**INSTITUTO MILITAR DE ENGENHARIA**

**CARLOS ALBERTO PADILHA PINHEIRO**

# VEÍCULOS AÉREOS AUTÔNOMOS NÃO TRIPULADOS PARA MONITORAMENTO DE AMBIENTES DESESTRUTURADOS E COMUNICAÇÃO DE DADOS

Dissertação de Mestrado apresentada ao Curso de Mestrado em Sistemas e Computação do Instituto Militar de Engenharia, como requisito parcial para obtenção do título de Mestre em Sistemas e Computação.

Orientador: Maj. Paulo César Pellanda, Dr. ENSAE.
Co-orientador: Prof. Paulo Fernando Ferreira Rosa, Ph.D.

Rio de Janeiro
2006

INSTITUTO MILITAR DE ENGENHARIA
Praça General Tibúrcio, 80 – Praia Vermelha
Rio de Janeiro - RJ CEP: 22290-270



P654 Pinheiro, Carlos A. P
Veículos Aéreos Não Tripulados para Monitoramento de Ambientes Desestruturados e Comunicação de Dados / Carlos Alberto Padilha Pinheiro. –

Rio de Janeiro : Instituto Militar de Engenharia, 2006.
xxx p.: il, graf., tab.

Dissertação (mestrado) – Instituto Militar de Engenharia – Rio de Janeiro, 2006

1. Dirigível. 2. Sistemas Multiagentes 3. JADE. I. Pinheiro, Carlos A. P. II. Instituto Militar de Engenharia. III. Título.

CDD 629.8312

**INSTITUTO MILITAR DE ENGENHARIA**

**CARLOS ALBERTO PADILHA PINHEIRO**

# VEÍCULOS AÉREOS AUTÔNOMOS NÃO TRIPULADOS PARA MONITORAMENTO DE AMBIENTES DESESTRUTURADOS E COMUNICAÇÃO DE DADOS

Dissertação de Mestrado apresentada ao Curso de Mestrado em Sistemas e Computação do Instituto Militar de Engenharia, como requisito parcial para obtenção do título de Mestre em Sistemas e Computação.

Orientador: Maj. Paulo César Pellanda, Dr. ENSAE.

Co-orientador: Prof. Paulo F. F. Rosa, Ph.D.

Aprovada em 3 de março de 2006 pela seguinte Banca Examinadora:

_____

Prof. Paulo F. F. Rosa – Ph.D. do IME - Presidente

_____

Maj. Paulo César Pellanda – Dr. ENSAE do IME

_____

Prof. Ronaldo Ribeiro Goldschmidt – D.Sc. da PUC

_____

Prof. Ricardo Choren Noya – D.Sc. do IME

Rio de Janeiro
2006

Às minhas avós, Priscilla Ramalho Pinheiro (*in memoriam*) e Emília Padilha Costa (*in memoriam*).

## AGRADECIMENTOS

Agradeço aos meus pais, Carlos Ramalho e Vilma Padilha;

Aos meus professores orientadores, Prof. Paulo Fernando Ferreira Rosa e Maj. Paulo César Pellanda, pelo apoio nos momentos difíceis;

Aos professores, Cel. Ney Bruno e Profa. Maria Thereza Miranda Rocco Giraldi, por desenvolverem em mim o gosto pela pesquisa; e

Aos meus colegas, Apolinário, Jacy e Alexandre, pela ajuda ao longo dessa caminhada.

Agradeço também à Seção de Engenharia de Sistemas e à Prof. Wilma de Araújo Gonzalez, por viabilizarem este trabalho.

*Carlos Alberto Padilha Pinheiro.*

“O homem é o animal que usa ferramentas. Sem elas, não é nada; com elas, é tudo.”
(CARLYLE).

"É depois do inverno mais frio que se vê que o pinheiro e o cedro sobreviveram à degeneração."
(CONFÚCIO).

**SUMÁRIO**

## LISTA DE ILUSTRAÇÕES

# RESUMO

Veículos aéreos autônomos não tripulados, tomados individualmente ou organizados em frota, têm ampla aplicação potencial e grande complexidade. O paradigma de sistemas multiagentes é um dos instrumentos para o seu tratamento, pois lida com complexidade e com autonomia de agentes. Além disto, sistemas multiagentes podem modelar sistemas multirobôs, e a frota de veículos aéreos é um sistema multirobô. Tal modelagem facilita a condução de testes de hipóteses e de soluções de projeto do sistema multirobô representado, tanto pela redução acentuada dos custos dos procedimentos, quanto pela presteza com que os testes são preparados e os respectivos resultados obtidos. Testes efetuados com os próprios robôs são demorados, caros e arriscados. Os que usam modelos físicos de robôs são geralmente de custo também elevado. Os sistemas multiagentes constituídos por agentes de *software* são, assim, alternativa competitiva como instrumento de projeto e desenvolvimento de sistemas multirobôs.

Esta dissertação trata da modelagem de um simulador de frota de veículos aéreos autônomos não tripulados. A frota deve operar em ambiente desestruturado e oferecer serviços de telecomunicações e de monitoramento do ambiente. O modelo do simulador visa a construção de um sistema multiagente composto de agentes de *software*. Além da modelagem em si, a dissertação trata da avaliação do modelo produzido, mostrando o desenvolvimento de subsistemas do simulador a partir do modelo.

**ABSTRACT**

# 1 INTRODUÇÃO

Em sentido amplo, esta dissertação trata de veículos aéreos autônomos não tripulados organizados segundo uma frota, portanto, segundo um sistema. A frota e os veículos que a compõem são de interesse, tanto por causa de sua utilidade, quanto da tecnologia que lhes é subjacente.

Os veículos aéreos autônomos não tripulados vêm tendo uso crescente nos últimos anos. Monitoramento, exploração, vigilância e transporte estão entre os serviços que eles podem prestar (GOMES, 1998). Comunicação de dados é outro uso potencial. No caso brasileiro, com grandes áreas despovoadas e com duzentas milhas marítimas para vigiar ao longo de extensa costa, dispositivos deste tipo podem ser muito úteis.

Os veículos desta classe têm sido modelados com o uso de agentes (HUFF, 2003). Mais especificamente, estes veículos têm sido modelados como sistemas compostos de agentes – chamados de sistemas multiagentes. Dentre as razões para tal, estão as seguintes: os problemas relativos aos veículos são inerentemente distribuídos, e os agentes podem ser modelados e distribuídos de acordo com a distribuição do problema; problemas muito complexos podem ser melhor atacados se decompostos (REIS, 2003), e a engenharia da frota de veículos é um problema muito complexo: cada veículo é um aglomerado complexo de agentes, e a frota é composta de vários veículos organizados de forma não trivial. A abordagem multiagente e a tecnologia a ela associada são um novo e promissor enfoque para o tratamento de uma extensa gama de situações, pois promete a solução de problemas complexos nas áreas de informática e robótica, as quais, por sua vez, têm aplicação em todas as áreas. Sinal das perspectivas alentadoras do paradigma multiagente é o grande volume de trabalhos que vem sendo produzidos sobre o assunto, resultando no surgimento de novas técnicas e na descoberta de novas aplicações.

O tema do qual esta dissertação trata no plano geral tem, então, aplicações relevantes e embute promissora tecnologia emergente, o que dá ao assunto interesse adicional.

No sentido estrito, a dissertação trata da modelagem de um simulador de uma frota de veículos aéreos autônomos não tripulados. Enquanto a frota de dirigíveis pode, com mais especificidade, ser caracterizada como um sistema multi-robô, o simulador é um sistema multiagente, constituído de agentes de *software*. A dissertação, *stricto sensu*, trata, então, da

modelagem do simulador segundo o paradigma multiagente, visando à construção de um sistema multiagente composto de agentes de *software*.

O desenvolvimento da frota de veículos tende a ser caro e demorado. O veiculo em si é de custo elevado e os dispositivos que podem lhe dar autonomia, também. O desenvolvimento envolve conhecimentos de pelo menos quatro especialidades: informática, aeronáutica, mecânica e aviônica. Esta diversidade de especialistas também significa despesa elevada. Testes mal sucedidos podem destruir o veículo, ou provocar acidentes. São necessárias grandes áreas livres para os testes de vôo, o que tem um custo associado. Há necessidade de hangar, para acomodar o veículo. Além de caras, muitas destas circunstâncias desdobram-se em retardos, como, por exemplo, o tempo gasto para recuperar o veículo após cair.

Uma das maneiras de contornar tais despesas é o uso da simulação física. O uso de modelos em escala, possivelmente necessário, tem também custo relativamente alto, pois, além dos modelos de veículo, tem de ser criado o modelo do ambiente, em escala compatível com a do modelo do veículo, com sensores, fontes de vento, obstáculos, correntes ascendentes, etc. – as variáveis de estado do ambiente – sob o completo domínio do pesquisador. Além disto, o volume do ambiente simulado é provavelmente muito grande, o que o encarece mais ainda. A simulação física corta custos e acelera o desenvolvimento, mas parece ser relativamente cara.

Alternativa de simulação é a baseada em *software.* De menor custo, permite o teste de muitas circunstâncias. Ela embute modelos dinâmicos de veículos que podem ser substituídos no simulador com relativa facilidade. Há na literatura diversos modelos dinâmicos de veículos aéreos. Trocando-se o modelo, troca-se o veículo a testar. A simulação baseada em *software* só requer computadores, cujos custos são relativamente baixos. O modelo dinâmico do ambiente talvez precise ser construído integralmente, e este é um custo a considerar. O simulador da frota é um simulador baseado em *software* que se configura como ferramenta relevante para o desenvolvimento da frota, tanto pelo custo relativamente baixo, quanto pela flexibilidade que oferece.

No caminho da modelagem do simulador, são levantadas propriedades dos veículos e da frota – o que é inevitável, afinal para modelar o simulador é necessário modelar o objeto a ser simulado – e, também por isto, como inicialmente mencionado, o escopo da dissertação transcende a modelagem do simulador.

Foram feitos, ainda, estudos buscando indícios do grau de adequação do modelo gerado e, em conseqüência, da técnica usada na sua elaboração, às necessidades de desenvolvimento do

sistema – não bastaria elaborar o modelo sem que sua utilidade para a construção do sistema fosse verificada. Por isto, algumas incursões tiveram de ser feitas abordando aspectos de construção do sistema, o que transcende, também, a modelagem do simulador.

Na primeira parte do texto, capítulos 2 e 3, o super-sistema e o super-projeto são analisados e o problema definido. Na segunda, capítulos 4 e 5, o paradigma multiagente e a técnica de engenharia de sistema adotada são estudados. Na terceira parte, Capítulo 6, o simulador é modelado. Na quarta parte, capítulos 7 e 8 plataformas de desenvolvimento e operação são levantados e uma escolhida. Na quinta parte, Capítulo 9, subsistemas simplificados extraídos do modelo são construídos e executados. Finalmente, no Capítulo 10, são apresentadas as conclusões.

# 2 ANTECEDENTES

O trabalho do qual a presente dissertação é um dos resultados – trabalho denominado aqui projeto do simulador – articula-se com outros esforços que caracterizam um projeto mais amplo, um super-projeto cuja finalidade é a construção de veículos aéreos autônomos não tripulados. Tal super-projeto precede e condiciona o projeto do simulador.

O propósito do sistema a ser produzido pelo projeto do simulador, sistema do qual trata esta dissertação, decorre, também, de conjecturas relativas ao super-projeto. Por este motivo, cada uma das principais suposições sobre o super-projeto e sobre o super-sistema a ele atinente é, a seguir, explicitada de maneira breve.

## 2.1 PROBLEMA GERAL

O problema geral – o problema do super-projeto – consiste na construção de uma frota de veículos aéreos autônomos não tripulados cooperativos para comunicação de dados e monitoramento de ambientes desestruturados.

### 2.1.1 REQUISITO FUNCIONAL GERAL

Como se depreende da enunciação anterior, o requisito funcional do super-sistema associado ao super-projeto é monitorar ambientes e prover comunicação de dados.

### 2.1.2 REQUISITO NÃO-FUNCIONAL GERAL

O fato da monitoração de ambiente e da comunicação de dados terem de se dar por meio de uma frota de veículos aéreos autônomos não tripulados é considerado aqui como um requisito não-funcional do super-sistema.

## 2.2 FROTA DE DIRIGÍVEIS

Uma solução possível para o problema geral compreende o uso de dirigíveis operando cooperativamente, organizados segundo uma frota. Vai-se considerar aqui que a solução de fato escolhida para resolver o problema geral usa frota de dirigíveis. Algumas das razões da viabilidade e conveniência desta escolha estão descritas sucintamente nos subitens seguintes.

### 2.2.1 PORQUE DIRIGÍVEIS

A frota de veículos aéreos é composta de dirigíveis – aeronaves mais leves do que o ar, com hélices propulsoras e sistema de navegação. A escolha do dirigível dentre os diversos tipos de veículos aéreos convencionais – aeroplano, helicóptero, dirigível e balão – decorre do que se requer dos veículos. Neles, são embarcadas plataformas de coleta de dados para monitoramento de ambientes. Tal monitoramento precisa ser feito em altitude e velocidade baixas. Os veículos devem também pairar e permitir monitoramentos de duração longa de uma mesma área; produzir turbulência e ruído muito baixos para não perturbar o ambiente que está sendo monitorado; gerar vibração muito pequena para reduzir o ruído sensorial e as chances de mau funcionamento do *hardware* embarcado; decolar e aterrissar verticalmente, possibilitando a manutenção e o reabastecimento, sem necessidade de pista de decolagem/aterrissagem. Os veículos precisam ser, ainda, altamente manobráveis – o que exclui os balões – possuir capacidade razoável de carga útil e custo operacional baixo. Este último requisito, aliás, distingue claramente os dirigíveis dos aeroplanos e helicópteros, cujos custos de aquisição, operação e manutenção são muitíssimo maiores do que os dos dirigíveis. Assim, dentre os quatro tipos de veículos aéreos convencionais possíveis, o dirigível é o mais adequado ao monitoramento, é menos demandante operacionalmente e é o de menor custo (ELFES, 1998).

Os veículos aéreos devem estabelecer enlace de telecomunicação. Em geral, tais enlaces precisam ser mantidos durante períodos longos. O baixo custo operacional dos dirigíveis torna viáveis estes *links*. Portanto, também sob o aspecto do enlace de telecomunicação, o dirigível parece ser escolha adequada.

Há já esforços bem desenvolvidos buscando a construção de dirigíveis autônomos não tripulados. Tais esforços ratificam a escolha deste tipo de veículo uma vez que, nestes projetos, os dirigíveis também se destinam a resolver, ao menos em parte, o requisito funcional geral (2.1.1). Eles atendem, também parcialmente, o requisito não-funcional geral (2.1.2). Um esforço particularmente interessante nesta linha é o Projeto AURORA.

### 2.2.2 O PROJETO AURORA

O projeto AURORA – ***A**utonomous **U**nmanned **R**emote M**o**nitoring **R**obotic **A**irship* – objetiva a criação de tecnologia para a operação semi-autônoma de dirigível não tripulado, dirigível este a ser usado como plataforma aérea em aplicações de inspeção, pesquisa e monitoramentos ambiental, climatológico e de biodiversidade (MAETA, 2001).

Como se vê, o objetivo do projeto AURORA tem pontos de toque com o do super-projeto antes mencionado. Por causa desta semelhança de propósitos, o estudo do projeto AURORA é útil aqui. Pode servir concretamente para inspirar e validar soluções, mostrar caminhos adequados e para permitir uma análise crítica do que foi feito naquele projeto, subsidiando assim o super-projeto.

Ao longo do desenvolvimento do projeto AURORA, vôos reais foram feitos, nos quais o dirigível foi controlado remotamente. Realizaram-se, também, vôos semi-automáticos.

As fases do projeto AURORA (MAETA, 2001) compreendem o desenvolvimento de sucessivos protótipos com crescente capacidade de vôo – isto é, vôos nos quais o dirigível cubra distâncias maiores e transporte maior quantidade de instrumentos – e com graus crescentes de autonomia – por meio do aumento da automação das diversas fases de vôo (decolagem, pouso, vôo obedecendo à trajetória definida, vôo regido por marcos de solo).

O AURORA I, protótipo da primeira fase, tem os seguintes componentes principais (MAETA, 2001):

- dirigível;
- sistema de controle e navegação de bordo com sensores e atuadores;
- sistema de comunicação, elo entre o sistema embarcado e a estação móvel de base;
- estação de base da qual o mais importante componente é o computador utilizado pelo operador em terra;
- sensores de bordo dedicados ao cumprimento de missões de monitoramento.

Vale a pena observar, a partir da composição do AURORA I, que a entidade constituída pelo dirigível, mais o sistema de controle e navegação de bordo com sensores e atuadores, mais o sistema de comunicação pode ser interpretada como um agente físico: ela percebe seu ambiente por meio de sensores e atua no ambiente por intermédio de atuadores – esta entidade satisfaz, portanto, a definição de agente físico de Russel e Norvig[1], por exemplo, uma vez que usa sensores e atuadores para navegar.

A entidade caracterizada no parágrafo anterior é a parte móvel do AURORA I. Esta entidade é o veículo robótico autônomo – ou semi-autônomo – (que aqui será também chamada de robô) cujos componentes, mais detalhadamente, são os seguintes (MAETA, 2001):

a) o **veículo,** que é a plataforma onde é montado o restante do sistema.

b) os **atuadores**, componentes que permitem que o veículo navegue. Os principais atuadores são o sistema de propulsão, o sistema de controle de direção e o sistema de controle de velocidade.

c) os **sensores[2],** que determinam onde o veículo se encontra e qual o seu estado atual. Por meio deles, o robô enxerga si mesmo e o ambiente.

d) o **computador de bordo**, coletor e processador dos dados dos sensores, e executor dos algoritmos que geram comandos para os atuadores.

e) o **sistema de comunicação**, que permite ao robô trocar mensagens com sistemas externos (tais como o computador de terra – usado por operador humano para programar missão, ou para receber dados de telemetria – ou outros robôs, no caso de realização de missão cooperativa).

f) e a **interface com o operador**, elo entre o operador humano e o robô.

O processo desenvolvido no computador de bordo – resultado da execução do programa nele registrado – com o suporte do próprio computador – *hardware* – e das facilidades de comunicação com os demais componentes do robô, coordena o robô na busca do cumprimento da missão navegacional. Este processo tende, também, a ser o responsável pela autonomia do robô, no curso das diversas versões do programa computacional. É assim porque os demais componentes não têm capacidade de interpretação de percepções, de resolver problemas, de traçar inferências e de determinar ações (Hayes-Roth), enquanto o

[1] Definição de Russell e Norvig (1995): "um agente é uma entidade que pode ser vista como percebendo seu ambiente através de sensores e agindo no seu ambiente através de atuadores".
[2] Estes sensores são os usados pelo robô para navegar. Os sensores usados para monitorar ambientes não são tratados aqui.

processo pode tê-las. É o *software* de bordo que encerra as propriedades robóticas do robô, isto é, é o *software* de bordo que dota o robô de inteligência e de autonomia.

Muito embora diversos componentes do robô não sejam estritamente agentes, ou, em outras palavras, não tenham inteligência – como visto no parágrafo anterior – ainda assim podem ser tratados no mesmo ambiente de modelagem no qual os agentes são tratados (DELOACH, 2001b). A MaSE – técnica de engenharia de sistemas multiagentes que será vista mais adiante – adotada neste trabalho, permite o tratamento indiferenciado de agentes e de módulos que não sejam agentes dentro da mesma *framework.*

O projeto AURORA ilustra a possibilidade de se usar o paradigma de agente para modelar veículos robóticos autônomos. A MaSE, técnica moderna de engenharia de sistema multiagente, permite que os componentes do veículo que não são agentes sejam tratados como se o fossem, no mesmo ambiente de modelagem de agente. Assim, todos os componentes do veículo robótico do AURORA I podem ser tratados sob o paradigma de agente – multiagente, mais precisamente. Como o veículo robótico do AURORA I representa bem, estruturalmente e funcionalmente, os veículos de sua classe, é razoável inferir que a modelagem de um veículo robótico autônomo em geral possa ser feita totalmente dentro do paradigma de agente. Tal conclusão é sustentada por Huff, N., Kamel, A. e Nygard, K. (HUFF, 2003).

# 3 DEFINIÇÃO DO PROBLEMA

Vai-se definir o problema por meio do objetivo do projeto do simulador, e, também, por meio do objetivo e de outros requisitos do simulador em si. Alguns aspectos adicionais pertinentes à definição do problema são tratados, como, por exemplo, o que diz respeito às premissas do trabalho.

## 3.1 OBJETIVO DO TRABALHO

O objetivo do projeto do simulador é o de modelar em *hardware* e *software* um simulador de frota de dirigíveis aéreos autônomos não tripulados capazes de operar cooperativamente – a frota servirá para comunicação de dados e monitoramento de ambientes desestruturados.

Parte-se da premissa de que é possível ter-se um modelo análogo em *hardware* e *software* do agente físico em que se constitui o dirigível robótico autônomo (como descrito em 2.2.2), dos componentes de cada dirigível robótico e dos dirigíveis robóticos compostos segundo uma frota. Nesta analogia, a plataforma de *hardware* do simulador corresponde a um dirigível como um todo; o *ethernet bus*, ou equivalente, que conecta as plataformas de *hardware* onde o *software* do simulador é executado, ao enlace de comunicação entre os dirigíveis robóticos; as diversas plataformas de *hardware* tomadas em conjunto, à frota; os atuadores, a módulos específicos do *software*[3] do simulador; os sensores, a módulos de *software*, que os representam no simulador; e, assim, para os demais componentes do simulador e do dirigível robótico, como será visto mais tarde.

O simulador, em si, destina-se a apoiar a definição, padronização e teste das conversações entre os módulos *intra* agentes físicos e entre os agentes físicos, bem como a experimentação de algoritmos de navegação.

O modelo do simulador consiste em um conjunto de especificações e de sub-modelos produzidos de acordo com a MaSE. Este modelo estabelece as classes de agentes do sistema, as conversações que podem ocorrer entre estas classes e fixa quais são os agentes propriamente ditos – quantidade, tipos e localização destes agentes – afora outras definições intermediárias (DELOACH, 2001b).

[3] Como se verá mais tarde, todos os módulos de *software* do simulador são considerados agentes.

## 3.2 OUTROS REQUISITOS DO SIMULADOR

Além dos mencionados anteriormente, outros requisitos do super-sistema são também de importância para a definição dos requisitos do simulador. Dentre os não-funcionais, tem-se os seguintes: o *software* embarcado deve ter arquitetura flexível; a semântica e a sintaxe das linguagens de comunicação entre robôs e intra-robôs devem promover a interoperabilidade dos robôs e dos módulos constituintes dos robôs; e os protocolos das conversações entre robôs e intra-robôs devem ser bem definidos, favorecendo também a interoperabilidade.

As decisões de projeto para o cumprimento desses requisitos não-funcionais são as seguintes: (1) escolher o critério mais adequado para a definição dos módulos do *software* embarcado e (2) escolher os padrões mais adequados de comunicação entre os robôs e intra-robôs de forma a que promovam a interoperabilidade de módulos e robôs.

Na escolha do critério de definição dos módulos do *software* embarcado, precisa-se considerar a heterogeneidade dos componentes físicos constituintes do robô e também a necessidade do *software* embarcado interagir com todos eles – atuadores, sensores etc. Um critério satisfatório é dotar o *software* embarcado de um módulo auto-contido para cada componente do robô – analogamente a *drivers* de componentes de PC sob o Windows – que funcione como uma interface entre o componente e o restante do *software* embarcado. Tal providência facilita a comunicação com os componentes e a substituição deles por equivalentes ou assemelhados funcionais, quando necessário, produzindo um arcabouço de projeto de *software* embarcado flexível, conveniente para as experimentações.

Para ilustrar, tome-se o AURORA I. Neste caso, distintos componentes físicos concorrem para o funcionamento do robô. São variadas modalidades possíveis de sensores – tais como, giroscópios, dos quais há muitos tipos; acelerômetros de pêndulo, ou vibracional, ou eletromagnético; inclinômetros; bússolas; *global positioning system* (GPS); etc. – muitos tipos de atuadores – motores de combustão interna, motores elétricos, motores de passo, servos, etc. – micro-controladores embarcados, etc. (MAETA, 2001). O *software* embarcado terá evidentemente de manter comunicação com os diversos componentes físicos. Se esta comunicação se der por meio de módulos auto-contidos, as trocas de componentes físicos terão menos impacto nas manutenções adaptativas do *software* embarcado, pois se restringirão apenas aos módulos diretamente relacionados com os componentes físicos substituídos. No decurso do tempo, os fabricantes de componentes físicos poderão produzir,

eles mesmos, os módulos de interface de seus próprios produtos, módulos que estenderão o *software* embarcado.

Assim como o *software* embarcado precisa considerar todos os componentes possíveis, o simulador deve igualmente levá-los em conta. A semelhança estrutural entre o *software* embarcado e o simulador facilita os testes das alternativas físicas do robô, aumentando a utilidade do simulador. Se, por exemplo, um novo componente físico do robô precisa ser testado, substitui-se no simulador apenas o agente correspondente por um novo, ajustado às especificações do novo componente físico, caso necessário.

A forma de comunicação entre robôs e intra-robôs, considerando a multiplicidade de atores – fabricantes dos componentes do robô, robôs construídos por entidades diferentes etc. – deveria obedecer a padrões sintáticos e semânticos bem conhecidos e testados. A escolha adequada pode favorecer a interoperabilidade de componentes e mesmo de robôs de origens distintas. Além desta vantagem, um proveito adicional importante é conseguido usando-se no simulador o mesmo padrão sintático e semântico de comunicação definido para os robôs: isto permite que as conclusões obtidas com o auxílio do simulador no que tange a comunicação entre componentes e entre robôs possam ser aplicadas diretamente nos robôs. Tal providência aumenta igualmente a utilidade do simulador.

Uma escolha possível de padrão de comunicação entre agentes é o padrão FIPA (*Foundations of Intelligent Physical Agents*), na realidade, um conjunto complexo de padrões – FIPA é uma organização padronizadora da sociedade de computação IEEE que promove tecnologia baseada em agente e a interoperabilidade de seus padrões com outras tecnologias. O padrão FIPA pode ser usado nos robôs e será usado no simulador.

Dos requisitos do super-sistema, aqueles com os quais o simulador lida realmente são os relativos à navegação autônoma de cada robô e à cooperação entre eles para o cumprimento da missão navegacional, quando organizados como uma frota. Questões relacionadas diretamente com monitoramento de ambientes e transmissão de dados não serão suportadas pelo simulador.

Em resumo, as hipóteses do projeto do simulador são as seguintes: a estrutura do *software* embarcado e a do simulador são semelhantes. O padrão FIPA é adotado nos robôs e no simulador. Os aspectos do super-sistema aos quais o simulador dá suporte são os referentes à navegação autônoma de cada robô e da frota de robôs.

Como conseqüência das hipóteses, os requisitos adicionais do simulador são os seguintes: o simulador terá módulos auto-contidos correspondentes a cada componente do robô. O

simulador adotará o padrão FIPA. O simulador dará suporte à navegação autônoma de cada robô e da frota de robôs.

Além destes requisitos, o simulador será projetado e construído sob o paradigma multiagente. Alguns outros argumentos em favor desta opção são apresentados nos itens subseqüentes.

# 4 PARADIGMA MULTIAGENTE

Sistemas multiagentes são, resumidamente, sistemas constituídos de agentes que interagem. A reflexão sobre sistemas multiagentes, portanto, pode ser iniciada pela consideração do conceito de agente.

## 4.1 AGENTE

Há diversas definições de agente. Algumas não fazem distinção entre agente físico e de *software*, como, por exemplo, a definição de Russell e Norvig (1995): "um agente é uma entidade que pode ser vista como percebendo seu ambiente através de sensores e agindo no seu ambiente através de atuadores". Já a definição de Smith, Cypher and Spohrer restringe-se aos agentes de *software*: "agente é uma entidade persistente de *software* dedicada a uma finalidade específica". Destas definições, uma primeira constatação que se pode fazer é a de que há, pelo menos, dois tipos de agentes: agente de *software* e agente "não-*software*", que chamaremos aqui de agente físico.

Convém observar, também, que as duas definições anteriores permitem classificar qualquer processo computacional[4] como agente. Realmente, se o ambiente produzir dados para o processo e receber dados do processo; se a recepção de dados (entrada) pelo processo for considerada como sentir; e a produção de dados (saída) for considerada como agir; então qualquer processo pode ser classificado, segundo a definição de Russell e Norvig, como agente (FRANKLIN, 1996) *apud* (KAYSER, 2001). Igualmente, praticamente todos os processos têm finalidade específica e são persistentes. Assim, as duas definições não distinguem processos computacionais em geral de agentes.

Há, ainda, definições de agentes inteligentes como, por exemplo, a de Hayes-Roth, de 1995: "agentes inteligentes executam continuamente três funções: percepção de condições dinâmicas no ambiente; ação para afetar condições no ambiente; raciocínio para interpretar percepções, resolver problemas, traçar inferências e determinar ações". Maes, 1995, define agente autônomo: "agentes autônomos são sistemas computacionais que habitam algum ambiente dinâmico complexo, fazem o sensoriamento deste ambiente e atuam autonomamente nele, e, em fazendo assim, realizam o conjunto de objetivos ou tarefas para os

[4] Processo resultante da execução de um programa computacional.

quais são projetados". De acordo com a literatura, portanto, os agentes podem ser, ou não, inteligentes. Mais ainda, segundo Maes, agentes autônomos são necessariamente sistemas computacionais.

Considerando-se, também, que um agente autônomo, em geral, tem de perceber seu ambiente, interpretar as percepções, resolver problemas, traçar inferências e, finalmente, agir no ambiente para a consecução autônoma de seus objetivos, tem-se que um agente autônomo é também um agente inteligente.

Para Ferber [5], 2001, a definição geral de agente é a seguinte:

Um agente é uma entidade física ou virtual

1. que é capaz de atuar em um ambiente,
2. que pode se comunicar diretamente com outros agentes,
3. que é orientado por um conjunto de tendências (na forma de objetivos individuais ou de uma função de satisfação/sobrevivência que ele tenta otimizar),
4. que possui recursos próprios,
5. que é capaz de perceber seu ambiente (mas em uma extensão limitada),
6. que tem somente uma representação parcial de seu ambiente (e talvez nenhuma),
7. que possui habilidades e que pode oferecer serviços,
8. que pode estar apto a reproduzir a si próprio,
9. cujo comportamento tende a satisfazer seus objetivos, levando em conta os recursos e habilidades disponíveis para ele, e que depende da percepção, representação e comunicações que este agente recebe.

Cabem algumas observações adicionais sobre agentes, que complementam e explicitam alguns pontos da definição de Ferber (D. Grimshaw).

1. Agentes são capazes de atuar, não, apenas, de raciocinar. As ações afetam o ambiente que, por seu turno, afetam futuras decisões de agentes.
2. Uma propriedade chave de agente é a autonomia. Eles são, ao menos em alguma medida, independentes. Eles não são inteiramente pré-programados, podendo tomar decisões baseadas em informações de seu ambiente ou de outros agentes.

[5] *Apud* D. Grimshaw em http://www.ryerson.ca/~dgrimsha/courses/cps720/agentDef.html.

3. Uma propriedade opcional de agente é a mobilidade, a capacidade de o agente mudar de posição no ambiente. Como observado por Grimshaw, é curioso que Ferber não tenha incluído a mobilidade como uma propriedade possível de agentes. Embora opcional, tal propriedade caracteriza alguns sistemas de agentes (Lange e Oshima, *apud* Grimshaw). Para ilustrar, observe-se que, no mundo natural, a inteligência de agente está sempre associada à mobilidade (animais), enquanto os outros seres vivos, as plantas, não têm inteligência (Grimshaw).

Uma última definição de agente, importante no contexto deste trabalho – pois adotada pela MaSE, e a MaSE é uma das técnicas usadas aqui – é a de Deloach (DELOACH, 2001b): agentes são uma especialização de objetos (na acepção do paradigma de orientação a objetos). Assim, ao invés de simples objetos com métodos que podem ser invocados por outros objetos, agentes coordenam-se entre si por meio de conversações e atuam pró-ativamente para cumprir metas individuais e de sistema. Agentes são meramente uma abstração conveniente, que podem, ou não, possuir inteligência.

Uma observação interessante é que Deloach define agente no contexto de agentes. Assim, para ele, o agente não existe isoladamente de outros agentes.

Esta definição operacional permite que no âmbito da técnica MaSE e de sua ferramenta, o agentTool, os agentes sejam vistos meramente como uma abstração conveniente, que podem, ou não, possuir inteligência, e, portanto, nelas, os componentes de sistema, inteligentes e não inteligentes, são tratados da mesma maneira, dentro de uma única estrutura de trabalho (*framework*) (DELOACH, 2001b). O tratamento indiferenciado dos vários tipos de componentes que a MaSE possibilita facilita a execução das tarefas de projeto.

Como se vê, há muitas definições de agentes, muitas incompletas, e mesmo conflitantes em alguns pontos. Para contornar esta dificuldade, vai-se considerar agente como um conceito primitivo, a exemplo do que acontece com ponto e plano, na geometria euclidiana, e com a interpretação de probabilidade – há várias, todas imperfeitas – na teoria de probabilidade[6]. Assim, considera-se que as diversas definições existentes explicitem algumas das reais propriedades de agente, embora não o definam completamente. Isto pode ser suficiente em situações bem concretas: para a MaSE, por exemplo, a principal propriedade de agente é a capacidade de comunicação, importando menos autonomia, por exemplo, e, com a MaSE, têm sido construídos sistemas multiagentes. Portanto, as definições parciais de agentes podem

[6] *Probability and Statistics,* DeGroot, Morris H. Addison-Wesley Publishing Company, Sec. Edition, 1986.

ser bastante úteis, pois têm possibilitado a construção de sistemas eficazes baseados em agentes.

## 4.2 SISTEMA MULTIAGENTE

Uma vez visto o conceito de agente, pode-se abordar o conceito de sistema multiagente. A abordagem se dá pelo levantamento das definições de sistema multiagente e motivações de uso de tais sistemas.

Por ser de fundamental importância, a coordenação dos agentes componentes é também considerada.

### 4.2.1 DEFINIÇÕES

Sistema multiagente (SMA) é aquele composto de diversos agentes capazes de interação mútua. A interação pode se dar sob a forma de passagem de mensagem, de produção de mudanças no ambiente comum a eles, ou de produção de mudança de ambiente *e* passagem de mensagem – como, por exemplo, quando uma pessoa fala com outra: o aparelho fonador do falante impressiona o ar para transmitir o som (mudança do ambiente), som que porta uma mensagem (que será decodificada pelo receptor); e os agentes podem ser autônomos.

Um SMA pode incluir agentes humanos. Organizações humanas e sociedades em geral podem ser consideradas como exemplos de sistema multiagente.

Uma segunda definição de SMA caracteriza-o como sendo aquele que contém os seguintes elementos (Grimshaw):

1. um ambiente E, em geral um espaço com volume.
2. Um conjunto de objetos O. É possível em um dado momento associar qualquer objeto de O a uma posição em E.
3. Um agrupamento de agentes A (um subconjunto de O) que representa as entidades ativas no sistema.
4. Um agrupamento de relações R que liga objetos (e, portanto, agentes) uns aos outros.
5. Um agrupamento de operações Op, tornando possível a agentes de A perceber, produzir, transformar e manipular objetos em O.

6. Operadores com a tarefa de representar a aplicação destas operações e a reação do mundo a esta tentativa de modificação, reações que são aqui chamadas de leis do universo.

Dois casos especiais importantes desta definição geral são o dos agentes exclusivamente situados e o dos agentes de comunicação pura[7].

Segundo Grimshaw, agentes exclusivamente situados – agentes situados: agentes dos quais se sabe a cada momento a posição no ambiente E – podem ser caracterizados usando-se as características e as circunstâncias operacionais de um robô. No caso de um robô, E, o ambiente, é o espaço-3 Euclidiano. A são os robôs e O, não somente os outros robôs, mas, também, objetos físicos, tais como obstáculos – observe-se que estes robôs, particularmente, não são capazes de qualquer comunicação, pois são exclusivamente situados; mesmo sem ela, contudo, formam um sistema, quando, por exemplo, se coordenam por modelagem mútua (Genesereth et al., 1986, *apud* (REIS, 2003)); uma outra possibilidade é a de obedecerem coletivamente a alguma diretriz comum a todos.

Também segundo Grimshaw, agentes de comunicação pura têm as seguintes características: A = O e E é vazio. Os agentes são todos interligados em redes de comunicação, e se comunicam enviando mensagens.

Para Grimshaw, por comparação com a definição geral de agente, um agente exclusivamente de comunicação (ou agente de *software*) é definido como uma entidade computacional que

1. está em um sistema computacional aberto (reunião de aplicações, redes e sistemas heterogêneos),
2. pode se comunicar com outros agentes,
3. é orientado pelos seus próprios objetivos,
4. possui recursos próprios,
5. tem somente uma representação parcial de outros agentes,
6. possui habilidades (serviços) que pode oferecer a outros agentes,
7. tem comportamento tendente à obtenção de seus objetivos, comportamento que leva em conta seus recursos e habilidades, e que é dependente de suas representações e das comunicações que recebe.

[7] Caracterizações adicionais de agente são feitas a seguir. Isto se deve ao fato delas decorrerem de interações do agente com outros agentes, com o ambiente e com o tipo de ambiente no qual o agente está imerso, isto é, de como ele interage no ambiente multiagente. Por tais razões, estes atributos adicionais de agente são descritos aqui, e, não, em 4.1.

Ainda segundo Grimshaw, um agente puramente situado é definido como uma entidade física (ou talvez uma entidade computacional, se ele é simulado) que

1. está situada em um ambiente,
2. é orientada por uma função de sobrevivência/satisfação,
3. possui recursos próprios em termos de energia e ferramentas,
4. é capaz de perceber seu ambiente (mas em extensão limitada),
5. não tem praticamente representação de seu ambiente,
6. possui habilidades,
7. pode talvez reproduzir-se,
8. tem comportamento tendente a atender sua função de sobrevivência/satisfação, comportamento que leva em conta seus recursos, percepções e habilidades.

Um agente puramente comunicativo distingue-se do conceito geral de agente porque (Grimshaw)

- não tem percepção de outros agentes,
- suas tendências tomam a aparência de objetivos,
- não atua em um ambiente espacial normal, mas em uma rede computacional.

Vale observar que, no âmbito destas definições, os agentes JADE[8] são primariamente puramente comunicativos.

Para a MaSE, o que interessa nos SMA é o comportamento coordenado de seus agentes individuais para produção de um comportamento especificado a nível-de-sistema (DELOACH, 2001b). Nesta visão pragmática, a propriedade de relevo de agente para a consecução do comportamento desejado a nível-de-sistema é a capacidade dos agentes se comunicarem. A técnica, portanto, não é a mais adequada a projetos de SMA puramente situados.

Merece ser mencionado que, apesar da afirmação constante em (DELOACH, 2001b), "*...to solve complex problems, these agents must work cooperatively with other agents in a heterogeneous environment. This is the domain of multiagent systems*…", a MaSE pode ser usada em projetos de SMA cuja coordenação seja competitiva – aliás, o domínio dos SMA inclui agentes que trabalham competitivamente: protocolos de negociação competitiva[9] traduzem-se, em última análise, em comunicações (mensagens) trocadas entre os agentes, e a MaSE é capaz de lidar com tais mensagens.

[8] Agentes JADE são agentes escritos em JAVA, desenvolvidos e executados em ambiente denominado JADE. Serão vistos em mais detalhes posteriormente.
[9] Norma de interação comunicativa de agentes cujos objetivos são distintos, visando a realização de objetivo do sistema.

### 4.2.2 SMA – MOTIVAÇÕES DE USO

Vários são os motivos de se usar SMA. Um é o fato de muitos dos problemas encontrados serem inerentemente distribuídos – este é o principal motivo. Outro é a grande dimensão de alguns problemas: problemas muito grandes são de difícil solução por meio de um único agente monolítico. Uma outra razão de se utilizar SMA é a de o paradigma SMA permitir solução natural de problemas geográfica e/ou funcionalmente distribuídos; e, ainda, porque projetos SMA são mais claros e simples do que seus correspondentes de um só agente (REIS, 2003).

Os casos específicos do simulador e do *software* embarcado podem ser categorizados como problemas muito grandes – muito complexos – para serem resolvidos por um único agente monolítico. O paradigma multiagente, portanto, pode ser adequado para tratá-los.

Mesmo a navegação autônoma de um único robô é problema complexo, sendo adequada, por isto, a adoção do paradigma de SMA. Esta adoção é recomendável não apenas por este motivo: o paradigma de programação de agente foi usado com sucesso na construção de um sistema de gerência de missão de – um único – veículo autônomo não tripulado (VANT)[10] (KARIN, 2004) – sistema semelhante ao de navegação autônoma tratado pelo simulador. O sucesso do uso do paradigma de SMA neste VANT – denominado Codarra Avatar – pode ser atestado pelo fato de que teria sido ele o primeiro a voar autonomamente[11]. Assim, o uso bem sucedido do paradigma de SMA no Codarra Avatar é mais um motivo para a sua adoção aqui.

Outra motivação de se usar o paradigma multiagente para modelar um único robô é que o *framework* correspondente é o mesmo do tratamento da frota de robôs – e a frota pode ser modelada com naturalidade como um SMA. Assim, a plataforma de projeto e desenvolvimento da navegação autônoma de um único robô pode ser estendida sem descontinuidades para a frota de robôs – uma das razões, aliás, do paradigma de SMA ter sido escolhido pela equipe do Codarra Avatar (KARIN, 2004).

---

[10] Este paradigma foi levado à prática por meio da linguagem de programação JACK Intelligent Agents. Embora no texto consultado não seja explicitamente dito que o paradigma de programação de agente é um paradigma multiagente, infere-se que seja, pois a linguagem JACK trata agentes (no plural).

[11] Ver *Agent Oriented Software*, www.agent-software.com e Codarra *Advanced Systems*, www.codarra.com.au.

Em (FRAZÃO, 2004), os agentes são classificados como compostos e simples. Agentes compostos são agentes constituídos de mais de um agente. O princípio subjacente aos agentes compostos é abstrair um grupo de agentes relacionados. Agentes compostos permitem que um grupo de agentes seja visto como um único agente pelos projetistas, operadores, ou outras partes do sistema. Para isto ser possível, um agente composto deve ter o controle dos agentes que o compõem.

Agentes simples são agentes que não controlam outros agentes. Representam dispositivos de *hardware*, fusão de dados e laços de controle. Ainda em (FRAZÃO, 2004), são definidos agente sensor – um *driver* ou servidor de um dispositivo de *hardware* do tipo sensor – e agente atuador – um *driver* ou servidor de um dispositivo de *hardware* do tipo atuador.

O robô pode ser visto como um agente composto, constituído de agentes sensores e agentes atuadores, dentre outros. Por ser constituído de mais de um agente, admite a abordagem multiagente. Esta é outra razão da adoção do paradigma SMA na modelagem da navegação autônoma do robô.

Como motivação final, há ainda a facilidade oferecida pela MaSE, que permite o tratamento de módulos inteligentes, ou não, da mesma maneira e na mesma estrutura de trabalho.

### 4.2.3 COORDENAÇÃO DE AGENTES COOPERATIVOS

Coordenação é o processo pelo qual um agente raciocina sobre suas ações locais e as ações (antecipadas) de outros agentes, com o objetivo de garantir que a comunidade de agentes funcione de maneira coerente (JENNINGS, 1996). Um SMA é um sistema de agentes, e, por isto, sob pelo menos um aspecto, os agentes têm de funcionar de maneira coerente para a produção dos resultados de sistema. Assim, a coordenação é processo essencial dos SMA.

Há diversos tipos de coordenação. Conhece-los pode auxiliar na escolha dos mais adequados ao simulador, robô e frota de robôs. A aquisição deste conhecimento, contudo, é dificultada porque não existe consenso na literatura sobre o significado de termos básicos, como, por exemplo, o próprio termo coordenação – a exemplo do que acontece com os termos agente e sistema multiagente. Assim, as diversas taxonomias de coordenação, que estabelecem os tipos de coordenação e seus significados, não são comparáveis, pois tratam de objetos conceituais distintos, dependendo da referência consultada. Em (FARINELLI, 2004),

trabalho relativamente recente, por exemplo, faz-se até mesmo distinção entre sistemas multirobôs (SMR)[12] e SMA, tratados como coisas diferentes: os SMA seriam compostos somente de agentes de *software*, enquanto os SMR, de robôs, afora outras diferenças.

Vai-se usar aqui, ora o conceito de SMR, ora o de SMA, sem maiores rigores dogmáticos, e sem ênfase em todos os aspectos que os distinguem. Quando conveniente, usar-se-á um, ou outro termo, em contextos nos quais eles claramente sejam aplicáveis.

Segundo (REIS, 2003), em um SMA, os agentes – e não, em princípio, a coordenação deles – podem ser competitivos, ou cooperativos. Derivam daí dois tipos de coordenação: a coordenação de agentes competitivos e a de agentes cooperativos.

Nos trabalhos recentes em SMR, os sistemas multirobôs competitivos têm recebido pouca atenção (FARINELLI, 2004), o que significa que este tipo de SMR tem atualmente raras aplicações. Por causa disto, a taxonomia de SMR apresentada em (FARINELLI, 2004), por exemplo, desconsidera os SMR competitivos. Desconsiderar-se-á aqui, também, os SMR e, também, os SMA competitivos.

Segundo (REIS, 2003), a definição de coordenação de agentes cooperativos é a mesma de coordenação em geral. Então, o que torna a coordenação de agentes cooperativos específica é justamente o fato deles, os agentes, serem cooperativos – agentes cooperativos são os capazes de atuar em conjunto com outros para atingir um propósito que os transcende, e que é o propósito da comunidade (sistema) à qual pertencem[13].

Em (REIS, 2003), os tipos de coordenação de agentes cooperativos considerados, pré-existentes ao texto na literatura, são os seguintes: coordenação por distribuição de tarefas e resultados, planejamento multiagente, modelagem mútua e coordenação por organização estrutural. São também analisados modelos de coordenação mais específicos propostos por diversos autores, incluindo as intenções conjuntas, planos partilhados, o *locker room agreement*, o esquema *TAEMS* e o modelo *STEAM*. No texto, são propostos, ainda, novos tipos de coordenação, distintos dos usuais, incluindo a coordenação por percepção inteligente, por controle parcialmente hierárquico e a coordenação estratégica. Aqui serão repassados os tipos de coordenação tratados em (REIS, 2003) que parecem mais adequados aos problemas em tela – frota de robôs e simulador – e aos recursos de que se dispõe.

---

[12] Um SMR cooperativo é aquele composto de robôs que operam em conjunto para executar alguma tarefa global (F.R. Noreils, "*Toward a robot architecture integrating cooperation between mobile robots: Application to indoor environment*", *Int. J. Robot Res., vol.* 12, *no* 1, *pp* 79-98, 1993, *apud* (FARINELLI, 2004) ).

[13] Enquanto os agentes cooperativos visam a satisfação, mesmo que parcial, dos objetivos do sistema ao qual pertencem, os agentes competitivos visam a satisfação de seus próprios interesses. Os agentes competitivos, na busca da satisfação de seus interesses, podem contribuir para o alcance dos objetivos do sistema do qual fazem parte.

A coordenação por distribuição de tarefas e resultados consiste em três fases (Smith, R. G. e Davis, R. *Frameworks for Cooperation in Distributed Problem Solving,* IEEE *Transactions on System, Man and Cybernetics*, Vol.11(1), 1980, *apud* (REIS, 2003)). Na primeira, o problema geral é decomposto hierarquicamente em problemas de nível inferior. A decomposição prossegue até que os subproblemas sejam tratáveis por agentes individuais e tenham mínima interdependência. Na segunda fase, os problemas são resolvidos. Esta fase envolve a alocação prévia dos problemas aos agentes e a troca de informação entre os agentes durante a resolução. A fase final consiste na integração das soluções dos subproblemas individuais na solução global.

Um protocolo[14] que se destina a apoiar a coordenação por distribuição de tarefas entre agentes é o protocolo da rede contratual – *contract-net* – (*idem, ibidem*). É essencialmente um protocolo de alto-nível para a contratação de tarefas. Para definir este protocolo, Smith, seu autor, inspirou-se no modo de executar a contratação usada em empresas reais. A FIG. 4.1 esboça o protocolo.

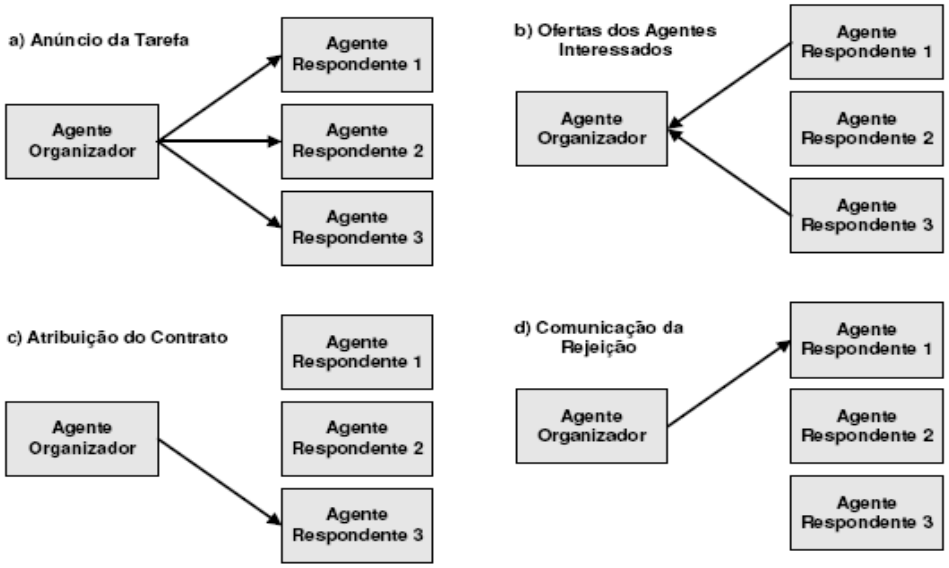


FIG 4. 1: O Protocolo *Contract-Net* (REIS, 2003).

O protocolo rede contratual se destina a determinar a possibilidade de realização, ou não, de cooperação e, em caso afirmativo, a forma concreta de como a cooperação se vai realizar: a qual agente será pedida a execução da tarefa. Apesar de muito simples, – e eventualmente por isso mesmo – a rede contratual tornou-se o protocolo de coordenação por distribuição de

[14] Aqui, na acepção de código de conduta entre os agentes.

tarefas mais usado e estudado. Comparando-se o protocolo com as fases em que se decompõe a coordenação por divisão de tarefas, percebe-se que o protocolo suporta diretamente, apenas, a subfase de alocação dos problemas aos agentes. Outro ponto a se observar é que os agentes respondentes são competitivos. Somente o agente organizador e o agente ao qual o contrato é atribuído atuam cooperativamente.

A coordenação por distribuição de resultados (*idem, ibidem*) implica a troca de informação relevante entre os agentes para a resolução do problema global. Esta informação pode ser trocada de forma pró-ativa, – um agente decide autonomamente enviar informação a outros agentes que pensa lhes poder ser útil – ou reativa – o envio de informação é realizado em resposta ao pedido de um dado agente. Na literatura, este tipo de coordenação é usualmente apresentado em conjunto com a distribuição de tarefas, no âmbito da descrição de agentes que efetuam a resolução distribuída de problemas. Nesta perspectiva, os resultados partilhados correspondem a soluções de problemas menores que são progressivamente compostas de modo a formarem a solução global do problema completo.

A resolução distribuída de problemas – método subjacente tanto à coordenação por distribuição de tarefas, quanto à coordenação por distribuição de resultados – apresenta dificuldades. Uma das principais está relacionada com o aparecimento de inconsistências entre os diversos agentes presentes no sistema (REIS, 2003). Os agentes podem apresentar inconsistências quanto a crenças[15] e objetivos[16] – quando o sistema possui dimensão razoável, os agentes são autônomos e o ambiente é inacessível e dinâmico, as inconsistências são inevitáveis (REIS, 2003). Não obstante as diversas propostas existentes na literatura para lidar com esta dificuldade, tem-se de levar em conta a possibilidade dos resultados produzidos pelos agentes serem inconsistentes, o que pode refletir-se adversamente na solução global do problema completo.

A coordenação por planejamento multiagente – planejamento das atividades de um conjunto de agentes – decorre da consideração de que as atividades dos diferentes agentes no sistema podem interferir umas com as outras (REIS, 2003). O planejamento multiagente pode se dar basicamente de três maneiras: planejamento centralizado de planos distribuídos, planejamento distribuído de um plano global e planejamento distribuído de planos distribuídos.

[15] Crença: informação que o agente tem sobre o estado do mundo.
[16] Objetivo: intenção do agente.

No planejamento centralizado de planos distribuídos, um agente desenvolve centralmente planos para um conjunto de agentes, planos nos quais as divisão e alocação de tarefas se encontram especificadas. O agente central distribui os planos parciais aos restantes agentes que funcionam em nível hierarquicamente inferior – como escravos – e que têm a responsabilidade de executá-los (REIS, 2003).

No planejamento distribuído de um plano global, um grupo de agentes coopera de forma a gerar um plano global de atividades. Neste caso, tipicamente, agentes componentes do sistema são especialistas em diferentes aspectos necessários para o desenvolvimento do plano global. É usual que os agentes geradores do plano não sejam os responsáveis pela sua execução (REIS, 2003).

No planejamento distribuído de planos distribuídos, um grupo de agentes coopera de forma a gerar planos individuais de ação, coordenando dinamicamente as suas atividades ao longo desse planejamento. Os agentes podem ser egoístas – competitivos – e os conflitos inerentes a esta característica resolvidos por meio de técnicas de negociação.

Claro que também é possível a elaboração centralizada de um plano global, mas, neste caso, não se está perante um sistema multiagente.

Em geral, o planejamento centralizado é mais simples do que o planejamento distribuído, pois o agente central tem visão global do problema e pode definir as relações de coordenação das quais necessitar. No entanto, tal planejamento centralizado nem sempre é possível, e, muitas vezes, não é desejável, seja devido à elevada dimensão do problema; à necessidade de especialistas distintos, necessários à elaboração do plano; à distribuição geográfica dos agentes; ou à necessidade de manutenção de privacidade da informação. O caso mais complexo da coordenação por planejamento multiagente é o terceiro, em que um plano global pode nem mesmo existir. Em tal situação, os agentes produzem em conjunto, dinamicamente, os planos parciais que serão executados por cada um deles.

Como foi visto em 4.1, um agente percebe seu ambiente apenas em extensão limitada. Assim, nenhum agente é capaz de gerar um plano completo, capaz de resolver o problema global, e o planejamento feito por qualquer agente para atingir a solução do problema global é, apenas, parcial.

A técnica de coordenação por planejamento global parcial (PGP) considera a limitação de percepção dos agentes que se mencionou no parágrafo anterior. Não obstante ser parcial, o planejamento é global, no sentido em que os agentes formam planos não-locais pela a partilha de planos locais, de forma a atingir a resolução do problema global (REIS, 2003). Este

planejamento tem três fases (Durfee e Lesser, *apud* (REIS, 2003)): geração de objetivos, troca de informação e alteração de planos locais. Na fase de geração de objetivos, cada agente decide quais são os seus objetivos e define seus planos de curto-prazo que lhe permitam atingir estes objetivos. Na fase de troca de informação, os agentes trocam informação sobre planos, objetivos e soluções, visando determinar onde os planos locais interagem. Na fase de alteração de planos locais, os agentes modificam seus planos locais para coordenarem suas atividades, de acordo com as informações de que dispõem.

O PGP baseia-se em uma estrutura de plano – estrutura de dados – denominada plano global parcial. Este plano é gerado cooperativamente por todos os agentes, e contém objetivos, mapas de atividade e grafo de construção de solução. Objetivos são aqueles compartilhados pela comunidade de agentes, e que o plano gerado pelo sistema deve resolver. Os mapas de atividade representam as atividades de cada agente e os resultados esperados destas atividades. O grafo de construção da solução representa a forma segundo a qual os agentes devem interagir e as informações que devem trocar para gerar a solução sistêmica.

Algumas observações finais sobre o PGP. O PGP foi utilizado num dos primeiros simuladores para teste de SMA, a bancada de monitoração distribuída de veículos. Este simulador foi desenvolvido com o objetivo de permitir comparação de diferentes métodos de resolução distribuída de problemas. Permitia o teste de agentes de resolução distribuída de problemas, utilizando como problema *standard* o domínio da detecção e monitoração distribuída de veículos. O objetivo consistia em detectar e seguir um conjunto de veículos que passava por uma dada região, monitorada por um conjunto de sensores distribuídos espacialmente. O domínio de aplicação era essencialmente apropriado para agentes reativos, pois o aparecimento de novo veículo no sistema sempre dava início a um processo de detecção e monitoramento. A rapidez de processamento era essencial, uma vez que o domínio era dinâmico e exigia que os agentes rapidamente determinassem as trajetórias dos veículos presentes. Este domínio conduziu à criação do PGP (Durfee e Lesser, 1987, *apud* (REIS, 2003)).

O planejamento global parcial generalizado (PGPG) é uma extensão do PGP. Esta extensão se dá pela utilização de cinco técnicas na coordenação dos agentes, e que são as seguintes: comunicação de informação, comunicação de resultados, tratamento de redundância, tratamento de relações rígidas de coordenação e tratamento de relações flexíveis de coordenação. A comunicação de informação visa à atualização das perspectivas não-locais, pois os agentes possuem somente perspectivas locais em ambientes inacessíveis. A

comunicação de resultados significa a troca de todos os resultados obtidos, ou a comunicação dos resultados que o agente acredite interessem aos outros agentes, ou, ainda, unicamente, a comunicação dos resultados que interessem ao planejamento. O tratamento da redundância – redundância que acontece quando dois ou mais agentes resolvem o mesmo problema – pode limitá-la, para controlar o desperdício de recursos; ou promove-la, para aumentar a confiabilidade e a qualidade da solução obtida. O tratamento das relações rígidas de coordenação – acontecidas quando a execução de uma dada ação por parte de um agente interfere na execução da ação de outro agente – consiste, em essência, em evitar ações conflituosas por meio do re-escalonamento delas – um exemplo de uma relação deste tipo é *enables*(*T1,T2*), na qual o término da tarefa *T1* é necessário para que *T2* possa ser executada (Decker, *apud* (REIS, 2003)). O tratamento de relações flexíveis de coordenação – relações ocorridas quando as interferências conflituosas não são críticas para o sucesso das ações (embora possam alterar significativamente a eficiência ou qualidade de execução) – consiste no re-escalonamento não obrigatório das ações.

Em comparação com o PGP, o PGPG acrescenta, ainda, o escalonamento de tarefas com *"deadlines"*, heterogeneidade nos agentes e a comunicação a múltiplos níveis de abstração. Estas extensões tornam este mecanismo muito mais flexível e utilizável na prática. O mecanismo foi levado à prática na sua totalidade no simulador *TAEMS* (Decker, *apud* (REIS, 2003)).

Na coordenação por modelagem mútua, uma coordenação sem comunicação, cada agente cria um modelo de cada um dos outros agentes da sua equipe. Este modelo permite que o agente se coloque no lugar do outro agente para prever quais serão as suas ações em determinada circunstância. Do modelo, devem constar as crenças, intenções, desejos, capacidades, e qualquer outra informação relevante sobre cada um dos demais agentes a modelar. A coordenação por modelagem mútua foi sugerida inicialmente por Genesereth *et al*., e é baseada em modelos derivados da teoria dos jogos (REIS, 2003). No trabalho de Genesereth *et al* sob o tema "cooperação sem comunicação", cada agente dispunha de informação sobre as preferências dos outros agentes, e com ela podia estimar qual a ação racional eles deveriam executar em cada instante.

Les Gasser utilizou também um tipo de coordenação semelhante, no âmbito do seu sistema MACE (Gasser et al., 1987, apud (REIS, 2003)). O sistema MACE foi um dos primeiros ambientes de teste desenvolvidos para sistema multiagente. Contém cinco componentes principais: agentes de aplicação, agentes de sistema predefinidos (por exemplo,

interfaces com o utilizador), serviços disponíveis a todos os agentes, base de dados (com descrições dos agentes) e conjunto de *kernels* que tratam das comunicações físicas em cada máquina que executa o sistema.

Em seu estudo, Gasser identificou três aspectos essenciais dos agentes: contêm conhecimento; têm percepção do ambiente; e possuem capacidade para executar ações nesse ambiente. Dividiu, ele, também, o conhecimento de cada agente em dois tipos, conhecimento local e especializado do domínio, e conhecimento sobre os outros agentes (*acquaintance knowledge*).

O conhecimento sobre os outros agentes permite a execução de cooperação por modelagem mútua e inclui os seguintes tópicos: classe, os agentes MACE são organizado por classes. Nome, cada agente possui um único nome na classe; o par (classe, nome) identifica cada um dos agentes. Papéis, que descrevem as funções dos agentes no interior de cada classe. Habilidades, significando as funções e tarefas que os outros agentes são capazes de executar, ou os recursos que são capazes de providenciar. Objetivos, os objetivos dos outros agentes. Planos, a perspectiva do agente de como os outros agentes irão atingir os seus próprios objetivos; é uma lista de habilidades e operações que os demais agentes irão utilizar e executar para realizar seus objetivos.

A utilização de modelos dos outros agentes na construção de um SMA foi também utilizada no âmbito do projeto ARCHON (*ARchitecture for Cooperative Heterogeneous ON-line systems)* (Wittig *et al.*, 1994, e Oliveira *et al.*, 1992, *apud* (REIS, 2003)). O projeto *ARCHON* foi o maior projeto europeu que se realizou na área da Inteligência Artificial Distribuída (IAD) (REIS, 2003). No seu âmbito foi definida uma arquitetura genérica de agentes (FIG. 4.2) e uma técnica adequada à definição de sistemas baseados em IAD para um elevado número de problemas industriais reais. Duas destas aplicações (gestão da distribuição de energia elétrica e controle de acelerador de partículas) foram aplicadas com sucesso nas organizações reais para as quais foram desenvolvidas (Iberdrola no norte de Espanha e CERN – Centro Europeu para a Investigação em Física de Alta-Energia em Genebra) (REIS, 2003).

Em (FARINELLI, 2004) é apresentada uma taxonomia para SMR. A principal motivação deste estudo, como nele é afirmada, é a possibilidade de se aproveitar a coordenação para melhorar o desempenho do sistema. Portanto, a classificação proposta tem por foco aspectos de coordenação e é, por isto, inspirada pelas relações com o campo dos SMA (FARINELLI, 2004).

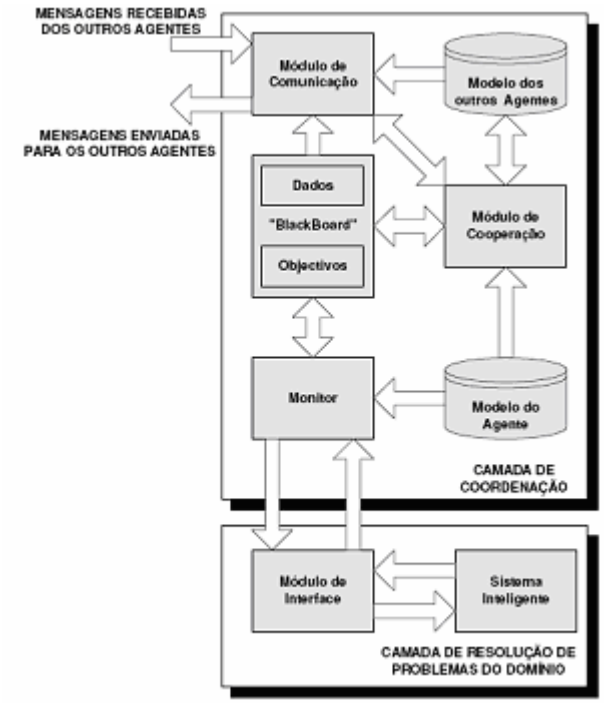


FIG 4. 2: Arquitetura Genérica de Agente (REIS, 2003).

A taxonomia proposta para classificar os trabalhos em SMR é caracterizada por dois grupos de dimensões[17]: dimensões de coordenação e dimensões de sistemas. Grosso modo, o primeiro grupo visa caracterizar o tipo de coordenação que é obtido em SMR, enquanto o último inclui aspectos do sistema que influenciam o desenvolvimento da equipe de robôs. Para as dimensões de coordenação da taxonomia proposta, uma estrutura hierárquica é apresentada na FIG. 4.3 (FARINELLI, 2004).

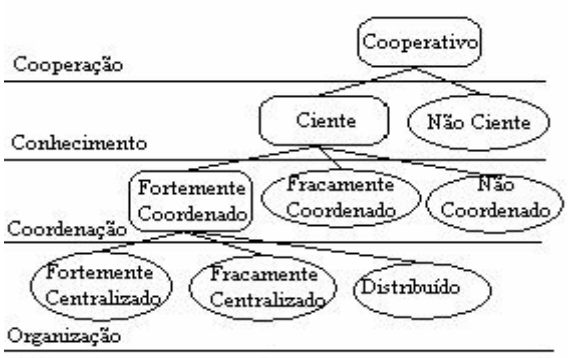


FIG 4. 3: Taxonomia SMR – dimensões de coordenação (FARINELLI, 2004).

[17] Em (FARINELLI, 2004), é dito o seguinte: "*...we use* dimension *to refer to specific features that are grouped together in the classification"*, ou, como é mais comumente dito na literatura, dimensão é uma categoria em uma taxonomia. Os níveis de interação humana com um robô, e.g., são uma dimensão (em (HOLLY, 2002)).

As dimensões de coordenação estão compostas segundo uma estrutura hierárquica na FIG. 4.3. Os valores possíveis em uma dimensão estão relacionados hierarquicamente e diretamente com os valores das dimensões adjacentes, quando a relação existe. Assim, por exemplo, os valores possíveis da dimensão "Conhecimento" são "Ciente" e "Não Ciente". "Ciente" e "Não Ciente" relacionam-se com o valor "Cooperativo" da dimensão adjacente de nível hierárquico mais elevado, a dimensão "Conhecimento". "Ciente" relaciona-se, ainda, com os valores "Fortemente Coordenado", "Fracamente Coordenado" e "Não Coordenado" da dimensão hierárquica de nível imediatamente inferior, "Coordenação".

Os níveis hierárquicos da taxonomia correspondem às dimensões de coordenação, e são os seguintes: nível de cooperação, nível de conhecimento, nível de coordenação e nível de organização.

O primeiro nível, o nível de cooperação, diz respeito à capacidade dos robôs componentes do sistema cooperarem para que o sistema leve a cabo uma tarefa específica. No nível de cooperação, distingue-se os sistemas cooperativos dos que não o são. A taxonomia, como já foi mencionado antes, trata, apenas, de SMR cooperativos, e, nela, SMR se refere a robôs cooperativos organizados como equipe. Esta noção de cooperação é muito similar à usada para SMA, contudo, em SMA, a cooperação é freqüentemente comparada e fundida com competição, a qual, até o momento em que (FARINELLI, 2004) foi escrito (dezembro de 2003), havia recebido pouca atenção nos trabalhos sobre SMR.

O segundo nível da estrutura hierárquica proposta, o nível de conhecimento, diz respeito ao conhecimento que cada robô da equipe tem sobre seus colegas de equipe. Robôs cientes têm algum tipo de conhecimento a respeito de seus colegas de time, enquanto robôs não cientes atuam sem qualquer conhecimento a respeito dos outros robôs no sistema. O interesse nos SMR com cooperação não ciente (incomum em SMA) é motivado pela simplicidade de tais sistemas, em relação aos cientes. A noção de conhecimento não é equivalente à de comunicação: o uso de mecanismo de comunicação não acarreta consciência – dos demais agentes – e vice-versa. Um SMR pode ser ciente e, mesmo assim, não haver comunicação direta entre os robôs.

O terceiro nível, o nível de coordenação, diz respeito aos mecanismos usados para a coordenação. Seguindo a literatura sobre SMA, a coordenação é considerada (em (FARINELLI, 2004)) como cooperação quando as ações executadas por um agente robótico leva em conta as ações executadas pelos outros agentes robóticos, "de tal modo que o todo acaba sendo uma operação coerente de alto desempenho" (J. Ferber, *Multi-Agent Systems*,

Reading MA: Addison-Wesley, 1999, *apud* (FARINELLI, 2004)). Há diferentes modos, contudo, segundo os quais um robô pode levar em conta as ações dos outros membros da equipe, e o traço distintivo subjacente a estes diversos modos é o protocolo de coordenação[18]. Portanto, pode-se aprofundar a classificação dos SMR coordenados com base no tipo de protocolo de coordenação. Considera-se coordenação forte (fraca) a forma de coordenação que confia (não confia) em um protocolo de coordenação. Uma diferença em relação a SMA é que os enfoques baseados em coordenação fraca são mais comumente adotados em SMR, pois em robôs físicos o uso efetivo de um protocolo de coordenação pode ser difícil (FARINELLI, 2004).

O quarto nível da estrutura hierárquica da taxonomia, o nível de organização, diz respeito ao modo segundo o qual o sistema decisão é realizado no SMR. O nível de organização discrimina formas de coordenação, distinguindo enfoques centralizados dos distribuídos. A distribuição é vista como a autonomia de cada componente do sistema tomar decisões sobre as ações a executar (E. Durfee, V. Lesser e D. Corkill, "*Trends in cooperative distributed problem solving*", IEEE *Trans. Knowledge Data Eng.,* vol I, pp. 63-83, março 1989, *apud* (FARINELLI, 2004)). Em particular, um sistema centralizado tem um agente (líder) que comanda a organização do trabalho dos outros agentes; o líder envolve-se no processo de decisão para a equipe inteira, e os outros membros podem atuar somente de acordo com as instruções do líder. Por outro lado, um sistema distribuído é composto de agentes que são completamente autônomos no processo de decisão com respeito uns aos outros; nesta classe de sistemas, não existe líder. A classificação dos sistemas centralizados pode ser melhorada dependendo do modo pelo qual a liderança do grupo é exercida. Especificamente, a centralização forte é usada para caracterizar um sistema no qual as decisões são tomadas durante toda a duração da missão pelo mesmo agente previamente definido como líder. Em um sistema fracamente centralizado, mais de um agente tem permissão para exercer o papel de líder durante a missão.

Merece ser observado que as dimensões cooperação e conhecimento são atributos de agentes, enquanto as dimensões coordenação e organização dizem respeito às interações entre os agentes, mais especificamente, às interações de coordenação entre os agentes. Assim, a taxonomia estabelece as classes de coordenação que podem ocorrer e quando – para quais classes de agentes – elas ocorrem.

[18] Conjunto de regras que os robôs devem obedecer para interagir uns com os outros no ambiente.

Além das diversas formas de coordenação, há vários aspectos de sistema que são relevantes para o desenvolvimento de SMR[19] (FARINELLI, 2004). Tais aspectos estão agrupados em dimensões de sistema e incluem comunicação, composição de equipe, arquitetura de sistema e tamanho da equipe.

Comunicação: a cooperação entre robôs é freqüentemente conseguida por meio de um mecanismo de comunicação que permite aos robôs trocar mensagens. Os mecanismos de comunicação dos SMR são muito diferentes dos mecanismos de comunicação dos SMA: os SMR têm em geral menos do que dez robôs, chegando, em recentes projetos de larga-escala, a 100; em projetos de SMA de larga-escala, o número de agentes está com freqüência na faixa de 10000-100000. Portanto, embora seja possível caracterizar sistemas de comunicação levando em conta sua topologia, alcance e largura de banda; na taxonomia proposta em (FARINELLI, 2004), dois tipos diferentes de comunicação são considerados, dependendo do modo segundo o qual os robôs trocam informação: comunicação direta e indireta. A comunicação direta faz uso de algum dispositivo de *hardware* dedicado a "bordo" (*on board*), enquanto a comunicação indireta faz uso da *stigmergy* [20] [21] – *stigmergy* é um termo cunhado pelo biólogo P. Grasse que significa incitar ao trabalho pelo efeito de trabalho prévio (C. R. Kube e E. Bonabeau, "*Cooperative transport by ants and robots*", Robot Auton. Syst., *vol.* 30, *no*. 1, *pp*. 85-101, 2000, *apud* (FARINELLI, 2004)). Em (FARINELLI, 2004), e aqui, comunicação *stigmergic* refere-se ao compartilhamento de informação por meio de modificações no ambiente. O fato de em SMR a comunicação direta ser baseada em dispositivo físico dedicado resulta em solução de coordenação muito mais cara e precária do que a que se consegue em SMA; por isto, a comunicação indireta tem recebido especial atenção na literatura sobre SMR, pois sua adoção pode reduzir custos de implementação e projeto. Já em SMA, comunicação direta tem sido usada extensivamente, embora haja exceções notáveis que visam – com a comunicação indireta – garantir o caráter local das interações e evitar procedimentos de sincronização dos agentes.

[19] Embora as dimensões de sistema não se refiram a coordenação, auxiliam a modelagem de SMR. Além disto, os aspectos de sistema são tratados em (FARINELLI, 2004) juntamente com a taxonomia de coordenação. Por conveniência, então, as dimensões de sistema são tratadas neste item, e, não, em um item específico.

[20] Não se encontrou o correspondente em português do termo em inglês *stigmergy*, e, por isto, vai-se usar o vocábulo em inglês (consultou-se o dicionário de inglês-português dos irmãos Vallandro). Curiosamente, nem mesmo o Webster Unabridged, segunda edição, tem verbete para o termo.

[21] *Stigmergy* é um método de comunicação em sistemas emergentes nos quais as partes individuais do sistema comunicam-se umas com as outras por meio da modificação de seu ambiente local. *Stigmergy* foi observado primeiramente na natureza: por exemplo, formigas comunicam-se umas com as outras depositando feromônios ao longo de seus caminhos trilhados, assim uma colônia de formigas é um sistema *stigmergic*.

Composição de Equipe: dependendo da composição da equipe, os SMR podem ser divididos em duas classes principais: heterogêneos e homogêneos. Equipes homogêneas são compostas de membros de equipe que têm exatamente o mesmo *hardware* e *software* de controle, enquanto nas equipes heterogêneas tal não acontece.

Arquitetura de Sistema: a arquitetura de sistema é uma característica importante para a classificação de SMR bem como de SMA. A arquitetura de SMR pode ser deliberativa ou reativa. Na arquitetura de equipe deliberativa é permitido que os membros da equipe lidem com as mudanças ambientais, provendo uma estratégia para reorganizar os comportamentos de toda a equipe. Na arquitetura de equipe reativa, por outro lado, cada robô na equipe lida com as mudanças ambientais adotando um enfoque individual de reorganização de sua própria tarefa para alcançar a meta atribuída a ele. A principal diferença entre as arquiteturas de equipe reativa e deliberativa é o enfoque adotado pelo SMR para se recuperar de uma situação imprevista: em um SMR deliberativo, um plano de longo prazo envolvendo o uso de todos os recursos disponíveis para os robôs coletivamente cumprirem a meta global do sistema é suprido. Em um SMR reativo, um plano para lidar com o problema é fornecido ao agente robótico diretamente envolvido com a questão.

Tamanho da Equipe: o tamanho da equipe é um aspecto importante para MA e está se tornando um aspecto importante também no desenvolvimento de SMR: vários trabalhos recentes tratam explicitamente de SMR de larga escala. Apesar disto, o número de robôs atuando no mesmo ambiente é ainda bem limitado em comparação com o número de agentes em SMA. Portanto, ao invés de uma medida quantitativa do tamanho de SMR na taxonomia, distingue-se os enfoques que explicitamente considerem como escolha de projeto a oportunidade de lidar com um grande número de robôs.

#### 4.2.3.1 OBSERVAÇÕES FINAIS SOBRE COORDENAÇÃO

A coordenação dos agentes é um dos aspectos mais importantes no projeto de SMA e SMR, senão o mais importante. É a coordenação que transforma um aglomerado de agentes, ou robôs, em sistema com propósito e comportamento bem definidos. A maior relevância da interação entre as partes em comparação com as partes[22] é denotada pela usual padronização das partes em uns poucos tipos: isto acontece na engenharia eletrônica, na engenharia

[22] Subjacentemente, aqui, está-se adotando a seguinte definição geral de sistema: sistema é um conjunto de partes que interagem visando a consecução de um conjunto de objetivos.

mecânica, na engenharia civil, por exemplo. É de se esperar, portanto, que os robôs acabem, também, por ser padronizados em uns poucos tipos, e que a engenharia de SMA e de SMR acabem por se concentrar quase que totalmente no projeto de coordenação de agentes – tornando-se, ambas, engenharia de sistemas, ou de aplicação. Por outro lado, o tipo de coordenação de agentes é usualmente um requisito não-funcional, dependente muito mais de escolhas técnicas do projetista do que de necessidades que precisem ser atendidas pelo sistema: em geral, um sistema pode satisfazer seus requisitos funcionais – pode ser eficaz – adotando coordenação por PGP ou PGPG, por exemplo. A coordenação, qualquer que seja o tipo, é essencial aos SMA e SMR, mas a eficácia dos SMA e SMR não depende geralmente da adoção de um tipo particular de coordenação.

Uma vez escolhido o arcabouço de coordenação, cabe ao projetista estabelecer as interações de coordenação entre os agentes, necessárias e suficientes para que o sistema cumpra seus requisitos funcionais e operacionais – requisitos operacionais: custo operacional, confiabilidade, robustez, segurança, tempestividade etc.

Embora o tipo de coordenação não condicione, em geral, a eficácia do sistema, condiciona o alcance dos requisitos operacionais e a complexidade do sistema. A arte da escolha do tipo de coordenação mais adequado ao problema tem por suporte o conhecimento dos diversos tipos de coordenação praticados, visando a minimização da complexidade do sistema e o pleno atendimento de seus requisitos operacionais.

Um particular tipo de coordenação é, em essência, um padrão de interação. É natural que os tipos de coordenação mais usados sejam, com o tempo, refinados. Tais refinamentos transformarão os padrões em protocolos prontos para uso. Em cada protocolo estará embutido, na forma de procedimentos admissíveis/não-admissíveis, o *rationale* do tipo de coordenação que lhe é subjacente. Há já protocolos como, por exemplo, o *contract-net*, pronto para ser usado, que embute, durante uma fase da conversação, coordenação competitiva, e, uma vez resolvida a concorrência, coordenação cooperativa. As vantagens de se normalizar os tipos de coordenação por meio de protocolos tão amplamente aceitos quanto possível são, pelo menos, as seguintes: o uso de esquemas de conversações confiáveis – que implementem modalidades de coordenação – por serem largamente testados; a facilitação do intercâmbio de técnicas de coordenação, o que pode agregar ao sistema competências desenvolvidas alhures; o aumento da eficiência da atividade de projeto.

As duas visões de coordenação apresentadas, coordenação de SMA e coordenação de SMR, são úteis ao projeto do simulador e ao super-projeto: o simulador, um SMA, simula

uma frota de veículos aéreos robóticos, que, por sua vez, é um SMR. O projeto do simulador tem foco tanto em SMA, quanto em SMR, e o super-projeto, em SMR.

As divergências conceituais persistem nas duas visões de coordenação: em uma, o SMA abrange, também, robô – agente físico – na outra, diz respeito, apenas, a agente de *software*. Aqui, para simplificar, vai-se considerar que a expressão SMA refere-se tanto a sistemas compostos de agente de *software*, quanto a agente físico – robô – ou mesmo a sistemas compostos pelos dois simultaneamente.

# 5 A *MULTIAGENT SYSTEMS ENGINEERING* – MASE

A análise e o projeto do simulador, conforme mencionado anteriormente, foram desenvolvidos com o concurso extenso da técnica de engenharia de sistemas multiagentes denominada *Multiagent Systems Engineering* – MaSE. A aplicação da técnica é viabilizada pela ferramenta computacional denominada agentTool. Por estas razões, vai-se, a seguir, descrever a técnica e a ferramenta citadas, usando-se extensamente o artigo de Deloach (DELOACH, 2001b) e também (MATSON, 2002), (WOOD, 2000) e (DELOACH, 2001a).

A *Multiagent Systems Engineering* (MaSE) é um método alegadamente de ciclo de vida completo[23] para analisar, projetar e desenvolver sistemas multiagentes heterogêneos (DELOACH, 2001b). Assenta-se na abstração de sistemas multiagentes, e usa diversos modelos gráficos para descrever os tipos de agentes do sistema e suas interfaces com outros agentes, bem como uma definição detalhada independente de arquitetura do projeto interno do agente.

Tanto a MaSE quanto a agentTool estão em desenvolvimento. A MaSE e a versão corrente do agentTool têm sido usadas para o desenvolvimento em uma dúzia de sistemas multiagentes[24], variando de uns poucos agentes a cerca de cem. As áreas de aplicação vão de sistemas de informação a sistemas imunes de base biológica a trabalhos recentes sobre equipes de veículos aéreos inabitados.

Na MaSE, os agentes são considerados especializações de objetos[25]: diferentemente de simples objetos cujos métodos podem ser invocados por outros objetos, os agentes coordenam-se entre si por meio de conversações e atuam pró ativamente para cumprir metas individuais e de sistema. Na técnica, os agentes são vistos meramente como abstração conveniente, e podem, ou não, possuir inteligência. Portanto, a técnica trata igualmente componentes de sistema inteligentes e não inteligentes, dentro da mesma estrutura.

A técnica dispõe de ferramenta de apoio, a agentTool, uma ferramenta que trata os modelos gráficos da técnica – quase todos os modelos da técnica gráficos. Suporta os modelos gráficos dos sete passos das fases de análise e projeto da MaSE. A ferramenta permite a

[23] O ciclo de vida completo de um sistema inclui, além das fases de análise, projeto e desenvolvimento, as fases de construção, teste, implantação, operação e manutenção. Apesar do que se alega, a MaSE não cobre todas estas fases, assim, não é de ciclo completo.
[24] Isto em 2001.
[25] Objetos na acepção do paradigma de orientação a objeto

verificação automática de comunicações inter-agentes e a geração de código para estruturas de sistemas multiagentes múltiplas – não se conseguiu gerar código durante o projeto do simulador usando a agentTool.

O processo de projeto segundo a MaSE subdivide-se nas fases de Análise e Projeto. As fases compõem-se de passos. A Fase de Análise consiste em três passos: Capturando Metas, Aplicando Casos de Uso e Refinando Papéis. A Fase de Projeto tem quatro passos: Criando Classes de Agentes, Construindo Conversações, Montando Classes de Agentes e Projeto de Sistema, ver FIG. 5.1.

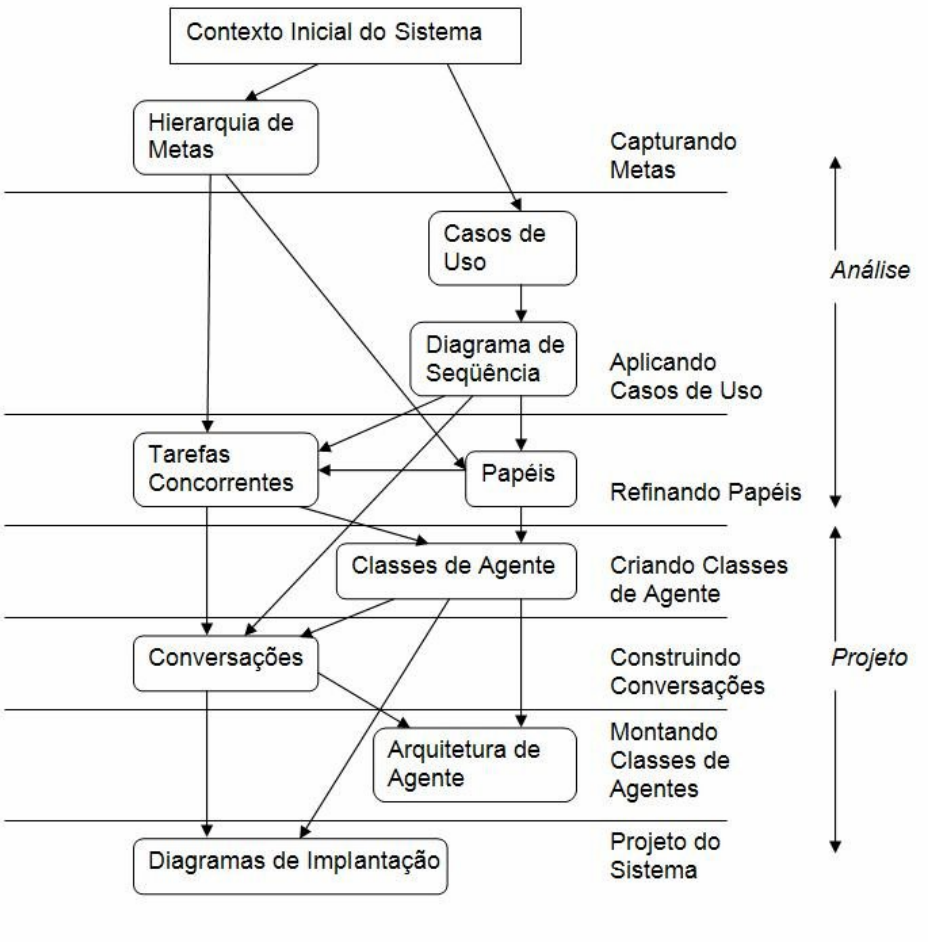


FIG 5. 1: Fases da MaSE (DELOACH, 2001b).

Na figura, os retângulos arredondados denotam os modelos MaSE usados para capturar a saída de cada passo, enquanto as setas entre eles mostram como os modelos afetam uns aos outros. Embora o desenho mostre um fluxo simples, de cima para baixo, com os modelos

criados em um passo, sendo entradas para os passos subseqüentes, na prática a método é iterativo. A intenção é que o analista ou projetista possa se mover livremente entre passos e fases tal que em cada passagem sucessiva sejam adicionados detalhes adicionais e que se chegue finalmente a um projeto de sistema consistente e completo.

A MaSE permite o acompanhamento das mudanças de análise e projeto durante o processo. Cada objeto criado durante as fases de análise e projeto pode seguido para frente ou para trás através de diferentes passos até outros objetos relacionados. Por exemplo, a partir de uma meta obtida no passo Capturando Metas pode-se seguir adiante pelos modelos até um papel, tarefa ou classe de agente criados para atendê-la. Igualmente, a partir de uma classe de agente pode-se seguir para trás, através de tarefas e papéis, até a meta em nível de sistema para cujo atendimento a classe de agente foi projetada.

## 5.1 A FASE DE ANÁLISE

O propósito da fase de Análise é produzir um conjunto de papéis[26] cujas tarefas descrevam o que o sistema tem de fazer para atender seus requisitos globais. Nesta fase, as metas de sistema[27] são inicialmente definidas a partir de um conjunto de requisitos funcionais, e, então, os papéis necessários para atender a estas metas são fixados.

Embora o mapeamento direto de metas em papéis seja possível, a técnica sugere que se use Casos de Uso[28] para ajudar a validar as metas de sistema e ajudar a deduzir o conjunto inicial de papéis.

Meta e caso de uso são conceitos básicos. Os modelos hierarquia de metas e casos de uso os usam fortemente e são os únicos derivados diretamente dos dados iniciais desestruturados do sistema. Todos os demais modelos da MaSE decorrem destes dois. Meta e caso de uso sustentam, portanto, toda a modelagem MaSE.

Dada a importância de meta e caso de uso para a modelagem, faz-se, a seguir, breve revisão destes conceitos, segundo a visão da MaSE.

[26] Ver 5.1.2.1.
[27] Ver 5.1.2.2.
[28] Ver 5.1.2.7.

### 5.1.1 META E CASO DE USO

Na MaSE, o conceito de meta é muito similar ao descrito por Cockburn (em *Structuring Use Cases with Goals* (COCKBURN, 1997)). Difere no seguinte ponto: ao invés do foco ser a meta do usuário do sistema, a MaSE a vê do ponto de vista do sistema. A MaSE considera, também, que a meta global do sistema é realizar os desejos do usuário. Portanto, se um usuário tem por meta "manter o rasto de possíveis violações de *login*", a meta – a meta do sistema – seria "informar ao usuário possíveis violações de *login*" (DELOACH, 2001b).

O trabalho de Cockburn trata de meta, e, também, de caso de uso. A MaSE tem os casos de uso como um de seus modelos – embora não na forma gráfica, uma exceção na MaSE. O conceito de caso de uso usado neste trabalho respeita a definição de meta da MaSE e atende, em boa parte, o estabelecido em (COCKBURN, 1997) – pois meta de sistema e caso de uso são conceitos fortemente interligados, e o conceito de meta da MaSE é semelhante ao de Cockburn, como foi mencionado antes. Assim, é conveniente rever-se a visão de Cockburn sobre caso de uso.

Segundo Cockburn, casos de uso são maravilhosos, mas confusos; pessoas, quando instadas a escrevê-los, não sabem o que pôr neles nem como estruturá-los; não há teoria publicada sobre eles (isto em 2000) – além, evidentemente, do trabalho do próprio Cockburn (pois a primeira versão do trabalho dele foi publicada em 1977). Ele, pessoalmente, como afirma, encontrou 18 definições diferentes de caso de uso, dadas por diferentes especialistas, professores e consultores (COCKBURN, 1997). As definições diferem ao longo de 4 dimensões: propósito, conteúdo, pluralidade e estrutura.

Propósito. O propósito dos casos de uso é colher histórias de usuário, ou construir requisitos? (os valores possíveis desta dimensão são histórias e requisitos).

Conteúdo. O conteúdo dos casos de uso devem ser consistentes, ou podem ser auto-contraditórios? Se consistentes, devem estar em prosa, ou em notação formal? (os valores são contraditório, prosa consistente e conteúdo formal).

Pluralidade. Um caso de uso é realmente somente um outro nome para uma seqüência, ou um caso de uso contém mais de um caso de uso? (os valores são 1 e múltiplos).

Estrutura. Uma coleção de casos de uso tem uma estrutura formal, uma estrutura informal ou eles formam uma coleção desestruturada? (os valores são desestruturado, semi-formal e estrutura formal).

A partir deste espaço de casos de uso, Cockburn define a sua seguinte versão de caso de uso:

Propósito = requisitos,

Conteúdo = prosa consistente,

Pluralidade = múltiplas seqüências por caso de uso,

Estrutura = semi-formal.

Esta versão de caso de uso é definida – definição provisória – como uma coleção de seqüências possíveis de interações entre o sistema sob discussão e seus atores externos. Esta coleção de seqüências está relacionada a uma meta particular. Na MaSE, o diagrama de seqüência é derivado diretamente dos casos de uso, o que denota adesão a esta interpretação de caso de uso como coleção de seqüências, embora o diagrama de seqüência use o conceito de papel (que não consta na definição de Cockburn).

A definição anterior menciona várias coisas. Atores: os atores podem ser pessoas ou sistemas computacionais. Sistema: o sistema é tratado como uma entidade única que interage com os atores. Demais termos: o caso de uso trata seqüências de interações e variações nestas seqüências.

Para aprofundar um pouco mais a definição, é necessário estabelecer um modelo de interação e comunicação. A FIG. 5.2 apresenta um modelo simples de comunicação ator-ator através de uma interação. Ela mostra que o sistema é em si mesmo um ator, assim, o modelo de comunicação trabalha somente com atores. Nesta figura, o ator central é o sistema sob discussão.

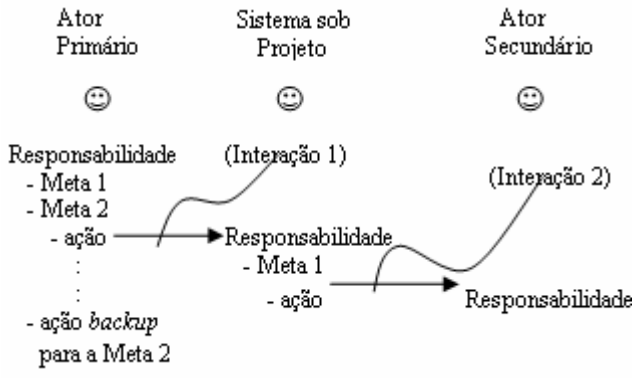


FIG 5. 2: Modelo de Comunicação (COCKBURN, 1997).

Um ator primário é aquele que tem meta que requer a assistência do sistema. Um ator secundário é aquele do qual o sistema precisa de assistência para satisfazer sua meta.

Cada ator tem um conjunto de responsabilidades. Para cumprir com sua responsabilidade, ele estabelece algumas metas. Para alcançar uma meta, ele executa algumas ações. A ação dispara uma interação com outro ator, invocando uma das responsabilidades do outro ator – uma ação conecta uma meta de um ator com a responsabilidade de outro. Se o outro ator cumprir a sua parte, então o ator primário ficará mais perto de alcançar sua meta. A interação é uma meta de ator invocando a responsabilidade de um outro ator (ou do próprio ator que fez a invocação) – a interação, portanto, é uma ação.

Se o segundo ator, por alguma razão, não cumprir com a sua responsabilidade, o ator primário terá de encontrar um outro modo de cumprir o seu objetivo. Isto é denotado na FIG 5.2 como "ação *backup*". Pessoas estão acostumadas com ações *backup*: parte de sua responsabilidade de trabalho é pensar em alternativas para alcançar uma meta, caso ocorra contratempo.

Uma interação, no seu caso mais simples, é o envio de uma mensagem: "Oi, João" e "Imprima(valor)" são exemplos. Uma interação pode ser também uma seqüência de interações, – esta definição é recursiva (FIG. 5.3) – e a interação no seu nível mais baixo consiste em mensagens.

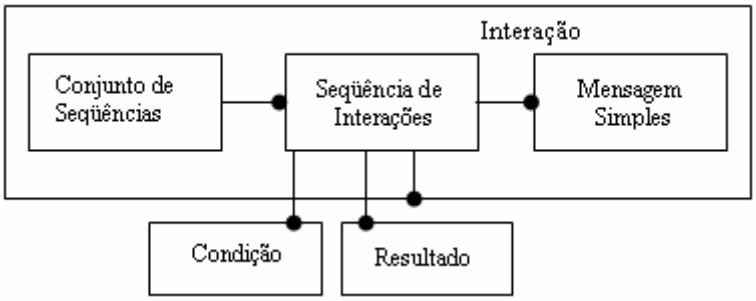


FIG 5. 3: Formas compostas de Interação (Diagrama E-R) (COCKBURN, 1997).

Uma seqüência de interações não tem ramificações ou alternativas. É usada, portanto, para descrever o passado ou o futuro definido, com condições fixas. Tal seqüência de interações é conhecida como *scenario*[29]. Assim, seqüência de interações é o mesmo que *scenario*.

[29] A palavra em inglês *scenario* significa originalmente sinopse de peça teatral, ou seqüência de cenas de um filme. No trabalho de Cockburn, *sequence of interactions* é o mesmo que *scenario*. Não há em português uma expressão de uma só palavra que corresponda a *scenario* – cenário não é a tradução de *scenario*. Por esta razão, vai-se manter no texto a palavra *scenario* em inglês, grafada em itálico.

Para descrever um sistema, precisa-se coletar e agrupar todos os *scenarios* que podem ocorrer durante uma interação. Uma coleção de *scenarios* é um caso de uso. Um caso de uso responde questões como, por exemplo, "Como pegar dinheiro daquele caixa automático? " Neste exemplo, o ator primário tem uma meta, pegar dinheiro do sistema, que é o caixa automático. Há várias situações nesse exemplo em que o ator primário poderia se encontrar. O caso de uso reúne-as em um único lugar. Ele decompõe a meta do ator primário em submetas, posteriormente em ações consistindo em mensagens individuais, mais as interações entre os vários atores ao tentarem alcançar aquela meta, ou eventualmente falharem em alcançá-la – *scenarios* e casos de uso seguem até terem sucesso, ou desistirem da meta.

Como saber quando parar de escrever um *scenario*, ou um caso de uso? Há duas cláusulas que delimitam caso de uso e *scenario*: (1) todas as interações têm de se referir à mesma meta. (2) As interações têm de começar com o evento gatilho e terminar quando a meta for cumprida ou abandonada e o sistema completar suas responsabilidades com respeito à interação.

A cláusula 2 leva à frente a noção de responsabilidade com respeito à interação. Muitos sistemas têm de registrar cada interação com cliente (e.g., sistemas bancários). Se a cláusula não incluísse o requisito de completeza da responsabilidade, o *scenario* terminaria sem que a responsabilidade tivesse sido tratada – gravação do registro de interação como o cliente. Liberar uma alça de arquivo é outro exemplo da necessidade desta parte da cláusula.

As definições consolidadas de *scenario* e caso de uso, segundo Cockburn, que resumem o que foi visto, são as seguintes: *Scenario*. Uma seqüência de interações acontecendo sob certas condições para consumar a meta do ator primário, tendo um particular resultado com respeito a esta meta. As interações começam com a ação gatilho e continuam até a meta ser cumprida ou abandonada, e o sistema completar toda e qualquer responsabilidade que tenha com respeito à interação. Caso de Uso. Uma coleção de *scenarios* possíveis entre o sistema sob discussão e atores externos, caracterizada pela meta que o ator primário tem através das responsabilidades declaradas do sistema, mostrando como o sistema poderia cumprir a meta do ator primário ou falhar.

Como se vê, Cockburn vincula fortemente caso de uso – e *scenario* – à meta – o que também é feito na MaSE.

A informação que caracteristicamente um caso de uso contém é a seguinte: (1) ator primário, ou atores; (2) meta; (3) *scenarios* usados.

A informação característica de um *scenario* é a seguinte: (1) ator primário, (2) meta, (3) condições sob as quais o *scenario* ocorre e (4) resultado do *scenario* (cumprimento da meta, ou falha).

Exemplo de *scenario*:

Sistema sob discussão: a companhia de seguro

Ator primário: o reclamante

Meta: ser pago pelo acidente de carro

Condições: tudo está em ordem

Resultado: a companhia de seguro paga a reclamação

1. O reclamante submete a reclamação com dados comprobatórios
2. A companhia de seguro verifica se o reclamante possui uma apólice de seguro válida (uma falha aqui significa provavelmente falha na consecução da meta)
3. A companhia de segura designa um agente para examinar o caso
4. O agente verifica se todos os detalhes do acidente estão cobertos pela apólice (uma interação entre o agente e atores secundários)
5. A companhia de seguro paga a reclamação (implica que todas as metas conseguiram ter sucesso)

Os casos de uso são compostos por *scenarios*. Dependendo de como os *scenarios* sejam escritos, a quantidade de *scenarios* pode se tornar excessiva. Para evitar o excesso, três técnicas podem ser usadas: casos de uso subordinados, extensões e variações.

Variações é uma seção do texto do caso de uso que tira proveito da estruturação semi-formal intra e entre casos de uso. Freqüentemente, um caso de uso em nível alto usa entrada ou saída de um de diversos tipos. Um exemplo é o pagamento em dinheiro, cheque, crédito, cartão de crédito, ou transferência eletrônica de fundos. As diferenças entre todas estas possibilidades terão de ser descritas em algum ponto em um nível mais baixo, mas seria desperdício gastar tempo fazendo isto nos níveis mais altos.

Usa-se uma entrada (*place holder*), chamada Variações, para rastrear nos níveis mais altos variações que, se sabe, ocorrerão. Prosseguindo com o exemplo mencionado, isto é melhor do que criar miríades de casos de uso de alto-nível para todas as diferenças, ou um meio de entrada ou saída falso que finge ser qualquer um dos meios reais.

Os passos nos *scenarios* são na realidade metas intermediárias dos atores. Cada meta é alcançada por uma seqüência de passos ou interações que podem ser bem sucedidas, ou falhar

(ou, de fato, serem parcialmente bem sucedidas). Esta composição hierárquica de metas e passos forma uma pequena álgebra consistente de composição (que é desenvolvida em (COCKBURN, 1997)). A estrutura ajuda a evitar a explosão de *scenarios* que ocorreria se a listagem pura e simples de todos os possíveis modos de interação com o sistema fosse tentada.

Um passo em um *scenario* é escrito como se ele trabalhasse, e.g., "Verifique se todos os detalhes do acidente estão cobertos pela apólice". Isto permite que o *scenario* seja escrito em um estilo muito legível, com a situação mais direta descrita simples e facilmente. O passo é uma meta que pode falhar.

A falha de um passo é manipulada por outro *scenario*, ou extensão de *scenario*. Muitas técnicas de estruturação podem ser usadas para combinar *scenarios* principais e extensões. Cockburn gosta de escrever fragmentos de *scenarios* como extensões para outros *scenarios*, para economizar escrita e leitura.

Um caso de uso coleciona todos os vários *scenarios* de interesse. O exemplo a seguir mostra um caso de uso completo. O caso de uso "Pegar o pagamento pelo acidente do meu carro", que contém o *scenario* "Tudo está em ordem", também terá de conter o *scenario* "Acidente fora da cobertura da apólice". Este *scenario* produzirá ou um caminho alternativo de sucesso, ou uma falha no alcance da meta.

Exemplo de caso de uso:

Sistema sob discussão: a companhia de seguro

Ator Primário: o reclamante

Meta: obter pagamento por acidente de carro

1. O reclamante submete a reclamação com dados comprobatórios
2. A companhia de seguro verifica se o reclamante possui uma apólice válida
3. A companhia de seguro designa um agente para examinar o caso
4. O agente verifica se todos os detalhes são cobertos pela apólice
5. A companhia de seguro paga ao reclamante

Extensões:

1a. Os dados submetidos estão incompletos:

1a1. A companhia de seguro requisita a informação faltante

1a2. O reclamante fornece a informação faltante

2a. O reclamante não possui uma apólice válida:

2a1. A companhia de seguro recusa a reclamação, notifica o reclamante, registra tudo e termina os procedimentos

3a. Nenhum agente está disponível no momento

3a1. (O que a companhia de seguro faz aqui?)

4a. O acidente não é coberto pela apólice

4a1. A companhia de seguro recusa a reclamação, notifica o reclamante, registra tudo e termina os procedimentos.

4b. O acidente viola a apólice em aspectos menores:

4b1. A companhia de seguro inicia negociações com o reclamante sobre o grau de pagamento a ser efetuado.

Variações:

1. O reclamante é
    a. uma pessoa
    b. uma outra companhia de seguro
    c. o governo
2. O pagamento é
    a. em cheque
    b. por transferência interbancária
    c. por pré-pagamento automático da próxima prestação
    d. pela criação e pagamento de uma outra apólice.

#### 5.1.1.1 APLICAÇÃO DE CASO DE USO NA MASE

Na modelagem do simulador, considerou-se que o caso de uso é visto a partir do sistema, e não do ator externo. Isto compatibiliza a visão de caso de uso com a visão de meta, que, como foi mencionado, é tomada a partir do sistema.

O modelo de caso de uso de Cockburn foi ligeiramente alterado para uso neste trabalho: adotou-se um modelo híbrido, síntese do de Cockburn e do de Guedes (GILLEANES, 2004).

Apesar de não ser previsto na MaSE, usou-se também o diagrama de casos de uso na modelagem do simulador. O diagrama de casos de uso (modificado para apresentar os casos de uso sob a ótica do sistema) permite uma visão abrangente do sistema.

### 5.1.2 CONCEITOS BÁSICOS DA MASE

Há alguns conceitos nos quais os modelos da MaSE se apóiam. Os principais são tratados nos itens a seguir.

#### 5.1.2.1 PAPEL (“ROLE”)

Um papel descreve uma entidade que desempenha alguma função dentro do sistema. Cada papel é responsável por cumprir ou ajudar a cumprir metas de sistema específicas, ou submetas. Os papéis são, assim, orientados por meta, e, por isto, os requisitos funcionais nesta técnica são abstraídos como um conjunto de metas de sistema que podem ser passadas a papéis individuais para serem por estes cumpridas. Os papéis da MaSE são análogos aos papéis representados por atores em uma peça, ou por membros de uma estrutura empresarial típica. A empresa (que corresponde ao sistema) tem papéis como “presidente”, “vice-presidente”, “encarregado de correspondência” que têm específicos direitos, responsabilidades e relações definidos para atender a meta global da empresa.

#### 5.1.2.2 META (“GOAL”)

Uma meta é a abstração de um conjunto de requisitos funcionais. Tipicamente, um sistema tem uma meta global e um conjunto de submetas que devem ser levadas a cabo para que a meta global do sistema seja alcançada. Metas consubstanciam o que o sistema esta intentando realizar, e, geralmente, permanecem constantes ao longo dos processos de projeto e análise.

#### 5.1.2.3 ATOR

Um ator representa um conjunto coerente de papéis que os usuários do sistema desempenham quando interagem com o sistema.

#### 5.1.2.4 ATOR PRIMÁRIO

Ator primário é o ator que tem uma meta que requer a colaboração do sistema para ser alcançada.

#### 5.1.2.5 ATOR SECUNDÁRIO

É o ator do qual o sistema necessita de colaboração para satisfazer sua meta.

#### 5.1.2.6 *SCENARIO*

É uma seqüência de interações ocorrendo sob condições específicas, visando a realização da meta do ator primário e tendo um resultado particular relacionado àquela meta.

#### 5.1.2.7 CASO DE USO

Uma coleção de *scenarios* possíveis entre o sistema sob discussão e atores externos, caracterizada pela meta que o ator primário tem através das responsabilidades declaradas do sistema, mostrando como o sistema poderia cumprir a meta do ator primário ou falhar.

### 5.1.3 OS PASSOS DA FASE DE ANÁLISE

A fase de Análise (e também de Projeto) é composta de passos, como já foi mencionado. Os passos da fase de análise, Capturando Metas, Aplicando Casos de Uso e Refinando Papéis são detalhados a seguir.

#### 5.1.3.1 CAPTURANDO METAS

Este primeiro passo da fase de Análise tem por objetivo transformar uma especificação inicial do sistema em um conjunto estruturado de metas de sistema. A técnica supõe que a especificação inicial do sistema inclua uma especificação de requisitos de *software* com um conjunto bem-definido de requisitos funcionais – requisitos funcionais informam ao analista os serviços que o sistema deve oferecer e como ele deveria, ou não, funcionar baseado nas

entradas e em seu estado corrente (SOMMERVILLE, 2001). A existência de conjunto bem-definido de requisitos funcionais na especificação inicial do sistema, o que a MaSE espera ocorrer, é pouco freqüente. Na prática, ao menos nos projetos de sistemas convencionais, os requisitos funcionais acabam sendo refinados e definidos adequadamente durante a fase de análise.

A fase de análise da MaSE baseia-se nas metas porque metas de sistema são mais estáveis do que funções, processos e estruturas de informação que frequentemente mudam com o tempo (Kendall et al.*Capturing and Structuring Goals: Analysis Patterns, Proceedings of the Third European Conference on Pattern Languages of Programming and Computing, Bad Irsee, Germany, July 1998, apud* (DELOACH, 2001b)). Metas incorporam o que o sistema está tentando alcançar e geralmente permanecem constantes durante o processo de análise e projeto.

Há dois sub-passos em Capturando Metas: Identificar as Metas e Estruturar as Metas. Primeiramente, as metas devem ser identificadas do contexto de sistema inicial. A seguir, as metas são analisadas e estruturadas.

#### 5.1.3.1.1 IDENTIFICANDO METAS

Neste primeiro sub-passo, pretende-se capturar a essência de um conjunto inicial de requisitos funcionais. O processo começa pela extração de *scenarios* da especificação inicial do sistema e pela descrição da meta daquele *scenario*.

O propósito de usar metas é identificar os aspectos críticos dos requisitos de sistema. Portanto, as metas devem ser especificadas tão abstratamente quanto possível, sem perda da essência do requisito.

As metas oferecem o fundamento do modelo de análise. Todos os papéis e tarefas definidos nos passos posteriores devem suportar pelo menos uma das metas identificadas neste passo.

Ao longo deste texto, um exemplo de análise e projeto de sistema é desenvolvido. Os requisitos funcionais do sistema exemplo são os seguintes:

- O sistema é responsável por lidar com violações de *host*, em particular violações de *login* e intrusões em arquivos de sistema. O administrador de sistema é notificado das intrusões tentadas ou suspeitas.

- É necessário validar a data, hora e existência dos arquivos de sistema periodicamente, de poucos em poucos minutos. Quando um arquivo não é encontrado ou uma nova versão aparece, o administrador do sistema precisa ser notificado.
- Um usuário tenta se conectar (fazer *login*) quando não tem uma conta válida. Se isto ocorrer uma ou duas vezes em um período curto de tempo, não é uma violação. Três ou mais tentativas são uma violação que precisa ser relatada.
- O administrador do sistema pode, ou não, estar disponível para receber uma notificação. Isto pode ser devido a uma falha de rede ou ao fato de o administrador estar executando uma outra tarefa. O relatório precisa ser armazenado e re-enviado após um retardo.

Um exemplo de metas derivadas deste conjunto de requisitos é o seguinte:

1. Informe ao administrador as violações de arquivo.
2. Informe ao administrador as violações de *login.*
3. Detecte tentativas de exclusão de arquivo inválidas.
4. Detecte tentativas de modificação de arquivo inválidas.
5. Detecte tentativas de *login* inválidas.
6. Notifique ao administrador as violações.

Observe-se que nenhum detalhe de como executar funções do sistema é incluído como meta (e.g., "É necessário validar a data, a hora e a existência dos arquivos do sistema periodicamente, de poucos em poucos minutos", um detalhe funcional, não é considerado meta).

#### 5.1.3.1.2 ESTRUTURANDO METAS

O passo final em Capturando Metas é a estruturação das metas em um Diagrama de Hierarquia de Meta. Um Diagrama de Hierarquia de Meta é um grafo orientado acíclico no qual os nós representam metas e os arcos definem um relacionamento de submeta. Uma hierarquia de meta não é uma árvore, pois uma meta pode ser submeta de mais de uma meta pai. As metas em um mesmo nível no diagrama, metas pares, estão aproximadamente no mesmo nível de detalhe.

As metas capturadas têm níveis diferentes de importância, tamanho e detalhe; se inter-relacionam e têm graus de importância variados. O Diagrama de Hierarquia de Meta preserva

tais relações e divide as metas em níveis de detalhe e importância, facilitando a compreensão e a administração delas.

A FIG. 5.4 mostra o resultado da estruturação das metas derivadas do conjunto de requisitos do exemplo.

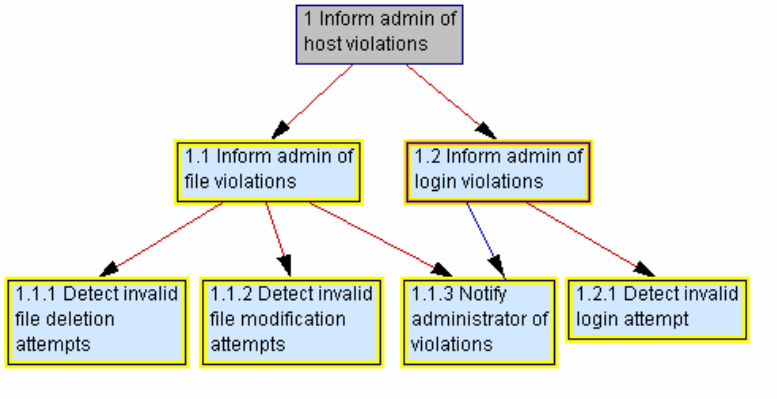


FIG 5. 4: Exemplo MaSE de Diagrama Hierárquico de Metas (DELOACH, 2001b).

Decomposição de meta não é simplesmente "decomposição funcional". A decomposição funcional resulta em um conjunto de passos para alcançar uma meta (DELOACH, 2001b). Por exemplo, os passos necessários para implementar a meta "Informar ao administrador as violações de *login*" são (1) detectar todos os *logins*, (2) determinar se eles são válidos e (3) enviar uma mensagem ao administrador para cada *login* inválido. Por outro lado, metas válidas são, dentre outras possíveis, "Detectar tentativas de *login* inválidas" e "Notificar o administrador sobre as violações". Os fatos, um *login* inválido é detectado e o administrador é notificado, são metas, enquanto como se detecta *logins* inválidos e notifica o administrador não são. A decomposição de meta continua até que qualquer decomposição adicional resulte em um requisito funcional em vez de uma meta (isto é, o analista começa a captar **como** a meta seria cumprida) (DELOACH, 2001b)[30].

Há quatro tipos especiais de metas em um Diagrama de Hierarquia de Meta: sumária, repartida, combinada e não funcional.

Uma *meta sumária* é obtida de um conjunto de metas pares, visando o estabelecimento de uma meta pai comum a todas as metas do conjunto. Isto freqüentemente acontece nos níveis mais altos da hierarquia. Por exemplo, se as metas "Informe ao administrador as violações de arquivo" e "Informe ao administrador as violações de *login*" constituírem o nível mais alto de metas do sistema, então tais metas poderão ser abstraídas pela criação de uma meta sumária

[30] Há outras interpretações de requisito funcional na literatura. Neste trabalho, quando necessário, a interpretação usada é explicitada.

"Informe ao administrador as violações ao *host*", que será a meta global do sistema, como mostrado na FIG 5.4.

Algumas metas não suportam diretamente a meta global do sistema, mas são críticas para o correto funcionamento dele. Estas *metas não-funcionais* são frequentemente obtidas de requisitos não-funcionais tais como confiabilidade ou tempos de resposta. Neste caso, um outro ramo no Diagrama de Hierarquia de Meta pode ser criado e posto sob a meta de nível global do sistema.

Há frequentemente um número de submetas na hierarquia que são idênticas ou muito similares podendo ser agrupadas em uma *meta combinada*. Por exemplo, as metas iniciais "Informe ao administrador as violações de arquivos" e "Informe ao administrador as violações de *login*" são combinadas na Figura 7 em uma única meta "Notifique ao administrador as violações". Neste caso, a meta combinada torna-se uma submeta das metas "Informe ao administrador as violações de arquivos" e "Informe ao administrador as violações de *login*".

Uma *meta repartida* é uma meta com um conjunto de submetas que, quando tomadas coletivamente, atendem efetivamente àquela meta. Em essência, as submetas devem cooperar para alcançar a meta de seu pai. Embora isto seja sempre verdadeiro para metas sumárias, pode ser verdadeiro para quaisquer metas com um conjunto de submetas. Definir uma meta como "repartida", libera o analista de considerá-la especificamente no resto do processo de análise, pois basta considerar suas submetas. Metas repartidas são representadas no Diagrama de Hierarquia de Meta com uma caixa de meta cinza em vez de uma caixa limpa. Por exemplo, na FIG. 5.4, a Meta 1 é uma meta repartida, pois é uma meta sumária.

#### 5.1.3.2 APLICANDO CASOS DE USO

O objetivo do passo Aplicando Casos de Uso é captar o conjunto de casos de uso a partir do contexto de sistema inicial e criar um conjunto de Diagramas de Seqüência para ajudar a identificar um conjunto inicial de papéis e de caminhos de comunicações dentro do sistema. Os casos de uso definem *scenarios* básicos que o sistema deveria ser capaz de executar. Os diagramas de seqüência capturam os casos de uso como um conjunto de eventos entre os papéis que compõem o sistema. Estas seqüências de eventos são usadas mais tarde na fase de Análise para definir tarefas que um papel em particular deve cumprir. Tais tarefas acham

finalmente seu espaço nas conversações inter-agentes durante a fase de Projeto, assegurando assim que os casos de uso sejam implementados no sistema multiagente resultante.

#### 5.1.3.2.1 CRIANDO CASOS DE USO

O primeiro passo em Aplicando Casos de Uso é extrair os casos de uso do contexto inicial de sistema. Casos de uso definem seqüência de eventos que podem ocorrer no sistema. Eles são exemplos de como o usuário pensa que o sistema deveria se comportar. Embora sendo parte do passo Aplicando Casos de Uso, Criando Casos de Uso pode na prática extrair mais informação ou clarificar a informação existente sobre as metas do sistema. Se isto acontecer, o analista deve voltar atrás imediatamente na técnica e fazer a adição da nova meta, ou modificar o Diagrama de Hierarquia de Meta original.

Os casos de uso podem já existir como parte do contexto de sistema inicial ou podem ter sido extraídos pelo analista. Os casos de uso podem ser extraídos das especificações de requisitos, histórias de usuários, ou quaisquer outras fontes disponíveis. Embora a existência de um grande número de casos de uso possa ajudar a compreender o sistema, é importante não deixar a criação dos casos de uso sair de controle [31]. O propósito da criação de casos de uso é o de identificar caminhos de comunicação, e não o de definir todas as possíveis combinações de eventos e dados, no sistema.

Em geral, os casos de uso deveriam servir para mostrar como cada meta pode ser alcançada. Deveriam ser capturados tanto os casos de uso positivos quanto os negativos. Um caso de uso positivo descreve o que deveria acontecer durante uma operação normal. Contudo, um caso de uso negativo ainda descreve uma seqüência desejada de eventos, mas ilustra um erro ou uma pane.

Embora os casos de uso não possam ser usados para capturar todos os requisitos possíveis, são de valia na dedução dos caminhos de comunicação e dos papéis. A verificação cruzada da análise final contra o conjunto de metas deduzidas e os casos de uso é um método redundante de obtenção do comportamento de sistema requerido.

[31] Aqui, os casos de uso, se for necessário, terão tratamento hierárquico para administrar sua complexidade. Para tanto se utilizará relacionamentos de especialização/generalização, extensão etc., e diagramas distintos.

#### 5.1.3.2.2 CRIANDO DIAGRAMAS DE SEQÜÊNCIA

Um diagrama de seqüência descreve a seqüência de eventos que são transmitidos entre papéis (DELOACH, 2001b)[32], identificados a partir dos casos de uso, como mostrado na FIG. 5.5. As caixas no topo do diagrama representam papéis de sistema e as setas entre as linhas representam eventos (mensagens) passados entre os papéis. O tempo é suposto fluir do topo do diagrama para baixo[33].

Na FIG. 5.5, o evento (mensagem) ViolaçãoArquivo é enviado do papel DetectorArquivoModificado para o papel NotificadorArquivo e deve preceder o evento (mensagem) NotificaçãoPedido que é enviado ao NotificadorAdmin. A transformação dos casos de uso para os Diagramas de Seqüência é relativamente direta. Entidades individuais nomeadas no caso de uso correspondem a papéis enquanto qualquer tipo de comunicação ou informação trocada entre entidades de caso de uso torna-se um evento. A seqüência de eventos está baseada na descrição de casos de uso. Qualquer tipo de participante em um Diagrama de Seqüência torna-se um papel. Os papéis identificados em Diagramas de Seqüência formam o conjunto inicial de papéis usado no passo seguinte, Refinando Papéis, onde eles podem ser renomeados, decompostos em múltiplos papéis ou combinados com outros papéis.

Em geral, um Diagrama de Seqüência é criado para cada caso de uso. Contudo, se houver diversas seqüências de execução possíveis, múltiplos Diagramas de Seqüência podem ser criados. Por exemplo, se um caso de uso tem diversas resoluções alternativas, tal como "o diagnóstico é enviado do médico a escrivaninha médica e da escrivaninha médica ao paciente, a menos que o paciente seja menor, caso em que é enviado da escrivaninha médica ao guardião legal do paciente", então o analista deveria criar dois diagramas de seqüência similares, mas distintos, para definir o caso de uso. O primeiro diagrama de seqüência deveria ser usado para descrever o que acontece quando o paciente é de menoridade e o segundo, o caso mais freqüente.

Após identificar os papéis participantes, criar o Diagrama de Seqüência consiste em ler o caso de uso e encontrar todas as instâncias de eventos que ocorrem entre dois dos papéis. Os eventos no caso de uso são desenhados como uma seta no Diagrama de Seqüência na ordem

[32] Em geral, na literatura, o termo evento é interpretado como mudança de estado de sistema, não tendo assim, inerentemente, a propriedade de transmissão. Considera-se aqui que haja mensagens associadas aos eventos disparadas automaticamente quando eles ocorrem.
[33] Portanto, o envio das mensagens é instantâneo.

em que ocorrem. Pela aplicação de casos de uso na criação de Diagramas de Seqüência, as seqüências principais de eventos dos casos de uso são explicitamente consideradas nas tarefas concorrentes e conversações projetadas a partir destes casos de uso.

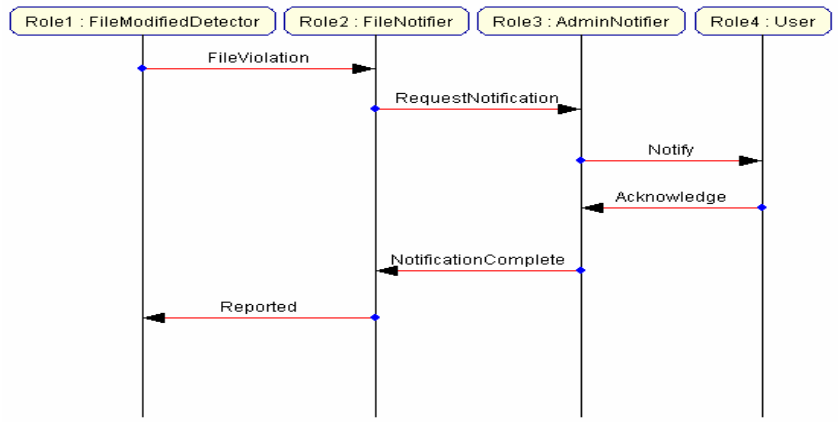


FIG 5. 5: Exemplo de Diagrama de Seqüência (DELOACH, 2001b).

### 5.1.3.3 REFINANDO PAPÉIS

O objetivo do último passo na fase de Análise, o passo Refinando Papéis, é o de transformar metas estruturadas e diagramas de seqüência em papéis e suas tarefas associadas, que são formas mais adequadas para o projeto de sistemas multiagentes. Papéis formam o fundamento da definição de classe de agente e representam as metas de sistema durante a fase de Projeto. Usando papéis desta maneira, as metas de sistema são levadas ao projeto do sistema. A MaSE postula que as metas de sistema serão satisfeitas se cada meta for associada a um papel e cada papel for representado em uma classe de agente.

O caso geral da transformação de metas em papéis é um-para-um, com cada meta mapeada em um papel. Contudo, há situações em que é útil haver um único papel responsável por múltiplas metas.

Metas similares ou relacionadas podem ser combinadas em um único papel no interesse da conveniência ou eficiência. Metas corriqueiras frequentemente implicam que papéis desenvolvidos em esforços prévios possam ser reusados. Por exemplo, no caso onde um sistema precisa encontrar recursos dinamicamente, algum tipo de papel facilitador pode ser necessário. Papéis facilitadores são bastante comuns e têm sido incluídos em muitos sistemas multiagentes.

Um mapeamento das metas da FIG. 5.4 em um conjunto de papéis é mostrado a seguir:

| | |
|---|---|
| NotificadorArquivo | (1.1) |
| NotificadorLogin | (1.2) |
| DetectorExclusãoArquivo | (1.1.1) |
| DetectorModificaçãoArquivo | (1.1.2) |
| NotificadorAdmin | (1.1.3) |
| DetectorLogin | (1.2.1) |

Devido à simplicidade do exemplo, mapeou-se as metas em papéis individuais com uma única exceção: a meta 1 não foi mapeada em um papel, pois foi repartida nas metas (1.1) e (1.2).

Algumas metas podem não ter sido declaradas nos requisitos e permanecer sem serem descobertas até este ponto da análise. Por exemplo, a interface com o usuário é um requisito freqüentemente não visto. Como a interface de usuário requer técnicas de projeto especiais, a interface deveria estar em um papel separado. Se uma meta é descoberta neste ponto da análise do sistema, deve ser juntada às metas existentes, como se fosse parte dos requisitos originais do sistema. Os passos prévios, tais como aditar a nova meta ao Diagrama de Hierarquia de Meta, são então re-executados para manter a análise do sistema consistente.

Metas inter-relacionadas podem com freqüência ser combinadas em um único papel. Por exemplo, se as metas tivessem sido decompostas em “Notificar o administrador das violações de arquivo” e “Notificar o administrador das violações de *login*”, poder-se-ia combinar os correspondentes papéis em um único papel, como por exemplo, em NotificadorAdmin. Embora o papel combinado seja mais complexo do que os originais, a combinação de metas em um único papel simplifica o projeto do sistema como um todo. Esta questão é uma solução de compromisso sobre a qual o analista deve decidir.

Estabelecer interface com recursos externos ou internos requer geralmente um papel separado para atuar como uma interface entre o recurso e o resto do sistema. Geralmente, considera-se o usuário humano como um recurso externo. Na MaSE não se modela explicitamente a interação humano-computador. Contudo, sugere-se que um papel específico seja criado para encapsular a interface de usuário. Desta maneira, pode-se definir os modos segundo os quais um usuário se relaciona com o sistema sem que se defina a interface de usuário em si mesma. Outros recursos tais como bases de dados arquivos ou sistemas antigos herdados podem também requerer seu próprio papel interface.

Definições de Papéis são capturadas em um Modelo de Papel MaSE, como mostrado na FIG. 5.6, o qual inclui informação sobre interações entre tarefas de papel e que é mais complexo do que os modelos de papel tradicionais.

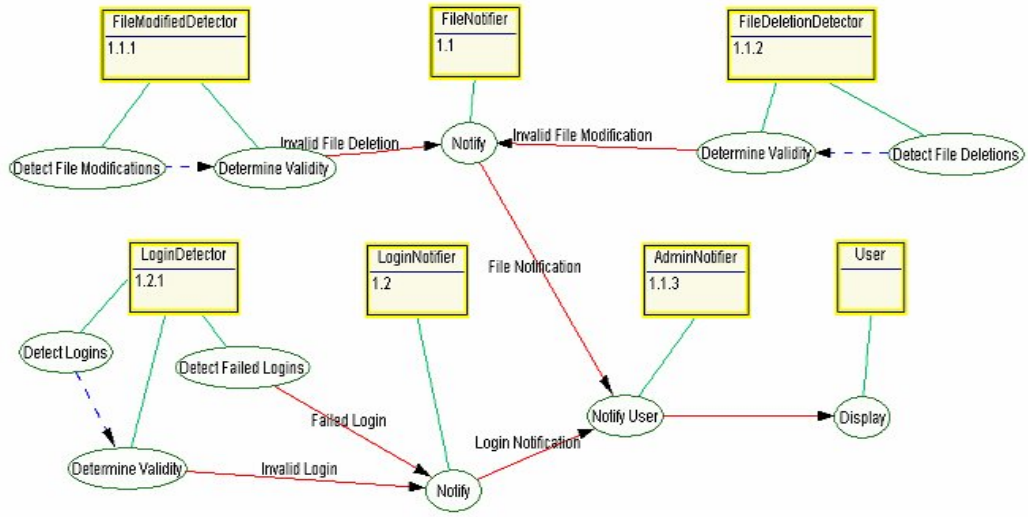


FIG 5. 6: Modelo de Papel MaSE (DELOACH, 2001b).

Papéis são denotados por retângulos, enquanto tarefas de papéis são denotadas por elipses conectadas ao papel. As linhas entre tarefas denotam protocolos de comunicações (possivelmente nomeados) que ocorrem entre as tarefas. As setas denotam o relacionamento iniciador/respondedor do protocolo com a seta apontando do iniciador para o respondedor. Linhas sólidas indicam comunicações par-a-par, que são geralmente implementadas como protocolos de comunicação externos. Protocolos externos envolvem passagem de mensagem entre papéis que podem se tornar mensagens de fato se estes papéis terminarem sendo implementados em agentes separados. Estes protocolos são derivados dos diagramas de seqüência desenvolvidos no passo anterior. Linhas pontilhadas denotam comunicação entre tarefas concorrentes dentro do mesmo papel. Uma linha é pontilhada se seus protocolos denotam comunicações ocorrendo somente dentro da mesma instância do papel.

As tarefas são geralmente derivadas das metas pelas quais um papel é responsável. Por exemplo, o papel DetectorExclusãoArquivo é responsável pelo alcance da meta (1.1.1), nomeadamente, “Detectar tentativas de exclusão de arquivo inválidas”. Portanto, para alcançar essa meta, papel deve ser capaz de detectar tentativas de exclusão e determinar se elas são válidas. Neste caso, o projetista decidiu criar duas tarefas: Detecte Exclusões de Arquivo e Determine a Validade.

Papéis não podem compartilhar ou duplicar tarefas. Compartilhamento de tarefas é sinal de decomposição imprópria de papel. Tarefas compartilhadas deveriam ser postas em um papel separado, que pode ser combinado em várias classes de agentes na fase de Projeto. Isto não implica que a noção mais geral de tarefa não possa ser distribuída entre vários agentes no sistema. Um agente encarregado de satisfazer uma meta pode distribuir tarefas entre vários agentes capazes de desempenhar o papel apropriado.

#### 5.1.3.3.1 DIAGRAMA DE TAREFA CONCORRENTE

Após os papéis terem sido criados, tarefas são associadas a cada papel. Tais tarefas descrevem o comportamento que o papel deve exibir para ser bem-sucedido no alcance de suas metas. Em geral, um único papel pode ter múltiplas tarefas sendo executadas concorrentemente, e estas tarefas definem o comportamento requerido do papel. Cada tarefa especifica um único *thread* de controle e define um comportamento particular que o papel pode exibir. Este comportamento abrange as inter-interações, bem como as intra-interações do papel. Tarefas concorrentes são especificadas graficamente usando um autômato de estado finito, o qual será chamado de Diagrama de Tarefa Concorrente, como mostrado na Figura 10.

Há dois tipos de tarefas: persistente e transiente. Uma tarefa *persistente* é uma tarefa que tem transição nula do estado de partida para o primeiro estado. Em outras palavras, a tarefa não tem um evento que inicie sua execução. Assumimos que tarefas persistentes comecem quando o agente é iniciado e continuem a ser executadas até o agente em si mesmo ser encerrado ou até que tarefa atinja um estado final. Por outro lado, uma tarefa *transiente* tem um gatilho específico para a transição a partir do estado de partida. Uma tarefa transiente não é executada quando o agente é iniciado, em vez disto, espera até o seu disparador ser recebido pelo agente. Com tarefas transientes, é possível haver tarefas múltiplas, do mesmo tipo, sendo executadas concorrentemente.

Tarefas concorrentes consistem em estados e transições, que são similares aos estados e transições de muitos outros modelos de autômatos finitos. Os estados abrangem o processamento que ocorre internamente ao agente, enquanto transições permitem comunicação entre agentes ou entre tarefas. Uma transição consiste em estado fonte, estado destinação, gatilho, condição de guarda e transmissões, e usa a sintaxe *trigger [guard]* ^ *transmission(s)*. Transmissões múltiplas podem ser separadas com um ponto e vírgula (;), contudo não há ordenação implícita das transmissões.

Geralmente, os eventos especificados no gatilho ou nas transmissões são assumidos virem de/irem para uma outra tarefa dentro do mesmo papel, permitindo assim a tarefas internas coordenarem seus comportamentos. Contudo, dois eventos especiais são usados para indicar mensagens que são trocadas entre agentes: *send* e *receive*. O evento *send* (seguindo a sintaxe *send(message, agent)*) é usado para enviar uma mensagem a um outro agente, enquanto o evento *receive* (denotado como *receive(message, agent)*) significa o recebimento de uma mensagem deste tipo. A *mensagem* é definida como uma *performative*[34] que descreve o intento da mensagem, juntamente com um conjunto de parâmetros que são o conteúdo da mensagem. O formato de uma mensagem é *perfomative*($p_1$,...,$p_n$), onde $p_1$,...,$p_n$ denotam n parâmetros possíveis. É também possível enviar uma mensagem a um grupo de agentes via *multicasting*. Ao invés de especificar um único agente ao qual enviar a mensagem, um nome de grupo é especificado, colocando o nome do grupo entre os sinais de menor e maior (e.g., <group-name>).

Os estados podem conter atividades (ou funções) que podem ser usadas para representar raciocínio interno, ler percepção de sensores, ou executar ações via atuadores. Atividades múltiplas podem ser incluídas em um único estado, e são executadas em seqüência. Uma vez em um estado, a tarefa permanece naquele estado, até que o processamento da atividade esteja completo e uma transição de saída do estado torne-se habilitada. Uma vez que o processamento se inicie em um estado, todas as atividades no estado devem ser completadas antes de qualquer transição para fora do estado ser habilitadas.

As variáveis usadas nas definições de atividades intra-estados e nas definições de mensagem e evento de transições são assumidas serem visíveis globalmente dentro da tarefa, mas não fora da tarefa, ou dentro de atividades. Todas as mensagens trocadas entre papéis e eventos trocados entre tarefas são enfileirados para assegurar que todas as mensagens sejam recebidas mesmo se o agente, ou a tarefa, não estiver no estado apropriado para manipular a mensagem, ou o evento, imediatamente.

Também se assume que cada tarefa está em exatamente um estado em qualquer ponto no tempo. Isto significa que transições entre estados são instantâneas, enquanto os estados consomem tempo. Se não há atividades em um particular estado ou todas as atividades foram completadas e nenhuma transição foi habilitada, então a tarefa esta ociosa, esperando uma transição para ser habilitada.

[34] Uma expressão é *performative* se seu significado é o da ação da coisa que ela enuncia.

Tarefas concorrentes têm atividades pré-definidas para lidar com a mobilidade e o tempo. A atividade *move* especifica que um agente deve mover-se para um novo endereço. O resultado da atividade *move* é um valor "booleano" que declara se o movimento ocorreu de fato. É possível que um agente queira se mover para uma nova localização, mas esteja inabilitado por alguma razão. O agente deveria estar apto a raciocinar sobre isto e a lidar com isto adequadamente. A sintaxe para a atividade *move* é *Boolean* = *move*(*location*).

Para raciocinar com base no tempo, o modelo de tarefa concorrente oferece uma atividade *timer* construída internamente. Um agente pode definir um *timer* usando a atividade *setTimer*. A atividade *setTimer* toma um tempo como entrada e retorna um *timer* que indicará "estouro de tempo", exatamente após ter transcorrido o intervalo de tempo especificado. Para ver se o *timer* indicou estouro de tempo, o *timer* pode ser testado pelo agente, usando a atividade *timeout*. Usando as atividades *setTimer* e *timeout*, um agente pode usar o tempo no cumprimento das responsabilidades que lhe são atribuídas. A sintaxe para as funções *setTimer* e *timeout* é mostrada abaixo.

t = *setTimer*(*time*)

*Boolean* = *timeout*(t)

Uma vez que uma transição seja habilitada, a transição é executada e sua execução ocorre instantaneamente. Isto significa que eventos e mensagens são enviados instantaneamente e que o estado da tarefa corrente torna-se o estado de destino da transição. Se transições múltiplas são habilitadas simultaneamente, o seguinte esquema de prioridade é usado.

1. Transições cujos gatilhos contêm eventos internos de outras tarefas.
2. Transições cujas transmissões contêm eventos internos.
3. Transições cujos gatilhos contêm uma mensagem *receive* oriunda de outros papéis.
4. Transições cujas transmissões contêm uma mensagem para um outro papel.
5. Transições com somente condições de guarda válidas.

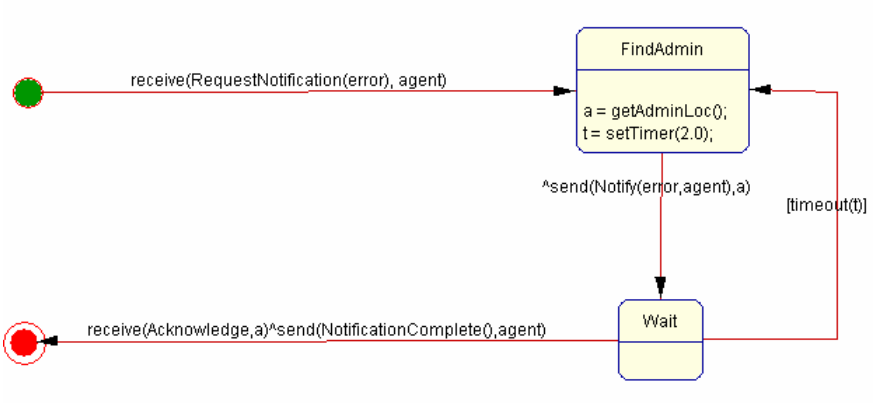


FIG 5. 7: Diagrama de Tarefa Concorrente: tarefa Notifique Usuário do papel NotificadorAdmin (DELOACH, 2001b).

A FIG. 5.7 mostra a tarefa Notifique Usuário do papel NotificadorAdmin. A tarefa é iniciada com o recebimento de uma mensagem SolicitaçãoNotificação de um outro agente. O erro a ser enviado ao administrador está capturado no parâmetro *error*. Após a mensagem ser recebida, a tarefa vai para o estado EncontreAdmin, onde ela localiza o administrador e configura um *timer*. Quando estas atividades estão completas, a tarefa envia uma mensagem Notifique ao administrador, passando simultaneamente o erro associado. A tarefa aguarda no estado de espera até o timer provocar uma interrupção ou uma mensagem de acusação de recebimento ser recebida. Se o *timer* indicar estouro de tempo, a tarefa retorna ao estado EncontreAdmin e exatamente as mesmas atividades são re-executadas. Contudo, se uma mensagem acusando o recebimento for recebida do Administrador, a tarefa simplesmente envia uma mensagem de NotificaçãoCompleta para a tarefa iniciadora e a tarefa corrente termina. Em razão da tarefa Notifique Usuário ser criada com base no recebimento de uma mensagem e terminar quando é completada, a tarefa Notifique Usuário é transiente.

Como discutido acima, Diagramas de Seqüência definem o conjunto mínimo de mensagens que um papel deve responder e enviar. O analista pode criar um modelo de tarefa concorrente a partir de uma seqüência tomando a seqüência de mensagens enviadas ou recebidas por aquele papel e usando-as para criar uma seqüência de estados e mensagens correspondentes. Um exemplo da versão inicial da tarefa Notifique Usuário, derivada diretamente do diagrama de seqüência na FIG. 5.5, é mostrado na FIG. 5.8. Obviamente, as maiores diferenças entre a FIG. 5.8 e a FIG. 5.7 são a adição dos parâmetros, atividades e capacidade de estouro de tempo, que foram adicionadas para que a operação se torne robusta.

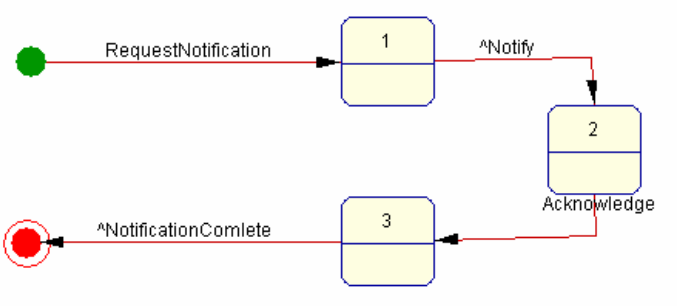


FIG 5. 8: Diagrama de Tarefa Concorrente Inicial da tarefa Notifique Usuário do papel NotificadorAdmin (DELOACH, 2001b).

Após criar o diagrama de tarefa concorrente inicial, o analista deve determinar o processamento interno que o papel deve executar para satisfazer o caso de uso. Esse processamento interno é capturado como atividades dentro dos estados existentes. O analista também preenche a informação sobre os dados passados nas mensagens bem como quaisquer mensagens adicionais necessárias a uma troca robusta de informação.

À medida que as tarefas são criadas para cada Diagrama de Seqüência, o analista pode notar que diversas tarefas são similares e podem ser combinadas. Neste caso, o analista pode combinar múltiplas tarefas em uma única tarefa, geralmente mais complexa, que pode tratar todos os casos de uso.

### 5.1.4 RESUMO DA FASE DE ANÁLISE

Uma vez que os modelos de Tarefa Concorrente tenham sido definidos para cada papel, a fase de Análise está completa. Embora haja três passos na fase de Análise MaSE, o analista está apto, e mesmo encorajado, a mover-se livremente entre os passos. O roteiro resumido do que tem de ser feito na fase de Análise é o seguinte:

1. Identifique as metas a partir dos requisitos e estruture-as em um Diagrama de Hierarquia de Meta.
2. Identifique casos de uso e crie diagramas de seqüência para ajudar a identificar um conjunto inicial de papéis e caminhos de comunicações.
3. Transforme metas em um conjunto de papéis
   a. Crie um modelo de papel para capturar papéis e suas tarefas associadas.
   b. Defina um Modelo de Tarefa Concorrente para tarefa par definir comportamento de papel

## 5.2 FASE DE PROJETO

Os quatro passos da fase de projeto são Criando Classes de Agente, no qual o projetista atribui papéis a tipos específicos de agente; Construindo Conversações, no qual as conversações que de fato ocorrerão entre os agentes são definidas; Montando Classes de Agentes, passo no qual a arquitetura interna e o processo de raciocínio das classes de agente são projetados; no último passo da fase de projeto, Projeto do Sistema, o projetista define a quantidade e a localização dos agentes no sistema implementado.

### 5.2.1 CRIANDO CLASSES DE AGENTE

Neste passo, classes de agente são criadas a partir dos papéis definidos na fase de análise. O produto final deste passo é um Diagrama de Classe de Agente que descreve a organização geral do sistema consistindo em classes de agentes e as conversações entre elas. Uma classe de agente é um gabarito para um tipo de agente no sistema e é análoga a uma classe de objeto sob o paradigma de orientação a objeto. Um agente é uma instância real de uma classe de agente. As classes de agentes são definidas neste passo em termos dos papéis que elas desempenham e das conversações das quais elas têm de participar.

Na prática, classes de agente podem desempenhar muitos papéis, com os papéis mudando dinamicamente durante a execução. Além disto, agentes da mesma classe podem desempenhar papéis diferentes ao mesmo tempo.

Durante este passo, identifica-se as conversações das quais diferentes classes de agente têm de participar. Neste passo, não se define todos os detalhes das conversações, os quais serão acrescentados durante o passo Construindo Conversações, como descrito adiante. O conjunto de conversações do qual uma classe de agente tem de participar é deduzido das comunicações externas dos papéis que os agentes desempenham. Por exemplo, suponha que os papéis A e B sejam definidos por tarefas concorrentes que se comunicam uma com a outra. Então, se o agente 1 desempenha o papel A e o agente 2 desempenha o papel B, o projetista tem de definir uma conversação entre o agente 1 e o agente 2 para implementar a comunicação descrita entre os papéis A e B.

As classes de agentes e as conversações são documentadas por meio de um Diagrama de Classe de Agente, que é similar a diagramas de classe orientados a objeto. Há duas principais diferenças. Primeiramente, as classes de agente não são definidas por atributos e métodos;

elas são definidas pelos papéis que elas desempenham. A segunda diferença é a semântica dos relacionamentos entre classes de agentes. Nos Diagramas de Classe de Agente todos os relacionamentos entre as classes são conversações que podem ocorrer entre duas classes de agente. Uma amostra de Diagrama de Classe de Agente é apresentada na FIG. 5.9. As caixas na FIG. 5.9 denotam classes de agentes e contêm o nome da classe e o conjunto de papéis que cada agente desempenha. As linhas com setas identificam conversações e apontam do iniciador da conversação para o respondedor. O nome da conversação é escrito sobre a linha ou próximo dela.

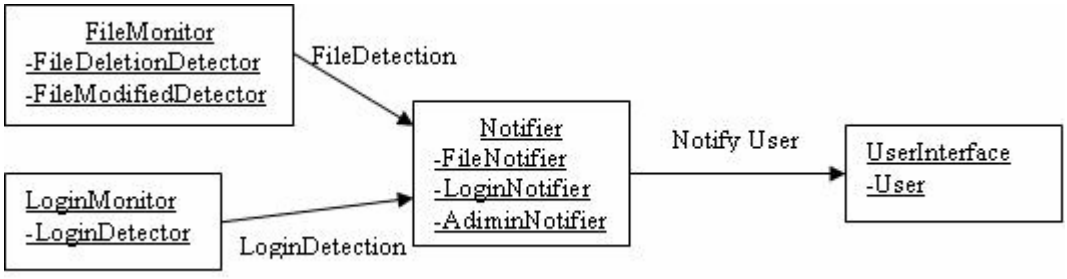


FIG 5. 9: Diagrama de Classe de Agente (DELOACH, 2001b)

O Diagrama de Classe de Agente é o primeiro objeto de projeto na MaSE que mostra o sistema multiagente completo como ele será implementado. Se tivermos seguido cuidadosamente a MaSE até este ponto, o sistema representado pelo Diagrama de Classe de Agente suportará as metas e os casos de uso identificados na fase de análise. De particular importância neste ponto é a organização do sistema – o modo segundo o qual as classes de agente são conectadas pelas conversações.

### 5.2.2 CONSTRUINDO CONVERSAÇÕES

Construindo conversações é o passo seguinte na fase de Projeto MaSE. Até este ponto, o projetista não definiu comunicações entre agentes além de declarar que elas existem. O fato de uma conversação ter de acontecer entre dois agentes é conhecido; a meta deste passo é definir efetivamente os detalhes destas conversações. Os detalhes internos das tarefas concorrentes são indispensáveis a este estudo.

Uma conversação MaSE define um protocolo de coordenação entre dois agentes. Especificamente, uma conversação consiste em dois Diagramas de Classe de Comunicação,

um para o iniciador e o outro para o respondedor. Um Diagrama de Classe de Comunicação é um autômato de estado finito que define os estados da conversação das duas classes de agentes participantes como mostrado na FIG. 5.10. O iniciador sempre começa a conversação enviando a primeira mensagem. Quando um agente recebe uma mensagem, ele compara-a com suas conversações ativas. Se encontrar uma conversação correspondente, o agente executa a transição na conversação apropriada para um novo estado e executa quaisquer ações ou atividades exigidas pela transição, ou pelo novo estado. Em caso contrário, o agente assume que a mensagem é um pedido de início de uma nova conversação e compara-a com todas as conversações possíveis das quais o agente possa participar com o agente que enviou a mensagem. Se o agente encontrar uma conversação correspondente, começa uma nova conversação. A sintaxe de uma transição segue a notação UML convencional como mostrada abaixo.

rec-mess(args1)[cond]/action^trans-mess(args2)

A sintaxe acima estabelece que se a mensagem *rec-mess* é recebida com os argumentos *args1* e a condição *cond* é satisfeita, então o método *action* é chamado e a mensagem *trans-mess* é enviada com argumentos *args2*. Todos os elementos da transição são opcionais. Analisando a transição a partir do estado de partida na FIG. 5.10, é óbvio que ele corresponde a um iniciador metade de uma conversação uma vez que a transição do seu estado de partida é disparada pela mensagem transmitida pelo agente.

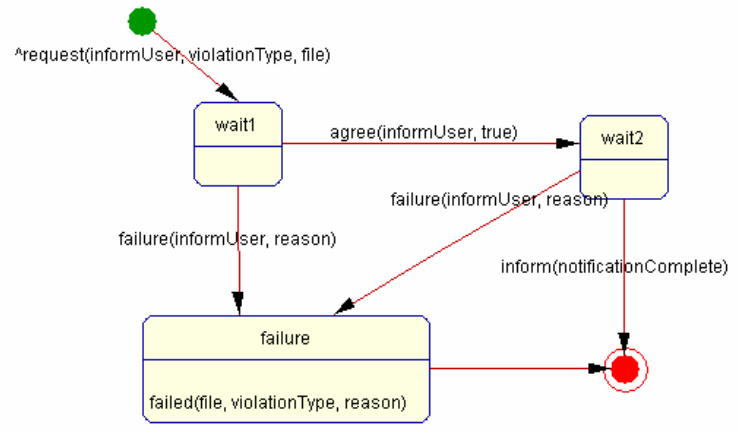


FIG 5. 10: Diagrama de Classe de Comunicação para conversação DetecçãoArquivo (Parte I) (DELOACH, 2001b).

Portanto, na FIG. 5.10, o agente MonitorArquivo (o iniciador da conversação FileDetection como definido na FIG. 5.9) envia uma mensagem para o agente Notificador pedindo-lhe que informe ao usuário uma violação de arquivo. Neste ponto, o agente MonitorArquivo entra em um estado de espera esperando por uma resposta. Se o Notificador puder notificar o usuário, envia uma mensagem de concordância e informa ao MonitorArquivo quando a notificação tiver sido completada. Se o Notificador não puder informar ao usuário atendendo o MonitorArquivo, ele retorna uma mensagem de falha com a razão apropriada (e.g. o usuário pode ter se desconectado do sistema, ou a rede pode ter caído etc.). Após receber uma mensagem de falha, o MonitorArquivo executa uma chamada interna para o método falhado.

O lado complementar a conversação, do ponto de vista do agente Notificador, é mostrado na FIG. 5.11. Em uma conversação bem projetada, cada seqüência possível de mensagens enviadas/respondidas por um lado da conversação deve corresponder a mensagens enviadas/respondidas pelo lado oposto. Conversações devem estar livres de "*deadlock*". Além de "*deadlock*", há outras maneiras de projetar impropriamente uma conversação. Por exemplo, cada mensagem enviada de um lado da conversação deve estar apta a ser recebida na outra metade da conversação. Adicionalmente, a conversação deve estar apta a sair de cada estado, significando que cada estado deve ter uma transição válida a partir dele que por fim leve ao estado final. A ferramenta agentTool permite a verificação automática das conversações durante o estágio de projeto. Uma vez que um conjunto de conversações tenha sido criado, o projetista pode optar por verificá-lo automaticamente. O processo de verificação começa com a transformação automática das conversações do sistema na linguagem de modelagem Promela. Então, o modelo Promela é analisado automaticamente usando a sub-ferramenta de verificação Spin (parte do agentTool) para detectar erros tais como "*deadlock*", laços que não progridem, erros de sintaxe, mensagens não usadas e estados não usados. Este processo realimenta o projetista automaticamente via mensagens de texto e acentuação (*highlighting*) gráfica de condições de erro. O tópico "*deadlock*" e os métodos usados no agentTool para evitá-lo e detectá-lo são cobertos em detalhe por Lacey (T. H. Lacey e S. A. Deloach, "*Verification of Agent Behavioral Models*", em *Proceedings of the International Conference on Artificial Intelligence*, CSREA Press, Las Vegas, Nevada, julho 2000 e T. H. Lacey e S. A. Deloach, "*Automatic Verification of Multiagent Conversations*" em *Proceedings of the Eleventh Annual Midwest Artificial Intelligence and Cognitive Science*

*Conference*, pp. 93-100, AAAI Press, Fayetteville, Arkansas, Abril 2000, *apud* (COCKBURN, 1997).

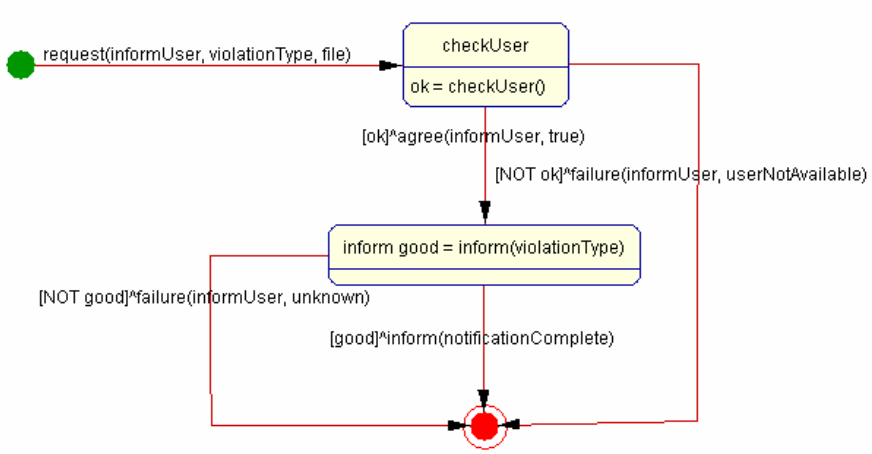


FIG 5. 11: Diagrama de Classe de Comunicação para a Conversação DetecçãoArquivo (Parte II) (DELOACH, 2001b)

Como mencionado anteriormente, o projetista determina o conjunto de conversações do qual uma classe de agente pode participar pelos dos papéis que a classe de agente desempenha. Igualmente, o projeto detalhado das conversações é derivado das tarefas concorrentes associadas a estes papéis. Dado que um Modelo de Tarefa Concorrente especifica um único *thread* de controle que integra interações inter-papéis e intra-papel, elas fornecem informação crítica necessária para definir conversações. Basicamente, cada tarefa que define comunicação externa cria uma ou mais conversações. Se toda a comunicação dentro da tarefa é com um único papel, ou conjunto de papéis que tenham todos sido mapeados em uma única classe de agente, a tarefa pode ser mapeada diretamente em uma única conversação. Mais geralmente, contudo, tarefas concorrentes são mais complexas e consistem em múltiplas conversações. As comunicações entre papéis ou agentes separados podem ser mapeadas em conversações individuais.

Uma vez que todas as informações dos Modelos de Tarefa Concorrente tenham sido capturadas como parte de conversações, o projetista deve assegurar que outros fatores, tais como robustez e tolerância à falha, sejam levados em conta. Por exemplo, se um particular agente envia uma mensagem a outro agente requerendo que alguma ação seja feita, o que acontece se o outro agente recusa ou é incapaz de completar o requerimento? A conversação deveria ser robusta o bastante para manipular estes possíveis problemas.

No projeto de conversações, o projetista defronta-se com o compromisso de ter muitas conversações simples ou poucas conversações complexas. Se o sistema tem um grande número de comunicações simples, elas seriam implementadas por uma série de conversações menores. Conversações maiores e mais complexas são apropriadas somente se um protocolo elaborado é necessário.

### 5.2.3 MONTANDO AGENTES

Durante o passo montando agentes da fase de projeto, as entranhas das classes de agente são criadas. Isto é consumado via dois sub-passos: definindo a arquitetura de agente e definindo os componentes que compõem a arquitetura. Os projetistas têm de escolher se projetam sua própria arquitetura ou usam arquiteturas pré-definidas tal como Crença-Desejo-Intenção (CDI). Da mesma maneira, um projetista pode usar componentes pré-definidos ou desenvolve-los desde o início. Os componentes consistem em um conjunto de atributos, métodos e, se complexos, podem ter uma sub-arquitetura.

Um exemplo de um Diagrama de Arquitetura de Agente é mostrado na FIG. 5.12. Componentes arquitetônicos (denotados pelas caixas) são conectados a um conector interno-ao-agente ou conector externo-ao-agente. *Conectores internos-a-agentes* (flechas finas) definem visibilidade entre componentes enquanto *conectores externos-a-agentes* flechas grossas tracejadas definem conexões com recursos externos tais como outros agentes, sensores e atuadores, bases de dados e depósitos de dados. O comportamento interno de componente pode ser representado pelas definições de operações formais bem como por diagramas de estado que representam eventos ocorridos entre componentes. A arquitetura e definição interna dos componentes devem ser consistentes com as conversações definidas no passo prévio. No mínimo isto requer que cada ação ou atividade definida em um Diagrama de Classe de Comunicação seja definida como uma operação em um dos componentes internos. Os diagramas de estado de componente interno e as operações podem também ser usados para iniciar e coordenar várias conversações.

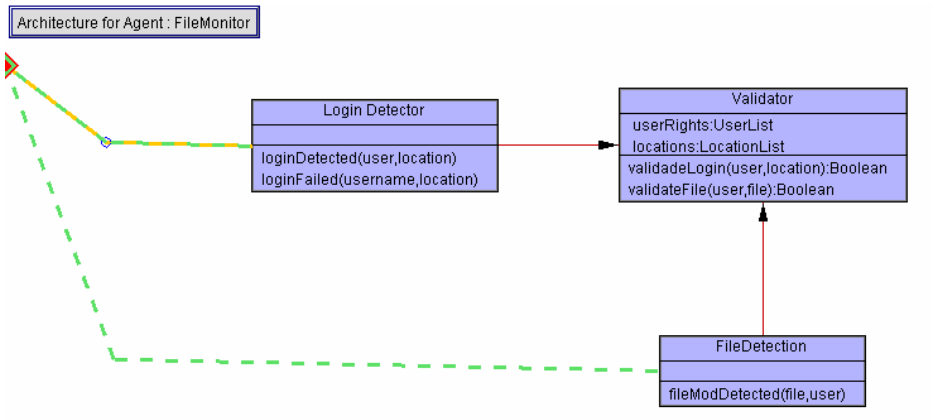


FIG 5. 12: Arquitetura do Agente MonitorArquivo (DELOACH, 2001b)

A arquitetura do agente MonitorArquivo é mostrada na FIG. 5.12. O agente MonitorArquivo tem três componentes. Os componentes DetectorLogin e DetectorArquivo executam basicamente o mesmo, interagindo com o sistema operacional para detectar tentativas de *logins* e de modificação de arquivo. Ambos os componentes chamam o componente Validador para determinar se o *login* e os acessos a arquivo foram válidos ou precisam ser relatados. Os *conectores exteriores-a-agente* nos componentes detectores denotam tanto o fato de que os componentes interagem como sistema operacional, quanto o fato de que se comunicam com o agente Notificador via conversações. Basicamente, cada tarefa de cada papel desempenhada por um agente define um componente na classe de agente. A tarefa concorrente em si mesma é transformada em uma combinação de diagrama de estado interno de componente e o conjunto de conversações. Atividades identificadas na tarefa concorrente tornam-se métodos do componente.

### 5.2.4 PROJETO DE SISTEMA

O passo final do método MaSE toma as classes de agente definidas previamente e instancia os agentes reais. Usa-se um Diagrama de Desenvolvimento para mostrar os números, tipos e localizações dos agentes dentro de um sistema. O projeto do Sistema é de fato o passo mais simples da MaSE, uma vez que a maior parte do trabalho foi feita nos passos prévios. O conceito de instanciamento de agentes a partir de classes de agentes é similar ao instanciamento de objetos a partir de classes de objetos na programação orientada a objeto.

Diagramas de Desenvolvimento descrevem um sistema baseados nas classes de agentes definidas nos passos prévios da MaSE. A FIG. 5.13 mostra um Diagrama de Desenvolvimento para o sistema exemplo. As caixas tridimensionais representam agentes enquanto as linhas de conexão representam as conversações de fato entre agentes. Os agentes são identificados por seu nome de classe ou pela forma de *designador:classe* se há múltiplas instâncias da classe. Qualquer conversação entre classes de agentes aparece entre agentes daquelas classes. Além disto, uma caixa de linha-pontilhada indica um agente sendo executado na mesma plataforma física.

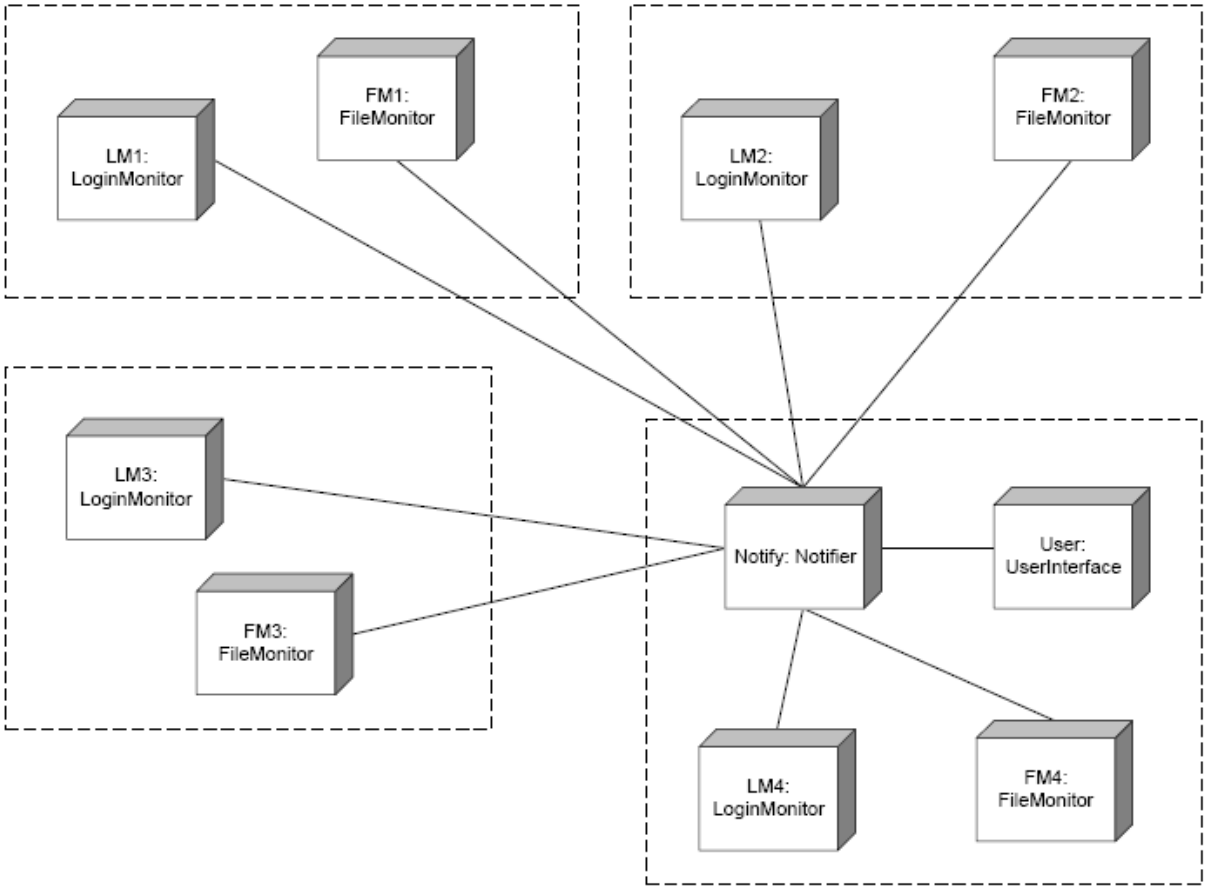


FIG 5. 13: Diagrama de Desenvolvimento (DELOACH, 2001b)

O projetista deve definir o sistema usando um Diagrama de Desenvolvimento antes de implementá-lo, pois os agentes requerem tipicamente a informação contida no diagrama, tal como o *hostname*, ou endereço, para poderem participar de comunicações externas. Diagramas de Desenvolvimento também oferecem oportunidade ao projetista de sintonizar o sistema com o seu ambiente. O projetista pode usar os diagramas de desenvolvimento para

definir várias configurações de agentes e computadores visando a maximização da utilização da potência de processamento e da largura de banda disponíveis. Em alguns casos, o projetista pode especificar um número particular de agentes no sistema, ou os computadores específicos nos quais certos agentes devam residir. O projetista deveria também considerar os requisitos de comunicação e processamento quando designando agentes a computadores. Para reduzir a sobrecarga de comunicação, o projetista pode escolher dispor agentes na mesma máquina. Contudo, pôr agentes em demasia em uma única máquina anula as vantagens da distribuição ganhas pelo uso do paradigma multiagente. Da mesma forma, se um agente tem grande necessidade de processamento, o projetista pode pô-lo sozinho em uma máquina. Um dos aspectos fortes da MaSE é possibilitar que o projetista faça estas modificações após projetar a organização do sistema, permitindo-lhe analisar uma variedade de configurações (DELOACH, 2001b).

.

### 5.2.5 RESUMO DA FASE DE PROJETO

Uma vez que se tenha terminado de fazer os diagramas de desenvolvimento, a fase de Projeto está completa. O roteiro resumido da Fase de Projeto é o seguinte:

1. Atribua papéis a classes específicas de agente e identifique conversações pelo exame dos modelos de tarefa concorrente com base nos papéis que cada classe de agente desempenha.
2. Construa conversações pela extração das mensagens e estados definidos pelos caminhos de comunicação nos modelos de tarefa concorrente, adicionando mensagens e estados para dotar o sistema de robustez.
3. Defina o interior das classes de agentes, definindo a arquitetura de cada classe de agente, usando componentes e conectores. Assegure que cada ação definida em uma conversação seja implementada como um método dentro da arquitetura de agente.
4. Defina a estrutura final de sistema usando diagramas de desenvolvimento.

## 5.3 OBSERVAÇÕES SOBRE A MASE

A MaSE em grande parte se consubstancia na agentTool. A agentTool encerra a disciplina do método, e o método não contempla tipos de sub-modelos importantes para a

caracterização de SMAs como, por exemplo, os modelos de coordenação de agentes. Também no que tange

A agentTool é flexível, permitindo que as alterações de projeto sejam feitas sem ônus excessivo.

A bibliografia sobre a MaSE é restrita.

Do ponto de vista do projetista, seria interessante existir na MaSE uma biblioteca de conversações, normalizadas em protocolos de comunicação, abrangendo as conversações mais freqüentes nos sistemas. Tais protocolos poderiam ser inerentemente robustos, o que também facilitaria o trabalho do projetista.

A MaSE não prevê diagramas de caso de uso.

A MaSE é um método *top down*, e, apesar de usar o conceito de classe, não enfatiza a re-usabilidade dos agentes, por exemplo.

# 6 MODELAGEM DO SIMULADOR

O objetivo do projeto do simulador é o de **modelar** em *hardware* e *software* um simulador de frota de dirigíveis aéreos autônomos não tripulados capazes de operar cooperativamente – a frota servirá para comunicação de dados e monitoramento de ambientes desestruturados.

Como também já foi explicitado, o simulador, em si, destina-se a apoiar a definição, padronização e teste das conversações entre os módulos *intra* agentes físicos e entre os agentes físicos, bem como a experimentação de algoritmos de navegação.

O paradigma de modelagem do simulador é o paradigma multiagente, pelas razões mostradas ao longo dos itens 2, 3 e 4, e resumidas em 4.2.2; o método de modelagem é a MaSE, por motivos já vistos.

O modelo do simulador consiste em um conjunto de especificações e de sub-modelos produzidos de acordo com a MaSE, – com diferenças menores realçadas quando necessário – abrangendo diversos aspectos de interesse do sistema. O modelo é construído a partir do Contexto Inicial do Sistema – o repositório de todos os dados preliminares relativos ao sistema. Para a MaSE, o Contexto Inicial do Sistema inclui um conjunto bem-definido de requisitos funcionais, a partir dos quais os modelos do método são deduzidos.

## 6.1 MODELO E SISTEMA

O simulador é um modelo da frota. Para um observador B, um objeto A* é um modelo de um objeto A na medida em que B possa usar A* para responder questões de interesse para ele sobre A (MINSKY, 1965). O simulador, em si mesmo, é um modelo da frota de dirigíveis, pois serve para responder questões sobre a arquitetura dos robôs constituintes da frota, sobre comunicação entre eles, sobre como se coordenam, etc., a quem esteja interessado nelas.

Na expectativa de que os componentes da frota pudessem ser mapeados em componentes de *hardware* e de *software* do simulador (3.1), modelou-se a frota por meio da MaSE, chegando-se a um modelo no qual os componentes da frota são mapeados respectivamente em modelos de agentes, plataformas de *hardware*, conexões de rede local, conversações, etc., todos modelos dos componentes do simulador. Desta forma, chegou-se a um modelo da frota

que é, também, do simulador: em ambos os casos, ele satisfaz a definição de modelo dada em (MINSKY, 1965).

Há funções no simulador que não existem na frota. Por exemplo, no simulador existem interfaces para interação com o usuário do simulador, – para inserção de dados de ambiente, para representação da frota interagindo com o ambiente, etc. – mas não na frota. Neste trabalho, contudo, muitas destas interfaces não são detalhadas.

As relações entre os modelos da frota e do simulador são importantes porque sugere uma forma de modelar o sistema: o modelo do simulador pode ser obtido por intermédio do modelo da frota. Nesta hipótese, os componentes do modelo da frota são modelos dos componentes do simulador. Assim, o modelo do simulador é obtido mapeando-se os componentes da frota nos modelos dos componentes do simulador. Procede-se desta maneira neste trabalho.

## 6.2 CONTEXTO INICIAL DO SISTEMA

A modelagem na MaSE parte de dados, em geral desestruturados, a respeito do sistema – a frota de veículos aéreos não tripulados, no caso. Tais dados compõem o que na MaSE é chamado de Contexto Inicial do Sistema. A MaSE espera que neste contexto inicial haja um conjunto bem definido de requisitos funcionais do sistema, conjunto no qual estariam "registrados os serviços que o sistema deve prestar e os comportamentos que o sistema deve ter" (DELOACH, 2001b).

Não se dispõe dos requisitos funcionais da frota e, também, na prática, de um ator primário do qual se possa obtê-los. Tal ator primário, se existisse, serviria, também, para validar as metas e demais produtos da fase de Análise. Para contornar tais lacunas, os requisitos da frota, funcionais e não funcionais, são deduzidos daquilo que seria a frota, – o super-sistema hipotético – da bibliografia e de suposições sobre o uso do simulador.

Duas hipóteses sobre a frota condicionam o restante da especulação. A primeira é que o tipo de veículo aéreo da frota é o dirigível – os motivos foram expostos anteriormente. Além disto, supõe-se, também, que todos os veículos da frota sejam iguais em todos os aspectos[35] –

[35] Embora possam desempenhar eventualmente papéis diferentes.

a frota é, portanto, um SMR homogêneo. O tipo dos veículos que compõem a frota é denominado aqui de Dirigível Aéreo[36] Não Tripulado (DANT).

Para que se levante os requisitos da frota, é preciso explicitar o que se entende aqui por frota de veículos aéreos não tripulados, pois os requisitos decorrem deste entendimento. Para tal, se considerará os veículos de per si e subseqüentemente os veículos tomados em conjunto.

### 6.2.1 O DANT

O DANT assemelha-se ao AURORA I. É um veículo robótico – significando que possui autonomia navegacional relativa[37]. Compõe-se do veículo, dos atuadores, dos sensores, do computador de bordo – mais o *software* de navegação – e do sistema de comunicação. Além dos sensores de navegação, possui sensores de monitoramento de ambiente. Diferentemente do AURORA I, o sistema de comunicação do DANT provê, a pontos em terra, facilidades de conexão à Internet: o DANT é também um provedor de acesso à Internet. O sistema de comunicação serve, ainda, para comunicação entre DANTs, para transmissão de dados de monitoramento e para comunicação com a estação de controle de terra. As facilidades de comunicação permitem o envio de comandos ao dirigível, troca de dados entre os dirigíveis, troca de dados entre pontos em terra e o tráfego dos dados de monitoramento. O DANT não tem interface de operação com operador, somente de comunicação.

O sistema de navegação do DANT tem por objetivo deslocar (decolar, pousar, pairar e mover-se de um ponto a outro no espaço) um único DANT. Este sistema persegue independentemente das missões que lhe são atribuídas as seguintes metas permanentes:

- Auto-preservação, desviando-se automaticamente de obstáculos de molde a evitar colisões.
- Correção automática de trajetória, permitindo ajuste da trajetória efetiva à planejada em razão de condições ambientais adversas (chuva, vento, variação de temperatura e variação de pressão atmosférica).
- Manutenção dos enlaces de telecomunicação do DANT com os DANTs que lhe são logicamente adjacentes na rede de telecomunicação constituída pela frota.

[36] A primeira acepção de dirigível é "que se pode dirigir" (Dicionário do Aurélio, 3ª edição). Assim, e para manter relação com a sigla VANT (veículo aéreo não tripulado), que ocorre com freqüência na área, optou-se por dirigível aéreo, e, não, por dirigível. DANT, portanto, é uma especialização de VANT.
[37] Outras autonomias não são consideradas neste trabalho.

#### 6.2.1.1 O *SOFTWARE* EMBARCADO DO DANT

O *software* embarcado é um componente importante porque embute as funções que dotam o DANT de autonomia navegacional. O *software* embarcado sustenta principalmente as seguintes funções:

- Execução de algoritmos de navegação autônoma. Tais algoritmos fazem uso dos dados sensoriais para evitar colisões, corrigir trajetória e geram os comandos apropriados para os atuadores.
- Controle contínuo da posição do DANT, consultando sensores e comandando atuadores, de forma a cumprir a missão navegacional e as metas permanentes.
- Interação com o SDANT, recebendo missões navegacionais e informando o andamento de seu cumprimento.

#### 6.2.1.2 OUTROS COMPONENTES

Outros componentes do DANT são responsáveis pelas comunicações e monitoramento. Estes componentes não serão detalhados neste trabalho.

### 6.2.2 SISTEMA DE DIRIGÍVEIS AÉREOS NÃO TRIPULADOS (SDANT )

O SDANT consiste em um conjunto de DANTs capaz de atuar autonomamente na busca de cumprimento de missão, comportando-se como um sistema. O SDANT é a frota. A missão cometida ao SDANT geralmente têm como componente ALGUMA missão navegacional. Assim, por exemplo, para monitorar uma área – a missão – é necessário que o SDANT se desloque previamente até a área a ser monitorada, – a missão navegacional – compondo, ou não, um enlace complexo de telecomunicação. A autonomia que se busca aqui é a autonomia navegacional do SDANT.

Cada DANT do SDANT opera autonomamente e interage quase sempre cooperativamente com os demais, por meio de comunicação – há casos em que a coordenação é competitiva, quando se busca, por exemplo, minimizar o custo da missão. Assim, o SDANT é composto pelos DANTs mais a comunicação entre eles.

As seguintes metas são perseguidas pelo SDANT independentemente das missões que lhe são atribuídas:

- manter permanentemente os DANTs em contato uns com os outros e
- minimizar os custos das missões.

Para a coordenação dos DANTs do SDANT optou-se basicamente pelo modelo de coordenação por planejamento multiagente, na modalidade planejamento centralizado de planos distribuídos (ver 4.2.3). Nesta modalidade, um agente é o líder e comanda a organização do trabalho dos outros agentes; o líder envolve-se no processo de decisão para a equipe inteira, e os outros membros podem atuar somente de acordo com as suas instruções; é ele o responsável por garantir que a frota opere de forma coesa, como um sistema (ver 4.2.3) – circunstancialmente, são usados outros tipos de coordenação.

Pretende-se que o DANT líder seja, também, o robô que representa a frota. Este robô, além de planejar e coordenar as operações dos demais DANTs, é também o responsável pelas comunicações do SDANT com os atores externos. É ele que recebe os dados das missões da frota e fornece os dados de execução da missão.

#### 6.2.2.1 O *SOFTWARE* EMBARCADO DO DANT LÍDER

No que tange ao *software* embarcado, o DANT líder, representante do SDANT, tem instâncias ativas de objetos que o capacitam a atuar como porta da frota e como o planejador e controlador de navegação dos demais robôs – os demais DANTs têm estes mesmos objetos, porém inativos. Afora estas funcionalidades especiais, o DANT líder tem as mesmas funcionalidades dos DANTs comuns.

### 6.2.3 OUTRAS CONSIDERAÇÕES

Supõe-se que as responsabilidades de comunicação estejam a cargo de uma camada inferior à que está sendo tratada pela modelagem – provavelmente o sistema operacional do computador embarcado ou, então, algum componente exclusivamente dedicado. Estas responsabilidades comunicativas incluem comunicações com a estação de terra, – comandos e dados de controle – entre robôs, para acesso à Internet e de monitoramento – comandos e dados de monitoramento.

### 6.2.4 RESUMO DOS REQUISITOS DO SDANT

Para facilitar a visualização, os requisitos principais do SDANT estão dispostos adiante. Os requisitos estão agrupados em funcionais e não-funcionais. Os funcionais referem-se à função[38] (finalidade) do sistema, enquanto os não funcionais a aspectos tais como flexibilidade, robustez, facilidade do sistema ser modificado, confiabilidade etc.

#### 6.2.4.1 PRINCIPAIS REQUISITOS FUNCIONAIS

- Monitorar ambiente.
- Prover acesso à Internet a pontos IP em terra.
- Deslocar a frota até a área a ser monitorada.
- Dispor a frota de maneira a compor enlace de telecomunicação.
- Pousar e decolar SDANT.
- Controlar o acesso à Internet.

#### 6.2.4.2 PRINCIPAIS REQUISITOS NÃO-FUNCIONAIS

- Manter os enlaces de telecomunicações entre os robôs.
- Evitar colisões.
- Minimizar os custos de monitoramento.
- Corrigir automaticamente a trajetória.
- Realizar o deslocamento da frota autonomamente.
- Coordenar-se de modo centralizado – robô líder.
- Centralizar as comunicações com os atores externos no robô líder.

Outros requisitos não-funcionais serão explicitados oportunamente ao longo da modelagem e, mesmo depois dela ter sido apresentada.

[38] Na acepção de causa final.

## 6.3 HIERARQUIA DE METAS

O Diagrama de Hierarquia de Meta, FIG. 6.1, foi deduzido principalmente dos requisitos funcionais do SDANT. As duas metas mais importantes são as relativas ao monitoramento de ambiente e ao provimento de acesso à Internet. Para atendê-las, há, pelo menos, duas alternativas de modelagem. Na primeira, as operações que preparam as suas respectivas execuções – como, por exemplo, posicionamento dos DANTs – são escolhidas, calculadas e disparadas por um ator externo. Na segunda, as operações preliminares são escolhidas, calculadas e disparadas pelo próprio SDANT. O grau de autonomia do SDANT é muito maior na segunda alternativa do que na primeira, e também maior sua complexidade.

Desenvolver-se-á aqui a primeira alternativa. O modelo dela, por ser mais simples, aumenta a probabilidade de o projeto chegar a bom termo. Outra razão da escolha da primeira alternativa é que um projeto satisfazendo a segunda pode ser derivado de projeto que atenda a primeira. Desta forma, desenvolvendo-se a primeira alternativa, tem-se como bônus adicional chegar-se mais próximo da segunda.

No Diagrama Hierárquico de Metas da MaSE, nós do grafo podem, ou não, corresponder diretamente a papéis no sistema final produzido. Quando uma meta não tem um papel que lhe corresponda diretamente, tal circunstância é explicitamente indicada no diagrama. Este procedimento deriva da importância que a metodologia empresta ao cumprimento das metas do sistema, assegurando que a cada uma delas corresponda direta, ou indiretamente, um papel. Assim, quando uma meta é levada a cabo exclusivamente por suas submetas, só estas terão papéis correspondentes no sistema final. As metas sem papéis delas derivados diretamente são ditas “repartidas”. No diagrama, as metas repartidas são a 1 Monitorar Ambiente e Comunicar Dados, a 1.2 Habilitar Usuário Internet e a 1.4 Posicionar SDANT.

As expressões que designam as metas no diagrama são *performatives*, por isto auto-explicativas. Pode ser conveniente, contudo, observar que a meta 1.2, Habilitar Usuário Internet, significa admitir um ponto IP no sistema habilitando-o a requerer comunicação com a Internet; que a meta 1.8, Gerenciar Troca de Msgs c/PCM, significa tratar todas as mensagens de controle trocadas entre o SDANT e o ator Planejador e Controlador da Missão – PCM; e que 1.5, Relatar Mensagens SDANT, informa ao PCM, sob pedido, as mensagens de controle ocorridas dentro do sistema e as trocadas com o próprio PCM.

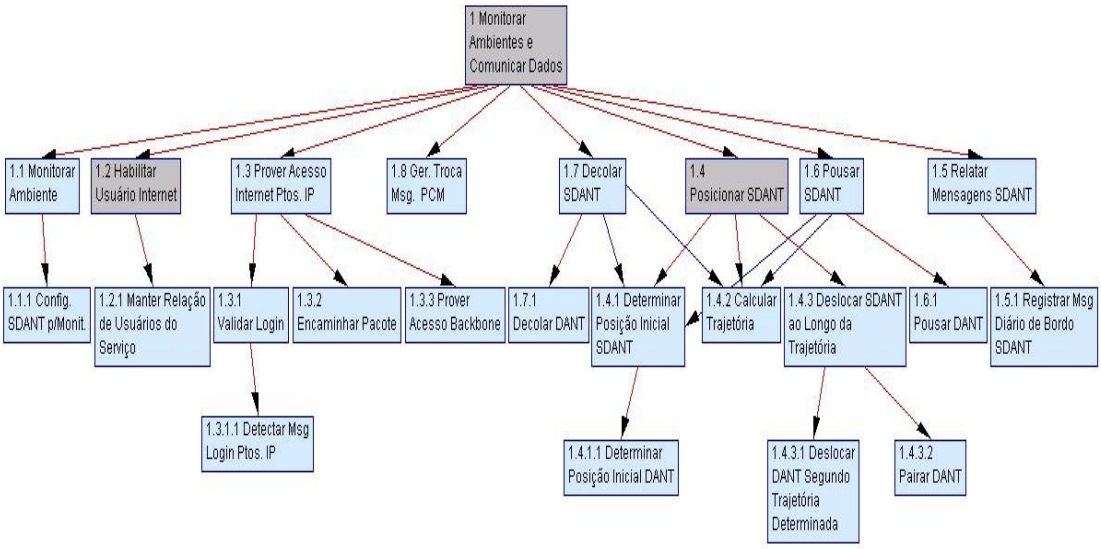


FIG 6. 1: Diagrama Hierárquico de Metas do SDANT

## 6.4 CASOS DE USO

A MaSE, não usa diagrama de casos de uso. Nela, os casos de uso são tratados um a um, e têm formato narrativo. Não obstante, usa-se na modelagem o diagrama, no estilo UML, para representar o conjunto dos casos de uso do sistema, atores e respectivos relacionamentos, visando facilitar a percepção do sistema como um todo (FIG. 6.2). Por questões de compatibilidade com a MaSE, os casos de uso são vistos a partir do sistema e interpretados como serviços que o sistema tem de prover ao ator.

Os múltiplos *scenarios* dos casos de uso são desenvolvidos subseqüentemente nos diagramas de seqüência – na MaSE, os casos de uso são detalhados e complementados pelos diagramas de seqüência. Além do diagrama para representação do conjunto de casos de uso, usa-se também narrativa estruturada (GILLEANES, 2004) para caracterizá-los individualmente.

Os atores primários são o Gerenciador de Monitoramento, pessoa ou organização responsável pelo monitoramento do ambiente; e os computadores que requerem acesso a Internet, aqui chamados coletivamente de Pontos IP. O ator primário Pontos IP requer do SDANT acesso à Internet. Este acesso é fornecido desde que o ponto IP esteja em área coberta pelo SDANT e esteja habilitado.

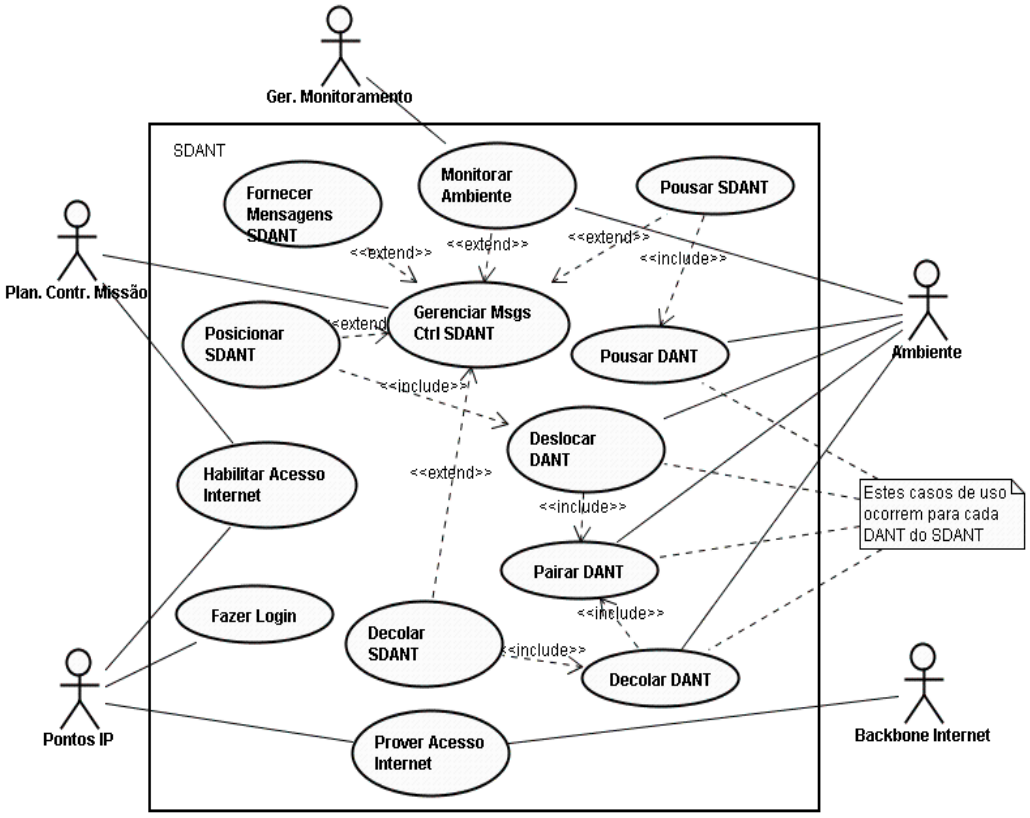


FIG 6. 2: Diagrama de Casos de Uso

O SDANT possui três atores secundários, o Ambiente, o PCM e o *Backbone* Internet. O Ambiente é um ator secundário do qual subconjuntos de estado que interessam à missão são expostos ao monitoramento. Este ator pode oferecer também dados de entes marcados, como, por exemplo, um animal com transmissor. O segundo ator secundário, o *Backbone* Internet, auxilia o SDANT a se conectar com a Internet. O terceiro ator o Planejador e Controlador da Missão – PCM – é a pessoa ou organização responsável pela definição e controle das missões do SDANT. O PCM requer do SDANT comportamentos como posicionamento em coordenadas específicas, decolagem, pouso etc., e deste recebe dados a respeito das mensagens trocadas com o ambiente pelo Sistema e as ocorridas em seu interior. As mensagens fornecidas ao PCM permitem-lhe a obtenção de informações sobre a trajetória percorrida pelo SDANT, o conhecimento das mensagens que provocaram algum tipo de falha de navegação etc.

O SDANT, por ser uma frota de dirigíveis, possui funcionalidades inerentes à sua capacidade de locomoção. É também uma frota autônoma cujos componentes atuam autônoma e cooperativamente. As duas funcionalidades suplementares da frota, – sob ponto de vista de um sistema de veículos – a capacidade de monitorar ambiente e a capacidade de

prover acesso à Internet, são exercidas com o concurso da capacidade de locomoção do SDANT. Durante a locomoção, os DANTs têm de ser mantidos intercomunicando-se e o SDANT tem de se manter em comunicação com o PCM.

Os casos de uso de movimentação abrangem dois níveis, o de SDANT e o de DANT. O de DANT trata da movimentação individual de cada DANT. Os casos de uso do nível SDANT são auxiliados pelos casos de uso do nível DANT para executar as operações de movimentação. Por exemplo, o caso de uso Pousar SDANT comanda o pouso de cada DANT, e para tanto usa o caso de uso Pousar DANT.

O caso de uso Gerenciar Troca Mensagens com PCM visa à manutenção do diário de frota e ao tratamento ágil das mensagens. O diário de frota permite o rastreio dos comandos trocados que eventualmente tenham produzido operação bem-sucedida, ou mal-sucedida.

Os casos de uso associados ao DANT, Pousar DANT, Decolar DANT e Posicionar DANT, não serão detalhados. Os papéis associados a estes casos de uso foram deduzidos diretamente do diagrama de metas e dos casos de uso associados ao SDANT.

A seguir, cada caso do uso constante no diagrama da FIG. 6.2 é detalhado, com exceção dos que se referem especificamente aos DANTs, conforme mencionado anteriormente. Este é um primeiro detalhamento, que será aprofundado e eventualmente corrigido nos diagramas de seqüência.

### 6.4.1 MONITORAR AMBIENTE

| **Caso de Uso** | Monitorar Ambiente |
|---|---|
| **Meta** | 1.1. |
| **Caso de Uso Geral** | |
| **Ator Principal** | Ger. Monitoramento |
| **Atores Secundários** | Ambiente, PCM |
| **Resumo** | Transferir o controle dos sensores de monitoramento ao Ger. Monitoramento, determinar qual DANT é o mais apropriado para o monitoramento e habilitar a comunicação dos dados de monitoramento e de controle de monitoramento entre o PCM e os sensores de monitoramento. |
| **Pré-Condições** | O SDANT deve estar pairando sobre a área a ser monitorada. |
| **Pós-Condições** | |
| **Ações do Demandante** | **Ações do Sistema** |
| 1 – Solicitar o monitoramento. | |

| | |
|---|---|
| | 2 – Selecionar DANT de menor custo |
| | 3 – Iniciar sensores de monitoramento do DANT. |
| | 4 – Receber confirmação de ativação de sensores do DANT. |
| | 5 – Habilitar o controle dos sensores pelo PCM. |
| 6 – Encerrar o monitoramento | |
| | 7 – Desabilitar o controle dos sensores. |
| | 8 – Desligar os sensores. |
| | 9 – Confirmar encerramento do monitoramento. |
| **Condições de Falha** | (1) o SDANT não consegue pairar sobre a área; (2) o enlace de comunicação de dados não é estabelecido; (3) o equipamento de monitoração não consegue enviar os dados. |
| **Observação** | O cumprimento da pré-condição requer que o PCM calcule as coordenadas de cada DANT. |

### 6.4.2 PROVER ACESSO INTERNET

| **Caso de Uso** | Prover Acesso Internet |
|---|---|
| **Meta** | 1.3; 1.3.2; 1.3.3. |
| **Caso de Uso Geral** | |
| **Ator Principal** | Pontos IP |
| **Atores Secundários** | Backbone Internet |
| **Resumo** | Conectar pontos IP em terra. |
| **Pré-Condições** | (1) O SDANT deve sustentar o enlace dos pontos que se deseja conectar. Para esta sustentação, cada DANT deverá estar pairando em coordenadas adequadas e o enlace de comunicação deverá estar operante.<br>(2) O usuário IP deve ter sido previamente habilitado.<br>(3) O login deve ter sido previamente aceito.<br>(4) O SDANT precisa estar conectado ao backbone da Internet. |
| **Pós-Condições** | |
| **Ações do Demandante** | **Ações do Sistema** |
| 1 – Solicitar a conexão dos pontos IP. | |
| | 2 – Habilitar a conexão. |
| | 3 – Informar ao PCM a habilitação da conexão. |
| 4 – Transmitir dados. | |
| 5 – Solicitar fim da conexão. | |
| | 6 – Desabilitar conexão. |
| | 7 – Informar que a conexão foi desabilitada. |
| **Condições de Falha** | O SDANT não consegue sustentar o enlace. |
| **Observação** | O SDANT funciona como se fosse um provedor de acesso à Internet. Assim, pelo menos um dos |

| | |
|---|---|
| | nós da rede implementada pelo SDANT, isto é, pelo menos um dos DANTs, tem de estar conectado ao *backbone* da Internet. Cada nó provê acesso na região em terra que cobre, ou seja, cada DANT pode permitir acesso de vários nós em terra à Internet. |

### 6.4.3 FORNECER MENSAGENS DO SDANT

| **Caso de Uso** | Fornecer Mensagens do SDANT |
|---|---|
| **Meta** | 1.5; 1.5.1. |
| **Caso de Uso Geral** | |
| **Ator Principal** | |
| **Atores Secundários** | PCM |
| **Resumo** | O SDANT informa sua condição ao PCM. |
| **Pré-Condições** | O SDANT deve estar ativo. |
| **Pós-Condições** | |
| **Ações do Demandante** | **Ações do Sistema** |
| 1 – Solicitar dados | |
| | 2 – Levantar dados no diário de frota. |
| | 3 – Encaminhar dados solicitados. |
| **Condições de Falha** | (1) o SDANT não está ativo; (2) o SDANT não consegue sustentar o enlace. |

### 6.4.4 DECOLAR SDANT

| **Caso de Uso** | Decolar SDANT |
|---|---|
| **Meta** | 1.4.1;1.7; 1.7.1. |
| **Caso de Uso Geral** | |
| **Ator Principal** | |
| **Atores Secundários** | PCM, Ambiente |
| **Resumo** | |
| **Pré-Condições** | O SDANT deve estar ativo. |
| **Pós-Condições** | |
| **Ações do Demandante** | **Ações do Sistema** |
| 1 – Solicitar que o SDANT decole. | |
| | 2 – Transmitir em seqüência comandos de decolagem para cada DANT. |
| | 3 – Receber as respostas de todos DANTs sobre o cumprimento dos comandos de decolagem. |
| | 4 – Informar ao PCM a decolagem do SDANT. |
| **Condições de Falha** | (1) O SDANT não está ativo; (2) o SDANT não consegue sustentar o enlace. |
| **Observação** | O envio de comandos de decolagem é condicionado a efetiva decolagem dos DANTs |

### 6.4.5 POUSAR SDANT

| **Caso de Uso** | Pousar SDANT |
|---|---|
| **Meta** | 1.4.1;1.6; 1.6.1. |
| **Caso de Uso Geral** | |
| **Ator Principal** | |
| **Atores Secundários** | PCM, Ambiente |
| **Resumo** | |
| **Pré-Condições** | O SDANT deve estar pairando. |
| **Pós-Condições** | |
| **Ações do Demandante** | **Ações do Sistema** |
| 1 – Solicitar que o SDANT se pouse. | |
| | 2 – Transmitir comandos de para que cada DANT pouse. |
| | 3 – Receber as respostas de todos DANTs sobre o cumprimento dos comandos. |
| | 5 – Informar ao PCM o pouso do SDANT. |
| **Condições de Falha** | (1) O SDANT não decolou; (2) o SDANT não consegue sustentar o enlace. |
| **Observação** | O envio de comandos de pouso é condicionado ao efetivo pouso dos DANTs |

### 6.4.6 POSICIONAR SDANT

| **Caso de Uso** | Posicionar SDANT |
|---|---|
| **Meta** | 1.4.1; 1.4.2; 1.4.3; 1.4.3.1; 1.4.3.2. |
| **Caso de Uso Geral** | |
| **Ator Principal** | |
| **Atores Secundários** | PCM, Ambiente |
| **Resumo** | Comanda a posicionamento dos DANTs. Este posicionamento segue os seguintes passos: determinar a posição inicial do SDANT; calcular a trajetória; deslocar o SDANT ao longo da trajetória determinada; e pairar SDANT. |
| **Pré-Condições** | O SDANT deve ter decolado. |
| **Pós-Condições** | |
| **Ações do Demandante** | **Ações do Sistema** |
| 1 – Solicitar que o SDANT se posicione. | |
| | 2 – Calcular as coordenadas de destino de cada DANT. |
| | 3 – Transmitir comandos de posicionamento para cada DANT. |
| | 4 – Receber as respostas de todos DANTs sobre o cumprimento dos comandos de |

| | |
|---|---|
| | posicionamento. |
| | 5 – Informar ao PCM o posicionamento do SDANT. |
| **Condições de Falha** | (1) O SDANT não decolou; (2) o SDANT não consegue sustentar o enlace. |
| **Observação** | |

### 6.4.7 GERENCIAR MENSAGENS DE CONTROLE DO SDANT

| | |
|---|---|
| **Caso de Uso** | Gerenciar Mensagens de Controle do SDANT |
| **Meta** | 1.8. |
| **Caso de Uso Geral** | |
| **Ator Principal** | |
| **Atores Secundários** | PCM |
| **Resumo** | Este caso de uso centraliza a recepção e distribuição das mensagens operacionais e de controle no SDANT. Seus objetivos são: detectar as mensagens oriundas do ambiente do SDANT, encaminhar estas mensagens aos módulos do SDANT de destino, receber as mensagens dos módulos do SDANT, encaminhar estas mensagens ao PCM e registrar as mensagens no diário de bordo. |
| **Pré-Condições** | O SDANT deve estar ativo. |
| **Pós-Condições** | |
| **Condições de Falha** | |

### 6.4.8 FAZER LOGIN

| | |
|---|---|
| **Caso de Uso** | Fazer Login |
| **Meta** | 1.3.1;1.3.1.1. |
| **Caso de Uso Geral** | |
| **Ator Principal** | Pontos IP |
| **Atores Secundários** | |
| **Resumo** | Analisar solicitações de login autorizando o uso da conexão com base nos dados de habilitação de usuário e analisar solicitações de logout autorizando o fim da conexão. |
| **Pré-Condições** | O ponto IP a partir do qual a solicitação é feita deve ser coberta pelo SDANT. |
| **Pós-Condições** | |
| **Condições de Falha** | |

### 6.4.9 HABILITAR ACESSO INTERNET

| | |
|---|---|
| **Caso de Uso** | Habilitar Acesso Internet |
| **Meta** | 1.2.1. |
| **Caso de Uso Geral** | |
| **Ator Principal** | Pontos IP |
| **Atores Secundários** | PCM |
| **Resumo** | Efetuar a habilitação do usuário que lhe permite conectar-se a rede Internet por intermédio do SDANT. |
| **Pré-Condições** | O Ponto IP a partir do qual a solicitação é feita deve ser coberta pelo SDANT. |
| **Pós-Condições** | |
| **Condições de Falha** | |

## 6.5 DIAGRAMAS DE SEQÜÊNCIA

Nos subitens a seguir são apresentados os diagramas de seqüência do sistema.

### 6.5.1 MONITORAR AMBIENTE

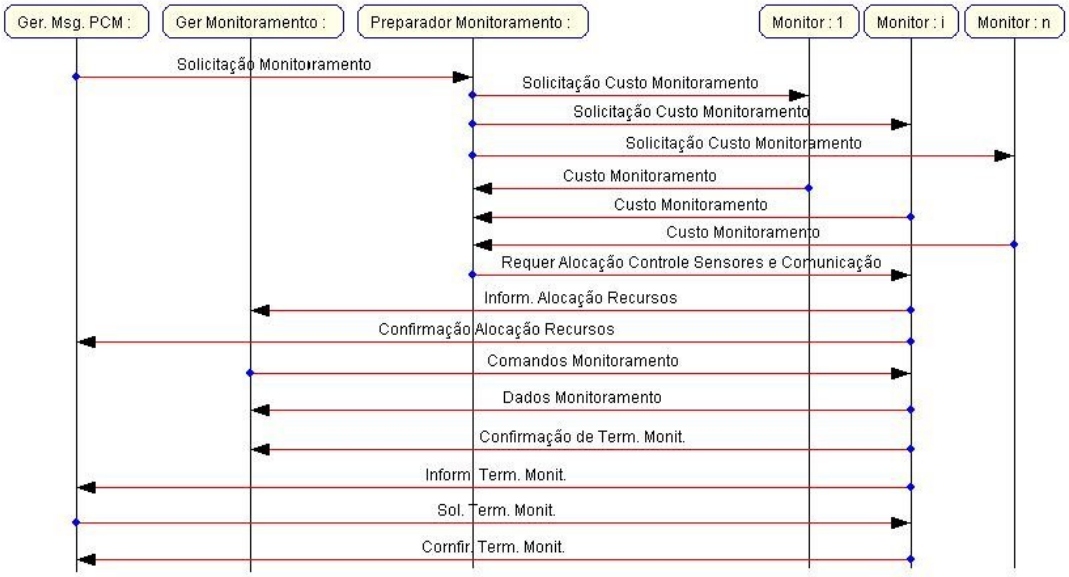


FIG 6. 3: Monitorar Ambiente

### 6.5.2 PROVER ACESSO INTERNET

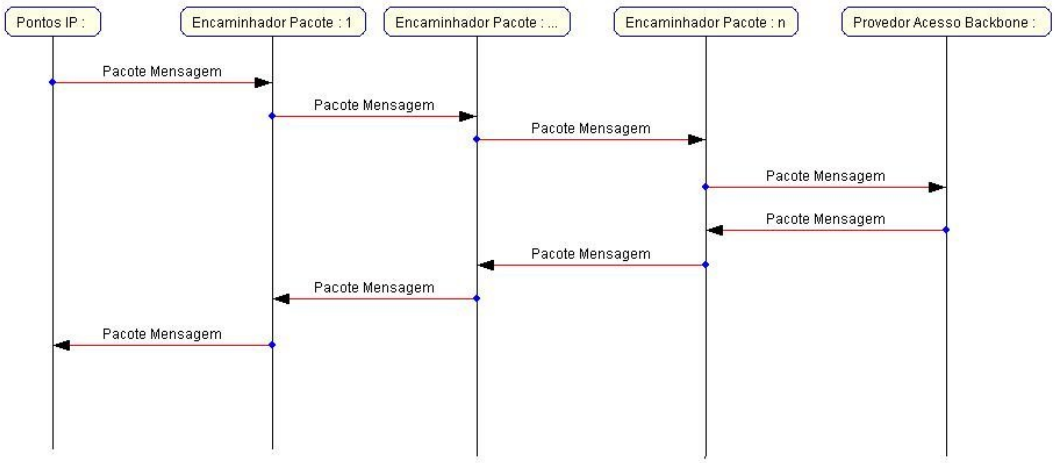


FIG 6. 4: Prover Acesso

### 6.5.3 FORNECER MENSAGENS DO SDANT

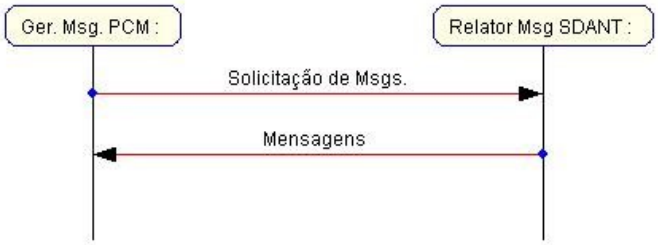


FIG 6. 5: Fornecer Mensagens

## 6.5.4 DECOLAR SDANT

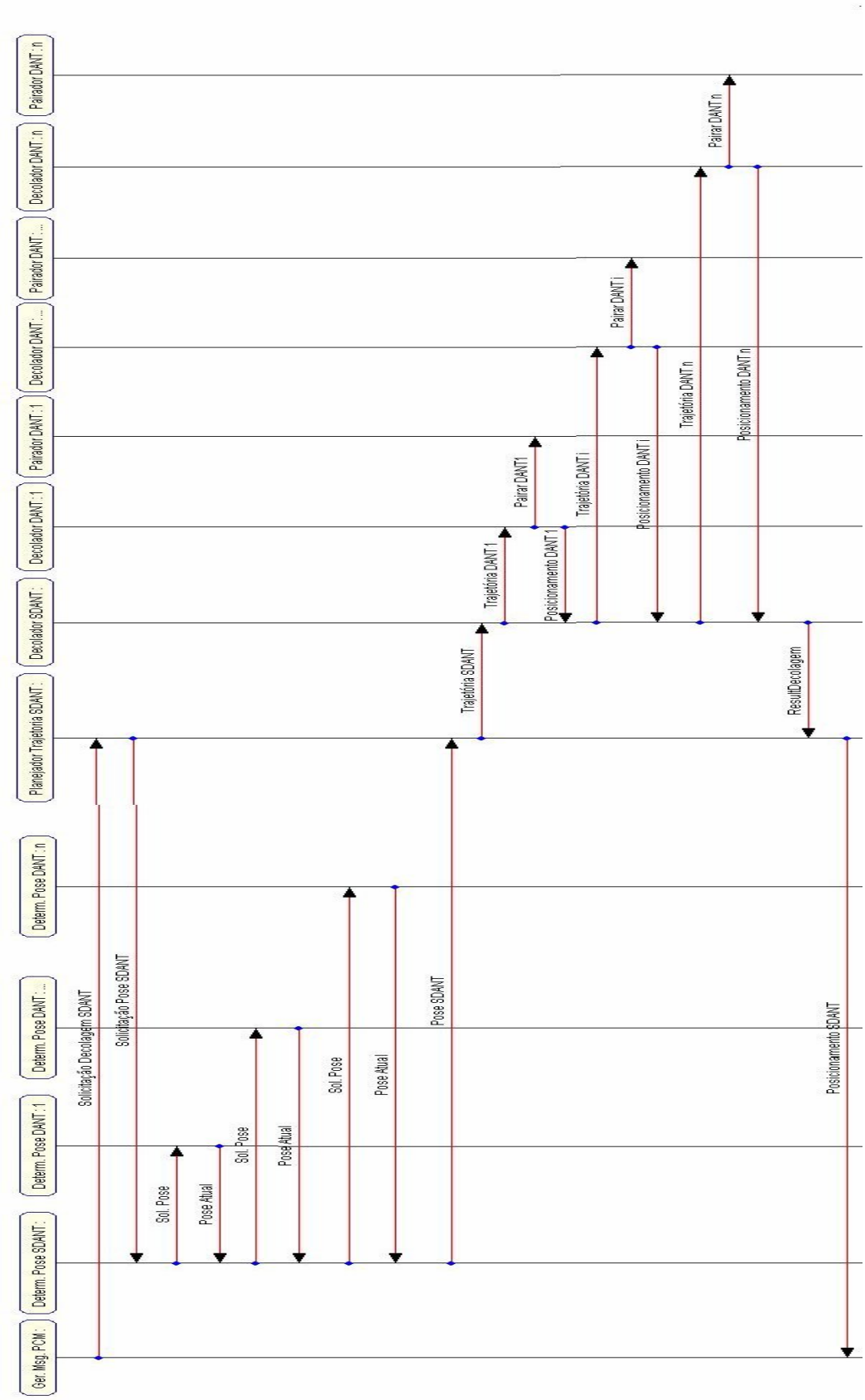


FIG 6. 6: Decolar SDANT

### 6.5.5 POUSAR SDANT

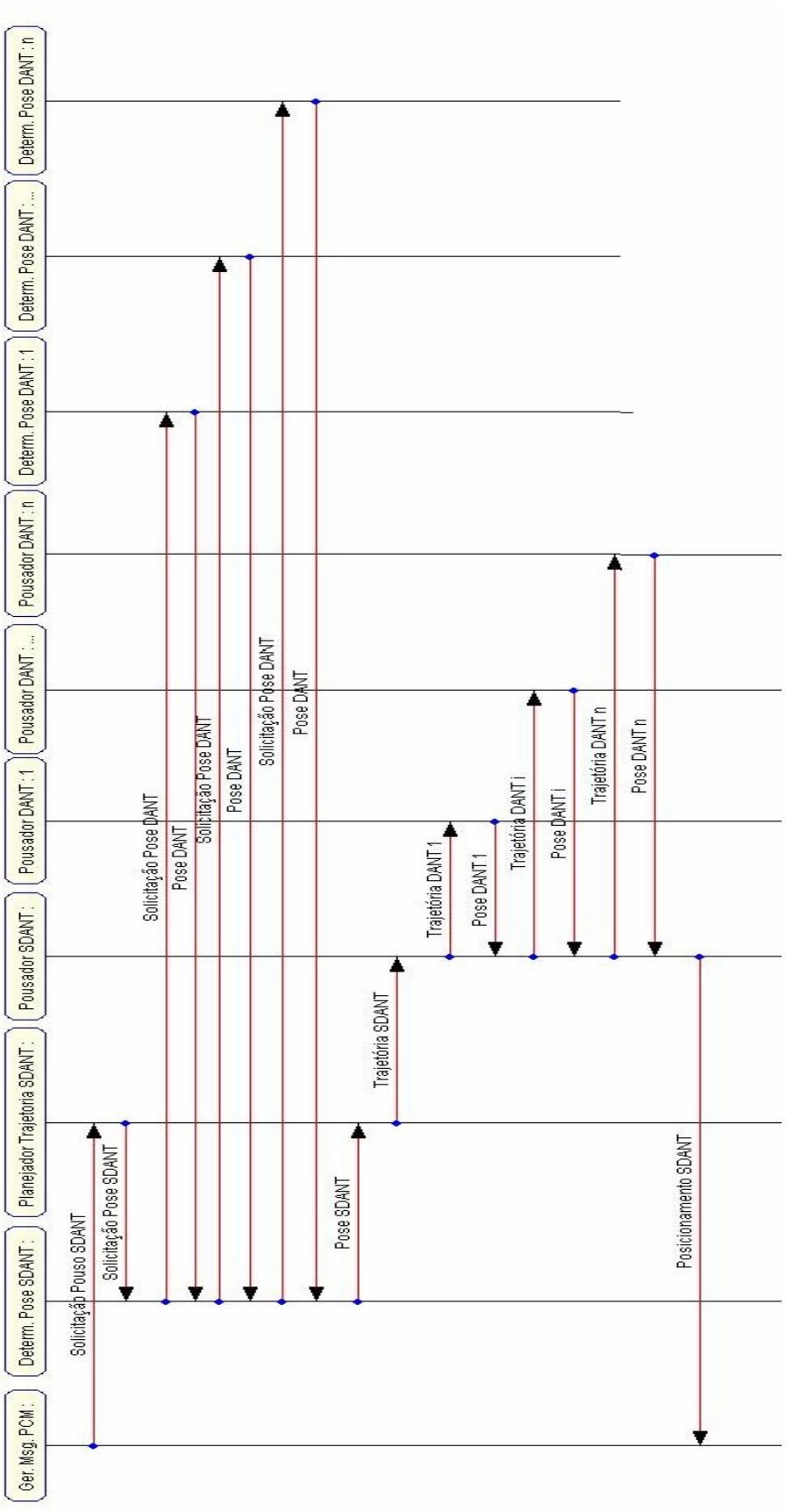


FIG 6. 7: Pousar SDANT

## 6.5.6 POSICIONAR SDANT

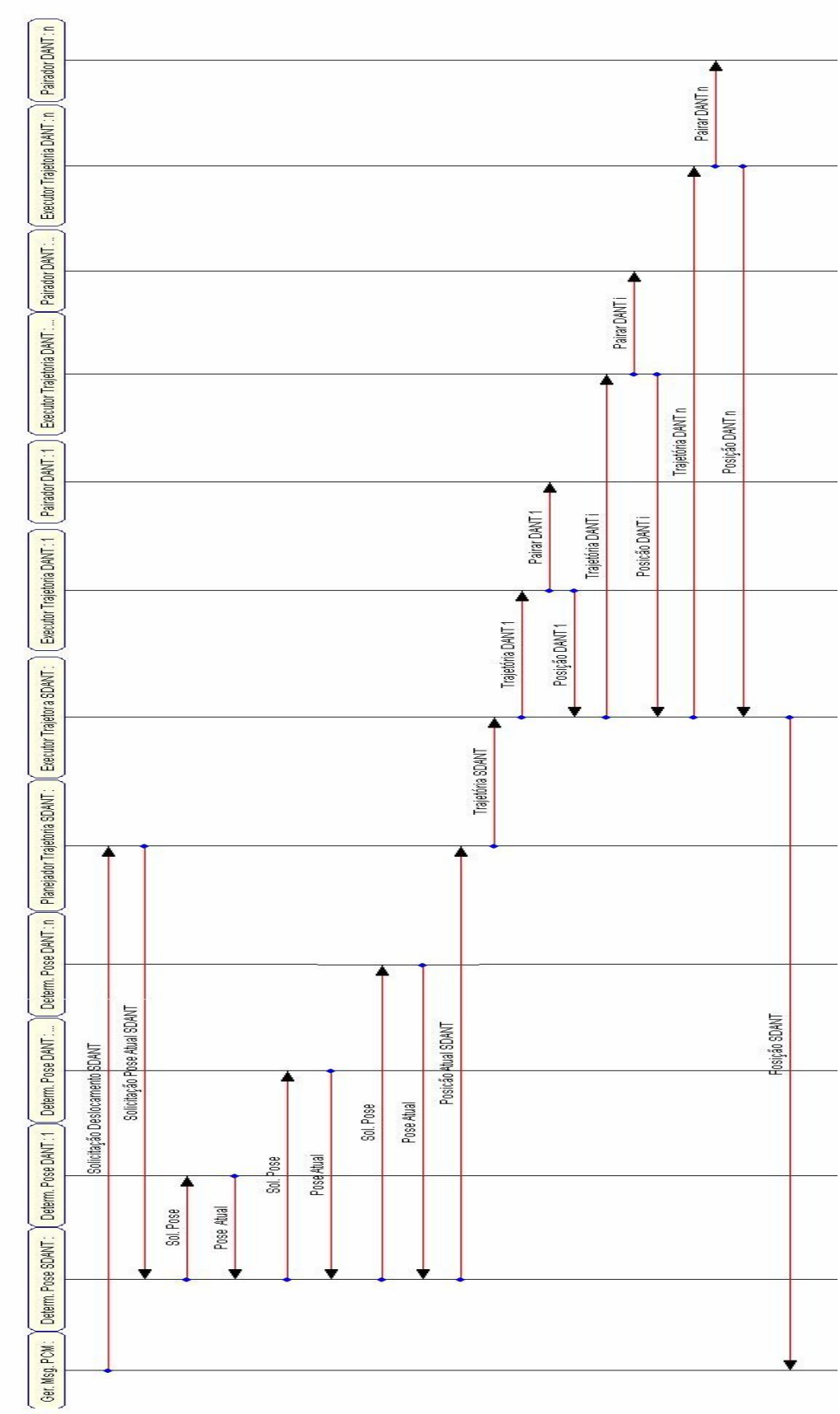


FIG 6. 8: Posicionar SDANT

### 6.5.7 GERENCIAR MENSAGENS DE CONTROLE DO SDANT

Nos próximos subitens as seqüências de Gerenciar Mensagens.

#### 6.5.7.1 PRIMEIRA SEQÜÊNCIA


FIG 6. 9: Solicitação de Emissão Mensagens

#### 6.5.7.2 SEGUNDA SEQÜÊNCIA


FIG 6. 10: Mensagens para monitoramento

### 6.5.8 FAZER LOGIN

Nos subitens seguintes as seqüências de Fazer Login.

### 6.5.8.1 PRIMEIRA SEQÜÊNCIA

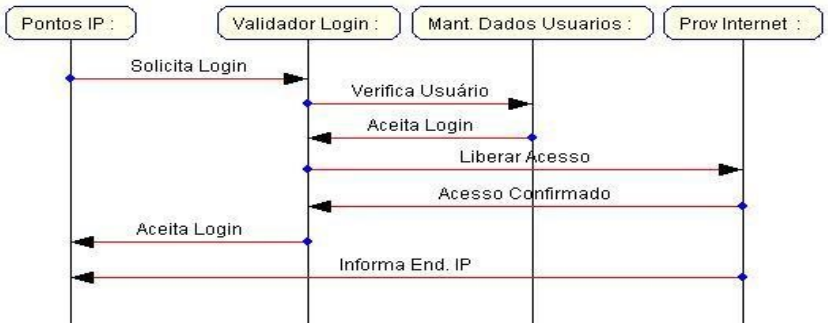


FIG 6. 11: Login Aceito

### 6.5.8.2 SEGUNDA SEQÜÊNCIA

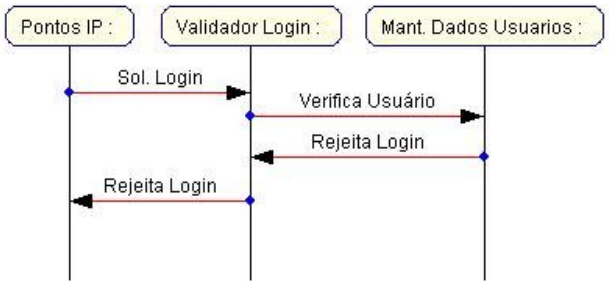


FIG 6. 12: Login negado por Validador Login

### 6.5.8.3 TERCEIRA SEQÜÊNCIA

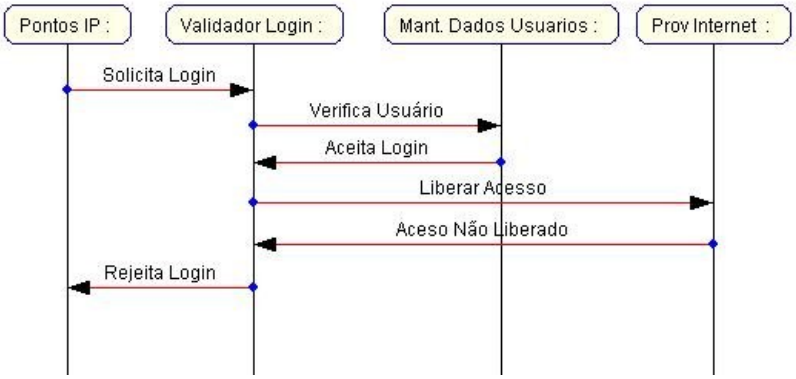


FIG 6. 13: Login negado por Prov. Internet

### 6.5.8.4 QUARTA SEQÜÊNCIA


FIG 6. 14: Logout

## 6.5.9 HABILITAR ACESSO INTERNET


FIG 6. 15: Habilitar Acesso Internet

## 6.6 DIAGRAMA DE PAPÉIS

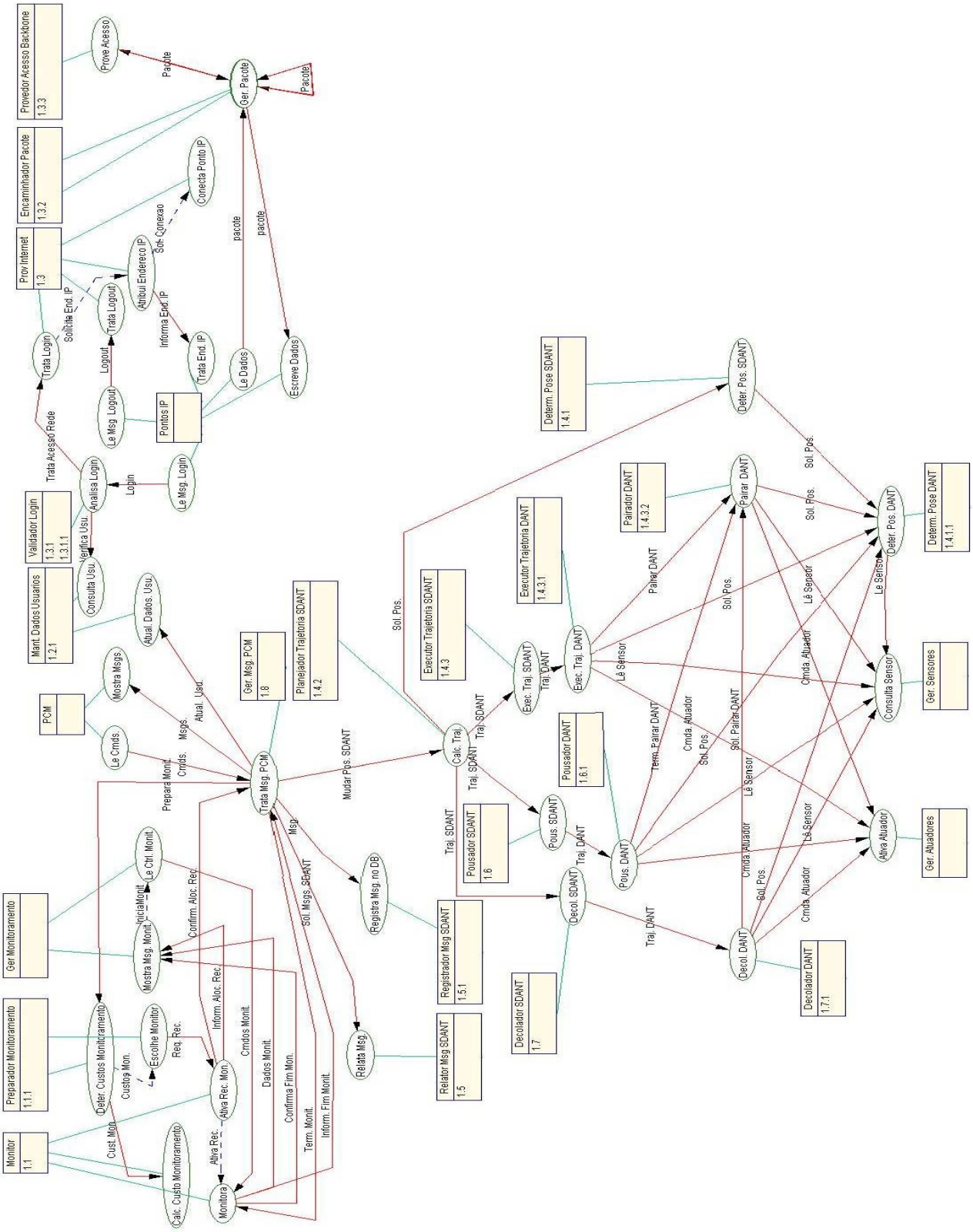


FIG 6. 16: Digrama de Papéis

## 6.7 DIAGRAMAS DE TAREFAS CONCORRENTES

Nos subitens a seguir são apresentados os diagramas de tarefas concorrentes do sistema.

### 6.7.1 DETER. CUSTOS MONITORAMENTO

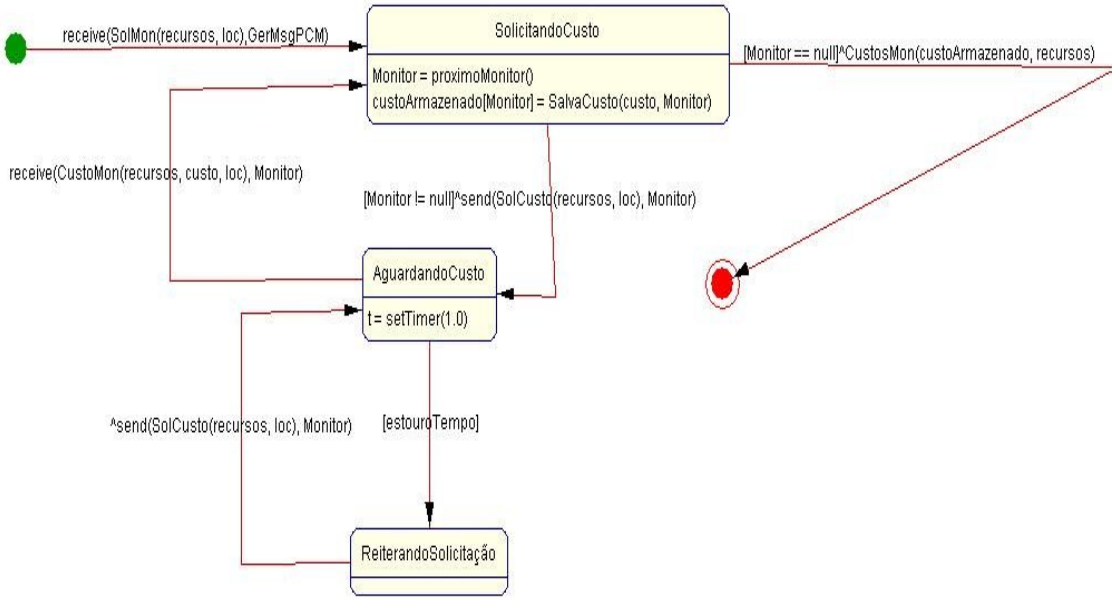


FIG 6. 17: Deter. Custos Monitoramento

### 6.7.2 CALC. CUSTO MONITORAMENTO

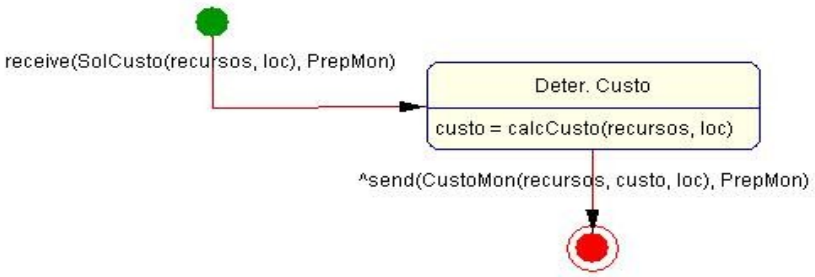


FIG 6. 18: Calc. Custo Monitoramento

### 6.7.3 MONITORA

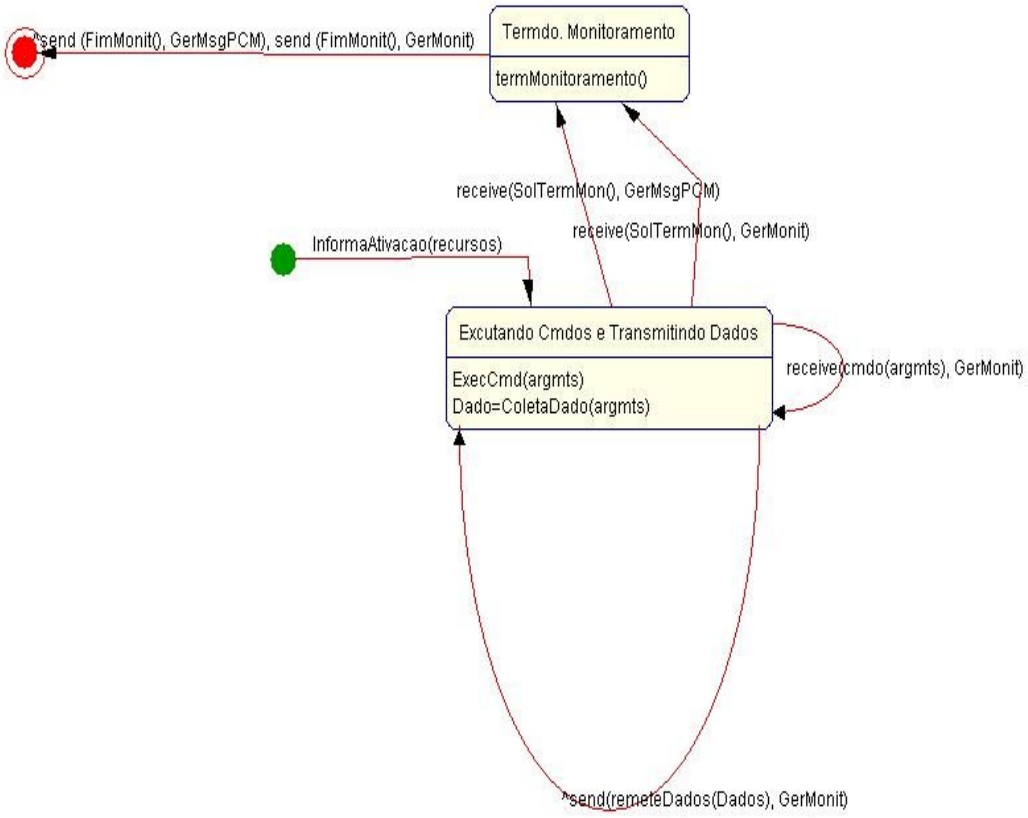


FIG 6. 19: Monitora

### 6.7.4 ATIVA REC. MON

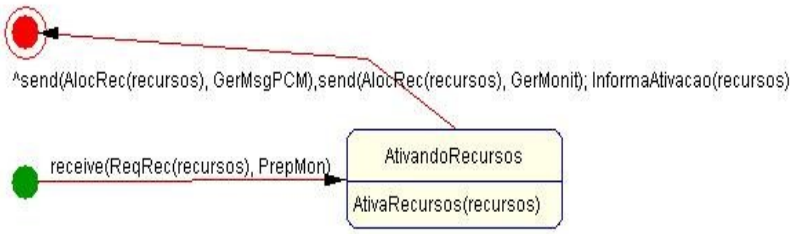


FIG 6. 20: Ativa Rec. Mon

### 6.7.5 ESCOLHE MONITOR

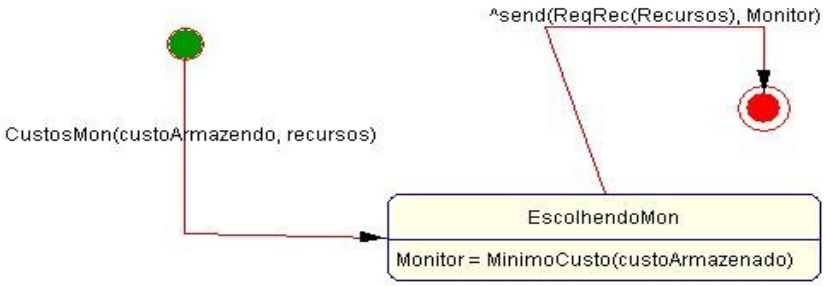


FIG 6. 21: Escolhe Monitor

### 6.7.6 MOSTRA MSG. MONIT

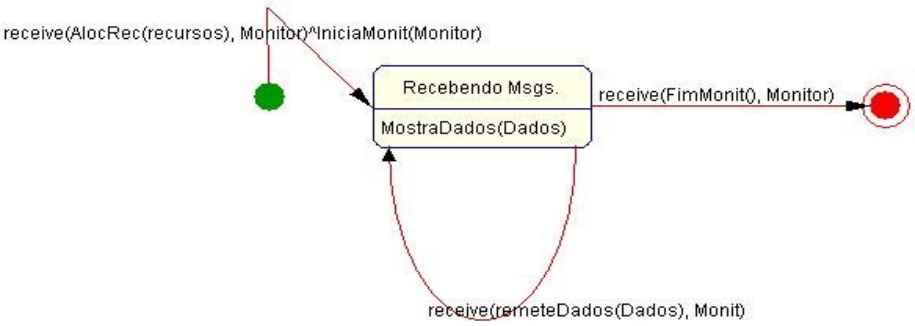


FIG 6. 22: Mostra Msg. Monit

### 6.7.7 LÊ CTRL. MONIT


FIG 6. 23: Lê Ctrl. Monit.

## 6.7.8 TRATA MSG. PCM

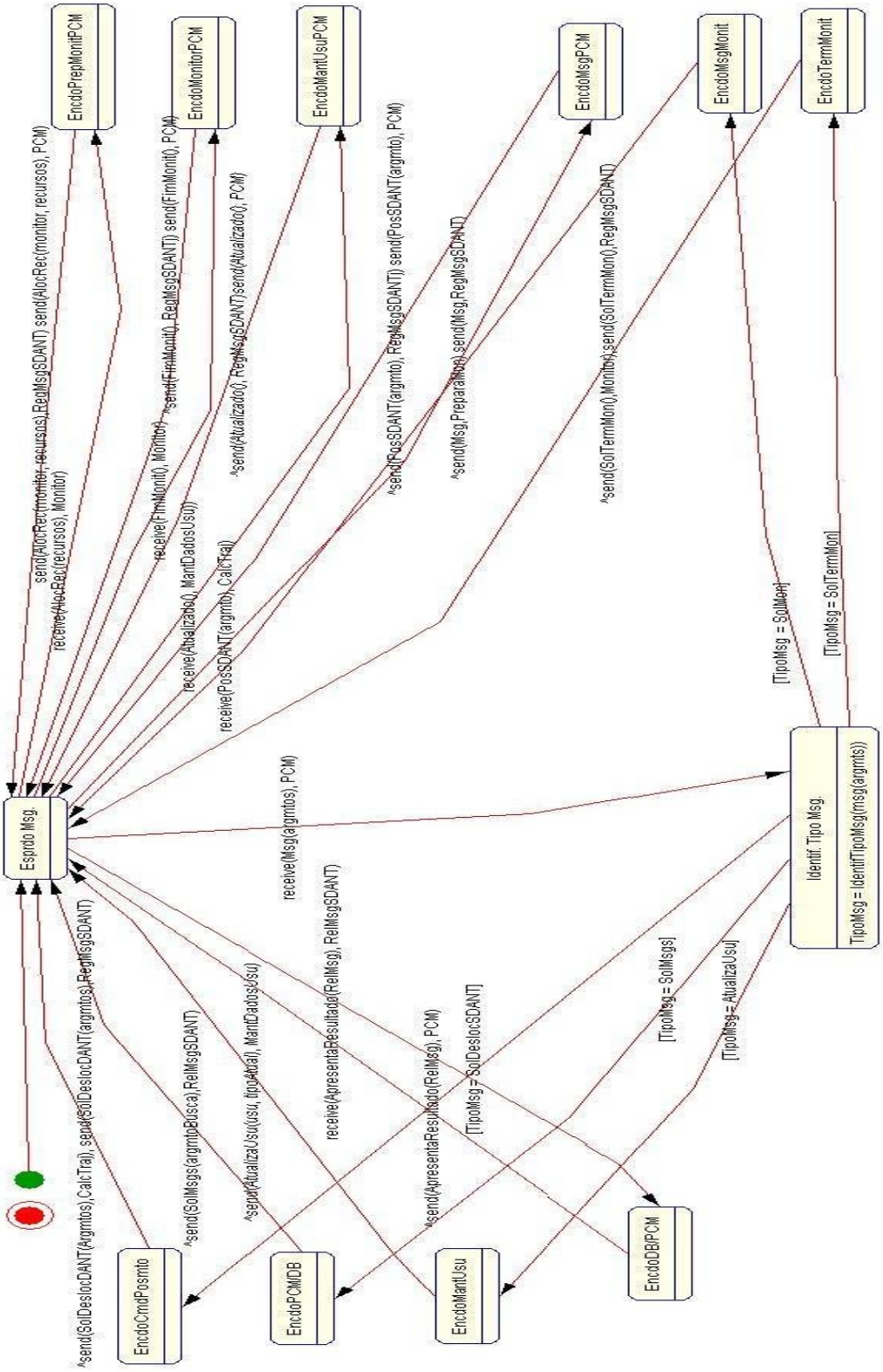


FIG 6. 24: Trata Msg. PCM

### 6.7.9 RELATA MSG.

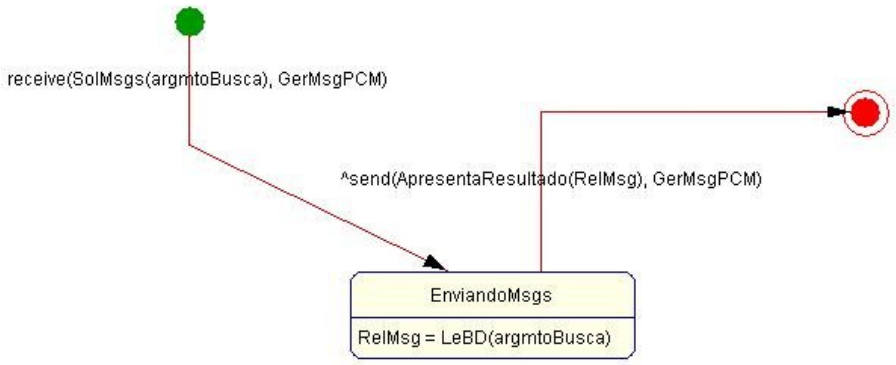


FIG 6. 25: Relata Msg.

### 6.7.10 REGISTRA MSG. NO DB.

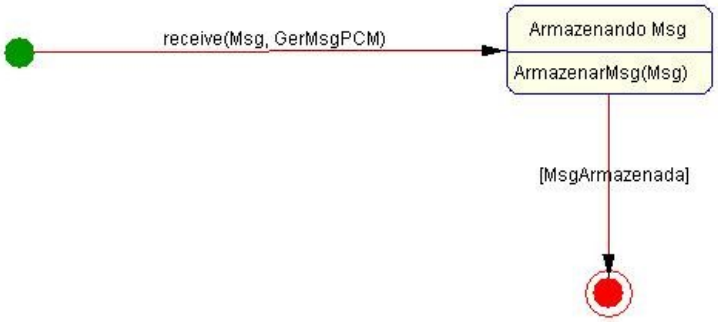


FIG 6. 26: Registra Msg. no DB

### 6.7.11 LÊ CMDS.

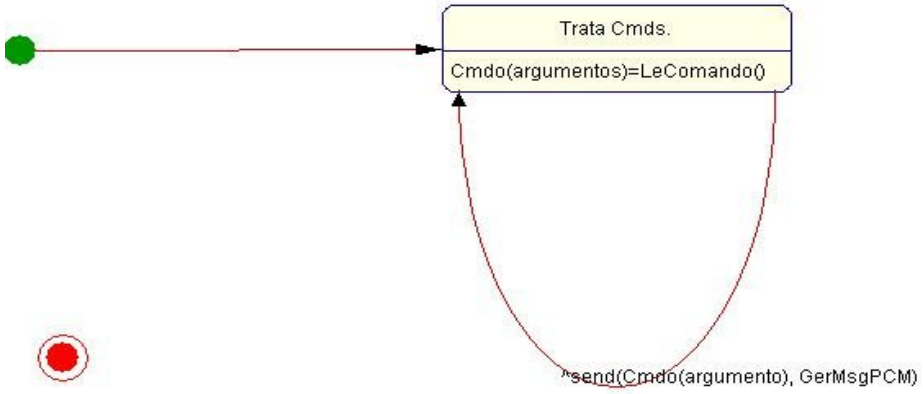


FIG 6. 27: Lê Cmds.

### 6.7.12 MOSTRA MSGS.

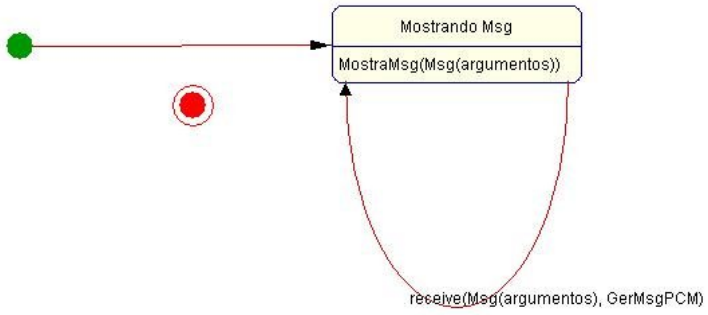


FIG 6. 28: Mostra Msgs.

### 6.7.13 ATUAL. DADOS. USU.

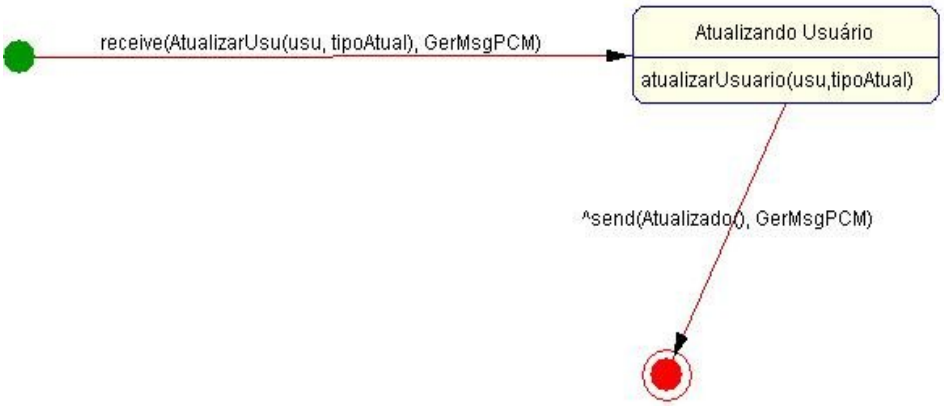


FIG 6. 29: Atual. Dados. Usu.

### 6.7.14 CONSULTA USU.

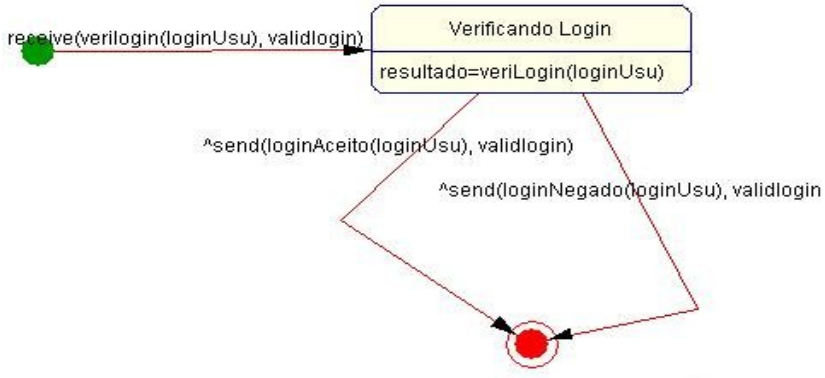


FIG 6. 30: Consulta Usu.

### 6.7.15 ANALISA LOGIN

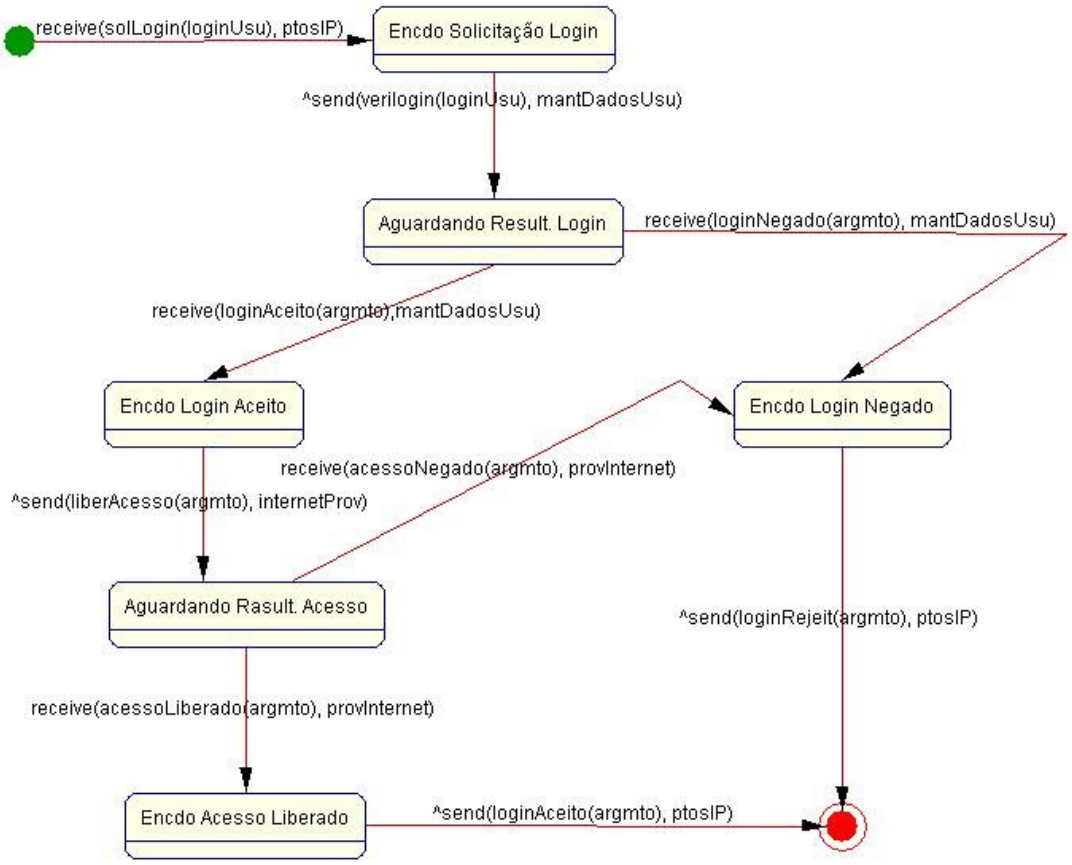


FIG 6. 31: Analisa Login

### 6.7.16 LÊ MSG. LOGIN

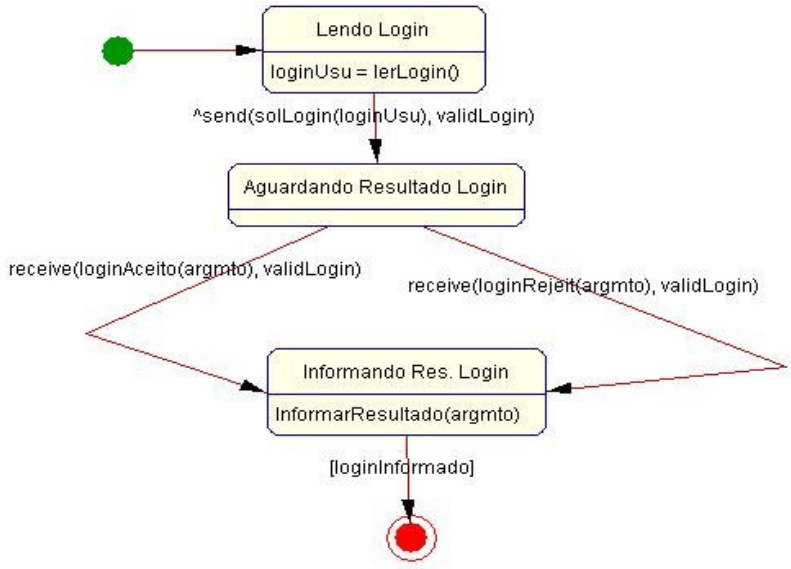


FIG 6. 32: Lê Msg. Login

### 6.7.17 LÊ MSG LOGOUT

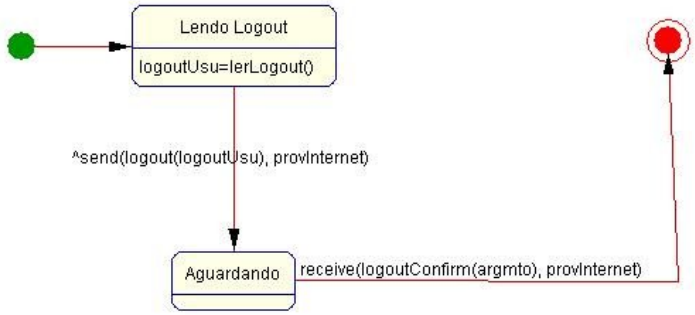


FIG 6. 33: Lê Msg. Logout

### 6.7.18 TRATA LOGOUT

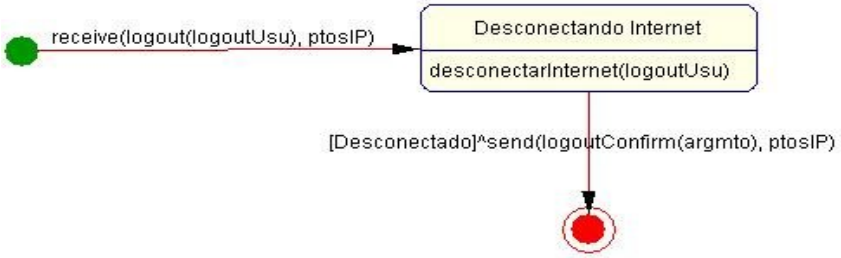


FIG 6. 34: Trata Logout

### 6.7.19 ATRIBUI ENDERECO IP

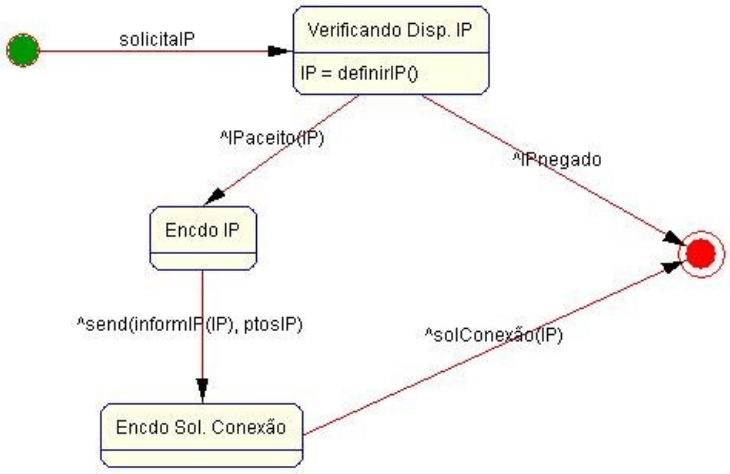


FIG 6. 35: Atribui Endereço IP

### 6.7.20 CONECTA PONTO IP

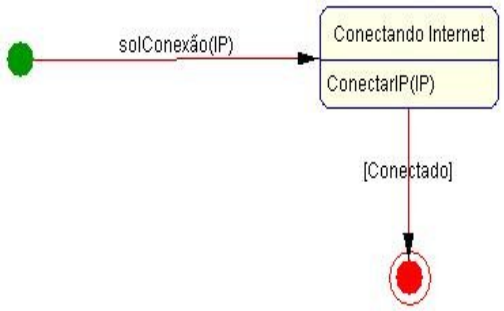


FIG 6. 36: Conecta Ponto IP

### 6.7.21 TRATA END. IP

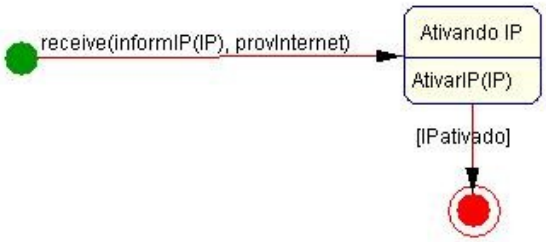


FIG 6. 37: Trata End. IP

### 6.7.22 LÊ DADOS

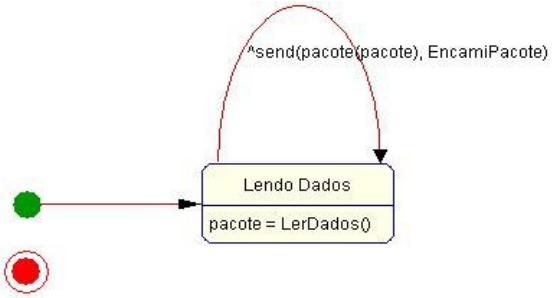


FIG 6. 38: Lê Dados

### 6.7.23 ESCREVE DADOS

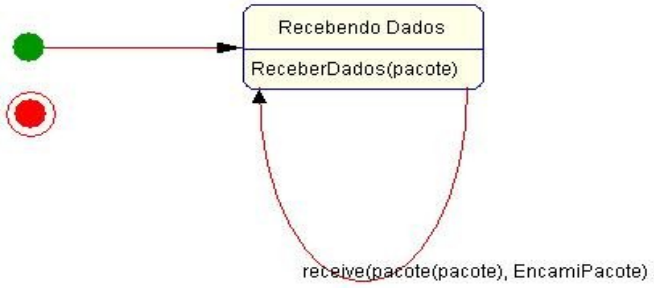


FIG 6. 39: Escreve Dados

### 6.7.24 GER. PACOTE

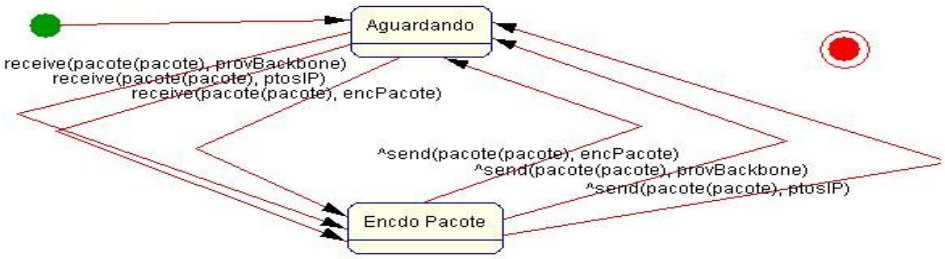


FIG 6. 40: Ger. Pacote

### 6.7.25 PROVÊ ACESSO

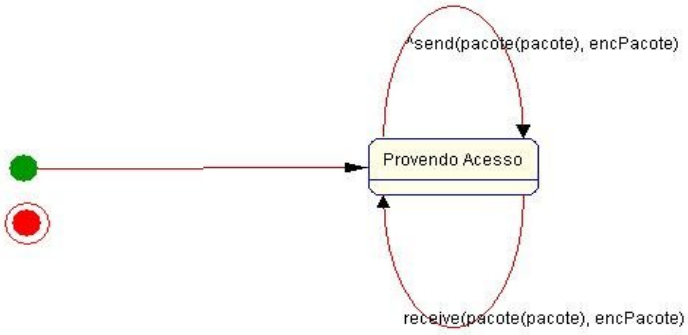


FIG 6. 41: Provê Acesso

## 6.7.26 CALC. TRAJ.

.

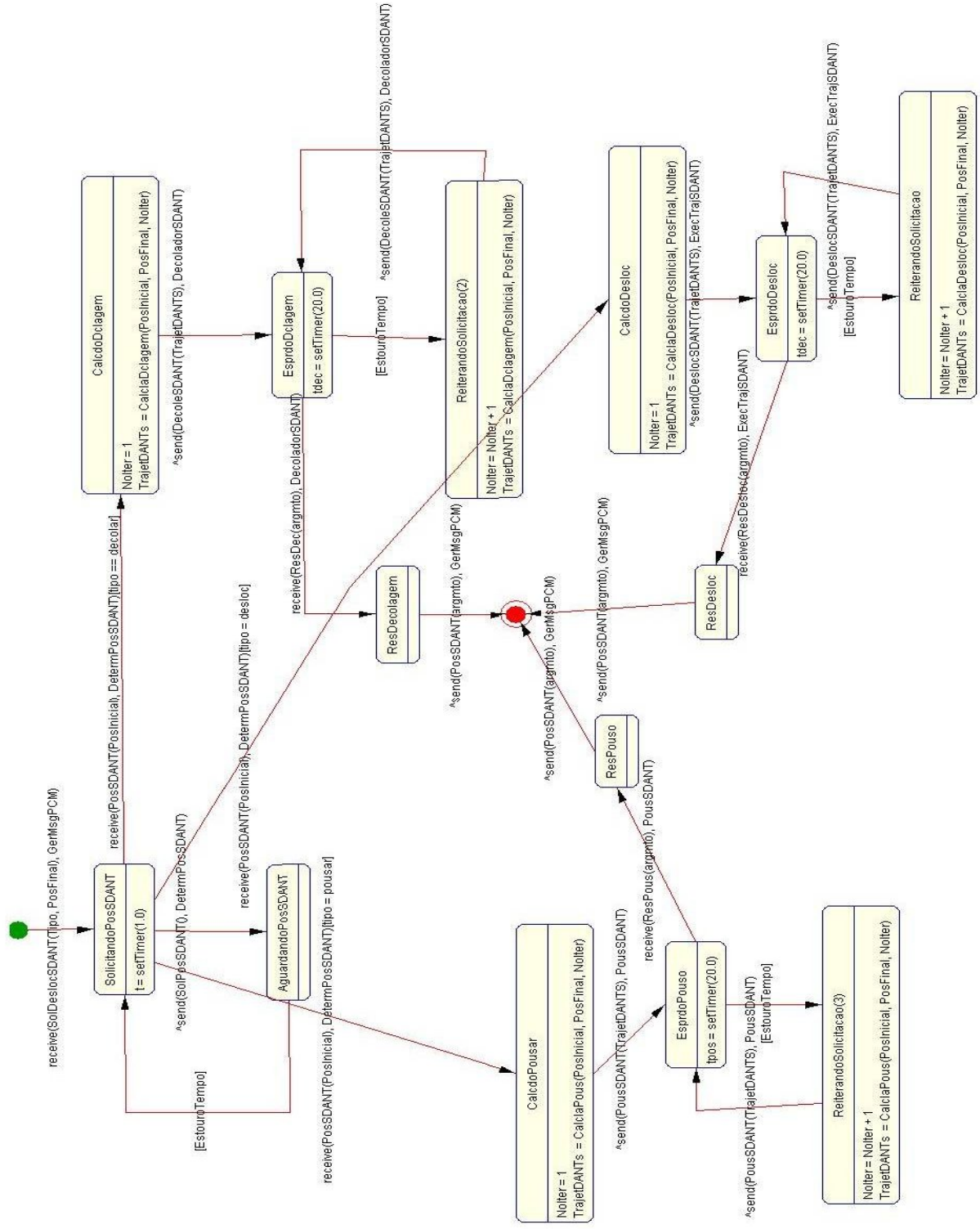


FIG 6. 42: Calc. Traj.

## 6.7.27 EXEC. TRAJ. SDANT

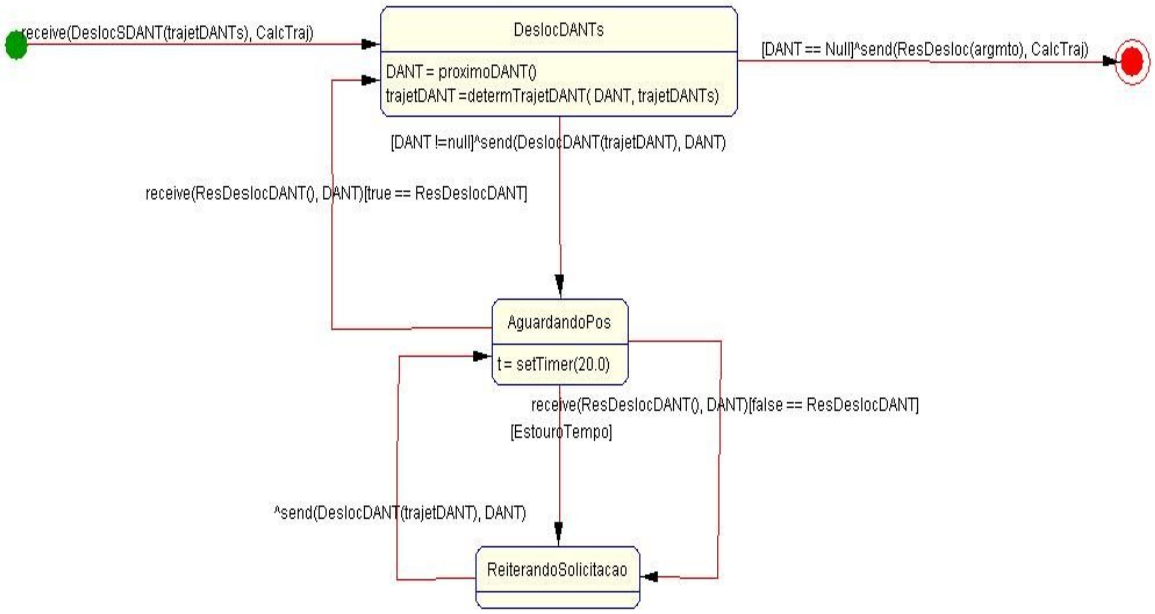


FIG 6. 43: Exec. Traj. SDANT

## 6.7.28 POUS. SDANT

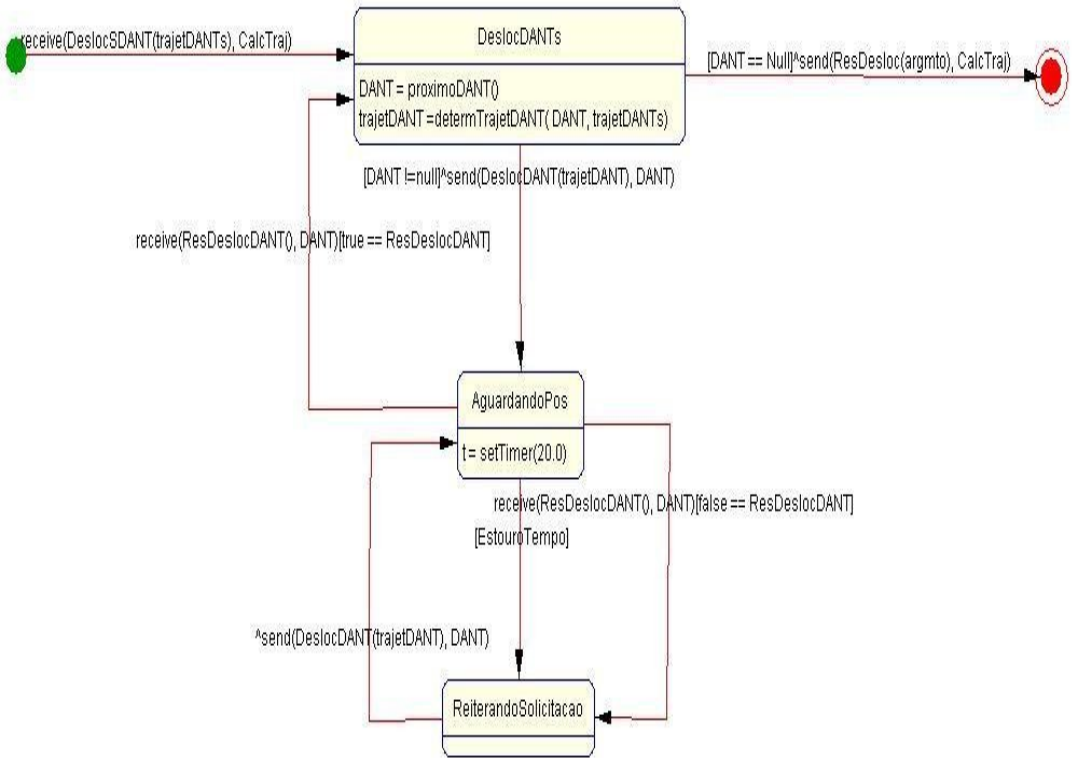


FIG 6. 44: Pous. SDANT

### 6.7.29 DECOL. SDANT

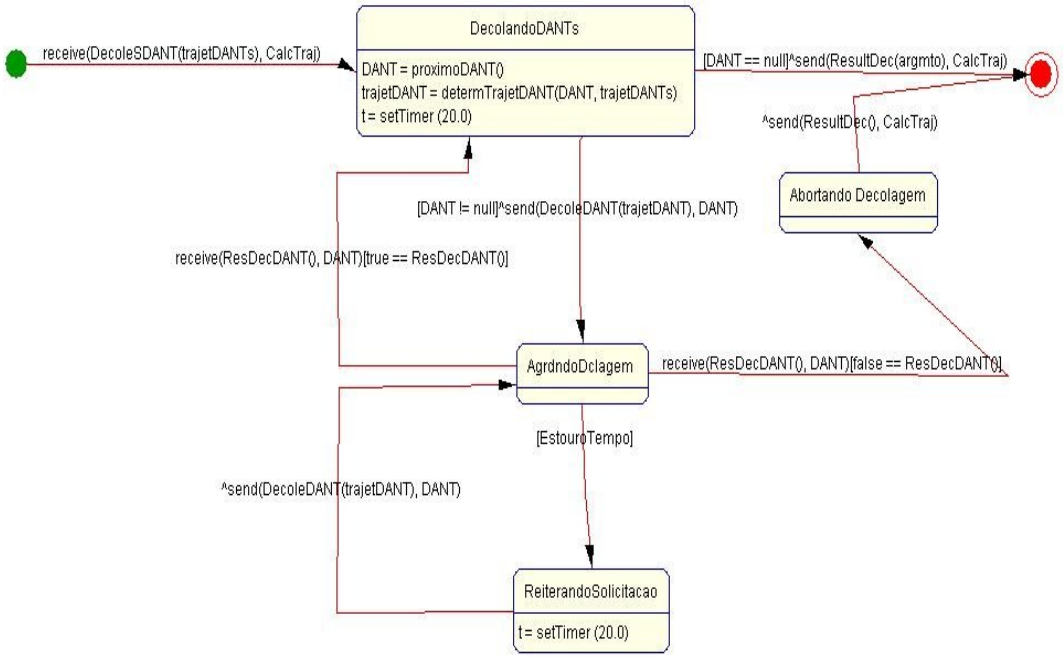


FIG 6. 45: Decol. SDANT

## 6.7.30 POUS. DANT

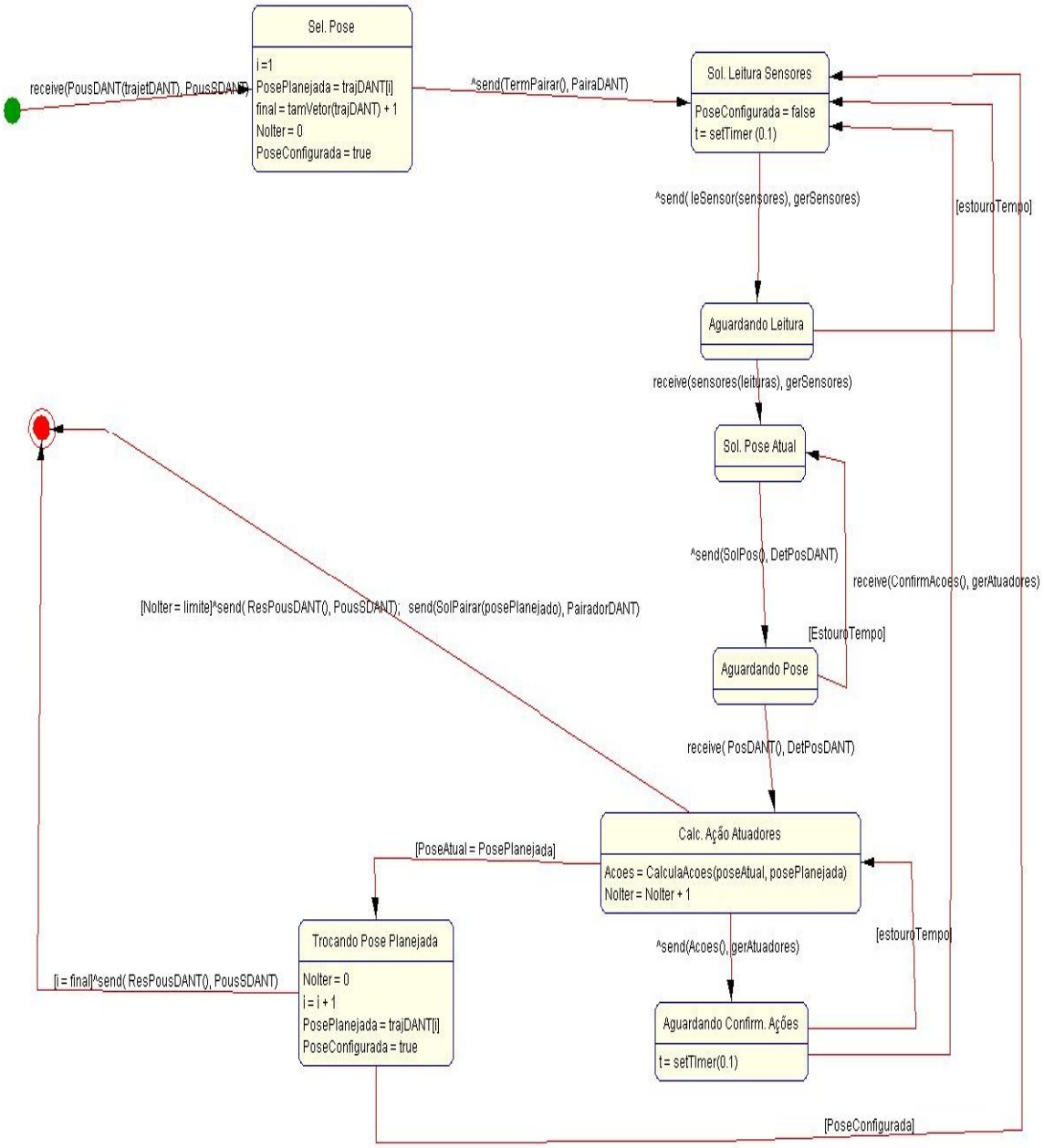


FIG 6. 46: Pous. DANT

### 6.7.31 DECOL. DANT

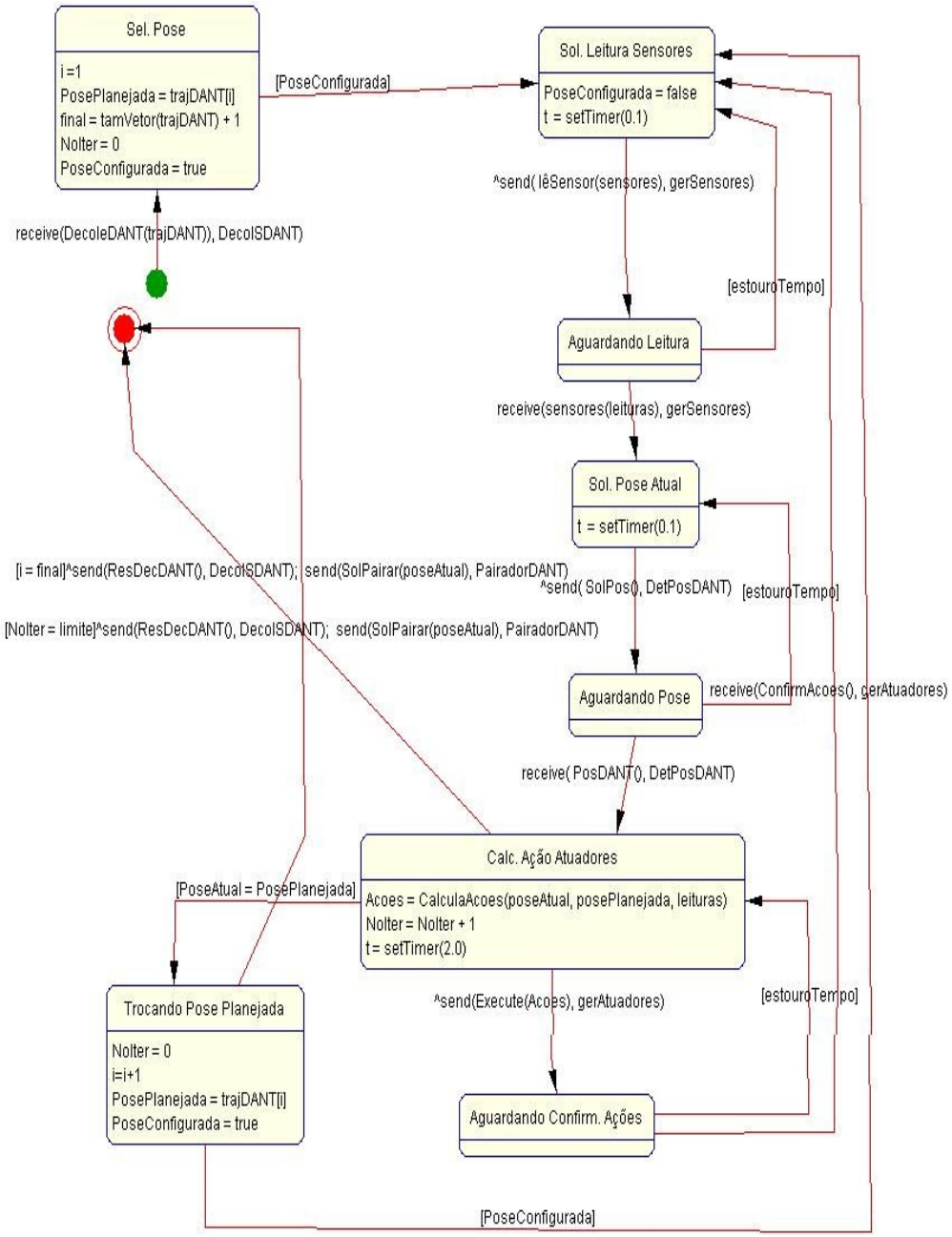


FIG 6. 47: Decol. DANT

### 6.7.32 EXEC. TRAJ. DANT

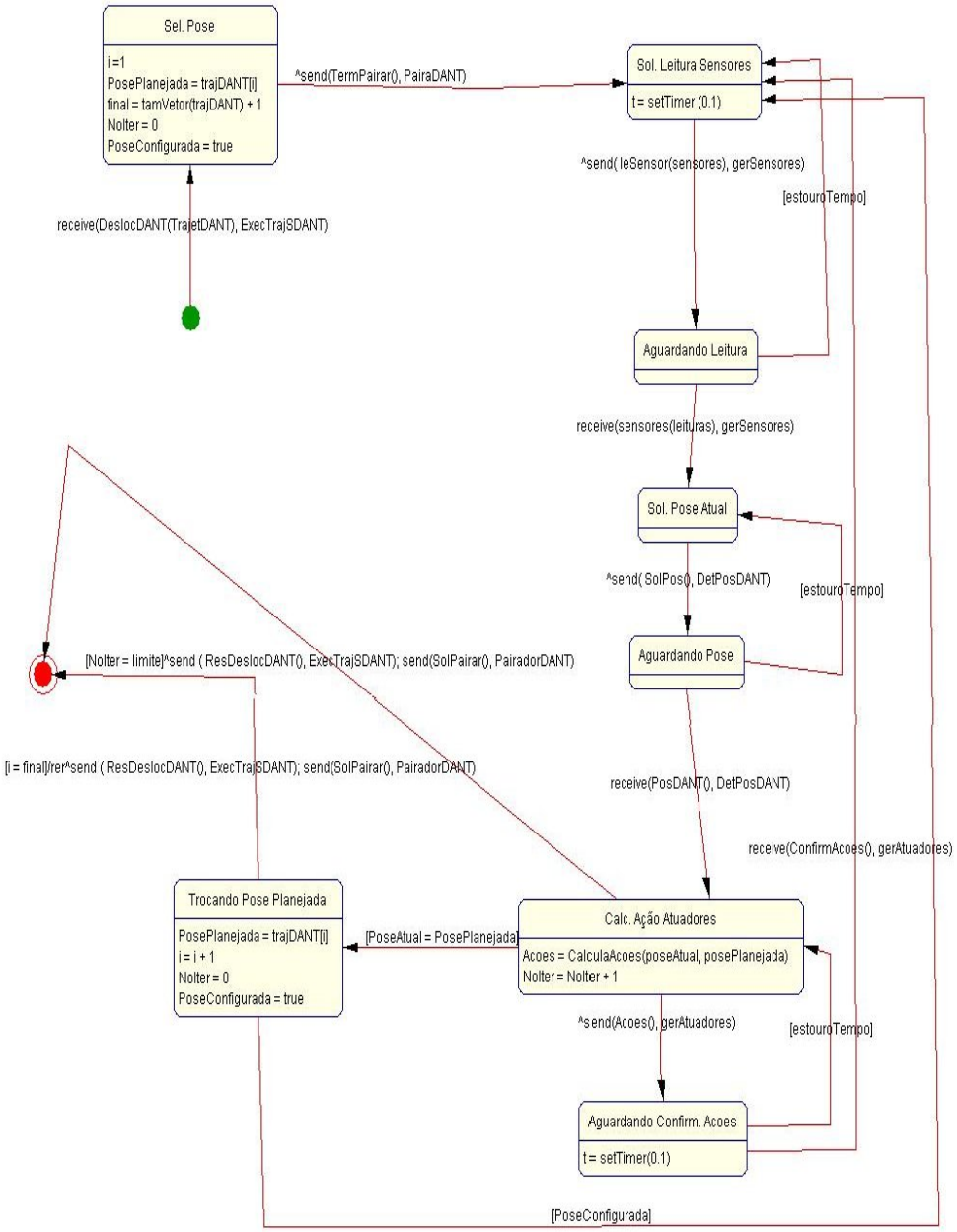


FIG 6. 48: Exec. Traj. DANT

## 6.7.33 PAIRAR DANT

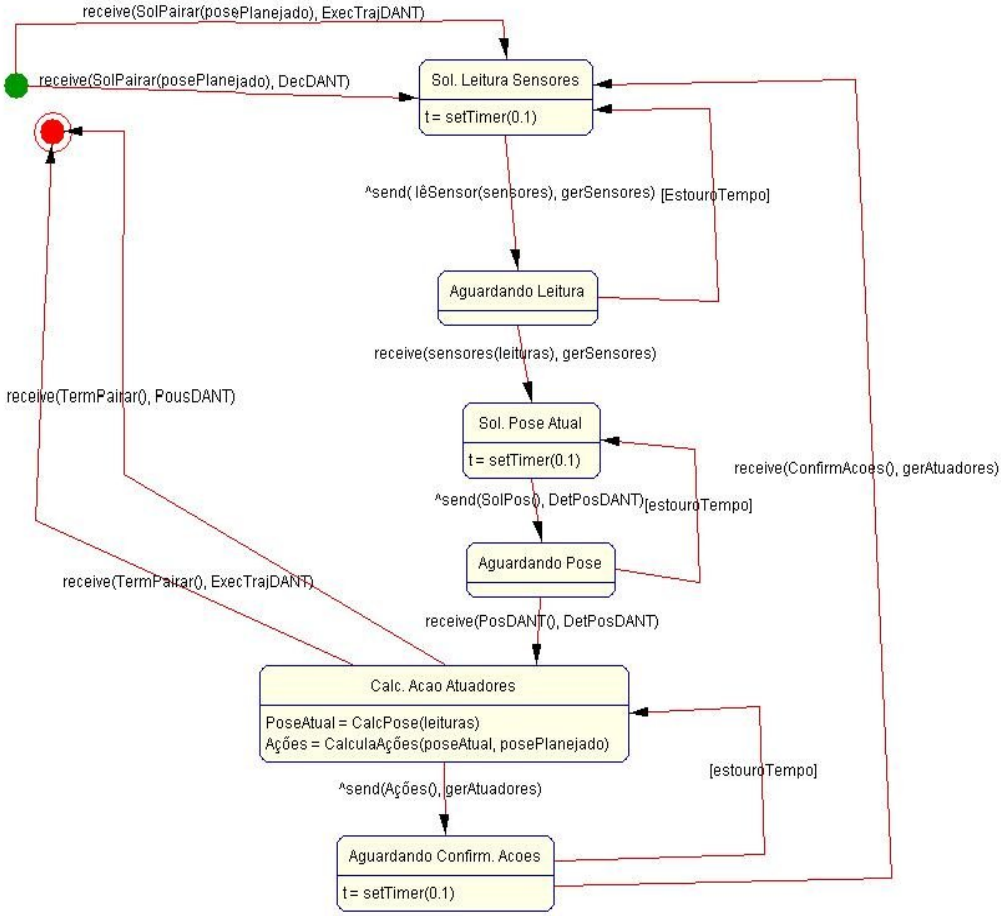


FIG 6. 49: Pairar DANT

## 6.7.34 ATIVA ATUADOR

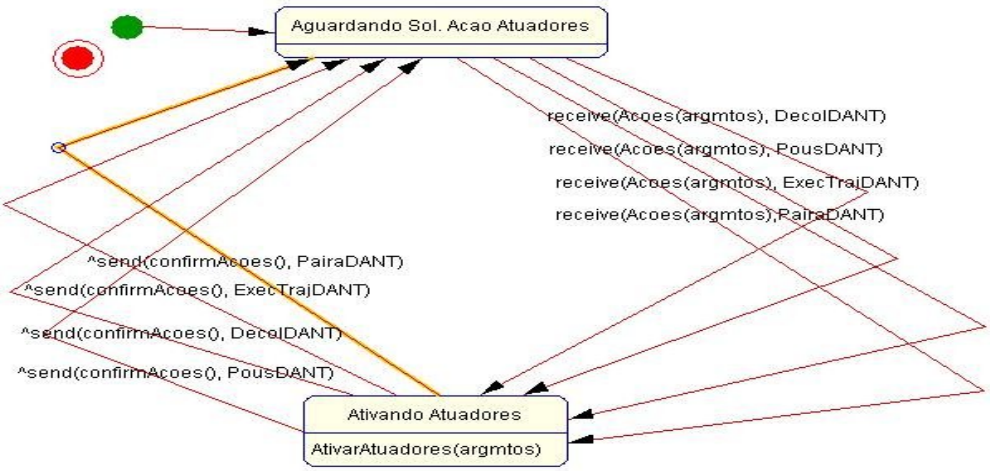


FIG 6. 50: Ativa Atuador

## 6.7.38 CONSULTA SENSOR

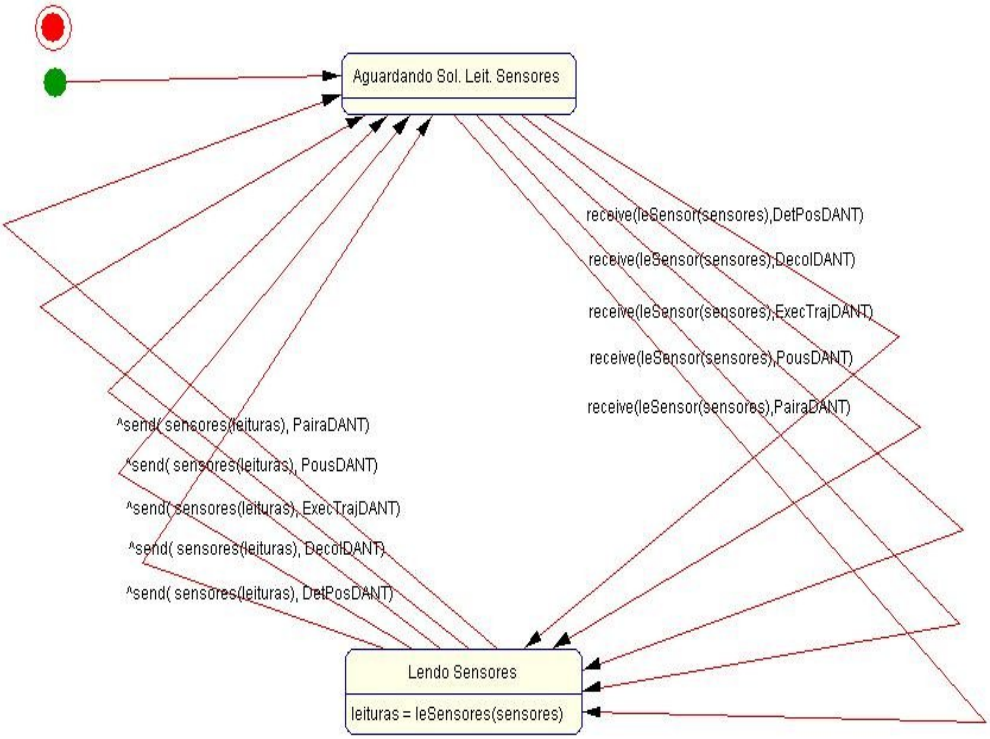


FIG 6. 51: Consulta Sensor

## 6.7.39 DETER. POS. DANT

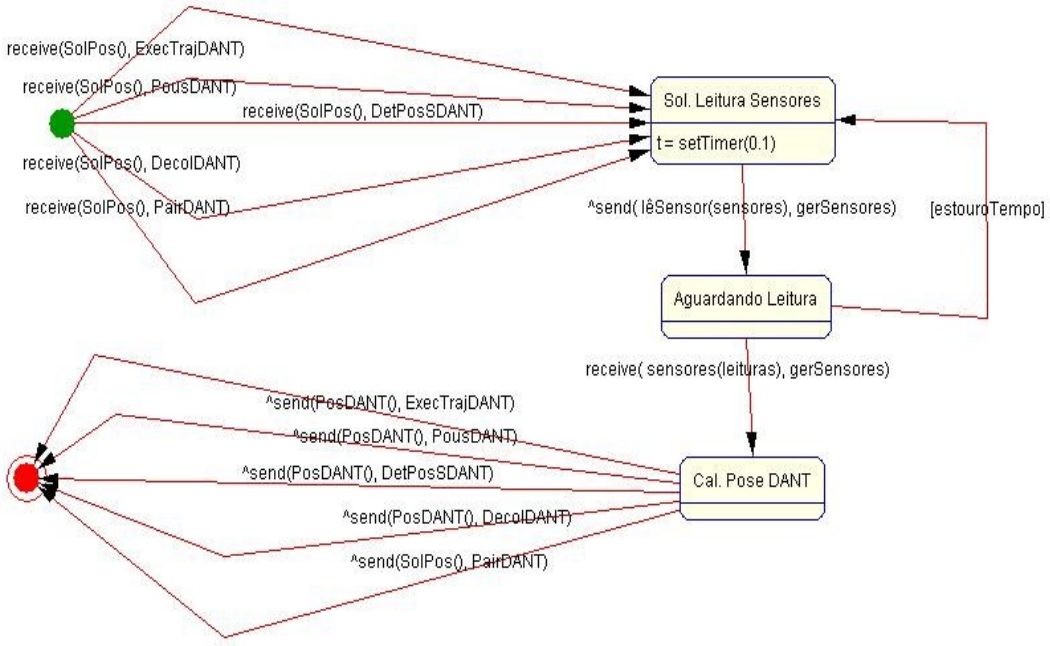


FIG 6. 52: Deter. Pos. DANT

### 6.7.40 DETER. POS. SDANT.

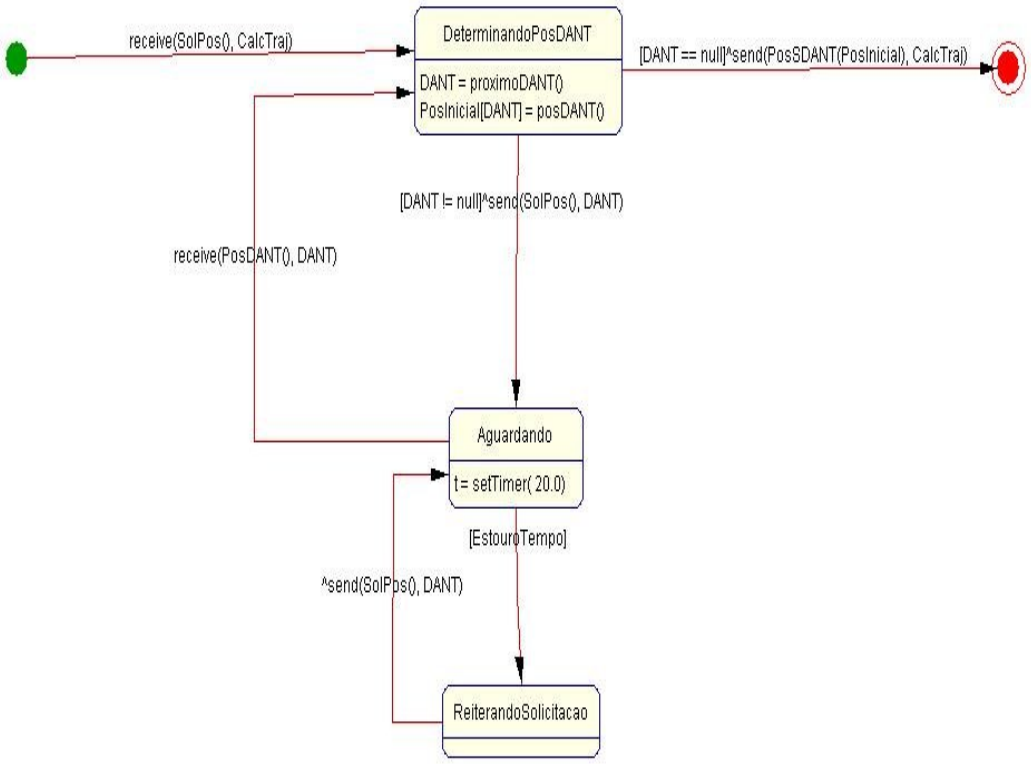


FIG 6. 53: Deter. Pos. DANT

## 6.8 DIAGRAMA DE CLASSES DE AGENTE

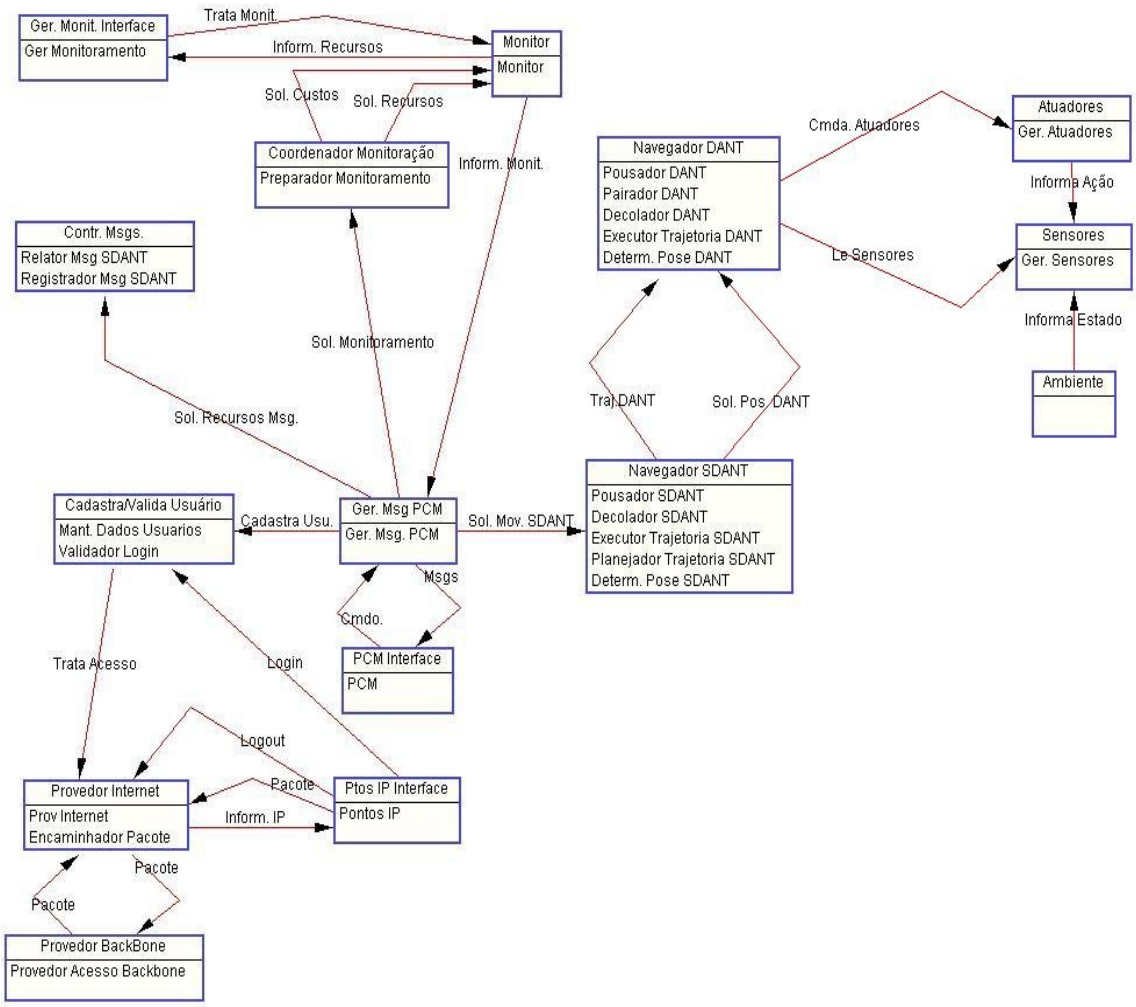


FIG 6. 54: Diagrama de Classes de Agente

## 6.9 DIAGRAMAS DE CLASSES DE CONVERSAÇÃO

Nos subitens a seguir os diagramas de classes de conversação.

### 6.9.1 CADASTRA USU.INITIATOR

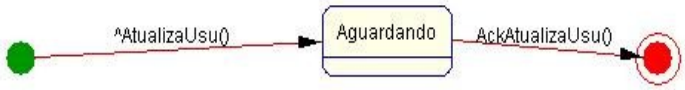


FIG 6. 55: Cadastra Usu. Initiator

### 6.9.2 CADASTRA USU.RESPONDER

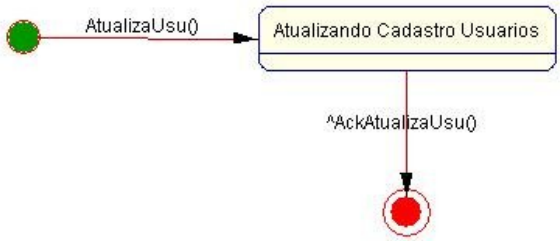


FIG 6. 56: Cadastra Usu. Responder

### 6.9.3 CMDA.ATUADORESINITIATOR

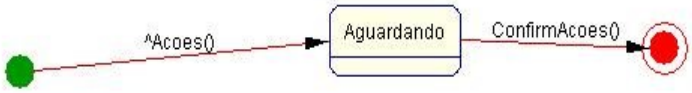


FIG 6. 57: Cmda. Atuadores Initiator

### 6.9.4 CMDA.ATUADORESRESPONDER

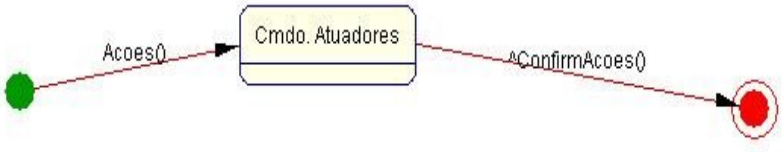


FIG 6. 58: Cmda. Atuadores Responder

### 6.9.5 CMDO.INITIATOR

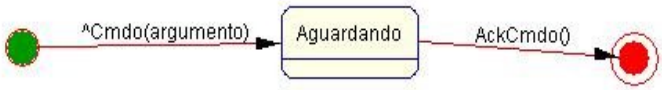


FIG 6. 59: Cmdo. Initiator

### 6.9.6 CMDO.RESPONDER

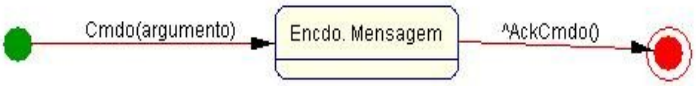


FIG 6. 60: Cmdo. Responder

### 6.9.7 INFORM. IPINITIATOR

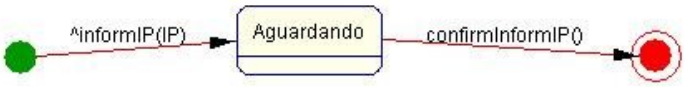


FIG 6. 61: Inform. IP Initiator

### 6.9.8 INFORM. IPRESPONDER


FIG 6. 62: Inform. IPResponder

### 6.9.9 INFORM. MONIT.INITIATOR


FIG 6. 63: Inform. Monit. Initiator

### 6.9.10 INFORM. MONIT.RESPONDER

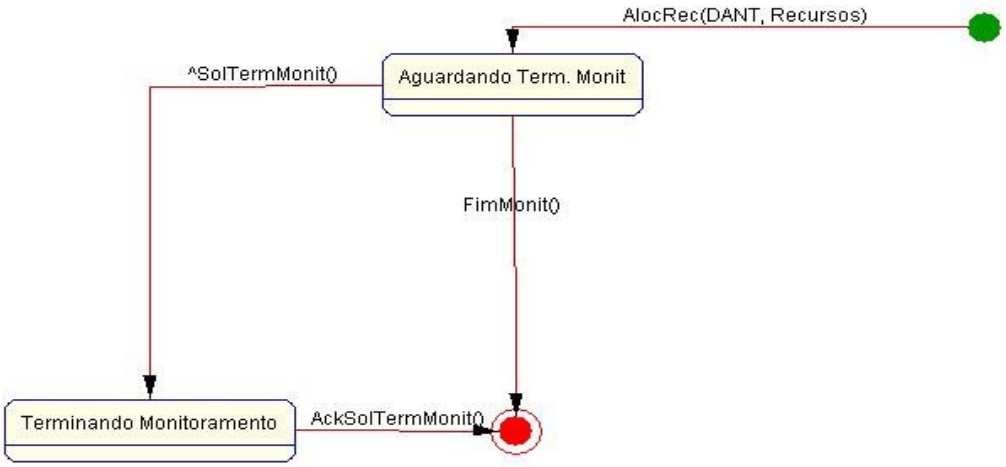


FIG 6. 64: Inform. Monit. Responder

### 6.9.11 INFORM. RECURSOSINITIATOR

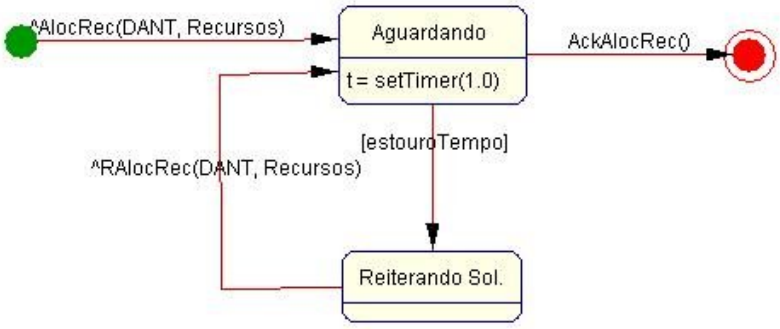


FIG 6. 65: Inform. Recursos Initiator

### 6.9.12 INFORM. RECURSOSRESPONDER

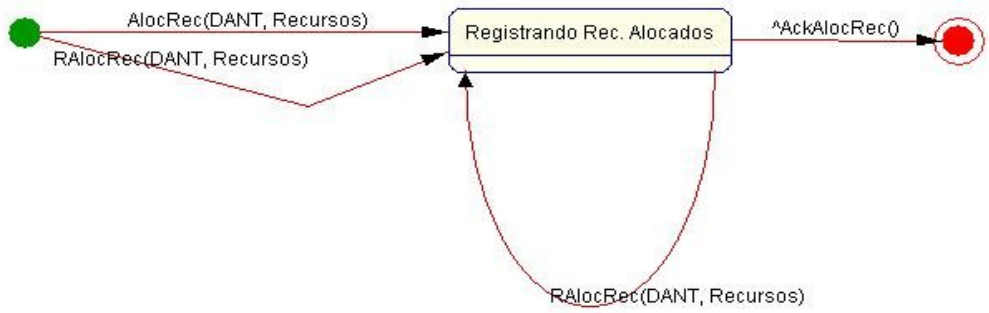


FIG 6. 66: Inform. Recursos Responder

### 6.9.13 LÊSENSORESINITIATOR

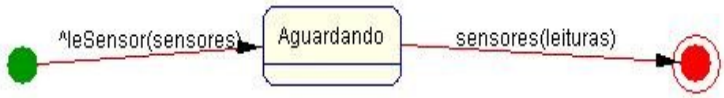


FIG 6. 67: Lê Sensores Initiator

### 6.9.14 LESENSORESRESPONDER


FIG 6. 68: Lê Sensores Responder

### 6.9.15 LOGININITIATOR


FIG 6. 69: Login Initiator

### 6.9.15 LOGINRESPONDER

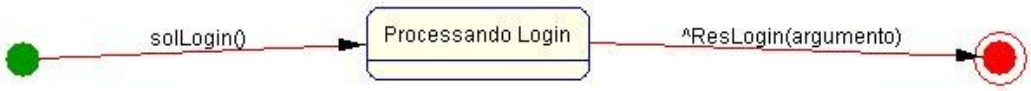


FIG 6. 70: Login Responder

### 6.9.17 LOGOUTINITIATOR

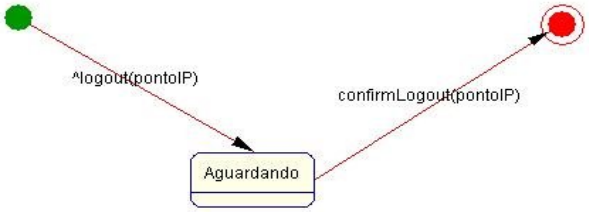


FIG 6. 71: Logout Initiator

### 6.9.18 LOGOUTRESPONDER

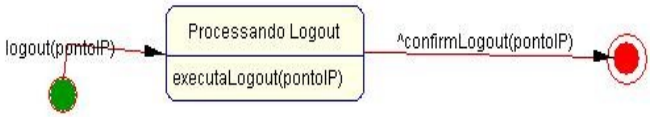


FIG 6. 72: Logout Responder

### 6.9.19 MSGS.INITIATOR

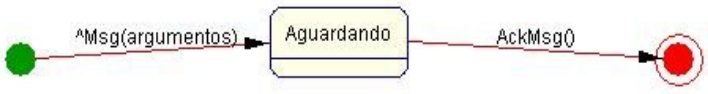


FIG 6. 73: Msgs. Initiator

### 6.9.20 MSGS.RESPONDER


FIG 6. 74: Msgs. Responder

### 6.9.21 PACOTEINITIATOR

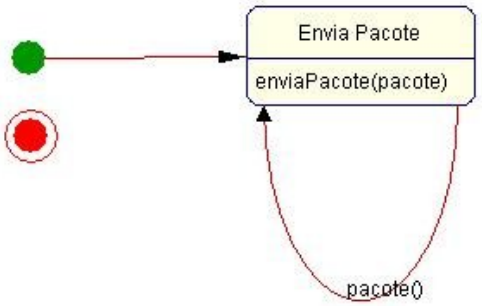


FIG 6. 75: Pacote Initiator

### 6.9.22 PACOTERESPONDER

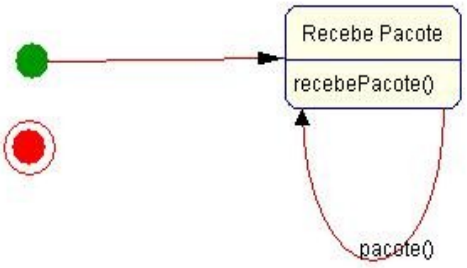


FIG 6. 76: Pacote Responder

### 6.9.23 SOL. CUSTOSINITIATOR.

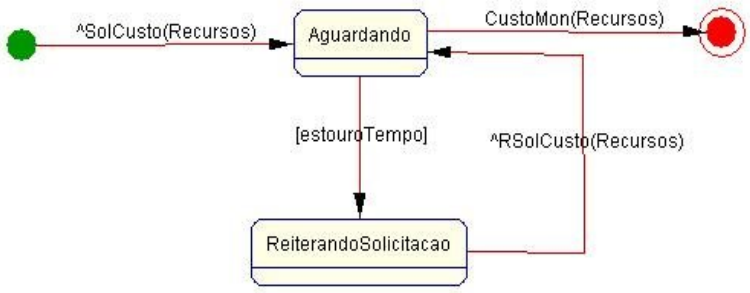


FIG 6. 77: Sol. Custos Initiator

### 6.9.24 SOL. CUSTOSRESPONDER

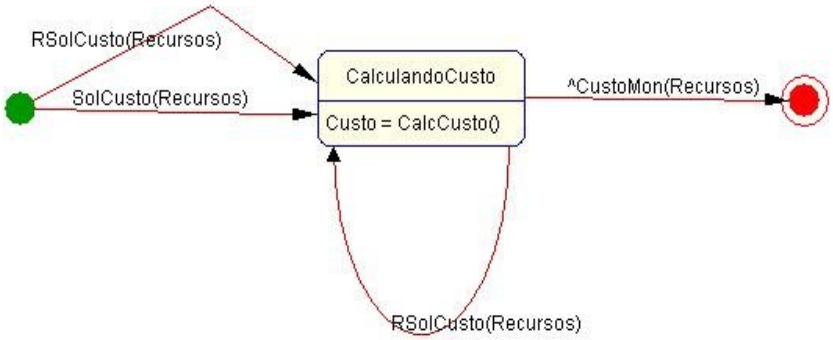


FIG 6. 78: Sol. Custos Initiator

### 6.9.25 SOL. MONITORAMENTOINITIATOR


FIG 6. 79: Sol. Monitoramento Initiator

### 6.9.26 SOL. MONITORAMENTORESPONDER


FIG 6. 80: Sol. Monitoramento Responder

### 6.9.27 SOL. MOV. SDANTINITIATOR


FIG 6. 81: Sol. Mov. SDANT Initiator

### 6.9.28 SOL. MOV. SDANTRESPONDER

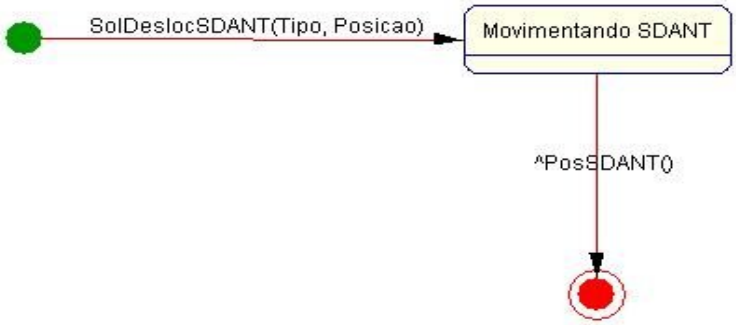


FIG 6. 82: Sol. Mov. SDANT Responder

### 6.9.29 SOL. POS. DANTINITIATOR

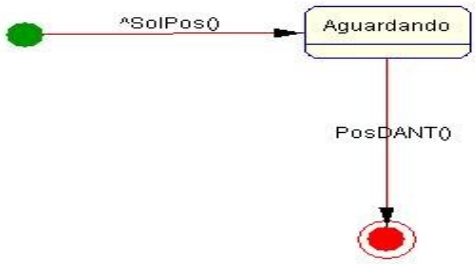


FIG 6. 83: Sol. Pos. DANT Initiator

### 6.9.30 SOL. POS. DANTRESPONDER

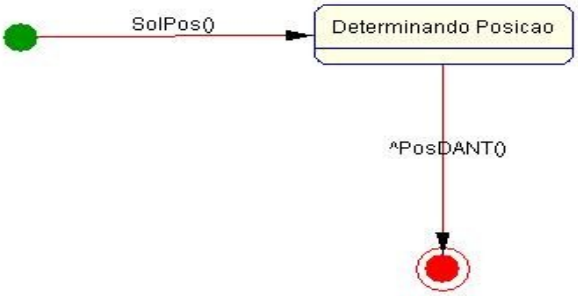


FIG 6. 84: Sol. Pos. DANT Responder

### 6.9.31 SOL. RECURSOS MSG.INITIATOR

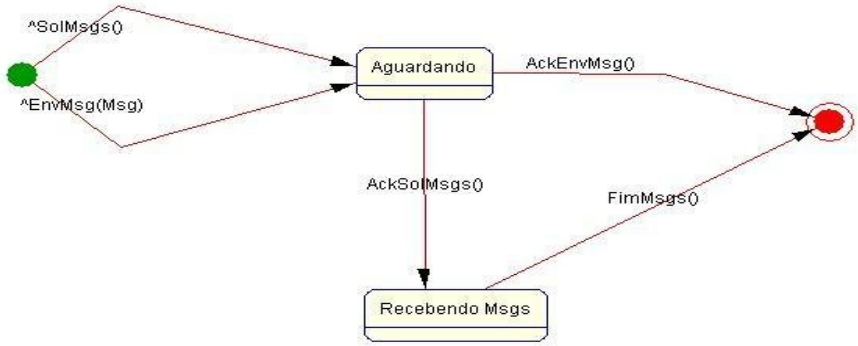


FIG 6. 85: Sol. Recursos Msg. Initiator

### 6.9.32 SOL. RECURSOS MSG.RESPONDER

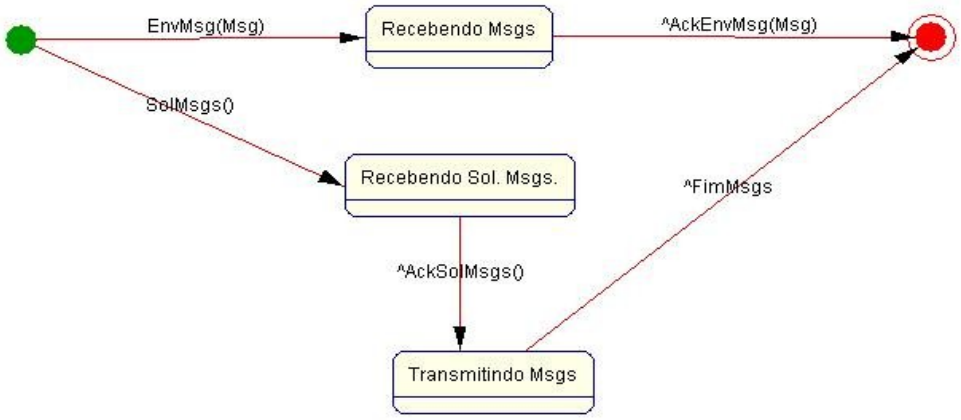


FIG 6. 86: Sol. Recursos Msg. Responder

### 6.9.33 SOL. RECURSOSINITIATOR

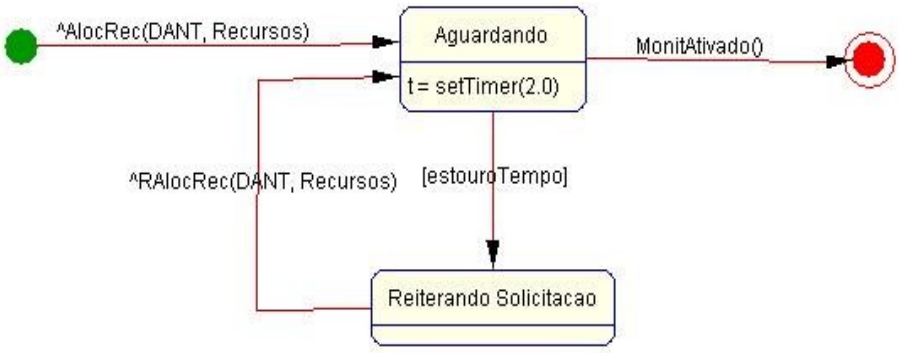


FIG 6. 87: Sol. Recursos Initiator

### 6.9.34 SOL. RECURSOSRESPONDER

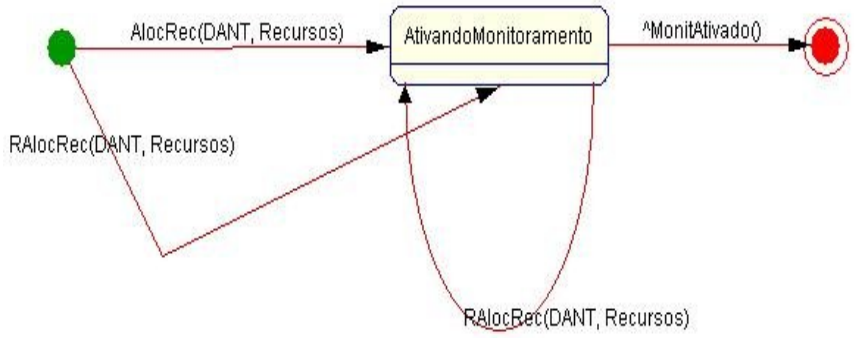


FIG 6. 88: Sol. Recursos Responder

### 6.9.35 TRAJ.DANTINITIATOR

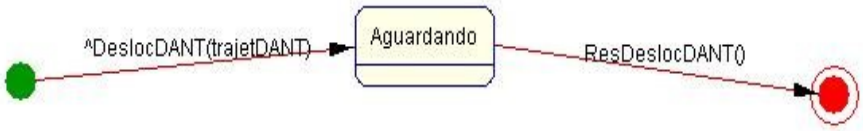


FIG 6. 89: Traj.DANT Initiator

### 6.9.36 TRAJ.DANTRESPONDER

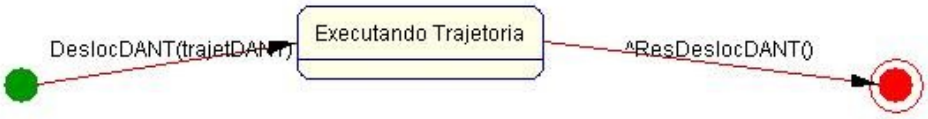


FIG 6. 90: Traj.DANT Responder

### 6.9.37 TRATA ACESSOINITIATOR


FIG 6. 91: Trata Acesso Initiator

### 6.9.38 TRATA ACESSORESPONDER

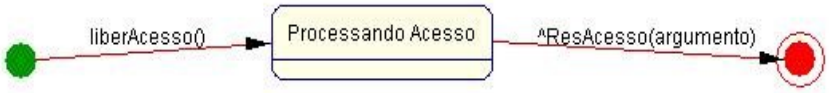


FIG 6. 92: Trata Acesso Responder

### 6.9.39 TRATA MONIT.INICIATOR

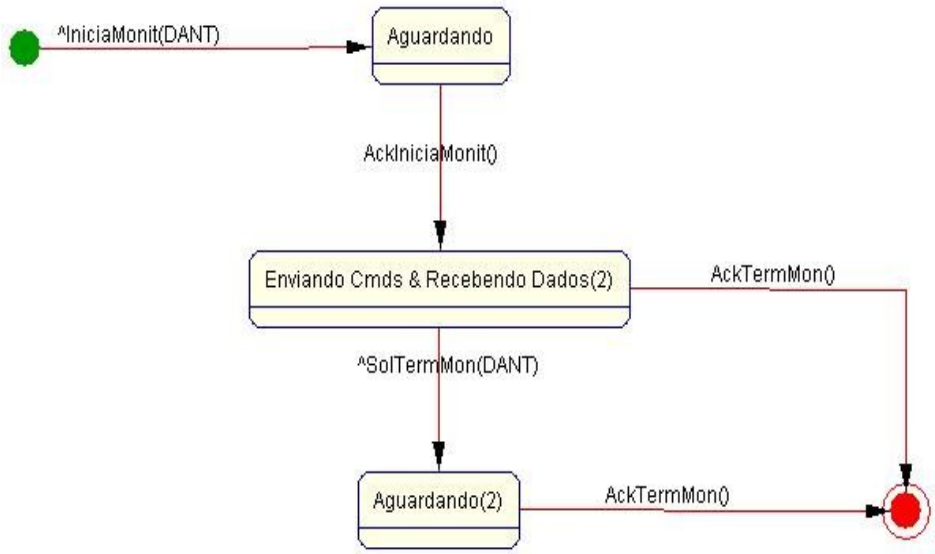


FIG 6. 93: Trata Monit. Iniciator

### 6.9.40 TRATA MONIT.RESPONDER

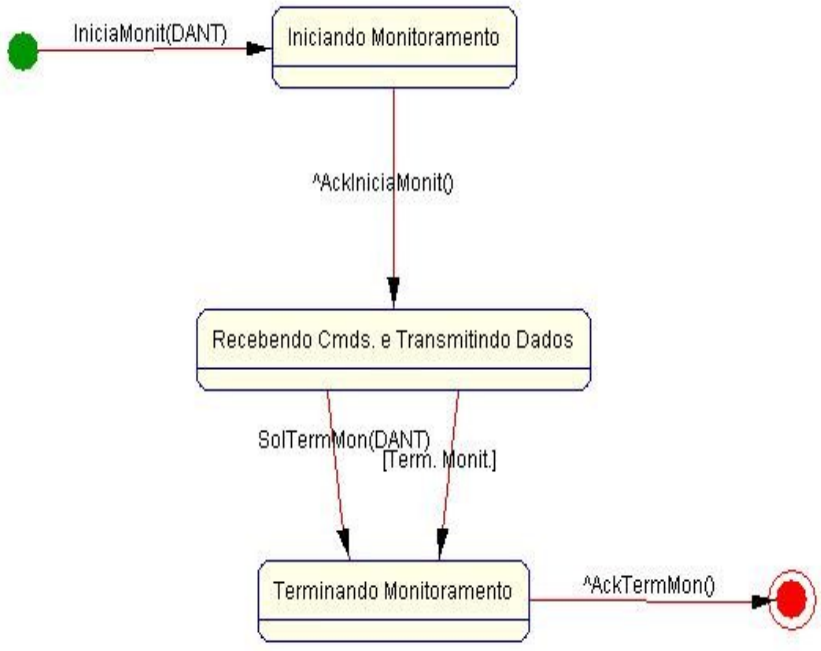


FIG 6. 94: Trata Monit. Responder

## 6.10 IMPLANTAÇÃO

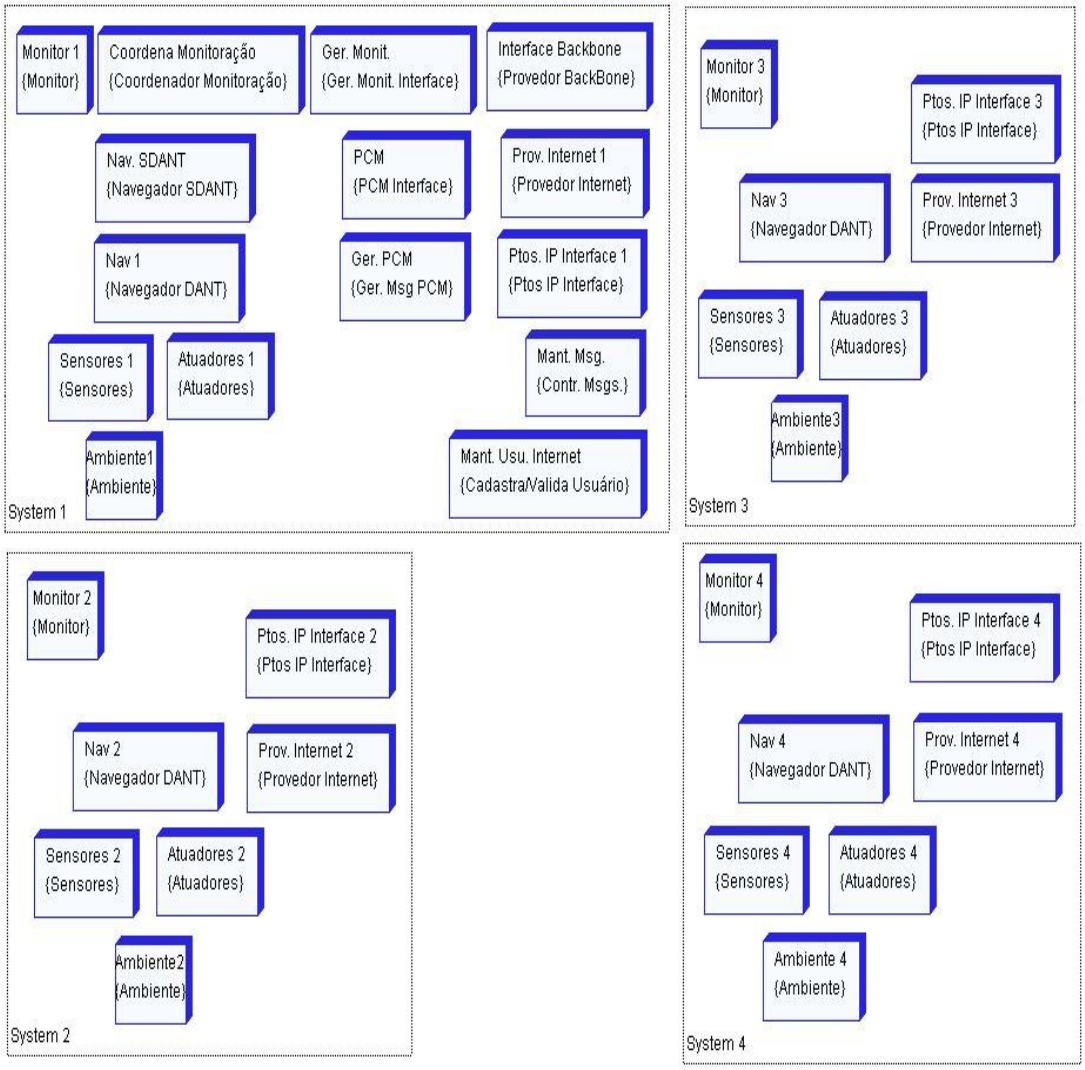


FIG 6. 95: Implantação

## 6.11 UTILIDADE DO MODELO DO SIMULADOR

Terminadas as fases de análise e projeto, segue-se a fase de construção do sistema – a fase de construção não é coberta pela MaSE. Na fase de construção, desdobra-se o "projeto" do sistema – o "projeto" do sistema é chamado aqui de modelo do sistema – em módulos executáveis e documentação operacional. No caso presente, as diversas especificações e modelos (parciais) do simulador que foram desenvolvidos ao longo da modelagem MaSE consistem, tomados em conjunto, no modelo do simulador. Idealmente, tal modelo deveria ser o necessário e suficiente para orientar a construção do simulador. A partir dele deveria ser

possível, por exemplo, desenvolver o correspondente *software* do sistema – pois o simulador é composto de agentes de *software*.

Para ser útil, o modelo do simulador precisa ser minimamente eficaz e eficiente. Neste trabalho, o modelo é considerado tanto mais eficaz, quanto menos esforços suplementares de projeto forem necessários para o sistema ser construído; e tanto mais eficiente, quanto menos esforços de construção o modelo requerer. Se, por exemplo, depois do modelo pronto, for necessário projetar um protocolo especial para alguma conversação, o modelo não será considerado totalmente eficaz. Por outro lado, se o modelo definir arquitetura desnecessariamente complexa, requererá mais esforço de construção do que o necessário, e, neste caso, será considerado ineficiente. O modelo seria inútil se em nada ajudasse na construção do sistema.

A construção de SMA acontece em alguma plataforma de construção, e os SMAs em si mesmos são objetos de padronização. De tais circunstâncias decorrem relações entre as fases de projeto e construção do sistema que afetam a eficácia e a eficiência do modelo. Por exemplo, se na fase de projeto forem adotadas soluções previstas pelos padrões escolhidos para reger a construção do SMA, o modelo produzido tenderá a ser mais eficiente, pois, durante a construção do sistema, os esforços de mapeamento da solução do modelo em soluções padronizadas não ocorrerão, ou serão mínimos. Por outro lado, se na fase de projeto forem consideradas características comuns das plataformas de desenvolvimento em geral (e.g., linguagem orientada a objetos, arquitetura geral de agente, modelagem ontológica, etc.) o modelo produzido será mais eficaz.

Os construtores da MaSE pretendem (ou pretendiam) estendê-la para que gerasse código automaticamente e, conseqüentemente, gerasse os módulos executáveis do sistema, tudo a partir do modelo produzido nas fases de análise e projeto (DELOACH, 2001b). Neste caso, a plataforma de análise & desenvolvimento e a plataforma de construção poderiam interagir otimamente – pois uma seria feita para atuar exclusivamente em favor da outra. Em oposição a esta situação ideal há, decerto, alguma plataforma de construção com máxima incompatibilidade com a MaSE. Então, para que se possa avaliar a efetividade e a eficiência do modelo produzido pelo projeto MaSE, e, mormente, a utilidade do modelo, é necessário que se verifique a interação do modelo com a plataforma de construção a ser efetivamente usada.

Alguns critérios orientam as decisões sobre quais são a plataforma e ferramentas de construção adequadas ao simulador – a plataforma e as ferramentas a serem efetivamente

usadas. Por exemplo, considerando-se que há padrões bem estabelecidos para agentes e SMAs, um critério seria o grau de compatibilidade da plataforma e das ferramentas de construção com os padrões escolhidos: quanto mais compatíveis, melhor. Outros critérios são vistos adiante.

A escolha dos padrões condiciona as escolhas das ferramentas e plataformas. A aderência de todas as ferramentas, e da própria plataforma, a um conjunto coerente de padrões favorece a eficiência das interações dos componentes da plataforma entre eles mesmos e entre eles e a plataforma: todos resultam do mesmo sistema conceitual subjacente ao conjunto de padrões (e este sistema de conceitos é, ainda, reforçado pelos próprios padrões). Assim, a escolha de um específico conjunto de padrões restringe o elenco inicial de plataformas e ferramentas passíveis de uso àquelas que aderem ao conjunto de padrões escolhido. Uma vez escolhidos os padrões, pode-se escolher a plataforma de construção – e, ainda, a de operação.

Tendo-se os padrões e as plataformas, pode-se avaliar, mesmo que parcialmente, o modelo. Para tal, sobre a plataforma e com as ferramentas escolhidas, tenta-se construir trechos do modelo, usando-se as informações nele contidas. Esta tentativa tem o bônus de permitir o aprendizado do uso dos recursos de construção, incluído dentre eles o próprio modelo. A análise dos resultados obtidos e das dificuldades encontradas durante a tentativa permite a avaliação da exeqüibilidade do modelo obtido, da MaSE, e das ferramentas e plataforma escolhidas.

Como preparação para a avaliação do modelo, padrões, ferramentas e plataformas são, nos itens seguintes, levantados, analisados, escolhidos e aplicados no desenvolvimento de pequenos trechos do modelo.

# 7 PADRÕES PARA A CONSTRUÇÃO E OPERAÇÃO DO SIMULADOR

Há diversas escolhas que conduzem a diferentes caminhos de construção e operação do simulador. Elas se dão segundo três dimensões: padrões de inter-operação de agentes, ambientes e ferramentas de construção de agente e ambientes e ferramentas de operação de agente (o ambiente de operação de agente serve também à sua construção durante os testes). Os valores escolhidos para as dimensões afetam os níveis de eficiência e eficácia do processo de construção do simulador; os níveis de eficiência, eficácia e flexibilidade da operação do simulador; e os níveis de robustez e de atualização tecnológica do sistema. Primeiramente, vai-se buscar escolher os padrões.

Padrões bem desenvolvidos, amplamente aceitos e coerentes ao longo do tempo funcionam como infra-estrutura tecnológica sobre a qual podem se assentar os provavelmente muitos projetos que a eles aderem. A simples existência de um padrão confiável e amplamente aceito pode estimular o surgimento de novos projetos na área que ele regula. Um exemplo é o padrão arquitetônico dos computadores pessoais: o número de fabricantes, modelos, tipos de dispositivos auxiliares, unidades produzidas cresceu acentuadamente após a IBM tê-lo tornado público (http://en.wikipedia.org/wiki/Home_computer).

Sob um mesmo conjunto de padrões coerentes, os resultados de diferentes projetos e os caminhos para consegui-los podem ser mais facilmente comparados, pois ocorrem sob um mesmo paradigma. Tais trocas de informação melhoram a qualidade dos métodos e processos disciplinados pelos padrões e, muitas vezes, melhoram a qualidade dos próprios padrões – principalmente quando a tecnologia é incipiente. A importância dos padrões é verdadeiramente excepcional, contudo, quando normalizam conectividade – os agentes do simulador são comunicativos. Exemplo notável dos benefícios que tal tipo de padrão pode produzir é a Internet.

A aderência incondicional a padrões, contudo, pode trazer, também, inconveniências: por exemplo, a submissão completa a padrões complexos pode levar a custos suplementares elevados em termos de tempo para conclusão do projeto e de aderência a normalizações desnecessárias.

Na área de SMA, há esforços para a padronização da inter-operação de agentes, abrangendo intercomunicação, coordenação, plataforma de operação, etc. A vantagem específica da adesão no caso presente é dotar o simulador de tecnologias robustas, eficientes e

razoavelmente documentadas, por serem largamente usadas, tanto de construção, quanto de operação de SMA.

Dentre os padrões de SMA, os mais importantes parecem ser os da FIPA (*Foundation for Intelligent Physical Agents*[39]), os do OMG (*Object Management Group*[40]) e os da DARPA (*Defense Advanced Research Projects Agency*).

## 7.1 PADRÃO OMG

O padrão do OMG chama-se *Mobile Agent System Interoperability Facility* (MASIF). Seu objetivo principal era (ou é ainda) o de integrar a tecnologia de agente móvel com o *middleware*[41] CORBA (*Common Object Request Broker Architecture*), integrando, assim, a tecnologia de agente com a tecnologia cliente-servidor convencional. A MASIF estabelece um padrão de arquitetura que contém interfaces com métodos (FIG. 7.1), e, portanto, a padronização também se dá por intermédio do uso destes métodos.

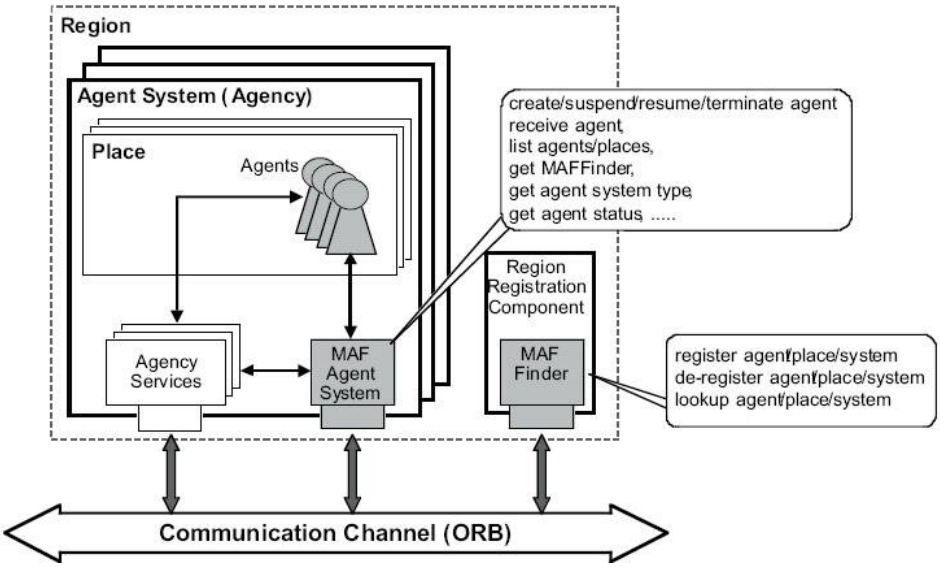


FIG 7. 1: Arquitetura de Plataforma de Agente MASIF (MAGEDANZ, 1998).

A MASIF padroniza o seguinte (MAGEDANZ, 1998):

[39] www.fipa.org
[40] www.omg.org
[41] *Middle-ware*, *software* que conecta duas aplicações separadas.

- gerência de agente – criação, suspensão, retomada e término de agentes, por meio dos métodos *create_agent*, *terminate_agent*, *suspend_agent* e *resume_agent* da interface *MAFAgentSystem* (FIG. 7.1).
- Transporte de agente – (para implementações similares de sistemas de agentes) mecanismos de recepção de agentes e de busca de suas classes. A interface *MAFAgentSystem* oferece dois métodos para suportar a migração de agentes, os métodos *receive_agent* para transferir o estado do agente e outros dados necessários e o método *fetch_class* para, adicionalmente, recuperar o código do agente, se necessário.
- Atribuição de nomes de agentes e de sistemas de agentes – sintaxe e semântica padronizadas dos nomes de agente e de sistema de agente
- Tipo de sistema de agente e sintaxe de localização.
- Rastreio de agente – rastreio de agentes registrados com serviços de atribuição de nome de sistemas de agente diferentes feito com o concurso da interface *MAFFinder*.

A primeira plataforma de agente aderente à MASIF é a Grasshopper (ikv++) (MAGEDANZ, 1998) – tomada aqui como ferramenta de construção e desenvolvimento de agente. Outra é a Aglets (IBM Japão) (HAYZELDEN, 2001).

## 7.2 PADRÃO FIPA

A FIPA é uma organização de padrões da Sociedade de Computador IEEE[42] (em inglês, IEEE Computer Society) que promove tecnologia baseada em agente e inter-operação de seus padrões com outras tecnologias[43]. Define padrões relativos a gerência de agente (ou serviços de plataforma de agente), linguagem de comunicação de agente, integração agente-*software*, suporte a gerência de agente para mobilidade, suporte a serviço de ontologia[44] etc. (MANOLA, 1998). Atualmente, a versão de padrões FIPA mais largamente usada é a FIPA 97, Versão 2.0 (HAYZELDEN, 2001) (de fato, somente as partes 1 e 2 da FIPA 97 têm segunda versão) – não obstante o amplo uso, a FIPA considera o conjunto FIPA 97 obsoleto.

[42] O IEEE (Institute of Electrical and Electronics Engineers, Inc.) é uma das mais prestigiosas instituições mundiais na área de tecnologia.
[43] www.fipa.org, página inicial.
[44] Ontologia: especificações formais explícitas dos termos de um domínio de conhecimento e das relações entre estes termos, Tom Gruber, 1993.

As partes centrais do conjunto (a FIPA 97 tem sete partes) são a Parte 1, que define o modelo de referência de agente; e a Parte 2, que especifica a linguagem de comunicação de agente (HAYZELDEN, 2001).

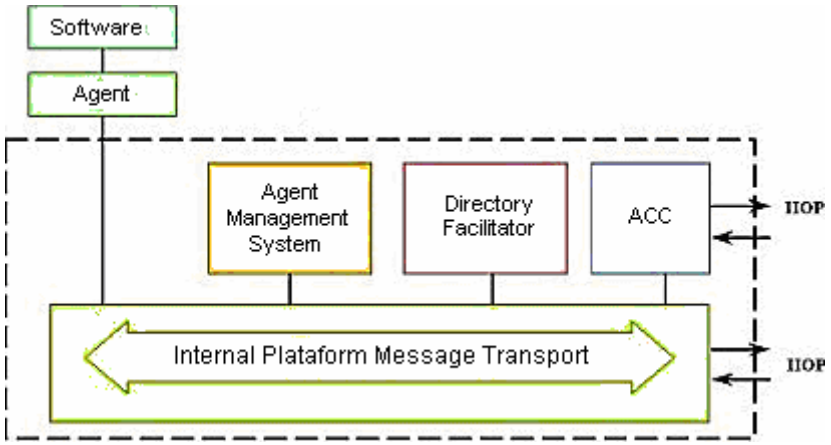


FIG 7. 2: Modelo FIPA 97 de referência de gerência de agente (FIPA 97, 1998).

O modelo FIPA 97 de referência de gerência de agente (FIG. 7.2)[45] – apresentado na Parte 1 – estabelece o arcabouço normativo dentro do qual agentes FIPA existem e operam. Combinado com o Ciclo-de-Vida-de-Agente, estabelece os contextos lógico e temporal para criação, operação e afastamento de agentes (FIPA 97, 1998). Consiste em Directory Facilitator (DF), Agent Management System (AMS) e Agent Communication Channel (ACC). O DF é um agente que provê serviços de "páginas amarelas" a outros agentes. O AMS é um agente que supervisiona o acesso e uso do ACC. O AMS é responsável pela inclusão, criação, exclusão de agentes etc. O ACC fornece o caminho para a intercomunicação básica entre um agente e outros agentes, incluindo o DF e o AMS. O ACC estabelece as rotas das mensagens trocadas entre agentes. O ACC, AMS, IPMT e DF formam o que é chamado de Plataforma de Agente (AP). O IPMT incumbe-se do transporte das mensagens trocadas dentro da AP.

O conjunto FIPA 97 (Parte 2) estabelece especificações normativas para a linguagem de comunicação de agente. A especificação da linguagem baseia-se em atos de discurso (*speech*

[45] IIOP, abreviação de **I**nternet **I**nter-**O**RB **P**rotocol, um protocolo desenvolvido pelo OMG para implementar soluções CORBA para a Web. O IIOP habilita *browsers* e servidores a trocar inteiros, *arrays* e objetos complexos, diferentemente do HTTP, que só suporta transmissão de texto. Assim, a FIPA, a exemplo do OMG, tem feito padrões que promovem a inter-operação com sistemas de tecnologia cliente-servidor.

*acts*)[46] e estabelece uma semântica formal para a linguagem; a especificação inclui também diversos protocolos – e.g., rede de contrato (*contract-net*) e leilão.

As principais questões tratadas pelos padrões FIPA são, em resumo, as seguintes (CHARLTON, 2000):

I. *Plataforma de Agente*, a infra-estrutura na qual os agentes executam suas operações;
II. *Linguagem de Comunicação de Agente* (ACL), que é a linguagem que os agentes têm de usar para codificar as mensagens que eles trocam. Na ACL, as mensagens são ações, ou atos comunicativos, pois elas intentam a execução de alguma ação em virtude de terem sido enviadas;
III. *Linguagem de Conteúdo*, que é a linguagem usada para a codificação do *conteúdo* da mensagem; o conteúdo é a parte da mensagem que representa o componente da comunicação dependente do domínio da aplicação;
IV. *Protocolos*, que são padrões de trocas de mensagens; protocolos são definidos por meio de atos comunicativos e especificam como a comunicação deve se dar entre agentes.

A principal crítica ao padrão FIPA na literatura é a de que suas especificações abrangem o interior dos agentes – o que interferiria desnecessariamente no projeto de agente. A crítica se deve ao uso da semântica de crença, desejo e intenção – BDI – para definir a linguagem de comunicação de agente. Esta semântica é baseada em um agente mental e não em agente social, que é mais apropriado para definir comportamento cooperativo de agente (CHARLTON, 2000).

## 7.3 PADRÃO DARPA

A DARPA identificou tipos específicos de padrões de agente que ela acredita serem necessários. Os principais são os seguintes (MANOLA, 1998):

- Modelo de referência de agente
    - gerência e controle de agente
    - privacidade, segurança, controle de acesso e veracidade.
- Comunicação agente-agente

[46] J. R. Searle. *Speech Acts.* Cambridge University Press, Cambridge, 1969.

    - linguagens de comunicação
        - compartilhamento de conhecimento
        - descrições de metas e tarefas
    - protocolos de coordenação
    - protocolos de segurança
- Comunicação agente-*software*
    - comunicação (APIs) entre agentes e *software* não-agente

Estas categorias de padrões parecem ser aproximadamente as mesmas identificadas pela FIPA. Os padrões relativos a segurança e a metas e descrições de tarefas parecem os menos desenvolvidos (MANOLA, 1998).

## 7.4 RESUMO COMPARATIVO

Dos três padrões, FIPA, OMG e DARPA, o que parece se adequar melhor às necessidades do simulador é o FIPA. O padrão FIPA lida com comunicação agente-agente, enquanto o padrão OMG não lida (MANOLA, 1998), ou não lidava. O padrão DARPA não é tão difundido quanto o padrão FIPA. A FIPA é uma instituição confiável, o que sua associação ao IEEE confirma. Há diversas ferramentas de construção e operação de agentes que adotam o padrão FIPA, como, por exemplo, APRIL (Fujitsu, USA), Comtec Agent Plataform (Japão), FIPA-OS (Nortel Networks), JADE (CSELT) e a ZEUS (BT UK), todas em código aberto (*open source*)[47]. Por estas razões, o padrão FIPA deve se ajustar às necessidades de construção e operação do simulador.

[47] O*pen source* é o atributo de um programa que indica, genericamente, que seu código fonte está disponível para o público em geral para uso e/ou modificação, sem custos.

# 8 FERRAMENTAS E AMBIENTES DE CONSTRUÇÃO E OPERAÇÃO

Uma das ferramentas que compõe os ambientes de construção e operação de agente – e definidora destes ambientes – que se considerou apropriada para a construção do Simulador é a JADE – Java Agent DEvelopment Framework. Possui a maior base de usuários nas áreas acadêmica e comercial, tem sido atualizada com constância para se conformar aos padrões FIPA que surgem e é considerada mais fácil de aprender e de usar, em comparação com o FIPA-OS – FIPA-OS seria outra ferramenta, dentre as mencionadas, passível de uso.

Do ponto de vista de tratamento de agente, a JADE é a ferramenta âncora em torno da qual gravitam as demais que completam os ambientes mencionados. No ambiente de operação, funciona em máquina Java, portanto, a máquina virtual Java é outra ferramenta constituinte do ambiente de operação – talvez seja mesmo, neste caso, "o" ambiente. Para o desenvolvimento, usa-se, também, a plataforma Eclipse cuja configuração padrão (ou inicial) contém um conjunto integrado de ferramentas voltado para a construção de programas em Java – o que é necessário, pois os agentes JADE são agentes Java. As ferramentas para elaboração de ontologias são a Protégé e o beangenerator – este, um *plug-in*[48] do Protégé que permite a geração de ontologias em formato aceito pelo JADE a partir de ontologias Protégé.

## 8.1 AMBIENTE DE CONSTRUÇÃO[49] (OU DE DESENVOLVIMENTO) JADE

A principal ferramenta do ambiente de desenvolvimento e de operação, sob a perspectiva de tratamento de agentes, é a JADE, uma estrutura de software para a construção, implementação e operação – regidas pelas especificações FIPA relativas a sistemas multiagentes, inteligentes e inter-operáveis – de aplicações baseadas em agente. Serve para simplificar a construção, implementação e operação de SMAs, assegurando simultaneamente aderência do SMA aos padrões FIPA por intermédio de um conjunto abrangente de agentes e serviços de sistema. A JADE pode ser considerada um *middle-ware* de agente que implementa uma Plataforma de Agente (local onde os agentes operam) e uma estrutura de

[48] *Plug-in,* um módulo de *hardware* ou de *software* que acrescenta uma característica ou serviço específico a um sistema maior. A idéia é que o novo componente seja simplesmente conectado (*plugs in*) ao sistema existente.
[49] Vai-se chamar também de ambiente de desenvolvimento e ambiente de programação.

desenvolvimento (que sustenta a construção e implementação de agentes). Ela lida com todos os aspectos que não são interiores aos agentes[50] e que são independentes das aplicações[51] .

Na prática, os componentes JADE de apoio específico ao desenvolvimento consistem, dentre outros, em pacotes (bibliotecas de classe Java) JADE. A plataforma Eclipse permite que estes pacotes sejam integrados à plataforma e as respectivas classes tratadas como classes Java nativas. Um outro componente JADE (um macro-componente) de apoio ao desenvolvimento é o ambiente de operação JADE. O ambiente de operação JADE é necessário para testar e depurar programas de agente. Este componente JADE será tratado em um item específico mais adiante.

Os demais componentes do ambiente de desenvolvimento, como antes mencionado, são o Eclipse, o Protégé e o beangenerator.

### 8.1.1 OS COMPONENTES JADE DE DESENVOLVIMENTO

Os componentes JADE de desenvolvimento são os pacotes JADE (bibliotecas de classe Java) e a documentação que explica como usar os pacotes e as classes. A documentação consiste principalmente nos comentários do código fonte das classes (acessíveis na IDE Eclipse) e em documentos (em .pdf), como, por exemplo, os seguintes: JADE Administrator's Guide, JADE Programmer's Guide, JADE Tutorial – JADE Programming for Beginners. A documentação e os pacotes mencionados, além de outros documentos e códigos, podem ser obtidos em http://jade.tilab.com/.

#### 8.1.1.1 PACOTES JADE

Os principais pacotes e sub-pacotes JADE são `jade.core`, `jade.core.behaviours`, `jade.lang.acl`, `jade.content`, `jade.content.lang.sl`, `jade.domain`, `jade.gui`, `jade.mtp`, `jade.proto`, `jade.domain.JADEAgentManagement`, `jade.wrapper` e o pacote `FIPA`.

O `jade.core`, que implementa o núcleo do sistema. Inclui a classe `Agent`, que tem de ser estendida pelos programadores de aplicação; além disto, a hierarquia de classe `Behaviour` está contida no sub-pacote `jade.core.behaviours`. Comportamentos consubstanciam as

[50] Com a exceção já mencionada.
[51] Em http://jade.tilab.com/community-faq.htm#q1.

tarefas, ou intenções, de um agente. Os comportamentos são unidades lógicas de atividade que podem ser compostas de variados modos para realizar estruturas complexas de execução e que podem ser executadas concorrentemente. Os programadores de aplicação definem as operações de agentes escrevendo comportamentos e interconectando-os por meio de rotas de execução.

O sub-pacote `jade.lang.acl` é fornecido para processar a Linguagem de Comunicação de Agente (ACL) de acordo com as especificações do padrão FIPA.

O pacote `jade.content` contém um conjunto de classes para suportar ontologias e linguagens de conteúdo definidas pelo usuário. Em particular, a `jade.content.lang.sl` contém o codec[52] SL, o analisador e o codificador (*encoder*). SL é uma linguagem de conteúdo denominada Linguagem Semântica (SL) FIPA. A SL é sugerida pela FIPA como candidata a linguagem de conteúdo para ser usada em conjunção com a ACL. Como se vê, a JADE dá suporte a uma linguagem de conteúdo *default*.

O pacote `jade.domain` contém todas as classes Java que representam as entidades de Gerenciamento de Agente definidas pelo padrão FIPA, em particular os agentes MAS e DF que tratam do ciclo-de-vida, serviços de páginas brancas e amarelas. O sub-pacote `jade.domain.FIPAAgentManagement` contém a Ontologia Gerência-de-Agente-FIPA e todas as classes representando seus conceitos. O sub-pacote `jade.domain.JADEAgentManagement` contém, diferentemente, as extensões JADE para Gerência-de-Agente (e.g. para mensagens de farejo – *sniffing messages* –, controle do ciclo-de-vida de agentes, etc.), incluindo a Ontologia e todas as classes representando seus conceitos. O sub-pacote `jade.domain.introspection` contém os conceitos usados no domínio do discurso travado entre as ferramentas JADE (e.g. o Sniffer e o Introspector) e o núcleo JADE. O sub-pacote `jade.domain.mobility` contém todos os conceitos usados para comunicar mobilidade.

O pacote `jade.gui` contém um conjunto de classes genéricas úteis à criação de GUIs para mostrar e editar Identificadores-de-Agentes, Descrições de Agentes, ACLMessages, etc.

O pacote `jade.mtp` contém a interface Java, interface que cada Protocolo de Transporte de Mensagem deveria implementar para ser facilmente integrado à estrutura de trabalho JADE, e a implementação de um conjunto destes protocolos.

[52] Codec é abreviação de *coder/decoder* expressão que diz respeito à função de conversão do formato da mensagem de conteúdo para transmissão ou para uso internamente pelo agente.

jade.proto é o pacote que contém classes para modelar protocolos de interação padrão (i.e. *fipa-request*, *fipa-query, fipa-contract-net, fipa-subscribe* e logo outros definidos pela FIPA), bem como classes para ajudar programadores de aplicação a criar seus próprios protocolos.

O pacote FIPA contém o módulo IDL (Interface Definition Language[53]) definido pela FIPA para transporte de mensagem baseado-em-IIOP.

Finalmente, o pacote `jade.wrapper` provê envoltórios das funcionalidades de alto-nível JADE que permitem o uso da JADE como uma biblioteca, onde aplicações Java externas lançam agentes JADE e continentes de agentes.

#### 8.1.1.2 O USO DAS CLASSES JADE

Muitos dos procedimentos relativos ao tratamento de agente são codificados nas classes JADE e estes códigos não precisam ser conhecidos pelo programador de agente – embora ele possa vê-los, se quiser, pois o JADE é *open source*. Contudo, algumas classes e métodos precisam ser modificados pelo programador – além dele ter de programar as classes específicas da sua aplicação. As primeiras classes e métodos JADE com os quais o programador tem de lidar são discutidas a seguir.

A classe `Agent` é a principal classe JADE. Para criar um agente JADE, o programador tem de estender a classe `jade.core.Agent` (ou seja, a classe `Agent`) e, em geral, modificar os seguintes métodos desta classe: `setup()` e `takeDown()`. No método `setup()`, deve ser disparado um objeto da classe `Behaviour`. Em geral, o programador de agente tem, também, de modificar dois métodos da classe `Behaviour`: o método `action()` e o método `done()`. No exemplo a seguir, um agente JADE é criado. Observe-se que, como este agente tem funcionalidade artificialmente mínima, somente um dos métodos, o método `setup()`, é modificado.

```
import jade.core.Agent;
public class PrimeiroAgente extends Agent
{
   protected void setup()
   {
   // O agente imprime uma mensagem
      System.out.println("Oi! Agente "+getAID().getName()+" está pronto.");
    doDelete();
   }
}
```

[53] Linguagem usada para descrever interfaces de objetos por seus nomes, métodos, parâmetros, eventos tipos de retorno. Um compilador usa a informação IDL para gerar código para transportar dados entre máquinas.

No exemplo anterior, o método `getAID()` – uma função – é invocado. Ele funciona da seguinte maneira: cada agente é identificado por um "identificador de agente" representado como uma instância da classe `jade.core.AID`. O método `getAID()` da classe `Agent` permite a recuperação do identificador do agente. Um objeto `AID` inclui um nome globalmente único mais um número de endereços. O nome em JADE tem a forma *<nickname>@<plataform-name>* tal que um agente chamado *Pedro* vivendo na plataforma *P1* terá *Pedro@P1* como nome globalmente único. Os endereços incluídos no `AID` são os endereços da plataforma onde o agente vive. Estes endereços só são usados quando um agente precisa se comunicar com um outro agente vivendo em uma plataforma diferente da dele.

Mesmo que não haja nada a fazer após a impressão da mensagem, o agente ainda estará "rodando". Para fazê-lo terminar, seu método `doDelete()` deve ser chamado. Semelhantemente ao método `setup()` que é invocado pelo *runtime* JADE tão logo o agente se inicia e que é projetado para incluir os procedimentos iniciais do agente, o método `takeDown()` é invocado imediatamente antes de o agente terminar, e é projetado para conter as operações de limpeza do agente.

#### 8.1.1.2.1 AS MODIFICAÇÕES DOS MÉTODOS

Inicialmente, os métodos que devem, em geral, ser modificados pelo programador para construir um agente com funcionalidade realista são, na classe `Agent`, os métodos `setup()` e `takeDown()`, e, na classe `Behaviour`, os métodos `action()` e `done()`.

No método `setup()`, devem ser programados os procedimentos iniciais tais como, atribuição de valores iniciais a variáveis, comportamentos iniciais, etc. No método `takeDown()`, os procedimentos de finalização, tais como limpeza de variáveis, etc.

No método `action()`, é posto o comportamento que o agente deve ter. O método `done()` deve ter um código tal que devolva *true* – `done()` é uma função – quando o comportamento tiver sido executado até o fim; e que devolva *false*, quando a execução do comportamento não tiver terminado.

O fluxograma adiante, FIG. 8.1, é um *template* genérico que resume quais métodos o programador deve modificar e o fluxo de execução da interação destes métodos com os procedimentos internos do JADE.

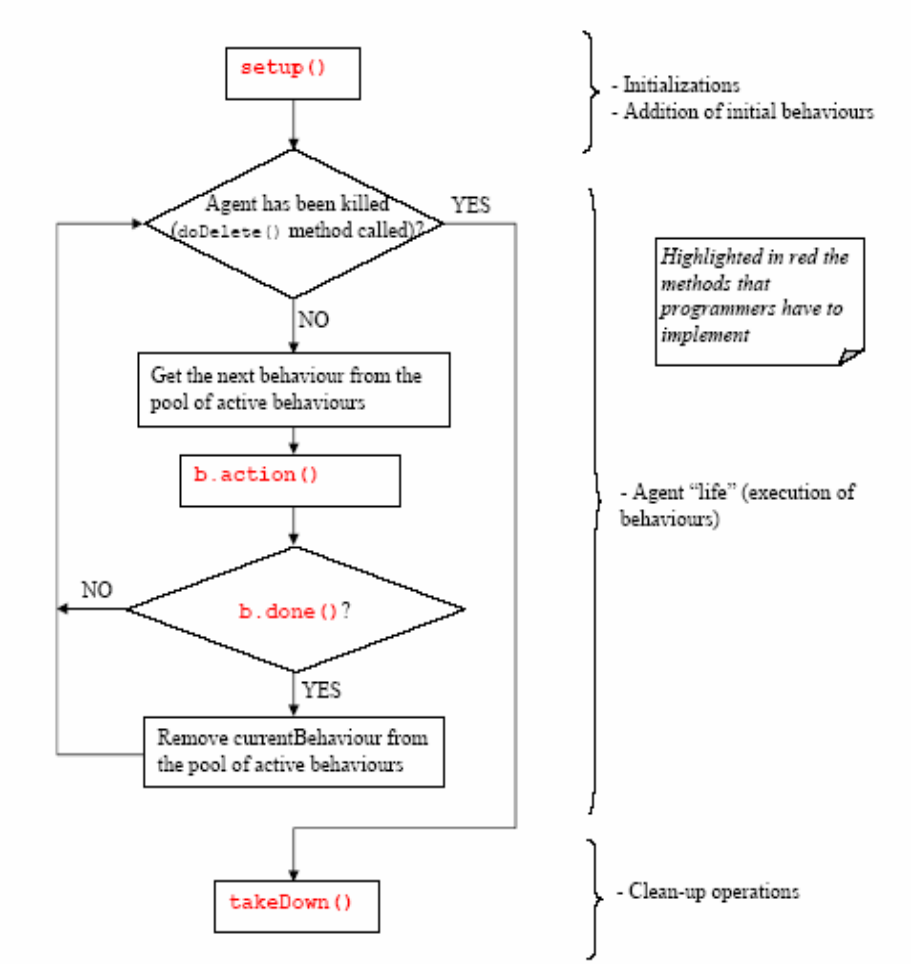


FIG 8. 1: Fluxo de execução dos métodos que devem ser modificados (CAIRE, 2003).

### 8.1.1.2.2 COMPORTAMENTOS

Conforme se viu, o programador codifica no método `action()` da classe `Behaviour` as ações que o agente deve executar. A essência da aplicação fica então codificada em uma classe `Behaviour`. Assim, qualquer simplificação no uso da classe `Behaviour` tem impacto positivo amplo nas tarefas de programação de agente. Não é de se estranhar, portanto, que a JADE preveja classes especializadas para facilitar a programação de tipos especiais de comportamento de agente que ocorrem com freqüência.

Para o JADE, há três tipos de comportamentos. Comportamentos que são (i) executados uma única vez, (ii) comportamentos cíclicos e (iii) comportamentos genéricos. Para (i), o JADE oferece uma classe específica chamada `OneShotBehaviour` cujo nome por extenso é (`jade.core.behaviours.OneShotBehaviour`). Para o (ii), a classe específica

`CyclicBehaviour`. Para (iii), a classe genérica `Behaviour` (os nomes por extenso destas duas últimas classes são análogos ao da primeira classe).

As classes especializadas têm o método `done()` tratado internamente por elas mesmas, e, por isto, são diferentes da classe genérica `Behaviour`. O programador de aplicação, nesses dois casos, só modifica o método `action()`. `OneShotBehaviour` faz automaticamente com que o método `done()` retorne *true* tão logo a operação tenha sido executada. A classe `CyclicBehaviour`, cujo correspondente comportamento cíclico nunca é completado, tem o método `action()` executando as mesmas operações a cada vez que é chamado. Esta classe faz com que o método `done()` retorne sempre *false* automaticamente.

Além das três classes de comportamento mencionadas, quais sejam, `OneShotBehaviour`, `CyclicBehaviour` e `Behaviour`, JADE oferece duas outras classes de comportamento especializadas, `WakerBehaviour` e `TickerBehaviour`. Estas duas últimas classes tratam internamente os dois métodos "modificáveis" das classes comportamento, isto é, estas classes tratam internamente os métodos `action()` e `done()`, e, por isto, o programador de aplicação não altera nenhum destes dois métodos destas duas classes. A classe `WakerBehaviour` permite que uma operação seja invocada após o transcurso de intervalo de tempo definido pelo programador da aplicação. A classe `TickerBehaviour` permite que uma operação seja invocada periodicamente, sendo o período definido pelo programador.

A seguir, estão exemplos de instanciação de objetos das classes comportamento.

#### 8.1.1.2.2.1 EXEMPLO 1

O fragmento de código a seguir mostra como se pode programar um comportamento genérico.

```
public class ComportamentoArbitrario extends Behaviour
{
   public void action()
   {
      while (true)
      {
      // faz alguma coisa
      }
   }
   public boolean done()
   {
      return true;
   }
}
```

#### 8.1.1.2.2.2 EXEMPLO 2

O código seguinte exemplifica o uso da classe `OneShotBehaviour`. A operação X é executada uma vez.

```
public class MeuOneShotBehaviour extends OneShotBehaviour
{
   public void action()
   {
      // executa a operação X
   }
}
```

Há, ainda, uma outra classe que dispara ação uma única vez, e que será vista adiante.

#### 8.1.1.2.2.3 EXEMPLO 3

A seguir, exemplo de uso da classe `CyclicBehaviour`. A operação Y é executada repetitivamente para sempre (até o agente desempenhando o comportamento terminar).

```
public class MeuComportamentoCiclico extends CyclicBehaviour
{
   public void action()
   {
      // executa a operação Y
   }
}
```

Esta classe, a exemplo da anterior, tem, também, uma especialização que será vista ainda neste item.

#### 8.1.1.2.2.4 EXEMPLO 4

O próximo exemplo é de um comportamento genérico que deve executar três operações, nomeadamente X, Y e Z, e depois terminar. Neste exemplo, os métodos `action()` e `done()` são modificados pelo programador.

```
public class MeuComportamentoDeTresPassos extends Behaviour
{
   private int passo = 0;
   public void action()
   {
      switch (passo)
      {
         case 0:
            // executa a operação X
```

```
            passo++;
         break;
         case 1:
            // executa a operação Y
            passo++;
         break;
         case 2:
            // executa a operação Z
            passo++;
         break;
      }
   }
   public boolean done()
   {
      return step == 3;
   }
}
```

As operações X, Y e Z são executadas uma após a outra e, então, o comportamento termina.

#### 8.1.1.2.2.5 EXEMPLO 5

O próximo exemplo mostra como usar a classe `WakerBehaviour`. Nele, a operação X é executada 10 segundos (10000 ms) após a instanciação do objeto da classe. Conforme foi mencionado, os métodos `action()` e `done()` já estão implementados nesta classe. As implementações são feitas de modo a que o método abstrato `handleElapsedTimeout()` seja executado após o transcurso de um dado intervalo de tempo (especificado no construtor). Após a execução do método `handleElapsedTimeout()`, o comportamento termina. No caso da classe `WakerBehaviour`, portanto, o método modificado é o `handleElapsedTimeout()`.

```
public class MeuAgente extends Agent
{
   protected void setup()
   {
      System.out.println("Adicionando comportamento despertador");
      addBehaviour(new WakerBehaviour(this, 10000)
         {
            protected void handleElapsedTimeout()
            {
               // executa a operação X
            }
         }
      );
   }
}
```

O exemplo que se acabou de ver e o próximo usam a técnica de classe interna anônima[54].

[54] H. M. Deitel, P. J. Deitel. Java, Como Programar, 4ª Edição, Capítulo 9, Bookman, 2003.

#### 8.1.1.2.2.6 EXEMPLO 6

O código a seguir mostra a técnica de instanciação da classe `TickerBehaviour`. Os métodos `action()` e `done()` são implementados de modo a que o método abstrato `onTick()` seja executado repetidamente a intervalos de tempo iguais (cujo valor é especificado no construtor). Um objeto da classe `TickerBehaviour` nunca é completado.

```
public class MeuAgente extends Agent
{
   protected void setup()
   {
      addBehaviour(new TickerBehaviour(this, 10000)
         {
            protected void onTick()
            {
               // executa a operação Y
            }
         }
      );
   }
}
```

A operação Y é executada periodicamente a cada 10 segundos.

### 8.1.1.2.3 CLASSES AUXILIARES

Além das classes `Agent` e `Behaviour` (incluindo as subclasses especializadas desta), as classes básicas, são usadas classes auxiliares, dentre as quais se pode citar as seguintes: a `AID` (cujo nome por extenso é `jade.core.AID`), a `ACLMessage` (`jade.lang.acl.ACLMessage`) e a `MessageTemplate` (`jade.lang.acl.MessageTemplate`).

A `AID` é usada para identificar agentes. Cada instância desta classe identifica um agente. Os atributos da `AID` incluem um nome globalmente único mais endereços. O nome tem a forma *<apelido>@<nome-da-plataforma>* tal que o agente *Pedro*, vivendo na plataforma chamada *P1*, terá *Pedro@P1* como nome globalmente único. Os endereços são da plataforma onde o agente vive. Eles são usados somente quando um agente precisa se comunicar com outro agente vivendo em plataforma diferente. O construtor da `AID` recebe dois argumentos, o apelido do agente e uma constante da classe `AID` – a sintaxe de invocação do método construtor é `AID` (*<apelido>*, *<constante-AID>*). A *constante-AID* pode ser ISGUID, ou ISLOCALNAME. A identificação do próprio agente, que não é criado por si próprio, e, sim,

por outro agente, pode ser obtida por meio do método `getAID()`, da classe `Agent`, e que é uma função que retorna um objeto `AID`.

Em JADE, os agentes trocam mensagens sob o padrão ACL da FIPA e cada mensagem é implementada como um objeto da classe `ACLMessage`. Esta classe possui métodos para atribuição de valores aos campos da mensagem, de acordo com o padrão ACL.

A classe `MessageTemplate` permite que sejam selecionadas da fila de entrada de mensagens do agente somente aquelas que satisfaçam a algum critério de seleção estabelecido pelo programador da aplicação.

#### 8.1.1.2.3.1 AS CLASSES `CONTENTMANAGER` E `ONTOLOGY`

A clara definição da forma de comunicação entre os agentes traz benefícios relevantes para o desenvolvimento e operação de SMAs. Quando ela ocorre, agentes de SMAs podem ser projetados e construídos por equipes diferentes e, mesmo, instituições diferentes, o que é particularmente conveniente se a aplicação é complexa. Tal distribuição de tarefas aumenta a eficiência de desenvolvimento porque reduz o tempo total de trabalho e permite a especialização dos desenvolvedores em certos tipos de agente, o que também melhora a eficiência. Portanto, para que o desenvolvimento de um agente possa se dar independentemente, permitindo a contribuição de equipes distintas para o projeto, é necessária a definição da forma de comunicação entre os agentes – além da definição da funcionalidade deles.

A comunicação é essencial aos SMAs: para que um SMA exista, é necessário que seus agentes se comuniquem. Falhas de comunicação podem produzir defeitos no sistema como um todo. Assim, do ponto de vista dos SMAs, a definição clara da forma de comunicação entre os agentes também é útil, pois definições claras tendem a reduzir falhas de comunicação por codificação e interpretação incorretas da informação.

Um dos caminhos para viabilizar as conveniências da definição clara da forma de comunicação entre os agentes é a padronização da linguagem de comunicação entre eles, isto é, a padronização do formato da linguagem, dos elementos usados por ela, significados dos elementos, etc.

Conforme se mencionou, a FIPA especifica uma linguagem de comunicação de agente, a ACL, que estabelece um formato padrão para as mensagens trocadas entre agentes. Sob este formato, a mensagem é dividida em campos, e um deles é o Conteúdo. Conteúdo é um campo

da mensagem no formato ACL que contém a real informação da mensagem (e.g., a ação a ser executada em uma mensagem REQUEST; ou o fato que o remetente deseja revelar em uma mensagem INFORM, etc.) – REQUEST e INFORM são performativas especificadas na ACL. Além de especificar a ACL, a FIPA define uma linguagem de Conteúdo, a SL, e sugere sua adoção. O JADE dá suporte à SL.

Sob a especificação da SL, o conteúdo do campo Conteúdo é uma *string*. Dentro dos agentes JADE, que são de fato agentes Java, o formato *string*, contudo, não é o mais adequado para o tratamento da informação. O formato mais adequado é o objeto Java.

Para tirar proveito da padronização da linguagem de conteúdo por meio da SL e da representação intra-agente da informação como um objeto, cada agente teria de se incumbir de converter a informação de um formato para o outro, uma operação complexa. Ademais, dotar os agentes desta funcionalidade infra-estrutural desviaria o desenvolvedor das atividades realmente específicas da aplicação. Para minorar o problema, a JADE dá suporte a linguagens de conteúdo e a ontologias. Tal suporte é construído para executar automaticamente todas as operações de conversão e verificação – verificação de se a informação é significativa[55] –, como modelado na FIG. 8.2, permitindo desta maneira que os desenvolvedores manipulem as informações dentro de seus agentes como objetos Java.

Observe-se que, independentemente das conversões e verificações antes referidas, é necessário definir os termos da informação posta em Content e atribuir-lhes significados, significados que têm de ser conhecidos pelos agentes transmissores e receptores para a mensagem ter utilidade. Provavelmente, a melhor maneira de definir os termos e os respectivos significados é usar alguma técnica já desenvolvida – em oposição ao uso de técnicas desenvolvidas *ad hoc*, ou a não usar qualquer técnica. A técnica já desenvolvida é a técnica ontológica. Portanto, mesmo que a SL, por exemplo, não seja usada, ainda assim, a definição de uma ontologia para o domínio é provavelmente necessária.

[55] Não necessariamente verdadeira. Dizer-se que "a idade de João é 45" pode ser uma afirmação falsa, porém significativa; enquanto "a idade de João é gato" é sem significado.

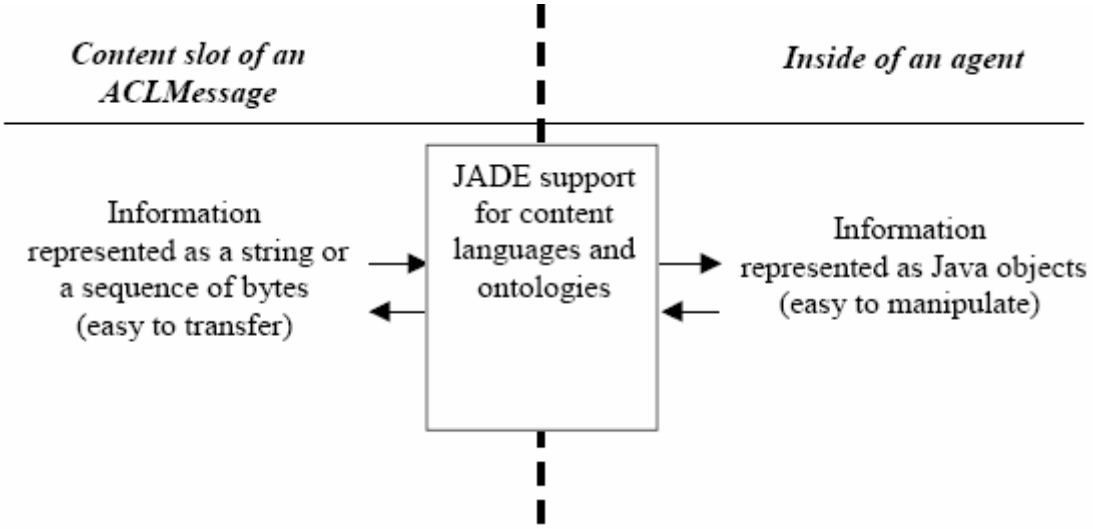


FIG 8. 2: Conversão executada pelo suporte JADE a linguagens de conteúdo e ontologias (CAIRE, 2004).

Na FIG. 8.2, a caixa "*JADE support for content languages and ontologies*" representa a função de conversão e verificação mencionadas anteriormente. As correspondentes operações são executadas automaticamente, sem necessidade de código de conversão e verificação do desenvolvedor. As operações são levadas a efeito por um objeto *content manager* (isto é, uma instância da classe `ContentManager`) – cada agente JADE tem embutido um objeto *content manager* acessível através do método `getContentManager()` da classe `Agent`. A classe `ContentManager` provê todos os métodos para transformar objetos Java em *strings* e inseri-los na ranhura `Content` da `ACLMessages` e vice-versa.

Embora o objeto *content manager* forneça interfaces convenientes para se ter acesso à funcionalidade de conversão, ele delega as operações reais de conversão e verificação a uma *ontologia* (isto é, a uma instância da classe `Ontology` incluída no pacote `jade.content.onto`) e a um *codec de linguagem de conteúdo* (isto é, a uma instância da interface `Codec` incluída no pacote `jade.content.lang`). A *ontologia* valida a informação a ser convertida do ponto de vista semântico enquanto o *codec* executa a translação para *strings* de acordo com as regras sintáticas da linguagem de conteúdo relacionada.

### 8.1.2 A PLATAFORMA ECLIPSE

Eclipse é uma plataforma código aberto de desenvolvimento, extensível, escrita em Java. Em si mesma, ela é simplesmente uma estrutura de trabalho e um conjunto de serviços para a construção de ambiente de desenvolvimento a partir de componentes *plug-in*. A plataforma

Eclipse é distribuída com um conjunto padrão de *plug-ins* com dois elencos de funcionalidades, um denominado Ferramentas de Desenvolvimento Java (JDT) e o outro chamado Ambiente de Desenvolvimento de Plug-in (PDE). A plataforma Eclipse não é restrita a uma única linguagem de programação, ou mesmo a desenvolvimento de *software*: *plug-ins* estão disponíveis, ou previstos, para suportar linguagens de programação tais como C/C++, COBOL e Eiffel. A plataforma pode ser usada também como base para outros tipos de aplicações não relacionadas a desenvolvimento de *software*, como sistemas de gerência de conteúdo[56].

JDT tem a funcionalidade de interesse aqui. Permite aos usuários escrever, compilar, testar, depurar e editar programas escritos na linguagem de programação Java, isto é, o conjunto padrão de *plug-ins* da plataforma sustenta um IDE completo Java. A plataforma, contudo, apóia-se em conceitos locais pouco usuais tais como "recurso", "perspectiva", "visão", "*workbench*", "*workspace*" e usa largamente siglas, como, por exemplo, JDT, PDE, CVS, SWT que tornam a interação com plataforma menos intuitiva quando se começa a usá-la.

Os aspectos da plataforma relacionados com a criação e adição de *plug-ins* não são de interesse neste trabalho.

### 8.1.3 DEMAIS COMPONENTES

É possível desenvolver classes de definição de ontologia (isto é, *schemas*, no jargão JADE) e as classes representando os predicados, ações de agentes e conceitos – os tipos de classes ontológicas tratados pelo JADE – digitando-se diretamente no editor Java o código correspondente. Contudo, tal técnica consome muito tempo e as operações de manutenção – alterações, inclusões, exclusões – dos *schemas* e classes são fortemente sujeitas a erros e demoradas. Há ferramentas que facilitam a geração e manutenção de *schemas* e classes ontológicas. A seguir, neste item, faz-se um breve resumo delas – apenas das que foram escolhidas – e dos conceitos JADE relacionados à ontologia (ontologia em si mesma, *schemas*, predicado etc.). As ferramentas são o Protégé e o beangenerator, um *plug-in* acoplável ao Protégé.

O Protégé é uma ferramenta integrada de *software* usada por desenvolvedores de sistema e especialistas de domínio para desenvolver sistemas baseados em conhecimento. O Protégé

[56] http://www-128.ibm.com/developerworks/library/os-ecov/

apóia as seguintes atividades: modelagem de ontologias de "classes" descrevendo um particular assunto; criação de ferramentas de aquisição de conhecimento; aquisição de instâncias específicas de dados e criação de base de conhecimento; e execução de aplicações. A ontologia define o conjunto de conceitos e suas relações. A ferramenta de aquisição de conhecimento é projetada para ser específica de um domínio, permitindo aos especialistas de domínio introduzir com naturalidade e facilmente seu conhecimento sobre a área. A base de conhecimento resultante pode então ser usada com um método de solução de problema (*problem-solving method*) para responder questões e resolver problemas relativos ao domínio. Finalmente, uma aplicação é o produto final criado quando a base de conhecimento é usada na solução de um problema de usuário final, empregando métodos apropriados de solução de problema, sistema especialista, ou suporte a decisão (*help on line* do Protégé, *User'Guide – What is* Protégé-2000?). Portanto, o Protégé serve para a produção de ontologias.

Para o JADE executar as devidas verificações semânticas em uma expressão de conteúdo, é necessário classificar todos os elementos no domínio do discurso[57] de acordo com suas características semânticas genéricas. Esta classificação é derivada da ACL que requer que o conteúdo de cada mensagem ACL tenha uma semântica própria de acordo com a performativa da mensagem ACL. Os principais itens da classificação são predicado, conceito e ação de agente. Por esta razão, a árvore de "classe" definida no Protégé terá de ter no nível imediatamente após a raiz os itens *Concept, AgentAction e Predicate.* Dito de outra maneira, qualquer elemento da ontologia ou é um conceito, ou uma ação de agente, ou um predicado.

Para o JADE, a ontologia de um dado domínio é um conjunto de esquemas (*schemas*) definindo a estrutura de cada predicado, ação de agente e conceito (basicamente seus nomes e suas ranhuras) que são pertinentes ao domínio. Em outras palavras, para o JADE, uma ontologia é uma instância da classe `jade.content.onto.Ontology` à qual são adicionados os *schemas* que definem a estrutura dos predicados, ações de agentes e conceitos relevantes para o domínio em questão – os *schemas* são instâncias das classes `PredicateSchema`, `AgentActionSchema` e `ConceptSchema`, classes incluídas no pacote `jade.content.schema`. Além disto, cada *schema* é associado a uma classe Java (ou interface). Observe-se, ainda, que a saída do Protégé não se dá no formato de classe ontologia, *schemas* e classes ontológicas.

O *plug-in* beangenerator converte a ontologia Protégé em classe de definição de ontologia e nas classes de predicados, ações de agentes e conceitos. Desta forma, com o Protégé mais o

[57] Os elementos que podem ocorrer dentro de uma sentença válida enviada por um agente como Conteúdo de uma mensagem ACL.

beangenerator consegue-se gerar as classes a partir de uma ferramenta gráfica *ad hoc*, em vez de se ter de definir a ontologia escrevendo código Java.

### 8.1.4 RESUMO DO AMBIENTE DE CONSTRUÇÃO JADE

Em resumo, o ambiente de construção JADE consiste em pacotes de classe Java, documentação para uso das classes JADE, IDE Java em plataforma Eclipse, máquina Java, ferramenta de ontologia Protégé acrescida do *plug in* beangenerator e o ambiente de operação de agente JADE.

## 8.2 O AMBIENTE DE OPERAÇÃO DE AGENTE JADE

O ambiente de operação de agente JADE inclui um ambiente *runtime* (onde agentes podem "viver", e que deve estar ativo em um dado *host* antes de agentes poderem ser executados no *host*) e em um conjunto de ferramentas gráficas (que apóiam a administração e o monitoramento das execuções dos agentes).

### 8.2.1 O AMBIENTE *RUNTIME*

O ambiente *runtime* JADE é uma camada de *software* (programas Java) que roda em uma máquina virtual Java (FIG. 8.3). Cada instância em operação do ambiente *runtime* JADE é chamada Continente, pois pode conter vários agentes. O conjunto de continentes ativos é chamado Plataforma. Um único continente especial deve sempre estar ativo na plataforma e todos os outros continentes registram-se com ele tão logo se iniciem. Decorre disto que o primeiro continente a se iniciar em uma plataforma deve ser um continente principal, e todos os outros continentes devem ser continentes normais (isto é, não principais). Deve ser "dito" aos continentes normais onde encontrar (*host* e porta) seu continente principal (isto é, o continente principal com que se registrar).

Se um outro continente principal é iniciado em algum lugar na rede, ele constitui uma plataforma diferente na qual continentes normais podem possivelmente se registrar. A FIG. 8.3, mostra duas plataformas JADE compostas respectivamente de três continentes e um continente.

Os agentes têm identificação única e, desde que eles saibam os nomes uns dos outros, podem se comunicar independentemente de suas localizações reais: mesmo continente (e.g., agentes A2 e A3), continentes diferentes na mesma plataforma (e.g., A1 e A2), ou diferentes plataformas (e.g., A4 e A5) (FIG. 8.3).

Além da capacidade de aceitar registro de outros continentes, o continente principal difere do continente normal por manter dois agentes especiais, automaticamente iniciados quando o continente principal é lançado. São o agente Sistema de Gerência de Agente (Agent Management System – AMS) e o agente Facilitador de Diretório (Directory Facilitator – DF).

O AMS provê o serviço de denominação (isto é, o AMS assegura que cada agente na plataforma tenha um nome único) e representa a autoridade na plataforma (por exemplo, é possível criar/matar agentes em continentes remotos requisitando a operação ao AMS).

O DF provê um serviço de Páginas Amarelas por meio do qual um agente pode encontrar outros agentes fornecendo os serviços dos quais ele necessita para cumprir suas metas.

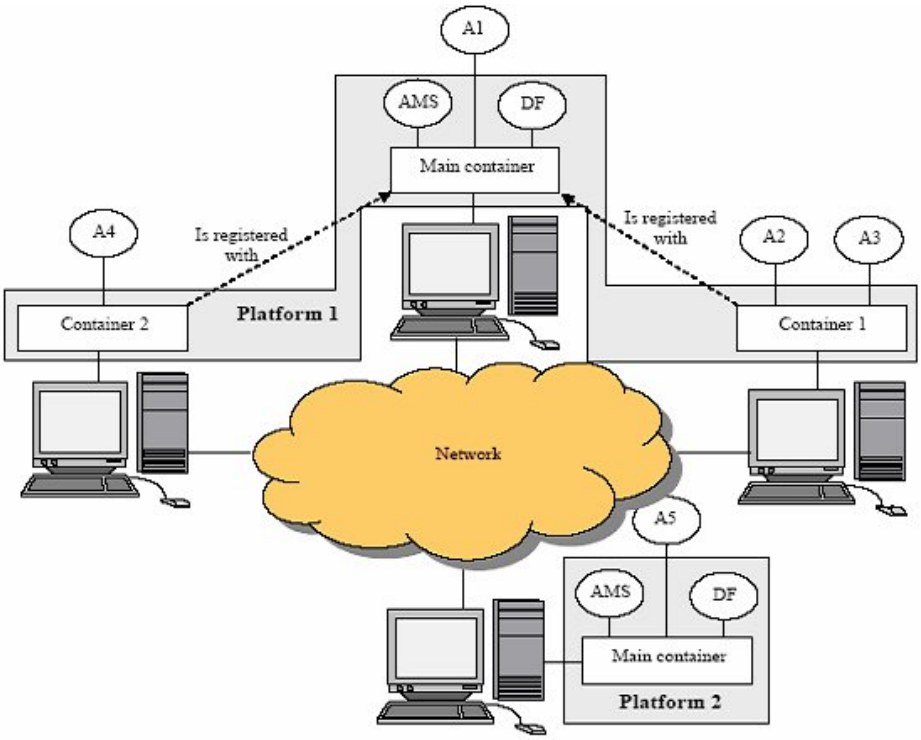


FIG 8. 3: Continentes e Plataformas (CAIRE, 2003)

### 8.2.2 FERRAMENTAS GRÁFICAS

Juntamente com o ambiente *runtime*, a JADE fornece diversas ferramentas. Cada uma delas é empacotada em um agente que obedece as mesmas regras, capacidades de comunicação e o mesmo ciclo de vida de um agente de aplicação genérico.

#### 8.2.2.1 AGENTE DE MONITORAMENTO REMOTO

O Agente de Monitoramento Remoto (RMA) permite controlar o ciclo de vida da plataforma de agente e de todos os agentes registrados. A arquitetura distribuída do JADE permite também controle remoto, onde a GUI do RMA é usada para controlar a execução dos agentes e seus ciclos de vida desde um *host* remoto. Um RMA é um objeto Java, instância da classe `jade.tools.rma.rma` e pode ser lançado a partir da linha de comando como um agente ordinário (isto é, com o comando `java jade.Boot myConsole:jade.tools.rma.rma`, por exemplo). Mais de um RMA pode ser iniciado na mesma plataforma desde que cada instância tenha um nome local diferente, mas somente um RMA pode ser executado no mesmo continente de agente. A FIG. 8.4 mostra a janela do RMA.

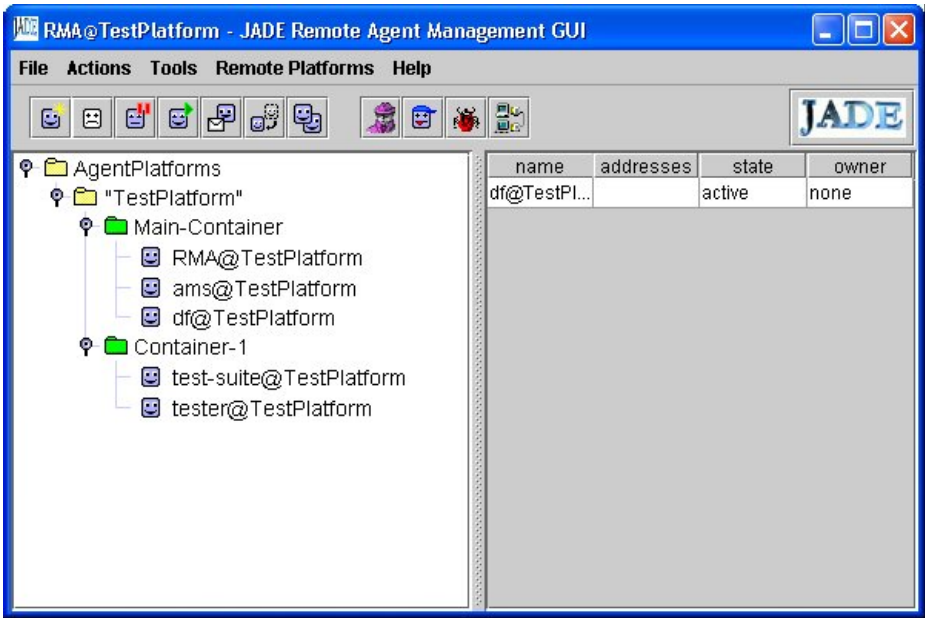


FIG 8. 4: Instantâneo da GUI do RMA (BELLIFEMINE, 2005a)

O RMA tem funcionalidade poderosa que permite, por exemplo, a criação, migração, suspensão, retomada e morte de agente; o envio de mensagem a agentes selecionados; adição e remoção de plataforma remota; etc.

### 8.2.2.2 O AGENTE FICTÍCIO

A ferramenta Agente Fictício (FIG. 8.5) permite a interação sob medida dos usuários com agentes JADE. A GUI permite compor e enviar mensagens ACL e manter uma lista de todas as mensagens ACL enviadas e recebidas. Esta lista pode ser examinada pelo usuário e cada mensagem pode ser vista em detalhe ou mesmo editada. Além disto, a lista de mensagem pode ser salva no disco e recuperada mais tarde. Muitas instâncias do Agente Fictício podem ser iniciadas como e onde necessário. O Agente Fictício pode ser lançado tanto do menu *Tool* do RMA, quanto da linha de comando, como a seguir:

```
Java jade.Boot theDummy:jade.tools.DummyAgent.DummyAgent.
```

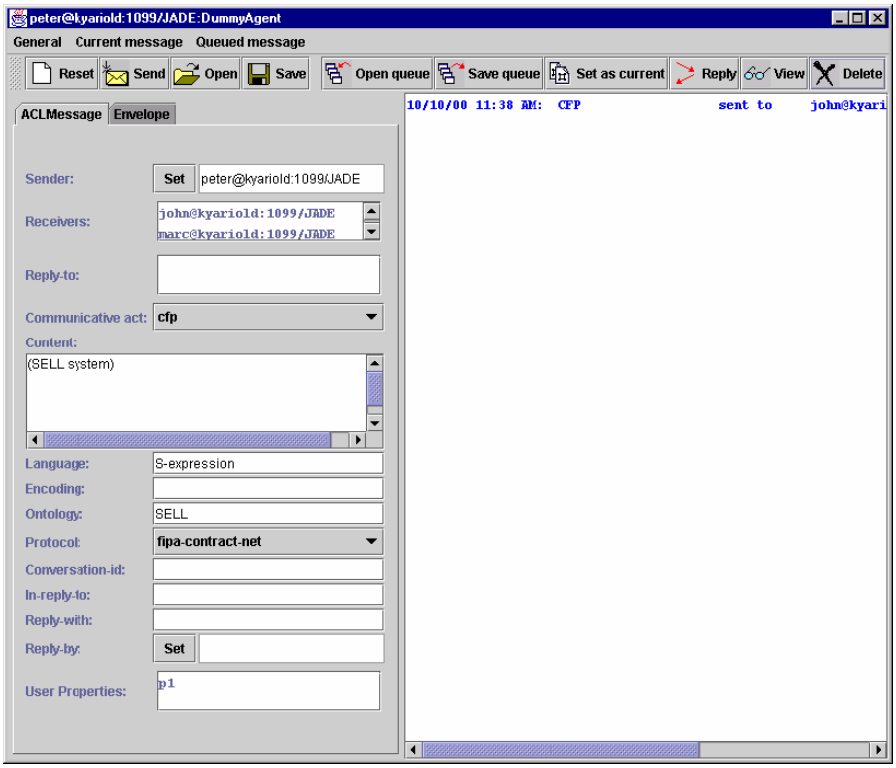


FIG 8. 5: Instantâneo da GUI do Agente Fictício (BELLIFEMINE, 2005a)

### 8.2.2.3 A GUI DO DF

Uma GUI do DF (FIG. 8.6) pode ser lançada a partir do menu Tools do RMA. Esta ação é na realidade implementada pelo envio de uma mensagem ACL ao DF solicitando-lhe que mostre sua GUI. Portanto, a GUI somente pode ser mostrada no *host* onde a plataforma (o continente principal) foi executada.

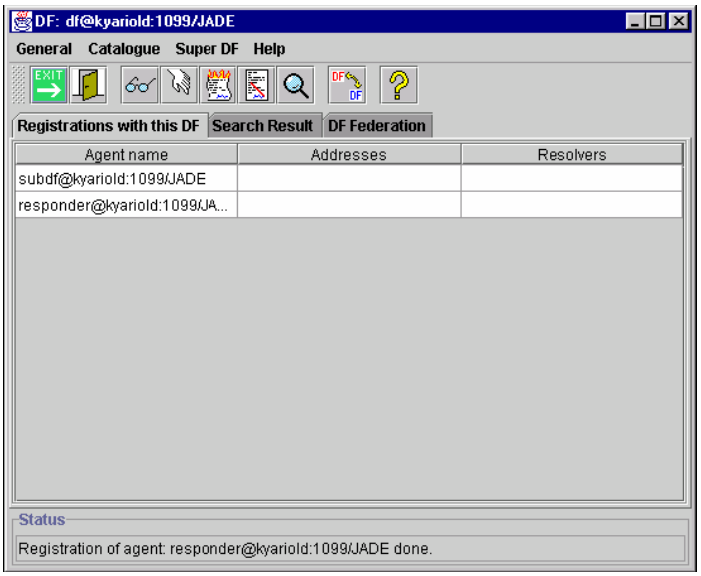


FIG 8. 6: Instantâneo da GUI do DF (BELLIFEMINE, 2005a)

Usando esta GUI, o usuário pode interagir com o DF: ver a descrição dos agentes registrados, registrar e cancelar registro de agentes, modificar a descrição de agente registrado e também pesquisar descrições de agentes. A GUI permite também federar o DF com outros DFs e criar uma rede complexa de domínios e sub-domínios de páginas amarelas. Qualquer DF federado, mesmo se residente em uma plataforma de agente não-JADE, pode ser controlado pela mesma GUI e as mesmas operações básicas (*view/register/deregister/modify/search*) podem ser executadas no DF remoto.

### 8.2.2.4 O AGENTE PERSCRUTADOR

O Agente Perscrutador (*Sniffer Agent*) mostra em sua GUI (FIG. 8.6) todas as mensagens dirigidas ao ou oriundas do agente/grupo de agentes que o usuário tenha decidido monitorar.

Ele pode ver todas as mensagens e salvá-las em disco. Pode também salvar todas as mensagens rastreadas e recarregá-las de um único arquivo para análise posterior.

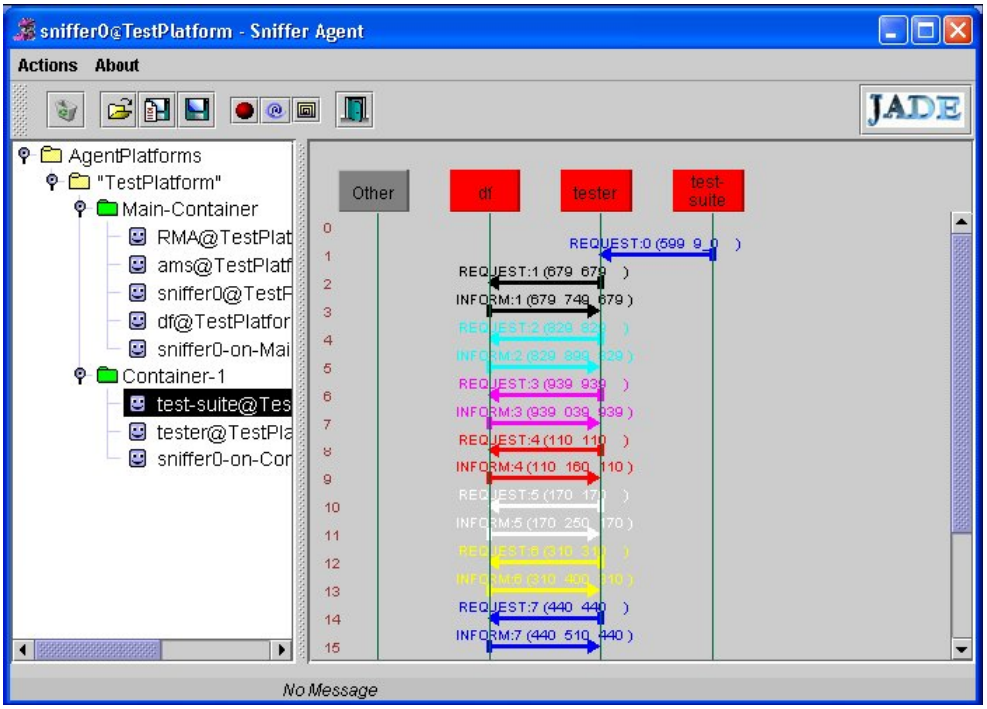


FIG 8. 7: Instantâneo da GUI do Agente Perscrutador (BELLIFEMINE, 2005a)

Este agente pode ser iniciado desde o menu *Tools* do RMA e também da linha de comando seguinte: `java jade.Boot sniffer:jade.tools.sniffer.Sniffer.`

### 8.2.2.5 AGENTE *INTROSPECTOR*

Esta ferramenta (FIG. 8.8) permite monitorar e controlar o ciclo de vida de um agente em execução e suas mensagens trocadas, as filas de mensagens enviadas e recebidas. Permite também monitorar as filas de comportamentos, incluindo executá-los passo a passo.

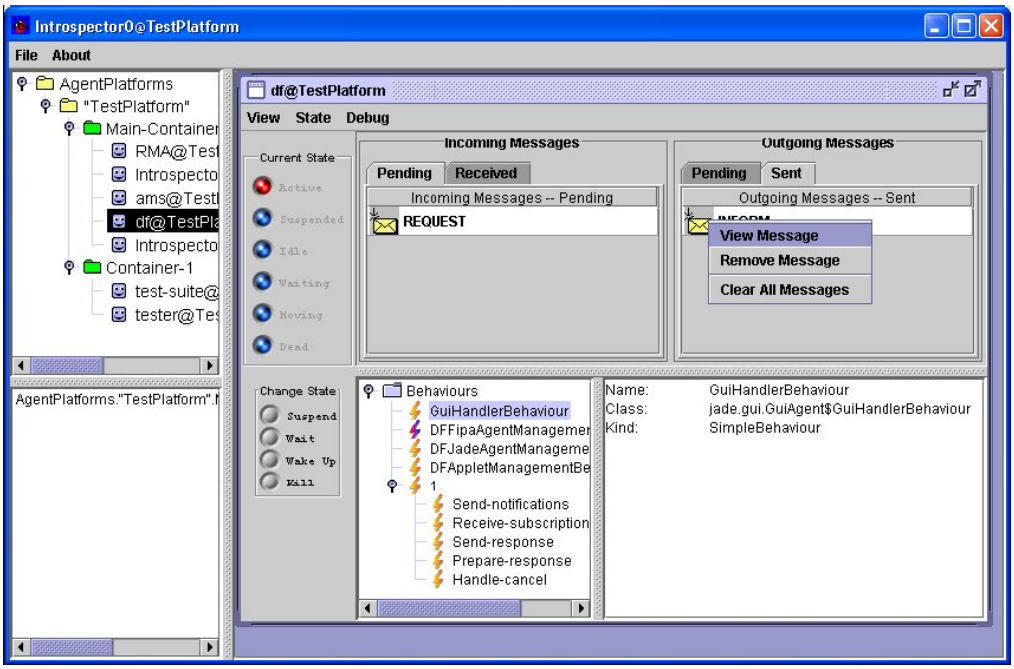


FIG 8. 8: Instantâneo da GUI do *Introspector* (BELLIFEMINE, 2005a)

### 8.2.3 OBSERVAÇÃO SOBRE O AMBIENTE DE OPERAÇÃO JADE

Além de sustentar a execução dos agentes, o ambiente de operação JADE dá suporte ao desenvolvimento do *software*. Como se viu, o ambiente de operação JADE consiste no ambiente *runtime* JADE e em ferramentas gráficas, diversas das quais permitem o monitoramento das mensagens trocadas pelos agentes. Pode-se, mesmo, ter acesso aos conteúdos das mensagens, e, caso tenha sido adotada a SL, pode-se decodificá-las. Para sistemas comunicativos, tais possibilidades são essenciais, pois o comportamento do sistema é fortemente dependente das mensagens trocadas pelos agentes. Assim, dando suporte ao monitoramento das mensagens, o ambiente de operação apóia atividades de desenvolvimento – no caso, as atividades de teste e depuração da troca de mensagens.

## 8.3 MaSE, JADE E FIPA

Há diferenças entre o tratamento dado pela MaSE à comunicação entre agentes e o tratamento dado pela FIPA e o JADE. A MaSE dá ênfase aos aspectos comunicativos dos agentes. Não sugere, contudo, um vocabulário de performativas e, tampouco, protocolos para

as conversações. A FIPA também dá importância à comunicação, mas, diferentemente da MaSE, especifica uma linguagem de comunicação de agentes (ACL) que padroniza as performativas e que estrutura os campos da mensagem ACL. A FIPA reconhece ainda a necessidade da informação líquida mantida na mensagem ser codificada e decodificada corretamente por todos os agentes. Sugere uma linguagem para tal (SL). O JADE, finalmente, implementa a ACL e a SL.

Uma outra diferença é que a FIPA considera o uso do conceito ontologia, o JADE o implementa, enquanto a MaSE não o cita. Como mencionado, a ontologia permite que as mensagens sejam interpretadas coerentemente por todos os agentes, independentemente da forma como sejam transmitidas – a importância da ontologia no projeto seria, portanto, a mesma das comunicações. O JADE aproveita a ontologia para codificar/decodificar automaticamente as mensagens, tornando-as adequadas ao tratamento intra-agente e, também, às necessidades de transmissão.

Essas diferenças entre a MaSE e, principalmente, o JADE têm reflexos no modelo que deveria ser produzido pelo Projeto para uso no Desenvolvimento. O modelo deveria prever uma ontologia que orientasse a definição das mensagens trocadas pelos agentes. Igualmente, as performativas deveriam estar contidas na especificação vocabular de alguma plataforma de desenvolvimento de agente escolhida pelo projetista, o que poderia ser alertado pela técnica de projeto – afinal, o sistema tem de ser desenvolvido em alguma plataforma. Finalmente, os protocolos que regem as conversações também deveriam ser escolhidos durante o projeto, e as escolhas deveriam constar em algum dos sub-modelos.

Na literatura é muito citada a lacuna que existiria entre as técnicas de análise e projeto e as técnicas e plataformas de desenvolvimento (SUDEIKAT, 2004), (AMOR, 2004), (CUESTA-MORALES, 2004). Tal *gap* tem ensejado o surgimento de técnicas e ferramentas que pretendem saná-lo. Não se conseguiu, contudo, na bibliografia, uma descrição clara do que seria ele de fato. Aqui, o *gap* é interpretado como sendo a falta dos sub-modelos antes referida.

# 9 SUBSISTEMAS DESENVOLVIDOS COM JADE

Visando a avaliação da exeqüibilidade do conjunto composto pela MaSE, modelo produzido com a MaSE, padrões FIPA, JADE, ontologia, etc. elaborou-se dois subsistemas simplificados do simulador. No primeiro, é usado um subconjunto restrito de técnicas JADE, enquanto que no segundo o elenco de técnicas JADE é bastante abrangente. O protocolo a que se recorreu no primeiro é mais complexo do que o protocolo usado no segundo.

## 9.1 ESCOLHA DE MONITOR

Este primeiro subsistema deriva das conversações "Solicitação de Custos", "Solicitação de Recurso" e "Informação de Recursos" que se dão entre os agentes "Coordenador de Monitoramento", "Monitor" e "Gerência de Monitoramento Interface" (FIG. 6.51) e apóia-se em técnicas de sugeridas em tutoriais JADE. O objetivo das conversações mencionadas é selecionar, dentre os diversos DANTs, aquele de custo de monitoramento mínimo (o custo é função da existência do recurso no DANT – inexistência significa custo infinito –, da distância do DANT à área a ser monitorada, da qualidade do recurso dentro de sua categoria, etc.). Neste modelo simplificado, os custos dos recursos não são calculados pelos agentes monitores, mas informados pelo usuário quando ingressa com os dados de recursos do DANT.

O agente "Gerência de Monitoramento Interface" não é considerado por ora, pois sua inserção *a posteriori* é simples, uma vez dominadas as técnicas JADE.

O resultado da conversação, qual seja, o DANT cujo recurso tem custo mínimo, é detido pelo "Coordenador de Monitoramento" ao final da conversação.

Usa-se aqui amplamente as classes `agent` e `behaviour`. Diversos tipos de classe `behaviour` fazem parte do subsistema e os métodos modificáveis das classes `agent` e `behaviour` são manipulados. Interage-se com os serviços de páginas amarelas da plataforma. A conversação entre as duas classes de agentes, "Monitor" e "Coordenador de Monitoramento", é tratada explicitamente pelos programas mostrados a seguir. Usa-se exclusivamente as performativas da ACL, e a conversação se dá segundo o protocolo *contract net* – sem o suporte JADE a protocolos e sem adesão estrita ao FIPA-ContractNet-Protocol.

Como o conteúdo (o *content* da mensagem ACL) é muito simples, o subsistema foi construído sem ontologia e sem uso do codec JADE.

Para permitir que sejam informados ao agente "Monitor" os recursos do respectivo DANT, e permitir que seja informado ao agente "Coordenador de Monitoramento" o recurso a ser procurado, há duas classes GUI, uma para cada classe destes agentes. Assim, há quatro arquivos de classe no sistema. Janelas GUI, por serem reativas, precisam ser acopladas cuidadosamente aos agentes que, diferentemente das janelas GUI, são pró-ativos. As janelas GUI em si mesmas, no geral, apóiam-se em técnicas Java, e não em técnicas JADE.

Os quatro arquivos de classe são o MonitorAgent, correspondente da classe de agente "Monitor"; o CoordrMonitAgent, que corresponde à classe "Coordenador de Monitoramento"; e as classes GUI, MonitorGui e CoordrMonitGui. Os códigos dos quatro arquivos estão postos a seguir.

### 9.1.1 CLASSE MONITORGUI

Cada DANT possui recursos e cada recurso tem um custo de uso associado. Os dados sobre estes recursos e respectivos custos são mantidos por instâncias da classe de agente MonitorAgent – há um e somente um agente desta classe para cada DANT. Neste modelo simplificado, os dados são passados ao agente MonitorAgent pelo usuário através da janela suportada pela classe MonitorGui – esta classe serve sustentar o diálogo do usuário com o agente MonitorAgent. A seguir, o código.

```
import java.awt.*;

class MonitorGui extends JFrame
{
   private MonitorAgent myAgent;

   private JTextField campoNomeRecurso, campoCustoRecurso;

   MonitorGui(MonitorAgent a)
   {
      super(a.getLocalName());

      myAgent = a;

      JPanel p = new JPanel();
      p.setLayout(new GridLayout(2, 2));
      p.add(new JLabel("Nome do Recurso:"));
      campoNomeRecurso = new JTextField(15);
      p.add(campoNomeRecurso);
      p.add(new JLabel("Custo:"));
      campoCustoRecurso = new JTextField(15);
      p.add(campoCustoRecurso);
      getContentPane().add(p, BorderLayout.CENTER);

      JButton addButton = new JButton("Insere");
      addButton.addActionListener( new ActionListener() {
```

```
      public void actionPerformed(ActionEvent ev)
      {
         try
         {
            String nomeRecurso = campoNomeRecurso.getText().trim();
            String custoRecurso = campoCustoRecurso.getText().trim();
            myAgent.atualizaCatalogo(nomeRecurso, Integer.parseInt(custoRecurso));
            campoNomeRecurso.setText("");
            campoCustoRecurso.setText("");
         }
         catch (Exception e)
         {
            JOptionPane.showMessageDialog(MonitorGui.this, "Dados incorretos.
                      "+e.getMessage(), "Erro", JOptionPane.ERROR_MESSAGE);
         }
      }});
      p = new JPanel();
      p.add(addButton);
      getContentPane().add(p, BorderLayout.SOUTH);

      // Faz o agente - que possui esta GUI - terminar quando o usuário fecha
      // a GUI usando o botão no canto direito superior
      addWindowListener(new WindowAdapter() {
         public void windowClosing(WindowEvent e)
         {
            myAgent.doDelete();
         }
      });

      setResizable(false);
   }

   public void show()
   {
      pack();
      Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
      int centerX = (int)screenSize.getWidth() / 2;
      int centerY = (int)screenSize.getHeight() / 2;
      setLocation(centerX - getWidth() / 2, centerY - getHeight() / 2);
      super.show();
   }
}
```

### 9.1.2 CLASSE MONITORAGENT

Classe de agente que mantém registro dos recursos do DANT. O registro inclui o nome do recurso e seu custo (no sistema real, o custo é calculado *ad hoc*, levando em conta, dentre outros elementos, as coordenadas da área a ser monitorada passadas pelo Coordenador de Monitoramento). Adiante, o código da classe.

```
* Created on Aug 10, 2005
import jade.core.Agent;

public class MonitorAgent extends Agent
{
   //O catálogo de recursos disponíveis
   //(relaciona o nome do recurso com o custo)
   private Hashtable catalogo;
   //O GUI por meio do qual o usuário pode acrescentar
   //recursos de monitoramento ao catálogo
   private MonitorGui myGui;

   //Põe as inicializações de agentes aqui
   protected void setup()
   {
```

```
   //Cria o catálogo
   catalogo = new Hashtable();

   /Registra o serviço de monitoramento nas páginas amarelas
   DFAgentDescription dfd = new DFAgentDescription();
   dfd.setName(getAID());
   ServiceDescription sd = new ServiceDescription();
   sd.setType("Monitoramento");
   sd.setName("Recursos-monitoramento");
   dfd.addServices(sd);
   try
   {
      DFService.register(this, dfd);
   }
   catch (FIPAException fe)
   {
      fe.printStackTrace();
   }
   //Cria e mostra o GUI
   myGui = new  MonitorGui(this);
   myGui.show();

   //Acrescenta o comportamento de atendendimento
   //de pedidos de oferta de recurso, pedidos oriundos do
   //agente coordenador de monitoramento
   addBehaviour(new ServidorPedidoOferta());
   //Acrescenta o comportamento de atendimento de
   //pedidos de reserva de recurso, pedidos oriundos do agente
   //coordenador de monitoramento
   addBehaviour(new ServidorPedidoReserva());
}//Fim do setup()

//Põe as operações de limpeza aqui
protected void takeDown()
{
   //Cancela registro nas páginas amarelas
   try
   {
      DFService.deregister(this);
   }
   catch (FIPAException fe)
   {
      fe.printStackTrace();
   }

   //Fecha a GUI
   myGui.dispose();
   System.out.println("Agente-Monitor "+getAID().getName()+" terminando");
}//Fim do método takeDown

/**
* Isto é invocado pelo GUI quando o usuário
* acrescenta um novo recurso ao DANT
*/
public void atualizaCatalogo(final String nome, final int custo)
{
   addBehaviour(new OneShotBehaviour()
   {
      public void action()
      {
         //System.out.println("Entrei no action()");
         catalogo.put(nome, new Integer(custo));
      }
   });
}
//CyclicBehaviour é um comportamento que nunca é completado,
//isto é, comportamento que implementa o método done() retornando
//false.
private class ServidorPedidoOferta extends CyclicBehaviour
{
   public void action()
   {
      MessageTemplate mt = MessageTemplate.MatchPerformative(ACLMessage.CFP);
      ACLMessage msg = myAgent.receive(mt);
      if (msg != null)
```

```
        {
           //Mensagem recebida. Processe-a.
           String nome = msg.getContent();
           ACLMessage reply = msg.createReply();

           Integer custo = (Integer) catalogo.get(nome);
           if (custo != null)
           {
              //O recurso pedido está disponível.
              //Responda com o custo
              reply.setPerformative(ACLMessage.PROPOSE);
              reply.setContent(String.valueOf(custo.intValue()));
           }
           else
           {
              //O recurso solicitado não está disponível para reserva
              reply.setPerformative(ACLMessage.REFUSE);
              reply.setContent("indisponível");
           }
           myAgent.send(reply);
           }
        else
           block();
     }
  }//Fim da classe interna ServidorPedidoOferta

  private class ServidorPedidoReserva extends CyclicBehaviour
  {
     public void action()
     {
        MessageTemplate mt = MessageTemplate.MatchPerformative(ACLMessage.ACCEPT_PROPOSAL);
        ACLMessage msg = myAgent.receive(mt);
        if (msg != null)
        {
           // Mensagem ACCEPT_PROPOSAL recebida. Processe-a.
           String nome = msg.getContent();
           ACLMessage reply = msg.createReply();
           Integer custo = (Integer) catalogo.remove(nome);
           if (custo != null)
           {
              reply.setPerformative(ACLMessage.INFORM);
              System.out.println(nome+" reservado ao agente "+msg.getSender().getName()
                                                    + " pelo agente "+ getAID().getName());
           }
           else
           {
              // O recurso solicitado foi reservado a outro coordenador enquanto
              // a solicitação estava sendo processada.
              reply.setPerformative(ACLMessage.FAILURE);
              reply.setContent("indisponível");
           }
           myAgent.send(reply);
        }
        else
        {
           block();
        }
     }
   } //Fim da classe interna OfferRequestsServer
  }
```

### 9.1.3 CLASSE COODRMONITAGENT

Esta classe é baseada na classe “Coordenador de Monitoramento”. Seu código está a seguir.

```
* Created on Aug 10, 2005
```

```
import jade.core.Agent;

//A classe pública CoordrMonitAgent é uma extensão da
//classe Agent
public class CoordrMonitAgent extends Agent
{
   //A variável NomeRecursoAlvo, do tipo String, contém
   //o nome do recurso necessário ao monitoramento
   private String NomeRecursoAlvo;
   //A lista dos agentes monitores conhecidos
   private AID[] agentesMonitores;

   private CoordrMonitGui myGui;
   //Põe inicializações de agentes aqui
   //Este é o primeiro método da classe CoordrMonitAgent
   protected void setup()
   {
      NomeRecursoAlvo = "";
      myGui = new CoordrMonitGui(this);
      myGui.show();
      System.out.println("Agente Coordenador de Monitoramento " + getAID().getName()
                                                                  + " iniciado.");

      addBehaviour(new TickerBehaviour(this, 5000)
      {
         protected void onTick()
         {
            if (NomeRecursoAlvo != "")
            {
               DFAgentDescription template = new  DFAgentDescription();
               ServiceDescription sd = new  ServiceDescription();
               sd.setType("Monitoramento");
               template.addServices(sd);
               try
               {
                  DFAgentDescription[] result = DFService.search(myAgent, template);
                  if (result != null && result.length > 0)
                  {
                     agentesMonitores = new AID[result.length];
                     for (int i=0; i < result.length; ++i)
                     {
                        agentesMonitores[i] = result[i].getName();
                     }
                  }
               }
               catch (FIPAException fe)
               {
                  System.out.println("Deu erro!");
                  fe.printStackTrace();
               }
               //myAgent é variável protegida que aponta para o
               //agente que está usando o comportamento
               //RequestPerformer é o comportamento
               myAgent.addBehaviour(new ExecutorPedido());
          }
         }
      });
   }//fim do método setup()
   /**
   * Êste método atualiza NomeRecursoAlvo, atributo de CoordrMonitAgent,
   * que recebe o título do livro a ser comprado
   *
   * @param recurso
   */
   public void atualizaRecursoAlvo(final String recurso)
   {
      addBehaviour(new OneShotBehaviour()
      {
         public void action()
         {
            NomeRecursoAlvo = recurso;
            if (NomeRecursoAlvo != "")
            {
               System.out.println("Tentando obter o recurso "+NomeRecursoAlvo);
```

```
            }
            else
            {
               //Faz o agente terminar imediatamente
               System.out.println("Nenhum nome de recurso foi especificado");
               doDelete();
            }
         }
      });
   }
   //Põe operações de limpeza de agentes aqui
   protected void takeDown()
   {
      myGui.dispose();
      //Imprime uma mensagem de encerramento
      System.out.println("Agente Coordenador de Monitoramento" + getAID().getName() +
                                                                  "terminando.");
   }
   /**
   * Classe interna ExecutorPedido
   * Este é o comportamento usado pelo agente coordenador de monitoramento para
   * requerer aos agentes monitores o recurso pretendido.
   */
   private class ExecutorPedido extends Behaviour
   {
      private AID melhorMonitor; //o agente que oferece a melhor oferta
      private int minimoCusto; //o mínimo custo oferecido
      private int ContRespostas = 0; //contador de respostas dos agentes monitores
      private MessageTemplate mt; //o gabarito das respostas a receber
      private int step = 0;

      public void action()
      {
         switch (step)
         {
            case 0:
            //envia a cfp (call for proposal) a todos os monitores
            if (agentesMonitores != null)
            {
               ACLMessage cfp = new ACLMessage(ACLMessage.CFP);
               for (int i=0; i < agentesMonitores.length; ++i)
                 cfp.addReceiver(agentesMonitores[i]);

               cfp.setContent(NomeRecursoAlvo);
               cfp.setConversationId("Monitoramento");
               cfp.setReplyWith("cfp"+System.currentTimeMillis()); //valor único
               myAgent.send(cfp);
               //prepara o gabarito para pegar as respostas
               //agentes monitores com as propostas
               mt = MessageTemplate.and(MessageTemplate.MatchConversationId(
                              "Monitoramento"), MessageTemplate.MatchInReplyTo(
                                                          cfp.getReplyWith()));
               step = 1;
            }
            break;
            case 1:
            //recebe todas as propostas/recusas dos agentes monitores
            ACLMessage reply = myAgent.receive(mt);
            if (reply != null)
            {
               //resposta recebida
               if (reply.getPerformative() == ACLMessage.PROPOSE)
               {
                  int custo = Integer.parseInt(reply.getContent());
                  if(melhorMonitor == null || custo < minimoCusto)
                  {
                     //Esta é a melhor oferta no presente momento
                     minimoCusto = custo;
                     melhorMonitor = reply.getSender();
                  }
               }
               ContRespostas++;
               if (ContRespostas >= agentesMonitores.length)
                  //recebemos todas as respostas
                  step = 2;
```

```
            }
            else
            {
               block();
            }
            break;
            case 2:
            //envia o pedido de recurso ao monitor que ofereceu
            //a melhor oferta
               ACLMessage order = new ACLMessage(ACLMessage.ACCEPT_PROPOSAL);
               order.addReceiver(melhorMonitor);
               order.setContent(NomeRecursoAlvo);
               order.setConversationId("Monitoramento");
               order.setReplyWith("pedido "+System.currentTimeMillis());
               myAgent.send(order);
               //prepara o gabarito para obter a resposta do pedido de reserva
               mt = MessageTemplate.and(MessageTemplate.MatchConversationId(
                         "Monitoramento"),
                             MessageTemplate.MatchInReplyTo(order.getReplyWith()));
               step = 3;
               break;
            case 3:
            //recebe a resposta do pedido de reserva do recurso
               reply = myAgent.receive(mt);
               if (reply != null)
               {
                  //resposta do pedido do recurso recebida
                  if (reply.getPerformative() == ACLMessage.INFORM)
                  {
                     //reserva bem sucedida. Podemos terminar
                     System.out.println(NomeRecursoAlvo+" alocação bem sucedida.");
                     System.out.println("Custo = "+minimoCusto);
                     myAgent.doDelete();
                  }
                  step = 4;
               }
               else
               {
                  block();
               }
               break;
         }
      }
      public boolean done()
      {
         return ((step==2 && melhorMonitor == null) || step == 4);
      }//fim do método done()
   }//fim da classe interna ExecutorPedido
}
```

### 9.1.4 CLASSE COORDRMONITGUI

As instâncias desta classe consubstanciam interfaces GUI de usuário por intermédio das quais o usuário informa qual recurso pesquisar. A seguir, o código.

```
* Created on Aug 19, 2005
import java.awt.*;

class CoordrMonitGui extends JFrame
{
   private CoordrMonitAgent myAgent;

   private JTextField campoNomeRecurso;
   CoordrMonitGui(CoordrMonitAgent a)
   {
      super(a.getLocalName());
      myAgent = a;
```

```
        JPanel p = new JPanel();
        p.setLayout(new GridLayout(2, 2));
        p.add(new JLabel("Nome do Recurso:"));
        campoNomeRecurso = new JTextField(15);
        p.add(campoNomeRecurso);
        getContentPane().add(p, BorderLayout.CENTER);

        JButton InsertButton = new JButton("Insere");
        InsertButton.addActionListener(

        new ActionListener()
        {

           public void actionPerformed(ActionEvent ev)
           {
              try
              {
                 String nomeRecurso = campoNomeRecurso.getText().trim();
                 myAgent.atualizaRecursoAlvo(nomeRecurso);
                 campoNomeRecurso.setText("");
              }
              catch (Exception e)
             {
                 JOptionPane.showMessageDialog(CoordrMonitGui.this, "Nome inválido. " +
                                  e.getMessage(), "Erro", JOptionPane.ERROR_MESSAGE);
              }
           }
        } );
        p = new JPanel();
        p.add(InsertButton);
        getContentPane().add(p, BorderLayout.SOUTH);

        // Faz o agente terminar quando o usuário fecha
        // a GUI usando o botão sobre o canto direito superior
        addWindowListener(new WindowAdapter()
        {
           public void windowClosing(WindowEvent e)
           {
              myAgent.doDelete();
           }
        } );

        setResizable(false);
     }
     public void show()
     {
        pack();
        Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
        int centerX = (int)screenSize.getWidth() / 2;
        int centerY = (int)screenSize.getHeight() / 2;
        setLocation(centerX - getWidth() / 2, centerY - getHeight() / 2);
        super.show();
     }
}
```

### 9.1.5 EXEMPLO DE EXECUÇÃO

De uma execução arbitrária deste subsistema, foram coletados dados sobre as mensagens trocadas pelos agentes, os valores de algumas destas mensagens e as informações prestadas pelos agentes na janela de comando sobre as operações por eles executadas. Neste exemplo, foram ativados três agentes monitores e um agente coordenador de monitoramento. Além deles, o agente DF é também monitorado.

### 9.1.5.1 MENSAGENS TROCADAS

A FIG. 9.1 é uma janela do Agente Perscrutador na qual estão representadas todas as mensagens trocadas pelos agentes do subsistema, mais o DF, na determinação de um recurso de monitoramento – uma câmara de vídeo – de mínimo custo.

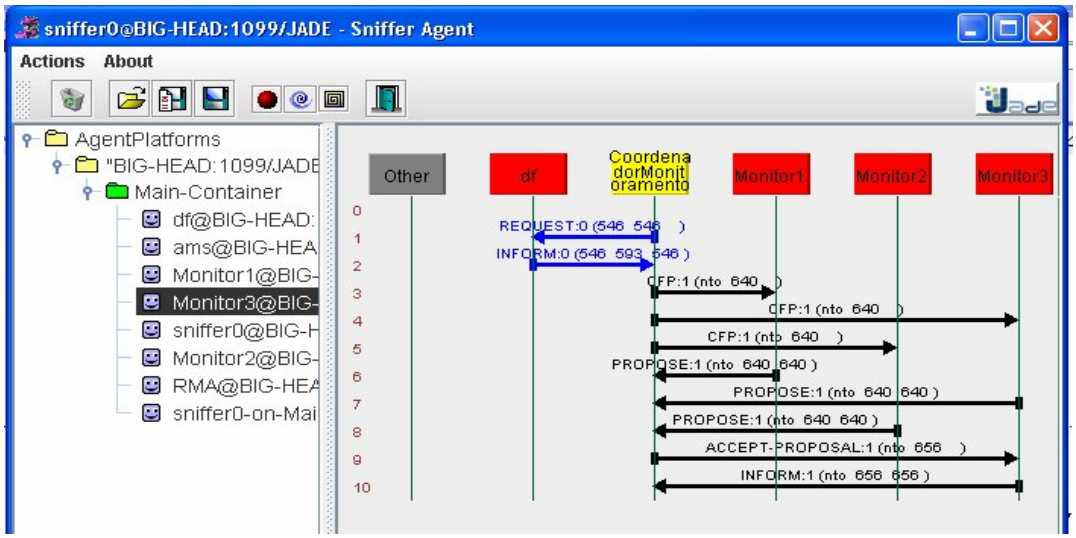


FIG 9. 1: Mensagens para Escolha de Recurso

### 9.1.5.2 EXEMPLO DE MENSAGEM TROCADA

A próxima figura, obtida a partir do Agente Perscrutador, mostra os componentes da mensagem CFP enviada a todos os agentes monitores (mensagens na linhas 3, 4 e 5, na figura anterior).

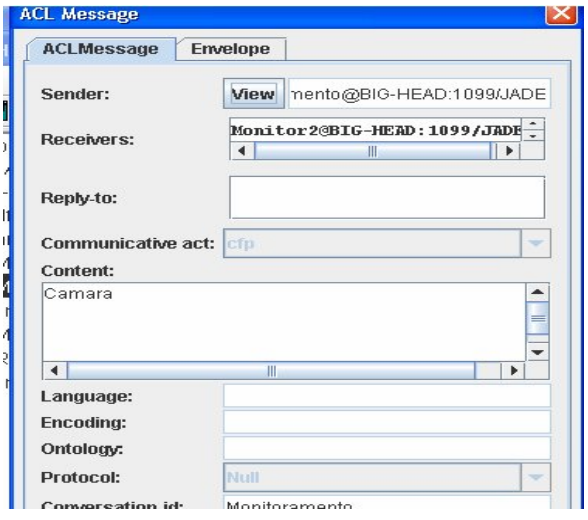


FIG 9. 2: A mensagem CFP

### 9.1.5.3 MENSAGENS DOS AGENTES NA JANELA DE COMANDOS

Além de usar as GUIs, os agentes produzem também mensagens na janela de comandos. As mensagens que ocorreram nesta execução são mostradas na FIG. 9.3.

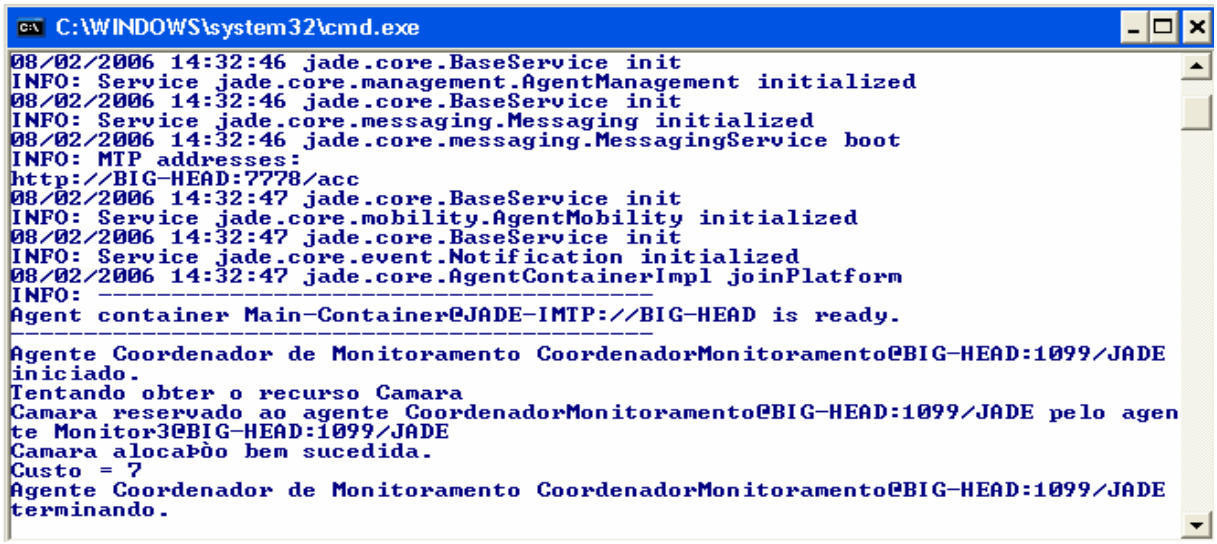


FIG 9. 3: Mensagens Produzidas pelos Agentes na Janela de Comando

## 9.2 COMANDO DE ATUADORES

Neste subsistema, usa-se os suportes JADE para linguagem de conteúdo, ontologia e protocolo de interação. Com eles, desenvolve-se uma versão simplificada da conversação "Comanda Atuadores" travada entre os agentes "Navegador DANT" e "Atuadores" (FIG. 6.51). Não se usou o Protégé e o beangenerator para a geração dos *schemas*.

Considera-se que os atuadores do DANT sejam dois propulsores vetorizados de empuxo (os respectivos eixos de inclinação podem ser variados) nos lados direito e esquerdo do dirigível; um propulsor fixo de empuxo na parte de trás; quatro pequenos balões (*balonetes*) para controle de flutuação; e, na traseira, dois lemes, um de direção e outro de profundidade[58]. Cada DANT possui um agente "Atuadores", que abstrai os atuadores do veículo, e um agente "Navegador DANT" (FIG. 6.92).

[58] Extraído de Dynamics and Control of a Herd of Sondes Guided by a Blimp on Titan, de Marco B. Quadrelli, Johnny Chang e Scott Kowalchuck. 14th AAS/AIAA Space Flight Mechanics Conference Maui, Hawaii, February 8-12,2004, AAS Publications Office, P.O. Box 28130, San Diego, CA 92198

O agente “Atuadores” controla três grupos de atuadores: propulsor, *balonete* e leme. Cada propulsor tem nome, localização e força de empuxo; cada *balonete*, nome, tipo e pressão interna. Cada leme, nome, tipo e direção. O agente “Atuadores” recebe requisição do agente “Navegador DANT” para alterar o estado de um dado atuador – e.g., a força de empuxo do propulsor direito, ou a pressão do *balonete* C – e, depois de alterado, assegurar que o estado seja mantido.

São usadas doze classes classificadas como classes de conceitos, ações de agente, agentes, interface gráfica e ontologia – neste caso específico, não se tem classe de predicado. A classe de ontologia define a ontologia (*schemas*) usada no subsistema – a combinação de ontologias é simples no JADE, portanto é fácil combinar as ontologias dos diversos subsistemas. As classes dos tipos conceito e ação de agente estendem a classe de ontologia e têm o código necessário para acesso aos seus respectivos atributos. Seus métodos são usados pelo *codec*, e, também, pelas classes do tipo agente. As classes do tipo interface gráfica permitem a interação usuário-agente. Finalmente, as classes do tipo agente contêm a funcionalidade específica do subsistema.

A classe de ontologia é a “OntoCmdoAtuadores”. As classes de conceitos são “balonete”, “direção”, “forcaEmpuxo”, “leme” e “propulsor”. As classes de ação de agente são “alteraBalonete”, “alteraLeme” e “alteraPropulsor”. As classes de agente são “Atuadores” e “Navegador DANT”. A classe de interface gráfica é a “JanelaNavegadorDANT”. A seguir, são listados os códigos das classes.

### 9.2.1 CLASSE “ONTOCMDOATUADORES”

```
import jade.content.onto.*;
import jade.content.schema.*;

public class OntoCmdoAtuadores extends Ontology {
   //O nome identificando esta ontologia
   public static final String ONTOLOGY_NAME = "Ontologia-Comando-Dirigivel";
   //VOCABULÁRIO
   public static final String FORCA_EMPUXO = "forcaEmpuxo";
   //a força é especificada pelos componentes (i,j,k)
   //obs.: refencial inercial a determinar
   public static final String FORCA_EMPUXO_I = "forcaEmpuxoI";
   public static final String FORCA_EMPUXO_J = "forcaEmpuxoJ";
   public static final String FORCA_EMPUXO_K = "forcaEmpuxoK";
   public static final String DIRECAO = "direcao";
   //a direção é especificada pelos componentes i, j e k
   public static final String DIRECAO_I = "direcaoI";
   public static final String DIRECAO_J = "direcaoJ";
   public static final String DIRECAO_K = "direcaoK";
   public static final String PROPULSOR = "propulsor";
   public static final String NOME_PROPULSOR = "nomePropulsor";
   public static final String LOC_PROPULSOR = "locPropulsor";
   //localizações: lateralDireita, lateralEsquerda, popa
   public static final String FORCA_EMPUXO_PROPULSOR = "forcaEmpuxoPropulsor";
```

```
public static final String LEME = "leme";
public static final String NOME_LEME = "nomeLeme";
public static final String TIPO_LEME = "tipoLeme";
//tipos: leme de direção e leme de profundidade
public static final String DIRECAO_LEME = "direcaoLeme";
public static final String BALONETE = "balonete";
public static final String NOME_BALONETE = "nomeBalonete";
public static final String TIPO_BALONETE = "tipoBalonete";
public static final String PRESSAO = "pressao";
public static final String ALT_EST_PROPULSOR = "alteraPropulsor";
public static final String PROPULSOR_A_ALTERAR = "propulsor";
public static final String ALT_EST_LEME = "alteraLeme";
public static final String LEME_A_ALTERAR = "leme";
public static final String ALT_EST_BALONETE = "alteraBalonete";
public static final String BALONETE_A_ALTERAR = "balonete";
//A instância única desta ontologia
private static Ontology AInstanciaOntologica = new OntoCmdoAtuadores();
//Este é o método para se ter acesso ao objeto
//único da ontologia loja de música
public static Ontology getInstance()
{
   return AInstanciaOntologica;
}
//Construtor privado
private OntoCmdoAtuadores()
{
   //a ontologia comando de atuadores extende a ontologia básica (do JADE)
   super(ONTOLOGY_NAME, BasicOntology.getInstance());
   try
   {
      add(new ConceptSchema(PROPULSOR), propulsor.class);
      add(new ConceptSchema(LEME), leme.class);
      add(new ConceptSchema(BALONETE), balonete.class);
      add(new ConceptSchema(FORCA_EMPUXO), forcaEmpuxo.class);
      add(new ConceptSchema(DIRECAO), direcao.class);
      add(new AgentActionSchema(ALT_EST_PROPULSOR), alteraPropulsor.class);
      add(new AgentActionSchema(ALT_EST_LEME), alteraLeme.class);
      add(new AgentActionSchema(ALT_EST_BALONETE), alteraBalonete.class);
      //Estrutura do schema para o conceito forçaEmpuxo
      ConceptSchema cs = ConceptSchema) getSchema(FORCA_EMPUXO);
      cs.add(FORCA_EMPUXO_I, (PrimitiveSchema) getSchema(BasicOntology.FLOAT),
                              ObjectSchema.MANDATORY); //ranhura para a componente x
      cs.add(FORCA_EMPUXO_J, (PrimitiveSchema) getSchema(BasicOntology.FLOAT),
                              ObjectSchema.MANDATORY); //ranhura para a componente y
      cs.add(FORCA_EMPUXO_K, (PrimitiveSchema) getSchema(BasicOntology.FLOAT),
                              ObjectSchema.MANDATORY); //ranhura para a componente z
      cs = (ConceptSchema) getSchema(DIRECAO);
      cs.add(DIRECAO_I, (PrimitiveSchema) getSchema(BasicOntology.FLOAT),
                              ObjectSchema.MANDATORY); //ranhura para a componente x
      cs.add(DIRECAO_J, (PrimitiveSchema) getSchema(BasicOntology.FLOAT),
                              ObjectSchema.MANDATORY); //ranhura para a componente y
      cs.add(DIRECAO_K, (PrimitiveSchema) getSchema(BasicOntology.FLOAT),
                              ObjectSchema.MANDATORY); //ranhura para a componente z
      //Estrutura do schema para o conceito propulsor
      cs = (ConceptSchema) getSchema(PROPULSOR);
      //atribui o schema PROPULSOR à variável cs
      cs.add(NOME_PROPULSOR, (PrimitiveSchema) getSchema(BasicOntology.STRING),
                        ObjectSchema.OPTIONAL); //ranhura para nomePropulsor (opcional)
      cs.add(LOC_PROPULSOR, (PrimitiveSchema) getSchema(BasicOntology.STRING),
                                                      ObjectSchema.MANDATORY);
      //acrescenta a ranhura LOC_PROPULSOR ao
      //schema cs; o valor da ranhura não pode ser
      //nulo; formato string.
      cs.add(FORCA_EMPUXO_PROPULSOR, (ConceptSchema) getSchema(FORCA_EMPUXO),
                                                            ObjectSchema.MANDATORY);
      //Estrutura do schema para o conceito leme
      cs = (ConceptSchema) getSchema(LEME);
      cs.add(NOME_LEME, (PrimitiveSchema) getSchema(BasicOntology.STRING),
                                                              ObjectSchema.OPTIONAL);
      cs.add(TIPO_LEME, (PrimitiveSchema) getSchema(BasicOntology.STRING),
                                                             ObjectSchema.MANDATORY);
      cs.add(DIRECAO_LEME, (ConceptSchema) getSchema(DIRECAO),
                                                             ObjectSchema.MANDATORY);
      //Estrutura do schema para o conceito balonete
      cs = (ConceptSchema) getSchema(BALONETE);
```

```
            cs.add(NOME_BALONETE, (PrimitiveSchema) getSchema(BasicOntology.STRING),
                                                              ObjectSchema.OPTIONAL);
            cs.add(TIPO_BALONETE, (PrimitiveSchema) getSchema(BasicOntology.STRING),
                                                              ObjectSchema.MANDATORY);
            cs.add(PRESSAO, (PrimitiveSchema) getSchema(BasicOntology.FLOAT),
                                                              ObjectSchema.MANDATORY);
            //Estrutura do schema para a ação de agente Altera Estado do Propulsor
            AgentActionSchema as = (AgentActionSchema) getSchema(ALT_EST_PROPULSOR);
            as.add(PROPULSOR_A_ALTERAR, (ConceptSchema) getSchema(PROPULSOR),
                                                              ObjectSchema.MANDATORY);
            //Estrutura do schema para a ação de agente Altera Estado do Leme
            as = (AgentActionSchema) getSchema(ALT_EST_LEME);
            as.add(LEME_A_ALTERAR, (ConceptSchema) getSchema(LEME),
                                                              ObjectSchema.MANDATORY);
            as = (AgentActionSchema) getSchema(ALT_EST_BALONETE);
            as.add(BALONETE_A_ALTERAR, (ConceptSchema) getSchema(BALONETE),
                                                              ObjectSchema.MANDATORY);
        }
        catch (OntologyException oe)
        {
            oe.printStackTrace();
        }
    }
}
```

### 9.2.2 CLASSE “BALONETE”

```
//Classe associada ao schema balonete
import jade.content.Concept; //Porque balonete é um conceito
public class balonete implements Concept
{
    private String nomeBalonete;
    private String tipoBalonete;
    private double pressao;
    public String getNomeBalonete()
    {
        return nomeBalonete;
    }
    public void setNomeBalonete(String N)
    {
        nomeBalonete = N;
    }
    public String getTipoBalonete()
    {
        return tipoBalonete;
    }
    public void setTipoBalonete(String T)
    {
        tipoBalonete = T;
    }
    public double getPressao()
    {
        return pressao;
    }
    public void setPressao(double P)
    {
        pressao = P;
    }
}
```

### 9.2.3 CLASSE “DIREÇÃO”

```
//Classe associada ao schema direcao
import jade.content.Concept; //Porque direcao é um conceito
public class direcao implements Concept
{
    private double direcaoI;
```

```
   private double direcaoJ;
   private double direcaoK;
   public double getDirecaoI()
   {
      return direcaoI;
   }
   public void setDirecaoI(double DI)
   {
      direcaoI = DI;
   }
   public double getDirecaoJ()
   {
      return direcaoJ;
   }
   public void setDirecaoJ(double DJ)
   {
      direcaoJ = DJ;
   }
   public double getDirecaoK()
   {
      return direcaoK;
   }
   public void setDirecaoK(double DK)
   {
      direcaoK = DK;
   }
}
```

### 9.2.4 CLASSE “FORCAEMPUXO”

```
//Classe associada ao schema forcaEmpuxo
import jade.content.Concept; //Porque forcaEmpuxo é um conceito
public class forcaEmpuxo implements Concept
{
   private double forcaEmpuxoI;
   private double forcaEmpuxoJ;
   private double forcaEmpuxoK;
   public double getForcaEmpuxoI()
   {
      return forcaEmpuxoI;
   }
   public void setForcaEmpuxoI(double FI)
   {
      forcaEmpuxoI = FI;
   }
   public double getForcaEmpuxoJ()
   {
      return forcaEmpuxoJ;
   }
   public void setForcaEmpuxoJ(double FJ)
   {
      forcaEmpuxoJ = FJ;
   }
   public double getForcaEmpuxoK()
   {
      return forcaEmpuxoK;
   }
   public void setForcaEmpuxoK(double FK)
   {
      forcaEmpuxoK = FK;
   }
}
```

### 9.2.5 CLASSE “LEME”

```
//Classe associada ao schema leme
import jade.content.Concept; //Porque leme é um conceito
```

```
public class leme implements Concept
{
   private String nomeLeme;
   private String tipoLeme;
   private direcao direcaoLeme;
   public String getNomeLeme()
   {
      return nomeLeme;
   }
   public void setNomeLeme(String N)
   {
      nomeLeme = N;
   }
   public String getTipoLeme()
   {
      return tipoLeme;
   }
   public void setTipoLeme(String T)
   {
      tipoLeme = T;
   }
   public direcao getDirecaoLeme()
   {
      return direcaoLeme;
   }
   public void setDirecaoLeme(direcao D)
   {
      direcaoLeme = D;
   }
}
```

### 9.2.6 CLASSE “PROPULSOR”

```
//Classe associada ao schema propulsor
import jade.content.Concept; //Porque propulsor é um conceito
public class propulsor implements Concept
{
   private String nomePropulsor;
   private String localizacaoPropulsor;
   private forcaEmpuxo forcaEmpuxoPropulsor;
   public String getNomePropulsor()
   {
      return nomePropulsor;
   }
   public void setNomePropulsor(String N)
   {
      nomePropulsor = N;
   }
   public String getLocPropulsor()
   {
      return localizacaoPropulsor;
   }
   public void setLocPropulsor(String L)
   {
      localizacaoPropulsor = L;
   }
   public forcaEmpuxo getForcaEmpuxoPropulsor()
   {
      return forcaEmpuxoPropulsor;
   }
   public void setForcaEmpuxoPropulsor(forcaEmpuxo F)
   {
      forcaEmpuxoPropulsor = F;
   }
}
```

### 9.2.7 CLASSE “ALTERABALONETE”

```
import jade.content.AgentAction;
public class alteraBalonete implements AgentAction
{
   private balonete baloneteAAlterar;
   public balonete getBalonete()
   {
      return baloneteAAlterar;
   }
   public void setBalonete(balonete BAA)
   {
      baloneteAAlterar = BAA;
   }
}
```

### 9.2.8 CLASSE “ALTERALEME”

```
import jade.content.AgentAction;
public class alteraLeme implements AgentAction
{
   private leme lemeAAlterar;
   public leme getLeme()
   {
      return lemeAAlterar;
   }
   public void setLeme(leme LAA)
   {
      lemeAAlterar = LAA;
   }
}
```

### 9.2.9 CLASSE “ALTERAPROPULSOR”

```
import jade.content.AgentAction;
public class alteraPropulsor implements AgentAction
{
   private propulsor propulsorAAlterar;
   public propulsor getPropulsor()
   {
      return propulsorAAlterar;
   }
   public void setPropulsor(propulsor PAA)
   {
      propulsorAAlterar = PAA;
   }
}
```

### 9.2.10 CLASSE “JANELANAVEGADORDANT”

```
import java.awt.*;
import java.awt.event.*;
import java.text.NumberFormat;
import javax.swing.*;
import javax.swing.JFormattedTextField;
/**
*
* Esta classe permite o uso de interface gáfica de usuário.
* Apresenta uma janela por intermédio da qual o detentor do agente
* NavegadorDANT informa os valores das variáveis de estado dos atuadores.
*
*/
public class JanelaNavegadorDANT extends JFrame
{
   private NavegadorDANT myAgent;
```

```
private final String atuador[]={"Balonete", "Leme","Propulsor"};
private final String tipoPropulsor[]={"bombordo","estibordo","popa"};
private final String tipoLeme[]={"direcao", "profundidade"};
private final String tipoBalonete[]={"A","B","C","D"};

private JComboBox campoAtuador;
private JComboBox campoTipoPropulsor;
private JComboBox campoTipoLeme;
private JComboBox campoTipoBalonete;

private double compnteI = 0.0;
private double compnteJ = 0.0;
private double compnteK = 0.0;
private double pressao  = 0.0;

private JFormattedTextField campoComponenteI;
private JFormattedTextField campoComponenteJ;
private JFormattedTextField campoComponenteK;
private JFormattedTextField campoPressao;

private NumberFormat formatoCompnteI;
private NumberFormat formatoCompnteJ;
private NumberFormat formatoCompnteK;
private NumberFormat formatoCampoPressao;

JanelaNavegadorDANT(NavegadorDANT c)
{
   super(c.getLocalName());
   myAgent = c;
   setUpFormats();
   JPanel p = new JPanel();
   p.setLayout(new GridLayout(0, 2));
   p.add(new JLabel(" Atuador: "));
   campoAtuador = new JComboBox(atuador);
   campoAtuador.addItemListener(
   new ItemListener()
   {
      public void itemStateChanged(ItemEvent ev)
      {
         try
         {
            String itemSelecionado;
            if (ev.getStateChange() == ItemEvent.SELECTED)
            {
               itemSelecionado = (String)
                        campoAtuador.getSelectedItem();
               if (itemSelecionado == "Balonete")
               {
                  campoTipoPropulsor.setEnabled(false);
                  campoTipoLeme.setEnabled(false);
                  campoTipoBalonete.setEnabled(true);
                  campoComponenteI.setEnabled(false);
                  campoComponenteJ.setEnabled(false);
                  campoComponenteK.setEnabled(false);
                  campoPressao.setEnabled(true);
               }
               else if (itemSelecionado == "Leme")
               {
                  campoTipoPropulsor.setEnabled(false);
                  campoTipoLeme.setEnabled(true);
                  campoTipoBalonete.setEnabled(false);
                  campoComponenteI.setEnabled(true);
                  campoComponenteJ.setEnabled(true);
                  campoComponenteK.setEnabled(true);
                  campoPressao.setEnabled(false);
               }
               else
               {
                  campoTipoPropulsor.setEnabled(true);
                  campoTipoLeme.setEnabled(false);
                  campoTipoBalonete.setEnabled(false);
                  campoComponenteI.setEnabled(true);
                  campoComponenteJ.setEnabled(true);
                  campoComponenteK.setEnabled(true);
                  campoPressao.setEnabled(false);
```

```
            }
         }
      }
      catch (Exception e)
      {
         JOptionPane.showMessageDialog(JanelaNavegadorDANT.this,
                                       "Atuador invalido. "+e.getMessage(),
                                                "Erro", JOptionPane.ERROR_MESSAGE);
      }
   }
});
campoAtuador.setMaximumRowCount(3);
campoAtuador.setSelectedIndex(0);
p.add(campoAtuador);

p.add(new JLabel(" Tipo de Propulsor: "));
campoTipoPropulsor = new JComboBox(tipoPropulsor);
campoTipoPropulsor.setMaximumRowCount(2);
campoTipoPropulsor.setSelectedIndex(0);
campoTipoPropulsor.setEnabled(false);
p.add(campoTipoPropulsor);

p.add(new JLabel(" Tipo de Leme: "));
campoTipoLeme = new JComboBox(tipoLeme);
campoTipoLeme.setMaximumRowCount(2);
campoTipoLeme.setSelectedIndex(0);
campoTipoLeme.setEnabled(false);
p.add(campoTipoLeme);

p.add(new JLabel(" Tipo de Balonete: "));
campoTipoBalonete = new JComboBox(tipoBalonete);
campoTipoBalonete.setMaximumRowCount(2);
campoTipoBalonete.setSelectedIndex(0);
p.add(campoTipoBalonete);

p.add(new JLabel(" Componente X: "));
campoComponenteI = new JFormattedTextField(formatoCompnteI);
campoComponenteI.setEnabled(false);
campoComponenteI.setValue(new Double (compnteI));
p.add(campoComponenteI);
getContentPane().add(p, BorderLayout.CENTER);

p.add(new JLabel(" Componente Y: "));
campoComponenteJ = new JFormattedTextField(formatoCompnteJ);
campoComponenteJ.setEnabled(false);
campoComponenteJ.setValue(new Double (compnteJ));
p.add(campoComponenteJ);
getContentPane().add(p, BorderLayout.CENTER);

p.add(new JLabel(" Componente Z: "));
campoComponenteK = new JFormattedTextField(formatoCompnteK);
campoComponenteK.setEnabled(false);
campoComponenteK.setValue(new Double (compnteK));
p.add(campoComponenteK);
getContentPane().add(p, BorderLayout.CENTER);

p.add(new JLabel(" Pressão: "));
campoPressao = new JFormattedTextField(formatoCampoPressao);
campoPressao.setValue(new Double (pressao));
p.add(campoPressao);
getContentPane().add(p, BorderLayout.CENTER);

JButton InsertButton = new JButton(" Insere ");
InsertButton.addActionListener(
new ActionListener()
{
   public void actionPerformed(ActionEvent ev)
   {
      try
      {
         String atuador = (String) campoAtuador.getSelectedItem();
         String tipoPropulsor = "";
         String tipoLeme = "";
         String tipoBalonete = "";
         if (atuador == "Propulsor")
```

```
            {
               tipoPropulsor = (String) campoTipoPropulsor.getSelectedItem();
               compnteI = ((Number)campoComponenteI.getValue()).doubleValue();
               compnteJ = ((Number)campoComponenteJ.getValue()).doubleValue();
               compnteK = ((Number)campoComponenteK.getValue()).doubleValue();
               pressao = 0.0;
            }
            else if (atuador == "Leme")
            {
               tipoLeme = (String) campoTipoLeme.getSelectedItem();
               compnteI = ((Number)campoComponenteI.getValue()).doubleValue();
               compnteJ = ((Number)campoComponenteJ.getValue()).doubleValue();
               compnteK = ((Number)campoComponenteK.getValue()).doubleValue();
               pressao = 0.0;
            }
            else
            {
               tipoBalonete = (String) campoTipoBalonete.getSelectedItem();
               compnteI = 0;
               compnteJ = 0;
               compnteK = 0;
               pressao = ((Number)campoPressao.getValue()).doubleValue();
            }
            myAgent.updateTargetItem(atuador, tipoPropulsor, tipoLeme,
                                     tipoBalonete, compnteI, compnteJ, compnteK, pressao);
            campoAtuador.setSelectedIndex(0);
            campoTipoLeme.setSelectedIndex(0);
            campoTipoBalonete.setSelectedIndex(0);
            campoTipoPropulsor.setSelectedIndex(0);
            campoComponenteI.setValue(new Double(0.0));
            campoComponenteJ.setValue(new Double(0.0));
            campoComponenteK.setValue(new Double(0.0));
            campoPressao.setValue(new Double(0.0));
         }
         catch (Exception e)
         {
            JOptionPane.showMessageDialog(JanelaNavegadorDANT.this,
                              "Valor invalido. "+e.getMessage(),
                                             "Erro", JOptionPane.ERROR_MESSAGE);
         }
      }
   } );
   p = new JPanel();
   p.add(InsertButton);
   getContentPane().add(p, BorderLayout.SOUTH);
   // Faz o agente terminar quando o usuário fecha
   // a GUI usando o botão sobre o canto direito superior
   addWindowListener(new WindowAdapter() {
   public void windowClosing(WindowEvent e)
   {
      myAgent.doDelete();
   }
   } );
   setResizable(false);
}
public void show()
{
   pack();
   Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
   int centerX = (int)screenSize.getWidth() / 2;
   int centerY = (int)screenSize.getHeight() / 2;
   setLocation(centerX - getWidth() / 2, centerY - getHeight() / 2);
   super.show();
}
private void setUpFormats()
{
   formatoCompnteI = NumberFormat.getNumberInstance();
   formatoCompnteJ = NumberFormat.getNumberInstance();
   formatoCompnteK = NumberFormat.getNumberInstance();
   formatoCampoPressao = NumberFormat.getNumberInstance();
   formatoCompnteI.setMinimumFractionDigits(2);
   formatoCompnteJ.setMinimumFractionDigits(2);
   formatoCompnteK.setMinimumFractionDigits(2);
   formatoCampoPressao.setMinimumFractionDigits(2);
}
```

```
}
```

## 9.2.11 – CLASSE “NAVEGADOR DANT”

```
import jade.core.behaviours.*;
import jade.core.AID;
import jade.lang.acl.*;
import jade.domain.DFService;
import jade.domain.FIPAException;
import jade.domain.FIPAAgentManagement.*;
import jade.content.*;
import jade.core.Agent;
import jade.content.lang.sl.*;
import jade.content.lang.*;
//import jade.content.abs.*;
import jade.content.onto.*;
import jade.proto.AchieveREInitiator;
import jade.domain.FIPANames;
//import jade.tools.sl.*;
import java.util.Vector;
import jade.content.onto.basic.Action;
public class NavegadorDANT extends Agent
{
   //estas variáveis servem para conter o atuador a ser alterado
   private propulsor propulsorAAlterar;
   private leme lemeAAlterar;
   private balonete baloneteAAlterar;
   private alteraPropulsor altPropulsor;
   private alteraBalonete altBalonete;
   private alteraLeme altLeme;

   //Vetor com os agentes Atuadores (só se vai usar o primeiro)
   private AID[] agentesAtuadores;

   //Esta classe serve para mostrar a janela de ingresso do atuador a ser modificado
   private JanelaNavegadorDANT myGui;

   //registro da linguagem de conteúdo e da ontologia - primeira parte de duas
   private Codec codec = new SLCodec();
   private Ontology ontologia = OntoCmdoAtuadores.getInstance();

   //O content manager tem os métodos para tratar o conteúdo (content) da mensagem ACL
   private ContentManager gerenciadorConteudo;

   //ACLMessage representa mensagem do tipo ACL
   ACLMessage msg;

   //Inicializações do agente
   protected void setup()
   {
      propulsorAAlterar = new propulsor();
      lemeAAlterar = new leme();
      baloneteAAlterar = new balonete();
      propulsorAAlterar.setNomePropulsor("");
      lemeAAlterar.setNomeLeme("");
      baloneteAAlterar.setNomeBalonete("");

      altPropulsor = new alteraPropulsor();
      altBalonete = new alteraBalonete();
      altLeme = new alteraLeme();

      myGui = new JanelaNavegadorDANT(this);//cria a janela de ingresso de dados
      myGui.show();//mostra a janela de ingresso

      gerenciadorConteudo = (ContentManager) getContentManager();//cria a variável
                           //que por meio da qual o conteúdo pode ser gerido

      //registro da linguagem de conteúdo e da ontologia - segunda parte de duas
      gerenciadorConteudo.registerLanguage(codec);
      gerenciadorConteudo.registerOntology(ontologia);

      //adiciona comportamento cíclico, ativado automaticamente a cada 10 segundos
      addBehaviour(new TickerBehaviour(this, 10000)
      {
```

```
        protected void onTick()
        {
        //Observar que as coisas somente acontecem se um dos atuadores a
        //alterar for diferente de null. Isto significa que a requisição
        //só é feita quando for inserido algum propulsor a alterar
        //por intermédio da janela de ingresso
        if (!(propulsorAAlterar.getNomePropulsor() == "" &&
                              lemeAAlterar.getNomeLeme() == "" &&
                                   baloneteAAlterar.getNomeBalonete() == ""))
        {
           //Início da consulta às páginas amarelas sobre o agente
           //Atuadores registrado
           DFAgentDescription template = new DFAgentDescription();
           ServiceDescription sd = new ServiceDescription();
           sd.setType("Comanda Atuadores");//configura o tipo de serviço
                                          //a ser pesquisado
           template.addServices(sd);
           try
           {
              DFAgentDescription[] result = DFService.search(myAgent,template);
              //Obtém o agente Atuadores
              if (result != null && result.length > 0)
              {
                 agentesAtuadores = new AID[1]; //só é pego o primeiro agente
                 agentesAtuadores[0] = result[0].getName();
              }//Fim da obtenção do agenteAtuador
           }
           catch (FIPAException fe)
           {
              System.out.println("Erro na obtencao do agente Atuador.");
              fe.printStackTrace();
           }
           //Fim da consulta às páginas amarelas

           //Só faz a requisição de ação se houver agente Atuadores registrado
           if (agentesAtuadores.length > 0 && agentesAtuadores != null)
           {
              //Se houver agente Atuadores, faz a requisição
              //prepara a mensagem
              //preenche a performativa
              msg = new ACLMessage(ACLMessage.REQUEST);
              msg.setProtocol(FIPANames.InteractionProtocol.FIPA_REQUEST);
              //preenche o protocolo
              msg.setLanguage(codec.getName()); //define o codec
              msg.setOntology(ontologia.getName()); //define a ontologia
                                    //myAgent é variável protegida que aponta para o
                                    //agente que está usando o comportamento (o
                                    //behaviour)

              //adiciona o comportamento RequisitaAcao passando a ele
              //a mensagem que dará início à conversação, conversação que
              //se dará segundo o protocolo FIPA_REQUESTY
              myAgent.addBehaviour(new RequisitaAcao(myAgent, msg));//vai
           }
           else//se não houver agente Atuadores, avisa.
           {
              System.out.println("Nao ha agente Atuadores no momento");
              //Limpa o atuador a comandar. Assim, este agente fica a espera
              //de um novo atuador a alterar (comandar). *** Aqui, se poderia
              //também não limpar os atuadores a alterar. Fazendo assim, depois
              //de 10 segundos, o agente iria verificar novamente nas páginas
              //amarelas se há agente Atuadores para o serviço.
              propulsorAAlterar.setNomePropulsor("");
              lemeAAlterar.setNomeLeme("");
              baloneteAAlterar.setNomeBalonete("");
           }
        }
     }
     });
}//fim do método setup()

//A classe RequisitaAcao estende a classe AchieveREInitiator (RE = rational
//effect) que implementa diversos protocolos FIPA. Aqui o protocolo implementado
//é o protocolo FIPA-REQUEST
class RequisitaAcao extends AchieveREInitiator
```

```
{
   //Construtor da classe
   public RequisitaAcao(Agent a, ACLMessage m)
   {
      super(a, m);
   }
   //Este método tem de ser implementado com este nome e com um argumento
   //ACLMessage. Este método é invocado automaticamente pelo JADE.
   protected Vector prepareRequests(ACLMessage re)//observar que a variável
                                    //re é a mensagem msg passada em setup()
   {
   Vector mensagens = new Vector();//mensagens é o vetor que este método
                           //tem de retornar. Nele vão as mensagens que dão início às
                           //conversações
   Action acao = new Action();
   //envia uma REQUEST ao agente Agentes
   String replyStuff = "request" + System.currentTimeMillis();
   re.setConversationId("Comanda Atuadores");
   re.setReplyWith(replyStuff);
   re.addReceiver(agentesAtuadores[0]);
   //preenche a variável que será enviada em Content
   if (propulsorAAlterar.getNomePropulsor() != "")
   {
      try
      {
         altPropulsor.setPropulsor(propulsorAAlterar);
         acao.setAction(altPropulsor);
         acao.setActor(agentesAtuadores[0]);
         gerenciadorConteudo.fillContent(re, acao);//altPropulsor);
         //preenche o content da mensagem re com
         //a variável acao que tem a ação altPropulsor
         //e o agente que fará a alteração
      }
      catch(Exception e)
      {
         e.printStackTrace();
      }
      propulsorAAlterar.setNomePropulsor("");
   }
   else if (lemeAAlterar.getNomeLeme() != "")
   {
      try
      {
         altLeme.setLeme(lemeAAlterar);
         acao.setAction(altLeme);
         acao.setActor(agentesAtuadores[0]);
         gerenciadorConteudo.fillContent(re, acao);
         //preenche o content da mensagem re com
         //a variável leme a alterar mais agente executor
      }
      catch(Exception e)
      {
         e.printStackTrace();
      }
      lemeAAlterar.setNomeLeme("");
   }
   else
   {
      try
      {
         //Action acao = new Action();
         altBalonete.setBalonete(baloneteAAlterar);
         acao.setAction(altBalonete);
         acao.setActor(agentesAtuadores[0]);
         gerenciadorConteudo.fillContent(re, acao);
         //preenche o content da mensagem re com
         //a variável acao
      }
      catch(Exception e)
      {
         e.printStackTrace();
      }
      baloneteAAlterar.setNomeBalonete("");
   }
   mensagens.add(re);
```

```
      return mensagens; //retorna as mensagens
   }//Fim prepareRequests

   //este método é invocado automaticamente se a reposta do respondedor
   //for uma mensagem do tipo inform (uma das previstas no protocolo)
   protected void handleInform(ACLMessage msg)//trata
   {
      System.out.println("A acao foi executada. Informacao prestada por " +
                                                                  msg.getSender());
   }
   protected void handleFailure(ACLMessage msg)
   {
      System.out.println("A comunicacao falhou. Mensagem enviada por " +
                                                                    msg.getSender());
   }
   protected void handleNotUnderstood(ACLMessage msg)
   {
      System.out.println(msg.getSender() + "nao entendeu a solicitacao");
   }
}

public void updateTargetItem(final String atuador, final String tipoPropulsor,
          final String tipoLeme, final String tipoBalonete, final double compnteI,
                   final double compnteJ, final double compnteK, final double pressao)
{
   addBehaviour(new OneShotBehaviour()
   {
      public void action()
      {
         if (atuador == "Propulsor")
         {
            propulsorAAlterar.setNomePropulsor("Propulsor");
            propulsorAAlterar.setLocPropulsor(tipoPropulsor);
            forcaEmpuxo forcaDeEmpuxo = new forcaEmpuxo();
            forcaDeEmpuxo.setForcaEmpuxoI(compnteI);
            forcaDeEmpuxo.setForcaEmpuxoJ(compnteJ);
            forcaDeEmpuxo.setForcaEmpuxoK(compnteK);
            propulsorAAlterar.setForcaEmpuxoPropulsor(forcaDeEmpuxo);
            System.out.println("Requerendo alteracao do estado do propulsor tipo "
                                              + propulsorAAlterar.getLocPropulsor());
         }
         else if (atuador == "Leme")
         {
            lemeAAlterar.setNomeLeme("Leme");
            lemeAAlterar.setTipoLeme(tipoLeme);
            direcao direcaoLeme = new direcao();
            direcaoLeme.setDirecaoI(compnteI);
            direcaoLeme.setDirecaoJ(compnteJ);
            direcaoLeme.setDirecaoK(compnteK);
            lemeAAlterar.setDirecaoLeme(direcaoLeme);
            System.out.println("Requerendo alteracao do estado do leme tipo "
                                                    + lemeAAlterar.getTipoLeme());
         }
         else if (atuador == "Balonete")
         {
            baloneteAAlterar.setNomeBalonete("Balonete");
            baloneteAAlterar.setTipoBalonete(tipoBalonete);
            baloneteAAlterar.setPressao(pressao);
            System.out.println("Requerendo alteracao do balonete tipo "
                                             + baloneteAAlterar.getTipoBalonete());
         }
         else
         {
            //Faz o agente terminar imediatamente
            System.out.println("Nenhum item especificado");
            doDelete();
         }
      }
      });
   }

   //Operações de limpeza de agentes
   protected void takeDown()
   {
      myGui.dispose();
```

```
      //Imprime uma mensagem de encerramento
      System.out.println("Agente "+getAID().getName()+" terminando.");
   }
}
```

## 9.2.12 – CLASSE “ATUADORES”

```
import jade.core.Agent;
import jade.content.lang.sl.*;
import jade.content.lang.*;
import jade.content.onto.*;
import jade.domain.DFService;
import jade.domain.FIPAException;
import jade.domain.FIPANames;
import jade.domain.FIPAAgentManagement.DFAgentDescription;
import jade.domain.FIPAAgentManagement.ServiceDescription;
import jade.lang.acl.ACLMessage;
import jade.lang.acl.MessageTemplate;
import jade.content.*;
import jade.content.onto.basic.Action;
import java.util.*;
import jade.proto.AchieveREResponder;
import jade.domain.FIPAAgentManagement.NotUnderstoodException;
import jade.domain.FIPAAgentManagement.RefuseException;
import jade.domain.FIPAAgentManagement.FailureException;
public class Atuadores extends Agent
{
   //Criação dos objetos codec e ontologia
   private Codec codec = new SLCodec();
   private Ontology ontologia = OntoCmdoAtuadores.getInstance();

   //Variáveis como os propulsores, lemes e balonetes
   private ArrayList propulsores = new ArrayList();
   private ArrayList lemes = new ArrayList();
   private ArrayList balonetes = new ArrayList();

   private DFAgentDescription dfd = new DFAgentDescription();
   private ContentManager gerenciadorConteudo = (ContentManager) getContentManager();

   protected void setup()
   {
      //Registro da linguagem de conteúdo e da ontologia
      gerenciadorConteudo.registerLanguage(codec);
      gerenciadorConteudo.registerOntology(ontologia);
      //Registra o serviço "Comanda Atuadores" nas páginas amarelas
      dfd.setName(getAID());
      ServiceDescription sd = new ServiceDescription();
      sd.setType("Comanda Atuadores");
      sd.setName("Atuador");
      dfd.addServices(sd);
      try
      {
         DFService.register(this, dfd);
      }
      catch (FIPAException fe)
      {
         fe.printStackTrace();
      }//Fim do registro nas páginas amarelas.

      //Cria propulsores
      for (int i=1; i <= 3; i++)
      {
         propulsor p = new propulsor();
         p.setNomePropulsor("Propulsor" + i);
         if (i == 1)
            p.setLocPropulsor("popa"); //traseira
         else if (i == 2)
            p.setLocPropulsor("estibordo");//lado direito
         else
            p.setLocPropulsor("bombordo"); //lado esquerdo
         forcaEmpuxo fe = new forcaEmpuxo();
         fe.setForcaEmpuxoI((float) 0.0);
```

```
   fe.setForcaEmpuxoJ((float) 0.0);
   fe.setForcaEmpuxoK((float) 0.0);
   p.setForcaEmpuxoPropulsor(fe);
   propulsores.add(p);
}
//Cria lemes
for (int i=1; i <= 2; i++)
{
   leme l = new leme();
   l.setNomeLeme("Leme" + i);
   if (i == 1)
      l.setTipoLeme("direcao");
   else
      l.setTipoLeme("profundidade");
   direcao d = new direcao();
   d.setDirecaoI((float) 0.0);
   d.setDirecaoJ((float) 0.0);
   d.setDirecaoK((float) 0.0);
   l.setDirecaoLeme(d);
   lemes.add(l);
}
//Cria balonetes
for (int i=1; i <= 4; i++)
{
   balonete b = new balonete();
   b.setNomeBalonete("Balonete" + i);
   if (i == 1)
      b.setTipoBalonete("A");
   else if (i == 2)
      b.setTipoBalonete("B");
   else if (i == 3)
      b.setTipoBalonete("C");
   else
      b.setTipoBalonete("D");
   b.setPressao((float)0.0);
   balonetes.add(b);
}
System.out.println("Agente "+getLocalName()+" esperando por requisicoes.");
MessageTemplate template = MessageTemplate.and(
MessageTemplate.MatchProtocol(FIPANames.InteractionProtocol.FIPA_REQUEST),
MessageTemplate.MatchPerformative(ACLMessage.REQUEST) );
addBehaviour(new AchieveREResponder(this, template)
{
   protected ACLMessage prepareResponse(ACLMessage requisicao)
                  throws NotUnderstoodException, RefuseException
   {
      System.out.println("Agente "+getLocalName()+
                        ": requisicao recebida de "+requisicao.getSender().getName()
                                          +". A acao e "+requisicao.getContent());
      return null;
   }
   protected ACLMessage prepareResultNotification(ACLMessage requisicao,
                                    ACLMessage resposta) throws FailureException
   {
   if (executaComando(requisicao))
   {
      System.out.println("Agente "+getLocalName()+ ": Acao bem sucedida.");
      ACLMessage inform = requisicao.createReply();
      inform.setPerformative(ACLMessage.INFORM);
                     //preenche o performative da mensagem com INFORM
      inform.setProtocol(FIPANames.InteractionProtocol.FIPA_REQUEST);
                        //preenche o campo protocolo com FIPA-REQUEST
      inform.setLanguage(codec.getName());
                        //define o campo Codec com o nome do codec
      inform.setOntology(ontologia.getName());
                           //preenche o campo ontologia com o nome da
                           //ontologia usada

                  //preencher o content da INFORM (descritor abstrato - ver)
      return inform;
   }
   else
   {
      System.out.println("Agente "+getLocalName()+ ": Acao mal sucedida.");
      throw new FailureException("erro inesperado.");
```

```
      }
   }
} );
}
private boolean executaComando(ACLMessage requisicao)
{
   boolean executou = true;
   Concept acaoInterna;
   try
   {
      ContentElement ce = gerenciadorConteudo.extractContent(requisicao);
      //extrai o content da requisicao
     Action acao = new Action();
     if (ce instanceof Action)
     {
        acao = (Action) ce;
        acaoInterna = acao.getAction();
     }
     else
        acaoInterna = null;
     if (acaoInterna instanceof alteraPropulsor)
     //se a ação interna alteraPropulsor
     {
        alteraPropulsor ap = (alteraPropulsor) acaoInterna;
        propulsor pa = ap.getPropulsor();
        //extrai o propulsor a alterar
        propulsor p = new propulsor();
        //cria uma variável de trabalho do tipo propulsor
        for (int i=0; i < propulsores.size(); i++)
        {
           p = (propulsor) propulsores.get(i);
           //coloca em p o propulsor na posição i
           if (pa.getLocPropulsor() == p.getLocPropulsor())
           //se a localização do propulsor a alterar for a mesma
           //do propulsor na posição i...
           {
              p.setForcaEmpuxoPropulsor(pa.getForcaEmpuxoPropulsor());
                       //altera a força de empuxo do propulsor em p
                       //para o valor do propulsor a alterar
              propulsores.set(i, p);
                      //altera o propulsor na posição i (só a força de
                      //empuxo)
           }
        }
     }
     else if (acaoInterna instanceof alteraLeme)
     //se a ação interna for alteraLeme
     {
        alteraLeme al = (alteraLeme) acaoInterna;
        leme la al.getLeme();
                  //extrai o leme a alterar
        leme l = new leme();
                  //cria uma variável de trabalho do tipo leme
        for (int i=0; i < lemes.size(); i++)
        {
           l = (leme) lemes.get(i);
                  //coloca em l o leme na posição i
           if (la.getTipoLeme() == l.getTipoLeme())
           //se a localização do leme a alterar for a mesma
           //do leme na posição i...
           {
              l.setDirecaoLeme(la.getDirecaoLeme());
                    //altera a direção do leme em l
                    //para o valor do leme a alterar
              lemes.set(i, l);
                       //altera o leme na posição i (só a direção)
           }
        }
     }
     else if (acaoInterna instanceof alteraBalonete)
     {
        alteraBalonete ab = (alteraBalonete) acaoInterna;
        balonete ba = ab.getBalonete();
                  //extrai o balonete a alterar
        balonete b = new balonete();
```

```
                    //cria uma variável de trabalho do tipo leme
            for (int i=0; i < balonetes.size(); i++)
            {
               b = (balonete) balonetes.get(i);
                           //coloca em b o balonete na posição i
               if (ba.getTipoBalonete() == b.getTipoBalonete())
               //se o tipo do balonete a alterar for a mesmo
               //do balonete na posição i...
               {
                  b.setPressao(ba.getPressao());
                  //altera a pressão do balonete em b
                  //para o valor do balonete a alterar
                  balonetes.set(i, b);
                          //altera o balonete na posição i (só a pressão)
               }
            }
         }
         else
         {
            executou = false;
         }
      }
      catch (Exception e)
      {
         executou = false;
         e.printStackTrace();
      }
      return executou;
   }
   protected void takeDown()
   {
      try
      {
         DFService.deregister(this);//cancela o registro nas páginas amarelas
      }
      catch (FIPAException fe)
      {
         fe.printStackTrace();
      }
      //Imprime uma mensagem de encerramento
      System.out.println("Agente "+getAID().getName()
                                                  +" terminando.");
   }
}//fim da classe Atuadores
```

### 9.2.13 EXEMPLO DE EXECUÇÃO

De uma execução arbitrária deste subsistema, dados sobre as mensagens trocadas pelos agentes, os valores de algumas destas mensagens e as informações prestadas pelos agentes na janela de comando sobre as operações por eles executadas são mostrados adiante. Foram ativados um agente da classe Atuadores e um agente da classe Navegador DANT – há um, e somente um, agente de cada uma destas classes em cada DANT. Estes agentes e o agente DF são monitorados neste exemplo de execução.

#### 9.2.13.1 MENSAGENS TROCADAS

A FIG. 9.4 mostra as mensagens trocadas pelos agentes do subsistema, mais o DF, durante as requisições de modificação de estado de atuadores. De se notar que as

conversações se dão estritamente segundo o protocolo FIPA-Request e que são suportadas por classes JADE para tratamento de protocolo.

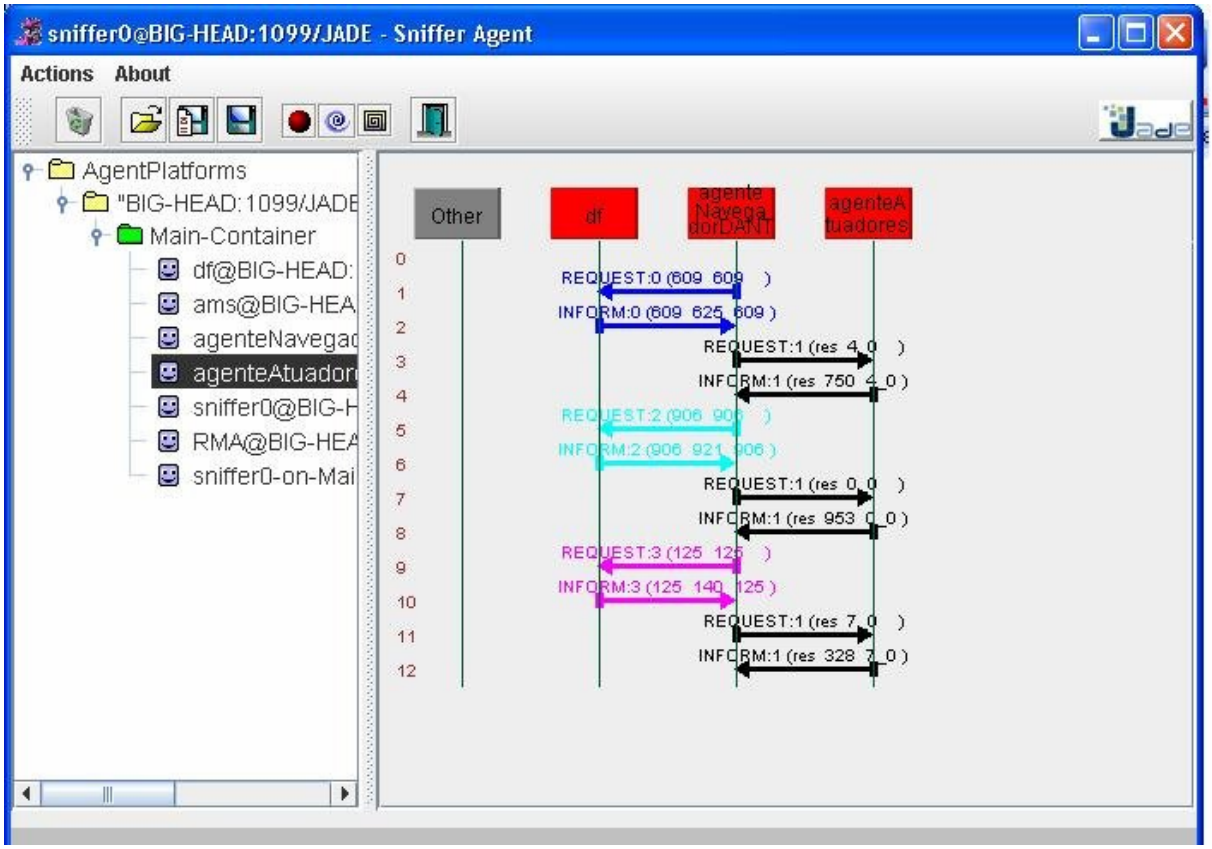


FIG 9. 4: Mensagens trocadas

Inicialmente, o agente "NavegadorDANT" consulta as páginas amarelas para determinar o endereço do agente "Atuadores". Isto é feito a cada solicitação de ação (a cada *Request*), mas poderia ser feito apenas uma única vez, antes do primeiro *Request*. Preferiu-se consultar a cada vez porque o custo é pequeno e tal providência da maior flexibilidade na manipulação do agente "Atuadores". O agente pode ser extinto e outro posto em seu lugar transparentemente para o agente "Navegador".

Após obter endereço do agente da classe "Atuadores", o agente "Navegador DANT" requer ao agente "Atuadores" a ação pretendida.

De acordo com o protocolo FIPA-Request, o agente "Atuadores" responder com o ato comunicativo *Inform* significa que a ação foi executada. Ele poderia responder também com os atos comunicativos *refuse* recusando-se a executar a ação; ou *agree*, concordando, para depois ir executá-la (neste exemplo, mensagens *agree* e *refuse* não são produzidas, por serem desnecessárias: o agente "Atuadores" tem sempre de tentar executar a ação requisitada; e

nesta versão do subsistema, o tempo de resposta é tão pequeno que a mensagem *agree* não tem sentido prático). O agente "Atuadores" poderia ainda responder com a performativa *failure*, indicando que houve falha na tentativa de execução da ação requisitada. Por fim, caso houvesse necessidade de informar o resultado da ação – e não, apenas, que a ação foi executada –, o agente "Atuadores", ainda de acordo com o protocolo, responderia *inform-result*, usando a performativa *inform*. No exemplo de execução, todas as respostas são performativas *inform* (*inform-done*).

#### 9.2.13.2 EXEMPLOS DE MENSAGENS

A seguir, são mostradas as mensagens das linhas 3, 4 e 7 da figura anterior (FIG 9.4).

##### 9.2.13.2.1 MENSAGEM DA LINHA 3

Na figura a seguir, os dados da mensagem da linha 3 são detalhados. Na parte esquerda, são mostrados o valor do ato comunicativo, *request*, neste caso; o valor do *content*, que é expandido no lado direito da figura para mostrar a totalidade do *content*; a ontologia usada etc.

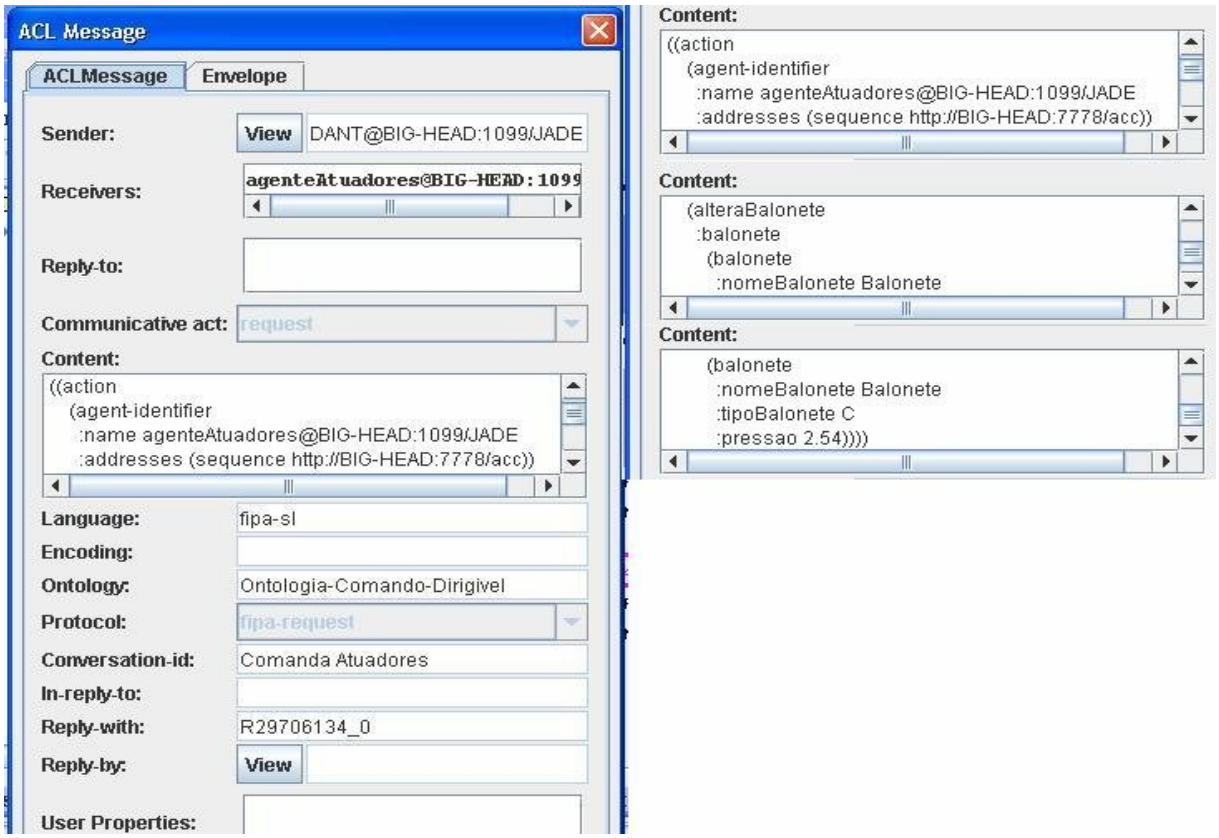


FIG 9. 5: A mensagem da linha 3

O conteúdo do *Content* expandido no lado direito da Figura 9.5 mostra o quê o ato comunicativo *request* requer que seja feito. O conteúdo expandido aparece na figura dividido em três partes. A primeira consiste nos dados do agente que se incumbe do atendimento da requisição. A segunda e terceira partes consistem nos dados daquilo que está sendo efetivamente requisitado: a ação de alterar o estado do balonete (alteraBalonete) do tipo C, mudando sua pressão interna para 2,54 atm.

O conteúdo do campo *Content* respeita integralmente as especificações da FIPA (linguagem SL). Foi composto com o suporte do JADE (classe de ontologia mais codec). Os conteúdos das mensagens, muito mais complexas neste subsistema do que no anterior, requereriam razoável esforço de programação para validação, extração de *tokens* etc., não fosse o suporte mencionado. Mesmo assim, mensagens complexas são trabalhosas para o programador: das doze classes deste subsistema, nove são classes requeridas pelo JADE para prover suporte ao tratamento do conteúdo do *Content*.

9.2.13.2.2 MENSAGEM DA LINHA 4

Esta mensagem consiste no ato comunicativo *inform*. O *content* é deixado vazio. Ela indica que a ação requerida na mensagem da linha 3 foi executada com sucesso.

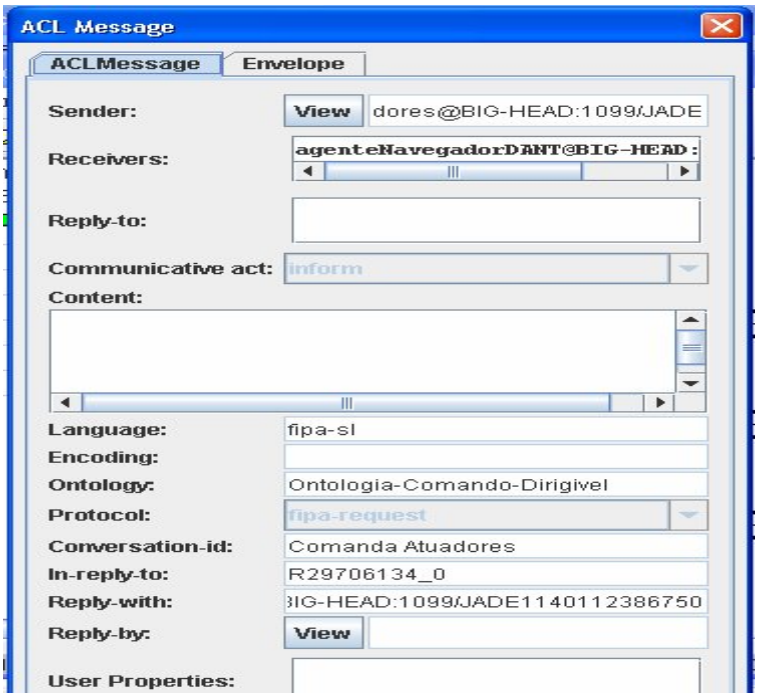


FIG 9. 6: A mensagem 4

9.2.13.2.3 MENSAGEM DA LINHA 7

Esta mensagem mostra o conteúdo do *content* no qual a ação requerida é a alteração do estado do leme. Em relação à mensagem da linha 3, também uma *request*, é de se notar a estrutura do valor do *content*, bastante diferente da estrutura do valor do *content* daquela. Ambas as estruturas são suportadas pelas nove classes mencionadas no item anterior por meio do suporte JADE.

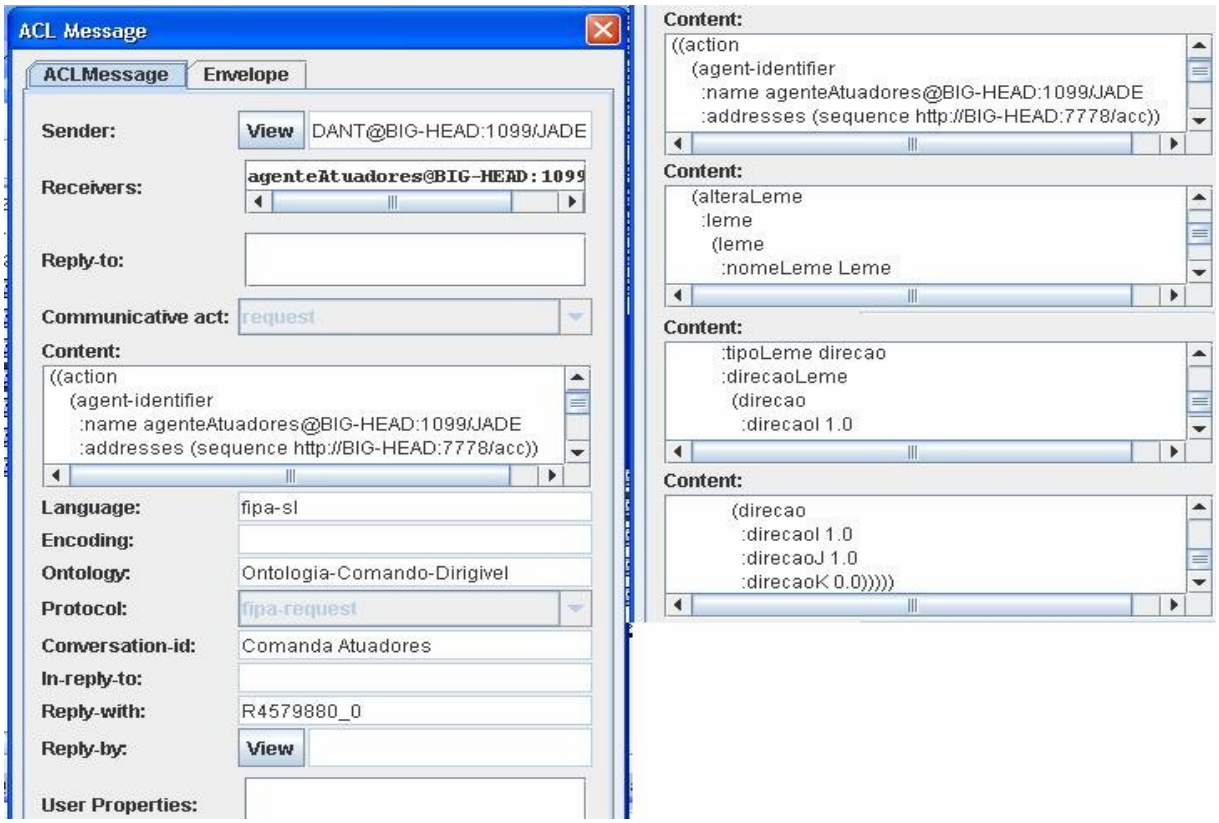


FIG 9. 7: A mensagem 7

### 9.2.13.3 MENSAGENS DOS AGENTES NA JANELA DE COMANDOS

A figura seguinte mostra as informações prestadas pelos agentes durante a execução do sistema.

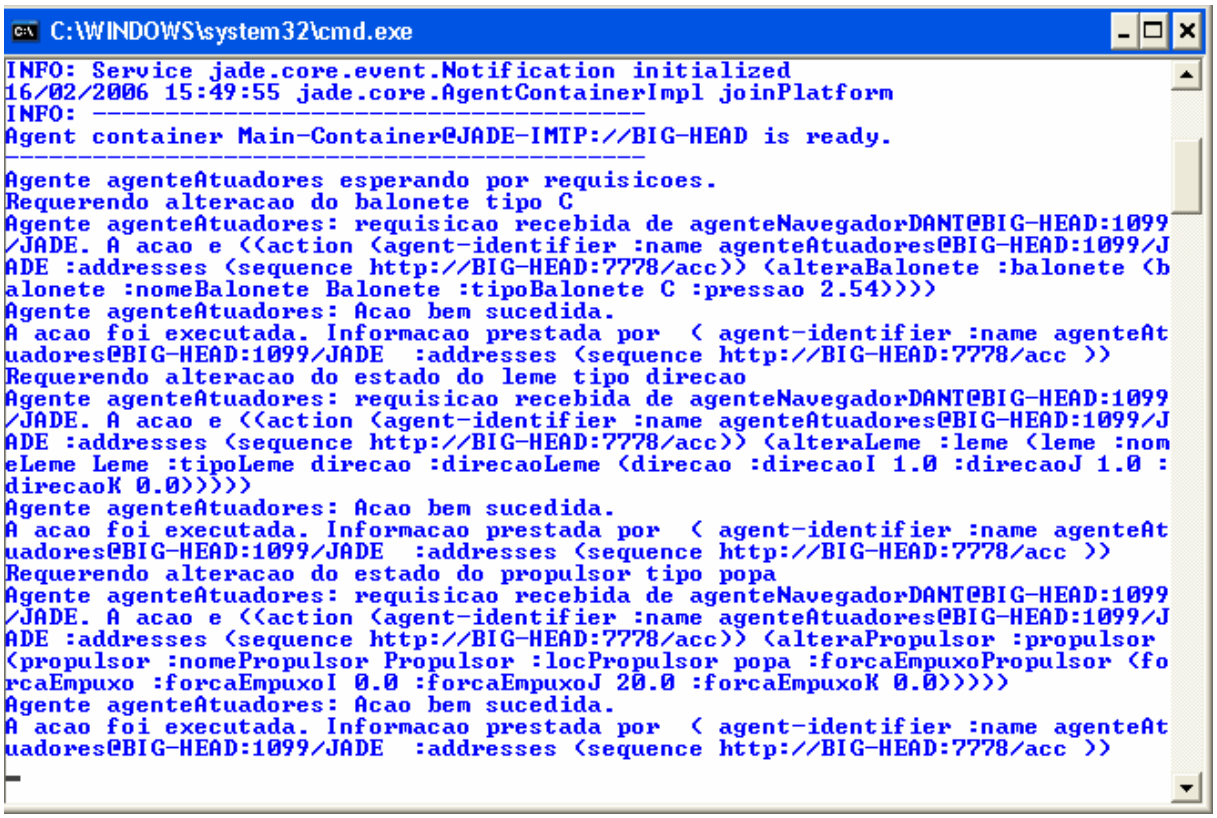


FIG 9. 8: Mensagens Produzidas pelos Agentes na Janela de Comando (com ontologia)

# 10 RESUMO, CONCLUSÕES E PERSPECTIVAS

Com o intuito de modelar o simulador, diversos conceitos de SMA foram estudados, bem como um dos casos conhecidos de projeto de DANT, o AURORA. O método MaSE de engenharia de SMA e sua ferramenta de apoio, o agentTool, foram estudados em detalhe. Um modelo do simulador foi elaborado de acordo com a MaSE, com uso amplo do agentTool. Para avaliá-lo, um ambiente de desenvolvimento de SMA no contexto do padrão FIPA foi estudado. Neste ambiente, o JADE, desenvolveu-se subsistemas simplificados a partir do modelo.

Durante o desenvolvimento dos subsistemas, sentiu-se falta (no modelo MaSE do simulador) de sub-modelos como o ontológico. A existência no método MaSE de "biblioteca" de protocolos padrões reduziria a lacuna entre o modelo MaSE e as ferramentas de desenvolvimento: o JADE, por exemplo, oferece suporte à implementação de protocolos padronizados pela FIPA. A técnica de representação das conversações da MaSE poderia ser melhorada, substituída, por exemplo, pela extensão da UML (ODELL, 2001) para representação de conversações.

O modelo do simulador não detalha a funcionalidade dos agentes. O agente "Sensores", por exemplo, tem de converter forças que atuam no dirigível – e.g., propulsão e vento, cujos respectivos valores lhe são passados respectivamente pelos agentes "Atuadores" e "Ambiente" – em valores de variáveis medidas por sensores físicos – velocidade, altitude, atitude etc. Para modelar a conversão, é necessário levantar os sensores físicos existentes e estudá-los; levantar e estudar os diversos tipos de dirigíveis; levantar e estudar seus modelos dinâmicos; escolher o dirigível, sensores e modelo dinâmico; e, a partir destes elementos, programar a funcionalidade do agente "Sensores" para que realize a conversão. O agente "Sensores" recebe ainda dados de obstáculos fixos ou móveis que ocorrem em sua esfera de visibilidade, e que lhe são passados pelo agente "Ambiente". Tais obstáculos podem ser gerados automaticamente pelo agente "Ambiente", segundo alguma distribuição de probabilidades; ou por ação do pesquisador. O agente "Sensores" tem ainda diversas outras entradas. Um outro exemplo é o agente "Navegador DANT". Este agente deve converter a dupla, posição corrente do DANT & posição pretendida do DANT, em uma sucessão de comandos aos atuadores, de molde a alcançar a posição pretendida a partir da posição corrente. Para tal, precisa também usar o modelo dinâmico do DANT. As funcionalidades de

virtualmente todos os agentes do sistema são igualmente complexas e para a sua especificação detalhada requerem o concurso de conhecimentos de diversas especialidades. Estes são alguns dos motivos pelos quais as funcionalidades dos agentes não foram tratadas aqui.

O modelo do simulador serve inicialmente para a construção de um SMA que simule a troca de mensagens entre os agentes do SDANT. Especificando-se em detalhe as funcionalidades dos agentes, o modelo serve também para o desenvolvimento de SMA que simule os demais aspectos do SDANT. A versão do simulador com as funcionalidades dos agentes implementadas facilita o desenvolvimento do SDANT. Por exemplo, diversos modelos dinâmicos de DANT podem ser programados nos agentes devidos do simulador, um modelo dinâmico para cada DANT a testar – a cada modelo dinâmico corresponde uma versão do simulador. Executando-se estas versões em condições ambientais simuladas fixas, por exemplo, pode-se comparar os DANTs – um experimento como este tem custo muito menor do que se feito com os DANTs. No caso presente, pretende-se que um SMA simule um SMR para auxiliar o desenvolvimento e validação do SMR. Algo análogo está sendo proposto para SMAs: o uso da simulação como técnica de desenvolvimento e validação de SMAs é defendido por (KLUGL, 2003).

Além do modelo, os estudos e levantamentos feitos para avaliá-lo e para avaliar a própria MaSE podem ser úteis na construção do simulador. Tais levantamentos e estudos apontam para uma plataforma de construção de SMA relativamente simples de ser operada, de baixo custo (código aberto) e aderente a um conjunto de padrões largamente aceito.

O modelo do simulador pode ser aproveitado em grande parte como modelo do próprio SDANT. De fato, o modelo do simulador foi gerado com base na funcionalidade do AURORA, um DANT. Umas poucas classes de agentes precisam ser retiradas do modelo, como a classe “Ambiente”; outras classes precisam ser adaptadas às condições reais de operação, como a classe “Atuadores”; mas estrutura geral do modelo atende o SDANT.

O código do simulador pode ser útil na construção do *software* embarcado. Diz-se do compilador GNU para Java [http://gcc.gnu.org/java/index.html] que é capaz de converter programas em Java para código de máquina. Tal conversão resolveria o principal fator impeditivo do uso de *bytecodes* no *hardware* embarcado: a lentidão de execução do código – no SDANT, tudo ocorre em *real-time* e tempos de resposta reduzidos são fundamentais. O compilador GNU permitiria que a potência da linguagem Java fosse aproveitada durante o desenvolvimento e que a eficiência do executável produzido a partir código Java permitisse seu uso em aplicação *real time*.

# 11 REFERÊNCIAS


AMOR, M., FUENTES, L. E VALLECILLO, A. Bridging the Gap Between Agent-Oriented Design and Implementation. *Agent-Oriented Software Engineering V, Fifth International Workshop AOSE,* Julho 2004.

BEELEN, M. Personal Intelligent Travel Assistant – A Distributed Approach. Tese de Mestrado, Delft University of Technology, Faculty of Electrical Engineering, Mathematics and Computer Science Knowledge Based Systems Group, Netherlands, Julho 2004.

BELLIFEMINE, F., CAIRE, G. e TRUCCO, T. Jade Administrator's Guide. JADE Board. 2005a.

BELLIFEMINE, F., CAIRE, G. e TRUCCO, T. Jade Programmer's Guide. TILab S.p.A. 2005b.

BRANDRETH, E. J. JR. Airship: An Ideal Platform for Human or Remote Sensing in the Marine Environment. OCEANS 2000 MTS/IEEE Conference and Exhibition, 2000.

CAIRE, G. e CABANILLAS, D. Jade Tutorial – Application-Defined Content Languages and Ontologies. TILab S.p.A. 2004.

CAIRE, G. Jade Tutorial - Jade Programming for Beginners. TILab S.p.A. 2003.

CHARLTON, P., CATTONI, R., POTRICH, A., MAMDANI, E. Evaluating the FIPA Standards and its Role in Achieving Cooperation in Multi-Agent Systems. Proceedings of 33rd Hawaii International Conference on System Sciences IEEE, 2000.

COCKBURN, A. Structuring Use Cases with Goals, Journal of Object-Oriented Programing, Set-Out1997 e Nov-Dez 1997.

CUESTA-MORALES, P., GÓMEZ-RODRÍGUEZ, A. E RODRÍGUEZ-MARTÍNEZ, F. J. Developing a Multi-Agent System Using MaSE and JADE. UPGRADE - European Journal for the Informatics Professional -, Vol. V, No. 4, Agosto 2004.

DEITEL, H. M. e DEITEL, P. J. JAVA como programar. 4ª Edição. Bookman. 2003.

DELOACH, S. A. Analysis and Design Using MaSE and agentTool. MAICS, Abril 2001.

DELOACH, S. A., MATSON, E. T., LI, Y. Applying Agent Oriented Software Engineering to Cooperative Robotics, 2002.

DELOACH, S. A., WOOD, M. F. E SPARKMAN C. H. Multiagent Systems Engineering, 2001b.

ELFES A. ET AL. A Semi-Autonomous Robotic Airship for Environment Monitoring Missions. IEEE, Maio 1998.

FARINELLI, A., IOCCHI, L. E NARDI, D. Multirobot Systems: A Classification Focused on Coordination. IEEE Trans. Syst., Man, Cybern. B, vol. 34, no. 5, p. 2015 2004.

FIPA. FIPA 97 Specification, Part 1, Version 2.0, Agent Management. FIPA, 1998.

FIPA. FIPA Contract Net Interaction Protocol Specification. 2002b.

FIPA. FIPA Request Interaction Protocol Specification. 2002a.

FRANKLIN, S. e GRAESSER, A. Is it an agent, or just a program?: A taxonomy for autonomous agents. ECAI'96 Third International Workshop on Agent Theories, Architectures, and Languages: Intelligent Agents III, LNAI, Vol. 1193, p. 21–36, Agosto 1996..

FRAZÃO, J. E LIMA, P. Agent-Based Software Architecture for Multi-Robot Teams. Institute for Systems and Robotics, Instituto Superior Técnico, Lisboa, Portugal, 2004.

GILLEANES T. A. G. UML – Uma Abordagem Prática, Novatec Editora, 2004.

GOMES, S. B. V. e RAMOS, J. JR. G. Airship Dynamic Modeling for Autonomous Operation. IEEE, Maio 1998.

HAYZELDEN, A. L. G. E BOURNE, R. Agent Technology for Communications Infrastructure, Capítulo 1. John Wiley and Sons, 2001.

HOLLY, A. Y. E JILL, L. D. A Taxonomy for Human-Robot Interaction. Proceedings of the AAAI Fall Symposium on Human-Robot Interaction, AAAI Technical Report FS-02-03, Falmouth, Massachusetts, p. 111-119, Novembro 2002.

HONAVAR, V. Intelligent Agents and Multi Agent Systems. Tutorial apresentado no IEEE CEC 1999.

HUFF, N., KAMEL, A. e NYGARD, K. An Agent Based Framework for Modeling UAV´s, ISCA 16th International Conference on Computer Applications in Industry and Engineering, Las Vegas, Nevada., p. 139-144 Novembro 2003.

JENNINGS, N. Coordination Techniques for Distributed Artificial Intelligence, em Foundations of Distributed Artificial Intelligence, chapter 6, John Wiley & Sons, p. 187-210, 1996.

KARIM, S., HEIZE C. E DUNN, S. Agent-Based Mission Management for a UAV. Proceedings of the Workshop on Unmanned Vehicle Systems at the International Conference on Intelligent Sensors, Sensor Networks and Information Processing ISSNIP '04, Melbourne, Australia, Dezembro 2004.

KAYSER, P. Implementação de Sistemas Multiagentes. Novembro 2001.

KLUGL, F., HERRLER, R., e OECHSLEIN, C. From Simulated to Real Environments: How to Use SeSAm for Software Development, Multiagent System Tecnologies – 1st German Conference MATES p. 13-14, 2003.

MAETA, S. M. Desenvolvimento da Infra-estrutura Embarcada do Projeto AURORA, dissertação de mestrado – UNICAMP, Junho de 2001.

MAGEDANZ, T., BREUGST, M., BUSSE, I. e COVACI, S. Integrating Mobile Agent Technology and CORBA Middleware. Agentlink Newsletter, issue 1, 1998. (http://www.agentlink.org)

MANOLA, F. Agents Standards Overview. Object Services and Consulting, Inc. Julho 1998.

MATSON, E. e, DELOACH, S. Organization Model for Cooperative and Sustaining Robotic Ecologies. Proceedings of the Robosphere, 2002.

MINSKY, M. Models, Minds, Machines. Em Proc. IFIP Congress, p. 45-49, 1965.

NOY, N. F. E MCGUINNESS, D. L. Ontology development 101: a guide to creating your first ontology. Stanford University. Março 2001.

ODELL, J., PARUNAK, V. D. e BAUER, B. Representeing agent interaction protocols in UML. Agent-Oriented Software Engineering. p. 121-140, 2001.

RAMOS, J. JR. G., MAETA, S. M., MIRISOLA, L. G. B,, BERGERMAN, M., BUENO, S. S., PAVANI, G. S., E BRUCIAPAGLIA, A. A Software Environment for an Autonomous Unmanned Airship. IEEE, Setembro 1999.

REIS, L. P. Coordenação em sistemas multi-agente: aplicações na gestão universitária e futebol robótico, Capítulos 3 e 5. Tese de Doutoramento, Faculdade de Engenharia de Universidade do Porto – FEUP –, Julho 2003.

SOMMERVILLE, I. Software Engineering, Pearson Education Limited, Essex England, 2001.

SUDEIKAT, J., BRAUBACH, L., POKAHR, A. E LAMERSDORF, W. Evaluation of Agent-Oriented Software Methodologies - examination of the gap between modeling and platform. Agent-Oriented Software Engineering V, Fifth International Workshop AOSE, Julho 2004.

WARREN R. e, DUFRENE, JR. Application of Artificial Intelligence Techniques in Uninhabited Aerial Vehicle Flight**.** The 22nd Digital Avionics Systems Conference, 2003.

WOOD, M. F. E, DELOACH, S. A. An overview of the Multiagent Systems Engineering Methodology. Agent-Oriented Software Engineering – Proceedings of the First International Workshop on Agent-Oriented Software Engineering, Junho 2000.

XIA, G., CORBETT, D. R. Cooperative Control Systems of Searching Targets Using Unmanned Blimps. Intelligent Control and Automation, Junho 2004.

# 12 APÊNDICE

A ontologia do subsistema Comando de Atuadores pode ser gerada com o auxílio do Protégé e do beangenerator. A geração da ontologia (*schemas*, na realidade) e das classes correspondentes com estas ferramentas é muito mais eficiente. A seguir, são mostradas algumas telas do Protégé com elementos da ontologia e alguns exemplos das classes geradas pelo beangenerator.

## 12.1 CLASSES DO SUBSISTEMA

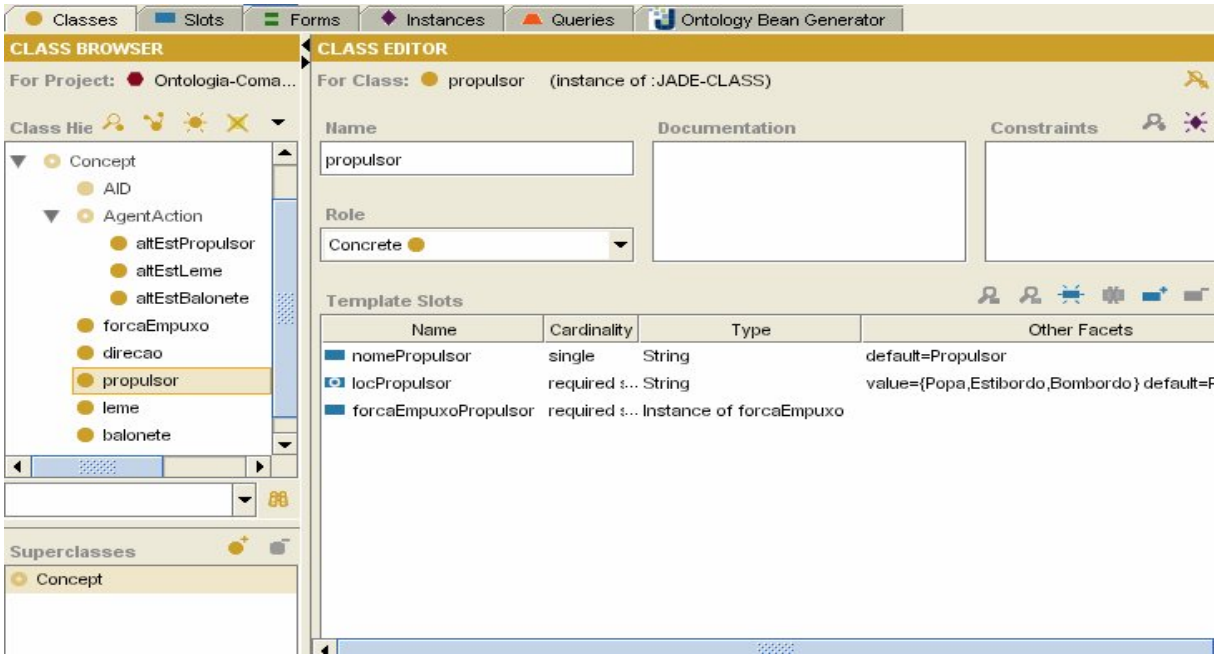


FIG 12. 1:Janela Protégé de Classes.

Todas as classes do subsistema são mostradas no lado esquerda da tela – o beangenerator acrescenta automaticamente a classe AID. No lado direito, as ranhuras da classe propulsor são mostradas.

## 12.2 RANHURAS

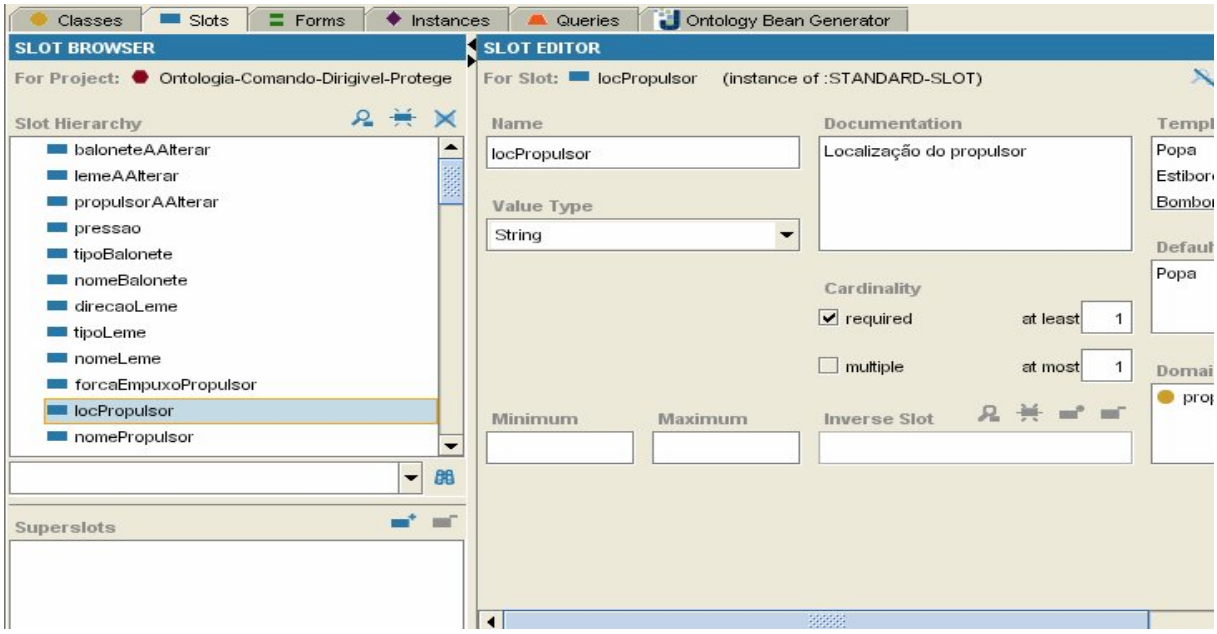


FIG 12. 2: Ranhuras das Classes

Nesta tela, as ranhuras das classes são mostradas. A ranhura locPropulsor (localização do propulsor) é mostra em detalhe no lado dirreito da Figura.

## 12.3 EXEMPLOS DE CLASSES GERADAS AUTOMATICAMENTE

A seguir, dois exemplos de classes geradas automaticamente pelo Protégé em combinação como o beangenerator.

### 12.3.1 Classe Ontologia

```
// file: ComandoDirigivelOntology.java generated by ontology bean generator. DO NOT
EDIT, UNLESS YOU ARE REALLY SURE WHAT YOU ARE DOING!
package comdoDirigProtege;
import jade.content.onto.*;
import jade.content.schema.*;
import jade.util.leap.HashMap;
import jade.content.lang.Codec;
import jade.core.CaseInsensitiveString;
/** file: ComandoDirigivelOntology.java
* @author ontology bean generator
* @version 2006/02/19, 18:37:00
*/
public class ComandoDirigivelOntology extends jade.content.onto.Ontology implements
                                                    ProtegeTools.ProtegeOntology {
   /**
   * These hashmap store a mapping from jade names to either protege names of
   SlotHolder
   * containing the protege names. And vice versa
```

```
*/
private HashMap jadeToProtege;
//NAME
public static final String ONTOLOGY_NAME "ComandoDirigivel";
// The singleton instance of this ontology
private static ProtegeIntrospector introspect new ProtegeIntrospector();
private static Ontology theInstance new ComandoDirigivelOntology();
public static Ontology getInstance() {
   return theInstance;
}
// ProtegeOntology methods
public SlotHolder getSlotNameFromJADEName(SlotHolder jadeSlot) {
   return (SlotHolder) jadeToProtege.get(jadeSlot);
}
// storing the information
private void storeSlotName(String jadeName, String javaClassName, String slotName){
   jadeToProtege.put(new SlotHolder(javaClassName, jadeName), new
   SlotHolder(javaClassName, slotName));
}
// VOCABULARY
public static final String ALTESTBALONETE_BALONETEAALTERAR "baloneteAAlterar";
public static final String ALTESTBALONETE "altEstBalonete";
public static final String ALTESTPROPULSOR_PROPULSORAALTERAR "propulsorAAlterar";
public static final String ALTESTPROPULSOR "altEstPropulsor";
public static final String ALTESTLEME_LEMEAALTERAR "lemeAAlterar";
public static final String ALTESTLEME "altEstLeme";
public static final String PROPULSOR_NOMEPROPULSOR "nomePropulsor";
public static final String PROPULSOR_FORCAEMPUXOPROPULSOR "forcaEmpuxoPropulsor";
public static final String PROPULSOR_LOCPROPULSOR "locPropulsor";
public static final String PROPULSOR "propulsor";
public static final String BALONETE_NOMEBALONETE "nomeBalonete";
public static final String BALONETE_TIPOBALONETE "tipoBalonete";
public static final String BALONETE_PRESSAO "pressao";
public static final String BALONETE "balonete";
public static final String FORCAEMPUXO_FORCAEMPUXOJ "forcaEmpuxoJ";
public static final String FORCAEMPUXO_FORCAEMPUXOK "forcaEmpuxoK";
public static final String FORCAEMPUXO_FORCAEMPUXOI "forcaEmpuxoI";
public static final String FORCAEMPUXO "forcaEmpuxo";
public static final String DIRECAO_DIRECAOK "direcaoK";
public static final String DIRECAO_DIRECAOI "direcaoI";
public static final String DIRECAO_DIRECAOJ "direcaoJ";
public static final String DIRECAO "direcao";
public static final String LEME_DIRECAOLEME "direcaoLeme";
public static final String LEME_NOMELEME "nomeLeme";
public static final String LEME_TIPOLEME "tipoLeme";
public static final String LEME "leme";
/**
* Constructor
*/
private ComandoDirigivelOntology(){
super(ONTOLOGY_NAME, BasicOntology.getInstance());
introspect.setOntology(this);
jadeToProtege new HashMap();
try {
   // adding Concept(s)
   ConceptSchema lemeSchema new ConceptSchema(LEME);
   add(lemeSchema, comdoDirigProtege.Leme.class);
   ConceptSchema direcaoSchema new ConceptSchema(DIRECAO);
   add(direcaoSchema, comdoDirigProtege.Direcao.class);
   ConceptSchema forcaEmpuxoSchema new ConceptSchema(FORCAEMPUXO);
   add(forcaEmpuxoSchema, comdoDirigProtege.ForcaEmpuxo.class);
   ConceptSchema baloneteSchema new ConceptSchema(BALONETE);
   add(baloneteSchema, comdoDirigProtege.Balonete.class);
   ConceptSchema propulsorSchema new ConceptSchema(PROPULSOR);
   add(propulsorSchema, comdoDirigProtege.Propulsor.class);
   // adding AgentAction(s)
   AgentActionSchema altEstLemeSchema new AgentActionSchema(ALTESTLEME);
   add(altEstLemeSchema, comdoDirigProtege.AltEstLeme.class);
   AgentActionSchema altEstPropulsorSchema new AgentActionSchema(ALTESTPROPULSOR);
   add(altEstPropulsorSchema, comdoDirigProtege.AltEstPropulsor.class);
   AgentActionSchema altEstBaloneteSchema new AgentActionSchema(ALTESTBALONETE);
   add(altEstBaloneteSchema, comdoDirigProtege.AltEstBalonete.class);
   // adding AID(s)
   // adding Predicate(s)
   // adding fields
```

```
        lemeSchema.add(LEME_TIPOLEME, (TermSchema)getSchema(BasicOntology.STRING),
                                      ObjectSchema.MANDATORY);
        lemeSchema.add(LEME_NOMELEME, (TermSchema)getSchema(BasicOntology.STRING),
                                      ObjectSchema.OPTIONAL);
        lemeSchema.add(LEME_DIRECAOLEME, direcaoSchema, ObjectSchema.MANDATORY);
        direcaoSchema.add(DIRECAO_DIRECAOJ, (TermSchema)getSchema(BasicOntology.FLOAT),
                                                    ObjectSchema.MANDATORY);
        direcaoSchema.add(DIRECAO_DIRECAOI, (TermSchema)getSchema(BasicOntology.FLOAT),
                                                ObjectSchema.MANDATORY);
        direcaoSchema.add(DIRECAO_DIRECAOK, (TermSchema)getSchema(BasicOntology.FLOAT),
                            ObjectSchema.MANDATORY);
        forcaEmpuxoSchema.add(FORCAEMPUXO_FORCAEMPUXOI,
                        (TermSchema)getSchema(BasicOntology.FLOAT), ObjectSchema.MANDATORY);
        forcaEmpuxoSchema.add(FORCAEMPUXO_FORCAEMPUXOK,
                        (TermSchema)getSchema(BasicOntology.FLOAT), ObjectSchema.MANDATORY);
        forcaEmpuxoSchema.add(FORCAEMPUXO_FORCAEMPUXOJ,
                        (TermSchema)getSchema(BasicOntology.FLOAT), ObjectSchema.MANDATORY);
        baloneteSchema.add(BALONETE_PRESSAO, (TermSchema)getSchema(BasicOntology.FLOAT),
                                            ObjectSchema.MANDATORY);
        baloneteSchema.add(BALONETE_TIPOBALONETE,
                    (TermSchema)getSchema(BasicOntology.STRING), ObjectSchema.MANDATORY);
        baloneteSchema.add(BALONETE_NOMEBALONETE,
                      (TermSchema)getSchema(BasicOntology.STRING), ObjectSchema.OPTIONAL);
        propulsorSchema.add(PROPULSOR_LOCPROPULSOR,
                    (TermSchema)getSchema(BasicOntology.STRING), ObjectSchema.MANDATORY);
        propulsorSchema.add(PROPULSOR_FORCAEMPUXOPROPULSOR, forcaEmpuxoSchema,
                                                            ObjectSchema.MANDATORY);
        propulsorSchema.add(PROPULSOR_NOMEPROPULSOR,
                      (TermSchema)getSchema(BasicOntology.STRING), ObjectSchema.OPTIONAL);
        altEstLemeSchema.add(ALTESTLEME_LEMEAALTERAR, lemeSchema, ObjectSchema.MANDATORY);
                altEstPropulsorSchema.add(ALTESTPROPULSOR_PROPULSORAALTERAR, propulsorSchema,
                                                        ObjectSchema.MANDATORY);
        altEstBaloneteSchema.add(ALTESTBALONETE_BALONETEAALTERAR, baloneteSchema,
                                                    ObjectSchema.MANDATORY);
        // adding name mappings
        storeSlotName("tipoLeme", "comdoDirigProtege.Leme", "tipoLeme");
        storeSlotName("nomeLeme", "comdoDirigProtege.Leme", "nomeLeme");
        storeSlotName("direcaoLeme", "comdoDirigProtege.Leme", "direcaoLeme");
        storeSlotName("direcaoJ", "comdoDirigProtege.Direcao", "direcaoJ");
        storeSlotName("direcaoI", "comdoDirigProtege.Direcao", "direcaoI");
        storeSlotName("direcaoK", "comdoDirigProtege.Direcao", "direcaoK");
        storeSlotName("forcaEmpuxoI", "comdoDirigProtege.ForcaEmpuxo", "forcaEmpuxoI");
        storeSlotName("forcaEmpuxoK", "comdoDirigProtege.ForcaEmpuxo", "forcaEmpuxoK");
        storeSlotName("forcaEmpuxoJ", "comdoDirigProtege.ForcaEmpuxo", "forcaEmpuxoJ");
        storeSlotName("pressao", "comdoDirigProtege.Balonete", "pressao");
        storeSlotName("tipoBalonete", "comdoDirigProtege.Balonete", "tipoBalonete");
        storeSlotName("nomeBalonete", "comdoDirigProtege.Balonete", "nomeBalonete");
        storeSlotName("locPropulsor", "comdoDirigProtege.Propulsor", "locPropulsor");
        storeSlotName("forcaEmpuxoPropulsor", "comdoDirigProtege.Propulsor",
                                                    "forcaEmpuxoPropulsor");
        storeSlotName("nomePropulsor", "comdoDirigProtege.Propulsor", "nomePropulsor");
        storeSlotName("lemeAAlterar", "comdoDirigProtege.AltEstLeme", "lemeAAlterar");
        storeSlotName("propulsorAAlterar", "comdoDirigProtege.AltEstPropulsor",
                                                        "propulsorAAlterar");
        storeSlotName("baloneteAAlterar", "comdoDirigProtege.AltEstBalonete",
                                                         "baloneteAAlterar");
        // adding inheritance
    }catch (java.lang.Exception e) {e.printStackTrace();}
    }
}
```

### 12.3.2 Classe Altera Estado do Propulsor

```
package comdoDirigProtege;
import java.io.Serializable;
    /**
    * Protege name: altEstPropulsor
    * @author ontology bean generator
    * @version 2006/02/19, 18:37:00
    */
public class AltEstPropulsor implements AgentAction, Serializable {
```

```
// bean stuff
   protected PropertyChangeSupport pcs new PropertyChangeSupport(this);
   public void addPropertyChangeListener(PropertyChangeListener pcl) {
      pcs.addPropertyChangeListener(pcl);
   }
   public void removePropertyChangeListener(PropertyChangeListener pcl) {
      pcs.removePropertyChangeListener(pcl);
   }
   /**
     * Protege name: propulsorAAlterar
   */
   private Propulsor propulsorAAlterar;
   public void setPropulsorAAlterar(Propulsor value) {
      pcs.firePropertyChange("propulsorAAlterar", (this.propulsorAAlterar==null?new
                                 Propulsor():this.propulsorAAlterar), value);
      this.propulsorAAlterar value;
   }
   public Propulsor getPropulsorAAlterar() {
      return this.propulsorAAlterar;
   }
}
```

# Livros Grátis

( http://www.livrosgratis.com.br )

Milhares de Livros para Download:

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo