefato. É importante notar que um agente pode utilizar um artefato sem necessariamente observá-lo.

- `stopFocus(ArtId)` - Para de observar um artefato.
- `observeProperty(PropNome):PropValor` - Lê o valor atual de uma propriedade observável. Neste caso o valor da propriedade é retornado pela ação, sem ser gerado uma percepção.
- Conectar e desconectar artefatos:
  - `linkArtifacts(LinkingArtId, LinkedArtId, ,Porta)` - Conecta o artefato `LinkingArtId` ao `LinkedArtId`. O parâmetro `Porta` é necessário quando se está conectando o mesmo artefato com múltiplos artefatos.
  - `unlinkArtifacts(LinkingArtId, LinkedArtId)` - Desconecta os artefatos `LinkingArtId` e `LinkedArtId`.

De acordo com (RICCI; PIUNTI; VIROLI, 2010) tipos de artefatos são modelados diretamente definindo uma classe Java que estenda a biblioteca `cartago.Artifact`, e usando um conjunto básico de anotações Java, suportadas a partir do JDK 1.5, e métodos herdados para definir os elementos da estrutura e comportamento dos artefatos. Um artefato concreto é criado como uma instância de um tipo de artefato.

Todo artefato possui um método `init`, que funciona como um construtor de um objeto, esse método define como a instância do artefato será inicializada em tempo de criação e pode receber parâmetros para definir características do artefato. Se esse método gera um erro o artefato não é criado e a ação do agente falha.

Abaixo estão descritas as primitivas disponíveis para os artefatos utilizarem:

- `defineObsProperty(String PropNome, Object... args)` - Cria uma nova propriedade observável com um nome e um valor inicial, que pode ser qualquer tupla de objetos de dados.
- `getObsProperty(String PropNome):ObsProperty` - Retorna um objeto `ObsProperty` encapsulando a propriedade observável.
- `prop.updateObsProperty(Object... args)` - Atualiza a propriedade observável com um determinado valor, sendo “prop” um objeto `ObsProperty`.



- `signal(String SinalNome, Object... args )` - gera um sinal que é percebido por todos os agentes que estão observando o artefato.
- `signal(AgentId id, String SinalNome, Object... args )` - gera um sinal que é percebido por apenas um agente específico, caso ele esteja observando o artefato.
- `await(String NomeGuarda, Object... args )` - Suspende a execução da operação até que a condição da guarda seja atendida, quebrando a execução da operação em múltiplas etapas. Ao suspender a execução de uma operação as outras operações do artefato podem ser invocadas antes da operação atual ser concluída. Quando a condição da guarda é atendida e nenhuma outra operação está em execução, a operação suspensa continua sua execução.
- `await_time(int TempoMilisegundos)` - Suspende a execução da operação até que o tempo específico, em milisegundos, tenha passado. Como no caso da primitiva `await`, ao suspender a operação o artefato se torna acessível para que os agentes executem outras operações e podem ocorrer atualizações no seu estado observável.
- `execInternalOp(String OpInternaNome, Object... args )` - Executa a operação interna `OpInternaNome`.
- `failed(String MensagemDeFalha)` - Gera uma exceção que interrompe o fluxo de controle na execução do método.
- `failed(String MensagemDeFalha, String Descricao, Object... args )` - Gera uma exceção que interrompe o fluxo de controle na execução do método. Além da mensagem também é enviado uma descrição da falha.
- `execLinkedOp(ArtifactId id, String OperacaoOutroArtefato, Object... args )` - Executa a operação `OperacaoOutroArtefato` do artefato `ArtefatoId` com os parâmetros indicados.
- `execLinkedOp(String Porta, String OperacaoOutroArtefato, Object... args )` - Executa a operação `OperacaoOutroArtefato` pela porta indicada com os parâmetros indicados.

Abaixo são descritas as anotações Java sobre métodos utilizadas no Cartago (RICCI; PIUNTI; VIROLI, 2010):

- **@OPERATION** [(guard=“métodoGuard”)]: Usada para indicar que um método é uma operação do artefato. O corpo do método será o comportamento que a operação possui. Uma operação pode opcionalmente possuir uma guarda. O método deve possuir retorno void, e os parâmetros do método serão usados tanto como entradas e saídas da operação.
- **@GUARD**: Um método de retorno booleano que define as condições que uma operação com guarda ou um await precisam possuir para continuar a execução.
- **@INTERNAL\_OPERATION**: Indica que um método é uma operação interna, ou seja, não está disponível na interface de utilização.
- **@LINK**: Indica que um método que é uma operação da interface de conexão do artefato.

Algumas observações:

- O mesmo método pode ser marcado com **@OPERATION** e **@LINK**.
- Ao invés de usar **@INTERNAL\_OPERATION** e `execInternalOp`, obtém-se o mesmo efeito deixando um método sem nenhuma anotação e chamando-o da maneira convencional de OO após uma chamada `await`.

Exemplos de implementações com Cartago são descritos no Apêndice B.2. Uma comparação entre ambientes de agentes modelados em Jason e em Cartago é abordada no Apêndice B.3.

## 2.3 ONTOLOGIAS E REPOSITÓRIOS SEMÂNTICOS

### 2.3.1 Web Semântica

Atualmente o conteúdo da Web consiste principalmente de arquivos hipertexto e hipermídia distribuídos, acessíveis através de links e de busca por palavra-chave (HORROCKS, 2008). O grande aumento da quantidade e de tipos de conteúdo na Web causa problemas a este paradigma hipertexto, tornando cada vez mais difícil localizar o conteúdo desejado. Além disso o paradigma possui pouco suporte para consultas complexas e para a recuperação, integração, compartilhamento e processamento de informações (HORROCKS, 2008).

Neste contexto, a Web Semântica surgiu como uma extensão da Web atual. Segundo (HORROCKS, 2008; HENDLER; HARMELEN, 2008) seu objetivo é permitir que dados sejam compartilhados efetivamente por grandes comunidades e que possam ser processados automaticamente tanto por máquinas como por humanos. (HORROCKS, 2008) cita que uma das maiores dificuldades em se cumprir este objetivo é que o conteúdo Web é primeiramente intencionado para ser consumido por usuários humanos e, além disso, grande parte deste conteúdo é desestruturado (textos, imagens, vídeos), sendo assim difícil processá-lo ou compartilhá-lo.

Considerando estes problemas e reconhecendo que ontologias podem solucioná-los, pois provêm um vocabulário de termos com significado bem-definido e legível por máquinas, concluiu-se que um padrão de uma linguagem para ontologias é um pré-requisito para desenvolver a Web Semântica (HORROCKS, 2008), em 2001 a W3C criou um grupo de trabalho para desenvolver este padrão. O resultado, em 2004, foi a primeira versão da linguagem OWL construída sobre a linguagem RDF. Nas seções 2.3.2 e 2.3.3. serão explicadas em maiores detalhes o que são ontologias e as linguagens RDF e OWL.

### 2.3.2 Ontologias

De acordo com (HORROCKS, 2008), para a ciência da computação uma ontologia serve para modelar algum aspecto do mundo, ela introduz um vocabulário que descreve características do domínio sendo modelado e provê uma definição explícita do significado deste vocabulário. (HORROCKS, 2008) comenta diversas áreas onde ontologias são utilizadas como biologia, medicina, geografia, geologia e agricultura. Algumas ontologias relevantes da área biomédica desenvolvidas na linguagem OWL citadas pelo autor são SNOMED (Systematized Nomenclature of Medicine-Clinical Terms), GO (Gene Ontology), BioPAX (Biological Pathways Exchange), e U.S. National Cancer Institute thesaurus.

(SCHREIBER, 2008) acrescenta que ontologias são divididas em categorias, sendo os dois tipos principais: ontologia de domínio e ontologia superior. Ontologias de domínio são as mais comuns, e servem para compartilhar conceitos e relações de uma área de interesse em particular, como exemplo as ontologias para biomedicina referidas no parágrafo anterior. Ontologias superiores provêm conceitos gerais que podem ser utilizados por diversas ontologias de domínio, um exemplo é a ontologia Dublin Core (WEIBEL, 1997), que busca utilizar metadados

para descrever objetos digitais.

A principal utilidade de uma ontologia é representar conhecimento sobre um domínio, segundo (WIKIPEDIA, 2012) as ontologias compartilham várias similaridades estruturais, independentemente da linguagem na qual são representadas. A maioria das ontologias utiliza os conceitos de indivíduos (instâncias de classes, objetos), classes (conceitos, conjuntos, tipos de objetos), propriedades (atributos que indivíduos ou classes podem ter) e relações (relacionamentos entre classes e indivíduos). Também é comum existirem restrições (devem ser verdadeiras para que um axioma seja adicionado) e regras (sentenças na forma se...então que descrevem inferências lógicas). Uma ferramenta frequentemente utilizada para o desenvolvimento de ontologias é a Protégé (KNUBLAUCH et al., 2004), um exemplo de uma ontologia desenvolvida em OWL com o Protégé é descrita na seção 2.3.3.

Outra funcionalidade importante das ontologias é permitir que possam ser realizadas inferências sobre este conhecimento. (W3C, 2012) descreve inferência como a capacidade de se descobrir, através de procedimentos automáticos, novas relações dentro de uma ontologia baseando-se em seus dados, vocabulário e regras sobre estes dados. Mecanismos de inferência também permitem realizar consultas complexas, checar inconsistências e depurar a ontologia.

(W3C, 2012) cita dois exemplos comuns de inferências: 1) Se existem as relações “Flipper é um Golfinho” e “Golfinhos são Mamíferos”, pode-se inferir que “Flipper é um Mamífero”, mesmo que isso não esteja explícito na base de dados; 2) A regra “Se duas Pessoas possuem o mesmo nome, homepage e endereço de e-mail, então elas são a mesma pessoa” pode ser usada para se inferir que duas pessoas possuem a mesma identidade.

Algumas APIs de motores de inferência OWL frequentemente utilizados são: Pellet (SIRIN et al., 2007), FaCT++ (TSARKOV; HORROCKS, 2006) e Racer (HAARSLEV; MÜLLER, 2001).

### 2.3.3 RDF e OWL / OWL 2

De acordo com (W3C, 2012) a linguagem RDF (Resource Description Framework) (BRICKLEY; GUHA, 2004) permite descrever recursos Web e relações entre eles. Uma característica chave da linguagem é a utilização de identificadores internacionalizados de recursos (internationalized resource identifier, IRIs) para se referir diretamente a recursos não-locais, facilitando a integração da informação. RDF é uma lingua-

gem simples, estruturada na forma de um grafo rotulado direcionado e sua única construção sintática são tuplas na forma SUJEITO - PREDICADO - OBJETO, como em “Mariana é\_mãe\_de Carlos”. Para facilitar o compartilhamento de grafos RDF na Web, esta linguagem possui uma serialização para XML.

Embora a linguagem RDF permita adicionar anotações aos dados de forma estruturada, ela não é capaz de resolver o problema de compreensão do significado e da semântica dos termos das anotações. Uma extensão do RDF, o RDFS (RDF Schema) (BRICKLEY; GUHA, 2004), foi criada para amenizar este problema, mas ainda assim é bastante limitada como uma linguagem para ontologias, pois não permite descrever, por exemplo, restrições de cardinalidade ou conjunção de classes.

A Web Ontology Language (OWL) (MCGUINNESS; HARMELEN et al., 2004) , lançada em 2004 como uma recomendação da W3C, é uma linguagem para ontologias mais expressiva em relação ao RDF/RDFS, e construída sobre o RDF/RDFS. (HENDLER; HARMELEN, 2008) descreve que a OWL pode ser utilizada para definir classes e propriedades assim como em RDFS, mas oferece funcionalidades adicionais, como: descrições de classes como combinações lógicas de outras classes (intersecções, união, complemento, equivalência), restrições de cardinalidade em propriedades, restrições de quantificação (someValuesFrom e allValuesFrom) e definição de propriedades inversas, transitivas, simétricas ou funcionais.

(HENDLER; HARMELEN, 2008) afirma que a OWL é única de duas maneiras: 1) É a primeira linguagem para ontologias com uma expressividade razoável a se tornar um padrão notoriamente reconhecido, o que é bastante importante para o reuso de ontologias e para a interoperabilidade das ferramentas que trabalham com elas. 2) OWL é a primeira linguagem para ontologias amplamente utilizada cujo design é baseado na arquitetura Web, isso ocorre porque é uma linguagem não-proprietária; utiliza Universal Resource Identifiers (URIs) para identificar recursos de forma não ambígua na Web (similar ao IRI); suporta a ligação de termos entre ontologias, permitindo referência cruzada e reuso de informação; e é baseada em RDF/XML, facilitando o intercâmbio de dados. Uma outra característica relevante da OWL é que ela trabalha com a suposição de um mundo aberto, que é mais adequada a Web, pois normalmente não é razoável assumir que se possui informação completa sobre um domínio.

Na sua primeira versão a OWL provê três sub-linguagens inten-

cionadas para diferentes aplicações, em ordem de expressividade elas são: OWL Lite, OWL DL, e OWL Full. Uma das linguagens mais utilizadas (HENDLER; HARMELEN, 2008) é a OWL DL, pois possui um modelo teórico formal baseado na Lógica Descritiva, uma família de formalismos para se representar conhecimento baseados em um conjunto decidível da lógica de primeira ordem. Por causa desta fundamentação teórica, a OWL DL permite que inferências sejam feitas de forma decidível, completa e automática, enquanto ainda mantém um bom nível de expressividade.

Segundo (HENDLER; HARMELEN, 2008) uma ontologia OWL consiste de um conjunto de axiomas TBox e ABox. O axiomas TBox descrevem restrições à estrutura do domínio, um exemplo são os axiomas que definem hierarquias de classes e propriedades, como `subClassOf`. Os axiomas ABox declaram fatos sobre situações concretas, como os axiomas “Augusto é uma instância de Estudante” e “Akita, instância de Cachorro, é um animal de estimação de Augusto”.

Na Figura 10 é mostrado um exemplo de uma ontologia OWL desenvolvida na ferramenta Protégé. Nela existem as classes “Animal”, “Mamífero”, “Pessoa” e “Cachorro”, havendo uma relação de subclasse entre elas, sendo que “Pessoa” e “Cachorro” são classes disjuntas. A classe “Pessoa” possui três instâncias (Ana, Alice e Augusto) e a classe “Cachorro” possui os indivíduos Akita e Rex. Existem as propriedades de dados nome, sexo e CPF, sendo que CPF tem seu domínio limitado apenas para indivíduos da classe Pessoa e é uma propriedade funcional, ou seja, cada indivíduo só pode ter um CPF. Além das propriedades padrão sobre objetos `has_individual` e `has_subclass`, também existem as propriedades sobre objetos `e_colega` (Pessoa =<sub>i</sub> Pessoa), `e_pet` (Cachorro =<sub>i</sub> Pessoa) e `tem_pet` (Pessoa =<sub>i</sub> Cachorro), com os respectivos domínio e alcance entre parênteses. A propriedade `tem_pet` é inversa de `e_pet`, e a propriedade `e_colega` é simétrica.

Usando o racionador Hermit 1.3.4 (SHEARER; MOTIK; HORROCKS, 2008) sobre esta ontologia foram geradas as seguintes inferências: da relação “Alice `tem_pet` Akita” inferiu-se “Akita `e_pet` Alice”, idem para Ana e Rex; e para a relação “Alice `e_colega` Augusto” inferiu-se “Augusto `e_colega` Alice”.

Para se realizar consultas sobre ontologias OWL pode-se usar a SPARQL (SPARQL Protocol and RDF Query Language) (PÉREZ; ARENAS; GUTIERREZ, 2009). SPARQL é uma linguagem para consulta e manipulação de dados RDF. A consulta é feita através do casamento de padrões de grafos ou triplas RDF, e também da conjunção, disjunção ou união destes grafos. Esta linguagem é uma das tecnologias-chave da

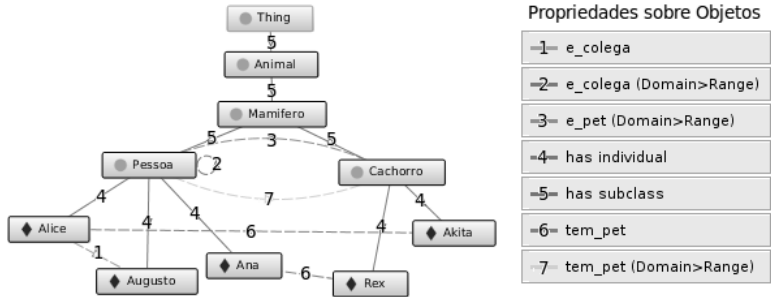


Figura 10 – Exemplo de ontologia OWL desenvolvida no Protégé.

Web Semântica e é uma recomendação oficial da W3C. A SPARQL permite trabalhar com os dados da Web Semântica de forma federada, possibilita realizar inferências sobre os dados consultados e também permite realizar consultas tanto sobre dados estruturados como semi-estruturados.

Abaixo é mostrado um exemplo de uma consulta em SPARQL, o objetivo dela é responder a pergunta “Quais são os gases nobres?”, após ela é mostrado o resultado retornado por esta consulta. Em PREFIX são definidas as ontologias que serão utilizadas, neste caso uma sobre a tabela periódica e uma auxiliar do XMLSchema. Em SELECT é definido quais dados a consulta retornará, neste caso o nome, símbolo, peso e número dos gases nobres. Em FROM é definida a base de dados que será utilizada. Em WHERE são definidas as condições da consulta, neste caso se deseja um elemento do grupo dos gases nobres, e após isso os axiomas da ontologia são associados com as variáveis da consulta.

```

PREFIX table: <http://www.daml.
2 org/2003/01/periodictable/PeriodicTable#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
4 SELECT ?nome ?simbolo ?peso ?numero
FROM <http://www.daml.org/2003/01/periodictable/
    PeriodicTable .
6 owl>
WHERE {
8 ?elemento table:group ?grupo .
  ?grupo table:name "Noble gas"^^xsd:string .
10 ?elemento table:name ?nome .
  ?elemento table:symbol ?simbolo .
12 ?elemento table:atomicWeight ?peso .
  ?elemento table:atomicNumber ?numero
14 }
```

ORDER BY ASC(?nome)

---

Tabela 1 – Resultado da consulta SPARQL “Quais são os gases nobres?”

nome	símbolo	peso	número
argônio	Ar	39.948	18
hélio	He	4.002602	2
criptônio	Kr	83.798	36
neônio	Ne	20.1797	10
radônio	Rn	222	86
xenônio	Xe	131.293	54

Por fim, a OWL2 é uma extensão da OWL que, entre outras mudanças, aumenta sua expressividade. Algumas das novas características descritas por (HENDLER; HARMELEN, 2008) incluem: restrições de cardinalidade qualificadas, permitindo, por exemplo, descrever que uma mão é composta por 4 dedos e 1 polegar; propriedades que podem ser reflexivas, irreflexivas, assimétricas ou disjuntas; e composição de propriedades em uma cadeia de propriedades, como para capturar o fato de que uma doença que afete uma parte de um órgão afeta o órgão como um todo. OWL2 também provê suporte estendido para tipos de dados e anotações. Informações mais detalhadas sobre a OWL2 podem ser encontradas em (GRAU et al., 2008).

### 2.3.4 Repositórios Semânticos

A explosão do uso de RDF para representar informações sobre recursos fez com que fosse necessário sistemas em escala Web que pudessem armazenar e processar uma quantidade massiva de dados, e que também pudessem prover funcionalidades poderosas para o acesso e mineração destes dados (OPENRDF, 2012b).

Neste contexto foram desenvolvidos os repositórios semânticos, segundo (OPENRDF, 2012b) não existe uma definição formal para este termo, os autores definem repositórios semânticos como uma ferramenta similar a um Sistema de Gerenciamento de Banco de Dados (SGBD), que pode ser usada para armazenar, administrar e realizar



consultas sobre dados estruturados de acordo com o padrão RDF. (ONTOTEXT, 2012) acrescenta que as maiores diferenças destes repositórios em relação a um SGBD são que eles utilizam ontologias como esquemas semânticos, permitindo que se possa realizar inferências sobre os dados; e que eles trabalham com um modelo de dados flexível e genérico (grafos), o que permite que eles facilmente adotem e interpretem novas ontologias ou esquemas de metadados em tempo de execução. Uma consequência disto é que os repositórios semânticos oferecem uma integração facilitada entre dados diversos e oferecem um maior poder analítico sobre os dados.

(ONTOTEXT, 2012) cita Sesame como um dos repositórios semânticos mais populares. Sesame(BROEKSTRA; KAMPMAN; HARMELEN, 2002) é um banco de dados RDF de código-aberto que suporta armazenamento, consulta e inferência sobre dados RDF(S). Ele é um framework Java que fornece uma API extensível, permitindo que outros repositórios possam ser construídos sobre ele. Sesame também é considerado um padrão de-facto para se processar dados RDF (OPENRDF, 2012a).

Outro exemplo citado pelos autores é o OWLIM. OWLIM(BISHOP et al., 2011; KIRYAKOV; OGNANOV; MANOV, 2005) é uma família de repositórios semânticos empacotados como uma camada de armazenamento e inferência (Storage and Inference Layer , SAIL) de alto desempenho para o Sesame. OWLIM suporta a linguagem de consulta SPARQL e oferece suporte para as linguagens RDF(S), OWL 2 RL e OWL 2 QL. Existem três versões de repositórios OWLIM: OWLIM-Lite, OWLIM-SE e OWLIM-Enterprise, que utilizam as mesmas semânticas e mecanismos de inferência. OWLIM-Lite foi projetado para volumes medianos de dados e para prototipação, ele é um repositório in-memory, com um rápido processamento. OWLIM-SE foi desenvolvido para uso comercial e permite administrar volumes massivos de dados. OWLIM-Enterprise é uma infraestrutura clusterizada baseada no OWLIM-SE.

Neste trabalho acessamos o repositório semântico nativo Open-Link Virtuoso(ERLING; MIKHAILOV, 2010). Virtuoso é um banco de dados SQL objeto-relacional que suporta consultas em SPARQL aos dados RDF. Ele também suporta dados relacionais típicos e dados em XML e possui um servidor para Linked Data. Este repositório é capaz de suportar até cerca de 1 bilhão de triplas.

### 2.3.5 DBPedia

DBPedia (BIZER et al., 2009) é construída sobre a Web Semântica, seu objetivo é extrair informações estruturadas da Wikipedia e disponibilizar estas informações na Web. Atualmente a base de conhecimento da DBPedia descreve mais de 2.6 milhões de entidades, juntamente com seus identificadores globais, relacionamentos, descrições, classificações, etc. Estas entidades pertencem a domínios como informações geográficas, pessoas, companhias, filmes, livros, e publicações científicas. A principal ontologia utilizada pela DBPedia é diretamente mapeada dos templates da Wikipedia. Outros ontologias utilizadas são a Yago (SUCHANEK; KASNECI; WEIKUM, 2007) e a FOAF (GRAVES; CONSTABARIS; BRICKLEY, 2007).

DBPedia fornece um endpoint SPARQL público <sup>1</sup> que permite que consultas sejam feitas a sua base de conhecimento. O repositório semântico OpenLink Virtuoso (ERLING; MIKHAILOV, 2010) é utilizado como um banco de dados back-end. Para utilizar o endpoint SPARQL nós adotamos o mecanismo de consultas da Jena API (MCBRIDE, 2002), um framework Java de código aberto que permite a construção de aplicações para a Web Semântica.

A DBPedia foi escolhida como a base de conhecimento para este trabalho pois é baseada na Wikipedia, uma fonte frequentemente utilizada por estudantes. Além disso, a DBPedia é considerada como a mais famosa base da Web Semântica (BERNERS-LEE, 2008).

## 2.4 FERRAMENTAS DE AUTORIA DE OBJETOS DE APRENDIZAGEM

### 2.4.1 Introdução

Nesta seção estudamos as ferramentas de autoria para facilitar o processo de levantamento de requisitos da ferramenta desenvolvida neste trabalho. Segundo a W3C (RICHARDS et al., 2012) uma ferramenta de autoria é qualquer aplicação ou coleção de aplicações que pode ser utilizadas por autores, colaborativamente ou não, para criar ou modificar conteúdo Web que será usado por outros autores ou usuários finais. Alguns exemplos de ferramentas de autoria segundo esta definição seriam: editores WYSIWYG, ferramentas de autoria multimídia, siste-

---

<sup>1</sup><http://DBpedia.org/sparql>

mas de gerenciamento de conteúdo e ferramentas de autoria de objetos de aprendizagem. (XAVIER; GLUZ, 2009) acrescenta que ferramentas de autoria em geral permitem que autores produzam aplicações gráficas através do agrupamento e sequenciamento de diferentes objetos, como texto, imagens, áudio e vídeo.

De acordo com a IEEE (IEEE, 2002) um objeto de aprendizagem (OA) é qualquer entidade que pode ser utilizada, reutilizada ou referenciada durante o aprendizado apoiado por computador, e pode ser algo simples como um texto ou um vídeo, ou mais complexo, como um hipertexto, um curso ou uma animação. (WILEY, 2000) considera que um objeto de aprendizagem deve possuir as seguintes características: ser auto-explicativo, modular, agregável, digital, interoperável e reutilizável.

Para (XAVIER; GLUZ, 2009) as ferramentas de autoria de objetos de aprendizagem possuem duas características principais: Oferecem um ambiente para a autoria de conteúdo digital e permitem gerar objetos em conformidade com padrões de metadados. Uma ferramenta de autoria deste tipo pode ter ambas ou apenas uma destas características. No caso deste trabalho, a ferramenta de autoria desenvolvida possui apenas a segunda característica, tendo como principal função auxiliar o preenchimento de metadados conforme um padrão de metadados.

(DUVAL et al., 2002) define metadados como dados estruturados sobre outros dados, que podem ser objetivos (dados sobre os autores, data de criação, versão...) ou subjetivos (palavras-chave, resumo, revisão...). Metadados servem para fornecer informações adicionais sobre um dado, e assim fornecer o contexto em que aquele dado existe e que relações ele possui com outros dados. Frequentemente metadados são inteligíveis por computadores.

Padrões de metadados estabelecem metadados com uma semântica definida buscando garantir o seu uso e interpretação corretos. Os padrões de metadados para OAs facilitam o compartilhamento, catalogação, descoberta e reuso dos objetos de aprendizagem. Alguns exemplos de padrões para OAs são: DC (Dublin Core) (WEIBEL, 1997), LOM (Learning Object Metadata) (IEEE, 2002), Scorm (Sharable Content Object Reference Model) (BOHL et al., 2002) e CanCore (FRIESEN; ROBERTS; FISHER, 2002).

### 2.4.2 Exemplos de Ferramentas de Autoria

Em (XAVIER; GLUZ, 2009) é realizado um estudo comparativo entre oito ferramentas de autoria de objetos de aprendizagem: Ardora, CourseLab, eXe Learning, FreeLoms, LomPad, Paloma, Reload e Xerte. Os critérios de comparação foram: facilidade de instalação, abrangência, grau de cobertura do padrão, grau de inteligência, disponibilidade do código-fonte, integração, adaptabilidade, existência de um manual, representação de metadados através de ontologias e enfoque pedagógico. Um resumo dos resultados para cada um destes critérios segue na Tabela 2 abaixo:

Tabela 2 – Comparação resumida entre ferramentas de autoria do estudo feito por (XAVIER; GLUZ, 2009)

Critério	Resultado
Facilidade de instalação	Fácil ou Web
Abrangência	As três primeiras permitem criação de conteúdo, a maioria dá suporte para os padrões Scorm 1.2 e LOM.
Grau de cobertura do padrão	Completo
Grau de inteligência	Possuem Dialog Box agrupados por nível de afinidade.
Disponibilidade do código-fonte	Sim, menos FreeLOms, LOMPad e Paloma.
Integração com algum LMS	Sim, de acordo com os padrões de metadados suportados.
Adaptabilidade dos OAs para outras plataformas	Sim para eXe Learning, LOMPad, RELOAD e Xerte.
Existência de um manual	Não
Representação de metadados através de ontologias	Não
Enfoque pedagógico	Não

Entre os resultados da pesquisa, os que mais chamaram a atenção em relação a este trabalho foram os itens “grau de inteligência” e “representação de metadados através de ontologias”, ambos tiveram um resultado unânime. Para o primeiro critério, todas as ferramentas ti-

nham apenas Dialog Box agrupados por nível de afinidade, sem automatização do preenchimento e com pouca ou nenhuma informação sendo fornecida quanto ao significado das tags dos padrões de metadados. Para o segundo critério, nenhuma das ferramentas analisadas representavam os metadados através de ontologias.

Algumas conclusões do próprio estudo sobre esta comparação foram: 1) É necessário facilitar o preenchimento de metadados, pois embora existam ferramentas que auxiliem no preenchimento conforme os padrões, esta ainda é uma tarefa árdua, seja pelo processo de classificação e denominação das informações, seja pela quantidade de campos a preencher. Isso tem como consequências lentidão na produção de novos recursos e baixa taxa de preenchimento de metadados, fazendo com que sejam pouco padronizados. 2) É preciso que uma vez que o OA esteja em um Learning Management System (LMS), ele possa fornecer informações sobre a interação do aluno com o objeto, como quais foram os resultados obtidos e os caminhos percorridos. Esse serviço é fornecido por alguns LMS, mas o objeto precisa implementar os metadados necessários.

### **2.4.3 Trabalho Relacionado: Ferramenta CARLOS**

Um dos trabalhos relacionados a este TCC é a ferramenta CARLOS (Collaborative Authoring of Reusable Learning ObjectS)(PADRON et al., 2003) que utiliza um sistema multi-agente para modelar a autoria colaborativa de objetos de aprendizagem, esta ferramenta foi desenvolvida por serem poucas as ferramentas de autoria de objetos de aprendizagem que lidem com a colaboração entre os autores. Este trabalho possui as seguintes características principais: 1) Os autores colaboradores são representados por agentes e delegam a eles a negociação e avaliação das propostas; 2) A troca de mensagens entre os agentes é coordenada através de um protocolo de negociação; 3) A racionalidade dos agentes para avaliar as propostas é modelada em termos de relações de preferência e relevância, descritas com lógica nebulosa; 4) Os resultados das negociações são automaticamente armazenados e consolidados no OA.

Esta ferramenta é semelhante a este trabalho por ser uma ferramenta de autoria de OAs modelada com um sistema multi-agente. As principais diferenças são que CARLOS foca em criar o OA como um todo, tanto o conteúdo quanto os metadados, e utiliza um SMA com um objetivo diferente, auxílio na colaboração entre autores, ao invés

de auxílio no preenchimento dos metadados de um OA por uma única pessoa, como neste trabalho.

#### **2.4.4 Requisitos da Ferramenta de Autoria deste trabalho**

De acordo com os resultados da pesquisa em (XAVIER; GLUZ, 2009) foram definidos alguns requisitos desejáveis para a ferramenta de autoria de metadados que será desenvolvida neste trabalho:

- Automatizar e facilitar o preenchimento de metadados, utilizando agentes e o modelo de Agentes & Artefatos.
- Representar os metadados e os OAs através de ontologias, permitindo que agentes possam utilizá-los e que seja possível realizar inferências sobre os OAs, ampliando sua utilização pelos agentes.
- Suportar o padrão de metadados brasileiro OBAA (Projeto Objetos de Aprendizagem baseados em Agentes) (BEZ et al., 2009) , que pretende utilizar as tecnologias de SMAs, OAs e computação ubíqua para possibilitar a autoria, armazenamento e recuperação de OAs em tempo hábil, em contextos diversos e através de diferentes plataformas de acesso.
- Fornecer um tutorial ou manual para a ferramenta, visto que isso não foi disponibilizado por nenhuma das ferramentas estudadas.

Também é desejável que a ferramenta seja Web, para facilitar seu acesso e pela ausência de instalações para o usuário, caso isso não seja possível, então a ferramenta será disponibilizada em um arquivo .jar. Outro requisito adicional relevante seria que a ferramenta suporte os populares padrões Scorm e LOM, para que possa ser mais amplamente utilizada.

## **2.5 TRABALHOS RELACIONADOS COM ACESSO DE AGENTES A WEB SEMÂNTICA**

Nesta seção serão descritos alguns trabalhos relacionados que utilizam agentes para acessar a Web Semântica. Em (KLAPISCAK; BORDINI, 2009) os autores desenvolveram a arquitetura de agentes JASDL (Jason AgentSpeak DescriptionLogic), uma extensão da plataforma Jason. O objetivo deste trabalho é fornecer funcionalidades de programação de agentes combinadas com raciocínio ontológico, como

generalização da ativação de planos e consultas a base de crenças baseadas em conhecimento ontológico. Para desenvolver esta arquitetura os autores utilizaram a OWL-API como o motor de inferência, que eles consideraram como mais elegante que a Jena API, além de fornecer funcionalidades de debug em caixa preta. (KLAPISCAK; BORDINI, 2009) consideram JASDL como a primeira implementação completa de uma linguagem de programação orientada a agentes que utiliza, de forma transparente, ontologias e raciocínio ontológico em um contexto declarativo.

No artigo (SILVA; VIEIRA, 2007) é proposto Argonaut, um protótipo que integra Jason com o framework Jena com o objetivo de suportar computação baseada em contexto através de ontologias OWL. Os autores desenvolveram um SMA para a área de computação móvel, onde os agentes auxiliam o usuário a encontrar serviços localmente situados. Argonaut é a primeira abordagem prática a integrar OWL, Jason e Jena (SILVA; VIEIRA, 2007). Neste modelo todas as interações com ontologias são definidas dentro de ações internas do Jason, ou seja, código Java estático. No caso da JASDL, a integração entre agentes e a Web Semântica ocorre de forma mais transparente, ampliando a base de crenças do Jason com a capacidade de raciocínio ontológico.

Em (KAGAL et al., 2003) os autores descrevem ITTALKS, um sistema Web que oferece acesso a informações sobre palestras, seminários e colóquios da área de tecnologia da informação através de notificações automáticas e inteligentes. Para isso, os autores desenvolveram um SMA capaz de acessar a Web Semântica utilizando a linguagem DAML (DARPA Agent Markup Language). A DAML é utilizada para definir as ontologias utilizadas pelo sistema, para a representação da base de conhecimento, raciocínio, e na comunicação entre agentes. Por utilizarem DAML, os agentes do ITTALKS são capazes de inferir informações adicionais sobre as palestras, como por exemplo encontrar palestras compatíveis com os interesses, rotina e localização do usuário, considerando inclusive o trânsito no momento.

Outros trabalhos relacionados relevantes são: (ZOU et al., 2003) utiliza RDF e OWL como linguagens de descrição de conteúdo e serviços no contexto do framework TAGA (Travel Agent Game in Agentcities); (CHEN et al., 2004) desenvolveram o EasyMeeting, um sistema ubíquo e inteligente para salas de reunião, que utiliza agentes e ontologias OWL para fornecer serviços e informações baseadas no contexto dos participantes da reunião; e em (KATASONOV; TERZIYAN, 2008) é implementado S-APL (Semantic Agent Programming Language), uma linguagem baseada em RDF para a Web Semântica que integra a des-

crição semântica dos recursos do domínio com a prescrição semântica do comportamento dos agentes.

A principal diferença entre o projeto desenvolvido neste trabalho e os trabalhos relacionados é que nesta proposta a integração entre os agentes e a Web Semântica é feita através de artefatos. Os artefatos são utilizados como um middleware onde as interações ontológicas são processadas, e como uma ferramenta para coordenar os agentes. Algumas das vantagens de se utilizar artefatos para modelar ferramentas e recursos foram descritas na Seção 2.2.3.



### 3 DESENVOLVIMENTO

#### 3.1 VISÃO GERAL

Neste trabalho foi desenvolvida uma ferramenta para o preenchimento de metadados de objetos de aprendizagem. Esta ferramenta auxilia no preenchimento dos metadados através recomendações. Na Figura 11 é mostrada uma visão geral do sistema, onde o usuário preenche os metadados já conhecidos, e então solicita ao sistema que lhe forneça recomendações para preencher um dos campos de metadados. O conhecimento parcial já fornecido pelo usuário é então enviado para o SMA, no qual os agentes utilizam os artefatos para processar consultas em SPARQL na DBPedia, de forma a obter as recomendações. Essas recomendações são então retornadas para a GUI e disponibilizadas para o usuário. É utilizada a linguagem Jason para modelar os agentes e o framework Cartago para programar os artefatos.

Na Figura 12 é mostrada uma visão geral do SMA desenvolvido neste trabalho. Existem três tipos de artefatos especializados em buscas semânticas, no caso, busca por indivíduos, por classes, e por categorias. Onde indivíduos e classes são os conceitos estruturais típicos de uma ontologia, e categorias é um conceito com função informativa, derivado no mapeamento da Wikipedia para a DBPedia. Além disso, também existem sub-buscas, busca abrangente e busca específica, na qual a primeira busca uma grande quantidade de resultados, mas de baixa qualidade, e a segunda busca poucos resultados de alta qualidade. Essas duas sub-buscas são disponibilizadas como operações no artefato, que os agentes podem acessar fornecendo como parâmetros a palavra-chave que será utilizada na busca e o limite da busca em largura. O resultado retornado aos agentes são os indivíduos obtidos pela busca. No caso da busca por classes e categorias, são retornados os indivíduos que pertencem as classes e categorias resultantes.

Cada agente é especializado em utilizar um destes três tipos de artefatos, havendo um mapeamento de 1 para 1 entre os agentes e os artefatos, pois o modelo de A&A não permite que um artefato seja utilizado ao mesmo tempo por mais de um agente. No total existem nove agentes e nove artefatos de busca semântica. Além destes artefatos a Figura 12 também representa os artefatos ArtefatoEntrada e ArtefatoSaida, que funcionam como uma integração entre a GUI e o SMA da ferramenta. Eles também servem para coordenar os agentes, onde o ArtefatoEntrada processa o conhecimento parcial em palavras-chaves

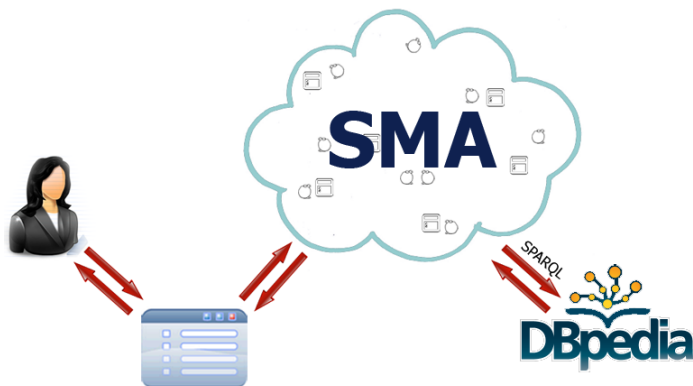


Figura 11 – Visão geral da ferramenta.

e as fornece aos agentes, e o ArtefatoSaida recebe os resultados das buscas semânticas dos agentes e os processa em recomendações.

### 3.2 FERRAMENTA DE AUTORIA DE METADADOS

Na versão atual os campos para os quais podem ser geradas recomendações são Title, Description e Keywords da categoria General do Perfil Compatível Mínimo com OBAA (PM-OBAA-CORE). O perfil PM-OBAA-CORE foi escolhido por ser o perfil básico para um OA se adequar ao padrão OBAA e por ser simples o suficiente para permitir focar em poucos atributos. Este perfil possui quatro categorias de metadados: General, LifeCycle, Technical e Rights. Entre estas categorias somente os metadados Title, Description e Keywords foram escolhidos para gerar sugestões pois são os que possuem uma relação mais direta com as propriedades utilizadas pela DBpedia, como title, abstract e name. A GUI para cada uma das categorias é mostrada nas figuras abaixo (Figuras 13 e 14), também é mostrada uma aba adicional (Figura 15), que fornece outra forma de ajuda ao usuário explicando o perfil e cada categoria. Ao passar o mouse pelos símbolos de interrogação o usuário pode ver a descrição do metadado e um exemplo de como utilizá-lo, de forma a ajudar o preenchimento dos metadados mesmo quando não podem ser geradas sugestões para os campos.

Conforme apresentado na Figura 13, os campos para os quais se pode gerar sugestões possuem o botão “Help me!” ao lado. Quando o

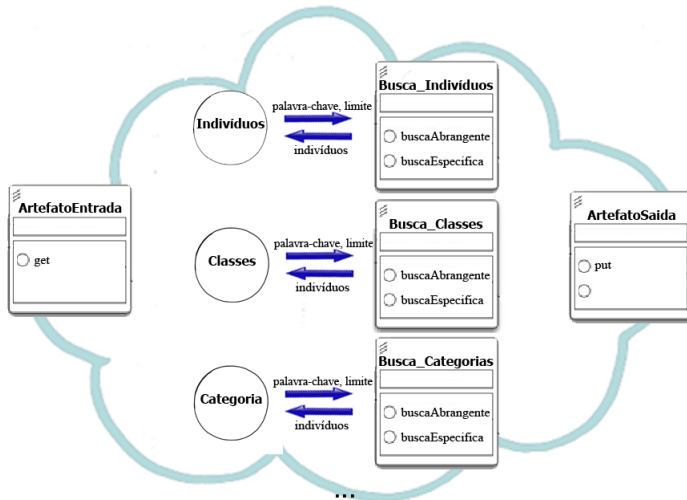


Figura 12 – Visão geral do sistema multi-agente.

Figura 13 – Screenshot da categoria General.

The figure consists of two side-by-side screenshots of a software interface titled 'Current Metadata Profile: Perfil Compatível Mínimo com OBAA'. Both windows have a menu bar with 'File', 'Profiles', and 'About'. The left window has tabs for '1. General', '2. LifeCycle', '4. Technical', and '6. Rights', with '2. LifeCycle' selected. It contains the following fields: '1. Version:' with a dropdown set to 'en'; '2. Status:' with a dropdown set to 'final'; '3. Contributors:' with a '+' button; '3.1. Role:' with a help icon and a tooltip that reads 'Those entities (i.e., people, organizations) that have contributed to the state of this learning object during its life cycle (e.g., creation, edits, publication)'; '3.2. Entity:' with sub-fields for 'First Name', 'Middle Name', and 'Last Name'; 'E-mail:'; 'Organization:'; and '3.3. Date:' with dropdowns for day (1), month (jan), and year (2013). The right window has the same tabs, but '4. Technical' is selected. It contains: '1. Format:' with a help icon and a tooltip that reads 'MIME type(s) of (all the components of) this learning object. Ex: application/pdf, image/jpeg, text/html. Lista de MIME types: http://www.iana.org/assignments/media-types. Caso haja mais de uma informação separe os valores com ;'; '1. Cost:' with a dropdown set to 'no'; '2. Copyright and Other Restrictions:' with a dropdown set to 'no'; and '3. Description:' with a text area and a language dropdown set to 'en'.

Figura 14 – Screenshots das categoria LifeCycle, Technical e Rights.

The figure is a screenshot of the 'Current Metadata Profile' window with the '6. Rights' tab selected. The window title is 'Current Metadata Profile: Perfil Compatível Mínimo com OBAA (PM-OBAA-CO'. The tabs are '1. General', '2. LifeCycle', '4. Technical', '6. Rights', and a help icon '?'. The content area includes: a 'Legenda:' section with '- campo obrigatório' and '- campo opcional'; a section titled 'Perfil Compatível Mínimo com OBAA (PM-OBAA-CORE)' with a text box explaining that the profile consists of essential minimum metadata for learning objects according to the OBAA standard, aiming for integration with legacy OAs with minimal human effort; a 'Referência:' section with the text 'Relatório Técnico RT-OBAA-01'; and a 'Categorias:' section listing '1. General', '2. LifeCycle', '4. Technical', and '6. Rights', each with a help icon.

Figura 15 – Screenshot da aba de ajuda.

usuário clica neste botão requisitando ajuda a GUI coleta os valores já preenchidos pelo usuário para os campos Title, Description e Keywords e os envia para a classe Controladora, responsável por integrar a GUI aos artefatos. Na Figura 14 são mostradas screenshots das abas que contém as categorias LifeCycle, Technical e Rights com seus respectivos campos de metadados. Nota-se que em duas das screenshots há um texto em um balão, este texto é mostrado cada vez que o usuário passa o mouse por cima do símbolo de interrogação de um metadado, auxiliando-o com informações extras sobre o preenchimento do campo do metadado. Na Figura 15 é mostrada a aba de ajuda, que descreve o perfil de metadados sendo utilizado e dá mais informações sobre as suas categorias.

Após receber os valores da GUI a classe Controladora os processa obtendo as palavras-chaves de cada campo. Os valores obtidos nos campos Title e Description são transformados em palavras-chave utilizando técnicas de NLP (Processamento de Linguagem Natural), esse processamento consiste em utilizar a biblioteca Apache Lucine para remover stopwords (tais como he, she, my, does, was), converter as palavras no plural para o singular, armazenando-as em um conjunto de strings. Para o campo Keywords as palavras-chaves são obtidas simplesmente separando-as pelo símbolo ;.

O conjunto de palavras-chaves é então agrupado em um único conjunto e enviado para um algoritmo de ranqueamento, que seleciona até oito palavras-chaves que apareceram com mais frequência no conjunto. Isso é feito pois se considera que se uma mesma palavra-chave é comum a vários campos ou é repetida várias vezes em uma Description então ela deve ser mais importante do que as que aparecem com menor frequência. O número de palavras-chaves obtidas foi definido como oito pois se considerou que apenas as palavras-chaves mais importantes deveriam ser usadas na busca semântica, de forma a reduzir o tempo de processamento e a quantidade de resultados irrelevantes para o usuário.

Ao completar o pré-processamento dos valores já preenchidos na GUI a classe Controladora envia as 8 palavras-chave selecionadas para o artefato ArtefatoEntrada, juntamente com o campo para o qual as sugestões devem ser geradas. As palavras-chaves ficam então disponíveis no ArtefatoEntrada, que é checado a cada meio segundo pelos agentes. Quando novas palavras-chaves estão disponíveis cada uma é obtida por um agente de um tipo de abordagem de busca semântica, de forma que possam ser processadas paralelamente.

Quando obtém uma palavra-chave o agente a processa utilizando um dos artefatos especializados em uma das abordagens de acesso a DB-

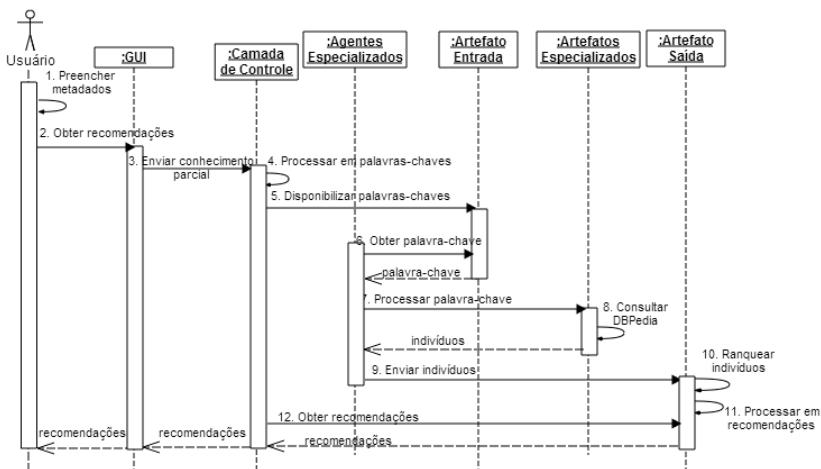


Figura 16 – Diagrama de fluxo da ferramenta de autoria.

Pedia. Existe um artefato para cada um dos três tipos de abordagens: por indivíduo, por classe e por categoria. Após processar os dados os agentes retornam os indivíduos obtidos ao artefato ArtefatoSaida, que os usa para obter os valores de propriedades equivalentes ao campo para o qual foram pedidas sugestões. Então o artefato retorna a GUI que mostra ao usuário as sugestões solicitadas para o campo. O diagrama para o fluxo de atividades descrito nessa seção é mostrado na Figura 16. O processamento das palavras-chaves pelos agentes, artefatos especializados e pelo ArtefatoSaida é descrito em mais detalhes na seção a seguir.

### 3.3 MODELO DE SUGESTÕES

#### 3.3.1 Abordagens de Busca Semântica

Para realizar a busca semântica de componentes da ontologia semelhantes às palavras-chaves são utilizadas três tipos de abordagem de busca semântica implementadas por agentes e artefatos: por indivíduos, por classe e por categorias. A busca por indivíduos obtém resultados filtrando diretamente os indivíduos da ontologia, enquanto as por classe e por categorias filtram um grupo de indivíduos (no caso, classes e cate-

gorias) e obtém resultados como sendo indivíduos que pertencem a estes grupos. As buscas são feitas pelos artefatos no repositório semântico DBPedia com consultas SPARQL no ponto de acesso Virtuoso, estas consultas são realizadas utilizando a API Jena. Existe um artefato especializado para cada tipo de abordagem, eles fornecem duas formas de utilizar a abordagem: através de uma busca abrangente e de uma busca específica. Essas buscas são um trade-off entre quantidade de resultados e velocidade de processamento, na busca abrangente, por qualidade de resultados, na busca específica. O agente inicialmente utiliza a busca abrangente da abordagem no qual é especializado, e dependendo dos resultados utiliza a busca específica para refiná-los.

### 3.3.2 SMA

No sistema multi-agente deste trabalho existem três tipos de agentes especializados nas abordagens (`atuador_por_individuos`, `atuador_por_classes` e `atuador_por_categorias`) e um único agente inicializador, responsável por criar o artefatos que os outros agentes utilizarão. Existem três agentes para cada uma das três abordagens possíveis, então no total o SMA é composto por 10 agentes.

Segue abaixo o arquivo `.mas2j` que configura o sistema:

---

```

1 MAS example01_useobs {
3   environment: c4jason.CartagoEnvironment

5   agents:
      inicializador agentArchClass c4jason.CAgentArch #1;
7   atuador_por_individuos agentArchClass c4jason.CAgentArch #3;
      atuador_por_classes agentArchClass c4jason.CAgentArch #3;
9   atuador_por_categorias agentArchClass c4jason.CAgentArch #3;

11  classpath: ‘../lib/cartago.jar’; ‘../lib/c4jason.jar’; ‘../
      lib/lib-jena/*.jar’; ‘../lib/lucene-3.6.2/*.jar’; ‘../
      lib/lucene-3.6.2/contrib/analyzers/common/*.jar’;

13  aslSourcePath:
      ‘agents’;

```

---

O agente inicializador cria um ArtefatoEntrada, um ArtefatoSaida e três de cada tipo de artefato específico para as abordagens de busca. Os artefatos das abordagens recebem os mesmos nomes dos agentes com números de 1 a 3, desta forma os agentes sabem qual ar-

tefato devem usar apenas sabendo o seu próprio nome. Foi necessário criar várias instâncias de cada artefato pois o modelo Cartago possui a propriedade de atomicidade das operações, de forma que mais de um agente não pode utilizar uma mesma operação ao mesmo tempo.

---

```

!create_and_use.

2
+!create_and_use : true
4
  <- !setupTools;
    println("artefatos criados").

6
// create the tools
8
+!setupTools: true
  <- makeArtifact("entrada", "artifacts.ArtefatoEntrada"
    ,[], A1);
10
    makeArtifact("saida", "artifacts.ArtefatoSaida", [], A2)
      ;
    makeArtifact("atuador_por_individuos1", "artifacts.
      AbordagemPorIndividuo", [], A3);
12
    makeArtifact("atuador_por_individuos2", "artifacts.
      AbordagemPorIndividuo", [], A6);
    makeArtifact("atuador_por_individuos3", "artifacts.
      AbordagemPorIndividuo", [], A9);
14
    makeArtifact("atuador_por_classes1", "artifacts.
      AbordagemPorClasses", [], A4);
    makeArtifact("atuador_por_classes2", "artifacts.
      AbordagemPorClasses", [], A7);
16
    makeArtifact("atuador_por_classes3", "artifacts.
      AbordagemPorClasses", [], A10);
    makeArtifact("atuador_por_categorias1", "artifacts.
      AbordagemPorCategoria", [], A5);
18
    makeArtifact("atuador_por_categorias2", "artifacts.
      AbordagemPorCategoria", [], A8);
    makeArtifact("atuador_por_categorias3", "artifacts.
      AbordagemPorCategoria", [], A11).

```

---

### 3.3.3 Prefixos

Em SPARQL são definidos prefixos para indicar de forma simplificada que ontologias podem ser utilizadas pela busca semântica. Os prefixos utilizados neste trabalho são:

---

```

1 PREFIX db-prop: <http://dbpedia.org/property/>
  PREFIX dbpedia-owl:<http://dbpedia.org/ontology/>
3 PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
  PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>

```



```

5 PREFIX dbpedia_category: <http://dbpedia.org/resource/
  Category:>
  PREFIX dcterms: <http://purl.org/dc/terms/>
7 PREFIX foaf: <http://xmlns.com/foaf/0.1/>
  PREFIX yago:<http://dbpedia.org/class/yago/>
9 PREFIX skos: <http://www.w3.org/2004/02/skos/core#>

```

---

Este conjunto de prefixos será representado pela chamada ao método `getPrefixes()` nas buscas com SPARQL.

### 3.3.4 Filtros

Filtros são utilizados para refinar uma busca semântica com SPARQL de forma a aumentar a qualidade dos resultados obtidos. Foram definidos quatro tipos de filtros: classes simples, classes avançado, categorias simples e categorias avançado. Filtrar por classe ou por categoria significa que indivíduos que sejam direta ou indiretamente subclasses de classes filtradas ou pertençam a categorias filtradas não serão retornados pela busca. Os subtipos simples e avançado indicam o quão restritiva é a filtragem, com avançado considerando mais parâmetros e tendo um processamento mais lento.

O critério para definir quais classes e categorias seriam filtradas foi baseado na funcionalidade desta ferramenta: como ela é usada para preencher metadados apenas de OAs, então são filtradas conceitos que tem pouca relação com ensino e aprendizagem, como literatura de periódicos, shows de televisão, bandas, músicas e times de futebol. Outro critério para definição da filtragem é que foi percebido durante os testes da ferramenta que alguns conceitos que tinham grandes chances de trazerem resultados inúteis para o contexto da ferramenta, como pessoas e profissões, então estes também foram adicionados à filtragem. Em uma versão futura da ferramenta poderia ser habilitada a customização de filtros, de forma que o usuário possa escolher receber sugestões que incluam músicas, por exemplo, caso o OA em questão seja sobre música. A lista completa dos filtros utilizados neste trabalho encontra-se no Apêndice .

### 3.3.5 Abordagem por Indivíduo

Na abordagem por indivíduo o objetivo é obter os indivíduos da ontologia que sejam mais semelhantes a palavra-chave recebida.

Para isso são procurados na DBPedia indivíduos cujas propriedades `db-prop:title` ou `db-prop:name` que contenham a palavra-chave. Na abordagem por indivíduos os filtros de classe simples e avançado são utilizados na busca abrangente, e os filtros de categoria simples e avançado são utilizados para refinar os resultados na busca específica. No momento os agentes sempre efetuam uma busca abrangente e logo em seguida uma busca específica, para aumentar a qualidade dos resultados. Caso a busca específica sobre os resultados da busca abrangente zere os resultados obtidos, o agente retorna para o `ArtefatoSaida` os resultados da busca abrangente, ao invés dos da busca específica.

O código do agente `atuador_por_indivíduo` que efetua este raciocínio segue abaixo. Primeiro o agente obtém os artefatos `ArtefatoEntrada`, `ArtefatoSaida` e `AbordagemPorIndividuo`. Então com a intenção `!consumeItems` ele passa a checar o `ArtefatoEntrada` a cada 1 segundo, com a operação `get_para_Individuos(Item)`. Ao receber uma palavra-chave para processar, o agente a envia junto com um limite de resultados para a operação `executar_abrangente` do artefato `AbordagemPorIndividuo`. Após receber os indivíduos resultantes, os envia para a operação `executar_especifico`. Na etapa final o agente checa se ao executar a operação `executar_especifico` os resultados foram zerados. Se sim, os resultados da busca abrangente são enviados para o `ArtefatoSaida`, se não os resultados enviados são os da busca específica. O agente utiliza duas operações `put` neste envio, a primeira envia os resultados para um aglomerado geral de resultados de todos os agentes, e o `put_more_value` envia estes resultados para uma lista específica, utilizada para atribuir mais valor a resultados da abordagem por indivíduos. Este algoritmo de ranqueamento será explicado em mais detalhes na Seção 3.3.8. Artefato de saída.

---

```

1 !observe.

3 +!observe : true
  <- .wait(200);
5   ?myTools(A1, A2, Id); // discover the tools
   println("agente criado");
7   !consumeItems(Id).

9 +!consumeItems(Id): true
  <- .wait(1000);
11 get_para_Individuos(Item);
   !processItem(Item, Id);
13 !!consumeItems(Id).

15 +!processItem(null, Id) : true.

```

```

17 +!processItem(Keyword, Id) : true
    <- executar_abrangente(Keyword, 50, Result)[artifact_id(Id)
    ];
19     executar_especifico(Result, Result2)[artifact_id(Id)];
    isEmpty(Result2, Test)[artifact_id("entrada")];
21     !decidir(Result1, Result2, Test, Id).

23 +!decidir(Result1, Result2, Test, Id) : Test
    <- .my_name(Me); println(Me, ': vazio');
25     put_more_value(Result2);
    put(Result2).

27
29 +!decidir(Result1, Result2, Test, Id) : true
    <- .my_name(Me); println(Me, ': não vazio');
    put_more_value(Result2);
31     put(Result2).

33 +?myTools(A1, A2, A3): true
    <- lookupArtifact("entrada",A1);
35     lookupArtifact("saida",A2);
    .my_name(N);
37     lookupArtifact(N,A3).

39 -?myTools(A1, A2, A3): true
    <- .wait(100);
41     ?myTool(A1, A2, A3).

```

---

Abaixo segue o código da operação executar\_abrangente do artefato AbordagemPorIndividuo que executa a busca semântica abrangente utilizando a API Jena. Primeiro é montada uma string com a busca SPARQL para obter indivíduos cujos db-prop:title ou db-prop:name contenham a palavra-chave. Os métodos filtrarClasses(classes\_filtradas e filtrarClasses(classes\_filtradas\_avancado) retornam uma string com um filtro de classes, o código deste método está logo abaixo do código da operação. Com a query montada a API Jena é utilizada para se conectar ao ponto de acesso da DBPedia, processar a query, e extrair os resultados. Os resultados são então retornados para o agente através do parâmetro OpFeedbackParam;Object; result.

---

```

1 @OPERATION
    void executar_abrangente(String keywordrecebida, int
        limit, OpFeedbackParam<Object> result)
2     {
3         String keyword = NLPTool.toLowerCase(keywordrecebida
4             );
5
6         String queryString = getPrefixes()
7             + "SELECT distinct ?object "

```

```

9      + "WHERE {"
10     + "{?object db-prop:title ?title . ?title <
      bif:contains> \"'+keyword+'\" . }"
11     + " UNION "
12     + "{?object db-prop:name ?name . ?name <bif
      :contains> \"'+keyword+'\" . }"
13     + filtrarClasses(classes_filtradas_simples)
14     + filtrarClasses(classes_filtradas_avancado)
15     + "}"
16     + "LIMIT "+limit;

17     QueryExecution qe = new QueryEngineHTTP("http://
      dbpedia.org/sparql", queryString);

19     ArrayList<String> individuos = new ArrayList<String>
      >();
20     try
21     {
22         ResultSet results = qe.execSelect();
23         for (; results.hasNext(); )
24         {
25             QuerySolution s = results.next();
26             individuos.add(s.getResource("object").
27                 toString());
28         }
29     } finally
30     {
31         qe.close();
32     }

33     result.set(individuos);
34     System.out.println("LOG: Abordagem por Individuos
      ENDED: "+individuos.size());
35 }

```

---

```

private String filtrarClasses(ArrayList<String> classes)
2 {
3     String result = "";
4     for(String s: classes)
5         result += "filter (NOT EXISTS {?object a '"+s+"'
6         }) .";
7     return result;
8 }

```

---

O código da operação para a busca específica é igual ao da busca abrangente, com a diferença que a filtragem ocorre chamando o método

filtrarCategorias(categorias\_filtradas, 4) e filtrarCategorias(categorias\_filtradas.av  
3), e a busca não é mais filtrada para classes. A filtragem por categoria ocorre na busca específica por ser muito mais lenta que a filtragem por classes.

O código para filtrarCategorias é mostrado abaixo. Para este filtro foi necessário usar a option TRANSITIVE para limitar a profundidade da busca, pois um indivíduo pode pertencer a um grande número de categorias de forma indireta. O parâmetro T\_MAX define o valor da profundidade.

---

```

1 private String filtrarCategorias ( ArrayList<String>
   categorias , int t_max)
   {
3       String option = “option(TRANSITIVE , T_DISTINCT,
         t_max(“+t_max+”))”;
       String result = “”;
5       for (String s: categorias)
           result += “filter (NOT EXISTS {?object determs:
               subject ?Category . ?Category skos:broader
               dbpedia.category:‘+s+’ “+option+” }) ..”;
7       return result;
   }

```

---

### 3.3.6 Abordagem por Classes

Nesta abordagem são selecionadas as classes que possuam a propriedade `rdfs:label` que contenha a palavra-chave recebida. Na busca abrangente é feita uma seleção de classes com o filtro de classes simples. Após todas as classes tenham sido obtidas ou o número máximo de classes selecionadas tiver sido atingido (`limite1`) elas são retornadas para o agente. O agente pode então enviar estas classes para a busca específica, que as refina usando o filtro de classes avançado. Antes de enviar os resultados para o `ArtefatoSaida` o agente processa as classes para obter todos os indivíduos que são subclasses delas, até um limite de indivíduos determinado pelo agente (`limite2`). Esta abordagem produz mais resultados que a abordagem por indivíduos, pois o máximo de indivíduos retornados é a multiplicação `limite1*limite2`.

O código do agente `atuador_por_classes` é semelhante ao do agente para indivíduos e se encontra no Apêndice C.2.1. As principais dife-

renças entre os dois são sendo: ela utiliza o get específico para os agentes da abordagem por classes; possui valores de limite menores, por eles multiplicarem o resultado final; a operação “executar.getIndivíduos” ser chamada para obter os indivíduos das classes antes de retornar os resultados para o ArtefatoSaida; e a operação put\_less\_value ser chamada ao invés da put\_more\_value, pois percebeu-se nos testes da ferramenta que esta abordagem costuma gerar resultados bastante imprecisos. A imprecisão dos resultados ocorre pois a filtragem não é realizada diretamente sobre os indivíduos obtidos, estes indivíduos são obtidos se possuírem uma relação de subclasse de uma distância de até quatro classes, e porque classes não pertencem a categorias, então os filtros para categorias não puderam ser aplicados.

A operação para executar uma busca abrangente e método utilizado para fazer a filtragem por classe no artefato AbordagemPorClasses são apresentados no Apêndice C.2.2. A operação faz uma busca SPARQL onde se procura um valor Concept que seja uma classe e cujo título contenha a palavra-chave. Também é feita uma filtragem por classe simples com profundidade igual a 5. As classes obtidas são então retornadas para o agente. A busca específica segue o mesmo padrão da busca abrangente, porém o filtro utilizado é o para classes avançadas com profundidade 5.

O método para obter os indivíduos que são subclasses de uma classe segue abaixo. Este método é chamado várias vezes para cada classe obtida previamente. A profundidade de busca utilizada é 4.

---

```

1 public ArrayList<String> getAllSubClasses(String classe, int
    limit)
2 {
    String queryString = AbordagemPorIndividuo.
        getPrefixes()
3         + “select distinct ?subclass where {”
4         + “ ?subclass a <“+classe+”> option(
        TRANSITIVE, t_distinct, TMAX(4)).”
5         + “} LIMIT ”+limit;
6
7     QueryExecution qe = new QueryEngineHTTP(“http://
        dbpedia.org/sparql”, queryString);
8
9     ArrayList<String> subclasses = new ArrayList<String>
        >();
10    try
11    {
12        ResultSet results = qe.execSelect();
13        for (; results.hasNext(); )
14        {
15            QuerySolution s = results.next();

```

```

        subclasses.add(s.getResource("subclass").
            toString());
18     }
19     }
20     finally
21     {
22         ge.close();
23     }
24     return subclasses;
}

```

---

### 3.3.7 Abordagem por Categorias

Esta abordagem é análoga a abordagem por classes, porém são utilizadas categorias, um conceito específico da DBPedia. Cada indivíduo pode estar em várias categorias, e existe uma hierarquia de categorias, com algumas sendo mais amplas e outras mais específicas. Um indivíduo também pode participar de uma categoria indiretamente, por exemplo o indivíduo “[http://dbpedia.org/ontology/American\\_bison](http://dbpedia.org/ontology/American_bison)” pertence a categoria “<http://dbpedia.org/resource/Category:Bison>”, cuja categoria mais ampla é “<http://dbpedia.org/resource/Category:Bovines>”, então *American\_bison* também participa da categoria *Bovines*. Na busca abrangente é utilizado o filtro para categorias simples, e na busca específica é utilizado o filtro avançado. Por haverem muitas relações entre categorias estouros de memória são comuns durante a filtragem, para se adequar a isso esta abordagem começa com um valor de profundidade inicial igual a 6, e vai reduzindo-o em 2 reiniciando a busca cada vez que ocorre uma exceção de estouro de memória. A operação termina quando o artefato consegue processar a busca SPARQL sem um estouro de memória ou quando o valor da profundidade zera, retornando nenhuma categoria.

O código do agente *atuador\_por\_categorias* está disponível no Apêndice . Ele é análogo ao do *atuador por classes*, porém utiliza o *get* específico para agentes que utilizam a abordagem por categorias e ele chama apenas um método *put*, pois recebe uma pontuação padrão no ranqueamento, tendo resultados com qualidade intermediária em relação as outras abordagens.

O código da operação de busca abrangente é está disponível no Apêndice C.3.2. Ele é composto por três partes, a primeira administra as exceções de estouro de memória e o decréscimo do valor da pro-

fundidade como descrito acima. O código para busca específica é igual, apenas chamando o método `executar_especifico2(categorias, t_max)`; ao invés de `executar_abrangente2(keyphrase, t_max, limit)`. A segunda parte é o código do método que efetua a busca SPARQL. São pesquisados conceitos que sejam categorias e cuja propriedade `rdfs:label` contenha a palavra-chave. Na busca abrangente utiliza-se o filtro para categorias simples, e na específica o filtro avançado. Por fim, a terceira parte do código é o método para obter os indivíduos que pertencem a uma categoria. Este método é chamado várias vezes para cada categoria obtida previamente. Não há limite para a profundidade da busca, o único limite é a largura da busca definida pelo agente.

### 3.3.8 Processamento dos resultados

Quando os agentes terminam de processar a palavra-chave recebida para obter indivíduos, eles os retornam para o artefato `ArtefatoSaida`. Todos os agentes utilizam a operação `put` para retornar os indivíduos, essa operação adiciona os indivíduos a uma lista e aumenta um contador cada vez que é chamada. Quando esse contador chegar em  $(\text{quantidade de palavras-chaves}) \times (\text{quantidade de agentes por tipo})$ , ou seja, todos os agentes que obtiveram palavras-chave retornaram os resultados, a operação chama um algoritmo de ranqueamento que retorna os 20 melhores resultados e altera o atributo “pronto” para `true`. Este atributo é checado constantemente por uma `Thread` da classe `Controladora` para saber quando deve enviar os resultados para a GUI. Adicionalmente, os agentes que atuam por indivíduo também chamam a operação `put_more_value` e os por classe chamam a `put_less_value` para adicionar os indivíduos retornados em listas específicas que afetam a pontuação que esses indivíduos receberão no ranqueamento. Abaixo segue o código das operações `put`, `put_more_value` e `put_less_value`.

---

```

1 @OPERATION
   void put(Object obj)
3   {
       ArrayList<String> array = (ArrayList<String>)obj;
5       items.addAll(array);

7       cont++;
       int quantAgentsPorTipo = 3;
9       if(cont == Controladora.quantKeywords*
           quantAgentsPorTipo)
       {
11          resultados_por_quant = Rank.rankByQuant(20,
              items, items_less_value, items_more_value);

```



```

        System.out.println(“‘rankeado: ”+
                           resultados_por_quant);
13     pronto = true;
14     }
15 }

16 @OPERATION
17 void put_less_value(Object obj)
18 {
19     ArrayList<String> array = (ArrayList<String>)obj;
20     items_less_value.addAll(array);
21 }
22
23 @OPERATION
24 void put_more_value(Object obj)
25 {
26     ArrayList<String> array = (ArrayList<String>)obj;
27     items_more_value.addAll(array);
28 }
29

```

---

O algoritmo de ranqueamento efetuado pela classe Rank organiza os indivíduos em uma lista sem repetições atribuindo 0 pontos a cada um. Então o algoritmo percorre novamente a lista, aumentando a pontuação do indivíduo cada vez que ele é encontrado, pois se considera que se vários agentes utilizando diferentes abordagens encontraram um mesmo indivíduo, então ele deve ter mais importância que os outros. Indivíduos obtidos pela abordagem por indivíduos recebem 1.3 pontos, pela por categorias 1 ponto e pela por classes 0.8 pontos.

As abordagens valem diferentes quantidades de pontos pois a qualidade dos seus resultados varia bastante. Como a abordagem por indivíduos filtra diretamente os resultados finais, tem resultados melhores e vale mais pontos, mas por ser bastante específica na busca pode gerar poucos resultados. No caso das abordagens por categoria e por classe, ambas possuem o defeito de gerar resultados piores pois o filtro não ocorre diretamente nos indivíduos obtidos, elas também podem gerar uma grande quantidade de resultados finais. Porém nos testes com a ferramenta foi percebido que os resultados da abordagem por categoria ainda são melhores que a por classe, acredita-se que isso ocorra por Category ser um conceito próprio da DBPedia, então está melhor modelada no repositório semântico. O código utilizado para o algoritmo de ranqueamento é mostrado abaixo, o método `sortValue` usado pelo algoritmo serve para ordenar a lista em ordem decrescente de pontuação.

---

```

1 public static List<String> rankByQuant(int quant, ArrayList<
    String> individuos, ArrayList<String> less_value,
    ArrayList<String> more_value)
    {
3         if(individuos.isEmpty())
            return new ArrayList<String>();
5         Collections.sort(individuos);
        Hashtable<String, Double> ranked = new Hashtable<
            String, Double>();
7         double points;
        for(int i = 0; i < individuos.size(); i++)
9         {
            ranked.put(individuos.get(i), 0.0);
11        }
        for(int i = 0; i < individuos.size(); i++)
13        {
            points = 1;
15            if(less_value.contains(individuos.get(i)))
                {
17                points = 0.8;
                less_value.remove(individuos.get(i));
19            } else if(more_value.contains(individuos.get(i)))
                {
21                points = 1.3;
                more_value.remove(individuos.get(i));
23            }
            ranked.put(individuos.get(i), ranked.get(
                individuos.get(i)) + points);
25        }

27        ArrayList<String> s_ranked = sortValue(ranked, false
    );
        if(quant > s_ranked.size())
29            quant = s_ranked.size();
        return s_ranked.subList(0, quant);
31    }

```

---

Quando a Thread da classe Controladora percebe que os resultados terminaram de ser ranqueados através do atributo “pronto” ela chama o método `getItemsDevolvidos`. Este método obtém valores de propriedades específicas dos indivíduos ranqueados, dependendo do campo para qual foi pedida a sugestão de metadado. Para o campo “Description” são buscados os valores das propriedades `dbpedia-owl:abstract` e `rdfs:comment` de cada indivíduo. Para os campos “Title” e “Keyword” são buscados os valores das propriedades `db-prop:title`, `db-prop:name`, `foaf:name` e `rdfs:label`. Caso mais de um valor para uma propriedade seja encontrado apenas o primeiro da lista é retornado, também é retornado apenas um valor no total, com a ordem de priori-

dade sendo a ordem em que as propriedades apareceram listadas.

Optou-se utilizar várias propriedades por campo porque não existia uma propriedade que todos os indivíduos possuísem garantidamente. Também optou-se por obter as mesmas propriedades para ambos os campos “Title” e “Keyword” porque não foi encontrada uma propriedade específica para palavras-chaves na DBPedia, mas percebeu-se que as quatro propriedades citadas geralmente incluíam poucas palavras, então funcionavam também como palavras-chave. A busca SPARQL utilizada para obter essas propriedades é mostrada abaixo, onde “indivíduo” é um indivíduo da lista de indivíduos ranqueados e “prop” é a propriedade que se quer obter.

---

```

1 ‘SELECT distinct ?title ”
      + ‘WHERE ”
3      + ‘{<‘+indivíduo+’> ‘+prop+’ ?title .
      FILTER ( lang(?title) = \‘en\’) .}”
      + ‘LIMIT 5” ;

```

---

Após obter os valores específicos por campo, a Thread os concatena em uma String e envia como texto para a JTextArea no JDialog que foi mostrado para o usuário ao clicar em “Help me”. São mostrados tanto a URI do indivíduo da onde foi obtida a sugestão como o valor da propriedade buscada. Na Figura 17 o usuário digitou “bison” no campo Keywords e clicou no botão “Help me!” para o campo Title.

### 3.4 ANÁLISE DOS RESULTADOS DAS ABORDAGENS

Para analisar a qualidade das recomendações feitas pela ferramenta desenvolvida foram definidos alguns critérios. Estes critérios são subjetivos e nem sempre é óbvio se uma sugestão cumpre ou não um critério. O objetivo desta definição é obter uma ideia geral da qualidade das sugestões obtidas, considerando o contexto de um OA utilizado para ensino e aprendizagem.

Critérios:

- A sugestão é boa se for baseada em um indivíduo que...
  - For um subtipo da palavra-chave.
  - Se possuir uma relação direta de causa-efeito com a palavra-chave.

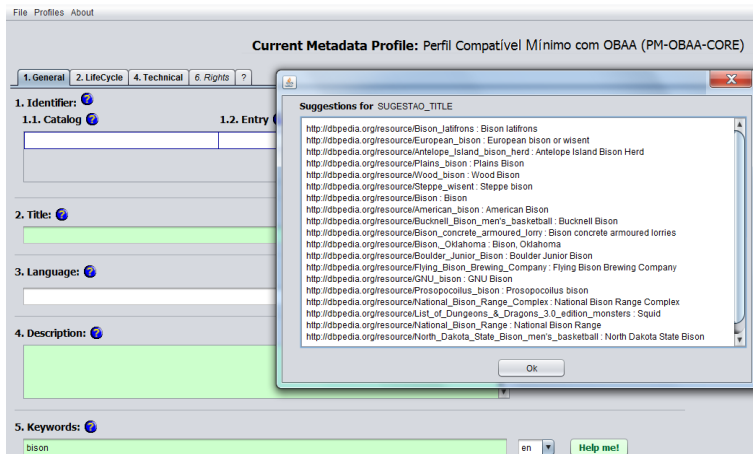


Figura 17 – Saída da ferramenta com o campo Keywords preenchido com a palavra “bison” para o campo Title.

- A sugestão é ruim se for baseada em um indivíduo que...
  - For um dia comemorativo que inclui a palavra-chave.
  - For o nome de uma organização, empresa, produto, livro ou time esportivo que contiver a palavra-chave.
  - Não houver uma relação visível entre a sugestão e a palavra-chave.
  - For o nome de uma pessoa que não seja a descobridora, inventora ou desenvolvedora da palavra-chave.
  - For um conceito altamente acadêmico da palavra-chave original, com poucas chances de ser útil em um OA para o Ensino Médio.

Nos casos que houve dúvida se uma sugestão era boa ou ruim, foi dada a ela um meio ponto, se não for possível definir a qualidade para mais uma sugestão, ela também vale meio ponto, totalizando 1 ponto, ou uma classificação como uma única sugestão boa.

Nas Tabelas 3 e 4 são mostradas as quantidades de indivíduos obtidas por cada abordagem e a qualidade destes resultados. Para ambas as palavras-chaves “bison” e “tuberculosis” foram obtidos poucos resultados pela abordagem de classes, isso foi comum para várias

das palavras-chaves testadas, sendo que quando eram obtidos resultados por essa abordagem eles geralmente eram de baixa qualidade. A abordagem por indivíduos foi a que obteve resultados consistentes com maior frequência, no caso dos dados abaixo, 10 de 23 e 6 de 8 indivíduos foram de boa qualidade.

Tabela 3 – Resultados por abordagem para a palavra-chave “bison”

Bison	Total de indiv. obtidos	Sugestão boa	Sugestão ruim
indivíduos	23	10	13
classes	1	0	1
categorias	26	10	16

Tabela 4 – Resultados por abordagem para a palavra-chave “tuberculosis”

Tuberculosis	Total de ind. obtidos	Sugestão boa	Sugestão ruim
indivíduos	8	6	2
classes	0	0	0
categorias	20	3	17

Na Tabela 5 é mostrada uma análise da qualidade dos 10 resultados finais ranqueados para diferentes palavras-chaves. Durante os testes, percebeu-se que uma outra utilidade da ferramenta é possibilitar a pesquisa de mais informações a respeito do tópico do OA, já que além da sugestão para o campo solicitado é também mostrado o endereço para o indivíduo na DBPedia. Por isso, essa tabela também mostra se nos indivíduos mostrados para o usuário está incluso um indivíduo que é a definição da palavra-chave enviada para a ferramenta. As palavras-chaves escolhidas foram tanto palavras que podem ser conteúdo de um OA para o Ensino Médio como palavras-chaves para as quais pudessem ser aplicadas os critérios de qualidade de forma bem definida.

Houveram 70% de resultados satisfatórios para as palavras-chaves tuberculosis e bison, pois os resultados retornaram vários subtipos destas palavras-chaves. Os resultados para a palavra-chave algebra foram enfraquecidos porque foram retornados vários programas que trabalham com algebra, ao invés de conceitos relacionados à algebra. E tanto para algebra como para geometry foram retornados resultados para conceitos de nível acadêmico, que são considerados de qualidade ruim pois

Tabela 5 – Qualidade dos resultados para algumas palavras-chaves

Palavra-chave	Sugestão boa	Sugestão ruim	Definição é mostrada?
tuberculosis	7	3	sim
bison	7	3	sim
algebra	2	8	não
geometry	1	9	não
morphology	0	10	não

está se considerando o cenário desta ferramenta ser usada para OAs para estudantes de Ensino Médio. Não houve nenhuma boa sugestão gerada para a palavra-chave morphology porque todos os indivíduos retornados representavam livros, e por tanto possuíam informações sobre os livros e não sobre o conceito sendo pesquisado.

Pelos resultados obtidos percebe-se que a ferramenta é eficiente para buscar mais informações sobre um conceito, mas possui um desempenho ruim para obter sugestões específicas para um OA, no cenário de ensino e aprendizagem. Sugestões melhores poderiam ser geradas se fosse usado um repositório semântico específico para ensino e aprendizagem. Tendo em vista que a DBPedia não possui este foco, a amplitude da base semântica gerou tanto problemas em termos de desempenho como de qualidade dos resultados obtidos. Em relação a desempenho, uma busca abrangente leva de 1 a 2 minutos para completar, e uma busca específica pode levar até 10 minutos, isso ocorre pois é necessário utilizar muitos filtros para tentar criar um foco nos resultados para ensino e aprendizagem. E quanto a qualidade, mesmo com a utilização de filtros e com o tempo gasto nas buscas ainda assim são obtidos vários resultados com informações muitos avançadas ou irrelevantes. Ao mesmo tempo, para algumas matérias comuns de sala de aula não foram encontrados nenhum resultado, como logaritmo, entalpia e mercantilismo, pois esses conceitos não foram adicionados à DBPedia.

## 4 CONCLUSÕES

O objetivo geral deste trabalho foi analisar e aplicar a abordagem de Agentes&Artefatos para o desenvolvimento de um sistema de autoria de metadados de objetos de aprendizagem utilizando um repositório semântico. Este objetivo possui os seguintes objetivos específicos:

1. Especificar os requisitos da ferramenta de autoria de metadados.
2. Compreender a tecnologia de A&A.
3. Compreender o armazenamento de dados em repositórios semânticos.
4. Modelar o repositório semântico com A&A para poder ser utilizado pelos agentes.
5. Modelar ou utilizar uma ontologia que será usada pelos agentes para acessar os objetos de aprendizagem do repositório semântico.
6. Implementar o sistema de autoria de metadados com A&A.
7. Contextualizar a contribuição do sistema desenvolvido nas áreas de SMA e e-learning.

Os três objetivos iniciais foram abordados na parte teórica do trabalho, onde se explicou amplamente SMAs e o modelo de A&A nas seções 2.1 e 2.2, o funcionamento e conceitos relacionados aos repositórios semânticos foram abordados na seção 2.3, e uma discussão sobre funcionalidades e requisitos de ferramentas de autoria de metadados foi feita na seção 2.4. Os quatro principais requisitos citados na seção 2.4 foram implementados:

- suportar o padrão de metadados OBAA: A ferramenta suporta o perfil PM-OBAA-CORE, considerado o essencial para que um OA seja adequado ao padrão OBAA.
- facilitar preenchimento dos metadados com A&A: São fornecidas sugestões para três campos de metadados: Title, Description e Keyword, que foram escolhidos pois tem alto potencial para se obter sugestões.
- representar metadados com ontologias: Ao invés de criar uma ontologia para representar os OAs, se escolheu utilizar as propriedades da ontologia da DBPedia que eram compatíveis com os

campos de metadados para os quais se queria gerar as sugestões, como `db-prop:title` e `db-prop:name` para sugestões para o campo Title.

- fornecer um tutorial para a ferramenta e os metadados: Na GUI da ferramenta são disponibilizados tanto uma aba específica com explicações sobre os metadados e categorias do perfil como ícones de interrogação que fornecem mais informações sobre um metadado específico.

Também foi sugerido que a ferramenta fosse Web, para ser mais acessível, e suportasse outros padrões de metadados comuns, como Scorm e LOM, mas se preferiu deixar estas funcionalidades para uma versão futura do trabalho, para que o foco fosse a geração de sugestões.

Quanto aos objetivos 4 e 5, estes foram atingidos utilizando a API Jena em artefatos para criar e processar consultas SPARQL no repositório semântico DBPedia através do ponto de acesso Virtuoso. Foram criados três artefatos especialistas em três tipos de abordagens (por indivíduos, por classes e por categorias) que implementam operações utilizando a API Jena para criar e processar buscas semânticas abrangentes ou específicas, permitindo que os agentes tenham acesso a Web Semântica. A ontologia utilizada foi a da própria DBPedia.

O objetivo 6 também foi alcançado com sucesso: Neste trabalho foi desenvolvida uma ferramenta capaz de obter sugestões para os campos de metadados do perfil PM-OBAA-CORE baseando-se nos campos já preenchidos pelo usuário, que são transformados em palavras-chaves. Estas sugestões são obtidas utilizando um SMA que acessa a DBPedia para obter indivíduos similares as palavras-chaves recebidas, e então obtém os valores de propriedades dos dez indivíduos mais relevantes para retornar como sugestões de preenchimento para o usuário na GUI.

Foram encontradas duas dificuldades na obtenção de sugestões por buscas semânticas na DBPedia, ambas baseadas no fato de a DBPedia não ser um repositório semântico específico para ensino e aprendizagem. Devido aos vários filtros de classes e categorias utilizados para remover resultados que não fossem relevantes para o contexto de ensino e aprendizagem os tempos de consulta à DBPedia são bastante grandes, e mesmo com a utilização destes filtros as sugestões obtidas tiveram uma qualidade baixa este cenário. Além disso, frequentemente não haviam conceitos registrados na DBPedia para matérias comuns de sala de aula. Um possível trabalho futuro para melhorar a performance e qualidade da obtenção de sugestões seria utilizar outro repositório semântico, mais específico para o cenário de ensino e aprendizagem.



Este repositório poderia ser tanto um já existente ou criado do zero.

Por fim, quanto ao último objetivo, a principal contribuição deste trabalho para a área de SMA é ter criado uma ferramenta utilizando A&A que permite que agentes tenham acesso à Web Semântica. Até nosso conhecimento, este é o primeiro protótipo de acesso de agentes à Web Semântica que utiliza esta abordagem. Em relação à área de e-learning, o trabalho desenvolvido é um protótipo inicial para uma ferramenta de geração automática de sugestões para metadados, de forma a facilitar o preenchimento e adequação de OAs à padrões de metadados.

Os artefatos para buscas semânticas desenvolvidos neste trabalho utilizam consultas em SPARQL com estruturas pré-definidas, sendo assim um tanto restritivos. Eles podem ser reutilizados sem grandes modificações para suportar outros tipos de consultas SPARQL, porém seria difícil obter um modelo completamente genérica, já que no final as operações dos artefatos são implementadas em código Java estático. Por outro lado, utilizar artefatos Cartago possibilita interoperabilidade: Qualquer agente BDI pode acessar os artefatos desenvolvidos neste trabalho, mesmo que utilizem linguagens diferentes. Como cada agente utiliza um artefato próprio, as consultas semânticas podem ser processadas em paralelo. Algumas características do modelo de A&A não foram utilizadas neste trabalho, como a capacidade dos agentes de configurar as propriedades de um artefato para alterar sua funcionalidade em tempo de execução, ou criar e remover artefatos em tempo de execução para alterar as funcionalidades do ambiente.

O principal problema encontrado utilizando o modelo A&A foi que ele suporta um agente usar vários artefatos ao mesmo tempo, mas um artefato não pode ser utilizado por mais de um agente por vez. O modelo A&A foi desenvolvido desta forma para evitar conflitos em que mais de um agente pudesse alterar o estado interno do artefato ao mesmo tempo. Contudo, neste trabalho era necessário que vários agentes pudessem executar uma operação para consultar a DBPedia ao mesmo tempo para permitir que o processamento fosse feito em paralelo. Para resolver isso, no nosso modelo foram instanciados um artefato de busca semântica para cada agente. Os artefatos foram identificados com o nome dos agentes que iriam utilizá-los, assim os agentes poderiam saber que artefato devem usar. Essa abordagem foi efetiva para este trabalho, mas não é muito elegante e pode escalar rapidamente em uma situação com um número maior de agentes ou artefatos de busca semântica.

Devido a limitação de tempo algumas funcionalidades pensadas no início do projeto não puderam ser implementadas. Inicialmente planejou-se utilizar o próprio OA para gerar sugestões, em conjunto com o que foi implementado neste trabalho. A princípio isso seria feito apenas para OA's em pdf, o arquivo seria processado utilizando NLP, algumas sugestões que poderiam ser obtidas desta forma são: 1.3. Language (o idioma usado pelo OA), 2.1. Version (a versão do arquivo, poderia ser obtida, por exemplo, procurando pela palavras 'versão' ou 'version') e 2.3.2. Entity (o nome das pessoas que criaram o OA). Também poderiam ser obtidas sugestões para 1.2. Title e 1.5. Keywords, através da análise do texto do OA. Outra funcionalidade seria a de obter sugestões através de comparações entre OAs semelhantes em um mesmo 3Store, através de similaridade entre ontologias. Outro trabalho futuro relevante é comparar a ferramenta de recomendação com outros sistemas similares voltados ao ensino a distância, para avaliar a qualidade das recomendações e o tempo de processamento.

Outros trabalhos futuros possíveis são: agentes configurarem os artefatos para modificar a estrutura das consultas em SPARQL em tempo real, agentes monitorarem os campos da GUI em tempo real; utilizar dados pessoais do usuário como contexto na inferência semântica; implementar outros perfis de metadados do OBAA; implementar suporte para os padrões de metadados Scorm e LOM; a ferramenta ser Web; permitir a customização dos filtros utilizados; e permitir salvar e carregar metadados preenchidos em formato XML.

## **5 AGRADECIMENTOS**

Gostaria de agradecer a orientação do professor Elder Rizzon Santos, que forneceu opiniões e revisões valiosas ao longo do trabalho. Também agradeço o apoio da minha família e do meu marido, presente em todos os momentos deste trabalho.



## REFERÊNCIAS

- ARISHA, K. A. et al. Impact: A platform for collaborating agents. 1999.
- BELLIFEMINE, F. et al. Jade: A java agent development framework. In: *Multi-Agent Programming*. [S.l.]: Springer US, 2005. v. 15.
- BELLIFEMINE, F. L.; CAIRE, G.; GREENWOOD, D. Developing multi-agent systems with jade. 2007.
- BERNERS-LEE, T. *Sir Tim Berners-Lee Talks with Talis about the Semantic Web*. 2008. Transcribed interview.
- BEZ, M. R. et al. Obaa project: An approach to interoperable learning objects based on web and digital television. 2009.
- BISHOP, B. et al. Owlim: A family of scalable semantic repositories. 2011.
- BIZER, C. et al. Dbpedia - a crystallization point for the web of data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 2009.
- BOHL, O. et al. The sharable content object reference model (scorm) - a critical review. In: *Computers in Education, 2002. Proceedings. International Conference on*. [S.l.: s.n.], 2002.
- BOISSIER, O. et al. Organisation and environment oriented programming. 2011.
- BORDINI, R. H.; HUBNER, J. F. Jason: A java-based interpreter for an extended version of agentspeak. 2007.
- BRICKLEY, D.; GUHA, R. V. Rdf vocabulary description language 1.0: Rdf schema. 2004.
- BROEKSTRA, J.; KAMPMAN, A.; HARMELEN, F. V. Sesame: A generic architecture for storing and querying rdf and rdf schema. In: *The Semantic Web—ISWC 2002*. [S.l.]: Springer, 2002. p. 54–68.
- BROOKS, R. A robust layered control system for a mobile robot. *Robotics and Automation, IEEE Journal of*, IEEE, v. 2, n. 1, p. 14–23, 1986.

CHEN, H. et al. Intelligent agents meet the semantic web in smart spaces. *IEEE Internet Computing*, IEEE Educational Activities Department, v. 8, n. 6, nov. 2004.

DASTANI, M. 2apl: a practical agent programming language. *Autonomous Agents and Multi-Agent Systems*, Springer US, v. 16, n. 3, 2008.

D'INVERNO, M. et al. A formal specification of dmars. In: *Intelligent Agents IV Agent Theories, Architectures, and Languages*. [S.l.]: Springer Berlin Heidelberg, 1998. v. 1365.

DUVAL, E. et al. Metadata principles and practicalities. *D-Lib Magazine*, 2002.

ERLING, O.; MIKHAILOV, I. Virtuoso: Rdf support in a native rdbms. In: *Semantic Web Information Management*. [S.l.: s.n.], 2010. p. 501–519.

FARACO, R. A. Uma arquitetura de agentes para negociação dentro do domínio do comércio eletrônico. 1998.

FISHER, M. et al. Computational logics and agents: A roadmap of current technologies and future trend. 1993.

FRANKLIN, S.; GRAESSER, A. Is it an agent, or just a program?: A taxonomy for autonomous agents. 1996.

FRIESEN, N.; ROBERTS, A.; FISHER, S. Cancore: Metadata for learning objects. *Canadian Journal of Learning and Technology / La revue canadienne de l'apprentissage et de la technologie*, v. 28, n. 3, 2002.

GRAU, B. C. et al. Owl 2: The next step for owl. 2008.

GRAVES, M.; CONSTABARIS, A.; BRICKLEY, D. FOAF: Connecting People on the Semantic Web. *Cataloging & classification quarterly*, v. 43, n. 3, p. 191–202, 2007.

GRUNEWALD, D. et al. Agent-based network security simulation. 2011.

HAARSLEV, V.; MÜLLER, R. Racer system description. In: *Automated Reasoning*. [S.l.]: Springer, 2001. p. 701–705.

- HENDLER, J.; HARMELEN, F. van. The semantic web: Webizing knowledge representation. In: *Handbook of KnowledgeRepresentation*. [S.l.]: Elsevier, 2008.
- HINDRIKS, K. et al. Agent programming with declarative goals. In: *Intelligent Agents VII Agent Theories Architectures and Languages*. [S.l.]: Springer Berlin Heidelberg, 2001.
- HOCH, N. et al. A user-centric approach for efficient daily mobility planning in e-vehicle infrastructure networks. 2011.
- HORROCKS, I. Ontologies and the semantic web. *Commun. ACM*, 2008.
- HOWDEN, N. et al. Jack intelligent agents - summary of an agent infrastructure. In: *In 5th International conference on autonomous agents*. [S.l.: s.n.], 2001.
- HUBNER, J. F.; BORDINI, R. H.; VIEIRA, R. Desenvolvimento de sistemas multiagentes. 2004.
- IEEE. Ieee-sa standard 1484.12.1 draft standard for learning object metadata. 2002.
- JENNINGS, N. R.; SYCARA, K.; WOOLDRIDGE, M. A roadmap of agent research and development. *Autonomous Agents and Multi-Agent Systems*, 1996.
- KAGAL, L. et al. Agents making sense of the semantic web. In: *Innovative Concepts for Agent-Based Systems*. [S.l.]: Springer Berlin Heidelberg, 2003.
- KATASONOV, A.; TERZIYAN, V. Semantic agent programming language (s-apl): A middleware platform for the semantic web. In: *Proceedings of the 2008 IEEE International Conference on Semantic Computing*. [S.l.]: IEEE Computer Society, 2008. (ICSC '08), p. 504–511.
- KIRYAKOV, A.; OGNYANOV, D.; MANOV, D. Owlim—a pragmatic semantic repository for owl. In: *Web Information Systems Engineering—WISE 2005 Workshops*. [S.l.]: Springer, 2005.
- KLAPISCAK, T.; BORDINI, R. H. Jasdl: A practical programming approach combining agent and semantic web technologies. In: *In Proc. of DALT 2008, volume 5397 of LNAI*. [S.l.]: Springer, 2009.

KNUBLAUCH, H. et al. The protégé owl plugin: An open development environment for semantic web applications. In: *The Semantic Web-ISWC 2004*. [S.l.]: Springer, 2004. p. 229–243.

KÜSTER, T. et al. Distributed optimization of energy costs in manufacturing using multi-agent system technology. 2012.

MCBRIDE, B. Jena: a semantic web toolkit. *Internet Computing, IEEE*, v. 6, n. 6, p. 55–59, 2002.

MCGUINNESS, D. L.; HARMELEN, F. V. et al. Owl web ontology language overview. *W3C recommendation*, v. 10, n. 2004-03, p. 10, 2004.

OMICINI, A.; RICCI, A.; VIROLI, M. Artifacts in the a&a meta-model for multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 2008.

OMICINI, A.; RICCI, A.; VIROLI, M. Artifacts in the a&a meta-model for multi-agent systems. 2008.

ONTOTEXT. *Semantic Repository*. 2012.

<<http://www.ontotext.com/semantic-repository>>. Acessado em 25/06/2012.

OPENRDF. *RDF*. 2012. <<http://www.openrdf.org/about.jsp>>. Acessado em 20/06/2012.

OPENRDF. *Sesame*. 2012.

<<http://sourceforge.net/projects/sesame/>>. Acessado em 25/06/2012.

PADRON, C. Carlos: a collaborative authoring tool for reusable learning objects. 2003.

PADRON, C. L. et al. Carlos: A collaborative authoring tool for reusable learning objects. 2003.

PÉREZ, J.; ARENAS, M.; GUTIERREZ, C. Semantics and complexity of sparql. *ACM Trans. Database Syst.*, 2009.

RICCI, A.; OMICINI, A.; DENTI, E. Activity theory as a framework for mas coordination. 2003.

RICCI, A. et al. Integrating artifact-based environments with heterogeneous agent-programming platforms. 2008.



RICCI, A.; PIUNTI, M.; VIROLI, M. Environment programming in multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 2010.

RICCI, A. et al. Environment programming in cartago. 2009.

RICCI, A.; VIROLI, M. simpa: An agent-oriented approach for prototyping concurrent application on top of java. 2007.

RICCI, A.; VIROLI, M.; OMICINI, A. The a&a programming model and technology for developing agent environments in mas. In: *Programming Multi-Agent Systems*. [S.l.]: Springer Berlin Heidelberg, 2008.

RICCI, A.; VIROLI, M.; OMICINI, A. Integrating heterogeneous agent programming platforms within artifact-based environments. 2008.

RICCI, A.; VIROLI, M.; OMICINI, A. A meta-model for multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 2008.

RICCI, A.; VIROLI, M.; OMICINI, A. Give agents their artifacts: The a&a approach for engineering working environments. In: *AAMAS '07, Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*. [S.l.]: ACM, 2010.

RICHARDS, J. et al. Authoring tool accessibility guidelines (atag) 2.0. 2012.

RUSSEL, S.; NORVIG, P. *Inteligência Artificial*. [S.l.]: Elsevier, 2003.

SCHREIBER, G. Knowledge engineering. 2008.

SESSELER, R.; KEIBLINGER, A.; VARONE, N. Software agent technology in mobile service environments. 2002.

SHEARER, R.; MOTIK, B.; HORROCKS, I. Hermit: A highly-efficient owl reasoner. In: *OWLED*. [S.l.: s.n.], 2008. v. 432.

SHOHAM, Y. Agent-oriented programming. *Artificial Intelligence*, 1993.

SILVA, D. M. D.; VIEIRA, R. Argonaut: Integrating jason and jena for context aware computing based on owl ontologies (short paper). In: *Proceedings of the Workshop on Agents, Web Services, and Ontologies – Integrated Methodologies (AWESOME'007) held as part of MALLOW'007, Durham 3–7 September*. [S.l.: s.n.], 2007.

SIRIN, E. et al. Pellet: A practical owl-dl reasoner. *Web Semantics: science, services and agents on the World Wide Web*, Elsevier, v. 5, n. 2, p. 51–53, 2007.

SPANOUidakis, N.; MORAITIS, P. An ambient intelligence application integrating agent and service-oriented technologies. 2007.

SUCHANEK, F. M.; KASNECI, G.; WEIKUM, G. Yago: a core of semantic knowledge. In: *Proceedings of the 16th international conference on World Wide Web*. [S.l.]: ACM, 2007. (WWW '07).

TSARKOV, D.; HORROCKS, I. Fact++ description logic reasoner: System description. In: *Automated reasoning*. [S.l.]: Springer, 2006. p. 292–297.

TWEEDALEA, J. et al. Innovations in multi-agent systems. 2006.

W3C. *SemanticWeb: Inference*. 2012.

<<http://www.w3.org/standards/semanticweb/inference>>.

Acessado em 20/06/2012.

WEIBEL, S. The dublin core: A simple content description model for electronic resources. *Bulletin of the American Society for Information Science and Technology*, Wiley Subscription Services, Inc., A Wiley Company, v. 24, n. 1, 1997.

WEYNS, D.; OMICINI, A.; ODELL, J. Environment as a first class abstraction in multiagent systems. *Autonomous Agents and Multi-Agent Systems*, 2006.

WEYNS, D.; PARUNAK, H. V. D. Special issue on environments for multi-agent systems. *Journal of Autonomous Agents and Multi-Agent Systems*, 2007.

WEYNS, D.; PARUNAK, H. V. D. Special issue on environments for multi-agent systems. 2007.

WIKIPEDIA. *Ontology Components*. 2012.

<[http://en.wikipedia.org/wiki/Ontology\\_components](http://en.wikipedia.org/wiki/Ontology_components)>. Acessado em 20/06/2012.

WILEY, D. A. Learning object design and sequencing theory. 2000.

WOOLDRIDGE, M. Intelligent agents: The key concepts. 2002.

WOOLDRIDGE, M.; JENNINGS, N. R. Intelligent agents: Theory and practice. 1995.

WOOLDRIDGE, M. J.; JENNINGS, N. R. Agent theories, architectures, and languages: A survey. 1995.

XAVIER, A. C.; GLUZ, J. C. *Relatorio Tecnico RT-OBAA-06 Subgrupo Agentes e Ontologias: Analise Comparativa de Editores de Objetos de Aprendizagem*. [S.l.], 2009.

ZOU, Y. et al. Using semantic web technology in multi-agent systems: a case study in the taga trading agent environment. In: *Proceedings of the 5th international conference on Electronic commerce*. [S.l.]: ACM, 2003. (ICEC '03), p. 95–101.



## **APÊNDICE A – Background de Sistemas Multi-Agentes**



## A.1 TEORIAS DE AGENTES

Para (WOOLDRIDGE; JENNINGS, 1995a) teorias de agentes são essencialmente especificações, que buscam definir aspectos tais como o que um agente é, que propriedades ele deveria ter, e como pode-se representar e raciocinar sobre estas propriedades.

A abordagem tradicional para se construir agentes é a teoria baseada em lógica, segundo (WOOLDRIDGE, 2002) ela sugere que um comportamento inteligente pode ser gerado em um sistema se este sistema possuir uma representação simbólica do seu ambiente e do seu comportamento desejado, além de ser capaz de manipular sintaticamente esta representação. A representação simbólica é feita através de um banco de dados de fórmulas lógicas, e a manipulação é realizada através de deduções lógicas ou prova de teoremas. Os agentes que eram implementados de acordo com essa teoria eram chamados de agentes deliberativos e o seu comportamento era determinado pelo seu banco de dados atual e pelas suas regras de dedução.

Um exemplo citado por (WOOLDRIDGE, 2002) de um agente deliberativo é o “Mundo do Vácuo” (Vacuum World). Como mostrado na Figura 18, nele um agente aspirador habita um mundo de quadrados 3 por 3, e o seu objetivo é manter este mundo limpo. Alguns predicados que ele possui para armazenar informações do ambiente são: *localizacao(x,y)*, que armazena sua localização no mundo; *sujeira(x,y)*, que indica que a posição (x,y) está suja; e *direcao(d)*, que guarda a direção para a qual o agente está virado. Ações que o agente poderia possuir para cumprir seu objetivo são: *limpar*, para sugar a sujeira de uma célula suja; *andar*, para mover-se para frente; e *virar*, para alterar a direção do movimento. Uma regra de dedução seria “*localizacao(x,y)* AND *sujeira(x,y)* -> *executar(limpar)*”, para o agente limpar uma célula caso ele esteja localizado nela e esta célula esteja suja.

Segundo (WOOLDRIDGE, 2002) esta teoria possui as qualidades de ser elegante e possuir uma semântica lógica, porém possui várias desvantagens. A primeira é o fato de ser difícil para um ser humano criar e compreender o significado de um grande conjunto de predicados lógicos e regras de dedução, ou conseguir fazer o mapeamento de um ambiente para este conjunto de predicados. O segundo é um problema de performance, porque há um problema de complexidade computacional inerente para se realizar prova de teoremas, isso faz com que o uso de agente deliberativos em sistemas com restrições de tempo seja questionável. Outra desvantagem relevante é que a tomada de decisão destes agente é baseada no princípio da racionalidade calculável, ou

seja, é assumido que o ambiente não mudará de forma significativa enquanto o agente decide que ação tomar, o que é bastante improvável de ocorrer em sistemas dinâmicos e complexos.

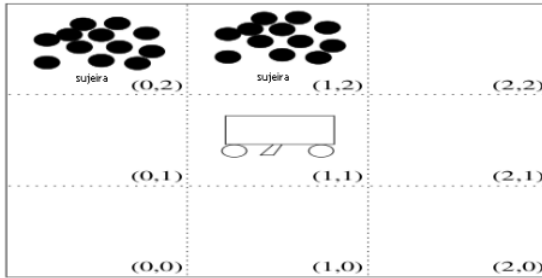


Figura 18 – Agente deliberativo Aspirador no Mundo do Vácuo (WO-OLDRIDGE, 2002)

Atualmente a teoria mais utilizada para se modelar agentes é a BDI (Belief-Desire-Intention, Crença-Desejo-Intenção) (SHOHAM, 1993). Esta teoria é baseada em estudos sobre o raciocínio prático, que envolve dois processos principais: decidir que metas queremos atingir (deliberação) e como as atingiremos (meios-fins). Noções mentais humanas e uma perspectiva social são utilizadas na modelagem dos agentes.

O processo de raciocínio prático de um agente em uma arquitetura BDI está resumido na Figura 19, que ilustra os sete componentes principais de um agente BDI. Destes, os conceitos básicos são:

- **Crenças:** Um conjunto de crenças que representa a informação que o agente possui sobre si, sobre o ambiente e sobre outros agentes. Função de Revisão de Crenças (FRC) recebe como entrada uma percepção e as crenças atuais do agente, e baseando-se nisso atualiza as crenças do agente.
- **Desejos:** São os estados do mundo desejados pelo agente, podem ser contraditórios. Os desejos são gerados pela Função de Geração de Opções, que utiliza as crenças para verificar a viabilidade dos desejos e utiliza as intenções atuais do agente de forma que o desejo não entre em conflito com elas. O subconjunto de desejos consistentes é chamado de objetivos.
- **Intenções:** São sequências de ações que o agente se compromete a executar para atingir seus objetivos. As intenções são geradas



pelo processo de deliberação, levando-se em conta as crenças, intenções e desejos atuais. Um agente pode possuir várias intenções simultaneamente, a escolha sobre qual dessas intenções será executada é feita pelo componente Ação.

Outros conceitos comuns que não aparecem na Figura 19, mas que são usados por algumas arquiteturas BDI:

- Eventos: Alterações nas crenças ou objetivos dos agentes.
- Capacidades: Atividades que um agente pode realizar.
- Planos: Raciocínio sobre os cursos de ações necessários para se atingir um objetivo.
- Regras: Raciocínio sobre as crenças, de forma a inferir conhecimento a partir delas.

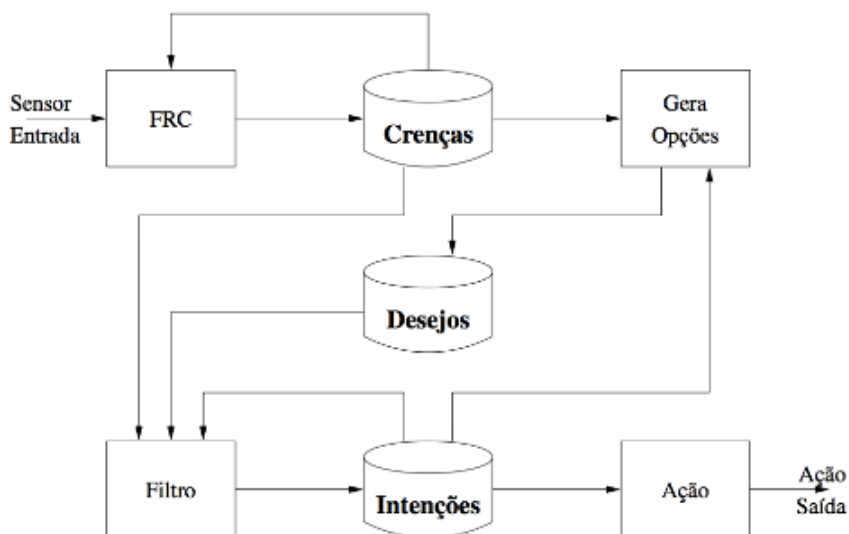


Figura 19 – Diagrama do processo de raciocínio em uma arquitetura BDI genérica. (HUBNER; BORDINI; VIEIRA, 2004)

(WOOLDRIDGE, 2002) considera que um dos maiores problemas nas arquiteturas BDI é alcançar um equilíbrio na frequência com que

um agente deve repensar suas intenções. Em ambientes altamente dinâmicos e imprevisíveis cumprir um objetivo pode se tornar impossível, ou não ser mais necessário. Porém o agente também deve manter compromisso por tempo o suficiente com seus objetivos para que ele possa cumpri-los. O autor também cita que outra dificuldade é implementar os processos de deliberação e meios-fim de forma eficiente.

Segundo (WOOLDRIDGE, 2002) as duas principais qualidades das arquiteturas BDI são que a forma de raciocínio que ela descreve é bastante intuitiva, e que o sistema possui uma decomposição clara.

## A.2 ARQUITETURAS DE AGENTES

Para (WOOLDRIDGE; JENNINGS, 1995a) um arquitetura de agentes é um modelo de engenharia de software para agentes, que se preocupa principalmente em satisfazer as propriedades especificadas pelas teorias de agentes. Abaixo seguem-se as arquiteturas de agentes reativos, a arquitetura em camadas e algumas arquiteturas baseadas na teoria de agentes BDI.

As arquiteturas reativas surgiram, na década de 1980, como uma alternativa aos problemas aparentemente intratáveis da abordagem lógica, descritos na seção anterior. (WOOLDRIDGE, 2002) comenta que a principal semelhança entre estas arquiteturas era a rejeição da IA simbólica, além de considerarem que o comportamento racional de um agente emerge da interação de vários comportamentos mais simples e do ambiente em que o agente está situado. Estas arquiteturas foram chamadas de reativas porque a tomada de decisão é implementa com um mapeamento direto de uma situação para uma ação, sem que haja um raciocínio sobre o ambiente.

Segundo (WOOLDRIDGE, 2002) a arquitetura reativa mais conhecida é a arquitetura de subsunção (subsumption architecture), desenvolvida por Rodney Brooks (BROOKS, 1986). Nesta arquitetura o processo de tomada de decisão é realizado através de um conjunto de comportamentos, no qual cada comportamento pode ser considerado como uma função que continuamente mapeia as percepções de entrada em

uma ação a ser executada. Outra característica importante da arquitetura de subsunção é que vários comportamentos podem ser ativados ao mesmo tempo. Para ser possível escolher entre eles, os comportamentos são organizados em camadas. As camadas mais baixas tem maior prioridade para executar e representam comportamentos menos abstratos, como desviar de um obstáculo, enquanto as camadas mais altas possuem menor prioridade e comportamentos mais abstratos, como a comunicação.

Em (WOOLDRIDGE, 2002) é citado um exemplo no qual um conjunto de veículos autônomos deve explorar um terreno desconhecido em busca de um tipo específico de rocha a ser recolhida, além disso, os veículos não são capazes de se comunicar entre si. Nesta situação, o comportamento de mais baixo nível era desviar de obstáculos, por ser algo que deve ser executado de imediato, evitando que o veículo quebre. Os comportamentos de nível intermediário eram relacionados a buscar e trazer as rochas, e o comportamento de mais alto nível era andar aleatoriamente pelo mapa em busca de rochas, efetuado quando não houvesse nenhum obstáculo ou rocha por perto.

(WOOLDRIDGE; JENNINGS, 1995a) também aborda outra arquitetura reativa, o autômato situado (*situated automata*), desenvolvido por Rosenchein e Kaelbling. Nesta arquitetura um agente é especificado de forma declarativa, com lógica modal, e então essa especificação é compilada em uma máquina que a satisfaça. Esta máquina é capaz de rodar dentro de restrições de tempo, pois não utiliza nenhuma manipulação simbólica. Para esta técnica funcionar é necessário que o mapeamento do mundo para Semântica dos Mundos Possíveis (*Possible Worlds Semantics*) possa ter uma interpretação concreta em um autômato de estados.

Algumas vantagens da abordagem reativa descritas por (WOOLDRIDGE, 2002) são: simplicidade, economia, tratabilidade computacional, robustez e elegância. Porém também são citados alguns problemas destas arquiteturas: os agentes se baseiam em informação local, sendo difícil tomarem decisões levando em consideração informações não-locais; não é possível fazer um agente puramente reativo aprender com a experiência; é muito difícil modelar e entender agentes complexos com esta arquitetura, principalmente porque é difícil compreender as interações entre as camadas.

As arquiteturas em camadas surgiram como uma solução possível para o requisito de que um agente deve ser capaz de agir tanto de forma reativa como pró-ativa. Para resolver este problema estas arquiteturas

decompõe o sistema em subsistemas separados que lidam com diferentes tipos de comportamento com diferentes níveis de abstração, esses subsistemas são organizados em uma hierarquia de camadas que interagem entre si. O fluxo de controle entre estas camadas por ser horizontal ou vertical, sendo que o fluxo vertical pode passar por uma via de sentido único ou de sentido duplo (Figura 20). No fluxo horizontal as camadas competem para gerar ações, o que pode levar a inconsistências, além de haver um grande número de interações possíveis entre as camadas. Estes problemas são aliviados nas organizações com fluxo de controle vertical, porém desta forma a tomada de decisão perde flexibilidade, pois o fluxo deve passar por cada uma das camadas.

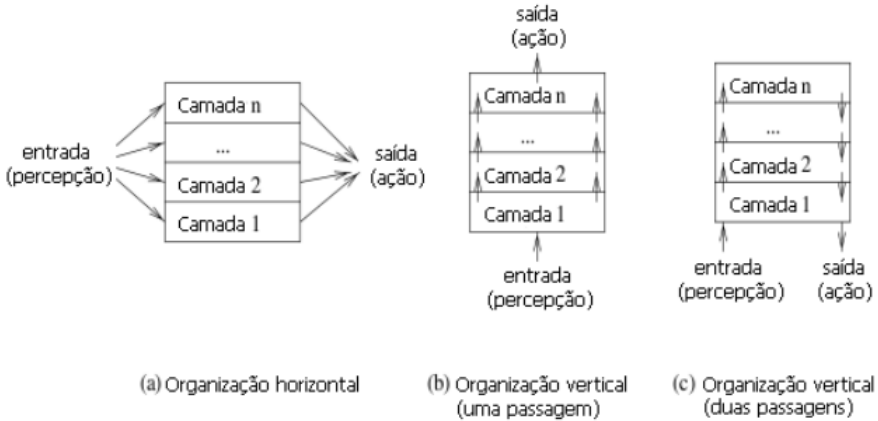


Figura 20 – Os três tipos de fluxo de controle e de informação que uma arquitetura em camadas pode ter (WOOLDRIDGE, 2002).

(WOOLDRIDGE, 2002) cita duas arquiteturas em camadas como exemplo: TouringMachines, com organização horizontal, e INTERRAP, com organização vertical de duas passagens. A arquitetura TouringMachines consiste de três camadas produtoras de atividades (camada reativa, camada de planejamento, camada de modelagem) que utilizam diferentes níveis de abstração para gerar continuamente sugestões sobre qual ação o agente deve tomar, e um sistema de controle, que decide qual camada decidirá as ações do agente. A arquitetura INTERRAP também possui três camadas de controle (camada reativa, camada de planejamento, camada de planejamento cooperativo), cada camada possui uma base de conhecimento associada à ela, que representa

o mundo em diferentes níveis de abstração de acordo com a camada. A utilização de bases de conhecimento é a principal diferença entre essas duas arquiteturas. É relevante notar que a arquitetura de subsunção também é considerada uma arquitetura em camadas (BROOKS, 1986).

De acordo com (WOOLDRIDGE, 2002), as arquiteturas em camadas possuem a vantagem de terem um conceito de fácil compreensão, porém são difíceis de se modelar e compreender para sistemas complexos, por causa das várias interações possíveis entre as camadas.

Abaixo serão descritas as seguintes arquiteturas, baseadas na teoria BDI descrita na seção anterior: PRS (Georgeff, Lansky), dMARS (Kinny) e IRMA (Bratman, Isreal, Pollack).

PRS (Procedural Reasoning System) (WOOLDRIDGE; JENNINGS, 1995a) : É um dos modelos de arquitetura de agentes mais conhecidos. A arquitetura inclui uma biblioteca de planos e uma representação simbólica de crenças, desejos e intenções. Crenças são fatos, tanto sobre o ambiente como sobre o próprio agente, e são expressas em lógica de primeira ordem. Desejos são representados como comportamentos do sistema (ao invés de uma representação estática dos estados dos objetivos). A biblioteca de planos contém um conjunto de planos parcialmente elaborados, chamados de áreas de conhecimento, que são associados às condições de invocação que determinam quando um plano será ativado. Áreas de conhecimento podem ser ativadas tanto de uma forma reativa como direcionada a objetivos. O conjunto de áreas de conhecimento ativadas no sistema representam as intenções do agente. Existe um interpretador, que é o responsável por atualizar crenças, invocar áreas de conhecimento e executar ações.

dMARS (Distributed Multi-Agent Reasoning System) (D'INVERNO et al., 1998): Este modelo é baseado na arquitetura PRS, sendo que um dos objetivos desta arquitetura é ter um tempo de resposta a mudanças confiável, além de prover suporte para operações distribuídas. As capacidades dos agentes dMARS são modeladas como um conjunto de planos sensíveis ao contexto, que podem tanto ser utilizados de forma reativa como pró-ativa. (D'INVERNO et al., 1998) cita algumas aplicações que utilizaram a arquitetura dMARS com sucesso: um sistema de diagnóstico de falhas para um ônibus espacial da Nasa, um sistema de simulação de defesa aérea tática para a Força Aérea da Austrália e um sistema de controle de tráfego aéreo civil para os Serviços Aéreos da Austrália.

(FISHER et al., 1993) resume o ciclo de funcionamento de agentes das arquiteturas PRS e dMARS com o seguinte comportamento:

1. Observam o mundo e o estado interno do agente;
2. Atualizam a fila de eventos para refletir os eventos que foram observados;
3. Geram novos desejos possíveis encontrando planos cujo evento de ativação esteja na fila de eventos;
4. Destes planos possíveis, um é selecionado para ser executado e é adicionado à pilha de uma intenção nova ou de uma já existente;
5. Por fim, uma intenção é selecionada e o plano mais alto da pilha desta intenção é executado.

IRMA (Intelligent Resource-bounded Machine Architecture) (WOOLDRIDGE; JENNINGS, 1995a) : Esta arquitetura tem quatro estruturas de dados chave: uma biblioteca de planos e uma representação explícita de crenças, desejos e intenções. A arquitetura também possui um motor de inferência (reasoner), para raciocinar sobre o mundo; um analisador meios-fins, para determinar que planos podem ser usados para concluir uma intenção; um analisador de oportunidades, que monitora o ambiente para determinar novas opções para o agente; um processo iterador, que determina o subconjunto das possíveis sequências de ações do agente que são consistentes com as intenções atuais; e um processo deliberador, que realiza a escolha entre opções concorrentes. A arquitetura IRMA foi avaliada com um cenário experimental conhecido como Mundo dos Azulejos (Tileworld).

### A.3 LINGUAGENS DE AGENTES

Para (WOOLDRIDGE; JENNINGS, 1995a) uma linguagem de agente é um sistema que permite programar e executar agentes, e que podem utilizar princípios estabelecidos pelas teorias de agentes. A seguir descrevem-se algumas linguagens de agentes, elas estão categorizadas em linguagens baseadas em lógica e linguagens baseadas em BDI.

Algumas linguagens baseadas na teoria lógica de agentes são descritas por (FISHER et al., 1993): AGENT-0, MetateM, Golog e IMPACT (ARISHA et al., 1999).

AGENT-0: É um framework que consiste de um sistema lógico para definir o estado mental dos agentes, junto com uma linguagem interpretada para programar estes agentes. AGENT-0 utiliza lógica

multi-modal quantificada, com três modalidades: crença, compromisso e habilidade. A linguagem também provê suporte para noções de tempo. Embora ela não tenha sido amplamente utilizada, a idéia de uma linguagem para agentes baseados em lógica multi-modal serviu de inspiração para o desenvolvimento de muitas outras linguagens de programação para agentes.

**MetateM:** Esta linguagem utiliza lógica temporal para especificar o comportamento dos agentes, como o que um agente deve fazer agora, o que ele deve fazer a seguir e o que ele garante que irá fazer em algum ponto no futuro. Paralelamente, a linguagem foi desenvolvida em uma versão concorrente para SMAs, chamada *Concurrent MetateM*.

**Golog:** É uma linguagem de especificação de alto nível descendente do Prolog e é baseada em cálculo situacional (*situation calculus*), ela o estende com estruturas de controle e procedimentos. A execução de um programa Golog envolve provar o teorema expresso pelo programa em relação a alguma teoria base. Uma das linguagens sucessoras da Golog foi a *ConGolog*, que incorpora concorrência, interrupções e ações externas à linguagem original.

**IMPACT:** Agentes Impact consistem de um código de software com um empacotador associado que “agentiza” o código. Um conjunto de servidores especializados facilita a interoperabilidade entre os agentes de uma maneira independente de aplicação.

A seguir serão descritas brevemente algumas linguagens para a programação de agentes baseadas na teoria e arquitetura BDI: *AgentSpeak*, *Jason*, *2APL*, *GOAL* e *JACK*.

**AgentSpeak** (FISHER et al., 1993): *AgentSpeak(L)* é uma linguagem de alto-nível baseada nos princípios da teoria BDI, e definida para ser uma extensão da programação lógica que incorpore crenças, eventos, objetivos, ações, planos e intenções. O seu ciclo de raciocínio é bastante similar ao ciclo da arquitetura dMARS. Arquiteturas de Agentes. O comportamento dos agentes é dirigido pelos planos escritos em *AgentSpeak*. Embora as crenças, desejos e intenções do agente não sejam representadas explicitamente como lógica modal, elas podem ser formalmente atribuídas ou checadas dada uma representação do estado atual do agente.

**Jason** (BORDINI; HUBNER, 2007): *Jason* é um interpretador de uma versão melhorada da linguagem *AgentSpeak(L)*, que inclui comunicação baseada na teoria de atos de fala. Um SMA nesta linguagem pode ser distribuído em uma rede utilizando os frameworks *Cartago*, *JADE* ou *Saci*. O *Jason* é implementado em Java, sendo

multi-plataforma, e é código-aberto.

2APL (DASTANI, 2008): 2APL é uma linguagem de programação de agentes cujo objetivo é facilitar a implementação de SMAs. No nível multi-agente, ela provê estruturas de programação para especificar um SMA em termos de um conjunto de agentes individuais e um conjunto de ambientes sobre os quais os agentes podem atuar. No nível de agentes individuais, a linguagem provê estruturas de programação para implementar crenças, objetivos, planos, ações, eventos e um conjunto de regras que o agente utiliza para decidir que ações irá executar. As ações podem ser atualização de crenças, ações externas ou ações de comunicação. Outras características da linguagem 2APL são: é uma linguagem de programação modular que permite encapsular os componentes cognitivos em módulos; suporta a implementação tanto de agentes reativos como pró-ativos; e permite que os SMAs sejam distribuídos, através do uso da plataforma Jade.

GOAL (HINDRIKS et al., 2001): Um agente GOAL é um conjunto de módulos que consiste de vários componentes incluindo: conhecimento, crenças, objetivos, regras de ações e especificações de ações. A maioria das seções é opcional. Cada uma destas seções é representada em uma linguagem de representação de conhecimento, como o Prolog ou a SQL. As principais características desta linguagem são: crenças e objetivos declarativos; estratégia de compromisso cega como padrão; seleção de ações baseada em regras; módulos de intenção baseados em políticas; e comunicação em nível de conhecimento.

JACK (FISHER et al., 1993; HOWDEN et al., 2001): JACK é uma plataforma para a construção e execução de SMAs em nível comercial usando uma abordagem baseada em componentes. Esta linguagem de programação de agentes estende o Java com conceitos orientados a agentes, como agentes, capacidades, eventos, planos, bases de conhecimento e administração de recursos e concorrência. JACK se baseia na arquitetura dMARS.

### A.3.1 2APL

Esta subseção mostra um código simplificado de um agente e um ambiente 2APL, e faz comparações entre os ambientes programados em Jason e em 2APL.

Os dois exemplos, retirados de (BOISSIER et al., 2011), se referem a um Mundo de Blocos. O ambiente contém 2 agentes e diversas bombas espalhadas por ele. O objetivo dos agentes é remover as bombas do



ambiente, sendo que um agente é responsável por procurar e trazer as bombas para um segundo agente, que está localizado na posição (0,0) do mundo, e apenas elimina as bombas que forem trazidas.

Uma versão reduzida do código do primeiro agente é mostrada abaixo. A seção `BeliefUpdates` possui um conjunto de regras na estrutura contexto-ação-atualização, que atualiza as crenças do agente dependendo das ações que ele executar. Neste exemplo, em “carregando(bomba) Soltar( ) not carregando(bomba)”, se o agente acreditar que está carregando uma bomba e realizar a ação “Soltar”, ele passará a acreditar que não está mais carregando a bomba. A seção `Belief` armazena as crenças e as regras de dedução de conhecimento do agente, como as crenças da posição inicial do agente (`inicio(0,1)`), da posição das bombas (`bomba(3,3)`) e a regra `limpar(mundoDeBlocos)`, que indica que se o agente não conhece nenhuma posição em que há uma bomba e não está carregando uma bomba, então o mundo está limpo. Os planos ficam na seção `Plans`, e neste exemplo há um plano que faz com que o agente realize a ação “entrar” no ambiente. A seção `Goals` possui os objetivos do agente, e neste caso o objetivo é cumprir a regra `limpar( mundoDeBlocos )`.

As duas últimas seções relevantes do exemplo são `PG-rules`, regras de planejamento de objetivo, e `PC-rules`, regras de chamada de procedimento. A `PG-rule` `limpar( mundoDeBlocos )` indica que, caso o agente saiba que existe uma bomba em algum (X,Y), ele deve ir até esse (X,Y), pegar a bomba, ir até (0,0) e soltar a bomba, para que o segundo agente possa pegá-la e descartá-la. Em `PC-rule`, a regra `goto(X, Y)` implementa os procedimentos que devem ser feitos para o agente ir para (X,Y), neste caso o agente percebe qual é sua posição atual (A,B) e tempo fazer ela se aproximar da posição (X,Y) chamando-se recursivamente.

---

```

2 BeliefUpdates :
  { carregando(bomba) }          Soltar( )
                                  { not carregando(bomba) }
  { true }                      AddBomba(X,Y)
                                  { bomba(X,Y) }
4   ...

6 Beliefs :
  inicio(0,1).
8 bomba(3,3).
  limpar( mundoDeBlocos ) :- not bomba(X,Y) , not carregando(
                             bomba).

10 Plans :
12 B(inicio(X,Y)) ;
```

```

@mundodebloco( entrar( X, Y, blue ), L )
14
Goals:
16 limpar( mundoDeBlocos )

18 PG-rules:
limpar( mundoDeBlocos ) <- bomba( X, Y ) |
20 {
    goto( X, Y );
22    @mundodebloco( pegar( ), L1 );
    Pegar( );
24    RemoverBomba( X, Y );
    goto( 0, 0 );
26    @mundodebloco( soltar( ), L2 );
    Soltar( )
28 }
...
30
PG-rules:
32 goto( X, Y ) <- true |
{
34    @mundodebloco( sentirPosicao( ), POS );
    B(POS = [A,B]);
36    if B(A > X) then
    { @mundodebloco( oeste( ), L );
38        goto( X, Y )
    }
40    ...
}
42
...
```

---

O ambiente em 2APL também é implementado como uma classe Java mas, ao contrário do ambiente em Jason que centraliza a execução de ações em um único método, em 2APL cada ação do agente é mapeada em um método da classe Java, e são esses métodos que geram as percepções dos agentes. No exemplo de código abaixo as ações entrar, sentirPosicao, pegar, norte, entre outras podem ser realizadas sobre o ambiente mundodebloco.

```

1 package mundodebloco;

3 public class Env extends apapl.Environment {

5     public void entrar(String agente, Term x, Term y, Term c
        ) {...}
    public Term sentirPosicao(String agente) {...}
7     public Term pegar(String agente) {...}
    public void norte(String agente) {...}
}
```

9

}

...

#### A.4 FRAMEWORKS DE SMA: JIAC E JADE

JIAC (Java-based Intelligent Agent Componentware) (SESSELER; KEIBLINGER; VARONE, 2002) é uma arquitetura e framework de agentes baseada em Java, possui uma estrutura modular e foi desenvolvida em 2002. O framework suporta a modelagem, implementação e implantação de sistemas multi-agentes. JIAC é focado em distribuição, escalabilidade, adaptabilidade, autonomia e segurança, e também permite o reuso de aplicações e serviços através de bibliotecas. Os agentes JIAC são baseados na arquitetura BDI, e sua arquitetura provê suporte para comunicação, administração de processos, planos declarativos e definições de ontologias. SMAs utilizando JIAC venceram os anos de 2007, 2008, 2009, 2010 e obtiveram o terceiro lugar em 2011 do Multi-Agent Programming Contest <sup>1</sup>.

A arquitetura e framework JADE (Java Agent DEvelopment Framework) (BELLIFEMINE et al., 2005) é um software implementado na linguagem Java que permite a implementação de sistemas multi-agentes através de um middleware que cumpre as especificações FIPA e através de um conjunto de ferramentas que auxiliam a depuração e a implantação do sistema. JADE permite que a plataforma de agentes possa ser distribuída em diferentes máquinas. Sua arquitetura de comunicação cria e administra filas de mensagens ACL (Agent Communication Language) recebidas, privadas para cada agente, que podem acessá-las de diferentes formas como por um tempo limite, por votação ou por um casamento de padrão. JADE é uma das plataformas de agentes mais utilizadas na comunidade de pesquisa. Jadex é uma extensão do framework básico do JADE utilizando uma arquitetura baseada na BDI.

---

<sup>1</sup><http://multiagentcontest.org>



## **APÊNDICE B – Informações adicionais sobre Agentes & Artefatos**



## B.1 FRAMEWORKS DE A&A: SIMPA

A descrição que se segue é baseada em (RICCI; VIROLI, 2007), o artigo também possui um exemplo de programação utilizando simpA para fazer o Jantar dos Filósofos.

simpA é uma extensão baseada em bibliotecas do Java que suporta as abstrações do modelo de A&A como entidades de primeira-classe, tratando-as como blocos básicos para programar aplicações concorrentes complexas. O framework provê uma camada de abstração orientada a agentes sobre a camada básica de orientação a objetos, utilizando as anotações do Java sobre classes e métodos para definir os novos componentes de programação necessários. simpA é código-aberto e está disponível para download neste website: <http://www.alice.unibo.it/simpa>

Abaixo segue um exemplo de código de um agente em simpA, adaptado de (RICCI; VIROLI, 2007). Este agente realiza uma sequência de atividades, na qual a atividade B depende da atividade A ter sido completada. Em simpA um agente é descrito por uma classe, que deriva da classe Agent, e são usadas anotações Java para definir características de agentes nesta classe, como @ACTIVITY para métodos que representem ações e @ACTIVITY\_WITH\_AGENDA para definir uma sequência de ações. A atividade A cria uma nova crença “x” de valor 1, com o método memo(nome, valor). A atividade B utiliza o método getMemo(nome) para obter a crença armazenada por A e imprimi-la na tela. A implementação de artefatos em simpA é muito semelhante a sua implementação no Cartago, e será descrita na seção seguinte.

---

```

2 public class MeuAgente extends Agent {
3     @ACTIVITY_WITHAGENDA({
4         @TODO(“atividadeA”),
5         @TODO(“atividadeB”, pre=“completed(atividadeA)” ),
6     }) void main(){}

7
8     @ACTIVITY void atividadeA(){
9         memo(“x”,1); // insere uma nova crença x(1)
10    }

11
12    @ACTIVITY void atividadeB(){
13        int v = getMemo(“x”).intValue(0); // lê o primeiro
14        argumento da crença x
15        log(“Resultado: ”+v);
16    }

```

---

## B.2 EXEMPLOS DE IMPLEMENTAÇÃO COM CARTAGO

Serão descritos exemplos implementando um SMA com um artefato Contador, um com um artefato Relógio, um implementando um artefato Sincronizador e um exemplo mais complexo, o Jantar dos Filósofos. Os primeiros exemplos foram retirados e adaptados de (RICCI et al., 2009), e o último exemplo foi retirado de (RICCI; PIUNTI; VIROLI, 2010), em ambos os artigos se encontram outros exemplos como um artefato sendo um buffer limitado, um artefato GUI, um artefato Web-Service, um artefato tabuleiro de jogo da velha, entre outros. Todos estes exemplos, inclusive os descritos abaixo, utilizam o Jason como plataforma de agentes. Outros exemplos encontram-se no site da plataforma Cartago: <http://cartago.sourceforge.net/>

Os exemplos serão descritos através da implementação do código Java dos artefatos, os trechos em **negrito** são da API do Cartago, e de uma imagem da implementação abstrata deste artefato, na qual aparecem as propriedades observáveis do artefato na parte de cima da imagem, e dentro da caixa são listadas as operações que o artefato fornece. Também serão mostrados os códigos dos agentes Jason e a saída da execução do SMA no console.

Como os agentes utilizando os artefatos nos exemplos usam a plataforma Jason, segue uma breve descrição da sintaxe da plataforma, de acordo com (RICCI; PIUNTI; VIROLI, 2010). Os agentes programados em Jason são definidos por um conjunto de crenças iniciais, um conjunto de objetivos e um conjunto de planos que os agentes podem instanciar e executar dinamicamente para atingir seus objetivos. Os planos dos agentes são descritos por regras do tipo Evento: Contexto **j**- Corpo, onde Evento representa o evento que ativou o plano, o Contexto é uma fórmula lógica definindo as condições sobre as quais o plano deve ser executado e o Corpo inclui ações, como criar sub-objetivos (!g), atualizar o estado do agente e executar ações externas sobre o ambiente. Eventos podem ser adição de crenças (+b), adição de um objetivo (+!g), falha de um plano (!g) ou remoção de uma crença (-b).



### B.2.0.1 Artefato Contador

Este primeiro exemplo serve para demonstrar o básico da criação de um artefato. O artefato Contador possui a propriedade observável “contagem” de valor 0, definida no método `init()` com o método `defineObsProperty(...)`, e provê uma operação para incrementá-lo, chamada “inc”, implementada pelo método `inc()` que obtém a propriedade “contagem” e a atualiza para `contagem+1`. Isso é feito através do método `getObsProperty(“contagem”)`, que retorna um objeto `ObsProperty prop`, representando a propriedade obtida, e se utiliza este objeto para invocar `updateValue(...)` e assim atualizar o valor da propriedade. A operação “inc” é composta de uma única etapa atômica. Neste caso, o artefato não possui nenhuma variável de estado, que seria implementada diretamente como um campo de variável do objeto. Um exemplo de uso para este artefato seria registrar a quantidade páginas Web com uma determinada informação encontradas por uma equipe de agentes. Abaixo segue o código da implementação.



Figura 21 – Representação abstrata do Artefato Contador

---

```

1 package Artefatos;
2
3 import cartago.*;
4
5 public class Contador extends Artifact {
6
7     void init(){
8         defineObsProperty(“contagem”,0);
9     }
10
11     @OPERATION void inc(){
12         ObsProperty prop = getObsProperty(“contagem”);
13         prop.updateValue(prop.intValue()+1);
14     }
15 }

```

---

Abaixo segue o código de um agente Jason simples que utiliza o artefato Contador. O agente no plano inicial `+!teste_contador` primeiro cria um artefato com a ação interna `makeArtifact`, provida pelo Cartago. Esta chamada recebe como parâmetros o nome da instância do artefato (`meuContador`), a classe do artefato (`Artefatos.Contador`), uma lista de parâmetros para o método `init` do artefato (`[]`) e uma variável que receberá um identificador para este artefato (`Id`). Após a criação do artefato, o agente o foca através da ação `focus(Id)`, assim as propriedades observáveis do artefato serão mapeadas como crenças no agente. Então o agente chama a operação `inc` do artefato `meuContador`, mapeada como uma ação interna, três vezes. Antes e após estas chamadas é usado o artefato padrão `Console` para chamar a operação `println`, escrevendo informações adicionais na tela.

Cada vez que o agente chama a operação `inc`, o valor da propriedade observável `contagem` é atualizado, gerando a adição da crença `+contagem(V)`, com `V` sendo o novo valor de `contagem`. O plano para a adição desta crença apenas imprime na tela a frase `“Valor do contador: ”` com o valor recebido em `V`.

---

```

1 !testar_contador.

3 +!testar_contador
  <- makeArtifact(“meuContador”, “Artefatos.Contador”, [], Id
    );
5   focus(Id);
   println(“Início do teste.”);
7   inc;
   inc;
9   inc;
   println(“Fim do teste.”).

11 +contagem(V): true
13 <- println(“Valor do contador: ”, V).
```

---

Abaixo segue a saída da utilização do artefato Contador pelo agente, no console do Jason:

---

```

1 [usuario_contador] Valor do contador: 0
2 [usuario_contador] Início do teste.
3 [usuario_contador] Valor do contador: 1
4 [usuario_contador] Valor do contador: 2
5 [usuario_contador] Valor do contador: 3
```

---

[usuario\_contador] Fim do teste.

---

Segue também o arquivo exemplo\_contador.mas2j, como exemplo do que o arquivo .mas2j deve ter para utilizar o Cartago. É necessário declarar que o ambiente do SMA é um CartagoEnvironment (ambiente Cartago) e que os agentes utilizam a arquitetura CAgentArch. Também é preciso definir o caminho onde se encontram os pacotes cartago.jar e c4jason.jar. C4Jason quer dizer “Cartago para Jason” e é o pacote que implementa a ponte entre as duas plataformas.

---

```

MAS exemplo_contador {
2   environment: c4jason.CartagoEnvironment

4   agents:
      usuario_contador agentArchClass c4jason.CAgentArch;

6   classpath: ‘ ‘.../ cartago -2.0.1/ lib/ cartago . jar ’ ;
8   ‘ ‘.../ cartago -2.0.1/ lib/ c4jason . jar ’ ;
}

```

---

### B.2.0.2 Artefato Relógio

O artefato implementado neste exemplo é um Relógio que lança um tick a cada 0.1 segundos. Este relógio pode ser inicializado e parado pelos agentes. O objetivo do exemplo é demonstrar como uma operação pode ser dividida em etapas e o funcionamento dos sinais.

O artefato Relógio começa sendo inicializado com a propriedade interna “funcionando” como false e não possui propriedades observáveis. Ele possui as operações “começar” e “parar” disponíveis para os agentes, que respectivamente começam e param a atividade do relógio. Ao ser executada a operação “começar” irá testar se o relógio já está funcionando, se sim ele lançará uma mensagem de falha chamando a primitiva “failed”, caso contrário “funcionando” será alterada para true, e a segunda etapa da operação, “contar”, será chamada, através da primitiva execInternalOp. “contar” faz com que o artefato lance um sinal “tick” a cada 0.1 segundos enquanto “funcionando” for true, a contagem de tempo é feita com a primitiva await\_time, que suspende a execução dessa etapa da operação até que o tempo definido nela tenha passado. A operação “parar” simplesmente altera “funcionando” para false, fa-

zendo com que quando a etapa “contar” for novamente executada após 0.1 segundos ela saia do loop.

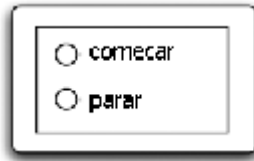


Figura 22 – Representação abstrata do Artefato Relógio

---

```

1 package Artefatos;
3 import cartago.*;
5 public class Relogio extends Artifact {
7     boolean funcionando;
8     final static long TEMPO_TICK = 100;
9
10    void init(){
11        funcionando = false;
12    }
13
14    @OPERATION void comenzar(){
15        if (!funcionando){
16            funcionando = true;
17            execInternalOp("contar");
18        } else {
19            failed("já está funcionando");
20        }
21    }
22
23    @OPERATION void parar(){
24        funcionando = false;
25    }
26
27    @INTERNAL_OPERATION void contar(){
28        while (funcionando){
29            signal("tick");
30            await_time(TEMPO_TICK);
31        }
32    }
33 }

```

---

Abaixo segue o código do agente Jason que utiliza Relógio. Primeiro ele cria uma instância “meuRelógio” do tipo “Relógio”, então foca essa instância para perceber os sinais que serão gerados, também cria a crença `n_ticks(0)` que será atualizada a cada tick do relógio, após isso o agente executa a operação “começar” do artefato. Cada vez que o artefato lançar um tick de relógio o sinal será capturado por um dos dois planos `+tick`. O primeiro plano é executado quando o agente acredita que 10 ticks já foram sinalizados, e então para o relógio. O segundo plano é executado enquanto a quantidade de ticks for menor que 10, e apenas atualiza a crença `n_ticks` e informa no console que um tick foi percebido.

---

```

1 !testar_relogio.

3 +!testar_relogio
  <- makeArtifact(‘‘meuRelógio”,‘‘Artefatos.Relógio”,[],Id);
5   focus(Id);
   +n_ticks(0);
7   começar;
   println(‘‘relógio começou.”).

9   +tick: n_ticks(10)
11  <- parar;
   println(‘‘relógio parou.”).

13
+tick: n_ticks(N)
15  <- ++n_ticks(N+1);
   println(‘‘tick percebido!”).

```

---

Abaixo segue a saída no MAS Console do Jason:

---

```

[usuario_relogio] relógio começou.
2 [usuario_relogio] tick percebido!
[usuario_relogio] tick percebido!
4 [usuario_relogio] tick percebido!
[usuario_relogio] tick percebido!
6 [usuario_relogio] tick percebido!
[usuario_relogio] tick percebido!
8 [usuario_relogio] tick percebido!
[usuario_relogio] tick percebido!
10 [usuario_relogio] tick percebido!
[usuario_relogio] tick percebido!
12 [usuario_relogio] relógio parou.

```

---

### B.2.0.3 Artefato Sincronizador

Este exemplo visa mostrar como se pode utilizar guardas para sincronizar agentes. O artefato Sincronizador serve para ser utilizado por uma equipe de agentes que precisam sincronizar antes de proceder com suas ações. Para isso o artefato possui a operação “sincronizar”, que deve ser ativada por cada um dos agentes do grupo que quiserem sincronizar.

Abaixo é mostrado o código no Cartago do artefato Sincronizador. Os atributos “nProntos” (quantidade de agentes prontos) e “nParticipantes” (quantidade de agentes participando da sincronização) são propriedades internas do artefato, invisíveis para os agentes. Estas propriedades são inicializadas no método “init”, que recebe a quantidade de participantes como parâmetro. A operação “sincronizar” primeiro aumenta o contador de agentes prontos, e então espera a guarda “todosProntos” ser liberada, essa espera é feita através da chamada `await` com o parâmetro “todosProntos”. Ao chamar `await` essa instância da operação “sincronizar” é suspensa, e o plano do agente que chamou esta operação também é suspenso. Quando todos os agentes tiverem ativado a operação “sincronizar” a guarda será liberada e a operação, e consequentemente o plano do agente, voltam a executar.



Figura 23 – Representação abstrata do Artefato Sincronizador

---

```

1 package Artefatos;
2
3 import cartago.*;
4
5 public class Sincronizador extends Artifact {
6
7     int nProntos, nParticipantes;
8
9     void init(int nParticipantes) {
10         nProntos = 0;
11         this.nParticipantes = nParticipantes;
12     }
13
14     @OPERATION

```

```

16     void sincronizar() {
17         nProntos++;
18         await(“ todosProntos”);
19     }
20     @GUARD
21     boolean todosProntos() {
22         return nProntos == nParticipantes;
23     }
24 }

```

---

Para demonstrar o funcionamento deste artefato foram criados cinco agentes Jason com o código abaixo. O objetivo inicial é !trabalhar, primeiro este plano cria o objetivo setupTools(Id) para preparar o artefato que irá utilizar, dessa forma primeiro o agente tenta chamar makeArtifact para criar o artefato “sincronizador”, caso este plano falhe porque o artefato já existe o agente usa a ação lookupArtifact para procurar o artefato chamado “sincronizador”. Após isso, o agente espera por um tempo entre 0 e 10 segundos, foca no artefato “sincronizador” e chama a operação “sincronizar”. Quando todos os agentes ativarem a operação “sincronizar” este plano voltará a executar e imprimirá na tela que a sincronização foi efetuada.

---

```

!trabalhar.
2
+!trabalhar
4   <- !setupTools(Id);
   println(“ Início da sincronização.”);
6   .random(N);
   N2 = N*10000;
8   .wait(N2);
   println(“ Trabalhei por ”,N2/1000, “ segundos.”);
10   focus(Id);
   println(“ Ativar operação sincronizar()”);
12   sincronizar;
   println(“ Sincronização efetuada!”);
14   println(“ Fim do teste.”).

16 +!setupTools(Id) : true <-
   makeArtifact(“ sincronizador”, “ Artefatos.Sincronizador”
   ,[5], Id).

18
20 -!setupTools(Id) : true <-
   lookupArtifact(“ sincronizador”, Id).

```

---

Abaixo segue a saída no MAS Console do Jason:

---

	[ usuario_sincr4 ]	Início da sincronização.
2	[ usuario_sincr2 ]	Início da sincronização.
	[ usuario_sincr3 ]	Início da sincronização.
4	[ usuario_sincr1 ]	Início da sincronização.
	[ usuario_sincr5 ]	Início da sincronização.
6	[ usuario_sincr4 ]	Trabalhei por 1.528724231200107 segundos.
	[ usuario_sincr4 ]	Ativar operação sincronizar()
8	[ usuario_sincr1 ]	Trabalhei por 5.256301264501251 segundos.
	[ usuario_sincr1 ]	Ativar operação sincronizar()
10	[ usuario_sincr5 ]	Trabalhei por 6.559502173560134 segundos.
	[ usuario_sincr5 ]	Ativar operação sincronizar()
12	[ usuario_sincr2 ]	Trabalhei por 6.648731124199021 segundos.
	[ usuario_sincr2 ]	Ativar operação sincronizar()
14	[ usuario_sincr3 ]	Trabalhei por 7.27029024339723 segundos.
	[ usuario_sincr3 ]	Ativar operação sincronizar()
16	[ usuario_sincr3 ]	Sincronização efetuada!
	[ usuario_sincr4 ]	Sincronização efetuada!
18	[ usuario_sincr5 ]	Sincronização efetuada!
	[ usuario_sincr2 ]	Sincronização efetuada!
20	[ usuario_sincr1 ]	Sincronização efetuada!
	[ usuario_sincr4 ]	Fim do teste.
22	[ usuario_sincr5 ]	Fim do teste.
	[ usuario_sincr3 ]	Fim do teste.
24	[ usuario_sincr2 ]	Fim do teste.
	[ usuario_sincr1 ]	Fim do teste.

---

#### B.2.0.4 Jantar dos Filósofos

Por último, este exemplo soluciona o Problema do Jantar dos Filósofos no Cartago. Este é um problema de programação concorrente proposto por Dijkstra em 1965 e é uma alegoria do que ocorre nos sistemas computacionais multiprocessados que disputam os mesmos recursos limitados e dependem de um algoritmo mestre ou de um sistema operacional para evitar a ocorrência de conflitos entre os processos concorrentes. Nele há cinco filósofos sentados ao redor de uma mesa sobre a qual estão cinco talheres. Diante de cada filósofo há uma refeição e os talheres estão dispostos nos dois lados de cada prato de forma que haja apenas um talher entre cada dois filósofos. Para comer, cada filósofo deve usar dois talheres. Enquanto não comem, os filósofos ficam pensando. Para solucionar este problema é necessário a utilização de um algoritmo que gerencie a janta de forma que nenhum filósofo passe fome.

O artefato utilizado para solucionar este problema é o TupleS-



pace (espaço de tuplas), este artefato pode ser utilizado sem ser instanciado, pois toda área de trabalho possui uma instância de TupleSpace, chamada “blackboard”, como artefato padrão. O código-fonte deste artefato não será traduzido, para que o exemplo demonstrado possa ser utilizado diretamente programando agentes no Jason+Cartago. As operações “inp” e “rdp” tiveram seus trechos de código escondidos, pois não serão utilizadas neste exemplo.

Abaixo segue o código do artefato TupleSpace, ele possui a propriedade interna “tset” que representa o conjunto de tuplas armazenadas no artefato, esse conjunto é inicializado no método “init”. A operação “out” apenas adiciona uma nova tupla, com seu nome e parâmetros, no espaço de tuplas. A operação “in” verifica se a tupla com o dado nome e parâmetros está no espaço de tupla, isso é feito através do uso de um template de tupla com os atributos da tupla procurada, então esse template é usado pela guarda “foundMatch”, que apenas retornará true quando esse template for encontrado no espaço de tuplas, até lá a operação fica suspensa. Após encontrar a tupla ela é removida do espaço de tuplas. A operação “rd” é semelhante a operação “in”, com a diferença de que ela apenas lê a tupla do espaço de tuplas, sem retirá-la.

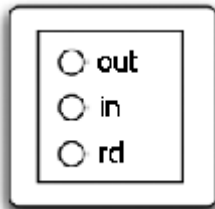


Figura 24 – Representação abstrata do Artefato TupleSpace

---

```

1 package cartago.tools;
2
3 import cartago.*;
4
5 public class TupleSpace extends Artifact {
6
7     TupleSet tset;
8
9     void init(){
10         tset = new TupleSet();
11     }
12
13     @OPERATION void out(String name, Object... args){

```

```

    tset.add(new Tuple(name, args));
15 }

17 @OPERATION void in(String name, Object... params){
    TupleTemplate tt = new TupleTemplate(name, params);
19     await(“foundMatch”,tt);
    Tuple t = tset.removeMatching(tt);
21     bind(tt,t);
    }

23 @OPERATION void inp(String name, Object... params){
25     (...)
    }

27 @OPERATION void rd(String name, Object... params){
29     TupleTemplate tt = new TupleTemplate(name, params);
    await(“foundMatch”,tt);
31     Tuple t = tset.readMatching(tt);
    bind(tt,t);
33 }

35 @OPERATION void rdp(String name, Object... params){
37     (...)
    }

39 private void bind(TupleTemplate tt, Tuple t){
    Object[] tparams = t.getContents();
41     int index = 0;
    for (Object p: tt.getContents()){
43         if (p instanceof OpFeedbackParam<?>){
            ((OpFeedbackParam) p).set(tparams[index]);
45         }
        index++;
47     }
    }

49 @GUARD boolean foundMatch(TupleTemplate tt){
51     return tset.hasTupleMatching(tt);
    }
53 }

```

---

Há dois tipos de agentes Jason que interagem com o artefato TupleSpace: um garçom e cinco filósofos. O código do agente garçom segue abaixo, ele possui crenças iniciais que definem o número, nome e garfos de cada um dos filósofos. No comando “for” são adicionadas tuplas (garfo, valor), sendo “valor” de 0 a 4 e tuplas (“filosofo.init”, Nome, Esquerda, Direita) para cada uma de suas crenças sobre os filósofos. Por fim são adicionadas quatro tuplas (ticket), que os filósofos precisam

obter para poder comer e servem para evitar deadlocks.

---

```

1 filosofo(0, "filosofo1", 0, 1).
  filosofo(1, "filosofo2", 1, 2).
3 filosofo(2, "filosofo3", 2, 3).
  filosofo(3, "filosofo4", 3, 4).
5 filosofo(4, "filosofo5", 4, 0).

7 !preparar_mesa.

9 +!preparar_mesa
  <- for ( .range(I,0,4) ) {
11     out("garfo", I);
        ? filosofo(I, Nome, Esquerda, Direita);
13     out("filosofo_init", Nome, Esquerda, Direita);
        };
15     for ( .range(I,1,4) ) {
        out("ticket");
17     };
        println("pronto. ").

```

---

Abaixo segue o código dos agentes filósofos. Primeiro o filósofo descobre seu nome através da ação `.my_name(Eu)` e então utiliza essa informação para retirar a tupla com seu nome, juntamente com a informação de quais são seus garfos, então o agente cria duas crenças para armazenar quais são seus garfos da direita e da esquerda e começa o plano `+!viver`. O plano `+!viver` consiste de um loop com os sub-objetivos `!pensar` e `!comendo`. `+!pensar` apenas imprime na tela que o filósofo está pensando. `+!comendo` consiste de três sub-objetivos: `!adquirirGarfos`, `!comer` e `!largarGarfos`. `+!adquirirGarfos` tenta obter um ticket e os garfos do filósofo retirando-os do espaço de tupla, caso ele não consiga porque já estão sendo usados, ele ficará esperando até serem liberados. `+!comer` apenas imprime na tela que o agente está comendo. `+!largarGarfos` coloca o ticket e os garfos novamente no espaço de tuplas, para que outros filósofos possam comer.

---

```

!comecar.

2
+!comecar
4   <- .my_name(Eu);
        in("filosofo_init", Eu, Esquerda, Direita);
6   +meu_garfo_esquerda(Esquerda);
        +meu_garfo_direita(Direita);
8   println(Eu, "pronto.");
        !!viver.

10
+!viver

```

```

12  <- !pensar;
      !comendo;
14      !! viver.

16  +!comendo
      <- !adquirirGarfos;
18      !comer;
      !largarGarfos.

20  +!adquirirGarfos :
22      meu_garfo_esquerda(F1) \& meu_garfo_direita(F2)
      <- in(“ticket”);
24      in(“garfo”,F1);
      in(“garfo”,F2).

26  +!largarGarfos :
28      meu_garfo_esquerda(F1) \& meu_garfo_direita(F2)
      <- out(“garfo”,F1);
30      out(“garfo”,F2);
      out(“ticket”).

32  +!pensar
34      <- println(“Pensando”).
      +!comer
36      <- println(“Comendo”).

```

---

Saída da execução do SMA no console do Jason:

---

```

[filosofo1] filosofo1 pronto.
2 [filosofo2] filosofo2 pronto.
[filosofo3] filosofo3 pronto.
4 [filosofo1] Pensando
[filosofo4] filosofo4 pronto.
6 [filosofo5] filosofo5 pronto.
[waiter] pronto.
8 [filosofo2] Pensando
[filosofo5] Pensando
10 [filosofo5] Comendo
[filosofo2] Comendo
12 [filosofo5] Pensando
[filosofo2] Pensando
14 [filosofo3] Pensando
[filosofo1] Comendo
16 [filosofo4] Pensando
[filosofo1] Pensando
18 [filosofo1] Comendo
[filosofo3] Comendo
20 [filosofo3] Pensando
[filosofo5] Comendo
22 [filosofo2] Comendo

```

```

[ filosofo1 ] Pensando
24 [ filosofo2 ] Pensando
[ filosofo4 ] Comendo
26 ( ... )

```

---

### B.3 COMPARAÇÃO ENTRE AMBIENTES JASON E CARTAGO

Nesta seção é feita uma comparação entre um ambiente Jason e um ambiente Cartago no contexto do “Robôs de Limpeza” (cleaning robots). Nesse SMA existem dois agentes, r1 e r2, o primeiro tem como objetivo andar por todo o mundo de Marte a procura de lixo, ao encontrar esse lixo r1 leva-o para r2, fixo em uma posição do mapa, que o queima, então r1 volta para o local em que estava antes e continua sua busca.

Primeiro os códigos dos agentes Jason são mostrados e explicados, então são mostrados os ambientes Jason e Cartago para esses mesmos agentes em uma versão adaptada, após isso é mostrada a saída no console do Jason para a execução destes SMAs e por fim, as conclusões desta comparação. Os agentes e ambiente Jason estão no capítulo 2 do manual do Jason, sendo que o ambiente foi traduzido e modificado para não ter a interface gráfica e ser mais fácil compreendê-lo e compará-lo com o ambiente Cartago.

Segue abaixo o código do agente r1. O agente só possui a regra “emP”, que conclui se r1 está no mesmo local de P. O objetivo inicial do agente é “chechar”. O plano deste objetivo é um loop que fica ativo enquanto o agente não encontrar lixo ou enquanto ele não terminar de checar o mundo, esse loop faz com que r1 vá para o próximo slot. O evento de adição de percepção +lixo(r1) indica que foi encontrado lixo, e faz o agente ter o objetivo de carregar o lixo, caso ele ainda não o esteja carregando. O plano para o evento +!carregar\_para(R) faz com que o agente guarde sua posição atual, crie o objetivo de pegar o lixo (e carregá-lo para r2), e após isso volte para a posição que foi armazenada e continue a checar o ambiente. O plano para +!pegar(S,L) cria os objetivos !assegurar\_pegar(S), que garante que o agente conseguirá pegar o lixo, e !ir\_para(L), que faz com que r1 vá para a posição onde está r2, e por fim executa a ação largar, para que r2 possa pegar o lixo.

---

```
// Agente r1
```

```

4  /* Initial beliefs and rules */
em_local(P) :- pos(P,X,Y) \& pos(r1,X,Y).
6
7  /* Initial goals */
8  !checar(slots).
10
11 /* Plans */
12 +!checar(slots) : not lixo(r1) \& not terminou
14   <- prox(slot);
   !!checar(slots).
16 +!checar(slots).

18 +lixo(r1) : not .desire(carregar_para(r2))
   <- !carregar_para(r2).
20
21 +!carregar_para(R)
22   <- // lembra para onde tem que voltar
   ?pos(r1,X,Y);
24   -+pos(last,X,Y);

26   // carrega o lixo para r2
   !pegar(lixo,R);
28
29   // volta e continua checando o mundo
30   !ir_para(last);
   !!checar(slots).
32
33 +!pegar(S,L) : true
34   <- !assegurar_pegar(S);
   !ir_para(L);
36   largar(S).

38 +!assegurar_pegar(S) : lixo(r1)
   <- pegar(lixo);
   !assegurar_pegar(S).
40 +!assegurar_pegar(_).
42
43 +!ir_para(L) : em_local(L).
44 +!ir_para(L) <- ?pos(L,X,Y);
   ir_para(X,Y);
46   !ir_para(L).

48 +terminou <- .println("Conclui a busca").

```

---

O código do agente r2 é um único plano de que é ativado quando o agente percebe que há lixo na sua posição, trazido por r1. r2 simplesmente efetua a ação de queimar este lixo.

---

```

// Agente r2
2
/* Plans */
4 +lixo(r2) : true <- queimar(lixo).

```

---

O código do ambiente Jason é mostrado abaixo. Os ambientes em Jason são organizados com uma classe ambiente principal (MarsEnv) e classes auxiliares para o modelo do ambiente (MarsModel) e a interface gráfica do ambiente (foi omitida neste exemplo). A classe principal possui as declarações globais e os métodos init, executado ao iniciar o sistema, e executeAction, que realiza os efeitos das ações executadas pelos agentes no ambiente chamando o método correspondente na classe modelo e chama o método auxiliar updatePercepts, que atualiza as percepções fornecidas pelo ambiente de acordo com o seu estado. A classe modelo inicializa a localização das entidades dos ambientes e possui os métodos que correspondem as ações efetuadas pelos agentes, além de atributos que auxiliam estes métodos.

Em relação aos métodos que representam ações na classe modelo: O método “proxSlot()” altera a posição do agente no mundo para a próxima posição; o método “irPara(x,y)” altera a posição do agente de forma a aproximá-la da posição (x,y); o método “pegarLixo()”, caso haja lixo na posição do agente, testa se o agente conseguiu pegar o lixo, e se sim altera o atributo auxiliar “r1TemLixo” para true e remove o lixo desta posição no modelo; o método “largarLixo()” testa se o agente está carregando lixo, e se sim adiciona lixo na posição do agente e altera o atributo auxiliar “r1TemLixo” para false; o método “queimarLixo()” faz com que o lixo seja removido do mundo, caso haja lixo na posição do agente r2.

---

```

import jason.asSyntax.*;
2 import jason.environment.Environment;
import jason.environment.grid.GridWorldModel;
4 import jason.environment.grid.Location;

6 import java.util.Random;
import java.util.logging.Logger;
8
public class MarsEnv extends Environment {
10
    public static final int GSize = 7; // tamanho da grade
12    public static final int LIXO = 16; // código do lixo no
        modelo de grade

```

```

14     public static final Term    ns = Literal.parseLiteral(“
        prox(slot)”);
    public static final Term    pg = Literal.parseLiteral(“
        pegar(lixo)”);
16     public static final Term    dg = Literal.parseLiteral(“
        largar(lixo)”);
    public static final Term    bg = Literal.parseLiteral(“
        queimar(lixo)”);
18     public static final Literal g1 = Literal.parseLiteral(“
        lixo(r1)”);
    public static final Literal g2 = Literal.parseLiteral(“
        lixo(r2)”);

20
    private Logger logger = Logger.getLogger(“marsEnvJason.
        ”+MarsEnv.class.getName());

22
    private MarsModel modelo;

24
    /** Chamado antes da execuo do SMA com os args
        informados no .mas2j */
26     @Override
    public void init(String[] args) {
28         modelo = new MarsModel();
        updatePercepts();
30     }

32     @Override
    public boolean executeAction(String ag, Structure action
    ) {
34         logger.info(ag+“ executando: ”+ action);
        try {
36             if (action.equals(ns)) {
                modelo.proxSlot();
38             } else if (action.getFunctor().equals(“ir_para”
                )) {
                int x = (int)((NumberTerm)action.getTerm(0))
                    .solve();
40                int y = (int)((NumberTerm)action.getTerm(1))
                    .solve();
                modelo.irPara(x,y);
42            } else if (action.equals(pg)) {
                modelo.pegarLixo();
44            } else if (action.equals(dg)) {
                modelo.largarLixo();
46            } else if (action.equals(bg)) {
                modelo.queimarLixo();
48            } else {
                return false;
50            }
        } catch (Exception e) {
52            e.printStackTrace();
        }
    }

```



```

54         updatePercepts();
56
57         try {
58             Thread.sleep(200);
59         } catch (Exception e) {}
60         return true;
61     }
62
63     /** cria as percepções dos agentes baseadas no MarsModel
64         */
65     void updatePercepts() {
66         clearPercepts();
67
68         Location r1Loc = modelo.getAgPos(0);
69         Location r2Loc = modelo.getAgPos(1);
70
71         Literal pos1 = Literal.parseLiteral("pos(r1," +
72             r1Loc.x + "," + r1Loc.y + ")");
73         Literal pos2 = Literal.parseLiteral("pos(r2," +
74             r2Loc.x + "," + r2Loc.y + ")");
75
76         addPercept(pos1);
77         addPercept(pos2);
78
79         if (modelo.hasObject(LIXO, r1Loc)) {
80             addPercept(g1);
81         }
82         if (modelo.hasObject(LIXO, r2Loc)) {
83             addPercept(g2);
84         }
85
86         if (modelo.buscaTerminou == true) {
87             addPercept(Literal.parseLiteral("terminou"));
88         }
89     }
90
91     class MarsModel extends GridWorldModel {
92
93         public static final int MErr = 2; // max erros ao
94             pegar o lixo
95         int nerr; // número de tentativas de pegar o lixo
96         boolean r1TemLixo = false; // se r1 está carregando
97             lixo ou não
98         boolean buscaTerminou = false;
99
100         Random random = new Random(System.currentTimeMillis());
101
102         private MarsModel() {
103             super(GSize, GSize, 2);

```

```

100         // localização inicial dos agentes
101     try {
102         setAgPos(0, 0, 0);
103
104         Location r2Loc = new Location(GSize/2, GSize
105                                     /2);
106         setAgPos(1, r2Loc);
107     } catch (Exception e) {
108         e.printStackTrace();
109     }
110
111     // localização inicial do lixo
112     add(LIXO, 3, 0);
113     add(LIXO, GSize-1, 0);
114     add(LIXO, 1, 2);
115     add(LIXO, 0, GSize-2);
116     add(LIXO, GSize-1, GSize-1);
117 }
118
119 void proxSlot() throws Exception {
120     Location r1 = getAgPos(0);
121     r1.x++;
122     if (r1.x == getWidth()) {
123         r1.x = 0;
124         r1.y++;
125     }
126     // termina de procurar na grade
127     if (r1.y == getHeight()) {
128         buscaTerminou = true;
129         return;
130     }
131     setAgPos(0, r1);
132 }
133
134 void irPara(int x, int y) throws Exception {
135     Location r1 = getAgPos(0);
136     if (r1.x < x)
137         r1.x++;
138     else if (r1.x > x)
139         r1.x--;
140     if (r1.y < y)
141         r1.y++;
142     else if (r1.y > y)
143         r1.y--;
144     setAgPos(0, r1);
145 }
146
147 void pegarLixo() {
148     // local de r1 tem lixo
149     if (modelo.hasObject(LIXO, getAgPos(0))) {
150         // as vezes a ação “pegar” são funciona
151         // mas nunca mais que MErr vezes

```

```

152         if (random.nextBoolean() || nerr == MErr) {
            remove(LIXO, getAgPos(0));
            nerr = 0;
154         r1TemLixo = true;
        } else {
156         nerr++;
        }
158     }
    }
160    void largarLixo() {
        if (r1TemLixo) {
162        r1TemLixo = false;
            add(LIXO, getAgPos(0));
164        }
    }
166    void queimarLixo() {
        // local de r2 tem lixo
168        if (modelo.hasObject(LIXO, getAgPos(1))) {
            remove(LIXO, getAgPos(1));
170        }
    }
172 }
}

```

---

Para utilizar o artefato MarsEnv do ambiente Cartago os agentes r1 e r2 sofreram mais adaptações do que esperado inicialmente. Abaixo as principais adaptações serão explicadas e, após isso, primeiro é mostrado o código do ambiente Cartago e então são mostradas as versões adaptadas dos agentes.

A primeira adaptação decorre do fato que não existe nenhum tipo de controle centralizado padrão sobre as operações dos artefatos, como ocorre no método “executeAction” do ambiente Jason, que centraliza o controle tanto da execução de ações como da geração de percepções. Em Cartago, cada operação equivale a uma ação, e cada uma é capaz de gerar percepções. Embora um controle de ações e percepções centralizado possa ser forjado, acredito que isso vá contra os princípios do modelo de A&A. Então nesta primeira adaptação, cada operação para uma ação do artefato MarsEnv foi baseada no método para essa mesma ação no modelo do ambiente Jason. Após isso foi analisado quais e que tipos de percepções (persistentes ou não) cada operação deveria lançar. Para ter um log equivalente ao do ambiente anterior, cada operação chama o método “printLog” assim que começa a executar.

A segunda adaptação é relacionada às percepções geradas pelo ambiente. No ambiente Jason as percepções são persistentes, pois todas elas são atualizadas após o comando clearPercepts(). O ambiente

Cartago pode gerar eventos persistentes através das propriedades observáveis ou eventos não-persistentes emitindo sinais. Porém no Cartago não pode haver duas propriedades observáveis com um mesmo nome, o que gerou limitações para simplesmente se adaptar todos os eventos persistentes do ambiente Jason em propriedades observáveis no ambiente Cartago.

Por exemplo, no primeiro ambiente existem várias percepções na estrutura `pos(nome,X,Y)`, como o agente tem que saber constante qual é sua posição escolheu-se fazê-las com propriedades observáveis no ambiente Cartago, com a estrutura `pos_nome(X,Y)` para superar a limitação de não poder ter duas propriedades de mesmo nome, também foram criados atributos `pos_nome`, por ser uma informação usada com frequência e assim evitar poluição no código. A percepção `lixo(nome)` no ambiente original tinha tanto a função de informar que na posição do agente tinha lixo, como informar que o agente está carregando lixo. No ambiente Cartago essa percepção foi dividida em duas, para que pudesse haver duas percepções com o mesmo nome e para que o tipo de percepção esteja melhor relacionado a sua função. A primeira função se tornou o sinal `lixo(nome)`, já que o agente só precisa saber uma vez se tem lixo na posição em que ele está; a segunda função se tornou a propriedade observável `lixo_com_r1(boolean)`, para que o agente possa sempre saber se está ou não carregando lixo, e agir de acordo. Outra alternativa seria mapear a percepção `lixo(nome)` para uma propriedade observável `lixo_nome(boolean)`, o que teria deixado o código do agente adaptado mais similar ao original. Uma última alternativa seria caso houvesse um método `removeObsProperty`, de forma a permitir de uma maneira mais direta a existência de percepções persistentes por apenas um determinado período.

Uma outra adaptação relevante é que todas as variáveis que indicam nome de algo, como `r1` e `r2`, tiveram que ser trocadas para as strings `"r1"` e `"r2"`, porque essa é a forma de uma operação de um artefato enviar nomes para os agentes.

O método `"init"` também teve que ser adaptado. No ambiente original o próprio método de inicialização já era capaz de lançar eventos persistentes e não-persistentes, e esses eventos eram detectados pelos agentes. Porém no caso dos artefatos, primeiro um agente o cria e só depois este agente pode ou não focá-lo, ou seja, sinais emitidos no método `"init"` não podem ser detectados pelos agentes. Por causa disso, percepções que devem existir desde a criação do artefato independente das suas operações, como as de posição dos agentes, devem ser criadas como percepções persistentes, propriedades observáveis, no

método “init”.

A última adaptação é fazer com que o primeiro objetivo dos agentes r1 e r2 seja o objetivo inicializar. No caso de r1, esse objetivo chama a ação “makeArtifact” para criar uma instância do artefato MarsEnv, então foca a instância criada e chama o objetivo inicial anterior de r1, !checar. Para r2 o objetivo inicializar usa a ação “lookupArtifact” para encontrar a instância criada por r1, e então a foca.

Segue abaixo a imagem abstrata do artefato MarsEnv e o código deste artefato em um ambiente Cartago.

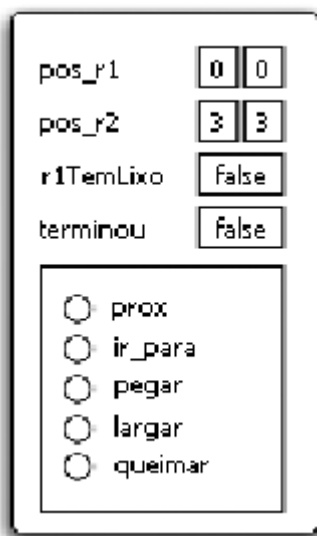


Figura 25 – Representação abstrata do Artefato MarsEnv

---

```

1 import cartago.*;
  import java.util.ArrayList;
3 import java.util.Random;

5 public class MarsEnv extends Artifact {

7     final int GSize = 7;
      Posicao pos_r1;
9     Posicao pos_r2;
      ArrayList<Posicao> pos_lixos;

```

11

```

Random random = new Random(System.currentTimeMillis());
13 public static final int MErr = 2; // max erros ao pegar o
    lixo
        int nerr; // número de tentativas de pegar o
            lixo
15
void init() {
17     // definir posição inicial dos agentes
    pos_r1 = new Posicao(0,0);
19     pos_r2 = new Posicao(GSize/2, GSize/2);

21     // definir posição inicial dos lixos
    pos_lixos = new ArrayList<Posicao>();
23     pos_lixos.add(new Posicao(3,0));
    pos_lixos.add(new Posicao(GSize-1, 0));
25     pos_lixos.add(new Posicao(1,2));
    pos_lixos.add(new Posicao(0, GSize-2));
27     pos_lixos.add(new Posicao(GSize-1, GSize-1));

29     // percepções iniciais
    defineObsProperty("pos_r1",pos_r1.x, pos_r1.y);
31     defineObsProperty("pos_r2",pos_r2.x, pos_r2.y);
    defineObsProperty("r1TemLixo",false);
33     defineObsProperty("terminou",false);
}

35 @OPERATION
37 void prox(String slot)
{
39     printLog("prox("+slot+"");

41     // termina de procurar na grade
    if(pos_r1.x == 0 && pos_r1.y == GSize)
43     {
        getObsProperty("terminou").updateValue(
            true);
45         return;
    }

47     pos_r1.x++;
49     if (pos_r1.x == GSize) {
        pos_r1.x = 0;
51         pos_r1.y++;
    }

53     getObsProperty("pos_r1").updateValues(pos_r1.x,
        pos_r1.y);

55     signal("pos_r1," + pos_r1.x + "," + pos_r1.y + "
        ");
57     if (temLixoEm(pos_r1))
    {

```

```

59         signal("lixo", "r1");
60     }
61 }

62 @OPERATION
63 void ir_para(int x, int y)
64 {
65     printLog("ir_para("+x+", "+y+")");
66     if (pos_r1.x < x)
67         pos_r1.x++;
68     else if (pos_r1.x > x)
69         pos_r1.x--;
70     if (pos_r1.y < y)
71         pos_r1.y++;
72     else if (pos_r1.y > y)
73         pos_r1.y--;
74
75     getObsProperty("pos_r1").updateValues(pos_r1
76         .x, pos_r1.y);
77 }

78 @OPERATION
79 void pegar(String lixo)
80 {
81     printLog("pegar("+lixo+")");
82     // local de r1 tem lixo
83     if (temLixoEm(pos_r1)) {
84         // as vezes a ação "pegar" são funciona
85         // mas nunca mais que MErr vezes
86         if (random.nextBoolean() || nerr == MErr
87             ) {
88             removerLixoEm(pos_r1);
89             nerr = 0;
90             getObsProperty("r1TemLixo").
91                 updateValues(true);
92         } else {
93             nerr++;
94         }
95     }
96 }

97 @OPERATION
98 void largar(String lixo)
99 {
100     printLog("largar("+lixo+")");
101     if (getObsProperty("r1TemLixo").booleanValue()) {
102         getObsProperty("r1TemLixo").updateValues(
103             false);
104         pos_lixos.add(new Posicao(pos_r1.x, pos_r1
105             .y));
106         if (pos_r1.x == pos_r2.x && pos_r1.y ==
107             pos_r2.y)

```

```

105             signal("lixo", "r2");
106         }
107     }
108
109     @OPERATION
110     void queimar(String lixo)
111     {
112         printLog("queimar("+lixo+")");
113         // local de r2 tem lixo
114         if (temLixoEm(pos_r2))
115             removerLixoEm(pos_r2);
116     }
117
118     private boolean temLixoEm(Posicao p)
119     {
120         for(Posicao pos: pos_lixos)
121             if(pos.x == p.x && pos.y == p.y)
122                 return true;
123         return false;
124     }
125
126     private void removerLixoEm(Posicao p)
127     {
128         for(Posicao pos: pos_lixos)
129             if(pos.x == p.x && pos.y == p.y)
130             {
131                 pos_lixos.remove(pos);
132                 break;
133             }
134     }
135
136     private void printLog(String acao)
137     {
138         System.out.println(getOpUserName()+" executando: "+ acao);
139     }
140
141     class Posicao
142     {
143         int x;
144         int y;
145
146         // ...
147     }
148 }

```

---

Código do agente r1 adaptado para usar o artefato MarsEnv, em negrito estão as partes alteradas:

---

```
// Agente r1
```



```

2      /* Initial beliefs and rules */
4      /* Initial goals */
6      !inicializar.
8      /* Plans */
10     +!inicializar
12     <- makeArtifact("mars","MarsEnv",[ ],Id);
        focus(Id);
14     !checar(slots).

16 +!checar(slots) : not lixo("r1") \& not r1TemLixo(true) \&
        not terminou(true)
        <- prox(slot);
18     !!checar(slots).
        +!checar(slots).
20
        +lixo("r1") : not .desire(carregar_para(-,-)) \& not lixo("
            r1")
22     <- +lixo("r1");
        ?pos_r2(X,Y);
24     !carregar_para(X,Y).
        +lixo("r1").
26
        +!carregar_para(A,B) : lixo("r1")
28     <- // lembra para onde tem que voltar
        ?pos_r1(X,Y);
30
        // carrega o lixo para r2
32     !pegar(lixo,A,B);

34     // volta e continua checando o mundo
        !ir_para(X,Y);
36     !!checar(slots).

38 +!pegar(S,X,Y) : true
        <- !assegurar_pegar(S);
40     !ir_para(X,Y);
        largar(S).
42
        +!assegurar_pegar(_) : r1TemLixo(true) <- -lixo("r1").
44 +!assegurar_pegar(S) : lixo("r1")
        <- pegar(lixo);
46     !assegurar_pegar(S).

48 +!ir_para(X,Y) : pos_r1(X,Y).
        +!ir_para(X,Y) <- ir_para(X,Y);
50                                     !ir_para(X,Y).

```

```
52 +terminou(true) : .println("Conclui a busca").
```

---

Código do agente r2 adaptado para usar o artefato MarsEnv, em negrito estão as partes alteradas:

---

```
// Agente r2
2
/* Initial beliefs and rules */
4
/* Initial goals */
6 !inicializar.

8 /* Plans */
+!inicializar <- lookupArtifact("mars",Id);
10         focus(Id).
-!inicializar <-wait(50);
12         !inicializar.

14 +lixo("r2") : true <- queimar(lixo).
```

---

Abaixo são mostrados parte das saídas no Console Jason dos SMAs executados com ambiente Jason (acima) e ambiente Cartago (abaixo). Como pode se notar pelas saídas, ambos os SMAs tiveram o mesmo comportamento, apenas variando um pouco na quantidade de mensagens repetidas enviadas, devido as diferenças do código.

---

```
[MarsEnv] r1 executando: prox(slot)
2 [MarsEnv] r1 executando: prox(slot)
[MarsEnv] r1 executando: pegar(lixo)
4 [MarsEnv] r1 executando: ir_para(3,3)
[MarsEnv] r1 executando: ir_para(3,3)
6 [MarsEnv] r1 executando: ir_para(3,3)
[MarsEnv] r1 executando: largar(lixo)
8 [MarsEnv] r1 executando: ir_para(3,0)
[MarsEnv] r1 executando: ir_para(3,0)
10 [MarsEnv] r2 executando: queimar(lixo)
[MarsEnv] r1 executando: ir_para(3,0)
12 [MarsEnv] r1 executando: prox(slot)
[MarsEnv] r1 executando: prox(slot)
14 [MarsEnv] r1 executando: prox(slot)
[MarsEnv] r1 executando: pegar(lixo)
16 [MarsEnv] r1 executando: pegar(lixo)
[MarsEnv] r1 executando: pegar(lixo)
18 [MarsEnv] r1 executando: ir_para(3,3)
[MarsEnv] r1 executando: ir_para(3,3)
20 [MarsEnv] r1 executando: ir_para(3,3)
[MarsEnv] r1 executando: largar(lixo)
```

```

22 [MarsEnv] r1 executando: ir_para(6,0)
[MarsEnv] r1 executando: ir_para(6,0)
24 [MarsEnv] r2 executando: queimar(lixo)
[MarsEnv] r1 executando: ir_para(6,0)
26 [MarsEnv] r1 executando: prox(slot)
[MarsEnv] r1 executando: prox(slot)
28 [MarsEnv] r1 executando: prox(slot)
...
30 [MarsEnv] r1 executando: prox(slot)
[MarsEnv] r1 executando: pegar(lixo)
32 [MarsEnv] r1 executando: ir_para(3,3)
[MarsEnv] r1 executando: ir_para(3,3)
34 [MarsEnv] r1 executando: ir_para(3,3)
[MarsEnv] r1 executando: largar(lixo)
36 [MarsEnv] r1 executando: ir_para(6,6)
[MarsEnv] r1 executando: ir_para(6,6)
38 [MarsEnv] r2 executando: queimar(lixo)
[MarsEnv] r1 executando: ir_para(6,6)
40 [MarsEnv] r1 executando: prox(slot)
[r1] Conclui a busca

```

---

```

1 r1 executando: prox(slot)
r1 executando: prox(slot)
3 r1 executando: prox(slot)
r1 executando: pegar(lixo)
5 r1 executando: pegar(lixo)
r1 executando: ir_para(3,3)
7 r1 executando: ir_para(3,3)
r1 executando: ir_para(3,3)
9 r1 executando: largar(lixo)
r2 executando: queimar(lixo)
11 r1 executando: ir_para(3,0)
r1 executando: ir_para(3,0)
13 r1 executando: ir_para(3,0)
r1 executando: prox(slot)
15 r1 executando: prox(slot)
r1 executando: prox(slot)
17 r1 executando: pegar(lixo)
r1 executando: pegar(lixo)
19 r1 executando: pegar(lixo)
r1 executando: ir_para(3,3)
21 r1 executando: ir_para(3,3)
r1 executando: ir_para(3,3)
23 r1 executando: largar(lixo)
r1 executando: ir_para(6,0)
25 r2 executando: queimar(lixo)
r1 executando: ir_para(6,0)
27 r1 executando: ir_para(6,0)
r1 executando: prox(slot)
29 ...

```

```

    r1 executando: prox(slot)
31 r1 executando: pegar(lixo)
    r1 executando: ir_para(3,3)
33 r1 executando: ir_para(3,3)
    r1 executando: ir_para(3,3)
35 r1 executando: largar(lixo)
    r2 executando: queimar(lixo)
37 r1 executando: ir_para(6,6)
    r1 executando: ir_para(6,6)
39 r1 executando: ir_para(6,6)
    r1 executando: prox(slot)
41 r1 executando: prox(slot)
    [r1] Conclui a busca

```

---

Por esta comparação pode-se perceber que o código do ambiente Cartago ficou menor, e também por isso, mais fácil de compreender do que o código do ambiente Jason. Isso também ocorreu em consequência do ambiente Jason possuir uma classe principal e uma classe modelo, o que poderia deixar o código mais organizado para ambientes grandes, mas tem a consequência de deixar o código dividido em partes e mais difícil de entender.

Uma conclusão relevante é que o ideal é que um ambiente Cartago seja modelado desde o princípio para ser estruturado em artefatos, de forma descentralizada e levando em consideração a existência de percepções persistentes e não-persistentes. Pois converter um ambiente modelado em Jason para um ambiente Cartago é trabalhoso e cheio de detalhes, como pode-se perceber comparando as duas versões do ambiente e dos agentes mostrados nesta seção, isso mesmo em uma conversão para um ambiente bastante simples como o MarsEnv.

## **APÊNDICE C – Informações adicionais sobre o Desenvolvimento**



## C.1 LISTA DE FILTROS

No filtro para classes simples, as classes filtradas foram: dbpedia-owl: PeriodicalLiterature, dbpedia-owl: Album, dbpedia-owl: VideoGame, dbpedia-owl: TelevisionEpisode, dbpedia-owl: TelevisionShow, dbpedia-owl: Film, dbpedia-owl: Band, yago: LivingPeople, yago: Gathering107975026, yago: Publication106589574 e umbel: TVShow\_SingleCW.

No filtro para classes avançadas, as classes filtradas foram: yago: Event100029378, yago: WebSite106359193, yago: Award106696483, yago: Professorship100598056, yago: Medium106254669, yago: HotSpot108586330, yago: Contestant109613191, yago: Trainer110722575, yago: MemoryDevice103744840, yago: Professional110480253, yago: Person100007846 e yago: Music107020895.

No filtro para categorias simples, as categorias filtradas foram: Sports, Comics, Albums, Television\_series e College\_football\_seasons

No filtro para categorias avançadas, as categorias filtradas foram: Clubs\_and\_societies, Economies\_by\_country, Hunting, Sports\_teams\_by\_country, Companies, Musical\_compositions, University\_and\_college\_sports\_clubs\_by\_country, Hospitals, National\_Register\_of\_Historic\_Places, Deaths\_from\_disease e National\_Register\_of\_Historic\_Places\_by\_county.

## C.2 CÓDIGOS DA ABORDAGEM POR CLASSES

### C.2.1 Código do agente

---

```

1  !observe.
2
3  +!observe : true
4  <- .wait(200);
5      ?myTools(A1, A2, Id); // discover the tools
6      println("agente criado");
7      !consumeItems(Id).
8
9  +!consumeItems(Id): true
10 <- .wait(1000);
11 get_para_Classes(Item);
12 !processItem(Item, Id);
13 !!consumeItems(Id).

```

```

14  +!processItem(null, Id) : true
16  <- .my_name(Me).
    //println(Me," : null").
18
19  +!processItem(Keyword, Id) : true
20  <-  executar_abrangente(Keyword, 30, Result1)[artifact_id(Id)
    ];
    executar_especifico(Result1, Result2)[artifact_id(Id)];
22  isEmpty(Result2, Test)[artifact_id("entrada")];
    !decidir(Result1, Result2, Test, Id).
24
25  +!decidir(Result1, Result2, Test, Id) : Test
26  <- .my_name(Me); println(Me," : vazio");
    executar_getIndividuos(Result1, 20, Result3)[artifact_id(
        Id)];
28  put_less_value(Result3);
    put(Result3).
30
31  +!decidir(Result1, Result2, Test, Id) : true
32  <- .my_name(Me); println(Me," : não vazio");
    executar_getIndividuos(Result2, 20, Result3)[artifact_id(
        Id)];
34  put_less_value(Result3);
    put(Result3).
36
37  +?myTools(A1, A2, A3): true
38  <- lookupArtifact("entrada",A1);
    lookupArtifact("saida",A2);
40  .my_name(N);
    lookupArtifact(N,A3).
42
43  -?myTools(A1, A2, A3): true
44  <- .wait(100);
    ?myTool(A1, A2, A3).

```

---

## C.2.2 Código da busca abrangente

---

```

1  @OPERATION
    void executar_abrangente(String keywordrecebida, int
        limit, OpFeedbackParam<Object> result)
3  {
        System.out.println("LOG: Abordagem por Classes
            STARTED");
5
        String keyword = NLPTool.toLowerCase(keywordrecebida
            );

```



```

7      String queryString = AbordagemPorIndividuo.
        getPrefixes()
9          + "select distinct ?Concept where {"
          + "[] a ?Concept ."
11         + " ?Concept rdfs:label ?title ."
          + " ?title <bif:contains> \""+keyword+"\" ."
13         + filtrarClasses(AbordagemPorIndividuo.
            classes_filtradas , 5)
          + "} LIMIT "+limit;

15
16      QueryExecution qe = new QueryEngineHTTP("http://
        dbpedia.org/sparql", queryString);
17
18      ArrayList<String> classes = new ArrayList<String>();
19
20      try
21      {
22          ResultSet results = qe.execSelect();
23          for (; results.hasNext(); )
24          {
25              QuerySolution s = results.next();
                classes.add(s.getResource("Concept").
                    toString());
27          }
28      }
29      finally
30      {
31          qe.close();
32      }
33
34      result.set(classes);
35
36      System.out.println("LOG: Abordagem por Classes ENDED
        : "+classes.size());
37  }

```

---

```

1  private String filtrarClasses(ArrayList<String> classes , int
    t_max)
    {
3      String option = "option(TRANSITIVE, t_distinct ,
        T_MAX("+t_max+"))";
        String result = "";
5      for(String s: classes)
        result += "filter (NOT EXISTS {?Concept rdfs:
            subClassOf "+s+" "+option+" }) .";
7      return result;
    }

```

---

## C.3 CÓDIGOS DA ABORDAGEM POR CATEGORIAS

### C.3.1 Código do agente

---

```

!observe.
2
+!observe : true
4  <- .wait(200);
    ?myTools(A1, A2, Id); // discover the tools
6    println("agente criado");
    !consumeItems(Id).
8
+!consumeItems(Id): true
10 <- .wait(1000);
    get_para_Categorias(Item);
12 !processItem(Item, Id);
    !!consumeItems(Id).
14
+!processItem(null, Id) : true.
16
+!processItem(Keyword, Id) : true
18 <-  executar_abrangente(Keyword, 30, Result1)[artifact_id(Id)
    ]];
    executar_especifico(Result1, Result2)[artifact_id(Id)];
20 isEmpty(Result2, Test)[artifact_id("entrada")];
    !decidir(Result1, Result2, Test, Id).
22
+!decidir(Result1, Result2, Test, Id) : Test
24 <- .my_name(Me); println(Me," : vazio");
    executar_getIndividuos(Result1, 20, Result3)[artifact_id(
        Id)];
26 put(Result3).

28 +!decidir(Result1, Result2, Test, Id) : true
    <- .my_name(Me); println(Me," : não vazio");
30 executar_getIndividuos(Result2, 20, Result3)[artifact_id(
        Id)];
    put(Result3).
32
+?myTools(A1, A2, A3): true
34 <- lookupArtifact("entrada",A1);
    lookupArtifact("saida",A2);
36 .my_name(N);
    lookupArtifact(N,A3).
38
-?myTools(A1, A2, A3): true
40 <- .wait(100);
    ?myTool(A1, A2, A3).

```

---

### C.3.2 Código da busca abrangente

---

```

1 @OPERATION
  void executar_abrangente(String keyphrase, int limit,
    OpFeedbackParam<Object> res)
3  {
    System.out.println("LOG: Abordagem por Categoria
      Abrangente STARTED");
5    int t_max = 6;
    ArrayList<String> result = new ArrayList<String>();
7    boolean success = false;
    while(!success)
9    {
        try
11        {
            result = executar_abrangente2(keyphrase,
              t_max, limit);
13            success = true;
        } catch (Exception e)
15        {
            t_max-=2;
17            if(t_max == 0)
            {
19                res.set(new ArrayList<String>());
                System.out.println("LOG: Abordagem por
                  Categoria Abrangente ENDED: 0");
21                return;
            }
23        }
    }
25    res.set(result);
    System.out.println("LOG: Abordagem por Categoria
      Abrangente ENDED: "+result.size());
27 }

```

---

```

1 public ArrayList<String> executar_abrangente2(String
  keywordrecebida, int t_max, int limit)
  {
3      String keyword = NLPTool.toLowerCase(keywordrecebida
        );

5      String queryString = AbordagemPorIndividuo.
        getPrefixes()
          + "select distinct ?Category where {"
7          + "?s dcterms:subject ?Category ."
          + " ?Category rdfs:label ?title ."
9          + " ?title <bif:contains> \""+keyword+"\" ."

```

---

```

        + filtrarCategorias(AbordagemPorIndividuo.
                           categorias_filtradas, t_max)
11      + " } LIMIT "+limit;

13      QueryExecution qe = new QueryEngineHTTP("http://
        dbpedia.org/sparql", queryString);

15      ArrayList<String> categorias = new ArrayList<String>
        >();
17      try
        {
            ResultSet results = qe.execSelect();
19            for (; results.hasNext(); )
            {
                QuerySolution s = results.next();
                categorias.add(s.getResource("Category").
                               toString());
23            }
25        } finally
        {
27            qe.close();
29        }

31      return categorias;
    }

```

---

```

1  public ArrayList<String> getAllIndividuals(String category,
    int limit)
    {
3      String queryString = AbordagemPorIndividuo.
        getPrefixes()
        + "select distinct ?ind where {"
5        + " ?ind dcterms:subject <"+category+"> ."
        + " } LIMIT "+limit;

7      QueryExecution qe = new QueryEngineHTTP("http://
        dbpedia.org/sparql", queryString);

9      ArrayList<String> ind = new ArrayList<String>();
10     try
        {
13         ResultSet results = qe.execSelect();
            for (; results.hasNext(); )
15         {
                QuerySolution s = results.next();
                ind.add(s.getResource("ind").toString());
17         }
19     } finally
    }

```

```

21         {
23             ge.close();
24         }
25         return ind;

```

---

## C.4 VERSÃO ALTERNATIVA DA FERRAMENTA UTILIZANDO COMUNICAÇÃO ENTRE AGENTES

Foi decidido no trabalho original implementar o principal algoritmo de ranqueamento através de um artefato pois o algoritmo de ranqueamento se mostrou bastante complexo de implementar com agentes, por envolver a pontuação e ordenamento de uma grande quantidade de resultados. Porém também foi desenvolvida uma segunda versão da ferramenta, que utiliza comunicação entre agentes para fazer o ranqueamento.

Nesta versão o agente inicializador também faz o papel de agente coordenador. A ação !processItem dos agentes por abordagem foi modificada para que após executar uma busca abrangente e obter os indivíduos os agentes se comunicam com o agente inicializador enviando os resultados obtidos como uma crença. Abaixo é mostrado como essa modificação foi efetuada para o agente atuador\_por\_classes.

---

```

1 +!processItem(Keyword, Id) : true
2 <- executar_abrangente(Keyword, 20, Result1)[artifact_id(Id)
3   ];
4   executar_getIndividuos(Result1, 30, Result2)[artifact_id(
5     Id)];
6   .send("inicializador", tell, results2(Result2)).

```

---

Para cada crença atualizada o agente inicializador atualiza um contador e tenta criar o objetivo !rankResults, que após será executado quando o contador atingir três, ou seja, todos os agentes tiverem enviado seus resultados. Ao ser criada a intenção rankResults o agente faz intersecções de cada dois conjuntos de resultados e os armazenadas nas variáveis X, Y e Z, que então são agrupadas em uma única variável final através de unions. Preferiu-se fazer o ranqueamento desta forma ao invés de uma intersecção entre os três conjuntos de resultados para que haja um conjunto maior de resultados finais, já que um resultado precisa estar presente em apenas 2 das 3 abordagens para ser selecionado.

O código do agente inicializador que efetua o procedimento descrito é mostrado abaixo.

---

```

+results2(R): true
2 <- .my_name(Me);
   println(Me,": got results2 ", R);
4   -cont(V);
   +cont(V+1);
6   !rankResults.

8 +!rankResults: cont(3) & results1(R1) & results2(R2) &
   results3(R3)
   <- println("Rank:");
10   .intersection(R1,R2,X);
   .intersection(R2,R3,Y);
12   .intersection(R1,R3,Z);
   .union(X,Y,Parcial);
14   .union(Parcial,Z,Results);
   .length(Results,L);
16   !showResults(Results , L).

18 +!rankResults: true.

20 +!showResults(Results , L): L > 0
   <- .nth(0,Results,A);
22   println("1: ", A);
   .nth(1,Results,B);
24   println("2: ", B);
   .nth(2,Results,C);
26   println("3: ", C);
   .nth(3,Results,D);
28   println("4: ", D);
   .nth(4,Results,E);
30   println("5: ", E);
   put2(Results).
32
+!showResults(Results , L): true
34 <- println("Nenhum resultado encontrado.").

```

---

## **ANEXO A – Artigo SBC**





## **ANEXO B – Código-fonte**

