PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

# AGENTS ANYWHERE (AA) – UMA LINGUAGEM PARA O DESENVOLVIMENTO DE APLICAÇÕES MULTIAGENTES UBÍQUAS

MAURICIO DA SILVA ESCOBAR

Tese apresentada como requisito parcial à obtenção do grau de Doutor, pelo programa de Pós-Graduação em Ciência da Computação da Pontifícia Universidade Católica do Rio Grande do Sul.

Orientador: Prof. Dr. Rafael H. Bordini

Porto Alegre
2013

**Dados Internacionais de Catalogação na Publicação (CIP)**

E74a Escobar, Mauricio da Silva
Agents Anywhere (AA) - uma linguagem para o desenvolvimento de aplicações multiagentes ubíquas / Mauricio da Silva Escobar. Porto Alegre, 2013.
129 f.

Tese (Doutorado) – Fac. de Informática, PUCRS.
Orientador: Prof. Dr. Rafael H. Bordini.

1. Informática. 2. Linguagens de Programação.
3. Sistemas Multiagentes. I. Bordini, Rafael H. II. Título.

CDD 005.13

**Ficha Catalográfica elaborada pelo**
**Setor de Tratamento da Informação da BC-PUCRS**

Pontifícia Universidade Católica do Rio Grande do Sul
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

# TERMO DE APRESENTAÇÃO DE TESE DE DOUTORADO

Tese intitulada "*Agents Anywhere* (aa) – Uma Linguagem para o Desenvolvimento de Aplicações Multiagentes Ubíquas", apresentada por Mauricio da Silva Escobar, como parte dos requisitos para obtenção do grau de Doutor em Ciência da Computação, Inteligência Computacional, aprovada em 07/03/2013 pela Comissão Examinadora:

Prof. Dr. Rafael Heitor Bordini – Orientador — PPGCC/PUCRS

Prof. Dr. Felipe Rech Meneguzzi - — PPGCC/PUCRS

Prof. Dr. Carlos Eduardo Pereira - — UFRGS

Prof. Dr. Jomi Fredi Hübner – — UFSC

Homologada em 14/05/13, conforme Ata No. 08 pela Comissão Coordenadora.

Prof. Dr. Paulo Henrique Lemelle Fernandes
Coordenador.

PUCRS
**Campus Central**
Av. Ipiranga, 6681 – P. 32 – sala 507 – CEP: 90619-900
Fone: (51) 3320-3611 – Fax (51) 3320-3621
E-mail: ppgcc@pucrs.br
www.pucrs.br/facin/pos

# AGRADECIMENTOS

Aos meus pais, Ângela e Mauro, pela educação, incentivo e pela incansável dedicação.

A minha namorada, Bianca, por todo o apoio, compreensão e paciência em todos os momentos.

Ao meu orientador, Professor Rafael H. Bordini por ter aceitado este desafio; pelo apoio e pelos ensinamentos.

A um grande mestre, Professor Marcelo Blois Ribeiro, pela paciência em minhas inúmeras dúvidas conceituais; por acreditar no meu trabalho desde a graduação; por ter me orientado no início deste trabalho; e pelos ensinamentos passados, que contribuíram imensamente para o meu crescimento pessoal e profissional.

Aos membros da banca, pela aceitação do convite de participação na avaliação e pelas contribuições realizadas para a melhoria deste trabalho.

A minha família e amigos pelo apoio.

Aos meus colegas do antigo grupo de pesquisa ISEG.

A todos os professores e funcionários da Faculdade de Informática da PUCRS.

# AGENTS ANYWHERE (AA) – UMA LINGUAGEM PARA O DESENVOLVIMENTO DE APLICAÇÕES MULTIAGENTES UBÍQUAS

## RESUMO

Nos últimos anos muitos esforços em pesquisas e desenvolvimento tem sido direcionados para a área de computação ubíqua e sistemas multiagentes (SMAs). O avanço nessas áreas é necessário para obter vantagem na utilização de suas tecnologias a fim de prover inteligência, flexibilidade e novas abstrações na construção de aplicações ubíquas. Nesse contexto é onde a inteligência artificial e as técnicas de sistemas multiagentes desempenharão um papel importante. Este trabalho apresenta a construção de uma linguagem de programação para aplicações multiagentes ubíquas. A linguagem define conceitos de ubiquidade no mesmo nível de abstração dos conceitos de sistemas multiagentes, onde conceitos importantes de ubiquidade como localização e dispositivos são abstrações de primeira ordem. O projeto da linguagem baseia-se no metamodelo FAML, um metamodelo genérico para o desenvolvimento de SMAs, e também em funcionalidades essenciais de linguagens de programação orientadas a agentes.

**Palavras chave:** linguagem de programação, computação ubíqua, sistemas multiagentes.

# AGENTS ANYWHERE (AA) – UMA LINGUAGEM PARA O DESENVOLVIMENTO DE APLICAÇÕES MULTIAGENTES UBÍQUAS

## ABSTRACT

In recent years, much research and development effort has been directed towards the fields of ubiquitous computing and multi-agent systems. Further progress is needed for taking full advantage of such technologies in order to provide a degree of intelligence, flexibility, and abstraction in building ubiquitous applications. This work presents the design of a programming language for the development of ubiquitous multi-agent applications. The language aims to define ubiquitous concepts at the same level of abstraction as multi-agent systems concepts, where important ubiquity concepts such as locations and devices are first-class abstractions. The design of the language was based on FAML, a generic meta-model for MAS development, and also draws upon some of the fundamental features of agent-oriented programming languages.

**Keywords:** programming language, ubiquitous computing, multi-agent systems.

# LISTA DE FIGURAS

# LISTAGENS

# LISTA DE SIGLAS

AA – Agents Anywhere

ACL – *Agent Communication Language*

AMS – *Agent Management System*

AOSE – *Agent-Oriented Software Engineering*

BDI – *Belief-Desire-Intention*

EBNF – *Extended Backus-Naur Form*

FAME – *Framework for Agent-Oriented Method Engineering*

FAML – FAME *Agent-oriented Modeling Language*

FIPA – *Foundation for Intelligent Physical Agents*

GPS – *Global Positioning System*

IDE – *Integrated Development Environment*

JADE – *Java Agent DEvelopment Framework*

MAS – *Multiagent Systems*

MDD – *Model Driven Development*

OO – Orientação a Objetos

PDA – *Personal Digital Assistant*

RMI – *Remote Method Invocation*

SMA – Sistema Multiagente

TI – Tecnologia da Informação

UML – *Unified Modeling Language*

U-MAS – *Ubiquitous-Multiagent Systems*

# SUMÁRIO

# 1 INTRODUÇÃO

A computação ubíqua consiste na integração de sistemas computacionais em nosso dia a dia, provendo serviços e informações em qualquer lugar e a qualquer momento [Wei99a, AS99]. Ela propõe um novo paradigma computacional capaz de prover acesso computacional a usuários de forma invisível, isto é, o usuário não precisa perceber a tecnologia para aproveitar seus benefícios.

As tecnologias de agentes e sistemas multiagentes (SMAs) desempenham um papel importante em computação ubíqua, e tem sido utilizadas como uma abordagem atrativa para a criação de aplicações para ambientes ubíquos [Sym10, GLZ+08]. Agentes, inseridos em ambientes ubíquos, serão capazes de responder às mudanças ocorridas no ambiente, seja ele virtual ou físico, melhorando seu desempenho e seu comportamento, adicionando flexibilidade a esse ambiente.

A computação ubíqua tem evoluído da área de pesquisa acadêmica para a realidade comercial através do desenvolvimento de infraestruturas, ferramentas e ambientes que suportam a criação de sistemas ubíquos. Muitas pesquisas em computação apontam características que devem existir em qualquer sistema ubíquo, tais como [Sym10, Kur07, Sat01]: contexto, localização, sensores e entidades capazes de perceber e responder às alterações no ambiente. Um dos problemas atuais, no entanto, é que existem muitas abordagens para o desenvolvimento de sistemas ubíquos que utilizam ou propõem soluções *ad-hoc* encontradas na literatura [BC06, SVP10]. Do ponto de vista da Engenharia de Software, abordagens imaturas para o desenvolvimento de aplicações ubíquas levam os desenvolvedores a reimplementar módulos comuns de infraestrutura a partir do zero, tornando as soluções específicas a um determinado contexto. Visando reduzir o esforço e o retrabalho no desenvolvimento de aplicações ubíquas, são necessários novos modelos que ofereçam aos desenvolvedores conceitos em alto-nível de abstração.

## 1.1 Questão de Pesquisa

Desenvolver aplicações onde a ubiquidade é um conceito inerente a qualquer agente executando em um ambiente ubíquo ainda é um desafio. Devido a quantidade e diversidade de propostas desenvolvidas nos últimos anos, a falta de integração entre tais propostas, e a falta de uma maneira uniforme de desenvolver aplicações multiagentes ubíquas capazes de tirar proveito das características das duas áreas motivaram a

realização desta pesquisa. Por esse motivo, esta pesquisa tem como foco o estudo dos aspectos relacionados à construção de uma linguagem de programação que suporte o desenvolvimento de sistemas multiagentes ubíquos.

Uma vez definido o foco de estudo, apresenta-se a questão de pesquisa que motivou a realização deste trabalho:

*"É possível facilitar o desenvolvimento de sistemas ubíquos dinâmicos e complexos através do uso de sistemas com múltiplos agentes autônomos que sejam intrinsicamente cientes de localização?".* Por facilitar o desenvolvimento, entende-se a utilização de abstrações em alto nível para a representação de elementos importantes de sistemas multiagentes e computação ubíqua. Já por dinâmicos e complexos, entende-se a natureza das duas áreas que esta proposta está inserida, dado que os sistemas ubíquos, por lidarem principalmente com ambientes reais, possuem diversas informações e elementos que são utilizados para caracterizar seus ambientes.

## 1.2 Objetivos

Do ponto de vista de contribuição teórica, a tese aqui proposta contribui para a engenharia de software de duas formas: na definição de um metamodelo contendo uma compilação de conceitos de computação ubíqua comumente encontrados em *frameworks* e aplicações ubíquas, que ao mesmo tempo modela conceitos de sistemas multiagentes, e na criação de uma linguagem de programação para o desenvolvimento de aplicações multiagentes ubíquas (baseada no metamodelo proposto) cujas primitivas de sistemas multiagentes e computação ubíqua são tratadas como elementos de primeira ordem.

O objetivo geral desta pesquisa é permitir a implementação de sistemas multiagentes ubíquos, onde a ubiquidade é um conceito inerente aos agentes executando em um ambiente.

Os objetivos específicos são:

- Prover o entendimento dos principais conceitos que caracterizam um sistema ubíquo;
- Prover a formalização dos conceitos de agentes e computação ubíqua através de um metamodelo unificado;
- Prover uma linguagem de programação independente de metodologia e plataforma de execução (hardware e software) para o desenvolvimento de sistemas multiagentes ubíquos, cujas primitivas sejam baseadas em um modelo unificado;

- Permitir o desenvolvimento de agentes (incluindo sua estrutura interna) e sistemas multiagentes;
- Prover aos agentes a noção de localização e dispositivos em um ambiente ubíquo;
- Permitir a decomposição de um ambiente em múltiplos espaços mapeáveis através de localizações simbólicas;
- Tornar transparente para o desenvolvedor a utilização dos recursos em um ambiente ubíquo, tais como dispositivos de hardware e software;
- Possibilitar a utilização de diferentes tecnologias de posicionamento;
- Projetar um ambiente de execução customizável capaz de ser adaptado para executar em diferentes plataformas de software.

## 1.3 Metodologia de Pesquisa

Esta pesquisa se caracteriza como um estudo exploratório, sendo a principal estratégia de pesquisa utilizada, de acordo com a classificação Projeto e Criação [Oat06].

Pesquisas guiadas pela estratégia projeto e criação têm foco no desenvolvimento de novos produtos de TI (Tecnologia da Informação), também chamados de artefatos [Oat06]. Geralmente o novo produto de TI é um sistema baseado em computador, mas ele também pode ser algum elemento relacionado ao processo de desenvolvimento, como um modelo ou um método [Oat06]. Cabe salientar que, para projetos que seguem essa estratégia serem considerados de fato uma pesquisa (e não somente uma prova de conhecimento técnico), eles devem demonstrar, segundo Oates, “qualidades acadêmicas, como análise, discussão, justificação e avaliação crítica” [Oat06].

Para o desenvolvimento da linguagem proposta, será utilizado o processo de desenvolvimento prototipal. Já para a avaliação da linguagem, será mostrada uma prova de conceito para demonstrar a utilização e comportamento dos principais elementos da solução proposta. As quatro grandes etapas que constituem esta pesquisa estão ilustradas na Figura 1.

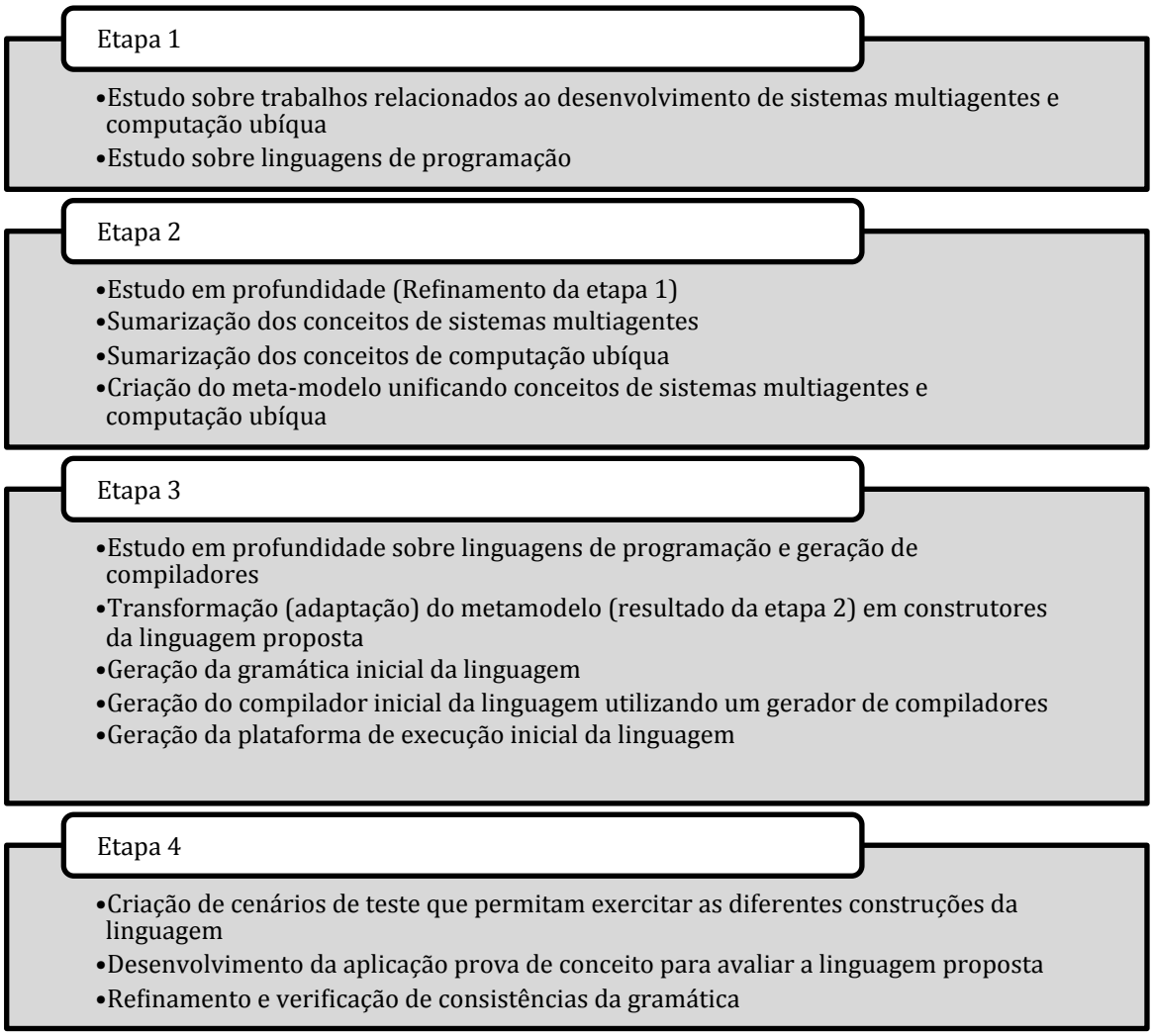


Figura 1 – Ilustração da metodologia de pesquisa proposta.

**Etapa 1 – Reconhecimento inicial do problema**: essa etapa foi eminentemente exploratória. Nela foram estudados conceitos e aspectos relevantes à formulação da linguagem proposta, como os conceitos de sistemas multiagentes e computação ubíqua (cujas definições encontram-se listadas no Capítulo 2). Também, durante os estudos preliminares, não foram encontrados trabalhos que sumarizassem os conceitos de computação ubíqua e como estes podem ser unificados aos conceitos de sistemas multiagentes, fato que confirmou a necessidade de uma revisão aprofundada sobre o tema.

**Etapa 2 – Estudo em profundidade sobre sistemas multiagentes e computação ubíqua**: nessa etapa, visando a identificação dos principais conceitos de sistemas multiagentes e computação ubíqua, primeiramente foram analisados os estudos identificados durante a Etapa 1. A partir dessa análise, inicialmente foi definido o conjunto de conceitos fundamentais para sistemas multiagentes e computação ubíqua. Esses

conceitos são frequentemente encontrados em aplicações ubíquas, e suas definições foram baseadas na literatura existente. O resultado desta etapa consiste de um metamodelo (U-MAS [EB11]) que promove um conjunto de conceitos genéricos, capazes de suportar o desenvolvimento de aplicações multiagentes ubíquas. Esse metamodelo torna os conceitos de computação ubíqua inerentes a qualquer agente executando em um ambiente ubíquo (vide Capítulo 3 para maiores informações sobre o metamodelo).

**Etapa 3 – Proposta da linguagem de programação de sistemas multiagentes ubíquos**: o objetivo da terceira etapa foi a proposição de uma linguagem de programação que possibilite o desenvolvimento de sistemas multiagentes ubíquos, cujos construtores de ubiquidade sejam inerentes aos conceitos de agentes e sistemas multiagentes. A linguagem é baseada em dois metamodelos: o FAML [Bey09] e o U-MAS [EB11]. O FAML é um metamodelo genérico para o desenvolvimento de sistemas multiagentes. O U-MAS é um metamodelo que estende o FAML adicionando a ele conceitos de computação ubíqua comumente utilizados no desenvolvimento de aplicações. O metamodelo final preservou os conceitos fundamentais do FAML, ao mesmo tempo em que modela conceitos de computação ubíqua no mesmo nível de abstração dos conceitos de agentes. Como resultado desta etapa tem-se: a gramática completa da linguagem, o compilador e o ambiente de execução capaz de executar programas escritos na linguagem proposta.

**Etapa 4 - Avaliação:** a partir dos conhecimentos teóricos e empíricos gerados nas fases anteriores, agregado à experiência dos pesquisadores na área, foram conduzidos estudos para avaliar a linguagem proposta ao término da Etapa 3. Desta forma, a quarta e última etapa desta pesquisa consistiu no desenvolvimento de uma aplicação prova de conceito que isola uma situação que permite exercitar os diferentes construtores providos pela linguagem. A prova de conceito (representada neste caso pelo desenvolvimento de uma aplicação) permitiu mostrar que os requisitos inicialmente identificados foram contemplados pela linguagem, atingindo-se assim os objetivos da pesquisa.

## 1.4 Organização da Tese

O restante deste trabalho está organizado da seguinte forma: o Capítulo 2 apresenta o referencial teórico sobre agentes e sistemas multiagentes, computação ubíqua e demais conceitos necessários para o trabalho. O Capítulo 3 apresenta como foi construído o metamodelo U-MAS, bem como seus quatro conjuntos de conceitos, que posteriormente foram utilizado como base para a concepção da linguagem proposta. No Capítulo 4 é apresentada a linguagem AA, detalhando sua gramática e funcionalidades.

Além da gramática é mostrada a arquitetura do ambiente de execução, suas principais classes e como estender seus principais pontos de flexibilidade. Também é detalhado o processo de construção do compilador da linguagem e a organização interna de seus componentes.

No Capítulo 5 são apresentados os estudos conduzidos para avaliar a proposta, que correspondem ao desenvolvimento de uma aplicação prova de conceito utilizando a linguagem proposta, o desenvolvimento de uma extensão da plataforma Jason, adicionando a ela conceitos de ubiquidade, e, o desenvolvimento de uma extensão do *framework* MoCA, adicionando a ele conceitos de agentes. Ao final são feitas considerações que buscam esclarecer ao leitor a vantagem em utilizar uma linguagem projetada para um domínio específico frente adaptações de abordagens existentes. No Capítulo 6 são apresentados alguns trabalhos relacionados. Por fim, o Capítulo 7 apresenta as considerações finais e oportunidades para trabalhos futuros.

# 2 REFERENCIAL TEÓRICO

## 2.1 Sistemas Multiagentes

Segundo Weiss [Wei99b], um agente pode ser definido como um sistema computacional situado em algum ambiente e capaz de agir de forma autônoma para atingir um objetivo. Em diversas áreas, o termo “agente” é utilizado de forma vaga. Segundo Ferber [Fer99], existem definições comuns que podem caracterizar um agente: um agente pode ser visto como uma entidade física ou virtual que é capaz de: (i) atuar em um ambiente; (ii) comunicar-se diretamente com outros agentes; (iii) guiar-se por um conjunto de objetivos; (iv) possuir seus próprios recursos; (v) perceber o ambiente (até determinado ponto); (vi) possuir uma visão limitada do ambiente; e (vii) possuir comportamento que tende a buscar seus objetivos, levando em conta seus recursos, capacidades, percepções e comunicações que ele recebe.

A Figura 2 apresenta uma visão abstrata em alto nível de um agente. No diagrama, o agente desempenha ações capazes de alterar o ambiente além de possuir sensores capazes de perceber alterações no ambiente.

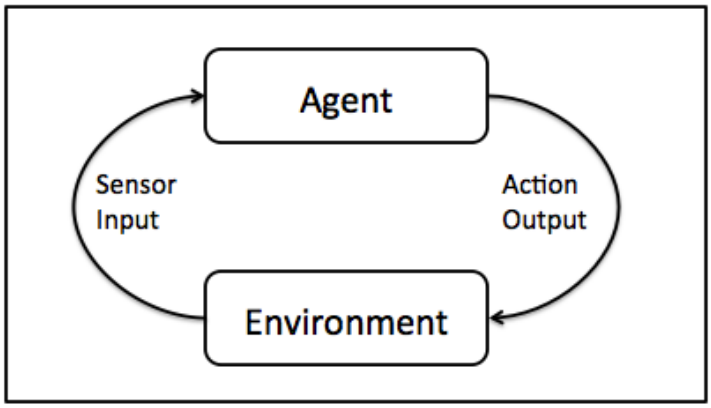


Figura 2 – Agente e seu ambiente [Fer99].

Agentes são capazes de atuar, e não somente raciocinar [Fer99]. O conceito de ação, que é fundamental em sistemas multiagentes, é baseado no fato de que agentes desempenham ações capazes de alterar seu ambiente, e dessa forma suas futuras tomadas de decisão através da percepção do estado do ambiente. Um agente também é capaz de comunicar-se com outros agentes, e essa é uma das maneiras pelas quais agentes interagem.

Agentes são dotados de *autonomia*. Isso significa que eles não são guiados diretamente por comandos vindos de um usuário (ou outro agente), mas sim por um

conjunto de objetivos, que podem ser objetivos individuais a serem perseguidos ou funções de sobrevivência que o agente visa aperfeiçoar. Um agente é então independente, isto é, ele pode aceitar ou rejeitar requisições vindas de outros agentes. Tais características diferenciam agentes de objetos, considerando o paradigma Orientado a Objetos (OO). Em OO, objetos comunicam-se através de mensagens, porém, um objeto sempre responde a uma mensagem, que significa a chamada a um método do objeto. Um agente, ao contrário, decide baseado em suas regras de raciocínio, se deve ou não responder a uma mensagem recebida.

Outras características podem ser encontradas em agentes, tais como [Fer99, Wei99b]: a *adaptação*, que possibilita ao agente modificar, em algum grau, os seus comportamentos devido às mudanças ocorridas no ambiente; o *aprendizado***,** que permite ao agente modificar seu comportamento baseado na sua experiência; a *racionalidade*, que permite ao agente selecionar suas ações com base em seus objetivos; a *mobilidade***,** que possibilita ao agente migrar de um ambiente para outro.

Para a construção de sistemas complexos é interessante considerar a utilização de vários agentes que desempenham tarefas voltadas à obtenção de seus objetivos e que estão de acordo com os objetivos de todo o sistema. Um sistema que possui vários agentes atuando em um ambiente em busca de seus objetivos é denominado sistema multiagente (SMA).

Assim como a definição de agentes de software, a definição de sistemas multiagentes também é variada na literatura [Fer99, Wei99, WJK99]. Os sistemas multiagentes estão ligados com o comportamento de uma coleção de agentes autônomos. Eles formam uma espécie de rede para resolução de problemas que estão além de suas capacidades individuais.

Os agentes atuam e existem em um ambiente aberto ou fechado que pode ser computacional ou físico [Wei99b]. Um ambiente deve permitir que os agentes atuem efetivamente e interajam entre si de forma produtiva. Ele proverá uma infraestrutura computacional de forma a proporcionar protocolos que permitam a comunicação e interação entre os agentes. Os protocolos de comunicação permitem aos agentes trocarem e entenderem mensagens, isto é, realizarem conversação, que de um modo geral, pode ser visto como a troca estruturada de mensagens [Wei99b].

Os sistemas multiagentes são apropriados para domínios que são naturalmente distribuídos. Com isso, o uso dos conceitos de agentes para a engenharia de sistemas

distribuídos provê diversas vantagens na redução de complexidade inerente a estes tipos de sistemas, como a autonomia e interações em alto nível [Jen01]. A interação entre agentes permite a construção de sociedades de agentes. A autonomia de um agente possibilita a interação com ambos ambientes físicos ou virtuais, tornando, assim, o paradigma de agentes apropriado para ser utilizado em computação ubíqua.

Em um SMA, os agentes podem ser organizados em sociedades, formando grupos e desempenhando diversos papéis. Um grupo pode definir um conjunto de papéis, enquanto que os papéis definem os comprometimentos associados aos agentes que os desempenham [Wei99b]. A sociabilidade é importante para a cooperação, que promove a mudança do paradigma cliente-servidor para um paradigma flexível e distribuído que as aplicações modernas necessitam, e onde a tecnologia de agentes pode encontrar seu grande potencial.

Os agentes em um SMA são projetados para resolver problemas com certos níveis de abstração e com as quais possam lidar, de acordo com os recursos disponíveis e os conhecimentos que ele possui ou possa buscar com outros agentes. Em um ambiente com recursos limitados, por exemplo, os agentes devem coordenar suas atividades para cumprirem seus próprios interesses ou para satisfazerem os objetivos do grupo [Wei99b]. As ações de múltiplos agentes necessitam ser coordenadas devido à existência de dependência entre elas, além da necessidade de manter as restrições globais do sistema.

Podemos dizer que os agentes estão cooperando se eles assumem ações em comum após identificarem e adotarem um objetivo comum [Fer99]. Em um ambiente cooperativo, a troca de informações entre agentes é fundamental, havendo assim o compartilhamento de informações. A cooperação entre agentes pode ser caracterizada de duas formas principais:

- Partilha de resultados: ocorre após a conclusão de um objetivo. Nesse caso, o agente verifica se existem outros agentes interessados nas informações provenientes do alcance de seu objetivo.
- Partilha de tarefas: ocorre quando um agente detecta que não possui capacidade ou informações suficientes para executar determinada tarefa. Para isso, o agente deve verificar se existem outros agentes capazes de lhe ajudar. Esse tipo de partilha pode ser vista como um balanceamento da carga computacional do sistema.

Tanto na cooperação quanto na competição, é preciso que os agentes planejem e executem suas ações de uma forma coordenada. O problema da coordenação consiste no gerenciamento das interdependências entre as atividades desempenhadas pelos agentes [WJK99]. Para isso, mecanismos de coordenação são essenciais se as atividades que um agente possui ocasionam a interação de alguma maneira com outros agentes.

Em sistemas cooperativos, a coordenação de agentes visa garantir que [Wei99b, Fer99]: (i) todas as partes componentes de um problema estejam incluídas nas atividades de pelo menos um agente; (ii) os agentes interajam de forma a permitir que suas atividades sejam desenvolvidas e integradas no sentido de uma solução global; (iii) os membros do grupo de trabalho atuem de forma determinada e consistente; (iv) o grupo de agentes respeite as restrições globais à solução do problema; e, (v) existam procedimentos que garantam a harmonia na execução de uma única ação de forma conjunta por mais de um agente. Alguns requisitos são fundamentais para permitir a coordenação [Wei99b, Fer99]: (i) comunicação entre os agentes; (ii) o reconhecimento das potenciais interações entre os planos de ação dos agentes; e, (iii) a capacidade de negociação.

As tecnologias de agentes e sistemas multiagentes vêm exercendo um papel importante no desenvolvimento de software. A próxima seção apresenta algumas das abordagens para a engenharia de sistemas multiagentes.

## 2.2 Engenharia de Software para Sistemas Multiagentes

O projeto de sistemas de software distribuídos possui diversos desafios, dentre eles podemos citar [BGZ04]: determinar os componentes que uma aplicação distribuída deve conter, organizar os componentes da aplicação e determinar as funções de cada componente a fim de implementar sistemas distribuídos escaláveis e flexíveis.

Nos últimos anos, junto com o aumento da aceitação de computação baseada em agentes como um novo paradigma de engenharia de software passaram a existir diversas iniciativas de pesquisa relacionadas à identificação e definição de modelos, ferramentas e técnicas para suportar o desenvolvimento de sistemas complexos em termos de SMAs. Estas pesquisas são agrupadas sobre o termo *Agent-Oriented Software Engineering* (AOSE) e podem ser organizadas como sugerido em [BGZ04]:

- Conceitos e abstrações de Engenharia de Software Orientada a Agentes: busca esclarecer a razão pela qual *agentes* é uma abordagem adequada para o desenvolvimento de sistemas complexos (diferenciando das abordagens tradicionais existentes);
- Metodologias para o desenvolvimento de software baseado em agentes: abordagens de propósito geral que guiam o desenvolvimento de SMAs. Dentre essas metodologias, podemos citar Gaia [WJK00] e Tropos [BPG+04].
- Metodologias de propósitos especiais: metodologias criadas para um determinado domínio, ou que exploram características específicas e não gerais de SMAs (como por exemplo, Sistemas Multiagentes adaptativos). Dentre essas metodologias podemos citar Adelfe [BGP+ 02], MESSAGE [BGZ04], SADDE [BGZ04] e Prometheus [PW02, PW04].
- Ferramentas e Infraestruturas: enquanto as metodologias definem um processo para a construção de um SMA, somente a disponibilidade de ferramentas e infraestruturas de software podem tornar o resultado desse processo em um sistema bem projetado.

Os princípios de agentes e sistemas multiagentes demonstram grande potencial em relação a alguns dos desafios apontados, devido à sua modularização inerente e pela facilidade com a qual eles podem ser combinados para formar novas aplicações. A AOSE distingue-se de orientação a objetos, pois ela considera conceitos de agência tais como *objetivos*, *papéis*, *contexto* e *mensagens* como entidades de primeira ordem. A orientação a agentes oferece abstrações em alto nível e mecanismos que tratam questões como representação de conhecimento e raciocínio, coordenação e cooperação entre partes heterogêneas e autônomas.

Como as técnicas de Engenharia de Software possuem limitações quanto à representação de requisitos específicos de SMAs [WJK99], vêm sendo propostas algumas arquiteturas e linguagens que incorporam conceitos de agência nativos em seus modelos. Algumas destas propostas tiveram origem em esforços de consórcios de instituições de pesquisa e em empresas de grande porte, que procuram utilizar SMAs como solução de problemas distribuídos complexos [OMG00].

Para que uma plataforma ou linguagem de implementação de SMAs possa ser considerada completa, é preciso tanto o suporte ao desenvolvimento da estrutura interna dos agentes quanto o suporte à criação da infraestrutura de atuação e de organização

social dos agentes. Dentre as abordagens disponíveis para a implementação de SMAs, tais como o JADE [JADE11], SemantiCore [BEC07], Jason [BHW07] e 3APL [3APL01] que serão apresentadas a seguir, são poucas as que oferecem suporte total à sua criação, como será mostrado a seguir, através da descrição das abordagens citadas. Estas abordagens foram escolhidas porque possuem, entre outros aspectos, implementação disponível para download que pode ser utilizada por usuários para o desenvolvimento de aplicações. Ainda nesta seção, após a descrição das linguagens Jason e 3APL, será apresentado o metamodelo FAML.

### 2.2.1 JADE

O JADE (*Java Agent DEvelopment Framework*) é um *framework* totalmente implementado na linguagem Java. Seu objetivo é dar suporte ao desenvolvimento de aplicações de SMAs através de uma plataforma (que segue as especificações da FIPA - *Foundation for Intelligent Physical Agents*) e de um conjunto de ferramentas gráficas que suportam as fases de desenvolvimento e validação [JADE11].

O JADE permite o desenvolvimento de sistemas capazes de trabalhar de uma maneira proativa (de acordo com regras predefinidas), de comunicar-se e negociar diretamente com outras partes do sistema e de coordenar-se a fim de solucionar problemas complexos de maneira distribuída [BCT+07; JADE11].

Dentre a lista de características do JADE, apresentada por Bellifemine e co-autores em [BCT+07], destacam-se:

- Plataforma de agentes distribuída: o JADE pode ser distribuído em várias máquinas, desde que elas possam ser conectadas via tecnologia RMI (*Remote Method Invocation*). Apenas uma aplicação Java e uma Java *Virtual Machine* são executadas em cada máquina. Os agentes são implementados como *threads* Java e são inseridos dentro de repositórios de agentes chamados de *Agent Containers*, que provêem todo o suporte para a execução dos mesmos.
- Interface gráfica: interface visual que permite gerenciar vários agentes e repositórios de agentes, inclusive remotamente.
- Ferramentas de depuração: ferramentas que ajudam no desenvolvimento e na depuração de aplicações multiagentes baseadas em JADE.
- Transporte de mensagens: transporte de mensagens no formato FIPA-ACL [FIPA13] dentro da mesma plataforma de agentes.

- IDE de agentes de acordo com as especificações da FIPA: o JADE contém um sistema gerenciador de agentes (*Agent Management System*), um facilitador de diretórios (*Directory Facilitator*) e um canal de comunicação dos agentes (*Agent Communication Channel*). Todos esses componentes são automaticamente carregados quando o ambiente é iniciado.

O JADE inclui bibliotecas de classes Java para o desenvolvimento dos agentes e do ambiente de execução que provê os serviços básicos. Esse ambiente deve estar ativo em um determinado dispositivo antes que um ou mais agentes sejam executados em um dispositivo. Como já mencionado, cada instância do JADE cria um ambiente de execução chamado *container*, conforme mostrado na Figura 3.

O conjunto de todos os *containers* em execução forma a *plataforma* e provê uma camada homogênea que abstrai dos agentes a complexidade e diversidade do hardware e sistema operacional dos dispositivos.

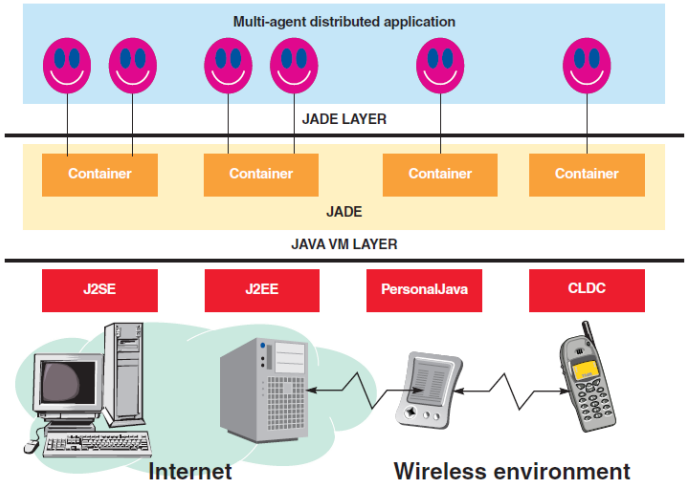


Figura 3 – A arquitetura do framework JADE [JADE11].

Uma aplicação multiagentes no JADE é composta de agentes providos pela plataforma JADE, e de um conjunto de agentes de aplicação criados pelo desenvolvedor da aplicação. Os agentes são implementados através de uma classe Java contendo um conjunto de classes internas que realizam diferentes comportamentos do agente. Um agente pode ser composto de diversos comportamentos, que podem executar uma vez (chamados de *one-shot behaviours*), ou diversas vezes (chamados de *cyclic behaviours*).

As classes de agentes são baseadas em um método chamado *setup*, que realiza a inicialização do agente, e do método chamado *takedown*, que realiza as operações de finalização e limpeza ao final da execução do agente. Os comportamentos do agente são baseados em um método chamado *action*, que define as operações a serem realizadas

quando o comportamento é executado. Além disso, os comportamentos cíclicos podem ter outro método, chamado *done*, que retorna um valor booleano indicando se os comportamentos finalizaram ou não seus ciclos de execução.

A Listagem 1 mostrado o código de uma classe de agente ilustrando sua estrutura e a definição do esqueleto de código de um comportamento.

Listagem 1 – Definição de um agente e comportamento no *framework* JADE.

```
 1  public class AgentClassName extends Agent {
 2        // definição de atributos
 3
 4        protected void setup () {
 5              // código de inicialização do agente
 6        }
 7
 8        protected void takeDown() {
 9              // operações de finalização
10        }
11
12        private class BehavClassName extends Behaviour {
13              // definição de atributos
14
15              public void action(){
16                    // método executor do comportamento
17              }
18
19              public boolean done() {
20                    // retorna true se a
21                     // execução foi completada
22              }
23        }
24
25        // demais classes para comportamentos
26  }
27
```

### 2.2.2 SemantiCore

O SemantiCore é estruturado como um *framework* que abstrai características de distribuição de computação e provê primitivas em alto nível de abstração para a criação sistemas multiagentes para a Web [BL04]. O SemantiCore, apresentado inicialmente em 2004 [BL04], surgiu a partir da extensão da arquitetura Web Life [Rib02] e atualmente encontra-se disponível na versão 2006 – SemantiCore 2006 [ELB06].

O SemantiCore é dividido em dois modelos: o modelo do agente (Figura 4), responsável pelas definições internas dos agentes, e o modelo do domínio (Figura 5), responsável pela definição da composição do domínio e suas entidades administrativas. Os dois modelos dispõem de pontos de flexibilidade (*hotspots*) permitindo aos desenvolvedores associarem diferentes padrões, protocolos e tecnologias.

O modelo do agente possui uma estrutura orientada a componentes, onde cada componente contribui para uma parte essencial do funcionamento do agente, agregando todos os aspectos necessários a sua implementação. São quatro os componentes básicos do agente: sensorial, decisório, executor e o efetuador.

O componente sensorial permite que o agente recupere objetos a partir do ambiente. Ele armazena os diversos sensores definidos pelo desenvolvedor (cada sensor captura um tipo diferente de objeto no ambiente) e também verifica se algum destes sensores deve ser ativado pelo recebimento de um objeto no ambiente. O componente decisório encapsula o mecanismo de tomada de decisão do agente. Ele representa um dos pontos de flexibilidade do *framework*, podendo ser implementado em combinação com *frameworks* existentes, como o Jena [JENA11], possibilitando o uso de máquinas de inferência neste componente.

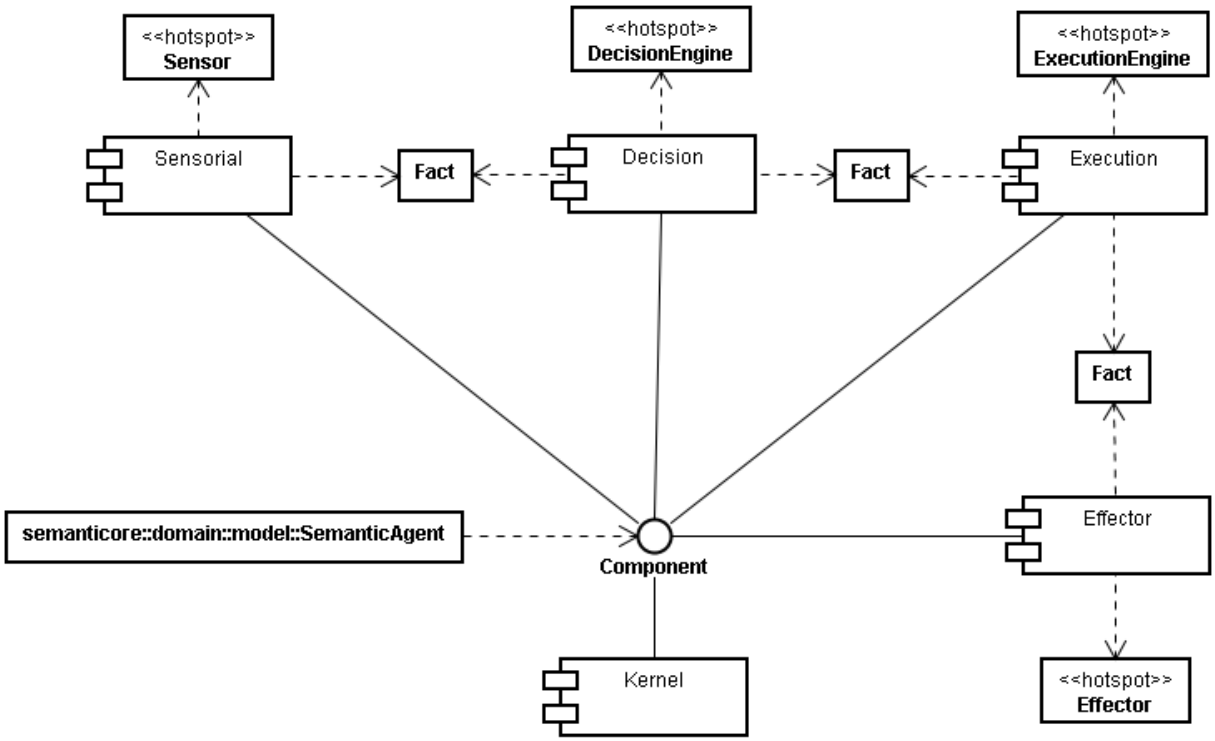


Figura 4 – Arquitetura de componentes do agente no SemantiCore [BEC07].

O componente executor é responsável por armazenar e controlar os planos de ação que serão executados pelo agente. Por fim, o componente efetuador recebe dados dos outros componentes e encapsula estes em objetos para serem transmitidos no ambiente. Similarmente ao componente sensorial, cada tipo de objeto a ser transmitido ao ambiente requer um efetuador apropriado no agente.

Para que um agente possa atuar, é necessário que ele esteja situado em um ambiente. No SemantiCore, este ambiente é denominado domínio semântico (Figura 5). Um domínio semântico requer um domínio Web para operar. Cada domínio semântico é composto por algumas entidades administrativas, como o Controlador de Domínio (*Domain Controller*) e o Gerente de Ambiente (*Environment Manager*). O Controlador de Domínio é responsável por registrar os agentes no ambiente, pela recepção de agentes móveis vindos de outros domínios e também pela manutenção e execução de aspectos relacionados à segurança. O Gerente de Ambiente representa uma ponte entre o domínio semântico do SemantiCore e os domínios Web convencionais.

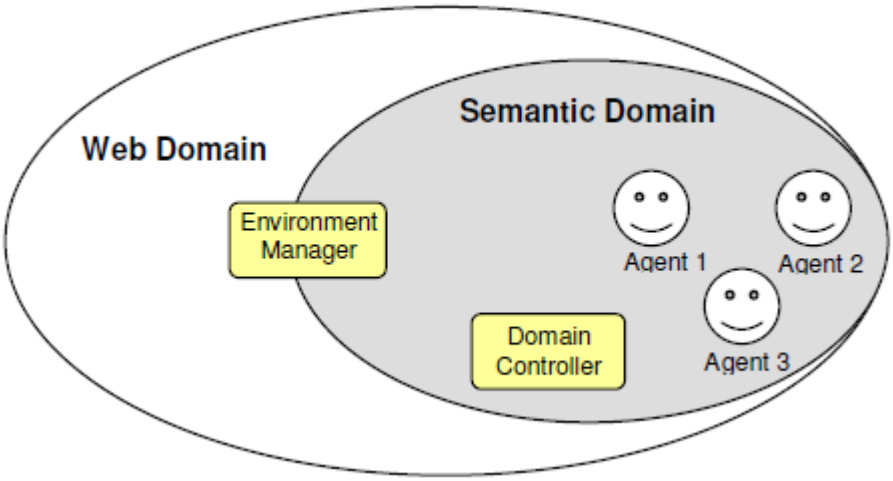


Figura 5 – Representação do modelo de domínio do SemantiCore [ELB06].

O SemantiCore é implementado em Java (compatível com a versão *standard* 5). Embora o SemantiCore tenha sido projetado para a Web Semântica, ele pode ser utilizado para a implementação de SMAs de propósitos gerais. O seu ambiente de execução permite a distribuição do domínio em diferentes máquinas, bem como a distribuição dos componentes dos agentes entre os nós que formam o domínio.

O SemantiCore também separa a lógica de comunicação da plataforma. Existem dois barramentos para o tráfego de mensagens: dados e controle. No barramento de dados trafegam todas as mensagens trocadas entre os agentes, em formato aberto e que pode ser customizado pelo desenvolvedor. Já no barramento de controle trafegam todas as mensagens de controle da plataforma (como por exemplo, sincronização dos domínios, estabelecimento de conexão), em formato proprietário e fixo.

Em um domínio distribuído, a primeira instância a entrar em execução é considerada a instância principal e contém, portanto, o Controlador de Domínio e o Gerente de Ambiente. As demais são chamadas de instâncias remotas. A distribuição de

agentes permite que os componentes do agente estejam espalhados nas diferentes partes do domínio. Com esta distribuição é possível, por exemplo, colocar componentes que necessitam de maior poder computacional em máquinas de maior porte.

No SemantiCore, a localização de um agente distribuído é armazenada de acordo com a localização de seu componente sensorial, isto porque é através deste componente que o agente recebe informações do ambiente. Quando o desenvolvedor determina o caminho de dados entre os componentes, é criada uma tabela de roteamento, que contêm o endereço de cada componente do agente. Com essa tabela, no momento que um componente solicitar o envio de informações, pode-se recuperar o componente destino e a sua localização. Em agentes distribuídos, as mensagens entre componentes são enviadas pelo barramento de controle, sendo a distribuição e a localização dos componentes responsabilidade do SemantiCore.

Um agente no SemantiCore deve estender a classe *SemanticAgent*. O agente inicia sua execução através da chamada ao método *setup* (Figura 6). Durante o *setup,* o desenvolvedor descreve a inicialização do agente, podendo, por exemplo, criar sensores, fatos, regras, efetuadores, ações, planos de ação e objetivos para o agente. Todas essas estruturas são criadas utilizando classes SemantiCore, formando assim, o modelo de referência do agente.

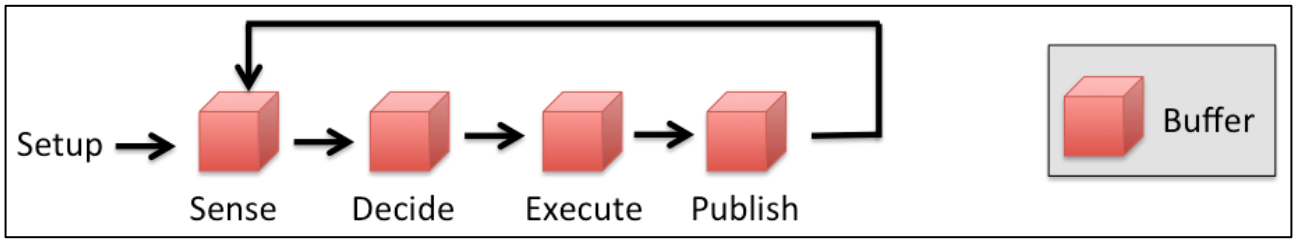


Figura 6 – Ciclo de vida do agente no SemantiCore [BEC07].

O método *setup* executa somente uma vez quando o agente é criado no ambiente. Após iniciado e registrado pelo Controlador de Domínio, ele realiza basicamente um laço com quatro operações durante sua execução: perceber o ambiente (*sense*), decidir de acordo com a informação sensorial (*decide*), executar ações dependendo das decisões tomadas (*execute*), e publicar informações de volta ao ambiente (*publish*). Este ciclo de vida é gerenciado automaticamente pelo SemantiCore.

A Listagem 2 mostra o código de uma classe de agente ilustrando sua estrutura. Nesse exemplo é declarado um agente e uma ação. No método *setup* do agente é criado um plano de ação (*ActionPlan*) e a ele é adicionada uma ação, declarada posteriormente

através da classe interna *ActionName*. Na classe que representa a ação existe o método *exec*, que consiste do método executor da ação. Nele deve ser colocado todo o código a ser executado ao ser iniciada a ação.

Listagem 2 – Definindo um agente no *framework* SemantiCore.

```
 1  public class AgentClassName extends SemanticAgent {
 2
 3          // declaracao de atributos
 4
 5          protected void setup() {
 6                  // Inicialização do agente
 7
 8                  ActionPlan planIdentifier = new ActionPlan("PlanName");
 9
10                  planIdentifier.addAction(new ActionName());
11
12                  addActionPlan(planIdentifier);
13          }
14  }
15
16  class ActionName extends Action {
17
18          // declaracao de atributos
19
20          public void exec() {
21                  // metodo executor da acao
22          }
23  }
```

### 2.2.3 Jason

A plataforma Jason [BHW07] implementa a semântica operacional de uma versão estendida da linguagem AgentSpeak(L). Ela provê uma infraestrutura para o desenvolvimento de agentes BDI (*Belief-Desire-Intention*) com uma série de funcionalidades customizáveis. A linguagem AgentSpeak foi originalmente projetada para a programação de um único agente. No Jason, foi adicionado um mecanismo de comunicação, estendendo a linguagem AgentSpeak e provendo um mecanismo para a troca de crenças e planos, através da utilização de comunicação baseada em atos de fala.

De modo geral, um agente no Jason consiste da especificação de um conjunto de crenças, que formam sua base inicial de crenças, uma lista de objetivos e um conjunto de planos. As crenças formam o componente informacional do agente. Os objetivos indicam os estados que o agente visa atingir (chamados de *achievement goals* e denotados pelo símbolo “!”), ou consultas por informações na sua base de crenças (chamados *test goals* e denotados pelo símbolo “?”).

Todo plano possui um evento disparador (que define quais eventos podem iniciar a execução de um plano), um contexto (conjunto de crenças que devem ser verdadeiras para um plano ser considerado aplicável) e uma sequência de ações básicas ou sub-

objetivos que o agente deve perseguir. Planos são executados pela adição (“+”) ou deleção (“-”) de crenças ou objetivos (“atitudes mentais” dos agentes).

As ações internas podem ser usadas no contexto ou no corpo de um plano. Essas ações são definidas pelo desenvolvedor utilizando a linguagem Java e executam internamente no agente.

A Figura 7 mostra um exemplo de código contendo um conjunto inicial de crenças e planos para um agente responsável pelo desarmamento de bombas [BHW07]. Inicialmente o agente acredita que possui habilidades para desarmar bombas plásticas e biológicas, mas não possui a habilidade de desarmar bombas nucleares. Ele sabe que “field1” é um lugar seguro para deixar uma bomba que ele não é capaz de desarmar. Ele possui quatro planos, identificados pelos rótulos “@p1” a “@p4”.

No plano 4 (rótulo “@p4”) é ilustrado um exemplo de *test-goal*, em que o agente consulta em sua base de crenças sobre onde levar uma bomba, e um exemplo de ação interna (`.send(...)`) utilizada para enviar uma mensagem.

```
skill(plasticBomb).
skill(bioBomb).
~skill(nuclearBomb).

safetyArea(field1).

@p1
+bomb(Terminal, Gate, BombType) : skill(BombType)
      <- !go(Terminal, Gate);
            disarm(BombType).
@p2
+bomb(Terminal, Gate, BombType) : ~skill(BombType)
      <- !moveSafeArea(Terminal, Gate, BombType).

@p3
+bomb(Terminal, Gate, BombType)
      : not skill(BombType) & not ~skill(BombType)
      <- .broadcast(tell, alter).
@p4
+!moveSafeArea(T,G,Bomb) : true
      <- ?safeArea(Place);
      !discoverFreeCPH(FreeCPH);
      .send(FreeCPH, achieve,
            carryToSafePlace(T,G,Place,Bomb)).
...
```

Figura 7 – Exemplo de Planos escritos em AgentSpeak [BDD+05].

A configuração do sistema multiagente a ser executado na plataforma Jason é feita através de um arquivo texto, cuja sintaxe é específica conforme a gramática mostrada na Figura 8. Nessa gramática, `<NUMBER>` é utilizado para números inteiros, `<ASID>` são identificadores em AgentSpeak, que devem iniciar com uma letra em

minúsculo, <ID> é um identificador (como usual), e <PATH> é utilizado, assim como em um sistema operacional, para definir um caminho para os arquivos.

O <ID> utilizado após a palavra-chave MAS e indica o nome da sociedade de agentes. A palavra infrastructure é usada para especificar qual infraestrutura para a execução do sistema multiagente deverá ser utilizada.

```
mas → "MAS" <ID> "{"
        [ infrastructure ]
        [ environment ]
        [ exec_control ]
        agents
        "}"

infrastructure → "infrastructure" ":" <ID>
environment → "environment" ":" <ID> [ "at" <ID> ]
exec_control → "executionControl" ":" <ID> [ "at" <ID> ]
agents → "agents" ":" ( agent )+
agent → <ASID>
         [ filename ]
         [ options ]
         [ "agentArchClass" <ID> ]
         [ "beliefBaseClass" <ID> ]
         [ "agentClass" <ID> ]
         [ "#" <NUMBER> ]
         [ "at" <ID> ]
         ";"

filename → [ <PATH> ] <ID>
options → "[" option ( "," option )* "]"
option → "events" "=" ( "discard" | "requeue" | "retrieve" )
           |  "intBels" "=" ( "sameFocus" | "newFocus" )
           |  "nrcbp" "=" <NUMBER>
           |  "verbose" "=" <NUMBER>
           |  <ID> "=" ( <ID> | <STRING> | <NUMBER> )
```

Figura 8 – Gramática BNF para a configuração de um SMA no Jason [BDD+05].

A seguir, o ambiente (environment) deve ser referenciado através do nome de uma classe Java que implementa o ambiente. A palavra agents é utilizada para a definição do conjunto de agentes que farão parte do sistema multiagente. Um agente é especificado inicialmente por um nome simbólico (utilizando a construção <ASID>). Esse nome será utilizado para os agentes referenciarem outros agentes na sociedade (comunicação entre agentes). Por fim, um nome de arquivo pode ser opcionalmente informado, indicando o código que implementa o agente. Por padrão, o Jason assume que o arquivo de código fonte do agente possui o mesmo nome simbólico do agente. Por fim, o desenvolvedor pode especificar uma lista de opções (palavra options), que consistem em configurações para o interpretador Jason.

A Figura 9 ilustra um exemplo de declaração de um sistema multiagente utilizando a ferramenta de desenvolvimento do Jason. Brevemente, no exemplo é declarado um SMA chamado “heathrow”, cujo ambiente é implementado pela classe “HeathrowEnv”. Além disso, o sistema possui três tipos de agentes: “mds”, “cph” e “bd”.

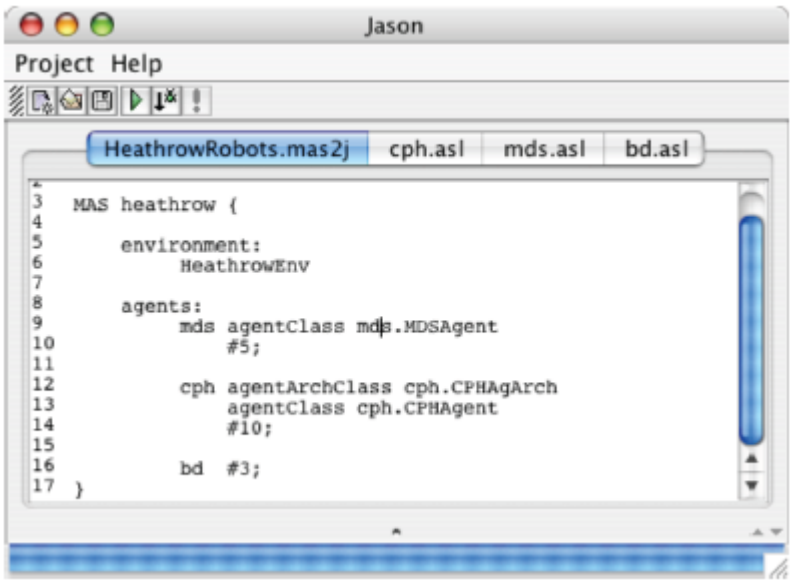


Figura 9 – Definindo um SMA em Jason [BDD+05].

### 2.2.4 3APL

O 3APL [DRM05] é uma linguagem de programação para o desenvolvimento de agentes cognitivos. Ela provê construtores para a implementação de crenças, objetivos e capacidades, tais como ações externas e ações de comunicação. No 3APL, um sistema multiagente é composto por um conjunto de agentes executando concorrentemente, que podem interagir através de comunicação ou indiretamente através do ambiente [DRM05].

Os agentes são implementados utilizando a linguagem 3APL, enquanto que o ambiente pode ser implementado utilizando a linguagem de programação Java. Esse ambiente é representado através de uma classe Java, e seus métodos correspondem às ações que os agentes podem realizar no ambiente (chamadas de ações externas).

No 3APL, existe uma separação entre as atitudes mentais de um agente (estruturas de dados) e o processo deliberativo (instruções de programação), que lidam com suas atitudes mentais. O 3APL permite a especificação de atitudes mentais tais como crenças, objetivos, planos, ações e regras de raciocínio.

A Figura 10 ilustra a arquitetura do 3APL em relação à sua plataforma de execução (A) e em relação à arquitetura interna do agente (B). A plataforma 3APL

consiste de um conjunto de agentes, de um facilitador de diretórios chamado *Agent Management System* (AMS), um sistema de transporte de mensagens que entrega mensagens trocadas entre os agentes, um ambiente compartilhado e uma interface que permite aos agentes executarem ações no ambiente compartilhado. A função do AMS é registrar os agentes que são carregados e executados na plataforma e responder a requisições de agentes sobre outros agentes que estão presentes na plataforma. Estas requisições podem ser, por exemplo, sobre os nomes dos agentes, suas funções e os serviços que eles provêem.

Cada agente individual 3APL consiste de uma base de crenças, uma base de objetivos, uma base de planos, uma base e ações que servem para a especificação das ações mentais internas, uma base de regras de planejamento de objetivos (que podem ser aplicadas para escolher um plano para atingir um objetivo) e uma base de regras para revisão de planos (que podem ser usadas para revisar, adotar e descartar planos).

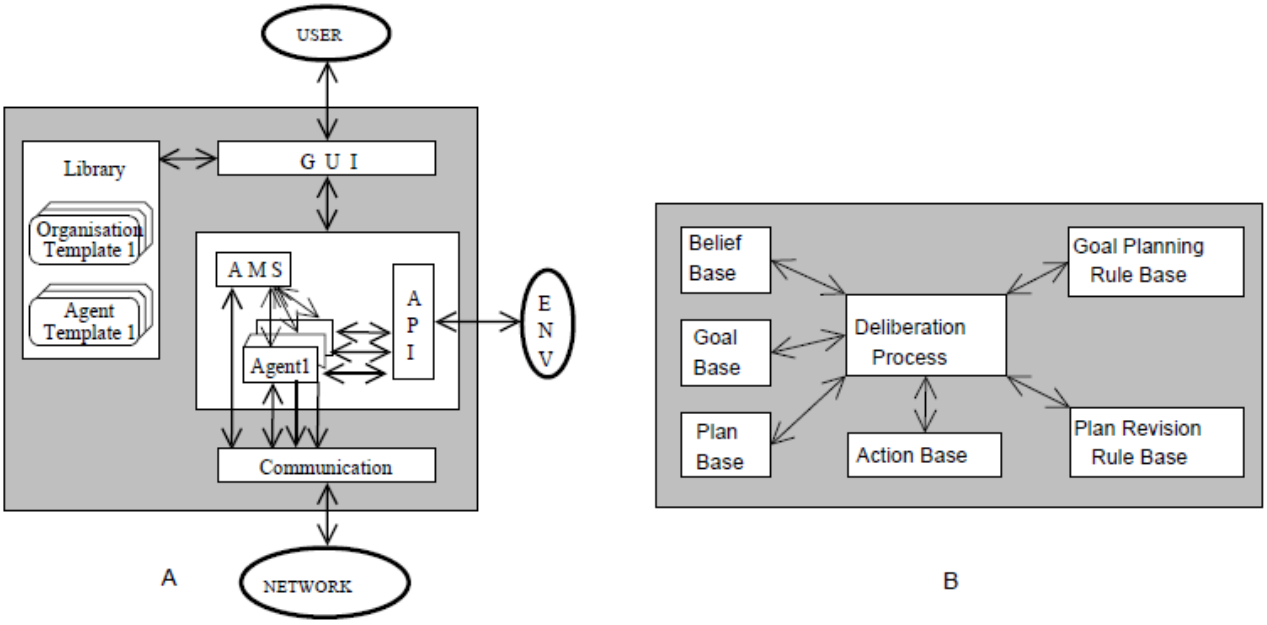


Figura 10 – (A) Arquitetura da plataforma 3APL e (B) a arquitetura de um agente 3APL [DRM05].

A especificação EBNF[1] da linguagem 3APL para a especificação de agentes é mostrada na Figura 11. As crenças (*beliefs*) em 3APL descrevem a situação em que o agente está. Elas compõem a base de crenças (*belief base*), que contêm as informações que o agente acredita sobre o mundo. Os objetivos (*goals*) do agente denotam situações que o agente deseja realizar, formando a base de objetivos do agente.

[1] EBNF – Versão estendida da BNF (*Backus-Naur Form*)

```
<Program>          ::=   "Program" <ident>
                         ( "Load" <ident> )?
                         "Capabilities :" ( <capabilities> )?
                         "BeliefBase :" ( <beliefs> )?
                         "GoalBase :" ( <goals> )?
                         "PlanBase :" ( <plans> )?
                         "PG-rules :" ( <p_rules> )?
                         "PR-rules :" ( <r_rules> )?
<capabilities>     ::=   <capability> ( "," <capability> )*
<capability>       ::=   "{" <query> "}" <Atom> "{" <literals> "}"
<beliefs>          ::=   ( <belief> )*
<belief>           ::=   <ground_atom> "." |
                         <atom> ": -" <literals>"."
<goals>            ::=   <goal> ( "," <goal> )*
<goal>             ::=   <ground_atom> ( "and" <ground_atom> )*
<plans>            ::=   <plan> ( "," <plan> )*
<plan>             ::=   <basicaction> | <composedplan>
<basicaction>      ::=   "ε" | <Atom> |
                         "Send(" <iv>, <iv>, <atom> ")" |
                         "Java(" <ident>, <atom>, <var> ")" |
                         <wff> "?" | <atom>
<composedplan>     ::=   "if" <wff> "then" <plan>
                         ( "else" <plan> )? |
                         "while" <query> "do" <plan> |
                         <plan> ";" <plan>
<p_rules>          ::=   <p_rule> ( "," <p_rule> )*
<p_rule>           ::=   <atom> "<-" <query> "|" <plan>
<p_rule>           ::=   "<-" <query> "|" <plan>
<r_rules>          ::=   <r_rule> ( "," <r_rule> )*
<r_rule>           ::=   <plan> "<-" <query> "|" <plan>
<literals>         ::=   <literal> ( "," <literal> )*
<literal>          ::=   <atom> | "not(" <atom> ")"
<wff>              ::=   <literal> | <wff> "and" <wff> |
                         <wff> "or" <wff>
<query>            ::=   <wff> | "true"
<iv>               ::=   <ident> | <var>
```

Figura 11 – Especificação EBNF da linguagem 3APL [DRM05].

A base de crenças é implementada por um programa escrito em Prolog que consiste de regras e fatos Prolog. A base inicial de crenças de um agente 3APL é precedida pela palavra reservada "`BeliefBase` :". A sintaxe Prolog está representada na produção `<beliefs>` da gramática. Assim como em Prolog, a especificação de crenças permite a utilização de negações no corpo de uma regra, como mostra o exemplo apresentado na linha 5 da Listagem 3.

Listagem 3 – Exemplo de declaração de crenças em 3APL.

```
1  BeliefBase :
2    on(a, f1).
3    on(b, f1).
4    on(c, a).
5    clear(Y) :- not(on(X,Y)).
```

A base de objetivos do agente 3APL é formada por um conjunto de objetivos, sendo cada um definido como conjunções em Prolog. A base inicial de objetivos é

precedida pela palavra "`GoalBase` :", como mostra o exemplo apresentado na Listagem 4. No exemplo são declarados dois objetivos. O primeiro objetivo é ter o bloco "a" no bloco "b" e o bloco "b" no bloco "c". O segundo objetivo é ter o bloco "d" no chão (*floor*).

Listagem 4 – Exemplo de declaração de um objetivo em 3APL.

```
1  GoalBase :
2    on(a,b) and on(b,c) , on(d,floor)
```

A fim de atingir seus objetivos, um agente 3APL adota planos. Um plano é construído através de ações básicas que podem ser compostas através de operadores. Em 3APL existe diversos tipos de ações tais como: ações mentais, ações de comunicação, ações externas e ações de teste. As ações mentais servem para atualizar a base de crenças do agente, caso seja executada com sucesso. Uma ação possui a forma de uma fórmula atômica que consiste de um nome de predicado e uma lista de termos. Uma ação mental também possui pré- e pós-condições, especificadas através de capacidades. Uma capacidade, por sua vez, é formada por: a própria ação mental, uma pré-condição que é uma expressão de consulta de crença (produção `<query>`), e uma pós-condição que consiste de uma lista de literais (produção `<literals>`).

Em 3APL, a especificação de uma capacidade é precedida pela palavra chave "`Capabilities` :". A Listagem 5 mostra um exemplo de capacidade que define o efeito da ação mental `Move`.

Listagem 5 – Exemplo de declaração de uma capacidade [DRM05].

```
1  Capabilities :
2  { on(X,Y) } Move(X,Y,Z) { not(on(X,Y)), on(X,Z) }
```

Uma ação de comunicação (predicado `Send`) pode ser utilizada para enviar uma mensagem para outro agente. Uma mensagem contém o nome do destinatário, o ato de fala ou um performativo da mensagem e o conteúdo. Ações externas são ações que alteram o ambiente externo no qual o agente executa. As ações externas são executadas pelos agentes assumindo que o ambiente externo é implementado através de uma classe Java. Em particular, todas as ações que podem ser executadas nesse ambiente são determinadas pelos métodos dessa classe Java. Os métodos nesse caso especificam a mudança de estado que essas ações causam ao ambiente, e o estado do ambiente é representado por variáveis de instância da classe. Uma ação de teste verifica se uma fórmula é derivável a partir da base de crenças.

As ações básicas, discutidas anteriormente, podem ser compostas através de operadores para a construção de planos. Em 3APL existe o operador sequencial (denotado pelo símbolo “`;`”), o operador de iteração (denotado pela construção “`while-do`”) e o operador condicional (denotado pela construção “`if-then-else`”). A base de planos de um agente 3APL consiste de um conjunto de planos. Na implementação do 3APL, a especificação da base inicial de planos de um agente é precedida pela palavra-chave “PlanBase :” e consiste de um conjunto de planos separados por vírgula.

Por fim, a plataforma 3APL provê uma IDE que permite a programação e execução de agentes 3APL [3APL11]. Ela provê um editor para a programação dos agentes, uma interface de comunicação entre os agentes e o AMS, que é responsável pelo registro dos agentes. Além disso, a plataforma 3APL pode ser executada em múltiplas máquinas conectadas através de uma rede.

### 2.2.5 FAML: um metamodelo genérico para o desenvolvimento de SMAs

O FAML [BHM+09] é um metamodelo que unifica conceitos e relacionamentos presentes em metamodelos e metodologias existentes para o desenvolvimento de sistemas multiagentes. O FAML foi construído com o objetivo de ser genérico, considerando conceitos que são comuns a qualquer SMA. Dessa forma, conceitos específicos de domínio, como atuadores (comuns na robótica), ou relacionados a um único agente foram omitidos. De acordo com os autores, a especialização de conceitos pode ser feita para incluir conceitos específicos de domínio ao FAML.

O FAML foi criado através de um processo iterativo, consistindo das seguintes fases: (1) determinação do conjunto inicial de conceitos fundamentais, presentes em qualquer SMA – como mencionado anteriormente, conceitos específicos de domínio foram omitidos; (2) listagem de definições (candidatas) para os conceitos determinados no passo um; (3) reconciliação entre as definições de cada conceito para manter a consistência interna do metamodelo; (4) separação dos conceitos em dois conjuntos: *design-time* e *runtime*; e, (5) identificação dos relacionamentos entre conceitos dos conjuntos *design* e *runtime*.

A saída para este processo resultou em quatro categorias de conceitos. Cada categoria refere-se a um escopo no metamodelo, que são: nível de sistema (*design-time* e *agent-external*), nível de definição do agente (*design-time* e *agent-internal*), nível de ambiente (*runtime*, *agent-externals*), e nível do agente (*runtime*, *agent-internal*). Nessa

seção, são apresentados brevemente os principais conceitos do FAML, necessários para o entendimento do metamodelo U-MAS, apresentado no capítulo 3. O conjunto completo de conceitos, e os diagramas UML que representam as quatro categorias de conceitos do FAML são apresentados no anexo A.

O conceito central em tempo de projeto é o **System**. Ele representa o produto final de um projeto de desenvolvimento orientado a agentes. Um **System** possui um **AgentDefinition**, que, por sua vez, é a especificação do estado inicial de um agente logo após sua criação. Um **System** também possui zero ou mais papéis. Um papel (**Role**) determina um padrão comportamental esperado para os agentes em um sistema. O papel também define quais facetas um agente interage, e quais agentes podem alterar ou sentir essas facetas. Uma faceta (**Facet**) é uma propriedade do ambiente com o qual os agentes podem interagir. Um sistema possui zero ou mais facetas, definidas através da classe **FacetDefinition**, que especifica a estrutura de uma faceta, incluindo seu nome, tipo de dado e modo de acesso.

Os agentes que existirão em tempo de execução são descritos através da classe **AgentDefinition**, que serve para inicializar todos os agentes relacionados ao sistema. Ela possui também uma função específica de papel, servindo para inicializar um agente quando ele passa desempenhar um papel durante sua execução. Um **AgentDefinition** consiste de um estado inicial e de um número de planos definidos através da classe **PlanSpecification**. Cada plano é composto por um conjunto de ações, definidas pela classe **ActionSpecification**.

Ações especificam como alterar uma faceta (**FacetActionSpecification**), ou especificam como enviar uma mensagem em uma determinada representação (**MessageActionSpecification**). Além da definição dos planos e ações, também é necessária a especificação dos recursos que um agente utiliza. Um recurso, então, é definido através da classe **ResourceSpecification**. Ele é algo que possui um nome, representação, pode ser adquirido, compartilhado ou produzido.

## 2.3 Computação Ubíqua

A computação ubíqua é uma visão de futuro onde sistemas computacionais estarão integrados em nosso dia-a-dia, provendo serviços e informações em qualquer lugar e a qualquer momento [Wei99a, AS99]. Os computadores ainda são vistos primariamente como máquinas que executam programas em um ambiente virtual que nós

acessamos para realizar uma tarefa e saímos ao terminá-la. A computação ubíqua presume uma visão diferente. Um dispositivo pode ser um portal para uma aplicação, e não somente um repositório de software que o usuário deve gerenciar. Uma aplicação é um meio pelo qual um usuário realiza uma tarefa, e não um software escrito para explorar as capacidades de um dispositivo. Um ambiente computacional é um espaço físico que provê informações, e não um ambiente virtual que existe para armazenar e executar programas [SM03].

A necessidade de informações percebidas acerca do ambiente diferencia a computação ubíqua da computação tradicional. Redes de sensores permitem a construção de sistemas ubíquos com informações tais como a localização de pessoas e dispositivos. Estes sistemas podem utilizar tais informações para interagirem mais naturalmente com os usuários, indo além de interações isoladas como ocorrem nas tradicionais estações de trabalho.

A computação ubíqua evolui a partir de dois grandes passos, que vêm se desenvolvendo desde a década de 70, que são a computação distribuída e a computação móvel. Alguns dos problemas técnicos da computação ubíqua correspondem a problemas já identificados e estudados nos passos anteriores à sua evolução. Existem ainda novos problemas introduzidos pela computação ubíqua que não possuem um mapeamento direto para problemas estudados anteriormente.

O campo de sistemas distribuídos surgiu da intersecção da computação pessoal e redes locais. Desde então, foram criados diversos *frameworks* conceituais e algoritmos para permitir que dois ou mais computadores operem juntos através de uma rede. A computação distribuída é o processo de divisão de uma tarefa computacional em um número de subtarefas menores para serem realizadas simultaneamente entre múltiplos computadores [Kur07]. Um sistema distribuído compreende dois ou mais dispositivos computacionais autônomos interconectados provendo a capacidade de compartilhar recursos lógicos e físicos.

Tipicamente, os sistemas distribuídos provêem transparência de acesso aos recursos através de interfaces abstratas, providas por um gerenciador de recursos. Dessa forma, cada recurso possui um identificador único, independente de sua localização. Um sistema distribuído também deve fornecer suporte a mobilidade de seus recursos. Tais características de sistemas distribuídos assemelham-se às características de sistemas ubíquos, onde a mobilidade de dispositivos, por exemplo, é um dos principais requisitos

provendo acesso aos recursos compartilhados que são embarcados no ambiente do sistema.

Segundo [Sat01], diversas áreas que são fundamentais para a computação ubíqua, que são:

- Comunicação remota (chamadas a procedimentos remotos);
- Tolerância a falhas;
- Alta disponibilidade;
- Acesso a informações remotas (sistemas de dados distribuídos);
- Segurança (encriptação e privacidade).

Além do campo de sistemas distribuídos, a computação móvel também exerce um papel importante para a computação ubíqua. O surgimento de computadores móveis e redes sem fio na década de 90 levaram os pesquisadores a confrontarem novos problemas ao construírem sistemas distribuídos para clientes móveis, que segundo [Sat01] são: variação imprevisível na qualidade da rede, baixa confiança e robustez dos elementos móveis, limitações nos recursos locais impostas por restrições de peso e tamanho e referentes ao consumo de energia.

Segundo [Sat01, SM03], grandes resultados da computação móvel podem ser destacados, tais como:

- Redes móveis: protocolos de redes *ad-hoc,* protocolos de redes móveis, técnicas de melhoria do protocolo TCP para dispositivos móveis;
- Acesso móvel a informações: operações desconectadas, acesso adaptativo a arquivos à banda de comunicação;
- Técnicas de economia de energia;
- Localização: detecção de localização, sistemas sensíveis a localização.

Os avanços tecnológicos necessários para a construção de um ambiente ubíquo (ou pervasivo) passa pelas seguintes áreas e tecnologias: dispositivos, redes, *middleware* e aplicações. Os relacionamentos entre essas áreas são ilustrados na Figura 12.

Um ambiente ubíquo caracteriza-se por conter dispositivos de diferentes tipos, tais como:

- Dispositivos de entrada e saída tradicionais, tais como teclados, caixas de som, monitores;
- Dispositivos sem fio, tais como celulares, *tablets e smartphones*;

- Sensores, tais como sensores de temperatura, luminosidade e posicionamento.

As tecnologias de **redes e comunicação** são necessárias para a interconexão e comunicação entre os diferentes dispositivos e aplicações em um ambiente, estabelecendo as interfaces e os protocolos de comunicação. A área de ***middleware*** representa uma camada ou interface entre os serviços providos por uma infraestrutura ubíqua e as aplicações para usuários finais, mantendo o usuário imerso no ambiente, abstraindo a heterogeneidade e tornando a computação ubíqua invisível ao usuário. Por fim, tem-se a camada de **aplicações**, que compreende todas as aplicações para o usuário final, e as aplicações que realizam o gerenciamento do ambiente ubíquo.

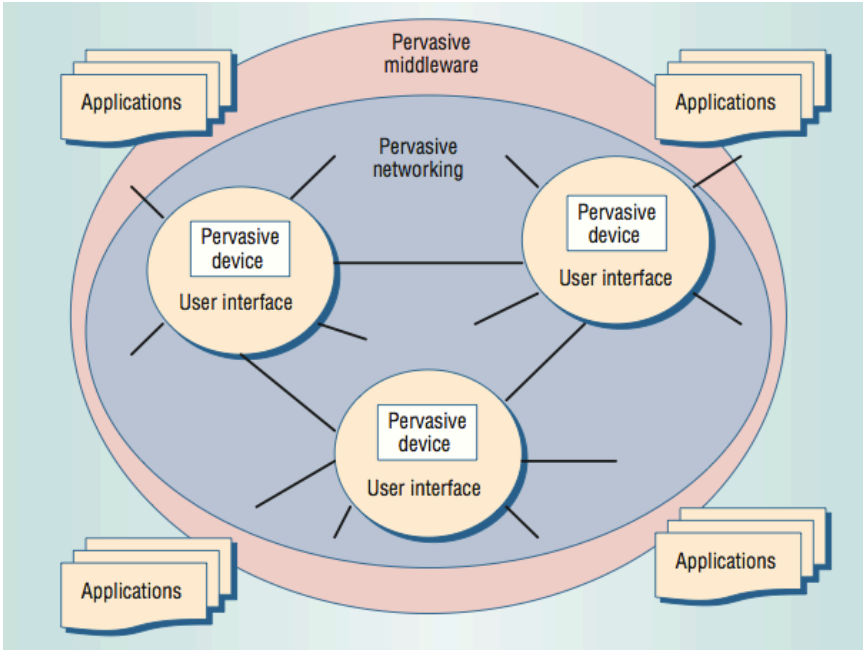


Figura 12 – Modelo de computação ubíqua [SM03].

A computação ubíqua também pode ser caracterizada por sistemas e dispositivos que são [Sym10]: (i) **embarcados**: diversos dispositivos integrados ao ambiente comunicando-se através de uma rede; (ii) **conscientes de contexto**: estes dispositivos podem reconhecer os usuários e o contexto do ambiente; (iii) **personalizados**: os sistemas podem ser customizados para as necessidades de cada usuário; (iv) **adaptativos**: os sistemas podem alterar seus comportamentos em resposta ao usuário; e (v) **antecipatórios**: os sistemas podem antecipar o desejo dos usuários sem a necessidade de mediação.

Os sistemas conscientes de contexto podem ajustar suas propriedades e comportamentos de acordo com informações sobre o estado atual do usuário, tais como

estado psicológico, padrões comportamentais, orientação e posição, localização geográfica, ou ainda propriedades do ambiente, tais como horário, usuários próximos, objetos e outros sistemas [Kur07]. Um sistema ubíquo é capaz de integrar essas diferentes fontes de informações contextuais e ajustar seu comportamento para situações ou oportunidades possivelmente inesperadas. Por exemplo, um sistema de navegação veicular que é consciente do nível de combustível e do tipo de combustível de preferência do usuário, pode alertar o motorista quando ele ou ela está próximo de um posto de combustíveis que vende esse tipo de combustível a um preço vantajoso.

A **invisibilidade** em um sistema ubíquo depende do grau de inteligência implementado pelo sistema a fim de permitir aos usuários focarem em suas tarefas ao invés de interagirem diretamente com o sistema [Kur07]. Esse tipo de sistema utiliza consciência de contexto na tentativa de prever as intenções do usuário baseando-se em um conjunto de informações contextuais. Os sistemas ubíquos precisam antecipar as ações dos usuários e se ajustar enquanto mantém balanceadas a proatividade e a invisibilidade do sistema. Manter um grau de invisibilidade é uma característica importante. A predição incorreta da intenção do usuário pode, por exemplo, revelar a presença do sistema, ou ainda distrair o usuário.

A habilidade do sistema de adaptar-se a fatores de mudança em seu ambiente é outra consideração importante para preservar a invisibilidade de um sistema. Tipicamente, um sistema ubíquo precisa adaptar-se a mudanças na disponibilidade de recursos sem revelar inconsistências no fornecimento destes recursos, como por exemplo, a largura de banda de uma rede, memória disponível ou capacidade de processamento. Por exemplo, devido à natureza das redes de comunicação sem fio, que podem sofrer interferência e *delay* devido a condições climáticas ou obstáculos físicos, é difícil prover (ou garantir) conectividade em ambientes ubíquos. Um sistema ubíquo que detecta problemas no fornecimento de recursos pode então ajustar seu comportamento para manter sua utilização, como por exemplo, diminuir a qualidade de um vídeo transmitido por *streaming* de acordo com a velocidade da conexão.

## 2.4 Linguagens de Programação

Os computadores são utilizados em diferentes áreas, e, por causa dessa grande diversidade, linguagens de programação com metas muito diferentes têm sido desenvolvidas para áreas como: aplicações científicas, aplicações comerciais, inteligência artificial, linguagens de *scripting,* linguagens de propósitos especiais, entre outras. Devido

a este trabalho propor uma linguagem de programação, o objetivo desta seção é examinar alguns dos recursos de uma linguagem de programação bem como seus critérios de avaliação.

### 2.4.1 Critérios de Avaliação de uma Linguagem de Programação

Avaliar uma linguagem de programação em função de suas características e construções é uma tarefa difícil de ser realizada. Porém, na literatura são encontrados alguns critérios que podem ser levados em consideração na análise de uma linguagem de programação [Seb09]. Algumas das características que influenciam os mais importantes critérios são mostradas na Figura 13, e serão discutidos ao longo da seção.

| | **Critérios** | | |
|---|---|---|---|
| **Características** | **Legibilidade** | **Capacidade de escrita** | **Confiabilidade** |
| Simplicidade/ ortogonalidade | * | * | * |
| Estruturas de controle | * | * | * |
| Tipos de dados | * | * | * |
| Projeto da sintaxe | * | * | * |
| Suporte para abstração | | * | * |
| Expressividade | | * | * |
| Verificação de tipos | | | * |
| Manipulação de exceções | | | * |

Figura 13 – Critérios de avaliação de uma linguagem de programação [Seb09].

A **legibilidade** é considerada um dos critérios mais importantes ao analisar-se uma linguagem de programação. Ela refere-se à facilidade com que os programas podem ser lidos e escritos. A legibilidade deve ser considerada no contexto do domínio do problema. Por exemplo, se um programa que descreve uma computação tiver sido escrito em uma linguagem não-projetada para esse uso, o programa pode tornar-se difícil de ser lido [Seb09]. Existem algumas características que contribuem para a legibilidade, como a **simplicidade global**, que refere-se ao conjunto de componentes básicos que o desenvolvedor deve aprender para poder programar utilizando a linguagem.

O conjunto de componentes que uma linguagem oferece remete a outra característica importante, a **ortogonalidade**. Ortogonalidade em uma linguagem de programação significa que um conjunto relativamente pequeno de construções primitivas pode ser combinado em um número relativamente pequeno de maneiras para construir as

estruturas de controle e de dados da linguagem. Além disso, toda combinação de primitivas é significativa. A ortogonalidade está diretamente relacionada à simplicidade: quanto mais ortogonal é uma linguagem, menos exceções às regras da linguagem existirão, aumentando a regularidade da linguagem e tornando-a mais fácil de ser aprendida e entendida. Dessa forma, a linguagem pode ser projetada visando facilitar sua legibilidade, utilizando instruções de controle de fluxo, tipos de dados e estruturas de dados. A presença de facilidades para a definição de tipos de dados e estruturas de dados podem ser um auxílio significativo para a legibilidade.

Uma linguagem também pode ser avaliada pela sua **capacidade de escrita**. Ela é uma medida de quão facilmente uma linguagem pode ser usada para criar programas em um domínio de problema escolhido. A maioria das características de uma linguagem que afetam a legibilidade também afeta a escrita. Isso remete ao fato de que escrever um programa exige uma releitura frequente da parte que já foi escrita pelo programador.

Se uma linguagem tiver um grande número de diferentes construções, alguns programadores podem não estar familiarizados com todas elas [Seb09]. Isso pode levar ao uso inadequado de alguns recursos, ou mesmo o desuso de outros que podem ser mais elegantes ou mais eficientes (ou ambos) do que aqueles comumente usados. Portanto, um número menor de primitivas e um conjunto consistente de regra para combiná-las (isto é, ortogonalidade) podem ser melhores do que ter um grande número de primitivas.

A **abstração** é fundamental no projeto de uma linguagem. O grau de abstração permitido por uma linguagem de programação e a naturalidade de sua expressão são muito importantes para sua capacidade de escrita. Normalmente as linguagens de programação suportam duas categorias de abstração: processo e dados. A abstração de um processo permite, por exemplo, a definição de subprogramas, enquanto que a de dados permite representar facilmente estruturas de dados e suas combinações para gerar novas estruturas.

Além da abstração, a **expressividade** é outra característica importante no que refere-se à capacidade de escrita de uma linguagem. Expressividade, em uma linguagem, pode referir-se a diversas características diferentes, como por exemplo, no caso de operadores que permitem que uma grande quantidade de computação seja realizada com um programa muito pequeno. Mais comumente, significa que uma linguagem tem formas relativamente convenientes, em vez de desajeitadas, de especificar computação.

A **confiabilidade** também é um fator que deve ser levado em consideração no projeto de uma linguagem. Existem fatores e mecanismos que influenciam a confiabilidade de uma linguagem. Dentre eles podemos citar a **verificação de tipos** e a **manipulação de exceções**, que visam testar e tratar erros que possam acontecer em um programa.

Por fim, pode-se avaliar o **custo** de uma linguagem, que pode ser avaliado sob diferentes aspectos. Primeiro, há o custo de treinamento de programadores para utilizar a linguagem. Em segundo lugar, há o custo para escrever programas nessa linguagem. Essa é uma função de capacidade de escrita, a qual depende do seu propósito com a aplicação em particular. Tanto o custo para treinar programadores como para escrever programas em uma linguagem podem ser significativamente reduzidos através do uso de um bom ambiente de programação. Em terceiro lugar, há o custo para compilar programas na linguagem, que depende da qualidade da implementação do compilador. Em quarto lugar, existe o custo para executar programas. Por fim, pode-se considerar o custo de manutenção dos programas, que inclui tanto correções quanto modificações para adicionar novas capacidades. O custo da manutenção depende de vários fatores, mas principalmente da capacidade de leitura da linguagem.

# 3 U-MAS: MODELANDO APLICAÇÕES MULTIAGENTES UBÍQUAS

Na literatura são encontradas diversas abordagens, como por exemplo [JAY+05, BC06, MPF04], que visam apoiar o desenvolvimento de aplicações ubíquas. Muitas dessas abordagens compartilham conceitos em comum, porém, em alguns casos, esses conceitos são contraditórios ou existem divergências em seus significados. Além disso, algumas dessas abordagens propõem soluções *ad-hoc* [RCR+02], dificultando a reutilização.

O processo de desenvolvimento do metamodelo U-MAS foi similar ao do FAML, exceto pela introdução de dois novos passos, que consistem no reuso e reconciliação de conceitos do FAML, a fim de integrar suas definições com as adaptações necessárias. Para determinar o conjunto inicial de conceitos, foram utilizadas aplicações ubíquas e definições recuperadas através de revisão de literatura em artigos de conferências, periódicos, bem como livros da área. O metamodelo final preservou os conceitos fundamentais do FAML, ao mesmo tempo em que modela conceitos de computação ubíqua no mesmo nível de abstração dos conceitos de agentes, separados em dois conjuntos: conceitos em tempo de projeto e em tempo de execução.

As próximas seções explicam os conceitos do U-MAS, onde são apresentados diagramas de classes UML que representam as quatro categorias de conceitos do metamodelo. As classes que representam conceitos de ubiquidade ou conceitos do FAML que foram adaptadas estão destacadas com fundo colorido.

## 3.1 Conceitos em Tempo de Projeto

Em tempo de projeto (Figura 14), o conceito **System** representa o produto final de um projeto de desenvolvimento orientado a agentes. Um sistema ubíquo possui um ou mais espaços onde os agentes atuam, definidos através da classe **SpaceDefinition**. Esses espaços agrupam elementos, tais como dispositivos, agentes e recursos no ambiente, e são definidos como a composição de diversas localizações (uma ou mais) [CCS07, JAY+05]. Assim sendo, uma localização (**Location**) pode ser definida como uma posição ou área de interesse [CCS07, BC06]. Através da localização, um sistema pode gerenciar recursos, agentes, e demais entidades presentes nos ambientes.

Uma localização possui uma posição (**Position**), e um sistema de posicionamento (**PositioningSystem**). Uma posição refere-se a pontos discretos em um espaço, que podem ser usados para definir os limites de uma localização. Um sistema de

posicionamento define os componentes básicos e a forma como uma posição deve ser interpretada. Uma posição consiste de um valor, enquanto um sistema de posicionamento consiste de um nome que o identifica.

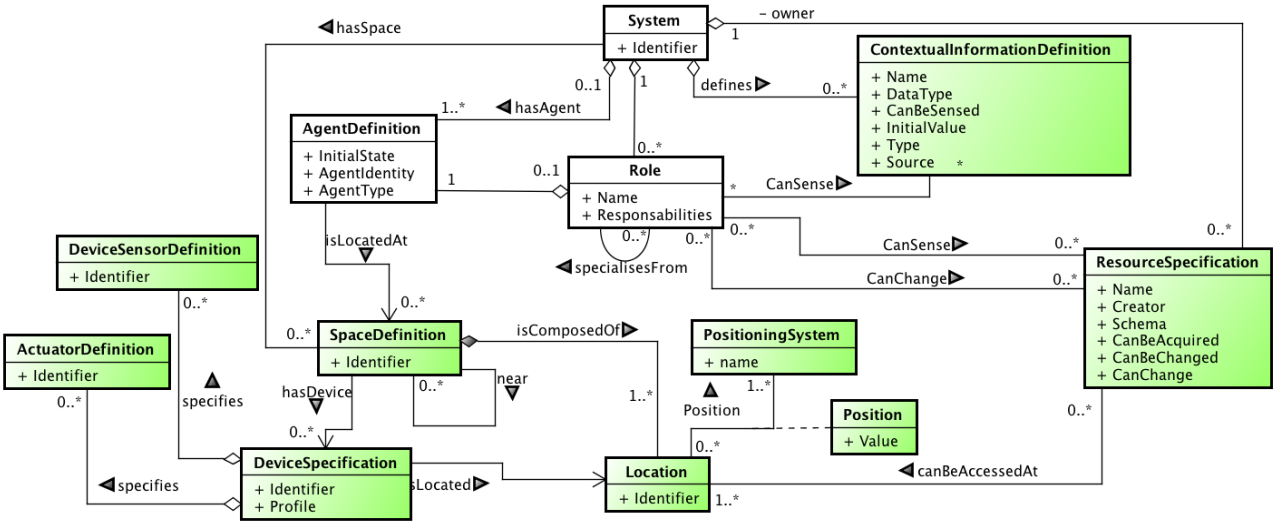


Figura 14 – Classes externas ao agente em tempo de projeto.

Um ambiente ubíquo também possui uma série de dispositivos, que podem ser de diversos tipos, tais como dispositivos embarcados no ambiente físico ou dispositivos móveis, tais como celulares e *tablets* [HMN+01]. Dispositivos são definidos através da classe **DeviceSpecification**. Cada dispositivo possui um identificador e um perfil que descreve suas características e capacidades, definidas em função de seus atributos [IM04]. Os atributos podem ser estáticos (invariantes no tempo) ou dinâmicos (variantes no tempo). Por exemplo, a resolução máxima da tela de um dado celular é um atributo estático, enquanto o nível atual de sua bateria é um atributo dinâmico.

Dispositivos também possuem uma localização dentro de um ambiente, permitindo assim, que possam ser percebidos. Além dos atributos, eles também podem possuir sensores e atuadores. Embora o FAML não considere atuadores e sensores em seu metamodelo por considerá-los específicos de domínio, no U-MAS eles são necessários pois representam conceitos importantes tratando-se de dispositivos físicos. Sensores, de acordo com [JAY+05, STB07], são componentes de software que provêem operações para a coleta de informações no ambiente. Eles não podem alterar o estado do ambiente, somente o observam. A fim de diferenciar sensores de dispositivos (abstrações do hardware) dos sensores de um agente, no U-MAS, um sensor de dispositivo é definido através da classe **DeviceSensorDefinition**. Dessa forma, um dispositivo especifica zero ou mais **DeviceSensorDefinition**. Assim como sensores, dispositivos podem especificar zero ou mais atuadores. Atuadores, portanto, são elementos capazes de influenciar o

estado do ambiente [JAY+05]. Por exemplo, um atuador de temperatura serve para aumentar ou diminuir a temperatura de uma sala, o qual será definido através da classe **ActuatorDefinition**.

Os agentes em um sistema ubíquo podem possuir uma localização dentro de um espaço. Para permitir o relacionamento entre um agente e um espaço, um novo relacionamento foi adicionado à classe **AgentDefinition**, relacionando-a a zero ou mais espaços. Um sistema também possui recursos, que podem ser vistos como objetos pertencentes ao sistema, tais como impressoras, arquivos e bancos de dados [Kur07]. O U-MAS utiliza a mesma definição do FAML para um recurso: algo que possui um nome, uma representação e pode ser adquirido, compartilhado ou produzido. A fim de adequar a definição de recurso, dois novos relacionamentos e novas propriedades foram adicionadas à classe **ResourceSpecification**. No U-MAS, um sistema é dono de zero ou mais recursos, e um papel (**Role**) define se um agente poderá perceber ou alterar um recurso. Um recurso, por sua vez, consiste de um nome, um criador, um esquema para a sua representação, e de duas propriedades indicando se ele pode ser adquirido ou alterado.

Outra característica importante em aplicações ubíquas é a noção de contexto. O contexto pode ser definido como qualquer informação utilizada para caracterizar a situação de uma entidade [SM03]. No FAML, as características de uma faceta são similares às de uma informação contextual. A fim de evitar interpretações errôneas, contudo, no U-MAS o conceito **FacetDefinition** foi substituído pelo conceito **ContextualInformationDefinition**. Os atributos **Name**, **DataType**, **InitialValue** e **CanBeSensed,** da classe **FacetDefinition,** foram mantidos na classe **Contextual-InformationDefinition**. Também foram preservados os relacionamentos **defines** e **canSense**, que agora relacionam respectivamente um sistema à suas informações contextuais e um papel, e as informações contextuais que ele pode sentir. Os atributos **canBeChanged** e **canChange** foram removidos de informação contextual, pois considera-se que ela pode variar independentemente dos agentes. Além disso, os agentes somente percebem o valor de uma informação contextual.

Na classe **ContextualInformationDefinition**, foram criados os atributos **Type** e **Source**. De acordo com [HIR02, CDL05], informações contextuais podem ser temporais, atemporais, percebidas ou inferidas. Assim sendo, o atributo **Type** foi criado para armazenar o tipo de informação contextual, de acordo com esse esquema de classificação. Já o atributo **Source** foi criado para indicar a fonte utilizada para a captura

da informação contextual. No entanto, a associação **canChange** foi removida, pois os agentes não possuem autonomia para modificar o estado de uma informação contextual.

Em relação aos aspectos internos do agente (Figura 15), foram feitas as seguintes alterações: a classe **SensorDefinition** foi criada para especificar a estrutura de um sensor em tempo de projeto, incluindo o seu padrão de captura de informações, pois o FAML não indica como capturar informações no ambiente. Também foi criado um relacionamento entre **SensorDefinition** e **AgentDefinition**. Este relacionamento indica que um agente pode possuir zero ou mais definições de sensores.

A classe **FacetActionSpecification** foi removida, devido à remoção do conceito **Facet**. Essa remoção justifica-se, pois agora um agente não pode alterar uma informação contextual. Com a remoção dessa classe, também foi removida a referência à classe **FacetDefinition** (que consistia originalmente no relacionamento **changes**). A classe **ResourceActionSpecification**, entretanto, foi criada para permitir que os agentes alterem, removam e publiquem novos recursos no ambiente. Essa classe herda as características da classe **ActionSpecification**, e é associada com a classe **ResourceSpecification** através do relacionamento **changes**, indicando que essa ação pode modificar um recurso.

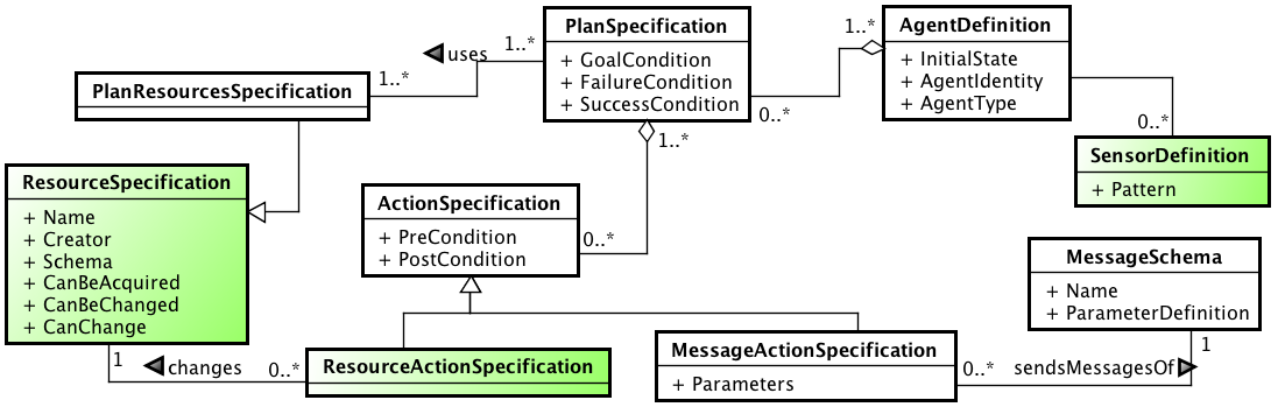


Figura 15 – Classes internas ao agente em tempo de projeto.

## 3.2 Conceitos em Tempo de Execução

Em tempo de execução (Figura 16), **Environment** representa o conceito principal. No FAML, ambientes são gerados a partir do elemento **System**, através do relacionamento **generates**. Embora esse relacionamento não seja explicado em [BHM+09], acredita-se que ele possa dificultar a construção de sistemas multiagentes abertos, composto por ambientes heterogêneos. O relacionamento **generates** permite a definição de um conjunto limitado de ambientes em tempo de execução. Dessa forma, o

relacionamento entre **System** e **Environment** foi alterado para **isComposedOf**, indicando que o sistema é composto por um ou mais ambientes, permitindo que novos ambientes possam ser adicionados em tempo de execução.

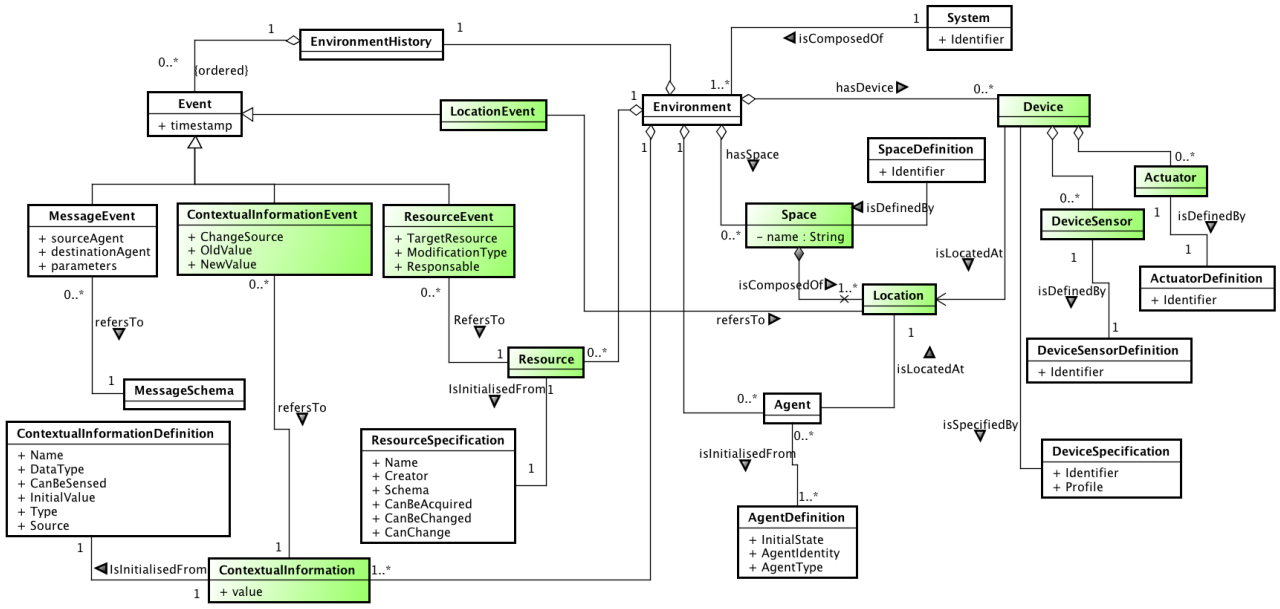


Figura 16 – Classes externas ao agente em tempo de execução.

O ambiente é essencial para um agente existir e atuar. Ele também restringe todas as informações contextuais que podem ser percebidas pelos agentes. Uma informação contextual (**ContextualInformation**), por sua vez, é qualquer informação ou propriedade do ambiente que pode ser percebida pelos agentes. O ambiente também hospeda um conjunto de recursos, definidos através da classe **Resource**. Em um ambiente ubíquo, a especificação de todos os espaços presentes no ambiente também é necessária.

Espaços servem para agrupar elementos no ambiente, e são representados pela classe **Space**. Eles são compostos por localizações, definidas através da classe **Location**. Uma localização serve para localizar dispositivos, agentes, e para delimitar o acesso aos recursos no ambiente. Dispositivos, por sua vez, são definidos através da classe **Device**. Cada dispositivo pode possuir seus próprios sensores e atuadores, definidos respectivamente através das classes **DeviceSensor** e **Actuator**. Adicionalmente, um ambiente também possui um histórico de todos os eventos ocorridos. Eventos (classe **Event**), nesse caso, podem ser de mensagens (gerados quando mensagens são trocadas entre os agentes), de informações contextuais (quando o estado de uma informação contextual é alterado), de recursos (quando um recurso é modificado, compartilhado ou removido), ou de localização (quando dispositivos ou agentes mudam de local).

Os aspectos internos ao agente são apresentados na Figura 17. A classe **ResourceAction** foi incluída baseada na classe **ResourceActionSpecification**, e definida em tempo de projeto para permitir que os agentes alterem, removam e publiquem recursos em tempo de execução. Um relacionamento de herança, portanto, foi criado, visto que **ResourceAction** representa um tipo de ação executada pelo agente. O relacionamento entre **ResourceAction** e **Resource** foi criado para indicar que a ação é capaz de modificar um recurso.

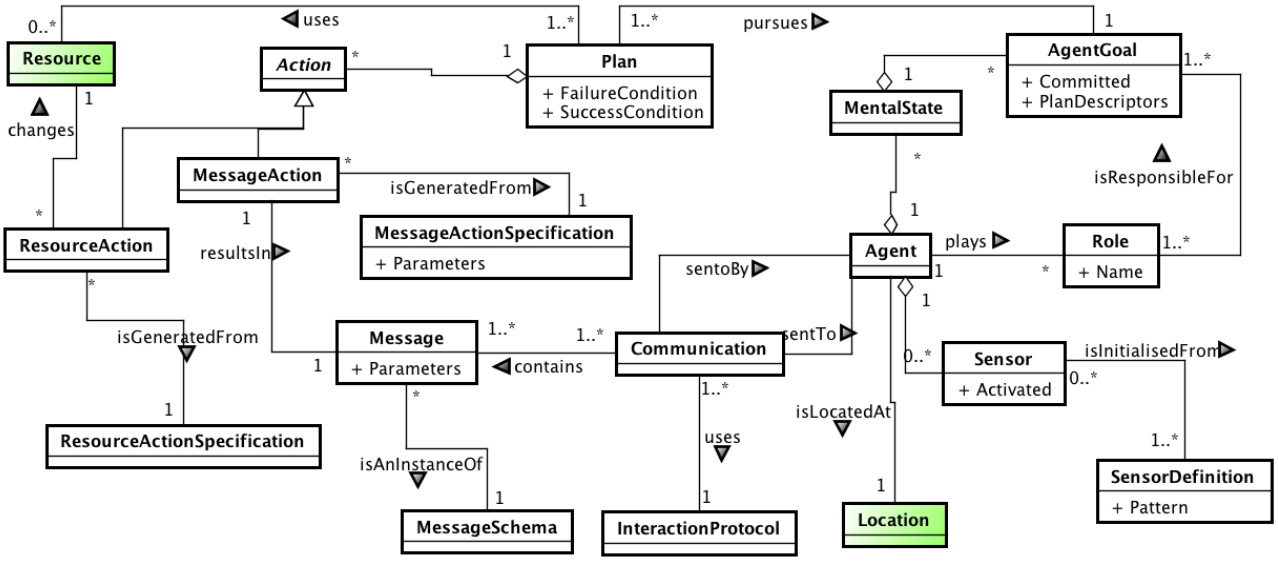


Figura 17 – Classes internas ao agente em tempo de execução.

# 4 AGENTS ANYWHERE (AA) – PROGRAMANDO SISTEMAS MULTIAGENTES UBÍQUOS

Este capítulo apresenta a linguagem AA (leia-se "*double* A"), uma linguagem de programação em que os conceitos de computação ubíqua são definidos no mesmo nível de abstração dos conceitos de sistemas multiagentes. A sintaxe da linguagem AA é fortemente baseada em Java, uma das linguagens de programação mais utilizadas, afim de facilitar sua adoção. Além de ser baseada em uma linguagem amplamente difundida na indústria e na academia, a linguagem AA foi projetada para ser independente de metodologia e plataforma de desenvolvimento de SMAs.

Para construir a linguagem, foram utilizados elementos pertencentes aos modelos de *runtime* do U-MAS e FAML. Foram selecionados conceitos centrais para a definição formal da linguagem, partindo da noção de agente e de seus componentes internos. Essa versão da linguagem constitui o *kernel* mínimo da linguagem. Nela são tratados conceitos centrais de agentes e conceitos fundamentais de computação ubíqua, como a definição de espaços (regiões simbólicas), localização e dispositivos.

## 4.1 Especificações e Aspectos Sintáticos da Linguagem AA

A linguagem AA inclui tipos primitivos de dados (os mesmos suportados pela linguagem Java), tipos específicos de dados (relacionados ao sistema multiagente, recursos, estruturas internas do agente e conceitos de ubiquidade), operadores que podem ser aplicados a estes tipos de dados e estruturas para controle de fluxo.

Além da definição dos tipos primitivos, a linguagem AA possui operadores para a manipulação de dados. A maioria dos operadores só podem ser utilizados sobre tipos primitivos. Atualmente somente o operador de atribuição pode ser utilizado para manipular os tipos específicos de dados. Nesse caso, a atribuição significa a cópia da referência do elemento para uma variável de mesmo tipo. A declaração de variáveis na linguagem AA é similar à linguagem Java, isto é, toda variável possui um tipo, um identificador bem definido e pode ser inicializada no momento da sua declaração.

Assim como na linguagem Java, todo elemento possui visibilidade condicionada ao escopo onde são declaradas. O *escopo* delimita o conjunto de sentenças onde ele é visível [Seb09]. Se um elemento é visível, significa que ele pode ser referenciado na sentença em consideração.

Para permitir o acesso dentre os diferentes escopos, a linguagem AA dispõe de três operadores. Elementos declarados no sistema multiagente devem ser acessados utilizando o operador **mas**. Elementos declarados no nível do agente devem ser acessados utilizando o operador **agent**, e elementos declarados internamente a um plano são acessados através do operador **plan**.

Além dos tipos de dados, a linguagem deve permitir o controle de fluxo para a execução de operações. Na linguagem AA, existem duas categorias de sentenças para controle de fluxo, que são: condicionais e repetição. Sentenças condicionais são o “if–then” e o “if-then-else”; sentenças de repetição são o “for” e o “while”. Na linguagem AA, as expressões utilizadas como testes para estruturas condicionais só podem ser de tipos primitivos, utilizando variáveis pertencentes ao escopo do SMA, agente ou plano de ação.

O fragmento de gramática na Figura 18 mostra a principal parte da especificação sintática da linguagem AA.

| | |
|---|---|
| <System> ::= | “MAS” <Ident> “{”<br>( <Agent> \|<br><Resource> \|<br><PS> \|<br><Space> \|<br><DeviceSpecification> )*<br>“}” |
| <Agent> ::= | “agent” <Ident> “{”<br>( <Plan> \| <Goal> \| <Resource> )*<br>[ “init” “{” (<Statement>)* “}” ]<br>[ “reacts” “{”<br>[ “on” “locationchanged” <Ident> “{” (<Statement>)* “}” ]<br>[ “on” “devicechanged” <Ident> “{” (<Statement>)* “}” ]<br>[ “on” “messagereceived” <Ident> “{” (<Statement>)* “}” ]<br>“}” ]<br>“}” |
| <Plan> ::= | “plan” <Ident> “for” <Ident> [ “if” “(” <Expression> “)” ]<br>“{”<br>( <Action> \| <Resource> )*<br>“}” |
| <Action> ::= | [ “first” ] “action” <Ident><br>“{”<br>[ “next” <Ident> ( “,” <Ident> )* “;” ]<br>“perform” “{” (<Statement>)* “}”<br>“}” |
| <Goal> ::= | “goal” <Ident> “;” |
| <Resource> ::= | [ (“volatile” \| “persistent”) ] “resource” <Ident> “;” |
| <PS> ::= | “positioningsystem” <ident> “{” ( <VarDecl > )* “}” |
| <Space> ::= | “space” <Ident> “{” ( <Location> \| <Device> )* “}” |
| <Location> ::= | “location” <Ident> “of” <Ident> “{” ( < VarDecl > )* “}” |
| <Device> ::= | “device” “of” <Ident> “;” |
| <DeviceSpecification> ::= | “device” <Ident> “{” ( [ “readonly” ] <VarDecl> )* “}” |

Figura 18 – Especificação em EBNF dos principais construtores da linguagem AA.

O elemento base da gramática é o construtor **System**. Ele é responsável pela produção de um Sistema Multiagente composto por: agentes, recursos, espaços, sistemas de posicionamento e especificações de dispositivos. Ele inicia com a palavra

reservada **MAS** seguido de um identificador. Um **identificador** é composto por um ou mais caracteres alfanuméricos, devendo ser iniciado obrigatoriamente por uma letra. Além disso, um identificador não pode conter espaços em branco.

Agentes, espaços, dispositivos e sistemas de posicionamento só podem ser declarados no escopo do sistema multiagente, enquanto que recursos podem ser declarados em diferentes níveis, sendo este detalhado adiante na Listagem 9.

### 4.1.1 Especificando Agentes e Recursos

Um agente é definido utilizando a palavra reservada **agent,** seguida de um identificador, que deve ser único dentro do escopo do sistema multiagente. No corpo do agente podem ser declarados zero ou mais: planos, objetivos ou recursos. Cada agente do sistema deve possuir seu próprio código, pois a linguagem trata os agentes como elementos individuais não instanciáveis. Os recursos declarados no escopo do agente serão visíveis apenas pelas próprias ações do agente. Note que o modificador **mas** pode ser utilizado caso o agente necessite acessar recursos no escopo do SMA.

Um agente possui também duas seções opcionais: *init* e *reacts*. A seção **init** pode ser usada para a programação do estado inicial do agente, após ser instanciado pela plataforma. No bloco *init* o desenvolvedor pode declarar variáveis locais (seguindo a mesma sintaxe da linguagem Java) utilizando os tipos primitivos de dados, *String*, vetores uni ou multidimensionais, iniciar explicitamente a execução dos objetivos do agente, ou utilizar estruturas condicionais ou de repetição.

A seção **reacts** é utilizada pelo agente para receber eventos do ambiente tais como: mensagens, alterações de localização ou alterações no estado de um dispositivo. Todos os eventos possuem uma propriedade comum, chamada *timestamp* (do tipo *long*), que representa a data e hora em que o evento ocorreu.

No exemplo de código na Listagem 6 é ilustrado um exemplo de declaração de um agente e suas estrutura interna. Um evento de localização é gerado sempre que o ambiente de execução detectar que o dispositivo que executa o SMA alterou sua localização, em relação aos espaços declarados pelo programador.

Na listagem acima, a variável “l” representa o evento de localização. Este evento contêm um tipo (*entered* ou *exited*) e o identificador do espaço, representados respectivamente pelas propriedades *type* e *identifier,* ambas do tipo *String*. A variável “m” representa a mensagem recebida pelo agente, possuindo as propriedades *from*, *to* e

*subject* do tipo *String* e *content* do tipo *Object*. Como o bloco *reacts* é opcional, uma cópia dessa mensagem também é armazenada na fila de mensagens do agente. Por fim, a variável "d" representa a alteração no estado de um dispositivo. Ela contém a identificação do dispositivo e os valores das propriedades que sofreram alteração armazenados em uma tabela em memória. As propriedades são armazenadas no formato chave-valor, e são acessíveis via método **get**. A chave a ser usada deve consistir de uma *string* que representa o nome de alguma das propriedades que o desenvolvedor utilizou ao definir o dispositivo (apresentado na Listagem 12). Dentro de cada reação o desenvolvedor pode declarar os mesmos tipos de estruturas permitidas no bloco *init*.

Listagem 6 – Exemplo de declaração um agente.

```
 1  agent <Algum Nome> {
 2     init {
 3        <Código de inicialização>
 4     }
 5
 6     reacts {
 7        on locationchanged l { <algumCódigo> }
 8
 9        on messagereceived m { <algumCódigo> }
10
11        on devicechanged d { <algumCódigo> }
12     }
13  }
```

Os objetivos de um agente são definidos através da palavra reservada **goal**. Todo objetivo possui um identificador, que deve ser único dentro do escopo do agente. Para atingir um objetivo, o agente precisa ter um plano associado a ele. O estado interno do objetivo é dado pela sua propriedade somente leitura ***state***, do tipo *string*. São considerados dois estados possíveis para um objetivo: novo, atribuído no momento de sua criação, e completo, atribuído quando o plano que o persegue finaliza corretamente sua execução. Além disso, um objetivo pode ser executado mais de uma vez, através do operador `achieve`. A linguagem AA não implementa um mecanismo elaborado para o controle dos objetivos. Note que implementações complexas para controle de estados de objetivos, como por exemplo [BPM+04, BP09], afetam principalmente o ambiente de execução, podendo não impactar na sintaxe da linguagem.

Um plano é definido através da palavra reservada **plan**, seguida de um identificador, que deve ser único no escopo do agente em que este é declarado, e um corpo. O corpo de um plano é composto de zero ou mais ações ou recursos. O plano precisa ser associado ao tipo de objetivo que ele persegue através do operador **for**. Este plano também pode possuir uma precondição para sua execução, definida através da

expressão **if**, como ilustrado na Listagem 7. Embora a linguagem permita a escrita de um programa contendo mais de um plano para atingir um mesmo objetivo, o ambiente de execução considera apenas o último destes planos, conforme ordem de escrita do código-fonte. Tal condição não foi implementada como uma regra semântica da linguagem.

A declaração de uma ação inicia com a palavra reservada **action**, seguida de um identificador único no escopo do plano onde ela é declarada. O código da ação (conjunto de sentenças) deve ser explicitamente declarado na seção **perform**. Na seção *perform* o desenvolvedor pode definir variáveis locais, estruturas condicionais e de repetição, chamar funções além de poder acessar estruturas previamente declaradas nos diferentes escopos do sistema através dos operadores de acesso *mas*, *plan* e *agent*.

Listagem 7 – Exemplo de declaração de um plano.

```
1  plan <nomeDoPlano> for <nomeDoObjetivo>
2     if ( <algumaPrecondicao> )
3  {
4     <Código>
5  }
```

Para permitir concorrência de execução entre os agentes, as ações são instanciadas como *threads* na plataforma. O conjunto de sentenças da seção *perform* são mapeados diretamente para o método executor da *thread*, que na Java é o método *run* da classe *java.lang.Thread*.

Uma vez que o plano inicia sua execução, a *primeira* ação (definida através do operador **first**) é invocada. Ao término de sua execução, uma ação também pode possuir uma próxima ação, determinada através do operador **next** seguido do identificador da próxima ação, como ilustrado na Listagem 8. Caso haja mais de uma próxima ação, seus identificadores devem ser separados por vírgula. Por serem implementadas como *threads*, estas ações serão executadas concorrentemente. Note que o operador *next* permite apenas o encadeamento entre as ações. A linguagem AA não dispõe de operadores para a especificação de sincronismo entre as ações.

Os recursos de um sistema podem ser declarados em três diferentes escopos: sistema multiagente, internamente a um agente ou internamente a um plano. A definição formal de um recurso torna-se difícil pois a literatura provê definições muito amplas ou abstratas para o conceito. No FAML, por exemplo, um recurso é definido como “algo que possui um nome, uma representação e pode ser adquirido, compartilhado ou produzido”.

A fim de adequar a gramática da linguagem à definição de recurso do FAML, a propriedade “nome” de um recurso é representada por um identificador único; a propriedade “adquirido” é considerada como a possibilidade de um recurso (dependendo do escopo onde é declarado) poder ser acessado por agentes; a propriedade “compartilhado” é considerada como a possibilidade de disponibilização de um recurso para diversos agentes no sistema multiagente; e, por fim, a propriedade “produzido” é considerada como a capacidade de um agente de instanciar um ou mais recursos. Para definir um recurso é utilizado o operador **resource**, seguido por um identificador que deve ser único no escopo no qual o recurso está sendo declarado.

Listagem 8 – Exemplo de encadeamento de ações.

```
 1  first action A {
 2     next B;
 3
 4     perform {
 5        print("First action (A).");
 6     }
 7  }
 8
 9  action B {
10     perform {
11        print("Second action (B).");
12     }
13  }
```

Na linguagem AA, existem dois tipos de recursos: voláteis e persistentes. Os **voláteis** são recursos cujos valores de suas propriedades são mantidos somente durante a execução da aplicação; isto significa que o estado não é mantido após a aplicação encerrar. Em contraste, os recursos persistentes terão suas propriedades persistidas na aplicação, isto é, seu estado (valor atual de suas propriedades) é salvo, permitindo que permaneçam por múltiplas execuções da aplicação.

Por padrão um recurso é volátil. Para indicar se o recurso em consideração é persistente, o operador **persistent** deve preceder o operador **resource** no momento da declaração de um recurso. Um recurso pode ser explicitamente declarado como volátil através do operador **volatile**. A Listagem 9 mostra um exemplo de declaração dos dois tipos de recurso. Na linha 2 é declarado um recurso volátil e na linha 3 é declarado um recurso persistente.

Listagem 9 – Declarando recursos.

```
1  MAS Sample {
2     volatile resource MemoryResource;
3     persistent resource DatabaseResource;
4  }
```

Dado que um recurso pode ser visto como um conjunto de propriedades, a linguagem deve prover uma forma padrão de acessá-las. Dessa forma, operações predefinidas são providas para ler e escrever as propriedades de um recurso, bem como para verificar se uma dada propriedade existe, que são **get**, **set** e **contains**. Uma propriedade consiste em um par chave-valor, onde a chave é uma *string* e o valor é qualquer tipo de dado aceito pela linguagem. A Listagem 10 exemplifica a utilização das propriedades de um recurso.

Listagem 10 – Acesso às propriedades de um recurso.

```
MAS exemplo {
   resource res1;

   agent Ag1 {
      plan ResourceTest  {
         first action setPropertyValue {
            perform {
               mas.res1.set("username", "Mauricio");
            }
         }
      }
   }

   agent Ag2 {
      plan ResourceTest  {
         first action showPropertyValue {
            perform {
               if(mas.res1.contains("username")){
                  string username =
                     (string) mas.res1.get("username");
                  print(username);
               }
            }
         }
      }
   }
}
```

No exemplo acima, são criados dois agentes, cada um com sua respectiva ação, acessando um recurso volátil, declarado no escopo do sistema multiagente, isto é, um recurso compartilhado. Na ação do primeiro agente é definida e inicializada uma propriedade chamada "username", atribuindo-se a ela o valor "Mauricio". Na ação do segundo agente, o valor a propriedade "username" é lido e atribuído a uma variável local, que posteriormente é passada por parâmetro para o procedimento **print**, a fim de exibir a informação no console da aplicação. Como o conteúdo de uma propriedade pode assumir qualquer tipo de dado, a linguagem implementa a operação de *type casting*, possibilitando ao desenvolvedor realizar conversões explícitas de dados.

### 4.1.2 Especificando Localizações e Dispositivos

Um espaço é declarado utilizando o operador **space** seguido de um identificador e de um conjunto de localizações ou dispositivos. Uma localização possui um identificador e uma estrutura, definida em termos de um sistema de posicionamento, utilizando o operador **of**, como ilustrado na Listagem 11.

Um sistema de posicionamento define os componentes básicos (estrutura) que compõem uma posição, expressos como variáveis. Esta definição é necessária para permitir ao desenvolvedor especificar os valores estáticos ao definir uma localização, e para permitir ao analisador semântico validar corretamente a estrutura da localização, em relação ao sistema de posicionamento que a define, visando reduzir erros de execução. O código apresentado na Listagem 11 ilustra a declaração de um sistema de posicionamento, um espaço e sua localização.

Listagem 11 – Sistema de posicionamento, espaço e localização.

```
 1  positioningsystem GPS {
 2        double latitude,  longitude;
 3        int radius;
 4  }
 5
 6  space FACIN {
 7     location loc of GPS {
 8        latitude = -30.056688;
 9        longitude = -51.172796;
10        radius = 300;
11     }
12  }
```

Note que a especificação do sistema de posicionamento é independente de plataforma. No exemplo, o sistema de posicionamento GPS define a estrutura básica de uma coordenada, que consiste em latitude, longitude e raio. Sua implementação em *runtime* é um ponto de flexibilidade provido pelo ambiente de execução da linguagem AA (detalhado na Seção 4.3), indicando que, para cada sistema de posicionamento utilizado em um programa, a plataforma deve possuir uma implementação padrão responsável pelo gerenciamento do sistema de posicionamento que consiste na captura dos valores de suas propriedades em tempo de execução.

O sistema de posicionamento também é responsável por calcular a interseção com localizações de um mesmo tipo. A linguagem AA não implementa a noção de localizações próximas. Para gerenciar esses elementos, a plataforma possui um serviço chamado **LocationManager**. Sua principal função é determinar a localização corrente

com base nos sistemas de posicionamento e localizações declaradas pelo desenvolvedor. Ao detectar alteração na localização, em função dos sistemas de posicionamento utilizados, a plataforma gera eventos no ambiente que podem ser *percebido* pelos agentes (reação `locationchanged` descrita nos detalhes do agente), permitindo a eles tomarem alguma decisão. Além de gerar eventos, alternar entre espaços influencia nos recursos e dispositivos que serão disponibilizados dinamicamente para acesso pelos agentes. O armazenamento dos eventos de localização fica a critério do desenvolvedor. Além disso, a implementação padrão do ambiente de execução não mantém histórico dos eventos gerados.

Como agentes e recursos podem ser localizados em um ambiente, **location** é uma propriedade predefinida, permitindo aos programadores acessá-la. Nesse contexto, **location** é uma propriedade somente-leitura que retorna o conjunto de posições (armazenadas em um vetor) que determinam a localização corrente. Seus valores mudam de acordo com o movimento do dispositivo executando o sistema multiagente, sendo gerenciados pela plataforma. A propriedade **location** também serve para determinar a *visibilidade* dos recursos e/ou agentes no ambiente. Em tempo de execução, esta propriedade é usada para restringir o acesso a estes elementos utilizando as fronteiras do *espaço* como referência. Por exemplo, um dispositivo só poderá ser acessado por agentes em um determinado espaço.

A especificação de um dispositivo é similar à especificação de um sistema de posicionamento. O desenvolvedor deve determinar a estrutura que compõe o dispositivo em termos de suas propriedades, expressas como variáveis. Essa especificação estática é necessária para melhorar a análise semântica e verificação de erros de programação e também para tornar clara a forma como ele deve ser utilizado.

Um dispositivo é especificado utilizando a palavra reservada **device** (Listagem 12), seguido por um identificador, que deve ser único no escopo do SMA. No corpo do dispositivo devem ser declaradas as propriedades que o compõe. A implementação lógica do dispositivo também é um ponto de flexibilidade abstraído via plataforma. Para cada tipo de dispositivo, a plataforma deve possuir uma implementação padrão capaz de coletar o valor de suas propriedades em tempo de execução, disponibilizando-as para acesso. Note que na gramática da linguagem AA, um dispositivo não especifica métodos, isto é, comportamentos dinâmicos. Caso necessário, na implementação padrão do dispositivo o programador pode monitorar o estado das propriedades e invocar manualmente métodos como resposta de tais alterações. Por exemplo, ao implementar um dispositivo do tipo

*notebook*, ao detectar que uma propriedade booleana *powerSource* foi alterada para desligado, o programador poderia comandar o dispositivo físico a entrar em modo economia de energia.

De forma a facilitar a utilização da abstração de um dispositivo, as propriedades declaradas em sua definição podem ser lidas e escritas (atribuindo-se um valor a ela). Por padrão, uma propriedade é de leitura e escrita. O desenvolvedor pode modificar o comportamento padrão informando se a propriedade será somente leitura, através do operador **readonly**. O código apresentado na Listagem 12 ilustra a declaração de dois dispositivos variando seus tipos de propriedades.

Listagem 12 – Exemplo de especificação da estrutura de um dispositivo.

```
1  device MacBook {
2     readonly int batteryLevel;
3     boolean power;
4  }
5
6  device Light {
7     int intensity;
8     boolean power;
9  }
```

Após especificar os tipos de dispositivos possíveis no sistema e suas estruturas, o desenvolvedor pode declarar instâncias destes dispositivos no escopo de um determinado espaço. Para declarar um dispositivo também é usado o operador **device**, seguido de um identificador e do tipo de dispositivo que essa instância deriva, através do operador **of**, como ilustra o exemplo na Listagem 13. Nas linhas 1 e 6 são especificados dois tipos de dispositivos e suas respectivas estruturas. Na linha 11 é declarado um espaço, e nas linhas 12, 13 e 14 são declaradas três instâncias de dispositivos (`lampA`, `lampB` e `ac`).

Listagem 13 – Declarando e utilizando dispositivos.

```
 1  device Light {
 2     int intensity;
 3     boolean power;
 4  }
 5
 6  device AirConditioning {
 7     boolean power;
 8     int temperature;
 9  }
10
11  space LivingRoom {
12     device lampA of Light;
13     device lampB of Light;
14     device ac of AirConditioning;
15  }
```

O acesso aos valores das propriedades dos dispositivos em tempo de execução só será válido caso a plataforma detecte que o sistema está situado no espaço em questão. Caso contrário, se o agente tentar ler uma propriedade será retornado o valor *null*.

Ao entrar em um espaço, os agentes em execução passam a perceber alterações no estado interno dos dispositivos. Essa funcionalidade é de responsabilidade de um serviço da plataforma chamado *DeviceManager*, que consulta periodicamente o estado dos dispositivos e notifica os agentes caso uma alteração seja detectada. Nesse caso, a plataforma gera um evento no ambiente que pode ser *percebido* (reação `devicechanged` descrita nos aspectos internos do agente, na subseção 4.1.1) pelos agentes, permitindo a eles tomarem alguma decisão. Assim como os eventos de localização, os eventos de dispositivos não são mantidos em histórico pela plataforma.

## 4.2 O Ambiente de Execução

O ambiente de execução da linguagem AA foi desenvolvido na linguagem Java Standard (versão 5 para *desktop*), com classes independentes para facilitar futuras extensões e portabilidade para diferentes plataformas.

Após o processo de compilação, é feita a geração de código intermediário, com classes pertencentes ao *kernel* da plataforma. Essas classes de nível intermediário não caracterizam uma aplicação, pois não possuem todos os recursos para permitir sua execução. Para permitir a execução do SMA em máquinas *desktop,* algumas dessas classes são traduzidas por *wrappers* para classes do *framework* SemantiCore [BEC07]. Utilizar mecanismos de tradução e pontos de flexibilidade visam permitir que o ambiente de execução possa utilizar outros *frameworks*, como por exemplo o JADE, ou até mesmo adaptados para outra plataforma, como o Android, com foco em dispositivos móveis, e que também utiliza o Java como linguagem nativa.

O SemantiCore foi escolhido por se tratar do resultado de um projeto de pesquisa no qual o autor deste estudo participou, e pela possibilidade de acesso ao código-fonte, permitindo assim a reutilização de código. Além disso, o SemantiCore possui implementados o controle de distribuição de computação, que permite a execução e sincronização de diferentes instâncias da plataforma em uma rede, o controle de concorrência entre agentes, mecanismos de comunicação e elementos que representam

a estrutura interna de uma agente, como por exemplo sensores, atuadores, objetivos, planos, entre outros.

No SemantiCore a classe *SemanticAgent* é abstrata. Ao traduzir código AA para as estruturas da máquina virtual, para cada agente declarado é criada uma classe concreta que estende a classe *SemanticAgent*. Dentro dessa classe são declaradas todas as estruturas relacionadas ao agente. A classe *Action* também é abstrata. Dessa forma, o gerador de código intermediário também gera classes concretas para as ações declaradas no código fonte AA. Tanto os identificadores de agentes quanto os de ações são utilizados como identificadores dos objetos no processo de geração de código.

Uma vez realizada a etapa de geração de código o programa já está pronto para ser executado em um ambiente Java. O método *main,* necessário para iniciar uma aplicação em Java, é inserido na classe gerada a partir da construção da linguagem que representa o sistema multiagente.

O diagrama apresentado na Figura 19 mostra a organização da arquitetura da plataforma. O objetivo do diagrama é mostrar em alto nível o relacionamento entre as classes da arquitetura, representando somente os principais métodos e atributos para o entendimento do seu funcionamento geral.

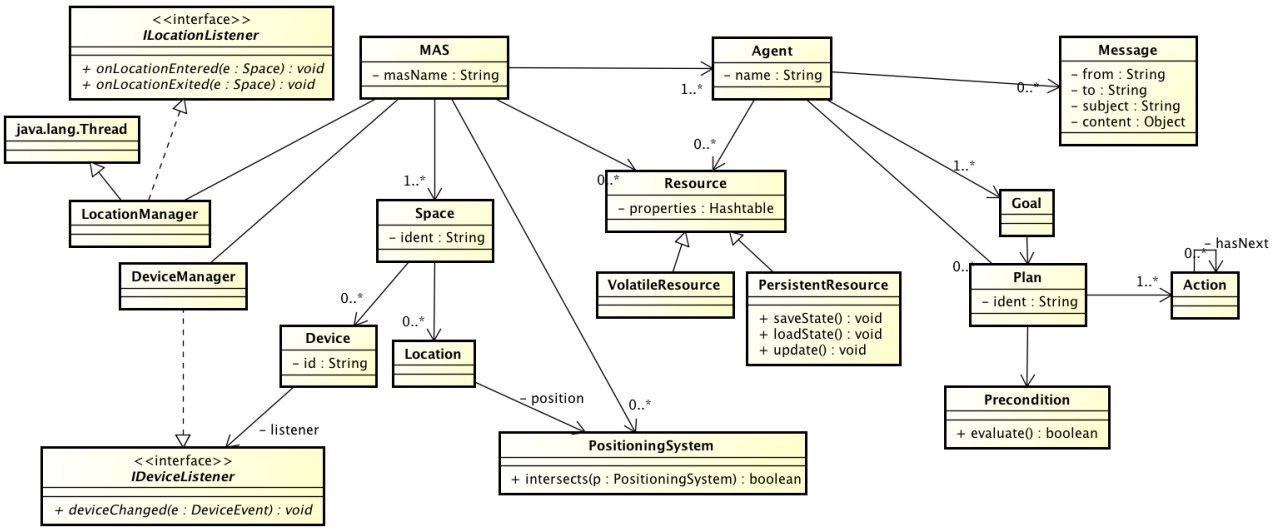


Figura 19 – Diagrama com as principais classes da plataforma de execução.

A classe **MAS** representa o sistema multiagente e centraliza os principais elementos da linguagem. Ela possui uma lista de agentes, espaços, sistemas de posicionamento e recursos. A classe **Agent** representa a estrutura interna básica de um agente, bem como a lista de objetivos e planos que ele possui. A classe **Goal** possui a referência para um plano de ação. A classe **Plan** representa um plano e possui uma lista

de ações, representadas pela classe **Action**. A classe *Action* implementa a interface *Runnable* para assumir o comportamento de *thread*. Ela também possui um método chamado *body*, que recebe em tempo de geração de código o conjunto de sentenças declaradas pelo desenvolvedor no bloco *perform*, conforme explicado na seção que detalha a sintaxe da linguagem.

A classe **Message** representa a estrutura básica de uma mensagem trocada entre agentes. A classe abstrata **Resource** representa um recurso no sistema, possuindo duas subclasses, **VolativeResource** e **PersistentResource** que implementam respectivamente a lógica de recurso volátil e recursos persistente. A classe *PersistentResource* visa abstrair a persistência do recurso em banco de dados. A versão atual da plataforma de execução não implementa essa funcionalidade. Entretanto, é apontado posteriormente nos trabalhos futuros a utilização de banco de dados SQLite[2], visto que o mesmo consiste de um banco de dados embarcado e por ser utilizado em sistemas modernos e atuais como o iOS[3] da Apple e o Android[4] da Google.

A classe **Location** representa uma localização e possui uma posição, especificada pela classe abstrata **PositioningSystem**. A classe *PositioningSystem* é um ponto de flexibilidade da arquitetura. O desenvolvedor deve estender essa classe para implementar a lógica dos sistemas de posicionamentos a serem utilizados pela aplicação. A classe **Space** possui uma lista de localizações e uma lista de dispositivos. A classe abstrata **Device** é outro ponto de flexibilidade da arquitetura. Ela deve ser estendida para cada tipo de dispositivo que o sistema implemente. A classe e *Device* possui um *listener* (do tipo **IDeviceListener**, que é uma *interface*), que deve ser invocado pelo desenvolvedor a partir do algoritmo que detecta alteração no estado do dispositivo.

A classe **DeviceManager** possui uma referência do objeto MAS para poder ter acesso à lista de espaços e agentes. Ela implementa a interface *IDeviceListener* sendo responsável por receber os eventos de alteração nos dispositivos e por repassar essas informações para os agentes.

Por fim, a classe **LocationManager** estende a classe *java.lang.Thread*. Ela é responsável por verificar periodicamente por alterações na localização corrente,

---

[2] http://www.sqlite.org/

[3] http://www.apple.com/br/ios/

[4] http://www.android.com/

verificando se o usuário entrou ou saiu de um determinado espaço, gerando eventos para os agentes.

## 4.3 Estendendo a Arquitetura Padrão

Devido aos dispositivos e sistemas de posicionamento serem dependentes do hardware no qual o sistema irá executar, torna-se difícil possuir implementações nativas na plataforma. Dessa forma, tais elementos foram definidos como abstratos na arquitetura.

A vinculação entre um dispositivo, ou sistema de posicionamento declarado na linguagem AA e sua implementação em Java só ocorre em tempo de geração de código, isto é, o identificador usado na sua declaração deve ser o nome da classe Java que o implementa.

A interface *PositioningSystem*, cujo código é apresentado na Listagem 14, possui os métodos abstratos `intersects` e `setup`. No método `intersects` o desenvolvedor implementa a lógica de interseção com outra posição, também definida por um sistema de posicionamento. Esse método é invocado periodicamente pelo *LocationManager*, que testa todas as posições estáticas definidas nos espaços a fim de determinar se alguma destas posições intersecta com a posição determinada dinamicamente pelo sistema de posicionamento, referente ao dispositivo móvel que executa a aplicação. No método `setup` o desenvolvedor deve definir a lógica de inicialização do sistema de posicionamento. Esse método será invocado automaticamente na inicialização do sistema multiagente.

Listagem 14 – Código de interface *PositioningSystem.*

```
public interface PositioningSystem {

      public boolean intersects(PositioningSystem position);

      public void setup();
}
```

O código apresentado na Listagem 15 ilustra a criação de um sistema de posicionamento para um GPS simulado. A classe GPS implementa a interface *PositioningSystem* e estende a classe *JFrame* para criar uma janela para a interação com o usuário. Ela possui duas caixas de texto para o usuário entrar com a latitude e longitude, e uma caixa de combinação para o usuário selecionar uma entre as

localizações predefinidas. Essa interface gráfica serve basicamente para o usuário simular a alteração de sua localização corrente, selecionando uma das opções da caixa de combinação. Dessa forma, o valor selecionado é considerado a localização corrente do dispositivo móvel do usuário.

Listagem 15 – Exemplo de criação de um sistema de posicionamento.

```
public class GPS extends JFrame implements PositioningSystem {

      private JTextField txtLatitude = new JTextField("0");
      private JTextField txtLongitude = new JTextField("0");
      private JComboBox cbLocations = new JComboBox();

      public double latitude, longitude;
      public int radius;
```

Além dos atributos de controle da interface, o sistema de posicionamento deve especificar a estrutura necessária para representar uma posição, neste caso, uma coordenada GPS simples, que é formada pelos atributos *latitude* e *longitude* do tipo *double*, e pelo atributo *radius* do tipo *int*.

O método `intersects` da classe *GPS* (Listagem 16) recebe por parâmetro uma posição (enviada pelo *LocationManager*) do tipo *PositioningSystem* e testa se esta posição é compatível (mesmo tipo de objeto, linha 5). O cálculo consiste em medir a distância entre a coordenada GPS da posição a ser testada e a coordenada GPS do dispositivo móvel (simulada através das caixas de texto (linhas 7 e 8). Se a distância for menor ou igual ao raio definido pela posição, o método retorna *true* indicando que houve interseção, ou *false* caso contrário.

Listagem 16 – Cálculo de interseção do GPS simulado.

```
1  @Override
2  public boolean intersects(PositioningSystem position) {
3        if (position instanceof GPS) {
4
5              GPS p = (GPS) position;
6
7              latitude  = Double.parseDouble(txtLatitude.getText());
8              longitude = Double.parseDouble(txtLongitude.getText());
9
10             double distance = GEO.distanceInMeters(latitude, longitude,
11                                    p.latitude, p.longitude);
12
13             if (distance <= p.radius)
14                   return true;
15       }
16
17       return false;
18 }
```

O segundo ponto de flexibilidade é a classe abstrata *Device*, cujo código é apresentado na Listagem 17. Um dispositivo possui um identificador (atributo `id`), que recebe o valor informado pelo desenvolvedor no momento de sua especificação. O atributo `listener` (do tipo *IDeviceListener*) permite ao desenvolvedor notificar a plataforma quando for detectada alteração no estado interno do dispositivo.

Listagem 17 – Código da classe *Device*.

```
1  public abstract class Device {
2
3        protected String id;
4        protected ArrayList<IDeviceListener> listeners;
5
6        public void addListener(IDeviceListener listener) {
7              this.listeners.add(listener);
8        }
9  }
```

O exemplo apresentado na Listagem 18 ilustra a implementação de um dispositivo simulado, que recupera dinamicamente o nível de bateria de um notebook. O atributo `last`, do tipo *DeviceEvent* guarda o último evento gerado pelo dispositivo. Assim como ocorre no sistema de posicionamento, o desenvolvedor deve especificar a estrutura do dispositivo. Neste caso, o dispositivo possui apenas a propriedade chamada `batteryLevel`.

Listagem 18 – Exemplo de implementação de um dispositivo.

```
1  public class Notebook extends Device implements Runnable {
2
3        private int batteryLevel;
4
5        private DeviceEvent last;
6
7        private void verifyState() {
8              int level = getBatteryLevel();
9
10             DeviceProperty p = last.getProperty("batterylevel");
11
12             if (!p.getValue().equals(String.valueOf(level))) {
13                   DeviceProperty newP =
14                         new DeviceProperty("batterylevel",
15                         p.getValue(), String.valueOf(getBatteryLevel()));
16                   last = new DeviceEvent(id);
17                   last.addProperty(newP);
18
19                   listener.deviceChanged(last);
20             }
21       }
22    ...
```

Essa classe também implementa a interface *Runnable*, permitindo-a executar como uma *thread.* Ela verifica a cada 30 segundos o nível de carga da bateria do dispositivo utilizando o método `getBatteryLevel`. Esse método realiza uma chamada de sistema e retorna um valor inteiro representando o percentual do nível restante de bateria, que posteriormente é comparado com o último valor coletado. Caso seja detectada uma alteração, um novo evento é gerado e enviado para o *listener* (linha 19).

Como um dispositivo pode conter inúmeras informações sobre seu estado interno, um evento consiste de uma lista de propriedades, definidas pela classe *DeviceProperty*. Uma propriedade é uma estrutura chave-valores, onde a chave é um valor *String* que identifica a propriedade (definida pelo desenvolvedor ao especificar o dispositivo) e os valores representam o valor antigo e o novo valor da propriedade em consideração.

## 4.4 O Compilador da Linguagem

O compilador da linguagem AA foi construído utilizando a ferramenta JavaCC (*Java Compiler Compiler*) [JCC11]. A especificação da gramática da linguagem é feita utilizando uma notação própria do JavaCC combinada com código Java para a montagem da árvore sintática. O compilador da linguagem AA foi construído a partir da especificação sintática da linguagem Java 5, disponível no repositório de gramáticas do site do JavaCC [JCC12]. A ideia foi aproveitar estruturas gramaticais consolidadas tais como declaração de variáveis, estruturas de controle de fluxo, etc. A partir dessa gramática foram adicionados os construtores propostos na linguagem AA. Como a linguagem Java é orientada a objetos, ela possui sentenças de declaração de classes e interfaces. Tais construções foram removidas da gramática, mantendo-se entretanto, a sentença de instanciação de objetos para serem utilizadas somente no escopo do corpo das ações.

O resultado do uso da ferramenta JavaCC é um programa Java que implementa o compilador da linguagem. Um programa de entrada na linguagem AA deve ser escrito em arquivo texto de extensão “.aa”. Opcionalmente o desenvolvedor poderá utilizar arquivos de extensão “.lib” para organizar o código. Nos arquivos “lib” somente são permitidas a declaração de dispositivos e sistemas de posicionamento. O restante do compilador também foi desenvolvido em Java. A Figura 20 mostra o diagrama de componentes referente à organização interna da arquitetura do compilador.

O componente *Compiler* reúne o arquivo .jj que representa a gramática da linguagem e as classes responsáveis pelo analisador sintático. O compilador é

representado pela classe *AACompiler*. As demais classes deste componente são geradas automaticamente pelo JavaCC. A classe *AACompiler* define também o fluxo principal de compilação que compreende as seguintes fases: (i) análise sintática, (ii) montagem da árvore sintática, (iii) análise semântica e montagem da tabela de símbolos, e (iv) geração de código.

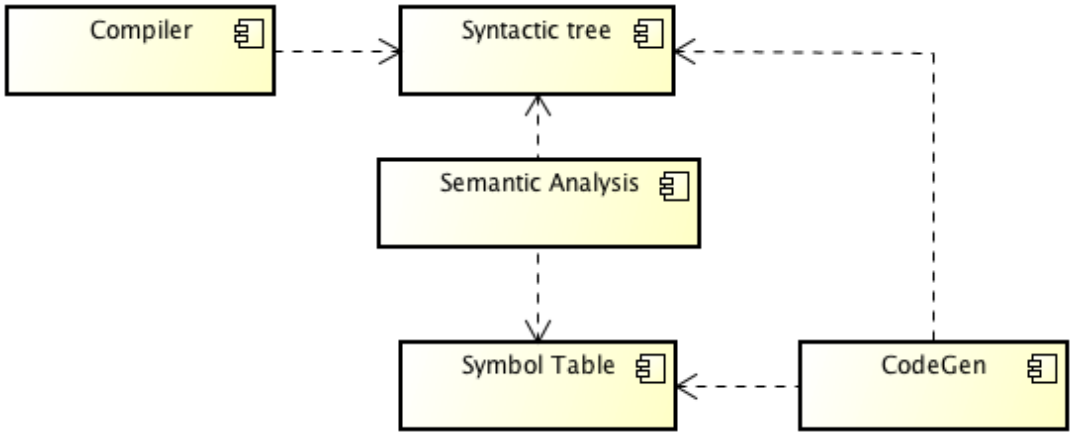


Figura 20 – Organização da arquitetura do compilador AA.

O componente *Syntactic Tree* contém todas as classes necessárias para representar a árvore sintática referente a um programa AA. Sua estrutura interna é organizada em três pacotes: *exp*, *general* e *statement*.

O pacote ***general*** possui as classes para representar elementos de propósito geral tais como os tipos de dados primitivos e os tipos de dados compostos, como agentes, dispositivos, localizações, espaços e demais conceitos abstraídos pela linguagem. A classe abstrata *GeneralNode* representa qualquer entrada na árvore sintática. Ela possui como atributo um objeto do tipo Token, que representa um lexema e seu tipo reconhecidos pelo analisador léxico. Para representar os conceitos de sistemas multiagentes na árvore sintática são utilizadas as classes: *AgentNode*, *ActionNode*, *PlanNode*, *GoalNode MASNode* e *ResourceNode*. Para representar os conceitos de ubiquidade são utilizadas as classes *DeviceNode*, *LocationNode*, *PositioningSystemNode* e *SpaceNode*. Existem também classes para representar tipos primitivos (PrimitiveTypeNode) e tipos de referência (ReferenceTypeNode).

O pacote ***ext*** contém as classes para representar expressões na linguagem, tais como operações aritméticas, operações lógicas, etc. Toda expressão deriva da classe *ExpreNode*, que por sua vez deriva da classe *GeneralNode*. O pacote ***statement*** contém as classes que representam as sentenças da linguagem, tais como sentenças de controle

de fluxo (representadas pelas classes *ForNode*, *WhileNode* e *IfNode*), sentenças de declaração de propriedades (FieldDeclNode), etc.

O componente *Semantic Analysis* contém as classes responsáveis pelo analisador semântico da linguagem. Essas classes utilizam a tabela de símbolos (classe *Symtable*) como estrutura auxiliar na validação das regras semânticas da linguagem. Sua principal função é controlar o escopo de declaração dos elementos ao longo da verificação da árvore sintática. São cinco classes que formam o analisador semântico: *AgentAnalysis*, *DeviceAnalysis*, *PSAnalysis*, *ResourceAnalysis* e *SpaceAnalysis*. Apesar de o compilador AA integrar partes da gramática do Java, o analisador semântico não incorpora todas as regras semânticas da linguagem Java.

Por fim, o componente *CodeGen* contém as classes responsáveis pela geração de código. A classe principal é a *MASCodeGen*. Ela recebe como entrada a raiz da árvore sintática, percorrendo seus nodos recursivamente e delegando a geração do código para classes especializadas. Para gerar estruturas de agentes existe a classe AgentCodeGen. As demais estruturas são geradas conforme sua classificação na tabela de símbolos. As principais classes são: *ExpressionCodeGen* para expressões e *StatementCodeGen* para as sentenças da linguagem. Internamente a essas classes existe um método para cada tipo de nodo previsto na árvore sintática. Caso o nodo não seja reconhecido é disparada uma exceção do gerador utilizando a classe *NotImplementedException*.

A saída do gerador de código consiste de um conjunto de arquivos Java compatíveis com a plataforma de execução. Visto que a entrada para o gerador é a árvore sintática, espera-se ser possível gerar código para outras plataformas apenas alterado os métodos das classes que formam o componente *CodeGen*. Ao gerar código para uma nova plataforma espera-se a verificação das estruturas dessa plataforma em relação aos modelos apresentados, a fim de verificar a compatibilidade entre elas.

# 5 AVALIANDO A LINGUAGEM PROPOSTA

A linguagem proposta será avaliada de duas formas. A primeira forma (seção 5.1) demonstra a aplicabilidade da linguagem proposta através do desenvolvimento de uma aplicação multiagentes ubíqua, utilizando os diferentes construtores que a linguagem oferece. A segunda forma demonstra como propostas que não foram projetadas para o desenvolvimento de SMAs ubíquos foram estendidas a fim de incorporar os conceitos necessários para produzir uma mesma aplicação multiagentes ubíqua. Ao final, são feitas avaliações qualitativas sobre o impacto que tais extensões causaram nas abordagens escolhidas.

A primeira extensão (seção 5.2) utiliza uma linguagem existente para o desenvolvimento sistemas multiagentes, adicionando a ela conceitos de ubiquidade. A segunda extensão (seção 5.3) utiliza um *framework* que possibilita a construção de aplicações conscientes de contexto e localização, adicionando a ele conceitos básicos de agentes e SMAs.

## 5.1 Desenvolvendo um Sistema Multiagente Ubíquo

Essa seção exemplificará a linguagem AA através do desenvolvimento de um sistema multiagente ubíquo. Para descrever a aplicação será utilizada uma abordagem narrativa, enfatizando quando apropriado o conceito da linguagem sendo utilizado e sua respectiva sintaxe.

A *UbiCampus* é uma aplicação que visa ajudar estudantes a localizarem seus colegas no campus de uma universidade. Os principais prédios são mapeados como regiões simbólicas, tais como uma biblioteca, uma sala de aula e um auditório.

Todos os usuários que estiverem com a aplicação executando em seus smartphones poderão ser localizados no câmpus. A localização de um usuário é determinada através do nome da rede atribuída ao *access point* Wi-Fi que o usuário está conectado.

A aplicação possui dois agentes administrativos: o *LocationServiceAgent* e o *EnrollmentServiceAgent*. O agente de *localização* é responsável por gerenciar a identificação dos usuários em um espaço. Ele possui duas funções principais: (i) recuperar a lista de usuários em um espaço em particular, e (ii) consultar em que espaço um determinado usuário está. A implementação desse serviço foi necessária pois a

plataforma não possui tais funções implementadas nativamente, isto é, cada agente tem acesso somente a sua localização.

O agente de *matrícula* é responsável pelo gerenciamento das inscrições dos alunos nos cursos. Sua principal função é recuperar a lista de alunos matriculados em mesmo curso que um estudante em particular. Combinando estes dois serviços, o usuário será capaz de pesquisar por seus colegas, verificando se algum deles está próximo.

Para facilitar a interação entre os agentes da aplicação, realizada através da troca de mensagens, foram criadas algumas palavras-chave que são utilizadas para identificar uma mensagem. Esse mecanismo é inspirado na linguagem ACL (*Agent Communication Language*) FIPA, onde são definidas performativas que informam do que se trata uma determinada mensagem.

Na linguagem AA, uma mensagem é formada basicamente pelos campos assunto, o agente remetente, o agente destino, e o conteúdo. O conteúdo da mensagem é implementado utilizando o tipo *Object* do Java, permitindo receber qualquer tipo de elemento. Dessa forma, as palavras-chave utilizadas na aplicação *UbiCampus* para permitir a interação entre os agentes são informadas no campo *suject*.

O código completo da aplicação é apresentado no Apêndice A. A Listagem 19 apresenta um fragmento de código da declaração do agente do usuário. Este agente possui um objetivo relacionado à principal funcionalidade da aplicação que é localizar os colegas de classe de um usuário.

Listagem 19 – Trecho de código do agente *UserAgent*.

```
1  agent UserAgent {
2     goal lookForColleagues;
3
4     reacts {
5        on locationchanged l {
6           if(l.type.equals("entered"))
7              achieve lookForColleagues;
8           }
9        }
10
11       plan lookupPlan for lookForColleagues {
12          first action lookupColleagues {
13             next lookupLocations;
```

Como o usuário pode movimentar-se pelo campus, a estratégia de implementação escolhida foi o agente implementar uma reação a mudanças na localização, conforme linhas 4 e 5. Sempre que o agente perceber que entrou em uma

localização (linha 6), ele inicia a busca pelo seu objetivo chamado **lookForColleagues** (linha 7).

Esse objetivo possui um plano associado (linha 11) com uma ação principal (*lookupColleagues*) que é responsável por enviar uma mensagem para o agente de matrícula solicitando a lista de colegas do usuário.

Para o envio da mensagem, o agente utiliza a função da plataforma chamada *send*. Essa função possui diversas sobrecargas, sendo a principal que recebe os argumentos nome do agente destino, assunto da mensagem e conteúdo, conforme Listagem 20, que ilustra a seção `perform` da ação *lookupColleagues*. O termo "Colleagues" serve neste caso para indicar que o agente do usuário requer a lista de colegas de um determinado usuário, informado no campo conteúdo da mensagem.

Listagem 20 – Código da seção *perform*.

```
perform {
   String username = "Mauricio";
   send("EnrollmentServiceAgent", "Colleagues", username);
}
```

A primeira ação do plano encerra-se após o envio da mensagem. Depois disso, o agente executará a ação chamada **lookupLocations**, conforme código apresentado na Listagem 21. Primeiramente essa ação espera pela recepção de uma mensagem, através da chamada ao método *nextMessage*, na linha 4. Este método da plataforma é bloqueante. Sua função é remover o primeiro elemento da lista de mensagens recebidas pelo agente, retornando um elemento do tipo *Message*. Note que a implementação neste caso é simplificada. O agente considera que a mensagem recebida neste momento será a resposta do agente de matrícula; não é testado o remetente da mensagem. Em uma aplicação de maior complexidade, mais testes seriam necessários para garantir a execução *correta* da funcionalidade em consideração.

O conteúdo da mensagem recebida do agente de matrícula será um vetor contendo o nome dos colegas do usuário (linha 5), que é encaminhado via mensagem para o agente de localização (linha 6), utilizando o termo "Locations" como performativo. O agente novamente aguarda por uma resposta, que agora será uma tabela, contendo o nome do estudante e sua respectiva localização (linhas 7 e 8). De posse dessas informações, no restante do código, o agente do usuário verifica se algum dos colegas do seu usuário colegas está próximo, exibindo uma janela de alerta (linha 18) para notificar o usuário.

O agente **EnrollmentServiceAgent** possui um objetivo que é prover a lista de colegas de um determinado usuário. Esse agente, conforme trecho da Listagem 22, possui comportamento reativo à recepção de uma mensagem no ambiente (linhas 13 a 19). Ao receber uma mensagem, o agente testa se esta contém o termo “Colleagues” no assunto. Nesse caso ele inicia seu objetivo *provideClassInfo*. Para simplificar a implementação, o agente define estaticamente a lista de estudantes, organizados em duas turmas (*classA* e *classB*). Essas informações são armazenadas em um recurso do agente chamado *base*, definidas no momento da inicialização do agente (linhas 5 a 11). Em uma aplicação real, esses recursos seriam substituídos pelos bancos de dados da universidade.

Listagem 21 – Declaração da ação *lookupLocations*.

```
1  action lookupLocations {
2     perform {
3        print("Lookup rLocations");
4        Message msg = nextMessage();
5        string[] colleagues = (string []) msg.content;
6        send("LocationServiceAgent", "Locations", colleagues);
7        msg = nextMessage();
8        string[][] locations = (string[][]) msg.content;
9        string myLocation = Network.apName;
10       boolean found = false;
11
12       for(string [] ul : locations) {
13          // ul[0] -> username
14          // ul[1] -> location
15          if(ul[0] != null) {
16             if(ul[1].equals(myLocation)) {
17                found = true;
18                alert(ul[0] + " is nearby.");
19             }
20          }
21       }
22       if(!found)
23          alert("Sorry, there are no friends nearby.");
24    }
25 }
```

Para cumprir seu objetivo, o agente de matrícula possui um plano de ação, como mostrado na Listagem 23. Esse plano possui somente uma ação, chamada *logic*. De posse do nome do usuário recebido por mensagem, ela percorre a lista de estudantes das turmas armazenadas em seu recurso e monta uma lista (declarada na linha 9) a ser retornada para o agente do usuário.

A estrutura do agente *LocationServiceAgent* (Listagem 24) é similar ao agente de matrícula. Seu comportamento também é reativo, isto é, ele reage ao recebimento de mensagens para determinar qual objetivo perseguir. Ao perceber uma mensagem, o

agente testa seu assunto, considerando apenas duas possibilidades: *Locations* ou *UpdateLocation*, conforme linhas 9 a 12.

Listagem 22 – Definição do plano que recupera a lista de colegas.

```
1  plan retrieveStudentList for provideClassInfo {
2     first action logic {
3        perform {
4           Message msg = nextMessage();
5           string username = (string) msg.content;
6           string [] c1 = (string[]) base.get("classA");
7           string [] c2 = (string[]) base.get("classB");
8
9           string [] colleagues = null;
10
11          for(string classUser : c1){
12              <restante do código>
```

Listagem 23 – Declaração do agente *EnrollmentServiceAgent*.

```
1  agent EnrollmentServiceAgent {
2     goal provideClassInfo;
3     resource base;
4
5     init {
6        string [] classA = { "Mauricio", "Rodrigo", "Anderson" };
7        string [] classB = { "Rafael", "Ana", "Ricardo", "Marcelo" };
8
9        base.set("classA", classA);
10       base.set("classB", classB);
11    }
12
13    reacts {
14       on messagereceived m {
15          if(m.subject.contains("Colleagues"))
16              achieve provideClassInfo;
17          }
18       }
19    }
20    ...
```

Listagem 24 – Declaração do agente *LocationServiceAgent*.

```
1  agent LocationServiceAgent {
2     goal provideLocationInfo;
3     goal updateLocation;
4
5     resource base;
6
7     reacts {
8        on messagereceived m {
9           if(m.subject.equals("Locations"))
10             achieve provideLocationInfo;
11          else if(m.subject.equals("UpdateLocation"))
12             achieve updateLocation;
13       }
14    }
15    ...
```

Para cada *tipo* de mensagem, o agente inicia a execução de um objetivo, definidos na linha 2 e 3. No objetivo *updateLocation*, o agente armazena em sua base a localização recebida de outros agentes. A *base* neste caso é um recurso declarado no escopo do agente, conforme linha 5. Já no objetivo *provideLocationInfo,* o agente recupera a localização de determinados usuários consultando em sua base e retornando uma tabela de estrutura simples (onde cada linha da tabela é um par chave-valor contendo <usuário, localização>) para o agente requisitante.

Por fim, o cenário exemplo da aplicação compreende três espaços que representam prédios da universidade, que são: *Classroom*, *LectureHall* e *Library*, conforme trecho de código ilustrado na Listagem 25. Cada espaço possui uma localização definida em função do sistema de posicionamento Network, na linha 1, que possui como estrutura uma propriedade do tipo *string* que representa o nome da rede Wi-Fi fornecida pelo *access point*.

Listagem 25 – Definição das regiões simbólicas da aplicação.

```
 1 positioningsystem Network {
 2       string apNane;
 3 }
 4
 5 space Classroom {
 6    location l of Network {
 7       apName = "Classroom";
 8    }
 9 }
10
11 space Library {
12    location l of Network {
13       apName = "Library";
14    }
15 }
16
17 space LectureHall {
18    location loc of Network {
19       apName = "LectureHall";
20    }
21 }
```

A plataforma AA abstrai diversos detalhes de implementação no que refere-se ao gerenciamento das localizações. O sistema de posicionamento *Network* é uma implementação padrão que utiliza o nome da rede provida pelo *access point wireless* para atribuir a localização corrente da instância do SMA em execução no dispositivo do usuário. Para a implementação de cenários mais complexos, e para permitir a decomposição de um espaço em múltiplas e precisas localizações, um sistema de posicionamento por GPS pode ser utilizado.

## 5.2 Estendendo a plataforma Jason

O Jason foi escolhido para incorporar conceitos de computação ubíqua devido a sua ampla utilização e também por ser disponibilizado em código aberto. Além disso, outra facilidade é o fato de que o Jason é baseado em Java e foi construído para ser extensível, permitindo aos desenvolvedores adicionarem novas funcionalidades.

Para um agente atuar efetivamente em um sistema multiagente, ele deve interagir com outros agentes. O interpretador do Jason provê uma arquitetura de agente que faz a fronteira entre o agente e o mundo externo. Tal arquitetura provê um mecanismo de percepção (que modela os sensores do agente), de atuação (que modela os atuadores do agente), e como os agentes recebem mensagens de outros agentes. Estes aspectos também podem ser customizados para cada agente individualmente.

Na extensão do Jason, conforme ilustrado na Figura 21, a arquitetura do agente foi customizada adicionando a ela características de ubiquidade, representadas pelas classes com fundo verde. A classe com fundo branco faz parte do *framework* Java, e a classe com fundo amarelo pertence a arquitetura do Jason. Esta nova arquitetura, chamada de **UbiquitousAgArch** estende a arquitetura padrão do Jason definida pela classe **AgArch**, permitindo ao agente acessar informações sobre localizações e espaços em um ambiente ubíquo.

Similarmente à linguagem AA, foi implementado um gerenciador de localizações (classe *LocationManager*) para gerenciar a localização corrente e notificar os agentes no caso de mudança de localização. Ela também customiza a implementação padrão da percepção do agente, gerando uma crença (utilizando a palavra reservada *location*) que representa uma percepção do agente em relação a sua mudança de localização no ambiente.

Esta percepção pode então ser usada como um evento disparador ou condição de teste para um objetivo do agente, como ilustrado na Listagem 26:

Listagem 26 – Exemplo de percepção utilizando o termo *location*.

```
1  +location(Type, Space)
2        <- .print(Type, " space ", Space);
```

No exemplo, *location* é o predicado padrão para um evento de localização. O argumento *Type* é o estado do evento (que pode ser *entered* ou *exited*), e o argumento *Space* representa o identificador do espaço detectado.

Na linguagem AA, *location* é uma palavra reservada. Na extensão do Jason, foi utilizado um predicado em particular que representa a noção de localização para o agente. Entretanto, os desenvolvedores podem utilizar o predicado com outro significado em seus programas, e similarmente para outros conceitos relacionados a ubiquidade que irão requerer alguma representação na base de crenças do agente. Isto implica que programas que executam corretamente no Jason, podem não funcionar quando executarem em uma extensão do Jason para ubiquidade. Esta é uma das desvantagens de abordagens que não possuem abstrações de primeira ordem para os conceitos de ubiquidade.

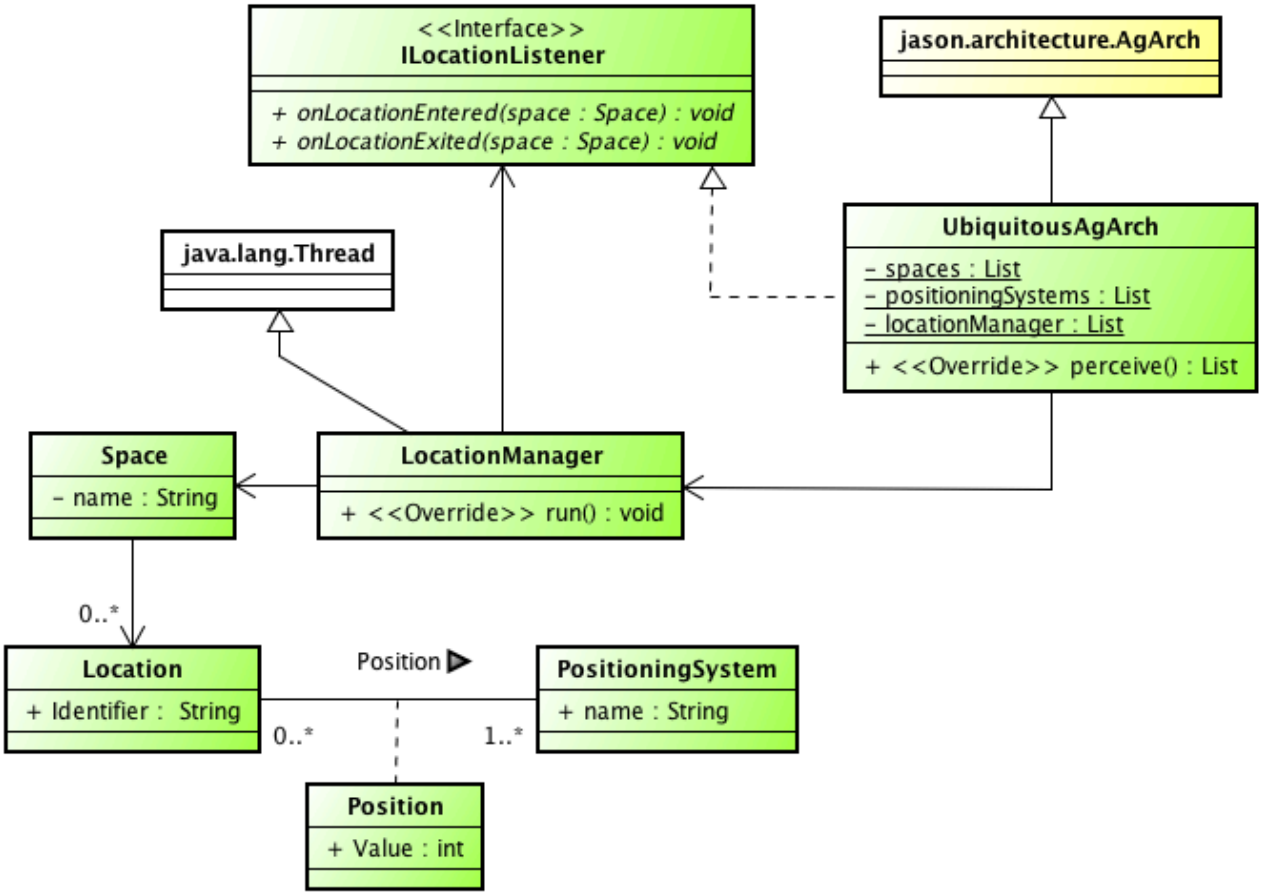


Figura 21 – Extensão da arquitetura do Jason.

Para demonstrar a utilização da extensão do Jason foi implementada a aplicação UbiCampus (apresentada na seção 5.1). O código completo é mostrado no Apêndice B. No Jason cada agente é implementado em um arquivo próprio, de extensão ".asl". O nome do arquivo representa o nome simbólico do agente. Na aplicação proposta existem três agentes principais: mauricio.asl, campusManager.asl e secretary.asl.

O código do agente *mauricio* pode ser visto na Listagem 27. O predicado *location* é utilizado para indicar quando uma alteração de localização é percebida pelo agente. Ao

perceber mudança na localização, ele envia uma mensagem para o agente *campusManager* transmitindo o evento de localização gerado como conteúdo da mensagem.

O agente *mauricio* também possui um objetivo cujo evento disparador é o predicado *nearby*. Como argumento, o objetivo recebe uma variável *Students* que representa a lista de estudantes em sua mesma localização, determinados pelo agente *campusManager*. De posse da lista de estudantes, o agente *mauricio* envia uma mensagem para o agente *secretary*, passando a lista de estudantes a fim de verificar se algum destes é seu colega de classe. Caso algum dos alunos seja seu colega, o agente *mauricio* mostra uma mensagem na console da aplicação informado o nome de seus colegas (linha 12).

Listagem 27 – Código contendo a lógica de recuperação dos *colegas próximos*.

```
 1 +location(Type, Space)
 2       <-
 3       .send(campusManager, achieve, locationUpdate(mauricio, Type, Space)).
 4
 5 +!nearby(Students) : .member(mauricio, Students)
 6       <-
 7       .send(secretary, askOne, classmates(Students, _), classmates(_,C));
 8       .delete(mauricio, Students, P);
 9       .print(" > People neaby ", P);
10       .member(mauricio, C);
11       .delete(mauricio, C, L);
12       .print(" > Neaby classmates ", L).
```

Para fins de demonstração, o agente *campusManager* (Listagem 28) possui algumas crenças predefinidas, utilizando o predicado *located* que representa uma pessoa e sua respectiva localização. A regra *prox* é utilizada para determinar pessoas em uma mesma localização. O objetivo locationUpdate é disparado toda vez que o agente do usuário varia de localização. Se o evento for de entrada em uma localização (linha 9), o *campusManager* adiciona um novo fato a sua base de crenças (linha 11) e, a seguir, determina a lista de pessoas próximas (utilizando a regra *prox*). Ao final essa lista é enviada para todos as pessoas (agentes) que a compõe (linha 13). Caso o evento seja de saída de localização (linha 16), o *campusManager* apenas remove a crença associada ao usuário que gerou o evento.

Por fim, o agente *secretary* (Listagem 29) possui apenas uma turma em sua base de crenças. Sua única função é determinar a lista de colegas de um determinado estudante, através da regra *classmates* (linha 3).

Listagem 28 – Código do agente *campusManager*.

```
 1 located(anderson, facin).
 2 located(rafael, facin).
 3 located(ana, ifrs).
 4 located(marcelo, rio).
 5
 6 prox(Place, People) :- .findall(P,located(P,Place),People).
 7
 8 +! locationUpdate (Username, Type, Location) [ source(Sender) ]
 9    : Type == entered
10    <-
11       +located(Username, Location);
12       ?prox(Location,R);
13       .send(R, achieve, nearby(R)).
14
15 +! locationUpdate (Username, Type, Location) [ source(Sender) ]
16    : Type == exited
17    <-
18       -located(Username, Location).
```

Listagem 29 – Código do agente *secretary*.

```
1 class([ana, rodrigo, anderson, mauricio]).
2
3 classmates(Students, C)
4      :- class(Class) & .intersection(Class, Students, C).
5
6 +?find (Students)
7       <-
8       ?class(Class);
9       .intersection(Class, Students, Classmates).
```

## 5.3 Estendendo o framework MoCA

O *framework* MoCA [SER+04] foi utilizado por possuir versão para *download*, possuir uma boa documentação no que refere-se a como estender sua arquitetura, e por disponibilizar bibliotecas em Java para acesso e customização de sua infraestrutura. O objetivo do MoCA é permitir o desenvolvimento de aplicações móveis com a capacidade de coletar e processar informações contextuais (sobre dispositivos) e informações sobre localização baseada em uma rede *wireless*.

O MoCA é organizado em componentes especializados que fornecem os serviços básicos da plataforma. O componente *monitor* é responsável por coletar informações de contexto no dispositivo do usuário, como por exemplo, nível de bateria e modelo. O componente distribuído CIS (*Context Information Service*) recebe e armazena as informações de contexto enviadas pelo *monitor*. O componente distribuído LIS (*Location Inference Service*) é responsável por inferir uma localização aproximada do dispositivo baseado em padrões de sinal *wireless* previamente determinados pelo administrador da

aplicação. O LIS também permite a definição de regiões simbólicas de forma retangular com tamanhos arbitrários, bem como a definição de regiões e sub-regiões.

Os componentes do MoCA podem ser distribuídos em diferentes máquinas interconectadas via rede local ou Internet. O componente *monitor* é um serviço que deve ser executado no dispositivo móvel a ser monitorado. O monitor possui um simulador e implementações para diferentes sistemas operacionais, tais como Windows XP, Windows CE, Symbian (Nokia S60) e Linux. Uma das dificuldades encontradas para testar o monitor foi que ele não acompanhou a evolução das tecnologias atuais, não possuindo compatibilidade com versões recentes do Windows (Vista ou 7), nem implementações para plataformas emergentes como iOS, Android ou Windows Phone.

O MoCA não possui elementos capazes de descrever entidades autônomas. Dessa forma foi necessária estender sua arquitetura a fim de adicionar os conceitos básicos para caracterizar uma aplicação multiagentes. O diagrama na Figura 22 mostra a arquitetura resultante da extensão do *framework* MoCA. As classes em verde representam os conceitos de ubiquidade e as classes em amarelo representam os conceitos de SMAs.

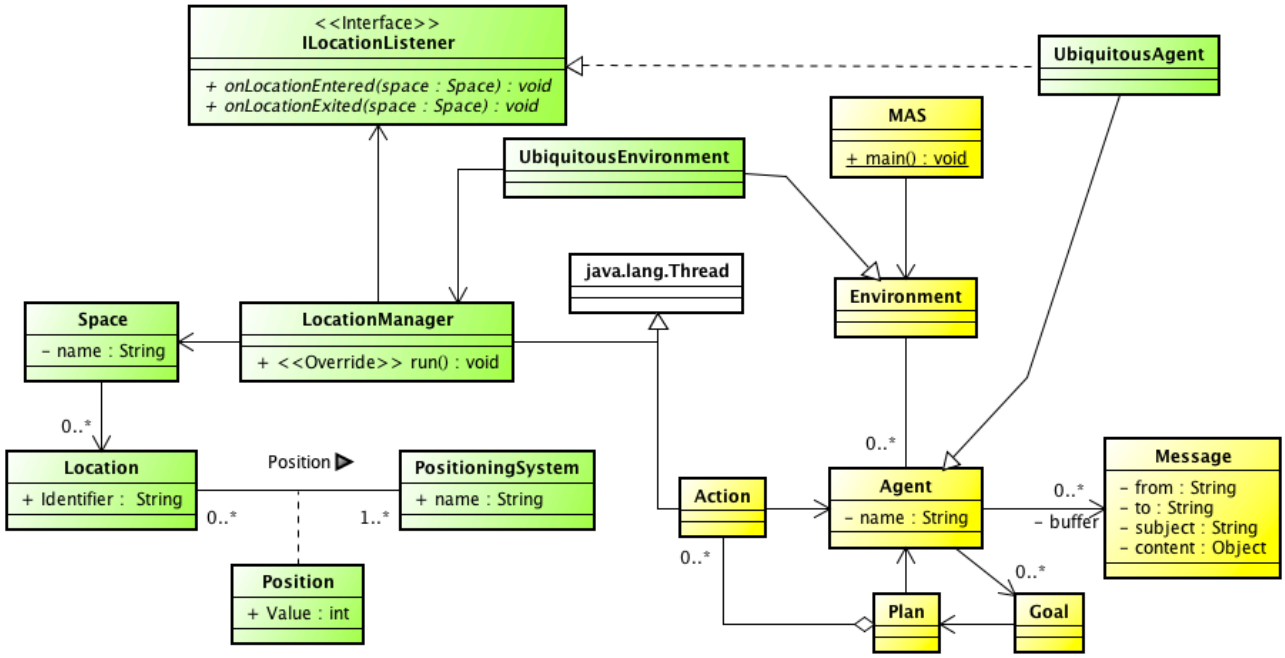


Figura 22 – Classes da extensão o *framework* MoCA.

A classe **MAS** é responsável pela instanciação do sistema multiagente. Ela possui um atributo do tipo **Environment** que representa o ambiente onde os agentes atuam e que por sua vez possui a lista de agentes e os espaços que formam o ambiente. A classe *Environment* também é responsável por transportar as mensagens trocadas entre os

agentes. Um agente é definido através da classe **Agent**, possuindo como atributo principal um nome, que o identifica unicamente no ambiente. Um agente possui três métodos principais: **setup**, **sense** e **plan**. No método *setup* o desenvolvedor pode definir o código de inicialização do agente. No *sense*, o desenvolvedor pode verificar informações capturadas no ambiente e no *plan* deve ser desenvolvida a lógica de atuação do agente. Tais métodos representam o ciclo de raciocínio simplificado do agente.

A classe **Goal** representa um objetivo do agente. Um objetivo possui um plano (classe **Plan**) associado, que por sua vez contém uma lista de ações, representadas pela classe **Action**. Uma ação é implementada como uma *Thread* em Java. O desenvolvedor deve criar classes concretas para cada tipo de ação que um agente desempenha. O controle de fluxo entre as ações fica a critério do desenvolvedor. Também foi criada uma classe chamada **Message**, para representar a estrutura de uma mensagem trocada entre os agentes. Uma mensagem possui quatro campos: agente remetente, agente destino, assunto e conteúdo.

Similarmente a extensão do Jason, foram criadas classes para representar as estruturas necessárias para caracterizar um sistema ubíquo. A classe **UbiquitousAgent** estende a classe padrão *Agent*, e implementa a interface **ILocationListener**. Essa interface possui dois métodos principais: *onLocationEntered(Space s)* e *onLocationExited(Space s)*. Esses métodos serão invocados pelo gerenciado de localização (explicado a seguir) dependendo do tipo de evento detectado. A classe **Space** representa um espaço ou região em um ambiente. Um espaço possui um identificador único e uma lista de localizações que definem seus limites. A classe **Location** representa uma localização ou ponto de interesse. Uma localização possui uma posição, que é um ponto discreto em um espaço, definida através de um sistema de posicionamento (classe **PositioningSystem**). Um sistema de posicionamento define os elementos que formam uma posição, expressos através de variáveis, e como esta posição deve ser interpretada.

A classe **LocationManager** possui a função de integração com os serviços do MoCA, e também centraliza dos sistemas de posicionamento que podem ser definidos pelo desenvolvedor. Ela é implementada como uma Thread em Java, recebendo periodicamente notificações sobre alterações na localização do dispositivo. A integração com o MoCA ocorre através de classes que formam a API *cliente*. Para acessar o serviço de localização (LIS) é utilizada a classe **moca.services.lis.LocationInferenteService** e seus métodos assíncronos.

O serviço LIS pode ser utilizado de duas formas: assíncrona e síncrona. O modo assíncrono funciona através da implementação de dois *listeners* a fim de receber notificações referentes a alterações em dispositivos (**moca.service.lis.even.DeviceListener**) ou alterações de localização (**moca.service.lis.even.RegionListener**). Para isso, a aplicação cliente deve registrar-se ao LIS utilizando o método **subscribe** da classe *LocationInferenteService*.

O LIS possui quatro métodos para chamadas síncronas, que são:

- `getDevices():String[]`, que retorna todos os dispositivos controlados pelo LIS;
- `getAreas():Area[]`, que retorna todas as áreas conhecidas pelo LIS;
- `getArea(String):String`, que retorna a área de um determinado dispositivo, cujo endereço MAC é informado por parâmetro; e,
- `getDevices(String):String[]`, que retorna os dispositivos em uma área específica informada por parâmetro.

Ao detectar uma alteração na localização corrente, utilizando os sistemas de posicionamento configurados, e os eventos do LIS, o *location manager* notifica os agentes (que implementam a classe *UbiquitousAgent*) invocando seus métodos `onLocationEntered`, ao entrar em um novo espaço, ou `onLocationExited`, ao sair de um determinado espaço. Ambos métodos recebem o espaço detectado como parâmetro.

Para demonstrar a utilização da extensão do MoCA, foi desenvolvida a aplicação UbiCampus (apresentada na seção 5.1). O código completo da aplicação é apresentado no Apêndice C.

**Definindo os agentes:** a aplicação é composta por três agentes principais, implementados em classes individuais sendo elas: *UserAgent*, *CampusManagerAgent* e *SecretaryAgent*. Apenas o agente do usuário (classe *UserAgent*) possui características de ubiquidade, dessa forma somente ela estende a classe *UbiquitousAgente*. Os demais agentes estendem a classe padrão *Agent*.

A Listagem 30 apresenta o código do agente de usuário. No método setup o agente realiza a configuração de seu plano, ação e objetivo. Ao entrar em uma localização (linha 16), ele inicia seu objetivo que é procurar por colegas próximos.

Listagem 30 – Código do agente *UserAgent* na extensão do MoCA.

```
1  public class UserAgent extends UbiquitousAgent {
2        private Goal lookForColleagues;
3        private Plan lookForColleaguesPlan;
4
5        public UserAgent(String name) {
6              super(name);
7        }
8
9        public void setup() {
10             lookForColleaguesPlan = new Plan();
11             lookForColleaguesPlan.add(new LookupForColleagues());
12             lookForColleagues = new Goal(lookForColleaguesPlan);
13       }
14
15       @Override
16       public void onLocationEntered(Space e) {
17             lookForColleagues.start();
18       }
```

Ao inicializar a execução de sua ação (Listagem 31), o agente do usuário envia uma mensagem para o agente *Secretary* solicitando a lista de seus colegas. O método `receive` bloqueia a execução da ação até que uma mensagem seja recebida. De posse da lista de colegas, o agente do usuário envia uma nova mensagem, agora para o agente *CampusManager*, enviando a lista de colegas solicitando suas localizações. Por fim, ao receber as localizações, o agente compara se alguma delas coincide com a localização atual mostrando um aviso para o usuário caso algum colega esteja próximo. Note que para fins de simulação não foram implementados protocolos de interação. Considera-se neste exemplo que as mensagens sempre chegarão na ordem correta.

Listagem 31 – Código da ação *LookupForColleagues* na extensão do MoCA.

```
1  public class LookupForColleagues extends Action {
2     @Override
3     public void run() {
4        Message request =
5           new Message(this.getName(), "Secretary", "Request", "Colleagues");
6
7        send(request);
8        Message reply = receive();
9        ArrayList<String> l = (ArrayList<String>) reply.getContent();
10       request = new Message(this.getName(), "CampusManager", "Request", l);
11
12       <restante do código>
```

Para fins de demonstração, as informações mantidas pelos agentes *Secretary* e *CampusManager* são predefinidas em memória, isto é, o agente *Secretary* possui uma lista fixa nomes que caracterizam uma turma, e o agente *CampusManager* possui uma tabela contendo uma lista de nomes e suas respectivas localizações. Embora estes dados

sejam fixos neste demonstração, a localização do usuário, utilizada pelo agente do usuário, é obtida dinamicamente utilizando os recursos do *framework*, não afetando dessa forma o objetivo geral da aplicação.

## 5.4 Considerações sobre o capítulo

Este capítulo apresentou os estudos realizados a fim de avaliar a linguagem proposta. Inicialmente foi demonstrada a implementação de uma aplicação (chamada UbiCampus) que utiliza características básicas de agentes (como a distribuição de processamento) e características de ubiquidade (localização e contexto) utilizando a linguagem proposta.

Em um segundo momento, foi proposta uma extensão da plataforma Jason. O objetivo foi identificar o impacto em sua abordagem da incorporação de elementos capazes de permitir o desenvolvimento de agentes ubíquos. Para demonstrar sua utilização, a aplicação UbiCampus também foi implementada utilizando essa extensão. Ao estender o Jason foi necessário adicionar um literal especial (*location*) para representar a percepção de um agente em relação a uma mudança de localização. Para demais conceitos que requerem um tratamento especial, novos literais deverão ser usados. Como esses literais não são palavras reservadas da linguagem, quanto mais literais forem utilizados pela extensão de ubiquidade, mais as chances de haver conflito com literais utilizados pelos desenvolvedores.

Por fim, foi proposta a extensão do *framework* MoCA, adicionando a ele conceitos de agentes, que não são nativos em sua arquitetura. Nesse caso, foram desenvolvidas classes necessárias que caracterizam o núcleo mínimo de um sistema multiagentes e a estrutura interna de um agente. Para finalizar a demonstração a aplicação UbiCampus também foi implementada utilizando essa extensão.

A arquitetura criada para o cenário implementa conceitos fundamentais e não trata características importantes como a distribuição de processamento, a troca de mensagens em um ambiente em rede, controle avançado de concorrência entre os agentes, dentre outras, que podem demandar um grande esforço de implementação.

A Figura 23 apresenta a comparação entre a linguagem AA, o Jason e o MoCA em relação aos principais conceitos apresentados no metamodelo U-MAS (abordado no capítulo 3). Considera-se como requisito atendido se a abordagem implementa o conceito como um recurso próprio.

| Critério \ Abordagem | | Jason | MoCA | AA |
|---|---|---|---|---|
| Ubiquidade | Regiões | | Sim | Sim |
| | Localização | | Sim | Sim |
| | Dispositivos | | Sim | Sim |
| SMA | Agentes | Sim | | Sim |
| | SMA | Sim | | Sim |
| | Plano de ação | Sim | | Sim |
| | Ação | Sim | | Sim |
| | Objetivo | Sim | | Sim |
| | Recurso | Não | | Sim |

Figura 23 – Tabela Comparativa em relação ao metamodelo U-MAS.

A Figura 24 apresenta um comparativo em relação aos aspectos que caracterizam um sistema ubíquo (abordados na Seção 2.3), que são:

- embarcados: diversos dispositivos integrados ao ambiente comunicando-se através de uma rede;
- conscientes de contexto: estes dispositivos podem reconhecer os usuários e o contexto do ambiente;
- personalizados: os sistemas podem ser customizados para as necessidades de cada usuário;
- adaptativos: os sistemas podem alterar seus comportamentos em resposta ao usuário;
- antecipatórios: os sistemas podem antecipar o desejo dos usuários sem a necessidade de mediação.

A classificação “Não” indica que a abordagem não implementa o conceito em consideração como um elemento nativo. A classificação “Parcial” indica que a abordagem fornece pontos de extensibilidade para que o conceito seja representado, e a classificação “Sim” indica que o conceito pode ser implementado utilizando os recursos oferecidos pela abordagem em consideração.

| *Critérios* | *Jason* | *MoCA* | *AA* |
|---|---|---|---|
| **Embarcados** | Não. O Jason não implementa a noção de dispositivo. | Parcial. O MoCA apenas armazena as informações sobre o estado do dispositivo do usuário que são coletadas através de um processo denominado monitor. | Sim. A linguagem AA abstrai a declaração de um dispositivo. Na linguagem AA os dispositivos são organizados em espaços que delimitam o seu acesso. A implementação da lógica do dispositivo é feita através de extensões de classes próprias na plataforma de execução. |
| **Conscientes de contexto** | Não. | Parcial. O MoCA oferece uma API cliente para acesso às informações sobre o estado dos dispositivos. | Sim. Permite aos agentes perceberem sua localização e o estado dos dispositivos que fazem parte do SMA. |
| **Personalizados** | Sim. | Sim. | Sim. |
| **Adaptativos** | Sim. | Sim. | Sim. |
| **Antecipatórios** | Sim. | Sim. | Sim. |

Figura 24 – Comparativo em relação aos aspectos de ubiquidade.

# 6 TRABALHOS RELACIONADOS

Existem muitas linguagens e plataformas para a implementação de agentes e sistemas multiagentes. Por exemplo, a plataforma **Jason** [BHW07], desenvolvida por Jomi Hübner e Rafael Bordini, implementa a semântica operacional de uma versão estendida da linguagem AgentSpeak(L) [Rao96]. Ela provê uma infraestrutura para o desenvolvimento de agentes BDI com uma série de funcionalidades customizáveis.

Em linhas gerais, um agente Jason consiste da especificação de um conjunto de crenças que formam sua base inicial de crenças, uma sequencia de objetivos iniciais, e um conjunto de planos. As crenças formam o componente informacional do agente. Os objetivos indicam os estados que o agente quer atingir (chamados de *achievement goals*), ou tentativas de recuperar informações da base de crenças (chamados de *test goals*). Os planos possuem um evento disparador, um contexto (conjunto de crenças que devem ser verdadeiras para um plano ser considerado aplicável) e uma sequencia de ações básicas (ou sub-objetivos) que o agente deve perseguir ou realizar quando um plano é executado. Ações internas podem ser usadas no corpo ou no contexto de um plano. Ações internas são ações definidas pelo desenvolvedor e executam internamente ao agente e são programadas em Java.

O Jason difere-se da linguagem AA em diversos aspectos. Primeiramente, o Jason utiliza o paradigma BDI, e é baseado no paradigma de programação em lógica. A linguagem AA, em contraste, provê suporte a implementação de estruturas relacionadas aos conceitos encontrados no FAML para representar a estrutura interna do agente, e o estilo de programação é fortemente inspirado na linguagem Java. Além disso, a linguagem AA combina conceitos e construções de agentes para a programação das estruturas internas do agente no mesmo nível de abstração dos conceitos e construções de computação ubíqua. Dessa forma, seus operadores para ubiquidade são tratados como elementos de primeira ordem. Em relação a portabilidade, ambas podem ser consideradas portáveis, pois tanto o interpretador do Jason quanto o ambiente de execução da AA são implementados em Java. Atualmente, o Jason possui mais ferramentas e documentação disponíveis, por se tratar de uma linguagem mais antiga que a AA.

O projeto **simpAL** [RS11] propõe uma linguagem de programação de propósito geral baseada em abstrações de agentes. A arquitetura do agente é uma simplificação do modelo *Belief-Desire-Intention*. Estruturalmente, uma crença é similar a variáveis em

linguagens imperativas. No simpAL, existem eventos referentes a mudanças no ambiente e troca de mensagens entre agentes.

Um programa em simpAL é composto por agentes concorrentes realizando atividades como membros de uma organização, trabalhando em um ambiente comum organizado em *workspaces,* como ilustrado na Figura 25. Além de agentes, o ambiente compartilhado é composto por recursos e ferramentas, representados através de artefatos.

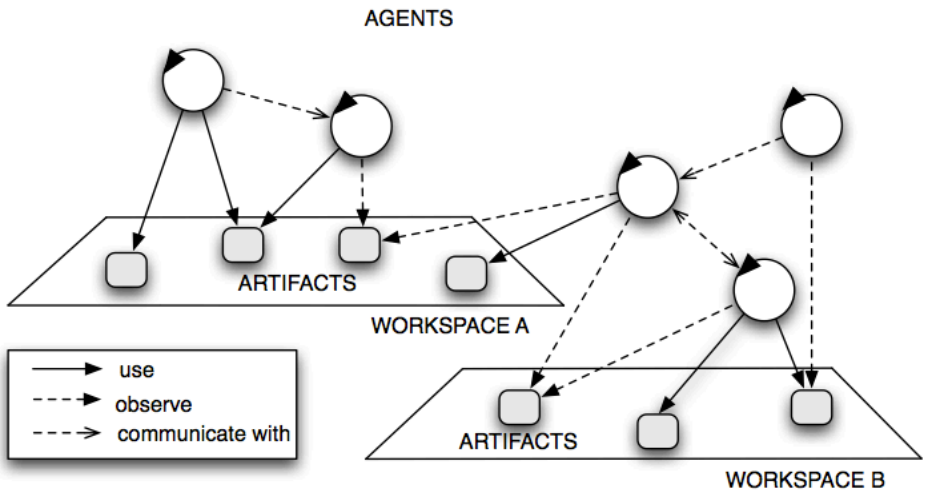


Figura 25 – Visão abstrata de um programa em simpAL [RS11].

Segundo os autores [RS11], são exemplos de artefatos: um quadro negro compartilhado, um relógio, ou até mesmo um banco de dados. Dessa forma, artefatos podem ser usados para representar qualquer entidade computacional que não precisa ser autônoma, mas que podem prover funcionalidades acessíveis pelos agentes.

A definição de artefato é similar à definição do um recurso no FAML, que posteriormente foi adaptada para a construção *resource* na linguagem AA. Na linguagem AA um recurso consiste de um conjunto de propriedades (leitura e escrita) que são dinamicamente definidas pelos agentes. Essas propriedades podem ser de qualquer tipo de dado suportado pela linguagem. Os recursos podem ser compartilhados (definidos no escopo do SMA) ou privados (definidos no escopo do agente). Diferente da linguagem AA, na simpAL os artefatos podem ser compostos em artefatos *complexos* através de conexão entre artefatos *simples* [RS11]. Um artefato pode possuir propriedades observáveis, referente ao seu estado dinâmico, que pode ser percebido pelos agentes. Para facilitar a implementação desse comportamento, a linguagem simpAL suporta a utilização do padrão de projeto *observer*. Esse mecanismo de observação é similar a forma como a linguagem AA permite que os agentes percebam alterações nos valores

das propriedades de um dispositivo, através de bloco `reacts` implementado no corpo do agente.

Em adição à AA, simpAL possui a noção de Tarefas e Papéis. As tarefas neste caso são usadas para decompor e modularizar a especificação de funções que o sistema deve realizar. Estas tarefas são explicitamente representadas e especificam o que determina o comportamento proativo e autônomo de um agente.

A noção de papel é usada para agrupar explicitamente um conjunto de tarefas, definindo então quais são as tarefas que um agente ao adotar uma determinado papel supostamente será capaz de realizar. O tipo de um agente é dado então pelo papel ou conjunto de papéis que ele realiza, independentemente do seu comportamento *concreto*.

A linguagem simpAL também inclui planos e ações. Um plano consiste de uma sequência de ações que o agente executa para cumprir uma determinada tarefa, ou simplesmente para definir um comportamento. Um plano também possui um contexto, que especifica uma condição (como sendo uma expressão booleana sobre a base de crenças) para que o plano possa ser utilizado. Uma ação pode ser tanto externa, que afeta o ambiente, ações de comunicação, pra enviar mensagens para outros agentes, ou interna, afetando o estado interno do agente.

Em relação a linguagem AA, os planos e ações da simpAL possuem diferenças estruturais. Na simpAL o plano define a sequência de ações que ele executa. Cada ação consiste de uma sentença simples, como por exemplo: de acesso a base de crenças, acesso a uma propriedade de um artefato, etc. Na AA, um plano encapsula um conjunto de ações sequenciais ou paralelas. Uma ação por sua vez especifica um conjunto de sentenças a ser executada, similarmente a um procedimento em linguagem estruturada.

O *framework* **Agent Factory** (AF) é uma coleção de ferramentas e plataformas para o desenvolvimento de sistemas baseados em agentes [RJO+11]. O AF foi desenvolvido sobre diversas camadas, onde no nível inicial é implementado o suporte a padrões FIPA para prover a interoperabilidade entre agentes. O AF possui componentes que podem ser adaptados e reutilizados quando necessário para permitir a prototipação de linguagens de programação orientadas a agentes, que são: um *framework* de lógica genérico com *forward chaining* e *backwards chaining*; um *framework* para planejamento e execução; uma interface padronizada entre os agentes e o ambiente; uma gramática e compilador implementados em JavaCC; e uma ferramenta de depuração que suporta a visualização de agentes e serviços.

Em relação ao trabalho proposto, AF não se concentra em uma única linguagem orientada a agentes, mas em permitir a criação de novas linguagens no topo de seu *framework*. Na medida que foi possível verificar, embora AF tenha sido usado para desenvolver aplicações pervasivas, nenhuma das linguagens derivadas suporta os conceitos de agentes e ubiquidade como elementos de primeira ordem. Por outro lado, AF poderia ser utilizada como uma das possíveis infraestruturas suportando a plataforma de execução da linguagem AA, agregando as funcionalidades mencionadas bem como cobrindo a implementação de noções básicas de ubiquidade tal como a localização física atual do agente.

O **MoCA** (*Mobile Collaboration Architecture*) é um *framework* para o desenvolvimento de aplicações colaborativas conscientes de contexto para dispositivos móveis [SER+04]. Ele é composto por APIs cliente e servidor, bem como serviços para o monitoramento e inferência do contexto de dispositivos. O MoCA possui um serviço chamado *monitor* que deve estar em execução nos dispositivos móveis. Ele coleta dados referentes ao dispositivo e ao estado da rede e envia estas informações para o *Context Information Service* (CIS), que pode estar executando em um ou mais nodos de uma rede sem fio. Os dados coletados incluem a qualidade do sinal *wireless*, nível restante de bateria, utilização da CPU, memória livre, *access point* atual, e o conjunto de *access point* com seus respectivos níveis de sinal que estão no alcance do dispositivo móvel.

O *Location Inference Service* (LIS) é responsável por inferir a localização geográfica aproximada de um dispositivo. Ele compara o sinal de rádio frequência atual, através de informações recuperadas do CIS, com padrões de sinais previamente medidos em pontos de referencia predefinidos de uma área.

O **PervML** [SVP10] é uma linguagem de modelagem para a representação de contexto em sistemas ubíquos em alto nível de abstração. Ela é uma linguagem específica de domínio para a especificação de sistemas ubíquos, utilizando primitivas conceituais (tais como serviços, *triggers*, interações, etc.). A linguagem oferece modelos para a descrição dos tipos de serviços que o sistema deve realizar, os diferentes serviços para cada tipo providos pelo sistema, como estes serviços interagem, e políticas de privacidade e dados pessoais de usuários. Um serviço é o principal elemento do metamodelo. Os serviços são utilizados para abstrair elementos do mundo real e funcionalidades que o sistema oferece, como por exemplo, a intensidade luminosa e a identificação da presença de pessoas em um ambiente são exemplos de serviços.

O PervML também oferece um diagrama para a modelagem de diferentes áreas por onde um usuário pode mover-se ou onde os serviços podem ser localizados. As áreas são representadas como pacotes, de um diagrama de pacotes UML. A hierarquia entre os pacotes representam hierarquias entre as áreas modeladas. Diferentemente da linguagem AA, o PervML mapeia relacionamentos entre regiões, através de dois tipos de associação: adjacência e mobilidade. A relação de adjacência indica que uma área está *próxima* de outra. A relação de mobilidade é um tipo inferido que representa a navegabilidade (ou caminho) entre as regiões adjacentes. Embora o PervML não seja uma linguagem de programação, ele possui ferramentas capazes de gerar esqueletos de código Java a partir dos modelos gerados.

Não foi encontrado na literatura uma linguagem de programação para a implementação de sistemas multiagentes ubíquos (em particular com abstrações em primeira ordem para ambos). A pesquisa foi realizada nas principais ferramentas de busca das bibliotecas digitais ACM *Digital Library*, *IEEEXplore* e *Science Direct*, utilizando uma *string* de busca padronizada e devidamente adaptada para cada mecanismo de busca a fim de evitar inconsistência nos resultados. As palavras-chave na *string* visam restringir o domínio de busca, consistindo de combinações de termos em alto-nível tais como: “pervasive” and “ubiquitous” “computing”, “agent” e “multiagent systems”, e “programming language”.

# 7 CONSIDERAÇÕES FINAIS E TRABALHOS FUTUROS

Esta tese apresentou a construção de uma linguagem de programação orientada a agentes para o desenvolvimento de sistemas multiagentes ubíquos, chamada de *Agents Anywhere* (AA). Essa linguagem é inspirada no metamodelo FAML, que unifica conceitos de agentes e sistemas multiagentes, e no metamodelo U-MAS, que estende o FAML introduzindo conceitos de computação ubíqua.

O objetivo da linguagem é permitir a implementação de sistemas multiagentes ubíquos, onde a ubiquidade é um conceito inerente aos agentes em execução no ambiente, e que seja independente de plataformas de SMA. A linguagem representa conceitos fundamentais de agentes, como objetivos, planos e ações, além de estruturas que podem ser compartilhadas, como recursos, que podem ser declarados em diferentes escopos dentro de um sistema multiagente. Também são representados conceitos importantes capazes de caracterizar um ambiente ubíquo que são espaços, dispositivos e localizações.

A noção de espaço permite ao desenvolvedor decompor um ambiente em regiões simbólicas, organizando os recursos e dispositivos disponíveis no ambiente. A linguagem oferece também operadores para o desenvolvedor programar agentes capazes de perceber alterações na localização bem como alterações nas propriedades de um determinado dispositivo no ambiente.

O ambiente de execução da linguagem permite que customizações sejam realizadas para permitir que programas escritos em AA possam executar em diferentes plataformas de hardware ou software. Além disso, ele também abstrai a implementação de sistemas de posicionamento e dispositivos. A linguagem AA permite que dispositivos e sistemas de posicionamento sejam descritos em função de suas propriedades, sem que sejam revelados detalhes de implementação. Sua implementação concreta deve ser realizada através da extensão de classes padrão da arquitetura do ambiente de execução. Dessa forma, novos sistemas de posicionamento e abstrações de dispositivos podem ser adicionadas conforme recursos suportados pela plataforma de software do dispositivo que executará o sistema multiagente.

A versão inicial do ambiente de execução utiliza o *framework* SemantiCore como base, devido a este possuir diversas funcionalidades já implementadas como controle de distribuição, comunicação entre agente, e elementos referentes a estrutura interna do agente. Além disso, e, visando facilitar a adoção da linguagem, sua sintaxe está sendo

inspirada na linguagem Java, que é uma linguagem amplamente utilizada na indústria e academia. Embora o SemantiCore permita que múltiplas instâncias de sua plataforma conectem-se formando um *domínio*, a linguagem apresentada ainda não trata da distribuição do ambiente em múltiplas instâncias de execução.

A fim de avaliar a linguagem proposta foram conduzidas três implementações diferentes. A primeira demonstrou a utilização do linguagem no desenvolvimento de uma aplicação com conceitos de ubiquidade e distribuição, demonstrando a utilização do principais construtores da linguagem. A segunda abordagem estendeu a linguagem Jason, verificando o esforço para a incorporação de conceitos de ubiquidade, e por fim, foi estendido o framework MoCA a fim de verificar o esforço na incorporação de conceitos de agentes. No caso do Jason, para representar os conceitos de ubiquidade novos predicados devem ser utilizados, porém, tais predicados não são palavras reservadas da linguagem, o que pode levar os desenvolvedores a terem problemas de contexto e semântica, no sentido de já utilizarem tais predicados em seus programas ao utilizar essa extensão. Estender o MoCA adicionando conceitos de agentes acarretou na criação de muitas classes para representar o núcleo mínimo.

Avaliar uma linguagem de programação é uma tarefa difícil. Os estudos conduzidos ajudam o desenvolvedor a comparar os níveis de abstração propostas sobre diferentes paradigmas, sendo o Jason o correspondente ao paradigma de programação em lógica e o MoCA sobre o paradigma orientado a objetos. É difícil afirmar qual abordagem é a melhor pois se tratando de uma linguagem de programação parte de sua efetividade depende do conhecimento sobre a linguagem, sobre boas práticas de programação e habilidades do programador.

O objetivo da linguagem AA  foi então propor uma nova forma de desenvolver aplicações ubíquas, tratando conceitos de agentes e ubiquidade como elementos de primeira ordem, em um mesmo nível de abstração.

Foram identificadas diversas melhorias e trabalhos futuros, organizadas em categorias, que são:

1. **Em relação à gramática da linguagem**:
    a. estender a linguagem para demais conceitos de agentes apresentados pelo metamodelo FAML; avaliar a incorporação de conceitos de Sociedade e Organização de agentes como operadores da linguagem;

   b. estender a linguagem para demais conceitos de computação ubíqua do metamodelo U-MAS, como por exemplo o conceito de contexto e informações contextuais;
   c. permitir a instanciação de múltiplos agentes referentes a um mesmo código-fonte;
   d. adição de operadores para tratamento de exceções;
   e. criação de elementos para descrever a distribuição do ambiente e onde os agentes executarão, inspirados no arquivo de configuração do SMA do Jason;
   f. adição de bibliotecas de funções essenciais como: acesso a arquivos, concorrência, acesso a banco de dados, etc.
2. **Em relação ao ambiente de execução**:
   a. características de segurança no ambiente de execução;
   b. implementação de persistência em banco de dados SQLite para os recursos do tipo *Persistent*;
   c. Customização do ambiente de execução para plataformas móveis atuais (Android, iOS e Windows Phone);
   d. incorporação de padrões FIPA para a interoperabilidade entre agentes, principalmente no formato de mensagens entre os agentes.
3. **Em relação à validação e divulgação da linguagem**:
   a. experimentos controlados a fim de avaliar qualitativamente a efetividade linguagem proposta;
   b. formalização computacional da linguagem;
   c. escrita de artigo científico para periódico.

# REFERÊNCIAS BIBLIOGRÁFICAS


[3APL11] 3APL WEB SITE. “3APL – An Abstract Agent Programming Language”. Capturado em: http://www.cs.uu.nl/3apl/, Maio 2012.

[AS99] Ark, W.; Selker, T. “A Look at Human Interaction with Pervasive Computers”. *IBM Systems Journal*, vol. 38-4, 1999, pp. 504-507.

[BC06] Barron, P.; Cahill, V. “YABS: a domain specific language for pervasive computing based on stigmergy”. In: Proceedings of the 5th international conference on Generative programming and component engineering (GPCE '06). ACM, New York, NY, USA, 2006, pp. 285-294.

[BCT+07] Bellifemine, F.; Caire, G.; Trucco, T.; Rimassa, G. “Jade Programmer’s Guide”. Relatório Técnico, Jade Web Site, 2007.

[BGP+02] Bernon, C.; Gleizes, P.; Peyruqueou, S.; and Picard, G. “ADELFE: a methodology for adaptive multi-agent systems engineering”. In: Proceedings of the 3rd international conference on Engineering societies in the agents world III (ESAW'02), Paolo Petta, Robert Tolksdorf, and Franco Zambonelli (Eds.). Springer-Verlag, Berlin, Heidelberg, 2002, pp. 156-169.

[BGZ04] Bergenti, F; Gleizes, M. P; Zambonelli, F (eds). “Methodologies and Software Engineering for Agent System: The Agent Oriented Software Engineering Handbook”. New York: Kluwer Academic Publisher, 2004, 536p.

[BHM+09] Beydoun, G.; Low, G.; Henderson-Sellers, B.; Mouratidis, H.; Gomez-Sanz, J. J.; Pavon, J.; Gonzalez-Perez, C. “FAML: A Generic Metamodel for MAS Development”. In: IEEE Transactions on Software Engineering, vol. 99, 2009, pp. 841-863.

[BEC07] Blois, M.; Escobar, M.; Choren, R. “Using Agents and Ontologies for Application Development on the Semantic Web”. *Journal of the Brazilian Computer Society*, vol. 1, 2007, pp. 1-15.

[BL04] Blois, M.; Lucena, C. “Multi-Agent Systems and the Semantic Web – The SemanticCore Agent-based Abstraction Layer”. In: Proceedings of Sixth International Conference on Enterprise Information Systems (ICEIS), Porto, 2004, pp. 263-270.

[BHW07] Bordini, R.; Hubner, J.; Wooldridge, M. “Programming Multi-Agent Systems in AgentSpeak Using Jason. John Wiley & Sons, 2007, 273p.

[BDD+05] Bordini, R. H.; Dastani, M.; Dix, J.; Seghrouchni, A. “Multi-Agent Programming: Languages, Platforms and Applications”. Springer, 2005, 296p.

[BP09] Braubach, L.; Pokahr, A. “A property-based approach for characterizing goals”. In: Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems - (AAMAS '09), Vol. 2. International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, 2009, pp. 1121-1122.

[BPM+04] Braubach, L.; Pokahr, A.; Moldt, D; Lamersdorf, W. “Goal representation for BDI agent systems”. In: Proceedings of the Second international conference on Programming Multi-Agent Systems (ProMAS'04), Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal Fallah Seghrouchni (Eds.). Springer-Verlag, Berlin, Heidelberg, 2004, pp. 44-65.

[BPG+04] Bresciane, P.; Perini, A.; Giorgini, P.; Giunchiglia, F.; Mylopoulos, J. “Tropos: An Agent-Oriented Software Development Methodology”. In: Autonomous Agents and Multi-Agent Sytems, 2004, pp. 203–236.

[CCS07] Campiolo, R.; Cremer, V.; Sobral, J. B. M. “On modeling for pervasive computing environments”. In: Proceedings of the 10th ACM Symposium on Modeling, analysis, and simulation of wireless and mobile systems (MSWiM '07). ACM, New York, NY, USA, 2007, pp. 240-243.

[CDL05] Cortese, G.; Davide, F.; Lunghi, M. “Context Awareness for Physical Service Environments”. *Ambient Intelligence*, IOS Press, 2005, vol. 6, pp. 71-97.

[DRM05] Dastani, M. M.; Riemsdijk, M. B.; Meyer, C. “Programming Multi-Agent Systems in 3APL”. *Multi-Agent Programming: Languages, Platforms and Applications*. 2005, pp. 39-67.

[ELB2006] Escobar, M.; Lemke, A.; Blois, M. “SemantiCore 2006 – Permitindo o Desenvolvimento de Aplicações baseadas em Agentes na Web Semântica.” In: Second Workshop on Engineering for Agent-oriented Systems, Florianópolis, Brazil, Outubro 2006, pp. 72-82.

[EB11] Escobar, M.; Blois, M. “U-MAS: Um Meta-modelo para o Desenvolvimento de Aplicações Multiagentes Ubíquas”. In: VII Simpósio Brasileiro de Sistemas de Informação, 2011, Salvador, pp. 310-321.

[Fer99] Ferber, J. “Multi-agent systems: an introduction to distributed artificial Intelligence”. Oxford: Addison-Wesley, 1999, 528p.

[FIPA13] FIPA ACL. “FIPA ACL Message Structure Specification”. Capturado em: http://www.fipa.org/specs/fipa00061/, Dezembro 2012.

[GZL+08] Gunasekera, K.; Zaslavsky, A.; Loke, S. W.; Krishnaswamy, S. “Context Driven Compositional Adaptation of Mobile Agents”. In: Ninth International Conference on Mobile Data Management Workshops (MDMW’ 08), 2008, pp. 201-208.

[HMN+01] Hansmann, U.; Merk, L.; Nicklous, M.; Stober, T. “Pervasive computing handbook”. Springer-Verlag New York, Inc., New York, NY, USA, 2001, 407p.

[HIR02] Henricksen, H.; Indulska, J.; Rakotonirainy, A. “Modeling context information in pervasive computing systems”. In: First International Conference on Pervasive Computing, 2002, pp. 167-180.

[IM04] Ilyas, M.; Mahgoub, I. “Mobile computing handbook”. CRC Press (1st edition), 2004, 1028p.

[JADE11] JADE. “Java Agent DEvelopment Framework”. Capturado em: http://jade.tilab.com/, Novembro 2011.

[JAY+05] Jansen, E.; Abdulrazak, B.; Yang, H.; King, J.; Helal, S. “A Programming Model for Pervasive Spaces”. In: Proceedings of 3rd International Conference on Service Oriented Computing, Amsterdam, Netherlands, 2005, pp. 12-15.

[JCC12] JavaCC. “JavaCC Grammars”. Capturado em: http://java.net/projects/javacc, Dezembro 2012.

[Jen01] Jennings, N. R. “An agent-based approach for building complex software systems”. *Communications of the ACM*, vol. 44-4, 2001, pp. 35-41.

[Kur07] Kurkovsky, S. “Pervasive computing: Past, present and future. Information and Communications Technology”. In: 5th International Conference on ITI,

2007, pp. 65-71.

[MPF04] Muñoz, J.; Pelechano, V.; Fons, J. “Model Driven Development of Pervasive Systems”. In: Intl. Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MOMPES), Canada: Hamilton, 2004, pp. 3-14.

[Oat06] Oates, B. “Researching Information Systems and Computing”. Sage Publications Ltd, 2006, 341p.

[OMG05] OMG - Object Management Group. “Agent Technology – the green paper – version 1.0”. Capturado em: http://www.jamesodell.com/ec2000-08-01.pdf, Agosto 2005.

[PW02] Padgham, L.; Winikoff, M. “Prometheus: A Methodology for Developing Intelligent Agent”. In: 3th International Workshop on Agent Oriented Software Engineering, at Autonomous Agents and Multi-Agent Systems (AAMAS), 2002, 12p.

[PW04] Padgham, L.; Winikoff, M. “Developing Intelligent Agent Systems: A Practical Guide”. J. Wiley & Sons, 2004, vol. 1, 240p.

[Rao96] Rao. A. “Agentspeak(l): Bdi agents speak out in a logical computable language”. In: Proceedings of the 7th European workshop on Modelling autonomous agents in a multi-agent world : agents breaking away: agents breaking away, MAAMAW ’96, pp. 42–55, Secaucus, NJ, USA, 1996. Springer-Verlag New York, Inc.

[Rib02] Ribeiro, M. B. “Web Life: Uma arquitetura para a implementação de sistemas multi-agentes para a Web”. Tese de Doutorado, Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, 2002, 204p.

[RS11] Ricci, A.; Santi, A. “Designing a general-purpose programming language based on agent-oriented abstractions: the simpal project”. In: Proceedings of the compilation of the co-located workshops on DSM’11, TMC’11, AGERE!’11, AOOPES’11, NEAT’11, & VMIL’11, SPLASH ’11 Workshops, pp. 159–170, New York, NY, USA, 2011. ACM.

[RCR+02] Román, M.; Hess, C.; Cerqueira, R.; Ranganathan, A.; Campbell, R.; Nahrstedt, K. “A Middleware Infrastructure for Active Spaces”. *IEEE Pervasive Computing*, vol. 1, 4, October 2002, pp. 74-83.

[RJO+11] Russell, S.; Jordan, H.; O’Hare, G.; Collier. R. “Agent factory: A framework for prototyping logic-based AOP languages”. In: Proceedings of the 9th German conference on Multiagent system technologies (MATES'11), Franziska Klügl and Sascha Ossowski (Eds.). Springer-Verlag, Berlin, Heidelberg, 2011, pp. 125-136.

[SER+04] Sacramento, V.; Endler, M.; Rubinsztejn, H. K.; Lima, L. S.; Goncalves, K.; Nascimento, F. N.; Bueno. G. A. “Moca: A middleware for developing collaborative applications for mobile users”. *IEEE Distributed Systems Online*, vol. 5-10, Oct. 2004.

[SM03] Saha, D.; Mukherjee, A. “Pervasive Computing: A Paradigm for the 21st Century”. *Computer* 36, Março 2003, pp. 25-31.

[Sat01] Satyanarayanan, M. “Pervasive computing: Vision and challenges”. *IEEE Personal Communications*, vol. 8, 2001, pp. 10-17.

[Seb09] Sebesta, Robert W. “Concepts of Programming Languages 9th ed.”. Addison-

Wesley Publishing Company. USA, 2009, 696p.

[SVP10] Serral, E.; Valderas, P.; Pelechano, V. “Towards the Model Driven Development of context aware pervasive systems”. In: Pervasive Mob.Comput, 2010, pp. 254-280.

[STB07] Seyler, F.; Taconet, C.; Bernard, G. “Context Aware Orchestration Meta-Model”. In: Proceedings of the Third International Conference on Autonomic and Autonomous Systems (ICAS '07), IEEE Computer Society, Washington, DC, USA, 2007.

[Sym10] Symonds, J. “Ubiquitous and Pervasive Computing: Concepts, Methodologies, Tools, and Applications”. New Zealand: Auckland University of Technology, 2010, 1962p.

[Wei99a] Weiser. M. “The computer for the 21st century”. *SIGMOBILE Mob. Comput. Commun*, vol. 3-3, 1999, pp. 3-11.

[Wei99b] Weiss, G. “Multiagent Systems: a Modern Approach to Distributed Artificial Intelligence”. MIT Press, 1999, 643p.

[WJK99] Wooldridge, M.; Jennings, N.; Kinny, D. “A methodology for agent oriented analysis and design”. In: Proceedings of International Conference on Autonomous Agents, Seattle, EUA, 1999, pp. 69-76.

[WJK00] Wooldridge, M.; Jennings, N. R.; Kinny, D. “The Gaia methodology for agent-oriented analysis and design”. *Journal of Autonomous Agent and Multi-Agent Systems*, vol. 3, Set 2000, pp. 285-312.

# APÊNDICE A - Código completo da aplicação *UbiCampus* utilizando a linguagem AA

```
MAS UbiCampus {

positioningsystem GPS {
double      latitude,  longitude;
int   radius;
}

positioningsystem Network {
string apNane;
}

space Classroom {
location l of Network {
apName = "Classroom";
}
}

space Library {
location l of Network {
apName = "Library";
}
}

space LectureHall {
location loc of Network {
apName = "LectureHall";
}
}

agent UserAgent {
goal lookForColleagues;

reacts {
on locationchanged l {
if(l.type.equals("entered"))
achieve lookForColleagues;
}
}

plan lookupPlan for lookForColleagues {
first action lookupColleagues {
next lookupLocations;

perform {
String username = "Mauricio";

print("Searching...");

send("EnrollmentServiceAgent", "Colleagues", username);

} // fim perform
} // fim action

action lookupLocations {
perform {
print("Lookup locations");
```

```
Message msg = nextMessage();

string[] colleagues = (string []) msg.content;

send("LocationServiceAgent", "Locations", colleagues);

msg = nextMessage();

string[][] locations = (string[][]) msg.content;

string myLocation = Network.apName;

boolean found = false;

for(string [] ul : locations) {
// ul[0] -> username
// ul[1] -> location

if(ul[0] != null) {
if(ul[1].equals(myLocation)) {
found = true;
alert(ul[0] + " is nearby.");
}
}
}

if(!found)
alert("Sorry, there are no friends nearby.");
}
} // fim acao
} // fim plan
}

//=================================================================
// ENROLLMENT AGENT
agent EnrollmentServiceAgent {
goal provideClassInfo;
resource base;

init {

string [] classA = { "Mauricio", "Rodrigo", "Anderson" };
string [] classB = { "Rafael", "Ana", "Ricardo", "Marcelo" };


base.set("classA", classA);
base.set("classB", classB);
}

reacts {
on messagereceived m {
if(m.subject.contains("Colleagues"))
achieve provideClassInfo;
}
}

plan retrieveStudentList for provideClassInfo {
first action logic {
perform {

Message msg = nextMessage();

string username = (String) msg.content;
```

```
string [] colleagues = null;

String [] c1 = (String[]) base.get("classA");
String [] c2 = (String[]) base.get("classB");

for(string classUser : c1){
if(classUser.equals(username)) {
colleagues = c1;
break;
}
}

send(msg.from, "Colleagues", colleagues);
} // perform
} // action
}
}

//================================================================
// LOCATION AGENT
agent LocationServiceAgent {

goal provideLocationInfo;
goal updateLocation;
resource base;

init {
string [][] l = { { "Rodrigo", "PORTAL" }, { "Anderson", "ULBRA" } };

base.set("locations", l);
}

reacts {
on messagereceived m {

if(m.subject.equals("Locations"))
achieve provideLocationInfo;

else if(m.subject.equals("UpdateLocation"))
achieve updateLocation;
}
}

plan retrieveLocations for provideLocationInfo {
first action a {
perform {
Message m = nextMessage();

string[][] userLocations =
(string[][]) base.get("locations");

// logica
int cont = 0;

string [][] reply = new string [10][2];

string [ ] users = (string []) m.content;

for(string user : users) {
for(string [ ] ul : userLocations) {
if(user.equals(ul[0])) {
        reply[cont] = ul;
```

```
        cont++;
}
}
}

send(m.from, reply);
}
} // action
} // plan

plan updateLocationPlan for updateLocation {
first action a {
perform {
Message request = nextMessage();

print(request.content);
}
}
}
}
}
```

# APÊNDICE B - Código completo da aplicação *UbiCampus* utilizando a extensão do Jason

**ubicampus.mas2j**

```
MAS ubicampus {

    infrastructure: Centralised

    environment: UbiCampus

    agents:
                  mauricio
                        agentArchClass
                        jason.ubiquitous.UbiquitousAgArch;

            campusManager;
            secretary;
}
```

**User Agent**

```
// Ubiquitous event
+location(Type, Space)
      <-
            .send(campusManager, achieve, locationUpdate(mauricio, Type,
Space)).


+!nearby(Students) : .member(mauricio, Students)
      <-
            .send(secretary, askOne, classmates(Students, _), classmates(_,C));
            .delete(mauricio, Students, P);
            .print(" > People neaby ", P);
            .member(mauricio, C);
            .delete(mauricio, C, L);
            .print(" > Neaby classmates ", L).
```

**Secretary Agent**

```
class([ana, rodrigo, anderson, mauricio]).

+?find (Students)
      <-
            ?class(Class);
            .intersection(Class, Students, Classmates).
            // .print("Colleagues: ", Classmates).


classmates(Students, C) :- class(Class) & .intersection(Class, Students, C).
```

**Campus Manager Agent**

```
located(anderson, facin).
located(rafael, facin).
located(ana, ifrs).
located(marcelo, rio).

prox(Place, People) :- .findall(P,located(P,Place),People).

+! locationUpdate (Username, Type, Location) [ source(Sender) ] : Type ==
entered
      <-
            +located(Username, Location);
            ?prox(Location,R);
            .send(R, achieve, nearby(R)).

+! locationUpdate (Username, Type, Location) [ source(Sender) ] : Type == exited
      <-
            -located(Username, Location).
```

# APÊNDICE C - Código completo da aplicação *UbiCampus* utilizando a extensão do *framework* MoCA

**UbiCampus MAS**

```
import br.pucrs.core.MAS;
import br.pucrs.ubiquitous.UbiquitousEnvironment;

public class UbiCampus extends MAS {

public UbiCampus() {
this.env = new UbiquitousEnvironment();

UserAgent ua = new UserAgent("mauricio");
SecretaryAgent sa = new SecretaryAgent("Secretary");
CampusManagerAgent cma = new CampusManagerAgent("CampusManager");

env.addAgent(cma);
env.addAgent(sa);
env.addAgent(ua);
}

public static void main(String[] args) {
new UbiCampus();
}
}
```

**User Agent**

```
import br.pucrs.core.Goal;
import br.pucrs.core.Plan;
import br.pucrs.ubiquitous.Space;
import br.pucrs.ubiquitous.UbiquitousAgent;

public class UserAgent extends UbiquitousAgent {
private Goal lookForColleagues;
private Plan lookForColleaguesPlan;

public UserAgent(String name) {
super(name);
}

public void setup() {
lookForColleaguesPlan = new Plan();
lookForColleaguesPlan.add(new LookupForColleagues());

lookForColleagues = new Goal(lookForColleaguesPlan);
}

@Override
public void onLocationEntered(Space e) {
lookForColleagues.start();
}
}


import java.util.ArrayList;
import java.util.HashMap;
```

```
import br.pucrs.core.Action;
import br.pucrs.core.Message;

public class LookupForColleagues extends Action {
@Override
public void run() {
Message request = new Message(this.getName(), "Secretary", "Request",
        "Colleagues");

send(request);

Message reply = receive();

if (reply.getContent() != null) {
ArrayList<String> l = (ArrayList<String>) reply.getContent();

l.remove(getName()); // remove o proprio usuario

request = new Message(this.getName(), "CampusManager", "Request", l);

send(request);

reply = receive();

HashMap<String, String> map = (HashMap<String, String>) reply
                .getContent();

for (String key : map.keySet()) {
if (map.get(key).equals(e.identifier))
System.out.println(key + " is NEAR");
}
}
}
}
```

**Secretary Agent**

```
package br.pucrs.ubicampus;

import java.util.ArrayList;

import br.pucrs.core.Agent;
import br.pucrs.core.Message;

public class SecretaryAgent extends Agent {

protected ArrayList<String> classA;

public SecretaryAgent(String name) {
super(name);
classA = new ArrayList<String>();
classA.add("ana");
classA.add("rodrigo");
classA.add("anderson");
classA.add("mauricio");
}

@Override
protected void sense() {
Message request = receive();
System.out.println(getName() + " : " + request.getContent());
```

```
Object result = null;
if (classA.contains(request.getFrom()))
result = classA;

Message reply = new Message(getName(), request.getFrom(), "Response",
result);

send(reply);
}
}
```

**Campus Manager Agent**

```
package br.pucrs.ubicampus;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;

import br.pucrs.core.Agent;
import br.pucrs.core.Message;

public class CampusManagerAgent extends Agent {

protected Map<String, String> map = new HashMap<String, String>();

public CampusManagerAgent(String name) {
super(name);
map.put("anderson", "facin");
map.put("rafael", "facin");
map.put("ana", "ifrs");
map.put("marcelo", "rio");
}

@Override
protected void sense() {
Message request = receive();

ArrayList<String> lista = (ArrayList<String>) request.getContent();

Map<String, String> mapResp = new HashMap<String, String>();

for (String usuario : lista) {
String localizacao = map.get(usuario);
if (localizacao != null) {
mapResp.put(usuario, localizacao);
}
}

Message reply = new Message(getName(), request.getFrom(), "Locations",
mapResp);
send(reply);
}
}
```

# ANEXO A – Metamodelo FAML

A Tabela A.1 mostra as definições dos conceitos em tempo de projeto utilizados no metamodelo FAML e a Tabela A.2 as definições dos conceitos em tempo de execução também utilizados no metamodelo.

As Figuras A.1-4 apresentam, respectivamente, as classes externas ao agente em tempo de projeto, as classes internas ao agente em tempo de projeto, as classes externas ao agente em tempo de execução e as classes internas ao agente em tempo de execução conforme o metamodelo FAML [Beydoun 2009]. Todas as definições e diagramas são uma cópia fiel do apresentado em [Beydoun 2009].

**Tabela A.1.** Conceitos em tempo de projeto e suas definições de acordo com [Beydoun 2009].

| Conceito | Definição |
|---|---|
| Action Specification | Specification of an action, including any preconditions and post-conditions. |
| Agent Definition | Specification of the initial state of an agent just after it is created. |
| Environment Statement | A Boolean statement about the environment. |
| Facet Action Specification | Specification of a facet action in terms of the facet definition it will change and the new value it will write to the facet. |
| Facet Definition | Specification of the structure of a given facet, including its name, data type and Access mode. |
| Functional Requirement | Requirement that provides added value to the users of the system. |
| Interaction Protocol | Specification of patterns of communications that occurs in the system. |
| Mental State Specification | Specification of the initial mental state in terms of specified beliefs and agent goals. |
| Message Action Specification | Specification of a message action in terms of the message schema and parameters to use. |
| Message Schema | Specification of the structure and semantics of a given kind of messages that can occur within the system. |
| Non-functional Requirement | Requirement about any limits, constraints, or impositions on the system to be built. |
| Ontology | Structural model of a given domain. |
| Organization Definition | Specification of a collection of roles and agents co-operating towards a system goal. |
| Plan Resource Specification | This is a specification of resources that are used in the Plan Specification. |
| Plan Specification | An organized collection of action specifications. |
| Policy | A rule that specifies an arrangement of events expected to occur in a given environment. |
| Requirement | Feature that a system must implement. |
| Resource Specification | A resource specification specifies something that has a name, may have reasonable representations and that can acquired, shared or produced. |
| Role | Specification of a behavioral pattern expected from some agents in a given system. |
| Role Compatibility | Role relationship in which the source role is incompatible with the destination role for a given purpose. |
| Role Dependency | Role relationship in which the source role depends on the destination role for a given purpose. |
| Role Relationship | Social relationship between two roles for a given purpose. |
| Service | A single, coherent block of activity in which an agent may engage. |
| System | Final product of an agent-oriented software development project. |
| System Goal | A specification of a state of the environment that the system tries to achieve. |
| Task | Specification of a piece of behavior that the system can perform. |

**Tabela A.2.** Conceitos em tempo de execução e suas definições de acordo com [Beydoun 2009].

| Conceito | Definição |
|---|---|
| Action | Fundamental unit of agent behaviour. |
| Agent | A highly autonomous, situated, directed and rational entity. |
| Agent Goal | An environment statement which represents a state pursued by an agent. |
| Belief | An environment statement held by an agent and deemed as true in a certain timeframe. |
| Communication | Composition of more than one message. |
| Environment | The world in which an agent is situated. |
| Environment History | The sequence of events that have occurred between the environment start-up and any given instant. |
| Environment Statement | A statement about the environment. |
| Event | Occurrence of something that changes the environment history. |
| Facet | Property of the environment with which agents can interact. |
| Facet Action | Action that results in the change of a given facet. |
| Facet Event | Event that happens when the value of a facet changes. |
| Mental State | Agent goals and beliefs held by an agent at a certain timeframe. |
| Message | Unit of communication between agents, which conforms to a specific message schema. |
| Message Action | Action that results in a message being sent. |
| Message Event | Event that happens when a message is sent. |
| Organization | A collection of agents with specified roles co-operating towards a system goal. |
| Plan | An organized collection of actions that can be executed to pursue a particular agent goal. |
| Resource | Something that has a name, may have reasonable representations and that can acquired, shared or produced. |
| Role | Specification of a behavioural pattern expected from some agents in a given system. |
| System | Final product of an agent-oriented software development project. |

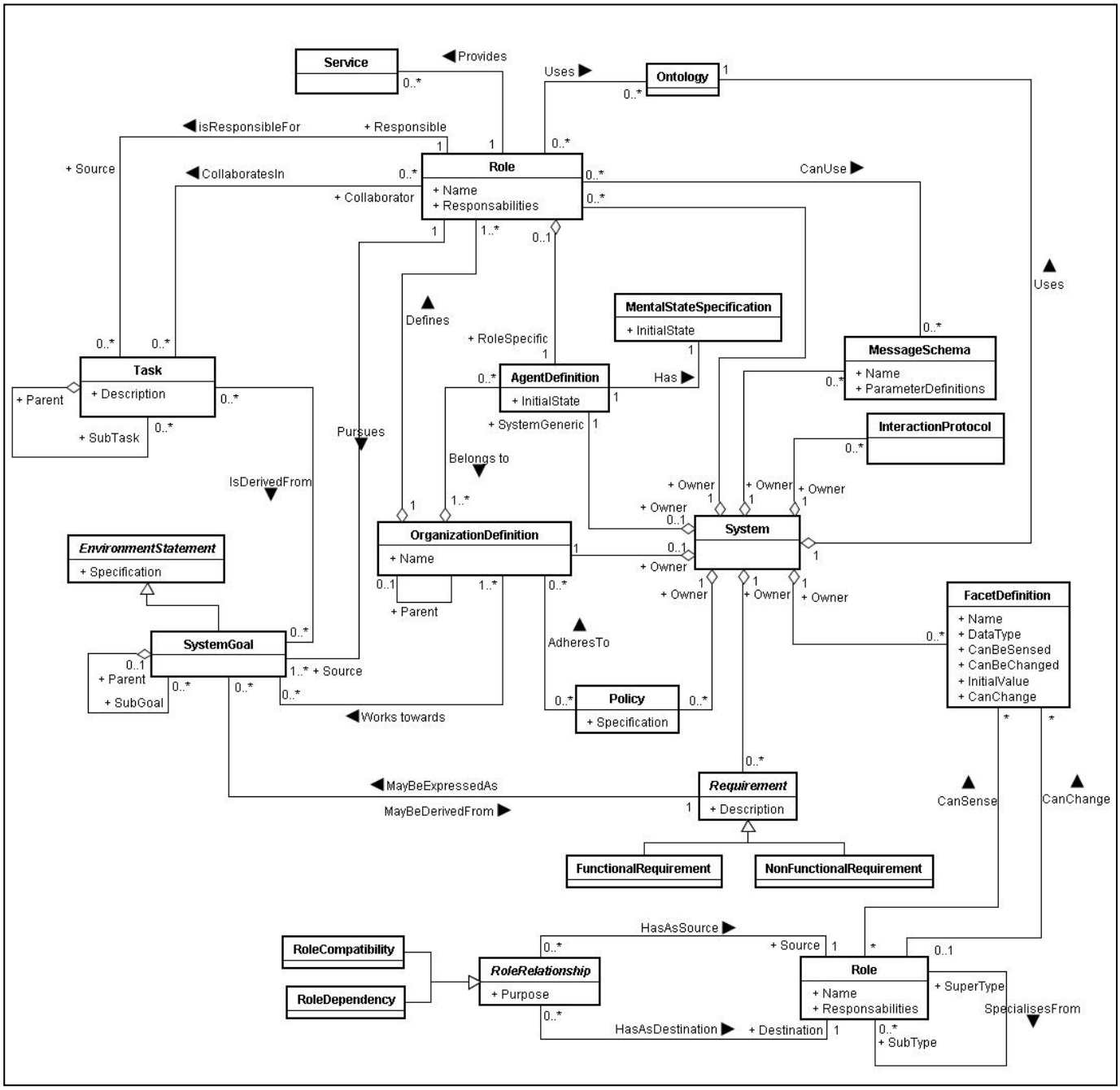


Figura A.1 - Classes externas ao agente em tempo de projeto segundo [Beydoun 2009].

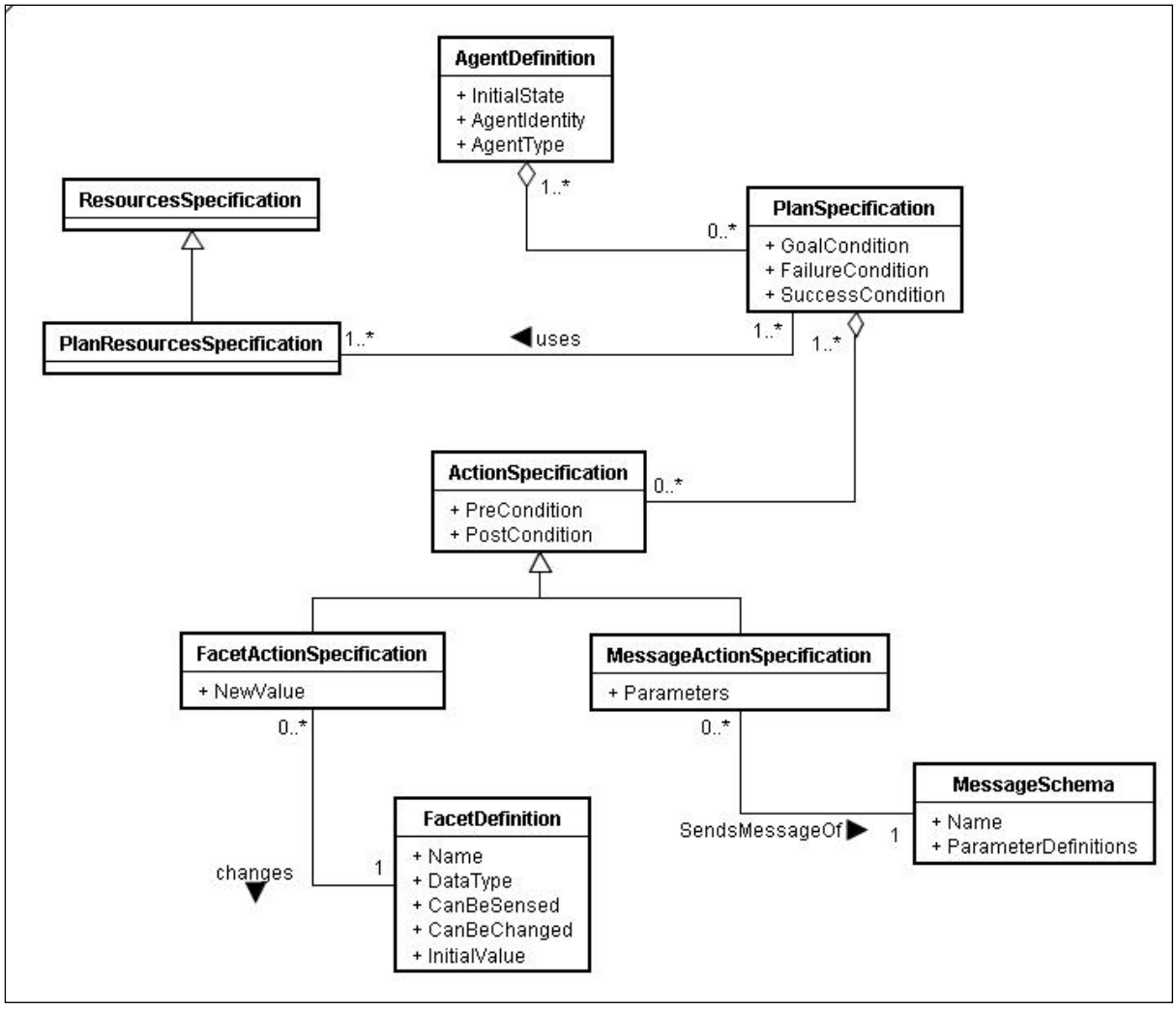


Figura A.2 – Classes internas ao agente em tempo de projeto segundo [Beydoun 2009].

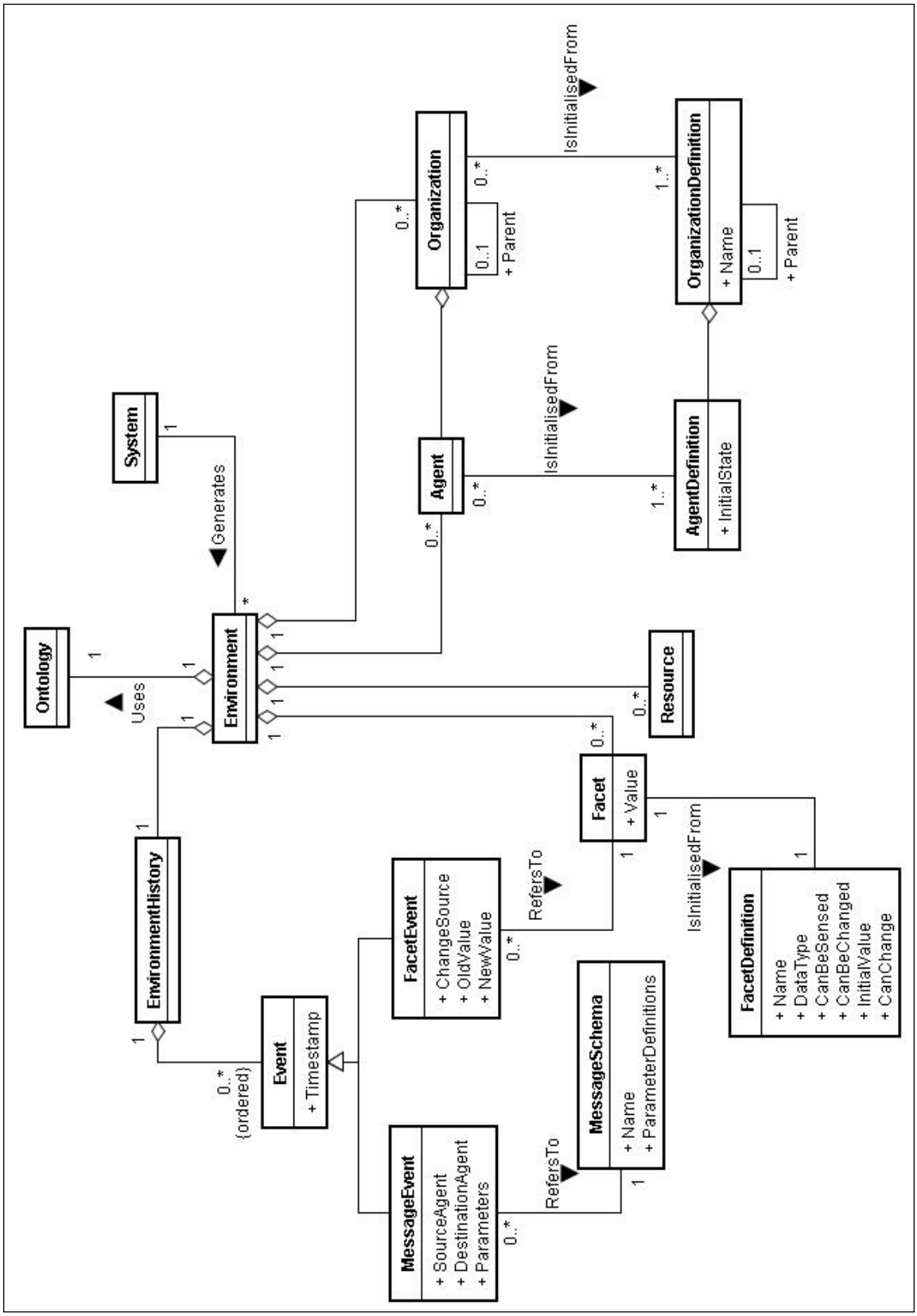


Figura A.3- Classes externas ao agente em tempo de execução segundo [Beydoun 2009].

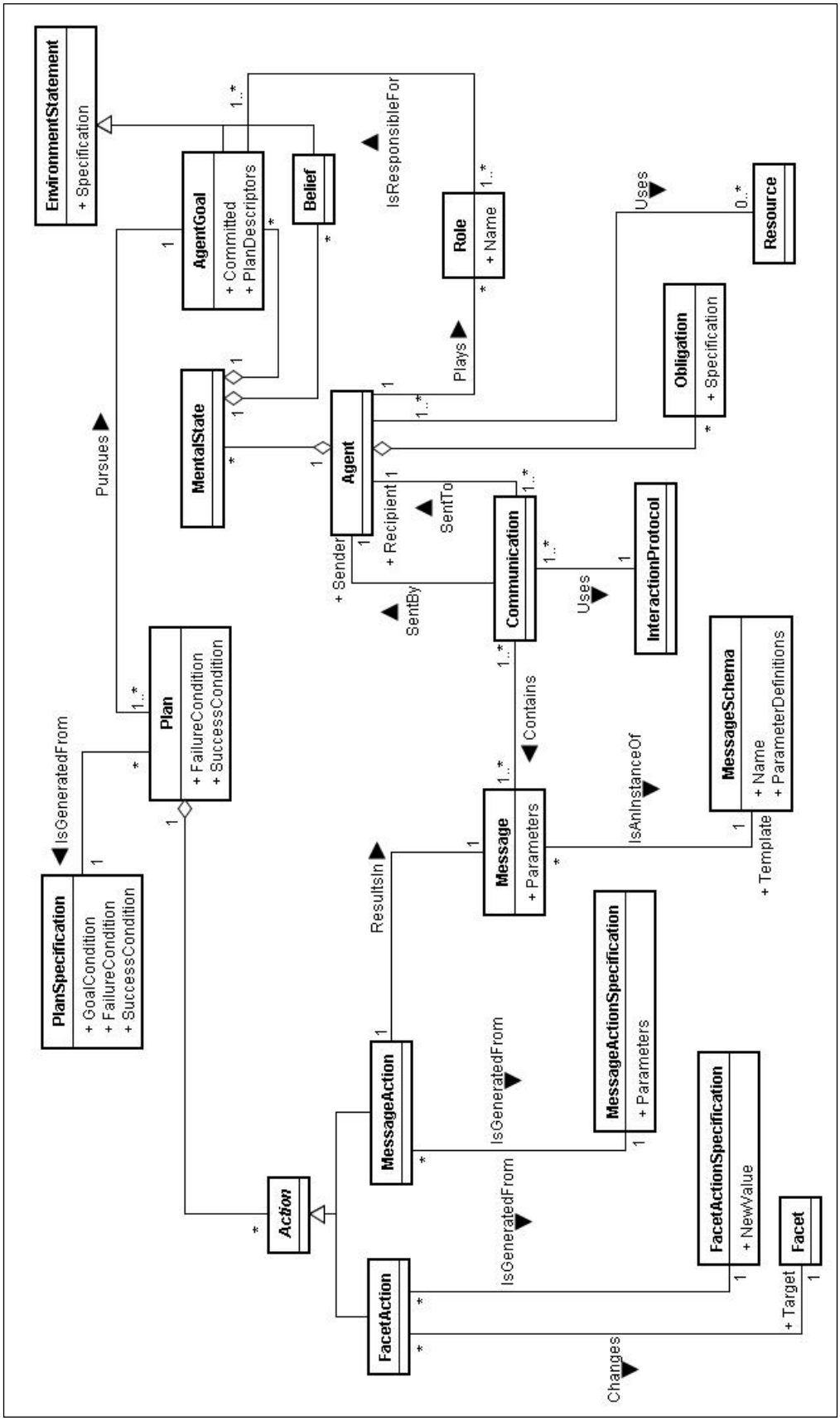


Figura A.4- Classes internas ao agente em tempo de execução segundo [Beydoun 2009].