UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

Jéssica Pauli de Castro Bonson

APLICAÇÃO DE AGENTES & ARTEFATOS PARA O DESENVOLVIMENTO DE UMA FERRAMENTA DE AUTORIA DE OBJETOS DE APRENDIZAGEM

Florianópolis

2012

Jéssica Pauli de Castro Bonson

APLICAÇÃO DE AGENTES & ARTEFATOS PARA O DESENVOLVIMENTO DE UMA FERRAMENTA DE AUTORIA DE OBJETOS DE APRENDIZAGEM

Relatório submetido à Ciências da Computação para a obtenção do Grau de Projetos I da Graduação.
Orientador: Prof. Elder Rizzon Santos

Florianópolis

2012

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

SUMÁRIO

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 conseguiriam demonstrar um 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, será 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 gerarão de objetos em conformidade com padrões de metadados. A ferramenta de autoria que será desenvolvida neste trabalho possui apenas a segunda característica.

Em relação a repositórios semânticos, (SOURCEFORGE, c) os definem como ferramentas similares a um Sistema de Gerenciamento de Bando de Dados (SGBD), que podem ser usadas para armazenar, administrar e realizar consultas sobre dados estruturados de acordo com o padrão RDF. (ONTOTEXT, b) 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.
- Analisar e descrever a contribuição deste trabalho nas áreas de e-learning e A&A.

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 Definição de Agente

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: automonia, 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 impedí-lo de atingí-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.

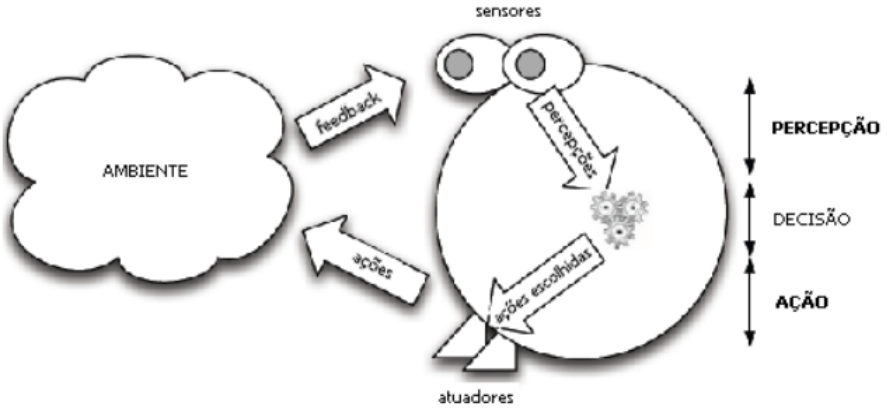

Figura 1 – Modelo básico de um agente (BOISSIER et al., 2011)

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.

2.1.2 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 2, 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 significante enquanto o agente decide que ação tomar, o que é bastante improvável de ocorrer em sistemas dinâmicos e complexos.

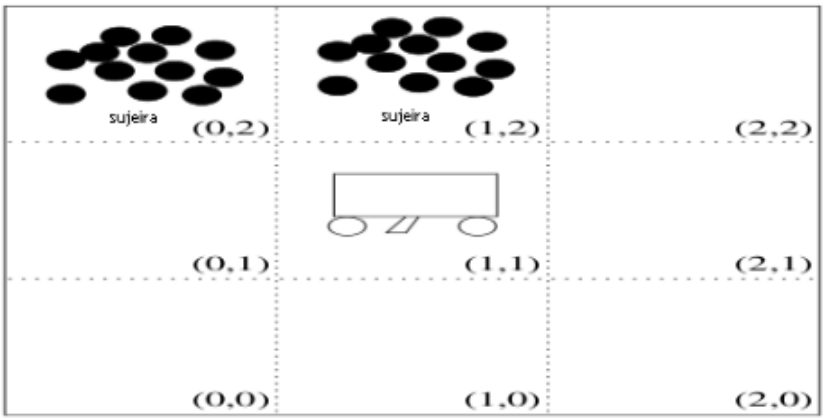

Figura 2 – Agente deliberativo Aspirador no Mundo do Vácuo (WOOLDRIDGE, 2002)

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 arquite-

tura BDI está resumido na Figura 3, 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 3, 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.

(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 cumprí-los. O autor também cita que outra dificuldade é implementar os processos de debiliberação e meios-fim de forma eficiente.

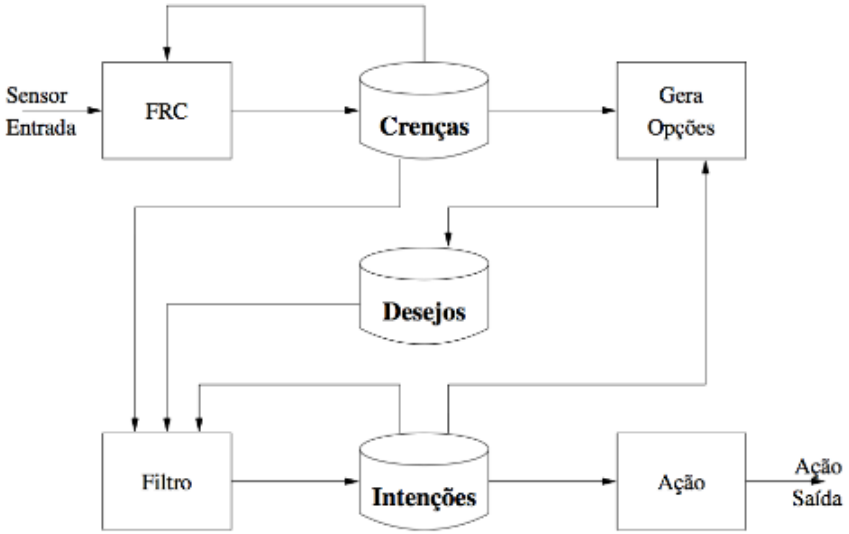

Figura 3 – Diagrama do processo de raciocínio em uma arquitetura BDI genérica. (HUBNER; BORDINI; VIEIRA, 2004)

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.

2.1.3 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. 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, com odesviar 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 Mundos Possíveis Semânticos (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, robusteza 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 4). 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.

(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 a ela, que representa o mundo em diferentes níveis de abstração de acordo com a camada. A utilização de bases de conheicmento é a principal diferença entre essas duas arquiteturas. É relevante notar que a arquitetura de

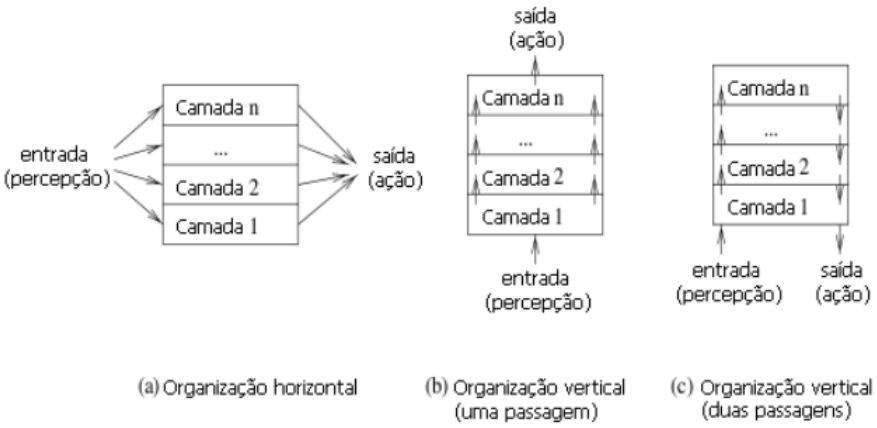

Figura 4 – Os três tipos de fluxo de controle e de informação que uma arquitetura em camadas pode ter (WOOLDRIDGE, 2002).

subsunção também é considerada uma arquitetura em camadas.

De acordo com (WOOLDRIDGE, 2002), as arquiteturas em camada 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 lógica clássica 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 à 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) (LTD.,): 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 a contexto, que podem tanto ser utilizados de forma reativa como pró-ativa. (LTD.,) 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 tráfico aéreo civil para os Servicos 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).

2.1.4 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. Jason e 2APL são explicados em mais detalhes em subseções 2.1.4.1 Jason e 2.1.4.2. 2APL.

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 extende 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. Golog é utilizada nas áreas de robótica cognitiva e na Web Semântica. 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, descrito na seção 2.1.3. 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 (SOURCEFORGE, b): 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. Outras características desta linguagem e as diferenças em relação ao AgentSpeak encontram-se na subseção 2.1.4.1. Jason.

2APL (SOURCEFORGE, a): 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 caracteríticas 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 (TRAC,): 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; GROUP,): 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 extende 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.

2.1.4.1 Jason

Nesta seção serão dadas mais informações sobre a linguagem de programação Jason, de acordo com (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).

+!reservar_ingressos(A,V) : not ocupado(telefone)
    <- ligar(V);
       ...;
       !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.

Informações específicas e exemplos sobre o uso da linguagem de agentes Jason para se programar SMAs são apresentadas na seção 2.2.6. Exemplos de implementação com Cartago e 2.2.7. Comparação entre ambientes Jason e Cartago.

```
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
   }
```

2.1.4.2 2APL

Além das características da linguagem 2APL explicadas na seção 2.1.4., 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 bom-

bas 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.

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

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

Plans:
```

```
B(inicio(X,Y)) ;
@mundodeblocos( entrar( X, Y, blue ), L )

Goals:
limpar( mundoDeBlocos )

PG-rules:
limpar( mundoDeBlocos ) <- bomba( X, Y ) |
{
    goto( X, Y );
    @mundodeblocos( pegar( ), L1 );
    Pegar( );
    RemoverBomba( X, Y );
    goto( 0, 0 );
    @mundodeblocos( soltar( ), L2 );
    Soltar( )
}
...

PC-rules:
goto( X, Y ) <- true |
{
    @mundodeblocos( sentirPosicao(), POS );
    B(POS = [A,B]);
    if B(A > X) then
    { @mundodeblocos( oeste(), L );
        goto( X, Y )
    }
    ...
}

...
```

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 mundodeblocos.

```
package mundodeblocos;

public class Env extends apapl.Environment {

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

```
    public void norte(String agente){...}
    ...
}
```

2.1.5 Definição de Sistema 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 provêem o poder fundamental deste paradigma. Outras vantagens relevantes de SMAs incluem: robusteza, eficiência, capacidade para permiter 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 tráfico 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 5 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.

Por fim, serão descritos os frameworks JADE e JIAC, escolhidos devido a sua ampla utilização (KüSTER et al., 2012; HOCH et al., 2011; GRUNEWALD et al., 2011; SPANOUDAKIS; MORAITIS, 2007; BELLIFEMINE; CAIRE; GREENWOOD, 2007).

JIAC (Java-based Intelligent Agent Componentware) (LABOR, ; 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, automonia e segurança, e também per-

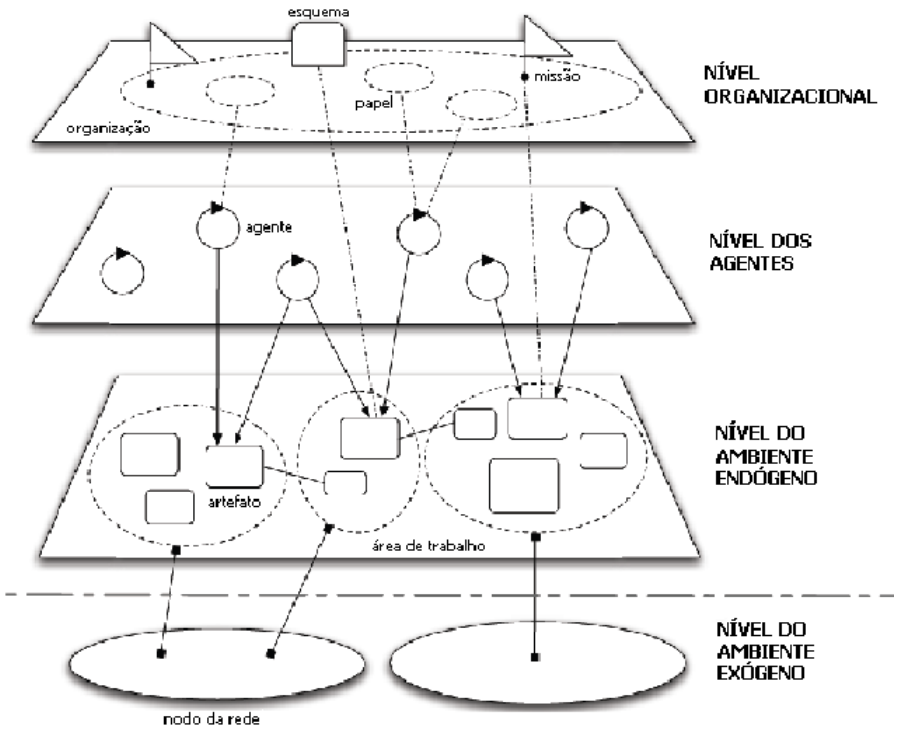

Figura 5 – Camadas de abstração de um SMA, retirado e traduzido de (BOISSIER et al., 2011).

mite 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 (CONTEST,).

A arquitetura e framework JADE (Java Agent DEvelopment Framework) (PROJECT,) 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 admi-

nistra 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.

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 citeref5. 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 indíviduos 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 pri-

meira 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 contruídos 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 6 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 na seção 2.2.2. Na Figura 6 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.

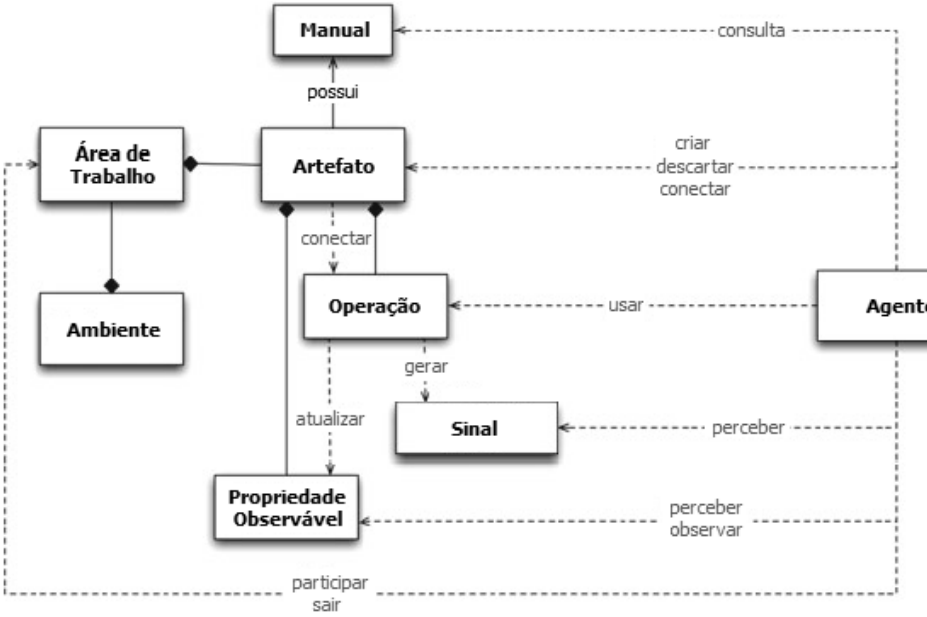

Figura 6 – O modelo A&A expressado em uma notação similar à UML (Unified Modelling Language), retirado e traduzido de (RICCI; PIUNTI; VIROLI, 2010).

A Figura 7 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.

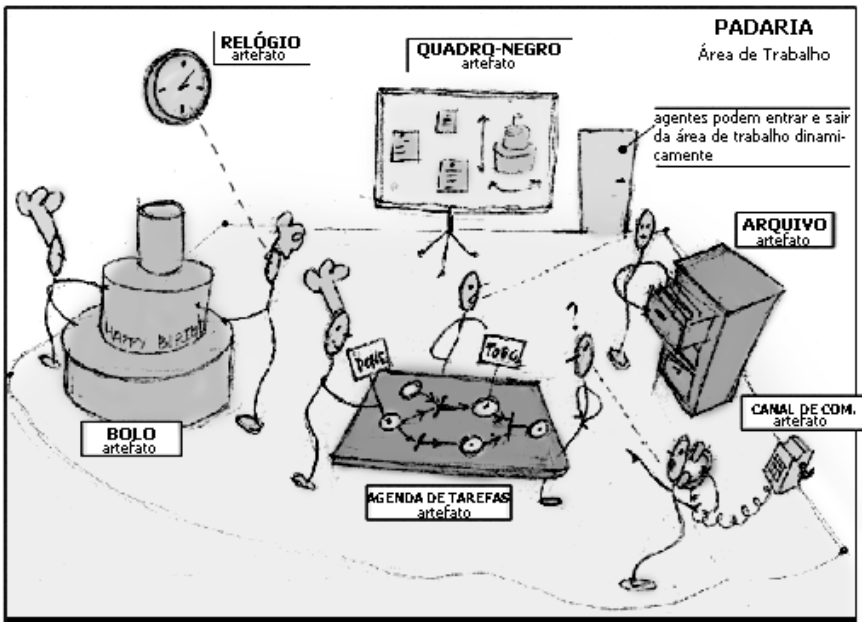

Figura 7 – Uma representação metafórica de um SMA de acordo com o modelo A&A, retirado e adaptado de (RICCI et al., 2009).

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 modulo 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 8 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).

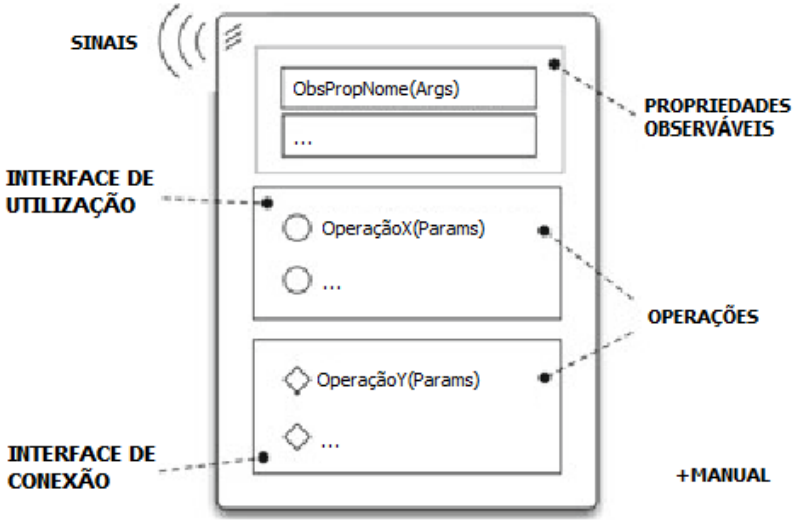

Figura 8 – Representação abstrata de um artefato, retirado e traduzido de (RICCI; PIUNTI; VIROLI, 2010).

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 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 pro-

cesso 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 artefato, 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 3 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 4, 5 e 6, 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 9 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.

Na Figura 10 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 11 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.

Para uma melhor compreensão do modelo dos artefatos, em (RICCI et al., 2009) é citado como exemplo um artefato simples: uma

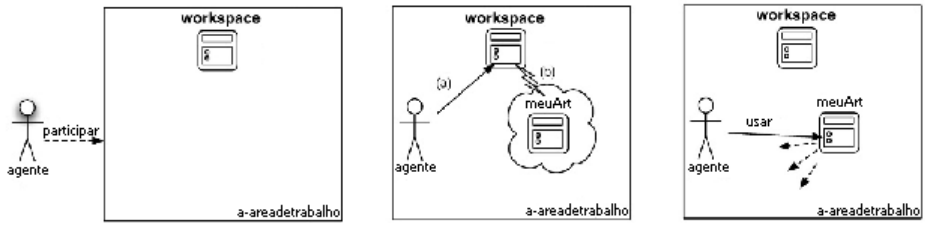

Figura 9 – Exemplos de interações dos agentes com uma área de trabalho, retirada e adaptada de (RICCI et al., 2009).

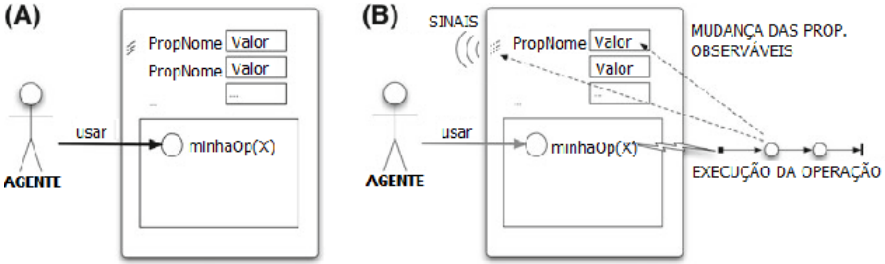

Figura 10 – Utilização de um artefato, retirada e traduzida de (RICCI; PIUNTI; VIROLI, 2010).

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 adiante na seção 2.2.5.2. 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

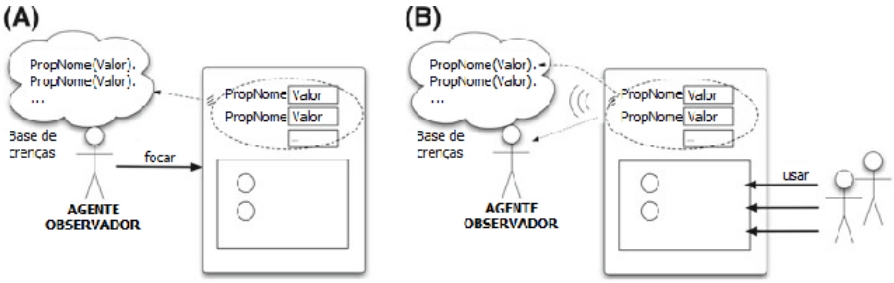

Figura 11 – Foco em um artefato, retirada e traduzida de (RICCI; PIUNTI; VIROLI, 2010).

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 tu-

plas, 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 na seção de exemplos. 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 concuíram 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 na seção de exemplos.

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 exemplos 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 extender 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 sobre o Cartago.

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 Frameworks para o modelo A&A: simpA e Cartago

Há dois frameworks principais para se trabalhar como o modelo A&A: simpA e Cartago. simpA será descrito brevemente, pois o framework usado neste trabalho é o Cartago. 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.

2.2.5.1 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 imprimí-la na tela. A implementação de artefatos em simpA é muito semelhante a sua implementação no Cartago, e será descrita na seção seguinte.

```
public class MeuAgente extends Agent {

@ACTIVITY_WITH_AGENDA({
    @TODO("atividadeA"),
    @TODO("atividadeB", pre="completed(atividadeA)"),
}) void main(){}

@ACTIVITY void atividadeA(){
    memo("x",1); // insere uma nova crença x(1)
}

@ACTIVITY void atividadeB(){
    int v = getMemo("x").intValue(0); // lê o primeiro
        argumento da crença x
    log("Resultado: "+v);
}
}
```

2.2.5.2 Cartago

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 12. 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 de ações para tornar possível que utilizem o ambiente baseado em artefatos. Abaixo serão explicadas as novas ações, mais explicações encontram-se na seção Artefatos 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

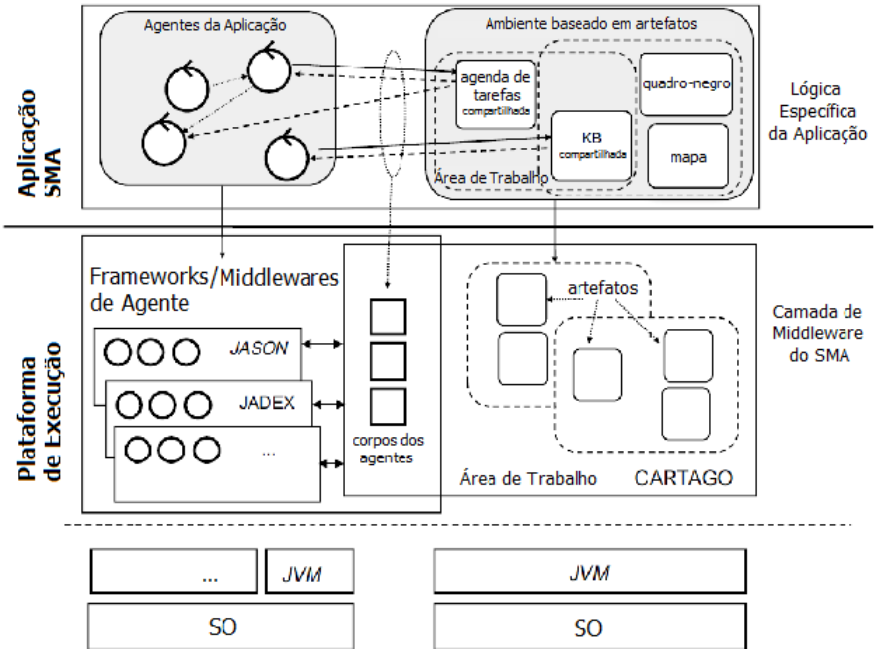

Figura 12 – Representação de como a integração entre as plataformas de agentes e o Cartago funciona, retirada de (RICCI et al., 2009).

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 extenda 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 contrutor 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 especifico, em milisegundos, tenha passado. Como no caso da primitica 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.

2.2.6 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.sourcefore.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 ¡- 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 esternas 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).

2.2.6.1 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 13 – Representação abstrata do Artefato Contador

```
package Artefatos;

import cartago.*;

public class Contador extends Artifact {

  void init(){
    defineObsProperty("contagem",0);
  }

  @OPERATION void inc(){
    ObsProperty prop = getObsProperty("contagem");
    prop.updateValue(prop.intValue()+1);
  }
}
```

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.

```
!testar_contador.

+!testar_contador
  <- makeArtifact("meuContador","Artefatos.Contador",[],Id);
     focus(Id);
     println("Início do teste.");
     inc;
     inc;
     inc;
     println("Fim do teste.").

+contagem(V): true
  <- println("Valor do contador: ",V).
```

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

```
[usuario_contador] Valor do contador: 0
[usuario_contador] Início do teste.
[usuario_contador] Valor do contador: 1
[usuario_contador] Valor do contador: 2
[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 {
    environment: c4jason.CartagoEnvironment

    agents:
    usuario_contador agentArchClass c4jason.CAgentArch;

    classpath: "../cartago-2.0.1/lib/cartago.jar";
    "../cartago-2.0.1/lib/c4jason.jar";
}
```

2.2.6.2 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 Relogio começa sendo inicializado com a propriedade interna "funcionando"como false e não possui propriedades observáveis. Ele possui as operações "comecar"e "parar"disponíveis para os agentes, que respectivamente começam e param a atividade do relógio. Ao ser executada a operação "comecar"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, fazendo com que quando a etapa "contar"for novamente executada após 0.1 segundos ela saia do loop.

```
package Artefatos;

import cartago.*;

public class Relogio extends Artifact {

    boolean funcionando;
    final static long TEMPO_TICK = 100;
```

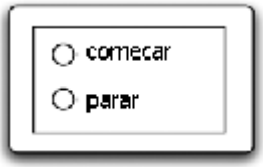

Figura 14 – Representação abstrata do Artefato Relógio

```
9
      void init(){
11      funcionando = false;
      }
13
      @OPERATION void comecar(){
15      if (!funcionando){
          funcionando = true;
17        execInternalOp("contar");
        } else {
19        failed("já está funcionando");
        }
21    }

23    @OPERATION void parar(){
        funcionando = false;
25    }

27    @INTERNAL_OPERATION void contar(){
        while (funcionando){
29        signal("tick");
          await_time(TEMPO_TICK);
31      }
      }
33  }
```

Abaixo segue o código do agente Jason que utiliza Relogio. Primeiro ele cria uma instância "meuRelogio"do tipo "Relogio", 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 "comecar"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 pára 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 for percebido.

```
!testar_relogio.

+!testar_relogio
  <- makeArtifact("meuRelogio","Artefatos.Relogio",[],Id);
     focus(Id);
     +n_ticks(0);
     comecar;
     println("relógio começou.").

+tick: n_ticks(10)
  <- parar;
     println("relógio parou.").

+tick: n_ticks(N)
  <- -+n_ticks(N+1);
     println("tick percebido!").
```

Abaixo segue a saída no MAS Console do Jason:

```
[usuario_relogio] relógio começou.
[usuario_relogio] tick percebido!
[usuario_relogio] tick percebido!
[usuario_relogio] tick percebido!
[usuario_relogio] tick percebido!
[usuario_relogio] tick percebido!
[usuario_relogio] tick percebido!
[usuario_relogio] tick percebido!
[usuario_relogio] tick percebido!
[usuario_relogio] tick percebido!
[usuario_relogio] relógio parou.
```

2.2.6.3 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("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 15 – Representação abstrata do Artefato Sincronizador

```
package Artefatos;

import cartago.*;

public class Sincronizador extends Artifact {

    int nProntos, nParticipantes;

    void init(int nParticipantes) {
        nProntos = 0;
        this.nParticipantes = nParticipantes;
    }

    @OPERATION
    void sincronizar() {
        nProntos++;
        await("todosProntos");
    }

    @GUARD
    boolean todosProntos() {
        return nProntos == nParticipantes;
    }
}
```

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.

+!trabalhar
  <- !setupTools(Id);
     println("Início da sincronização.");
   .random(N);
   N2 = N*10000;
   .wait(N2);
   println("Trabalhei por ",N2/1000, " segundos.");
             focus(Id);
             println("Ativar operação sincronizar()");
   sincronizar;
   println("Sincronização efetuada!");
   println("Fim do teste.").

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

-!setupTools(Id) : true <-
  lookupArtifact("sincronizador",Id).
```

Abaixo segue a saída no MAS Console do Jason:

```
[usuario_sincr4] Início da sincronização.
[usuario_sincr2] Início da sincronização.
[usuario_sincr3] Início da sincronização.
[usuario_sincr1] Início da sincronização.
[usuario_sincr5] Início da sincronização.
[usuario_sincr4] Trabalhei por 1.528724231200107 segundos.
[usuario_sincr4] Ativar operação sincronizar()
[usuario_sincr1] Trabalhei por 5.256301264501251 segundos.
```

```
[usuario_sincr1] Ativar operação sincronizar()
[usuario_sincr5] Trabalhei por 6.559502173560134 segundos.
[usuario_sincr5] Ativar operação sincronizar()
[usuario_sincr2] Trabalhei por 6.648731124199021 segundos.
[usuario_sincr2] Ativar operação sincronizar()
[usuario_sincr3] Trabalhei por 7.27029024339723 segundos.
[usuario_sincr3] Ativar operação sincronizar()
[usuario_sincr3] Sincronização efetuada!
[usuario_sincr4] Sincronização efetuada!
[usuario_sincr5] Sincronização efetuada!
[usuario_sincr2] Sincronização efetuada!
[usuario_sincr1] Sincronização efetuada!
[usuario_sincr4] Fim do teste.
[usuario_sincr5] Fim do teste.
[usuario_sincr3] Fim do teste.
[usuario_sincr2] Fim do teste.
[usuario_sincr1] Fim do teste.
```

2.2.6.4 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 TupleSpace (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. Mais informações sobre os espaços de tuplas estão na seção 2.2.3. 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.

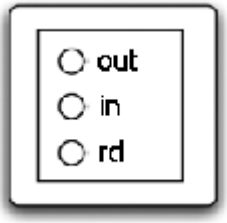

Figura 16 – Representação abstrata do Artefato TupleSpace

```
1  package cartago.tools;

3  import cartago.*;

5  public class TupleSpace extends Artifact {

7    TupleSet tset;

9    void init(){
       tset = new TupleSet();
11   }

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 adicionados tuplas (garfo, valor), sendo "valor"de 0 a 4 e tuplas ("filosofo_init",Nome,Esquerda,Direit 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).
```

```
!preparar_mesa.

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

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

+!viver
 <- !pensar;
    !comendo;
    !!viver.

+!comendo
 <- !adquirirGarfos;
    !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 (...)
```

2.2.7 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 (SOURCEFORGE, b), sendo que o ambiente foi 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 "checar". 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

/* Initial beliefs and rules */

em_local(P) :- pos(P,X,Y) \& pos(r1,X,Y).

/* Initial goals */

!checar(slots).

/* 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
+!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
          // volta e continua checando o mundo
30        !ir_para(last);
          !!checar(slots).
32
+!pegar(S,L) : true
34    <- !assegurar_pegar(S);
          !ir_para(L);
36        largar(S).

38 +!assegurar_pegar(S) : lixo(r1)
      <- pegar(lixo);
40        !assegurar_pegar(S).
+!assegurar_pegar(_).
42
+!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 correpondente 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.*;
import jason.environment.Environment;
import jason.environment.grid.GridWorldModel;
import jason.environment.grid.Location;

import java.util.Random;
import java.util.logging.Logger;

public class MarsEnv extends Environment {

    public static final int GSize = 7; // tamanho da grade
    public static final int LIXO  = 16; // código do lixo no
        modelo de grade

    public static final Term    ns = Literal.parseLiteral("
        prox(slot)");
    public static final Term    pg = Literal.parseLiteral("
        pegar(lixo)");
    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 execução 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
            try {
58              Thread.sleep(200);
            } catch (Exception e) {}
60          return true;
        }
```

```
62
    /** cria as percepções dos agentes baseadas no MarsModel
        */
64  void updatePercepts() {
        clearPercepts();
66
        Location r1Loc = modelo.getAgPos(0);
68      Location r2Loc = modelo.getAgPos(1);

70      Literal pos1 = Literal.parseLiteral("pos(r1," +
            r1Loc.x + "," + r1Loc.y + ")");
        Literal pos2 = Literal.parseLiteral("pos(r2," +
            r2Loc.x + "," + r2Loc.y + ")");
72
        addPercept(pos1);
74      addPercept(pos2);

76      if (modelo.hasObject(LIXO, r1Loc)) {
            addPercept(g1);
78      }
        if (modelo.hasObject(LIXO, r2Loc)) {
80          addPercept(g2);
        }
82
        if(modelo.buscaTerminou == true) {
84        addPercept(Literal.parseLiteral("terminou"));
        }
86  }

88  class MarsModel extends GridWorldModel {

90      public static final int MErr = 2; // max erros ao
            pegar o lixo
        int nerr; // número de tentativas de pegar o lixo
92      boolean r1TemLixo = false; // se r1 está carregando
            lixo ou não
        boolean buscaTerminou = false;
94
        Random random = new Random(System.currentTimeMillis
            ());
96
        private MarsModel() {
98          super(GSize, GSize, 2);

100         // localização inicial dos agentes
            try {
102             setAgPos(0, 0, 0);

104             Location r2Loc = new Location(GSize/2, GSize
                    /2);
                setAgPos(1, r2Loc);
106         } catch (Exception e) {
```

```
                e.printStackTrace();
108         }

110         // localização inicial do lixo
            add(LIXO, 3, 0);
112         add(LIXO, GSize-1, 0);
            add(LIXO, 1, 2);
114         add(LIXO, 0, GSize-2);
            add(LIXO, GSize-1, GSize-1);
116     }

118     void proxSlot() throws Exception {
            Location r1 = getAgPos(0);
120         r1.x++;
            if (r1.x == getWidth()) {
122             r1.x = 0;
                r1.y++;
124         }
            // termina de procurar na grade
126         if (r1.y == getHeight()) {
              buscaTerminou = true;
128             return;
            }
130         setAgPos(0, r1);
        }
132
        void irPara(int x, int y) throws Exception {
134         Location r1 = getAgPos(0);
            if (r1.x < x)
136             r1.x++;
            else if (r1.x > x)
138             r1.x--;
            if (r1.y < y)
140             r1.y++;
            else if (r1.y > y)
142             r1.y--;
            setAgPos(0, r1);
144     }

146     void pegarLixo() {
            // local de r1 tem lixo
148         if (modelo.hasObject(LIXO, getAgPos(0))) {
                // as vezes a ação "pegar" são funciona
150             // mas nunca mais que MErr vezes
                if (random.nextBoolean() || nerr == MErr) {
152                 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 "lookupArti-

fact"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 17 – 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);
    pos_r2 = new Posicao(GSize/2, GSize/2);

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

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

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

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

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

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

           signal("pos(r1," + pos_r1.x + "," + pos_r1.y + "
               )");
           if(temLixoEm(pos_r1))
           {
             signal("lixo", "r1");
           }
     }

  @OPERATION
  void ir_para(int x, int y)
  {
    printLog("ir_para("+x+","+y+")");
```

```
67              if (pos_r1.x < x)
                  pos_r1.x++;
69              else if (pos_r1.x > x)
                  pos_r1.x--;
71              if (pos_r1.y < y)
                  pos_r1.y++;
73              else if (pos_r1.y > y)
                  pos_r1.y--;
75
                    getObsProperty("pos_r1").updateValues(pos_r1
                        .x, pos_r1.y);
77    }

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

97    @OPERATION
      void largar(String lixo)
99    {
        printLog("largar("+lixo+")");
101     if (getObsProperty("r1TemLixo").booleanValue()) {
                    getObsProperty("r1TemLixo").updateValues(
                        false);
103                 pos_lixos.add(new Posicao(pos_r1.x, pos_r1
                        .y));
                    if(pos_r1.x == pos_r2.x \&\& pos_r1.y ==
                        pos_r2.y)
105                   signal("lixo", "r2");
                }
107   }

109   @OPERATION
      void queimar(String lixo)
111   {
        printLog("queimar("+lixo+")");
```

```
113     // local de r2 tem lixo
                if (temLixoEm(pos_r2))
115               removerLixoEm(pos_r2);
    }
117
    private boolean temLixoEm(Posicao p)
119 {
      for(Posicao pos: pos_lixos)
121     if(pos.x == p.x \&\& pos.y == p.y)
          return true;
123   return false;
    }
125
    private void removerLixoEm(Posicao p)
127 {
      for(Posicao pos: pos_lixos)
129     if(pos.x == p.x \&\& pos.y == p.y)
        {
131       pos_lixos.remove(pos);
          break;
133     }
    }
135
    private void printLog(String acao)
137 {
      System.out.println(getOpUserName()+" executando: "+ acao
          );
139 }

141 class Posicao
    {
143   int x;
      int y;
145
      //...
147 }
}
```

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)
  ...
```

```
[MarsEnv] r1 executando: prox(slot)
[MarsEnv] r1 executando: pegar(lixo)
[MarsEnv] r1 executando: ir_para(3,3)
[MarsEnv] r1 executando: ir_para(3,3)
[MarsEnv] r1 executando: ir_para(3,3)
[MarsEnv] r1 executando: largar(lixo)
[MarsEnv] r1 executando: ir_para(6,6)
[MarsEnv] r1 executando: ir_para(6,6)
[MarsEnv] r2 executando: queimar(lixo)
[MarsEnv] r1 executando: ir_para(6,6)
[MarsEnv] r1 executando: prox(slot)
[r1] Conclui a busca
```

```
r1 executando: prox(slot)
r1 executando: prox(slot)
r1 executando: prox(slot)
r1 executando: pegar(lixo)
r1 executando: pegar(lixo)
r1 executando: ir_para(3,3)
r1 executando: ir_para(3,3)
r1 executando: ir_para(3,3)
r1 executando: largar(lixo)
r2 executando: queimar(lixo)
r1 executando: ir_para(3,0)
r1 executando: ir_para(3,0)
r1 executando: ir_para(3,0)
r1 executando: prox(slot)
r1 executando: prox(slot)
r1 executando: prox(slot)
r1 executando: pegar(lixo)
r1 executando: pegar(lixo)
r1 executando: pegar(lixo)
r1 executando: ir_para(3,3)
r1 executando: ir_para(3,3)
r1 executando: ir_para(3,3)
r1 executando: largar(lixo)
r1 executando: ir_para(6,0)
r2 executando: queimar(lixo)
r1 executando: ir_para(6,0)
r1 executando: ir_para(6,0)
r1 executando: prox(slot)
...
r1 executando: prox(slot)
r1 executando: pegar(lixo)
r1 executando: ir_para(3,3)
r1 executando: ir_para(3,3)
r1 executando: ir_para(3,3)
r1 executando: largar(lixo)
r2 executando: queimar(lixo)
r1 executando: ir_para(6,6)
```

```
r1 executando: ir_para(6,6)
r1 executando: ir_para(6,6)
r1 executando: prox(slot)
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 díficil de entender.

Uma conclusão revelevante 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.

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 o 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 vocabulários 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 (DCMI,), que busca utilizar metadados para descrever objetos digitais.

A principal utilidade de uma ontologia é representar conhecimento sobre um domínio, segundo (WIKIPEDIA,) 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 (concei-

tos, 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 (PROTEGE,), um exemplo de uma ontologia exemplo 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, f) 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, f) 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 (CLARKPARSIA,), FaCT++ (BECHHOFER,) e Racer (SYSTEMS,).

2.3.3 RDF e OWL / OWL 2

De acordo com (W3C, f) a linguagem RDF (Resource Description Framework) (W3C, d) 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 linguagem 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 da-

dos 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) (W3C, e), 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) (W3C, a) , 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) afima 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 ambigua 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 intencionadas 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".

Uma ontologia OWL utiliza os conceitos de indivíduos, classes, propriedades (sobre objetos ou sobre dados), relações e restrições explicados na seção 2.3.2.. Na Figura 18 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 =¿ Pessoa), e_pet (Cachorro =¿ Pessoa) e tem_pet (Pessoa =¿ 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 (HERMIT,) 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) (W3C, i, g). 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 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

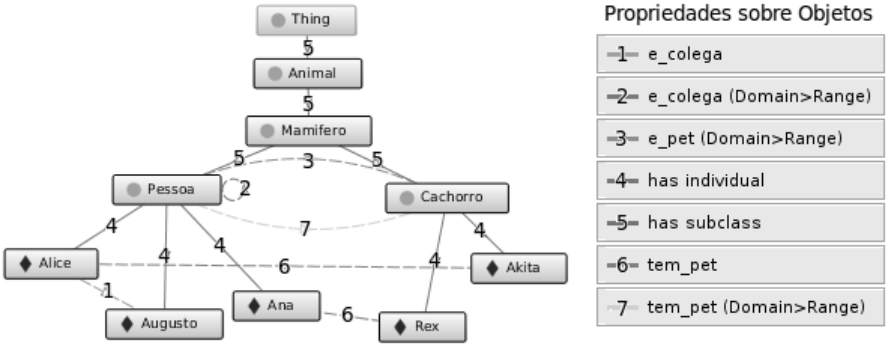

Figura 18 – Exemplo de ontologia OWL desenvolvida no Protégé.

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.
org/2003/01/periodictable/PeriodicTable#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT ?nome ?simbolo ?peso ?numero
FROM <http://www.daml.org/2003/01/periodictable/
    PeriodicTable.
owl>
WHERE {
?elemento table:group ?grupo .
?grupo table:name "Noble gas"^^xsd:string .
?elemento table:name ?nome .
?elemento table:symbol ?simbolo .
?elemento table:atomicWeight ?peso .
?elemento table:atomicNumber ?numero
}
ORDER BY ASC(?nome)
```

Por fim, a OWL2 é uma extensão da OWL que, entre outras mudanças, aumenta sua expressividade. Algumas das novas características

Tabela 1 – Tabela com 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

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, asimétricas 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 extendido para tipos de dados e anotações. Informações mais detalhadas sobre a OWL2 podem ser encontradas em (GRAU et al., 2008; W3C, b, c).

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 (SOURCEFORGE, c).

Neste contexto foram desenvolvidos os repositórios semânticos, segundo (SOURCEFORGE, c) não existe uma definição formal para este termo, os autores definem repositórios semânticos como uma ferramenta similar a um Sistema de Gerenciamento de Bando de Dados (SGBD), que pode ser usada para armazenar, administrar e realizar consultas sobre dados estruturados de acordo com o padrão RDF. (ONTOTEXT, b) 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, b) cita Sesame como um dos repositórios semânticos mais populares. Sesame(W3C, h; SOURCEFORGE, c) 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,).

Outro exemplo citado pelos autores é o OWLIM. OWLIM(BISHOP et al., 2011; ONTOTEXT, a) 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 semnânticas 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. O OWLIM-Lite pode ser obtido gratuitamente, os outros dois repositórios podem se obtidos gratuitamente apenas para um período de avaliação.

2.4 FERRAMENTAS DE AUTORIA DE OBJETOS DE APRENDIZAGEM

2.4.1 Introdução

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 multimidia, sistemas 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.

(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) (DCMI,), LOM (Learning Object Metadata) (IEEE, 2002), Scorm (Sharable Content Object Reference Model) (SOLUTIONS,) e CanCore (UNIVERSITY,).

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, Reaload 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 – Tabela com 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 tinham 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 escarsas 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 dessa pesquisa 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) (OBAA,) , 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.

3 DESENVOLVIMENTO

O desenvolvimento do trabalho até o momento será explicado baseando-se nos objetivos específicos descritos na Seção 1.1., citados abaixo:

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:

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. Analisar e descrever a contribuição deste trabalho nas áreas de e-learning e A&A.

Desenvolvimento:

Item 1: Concluído, os requisitos da ferramenta de autoria de metadados foram identificados e explicados na seção 2.4.3. Requisitos da Ferramenta de Autoria deste trabalho, e são resumidos na Tabela 3.

Item 2: Concluído através da pesquisa para este texto e do desenvolvimento de artefatos utilizando o Cartago.

Item 3: Em andamento, o funcionamento dos repositórios semânticos foi compreendido de forma teórica através desta pesquisa, mas ainda foram feitos poucos testes práticos. O repositório semântico utilizado será o OWLIM-Lite.

Item 4: Sem andamento.

Tabela 3 – Tabela com requisitos obrigatórios e desejáveis da ferramenta de autoria deste trabalho

Requisito Obrigatório	Requisito Desejável
Automatizar e facilitar o preenchimento de metadados.	Ferramenta Web.
Representar os metadados e os OAs através de ontologias.	Suportar os padrões Scorm e/ou LOM.
Suportar o padrão OBAA.	
Fornecer um tutorial ou manual para a ferramenta.	

Item 5: Em andamento, será utilizada a ontologia com os metadados do padrão OBAA, mas ainda não feito um trabalho prático quanto a isso.

Item 6: Em andamento, está sendo desenvolvido um artefato que encapsula uma ontologia, permitindo que os agentes tenham acesso a ela.

Item 7: Será feito durante o desenvolvimento da parte prática do trabalho.

REFERÊNCIAS

ARISHA, K. A. et al. Impact: A platform for collaborating agents. 1999.

BECHHOFER, S. *Fact++*. <http://owl.man.ac.uk/factplusplus/>. Acessado em 20/06/2012.

BELLIFEMINE, F. L.; CAIRE, G.; GREENWOOD, D. Developing multi-agent systems with jade. 2007.

BISHOP, B. et al. Owlim: A family of scalable semantic repositories. 2011.

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.

CLARKPARSIA. *Pellet*. <http://clarkparsia.com/pellet/>. Acessado em 20/06/2012.

CONTEST, M. *MultiAgent Contest*. <http://multiagentcontest.org/>. Acessado em 20/04/2012.

DCMI. *DublinCore*. <http://dublincore.org/>. Acessado em 01/06/2012.

DUVAL, E. et al. Metadata principles and practicalities. 2002.

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

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

GROUP, A. *JACK*. <http://www.agent-software.com.au/products/jack/>. Acessado em 20/04/2012.

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

HENDLER, J.; HARMELEN, F. van. The semantic web: Webizing knowledge representation. 2008.

HERMIT. *HermiT*. <http://hermit-reasoner.com/>. Acessado em 20/06/2012.

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. 2008.

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. 1996.

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

LABOR, D. *JIAC*. <http://www.jiac.de/>. Acessado em 20/04/2012.

LTD., A. A. I. I. *dMARS*. <http://web.archive.org/web/20000229224150/www.aaii.oz.au/proj/dMARS-prod-brief.html>. Acessado em 20/04/2012.

OBAA. *Portal OBAA*. <http://www.portalobaa.org/>. Acessado em 01/05/2012.

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

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

ONTOTEXT. *OWLIM*. <http://www.ontotext.com/owlim>. Acessado em 25/06/2012.

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

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

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.

PROJECT, J. *Jade*. <http://jade.tilab.com/>. Acessado em 20/04/2012.

PROTEGE. *Protege*. <http://protege.stanford.edu/>. Acessado em 20/06/2012.

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. 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. 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. 2008.

RICCI, A.; VIROLI, M.; OMICINI, A. "give agents their artifacts": The a&a approach for engineering working environments. 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.

SHOHAM, Y. Agent-oriented programming. 1993.

SOLUTIONS, J. *Scorm*. <http://www.scormsoft.com/scorm>. Acessado em 01/06/2012.

SOURCEFORGE. *2APL*. <http://apapl.sourceforge.net/>. Acessado em 20/04/2012.

SOURCEFORGE. *Jason*. <http://jason.sourceforge.net/Jason/Jason.html>. Acessado em 20/04/2012.

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

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

SYSTEMS, R. *Racer*. <http://www.racer-systems.com/>. Acessado em 20/06/2012.

TRAC. *GOAL*. <http://mmi.tudelft.nl/trac/goal>. Acessado em 20/04/2012.

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

UNIVERSITY, A. *CanCore*. <http://cancore.athabascau.ca/en/>. Acessado em 01/06/2012.

W3C. *OWL*. <65. http://www.w3.org/TR/owl-ref/>. Acessado em 20/06/2012.

W3C. *OWL2 New Features*. <http://www.w3.org/TR/owl2-new-features/>. Acessado em 20/06/2012.

W3C. *OWL2 Overview*. <http://www.w3.org/TR/owl2-overview/>. Acessado em 20/06/2012.

W3C. *RDF*. <http://www.w3.org/RDF/>. Acessado em 20/06/2012.

W3C. *RDF Schema*. <http://www.w3.org/TR/rdf-schema/>. Acessado em 20/06/2012.

W3C. *SemanticWeb: Inference*. <http://www.w3.org/standards/semanticweb/inference>. Acessado em 20/06/2012.

W3C. *SemanticWeb: Query*. <http://www.w3.org/standards/semanticweb/query>. Acessado em 20/06/2012.

W3C. *Sesame*. <http://www.w3.org/2001/sw/wiki/Sesame>. Acessado em 25/06/2012.

W3C. *SPARQL*. <67. http://www.w3.org/TR/sparql11-query/>. Acessado em 20/06/2012.

WEYNS, D.; OMICINI, A.; ODELL, J. Environment as a first class abstraction in multiagent 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*. <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. Relatório técnico rt-obaa-06 subgrupo agentes e ontologias: Análise comparativa de editores de objetos de aprendizagem. 2009.