Universidade Estadual de Campinas
Faculdade de Engenharia Elétrica e de Computação
Departamento de Engenharia de Computação e Automação Industrial

Uma Abordagem para Incorporar Mecanismos de Inteligência Artificial a Agentes Móveis

Autor: Paulo Sérgio da Silva
Orientador: Prof. Dr. Manuel de Jesus Mendes

Tese de Doutorado apresentada à Faculdade de Engenharia Elétrica e de Computação como parte dos requisitos para obtenção do título de Doutor em Engenharia Elétrica. Área de concentração: **Engenharia de Computação**.

Banca Examinadora

Prof. Dr. Edmundo Roberto Mauro Madeira	DSC/IC/Unicamp
Prof. Dr. Fernando Antonio Campos Gomide	DCA/FEEC/Unicamp
Prof. Dr. Flávio Morais de Assis Silva	DCC/UFBA
Prof. Dr. Marcelo Nicoletti Franchin	DEE/FEB/Unesp
Prof. Dr. Mario Jino	DCA/FEEC/Unicamp

Campinas, SP
Novembro/2004

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DA ÁREA DE ENGENHARIA - BAE - UNICAMP

Silva, Paulo Sérgio da
Si38a Uma abordagem para incorporar mecanismos de inteligência artificial a agentes móveis / Paulo Sérgio da Silva. -- Campinas, SP: [s.n.], 2004.

Orientadores: Manuel de Jesus Mendes.
Tese (doutorado) - Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Agentes móveis (Software). 2. Agentes inteligentes (Software). 3. Inteligência artificial. 4. Java (Linguagem de programação de computador). 5. Inferência (Lógica). 6. Framework (Programa de computador). 7. Programação orientada a objetos (Computação). I. Mendes, Manuel de Jesus. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. III. Título.

RMS - BAE

Resumo

Este trabalho contribui para a convergência de agentes móveis e agentes inteligentes através da proposta de um *framework* para incorporar técnicas de Inteligência Artificial (IA) a agentes móveis criados com os principais sistemas de mobilidade contemporâneos baseados em Java. Os requisitos a serem satisfeitos pelo *framework* são identificados e sua arquitetura é estabelecida. Um dos principais mecanismos de IA, o mecanismo de inferência com encadeamento progressivo baseado em regras, é implementado de acordo com o *framework* sugerido. Uma metodologia de desenvolvimento de software orientada a objetos e vários padrões de projeto são utilizados na concepção tanto do *framework* quanto do mecanismo de inferência. Os modelos resultantes são documentados através da Linguagem Unificada de Modelagem (UML) e as principais decisões de projeto, na forma de diretrizes a serem adotadas na criação de outros mecanismos. A proposta é avaliada através da construção de agentes móveis controlados pelo mecanismo desenvolvido e pelo levantamento do custo de transporte de seu código e dados. Os resultados mostram que o *framework* é viável e que o custo de transporte do mecanismo implementado é bem menor que o dos equivalentes encontrados na literatura.

Palavras-chave: Agentes Móveis; Agentes Inteligentes; Inteligência Artificial; Java.

Abstract

This thesis contributes for the convergence of mobile and intelligent agents by proposing a *framework* for embedding Artificial Intelligence (AI) techniques in mobile agents built with the main actual Java-based mobile agent systems. The requirements to be satisfied by the *framework* are identified and its architecture is established. One of the most essential AI engines, the rule-based forward-chaining inference engine is implemented in agreement with the suggested *framework*. An object-oriented software development methodology and several design patterns are used in the conception of both, the *framework* and the inference engine. The resulting models are documented using the Unified Modeling Language (UML) and the main design decisions, as directions to be adopted in the development of other engines. The proposal is evaluated by the construction of mobile agents controlled by the developed engine and by the estimation of the transportation cost of the engine's code and data. The results show the *framework* is feasible and that the transportation cost of the implemented engine is much lower than the cost of similar engines found in literature.

Key words: Mobile Agents; Intelligent Agents; Artificial Intelligence; Java.

Dedico este trabalho ao meu filho, Luís Fernando, cuja existência, alegria e amor incondicional têm sido decisivos em todos os momentos de minha vida.

Agradecimentos

Este trabalho não teria sido possível sem a ajuda e contribuição de várias pessoas. A todos, agradeço. Em especial, eu gostaria de expressar meus sinceros agradecimentos:

Ao meu orientador, Professor Manuel de Jesus Mendes, pela orientação, paciência, confiança e amizade, demonstradas, especialmente, nos momentos mais difíceis;

Ao Professor Leonardo Nepomuceno, do Departamento de Engenharia Elétrica da Faculdade de Engenharia de Bauru, pelo incentivo constante e por suas críticas e sugestões;

Ao Professor Hércules Araújo Feitosa, do Departamento de Matemática da Faculdade de Ciências de Bauru, por sua colaboração na área de lógica;

À Coordenadoria de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) pela bolsa de estudo que permitiu o desenvolvimento deste trabalho;

Aos demais amigos do Departamento de Engenharia Elétrica da Faculdade de Engenharia de Bauru, pelo apoio recebido durante os anos de realização deste trabalho.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

ABE	Agent Building Environment
ABLE	Agent Building and Learning Environment
ADK	Agent Development Kit
ACL	Agent Communication Language
AMI	Agente Móvel Inteligente
API	Application Programming Interface
ASCII	American Standard Code for Communication Information Interchange
ATP	Agent Transfer Protocol
BC	Base de conhecimento
BDI	Beliefs, Desires and Intentions
BNF	Backus - Naur Form
BRML	Business Rules Markup Language
CNRI	Corporation for National Research Initiatives
CORBA	Common Object Request Broker Architecture
CRIM	Le Centre de Recherche Informatique de Montréal
DAI	Distributed Artificial Intelligence
DCOM	Distributed Component Model
DLL	Dynamic Link Library
DzAI	Decentralized Artificial Intelligence
DTD	Document Type Definitions
FIPA	Foundation for Intelligent Physical Agents
GPS	General Problem Solver
HTTP	Hypertext Transfer Protocol
IA	Inteligência Artificial
IAD	Inteligência Artificial Distribuída
IBM	International Business Machines
IIOP	Internet Inter-ORB Protocol
IKV++	Informations– und Kommunicationstechnologie
JAR	Java Archive Resource
JDK	Java Development Kit
JESS	Java Expert System Shell
JNDI	Java's Naming and Directory Interface
JVM	Java Virtual Machine
KIF	Knowledge Interchange Format
KQML	Knowledge Query and Manipulation Language
LAN	Local Area Network
LISP	LISt Processing language
MAS	Multi-Agent Systems
MASIF	Mobile Agent System Interoperability Facility
NNTP	Network News Transfer Protocol
OMG	Object Management Group

OMT	Object Modeling Techniques
OO	Object-Oriented
OOSE	Object-Oriented Software Engineering
OOT	Object-Oriented Technology
ORB	Object Request Broker
PDA	Personal Digital Assistant
PROLOG	Programmation en Logique
RAISE	Reusable Agent Intelligence Software Environment
RMI	Remote Method Invocation
RMP	Recommended Models and Patterns
RUP	Rational Unified Process
SSL	Secure Socket Layer
STRIPS	Stanford Research Institute Problem Solver
UI	User Interface
UML	Unified Modeling Language
URL	Uniform Resource Locator
XML	Extended Markup Language

Capítulo 1

Introdução

1.1 Motivação

A proliferação das redes de computadores, tanto as de âmbito global, como a Internet, quanto as de âmbito local, como as intranets, tem criado muitas oportunidades para o desenvolvimento de novas aplicações, como por exemplo, ensino a distância, comércio eletrônico e comunicação multimídia. No entanto, esse movimento também está trazendo à tona novos desafios, como a organização e a recuperação de informações de forma fácil, eficiente e segura. Do ponto de vista da rede em si, surgem problemas relacionados ao grande aumento do tráfego de dados e à complexidade de seu gerenciamento. Para superar tais desafios, os pesquisadores vêm investigando uma variedade de abordagens inovadoras, dentre elas o uso de **agentes de software**.

Duas classes de agentes de software normalmente encontradas na literatura são os **agentes inteligentes** e os **agentes móveis** (WOOLDRIGE; JENNINGS, 1995). **Agentes inteligentes** são entidades tipicamente estáticas e que utilizam diversas técnicas de **Inteligência Artificial** (IA) para a realização de tarefas específicas. **Agentes móveis**, por outro lado, são programas capazes de migrar através de uma rede heterogênea de computadores, procurando e interagindo com os serviços oferecidos por seus hospedeiros, para realizarem diversas tarefas em nome de seu usuário. A abordagem procedural de programação de software é a dominante para esse tipo de agente (NWANA; NDUMU, 1999b).

Sistemas de agentes móveis utilizam-se de servidores especializados, denominados **agências**, para interpretar o comportamento do agente e permitir sua comunicação com os serviços oferecidos pelo computador hospedeiro ou por outros computadores da rede. Um computador conectado em rede pode hospedar uma ou mais agências, que, por sua vez, po-

dem servir a um ou mais agentes. O ambiente composto por todas as agências às quais os agentes têm acesso é denominado de **Ambiente Distribuído de Agentes** (ADA). Um agente móvel possui, inerentemente, autonomia de navegação, podendo, a qualquer momento, solicitar seu envio para uma outra agência. Qualquer computador conectado em rede, que possua uma agência, deve ser capaz de executar um agente móvel, sem que para isso o código executável do agente precise ser instalado com antecedência nessa máquina. Tais sistemas utilizam-se das facilidades de mobilidade de código oferecidas por certas linguagens, como por exemplo, Java (SUN, 2004), Telescript (WHITE, 1997), Tcl/Tk (OUSTERHOUT, 1995), Python (VAN ROSSUM, 2000), Obliq (CARDELLI, 1995) e Facile (THOMSEN et al., 1993), que prevêem mecanismos para transportes de classes através da rede, em tempo de execução (CETUS, 2000).

Agentes móveis oferecem uma série de vantagens potenciais em relação às abordagens tradicionais para aplicações em rede (CHESS, 1994; CHESS et al., 1995; MITSUBISHI, 1998). Por exemplo:

- **possibilitam a economia de largura de banda** através da migração da lógica da aplicação para o nó de rede onde os recursos a serem processados se encontram, evitando assim, possivelmente, a transferência de grandes quantidades de dados;
- **permitem a execução da aplicação sem a necessidade de conexões permanentes de rede**, pois podem ser transportados para um determinado nó quando a conexão estiver disponível e, então, interagir com os recursos presentes nesse nó, sem precisar de novas conexões com os sistemas que os enviaram;
- **facilitam o gerenciamento de configuração de sistemas**, pois permitem que as instalações dos componentes distribuídos dos sistemas e o controle de versões sejam realizadas remotamente;
- **simplificam a personalização de serviços oferecidos por servidores**, pois diferentes clientes podem possuir diferentes interfaces para um mesmo serviço oferecido por um servidor. Os clientes podem criar agentes móveis personalizados que, uma vez enviados ao servidor, atuam como interfaces de serviços mais simples, interagindo com seus clientes em alto nível e ocultando os detalhes de baixo nível da interação com o servidor;

- **contribuem para garantir uma melhor integridade de dados**. Neste item existem dois aspectos. Primeiro, uma vez que os dados brutos processados pelos agentes nunca deixam os servidores, eles não estarão sujeitos a toda sorte de ataques a que estariam se trafegassem pela rede. Segundo, como os aspectos de segurança devem, obrigatoriamente, ser parte integral de um sistema de agentes móveis, tanto o código dos agentes quanto os seus dados trafegarão pela rede criptografados, sem que seja necessária uma intervenção direta do usuário;
- **tornam viável o uso de dispositivos pessoais de comunicação portáteis e de baixo custo**. Esses equipamentos, geralmente, possuem capacidades limitadas de processamento e de armazenamento. Portanto, a vantagem reside na habilidade dos agentes móveis em recuperar informações e filtrá-las no servidor, retornando aos clientes somente as informações relevantes.

Embora cada uma das vantagens atribuídas aos agentes móveis possa ser obtida individualmente pelo uso de tecnologias específicas, os agentes móveis oferecem todas elas em uma única tecnologia. (CHESS, 1994).

Atualmente, existem diversas ferramentas e ambientes para a construção de aplicações baseadas em agentes móveis, como o *Aglets* da IBM (2002a), o *Grasshopper* da IKV++ (2001c), o *Voyager* da ObjectSpace (2000), o *Odyssey* da General Magic (1999) e o *Concordia* da Mitsubishi (2000a).

Esforços para padronização de interfaces, que assegurem a interoperabilidade de aplicações construídas com diferentes ferramentas, estão sendo realizados pela FIPA (1998) e pela OMG (1997). Alguns protótipos de serviços reais também estão sendo criados, como por exemplo (NWANA; NDUMU, 1999a): um serviço que integra a venda de passagem para trens e o aluguel de carros, desenvolvido na France Telecom, e um sistema para o controle de redes de telecomunicações, desenvolvido no BT Laboratories.

Uma vez que um agente móvel passa a maior parte de sua existência operacional distante de seu usuário, é importante a utilização, em seu desenvolvimento, de técnicas que ampliem sua autonomia, flexibilidade e capacidade de adaptação. Portanto, merece avaliação o uso de técnicas provenientes da IA como, por exemplo, o raciocínio lógico, o planejamento e o aprendizado de máquina, uma vez que elas podem contribuir para a melhoria das características mencionadas. Contudo, até o momento, os pesquisadores trataram, principalmente, de

assuntos tais como transporte (MENDEZ; MENDES, 1999; MENDEZ, 2000), tolerância a falhas e transações (ASSIS SILVA, 1999), comunicação (FININ; FRITZSON, 1994), cooperação (MINAR et al., 1999), segurança (RIELY; HENNESSY, 1999) e ciclo de vida (FIPA, 1998). Aspectos de integração entre as técnicas de IA e a mobilidade ainda não foram adequadamente considerados na literatura.

Mais especificamente, existe uma carência de arquiteturas e de bibliotecas de classes exclusivamente desenvolvidas para a construção de **Agentes Móveis Inteligentes** (AMI), isto é, agentes móveis que realizam suas tarefas com o auxílio das técnicas de IA. O que se encontra são ferramentas de uso geral para a construção de agentes inteligentes, como, por exemplo, o ***Java Expert System Shell*** (JESS) de Friedman-Hill (2001) e o ***Agent Building and Learning Environment*** (ABLE) da IBM (2000), que também podem ser utilizados para a criação de um AMI. Em todas essas ferramentas, a técnica de IA que mais está presente é a inferência baseada em regras. O aprendizado de máquina aparece em algumas delas e o planejamento em nenhuma. No entanto, na arquitetura e no desenvolvimento de todas, não são considerados dois requisitos extremamente importantes para o uso e aceitabilidade das técnicas provenientes da IA na construção de AMIs: primeiro, a redução do tamanho (em bytes) do código executável de suas classes e, segundo, a redução da quantidade de dados (em bytes) a serem transportados pelo agente. Por exemplo, a utilização de uma ferramenta como o JESS para incorporar inferência baseada em regras em um agente móvel acrescentaria uma quantidade extra de código executável de aproximadamente 442 kbytes e mais cerca de 19 kbytes relativos a dados. Esses valores não incluem ainda a base de conhecimento (regras e fatos) do agente. Justifica-se a preocupação com o tamanho, uma vez que estudos de desempenho dos principais sistemas de agentes móveis (SILVA et al., 1999; ISMAIL et al., 1999) demonstram que eles não são capazes de transportar um agente cuja quantidade de código e de dados exceda 1 Mbytes. Essa situação restringe, até o momento, o desenvolvimento dos agentes móveis à abordagem procedural de programação de software, o que, para muitos pesquisadores, é um retrocesso (NWANA; NDUMU, 1999b).

Uma estratégia realista para iniciar a integração entre mobilidade e inteligência é a construção de agentes móveis com poucas, mas bem estabelecidas técnicas de IA e, à medida que essas técnicas se mostrem úteis, e seus impactos sobre a mobilidade forem avaliados, acrescentar outras de forma incremental. A idéia não é abdicar da programação procedural,

mas sim acrescentar módulos “leves“ de IA que possam ser facilmente acoplados aos agentes móveis.

1.2 Proposta do Trabalho

A proposta deste trabalho é contribuir para a convergência entre agentes móveis e agentes inteligentes através do desenvolvimento de um *framework*, isto é, uma arquitetura e um conjunto colaborativo e adaptável de classes e interfaces, que permita, inicialmente, incorporar a agentes móveis, construídos com os principais sistemas de mobilidade contemporâneos, um **mecanismo de inferência baseada em regras com encadeamento progressivo**. Uma **metodologia de desenvolvimento de software orientada a objetos** (LARMAN, 1998) e vários **padrões de projeto** (GAMMA et al., 1995) deverão ser utilizados na concepção de ambos, visando um acoplamento “fraco” entre seus diversos componentes e subsistemas, de modo a reduzir a quantidade de código e de dados transportados pelos agentes e ampliar a possibilidade de extensão e reuso dos elementos identificados. A idéia é que o *framework* proposto, além de “leve”, seja suficientemente flexível para que, no futuro, outros mecanismos, como por exemplo, de inferência em lógica nebulosa e de aprendizado, também possam ser incorporados, ampliando ainda mais a autonomia e a capacidade dos agentes móveis na realização de tarefas em ambientes dinâmicos e distribuídos, sem reduzir sua mobilidade e aceitabilidade por parte de seus potenciais usuários. Portanto, as principais decisões de projeto e a experiência obtida com o desenvolvimento do mecanismo de inferência deverão ser registradas na forma de diretrizes a serem adotadas na criação dos demais.

A escolha do mecanismo de inferência baseada em regras com encadeamento progressivo como ponto de partida deve-se a diversos fatores: (1) ele representa uma das técnicas mais conhecidas e estabelecidas de IA; (2) de acordo com diversos pesquisadores (IBM, 1997), esse tipo de mecanismo é requisito primário para a maioria dos sistemas inteligentes, podendo ser usado na configuração e no controle de outros mecanismos – portanto, seu desenvolvimento potencializa futuras incorporações; (3) os agentes móveis enfrentarão, com maior freqüência, a necessidade de selecionar suas ações com base tanto no estado atual de sua base de conhecimento quanto em suas **percepções** sobre seu ambiente – justamente o tipo

de tarefa para a qual um procedimento de inferência com encadeamento progressivo é o mais adequado; (4) vários tipos de conhecimento, como por exemplo, políticas corporativas de negócios e de segurança são, por natureza, expressas em regras; (5) atualmente o paradigma simbólico ainda é a abordagem predominante em IA. Acredita-se que, provavelmente, esse cenário ainda perdurará por algum tempo (IBM, 2000). Existem várias razões para isso. Talvez a mais importante seja que muitas técnicas de IA simbólica, como os sistemas baseados em regras, levam consigo uma metodologia e uma tecnologia associada que estão se tornando familiares aos pesquisadores e engenheiros dos demais ramos da Ciência da Computação. Por exemplo, atualmente, as regras tornaram-se conceitos de primeira classe no desenvolvimento da *Web Semântica* (BERNERS-LEE et al., 2001).

1.3 Trabalhos Relacionados

Embora nenhum outro *framework* tenha sido identificado na literatura, exclusivamente desenvolvido para incorporar mecanismos de IA a agentes móveis, existem alguns trabalhos relacionados à proposta desta tese, especialmente no que se refere ao mecanismo de inferência implementado.

Desses, o mais conhecido e utilizado é o ***Java Expert System Shell*** (JESS), cujos principais componentes são um *shell* para sistemas especialistas e uma linguagem *script*, ambos desenvolvidos em Java por Friedman-Hill (2001) nos laboratórios da Sandia National, EUA. JESS suporta o desenvolvimento de sistemas especialistas baseados em regras, possibilitando sua inserção em qualquer outro código escrito em Java. A sintaxe da linguagem JESS é muito similar à sintaxe da linguagem CLIPS (***C Language Integrated Production System***) e seu mecanismo de inferência com encadeamento progressivo é baseado no algoritmo de casamento ***Rete*** (FORGY, 1982). No entanto, JESS não é capaz de interagir com procedimentos externos com o objetivo de sensoriamento, sendo possível a utilização desses somente para a atuação. Além disso, as atribuições das principais funções do mecanismo de inferência estão distribuídas por muitas classes, o que as torna fortemente interdependentes e com baixa coesão de responsabilidades. Isso faz com que a quantidade de código e de dados transportada seja maior que o necessário. Como já citado, a utilização do JESS para incorporar inferência

baseada em regras em um agente acrescenta uma quantidade extra de código de aproximadamente 442 kbytes e mais cerca de 19 kbytes relativos a dados, valores que não incluem a base de conhecimento (regras e fatos) do agente.

Bigus e Bigus (1998) desenvolveram uma biblioteca de classes em Java, denominada CIAgent (***Constructing Intelligent Agents***), na qual se encontram tanto um mecanismo de inferência com encadeamento progressivo, quanto um mecanismo com encadeamento regressivo, ambos **situados**, isto é, com a capacidade de interagir com procedimentos externos para sensoriamento e atuação. Contudo, a representação de conhecimento utilizada, na forma de regras em lógica booleana (proposicional), é muito limitada. Além disso, o mecanismo não mantém nenhum tipo de memória de trabalho para armazenar a descrição do estado atual do ambiente. Dessa forma, a inferência é realizada somente com base na percepção corrente. Por outro lado, nas classes que representam as regras e o seu conjunto, os autores incluem comportamentos (operações) claramente dependentes dos algoritmos de controle da inferência. Neste trabalho, adotou-se a postura de manter tais comportamentos exclusivamente nas classes que compõem o mecanismo de inferência, fazendo com que os objetos de representação de conhecimento sejam independentes do uso pretendido para eles. Essa estratégia aumenta a flexibilidade e o reuso de tais componentes. A utilização do CIAgent acrescenta uma quantidade extra de código de aproximadamente 47 kbytes.

O ***Agent Building and Learning Environment*** (ABLE) é uma coleção de ferramentas desenvolvidas em Java no T. J. Watson Research Center da IBM para facilitar a criação de agentes inteligentes e aplicações baseadas em agentes (IBM, 2000). Essas ferramentas oferecem um conjunto de componentes reutilizáveis, chamados de ***AbleBeans***, que implementam funcionalidades de acesso, filtragem e transformações de dados, além de mecanismos de inferência baseada em regras, usando lógicas booleana e nebulosa, e de aprendizado, baseado em redes neurais. Agentes específicos para uma determinada aplicação podem ser construídos, usando-se um ou mais desses componentes. Embora os mecanismos oferecidos sejam situados, eles apresentam as mesmas desvantagens encontradas no sistema de Bigus e Bigus (1998). A utilização do ABLE acrescenta uma quantidade extra de código de aproximadamente 496 kbytes.

O ***Agent Building Environment*** (ABE) da IBM (1997) fornece uma arquitetura e uma biblioteca de classes para o desenvolvimento de agentes inteligentes e sua utilização em

aplicações em C++ e Java. Basicamente, o ABE é constituído por seis tipos de componentes: **agentes**, **mecanismos**, **conhecimentos**, **bibliotecas**, **adaptadores** e **visualizadores**. Os **agentes** são criados pela especialização de uma classe básica (`IAgent`) que fornece as operações necessárias para que as aplicações configurem dinamicamente cada um deles, determinando quais mecanismos, bibliotecas e adaptadores cada agente utilizará. Essa interface de aplicação também permite o controle do início e do término das atividades dos agentes, bem como o monitoramento de seu estado corrente. Os **mecanismos** são responsáveis por interpretar as instruções que controlam o comportamento dos agentes. Estas informações são codificadas na forma de **conhecimento**. Cada tipo de mecanismo, por exemplo, de raciocínio e de aprendizado, pode utilizar um tipo diferente de representação de conhecimento. As **bibliotecas** permitem o armazenamento de conjuntos nomeados de conhecimentos e de seus respectivos metadados. **Adaptadores** são utilizados para integrar os agentes aos seus ambientes, por intermédio dos mecanismos. Por intermédio deles, os mecanismos podem acessar procedimentos externos, presentes no código do próprio agente ou em seu ambiente, para avaliar se determinadas condições são satisfeitas e, então, executar certas ações com base nos resultados obtidos. Os **visualizadores** determinam como ferramentas para autoria de conhecimento e interfaces gráficas de usuários podem ser incorporadas às aplicações por meio dos adaptadores, promovendo uma clara separação entre o processamento das aplicações e dos agentes e suas interações com o usuário.

A parte central do ABE é um mecanismo de inferência com encadeamento progressivo baseado em regras, denominado RAISE (***Reusable Agent Intelligence Software Environment***), desenvolvido no T. J. Watson Research Center. Esse mecanismo controla o comportamento do agente e permite a configuração e o uso dos adaptadores através de um conjunto de regras e fatos expressos em KIF (***Knowledge Interchange Format***). Um editor escrito em Java facilita a autoria desses conjuntos de conhecimento. O RAISE é o único mecanismo fornecido com o ABE. Embora a arquitetura permita que novos mecanismos sejam desenvolvidos e integrados à ferramenta, isso só é possível de forma proprietária, ou seja através de APIs de conhecimento exclusivo da IBM. Não existe na documentação nenhuma diretriz de como isso pode ser realizado pelo próprio usuário.

Vários adaptadores são fornecidos para conectar o RAISE ao mundo externo, incluindo: um adaptador HTTP para conectar os agentes à *Word-Wide-Web*, um adaptador NNTP

para conectá-los a Serviços de Notícias da Internet (*Internet News Services*), um adaptador FILE para conectá-los ao sistema de arquivos do computador hospedeiro, permitindo o monitoramento da criação, exclusão e alteração de arquivos e, finalmente, um adaptador TIMER, que possibilita que os agentes realizem tarefas baseadas em eventos temporais. Também fazem parte da ferramenta alguns exemplos de adaptadores para conexão com sistemas de e-mail e monitoração de preços de ações disponíveis em portais da Web. Esses adaptadores, escritos em Java, podem interoperar com adaptadores escritos em C++ através da infra-estrutura fornecida pela ABE.

Apesar de sua arquitetura bastante flexível, o ABE apresenta algumas características que dificultam e até mesmo impedem sua utilização com agentes móveis. Embora seja possível acrescentar novos adaptadores escritos em Java, o núcleo de suas funcionalidades é escrito em C++, o que impede sua mobilidade na maioria dos atuais sistemas de mobilidade. Além disso, mesmo que a mobilidade em C++ fosse suportada pela maioria dos sistemas de agentes móveis atuais, o tamanho do código contido nas bibliotecas de ligação dinâmica (*Dynamic Link Library* - DLL) do ABE é proibitivo para agentes móveis. Juntas, somente as bibliotecas principais, correspondem a cerca de 4,7 Mbytes. Outra dificuldade é o fato de que, para utilizarem as funcionalidades oferecidas pelos componentes ABE, os agentes devem herdar o comportamento básico oferecido por uma classe abstrata específica. No entanto, como se verá no Capítulo 3, para serem móveis os agentes também precisam herdar o comportamento móvel de uma classe base disponibilizada pelo sistema de mobilidade utilizado. Como Java não permite herança múltipla, não seria possível combinar os comportamentos inteligente e móvel dessa forma. Por outro lado, os fatos deduzidos durante um episódio de inferência não são mantidos na base de conhecimento após o seu encerramento. Essa característica é crítica para agentes móveis, que deverão poder interromper sua execução num computador hospedeiro e reiniciá-la noutro. Portanto, utilizando o ABE, as conclusões derivadas em um hospedeiro não estariam presentes na base de conhecimento do agente quando esse fosse reiniciado em outro hospedeiro.

Finalmente, o ABE também padece de dois problemas comuns às demais ferramentas citadas nesta seção: (1) as APIs dos mecanismos estão intimamente vinculadas a uma linguagem de representação específica de conhecimento, como por exemplo o KIF, e (2) todos eles possuem seus analisadores sintáticos (*parsers*) acoplados de forma inseparável de suas

funcionalidades centrais. A primeira questão é crítica para agentes móveis, pois não é possível garantir, pelo menos atualmente, que as representações de conhecimento sejam as mesmas em todos os computadores nativos e visitados. Já a segunda, pode representar um desperdício de banda, uma vez que não raro, as tarefas de análise e conversão para um formato interno de representação envolvem quantidades razoáveis de código e dados, nem sempre necessários após a inicialização do agente em seu computador nativo.

Dos trabalhos descritos acima, o único comparável, em termos de código, com a proposta desta tese é a biblioteca CIAgent que, no entanto, é mais restrito em termos de expressividade de representação, funcionalidades e flexibilidade. As classes e interfaces do *framework* proposto somam aproximadamente 9 kbytes e as do mecanismo concreto de inferência cerca de 53 kbytes.

1.4 Estrutura deste Documento

Este trabalho está estruturado da seguinte forma: no Capítulo 2 são apresentados um histórico e os conceitos básicos da tecnologia de agentes. No Capítulo 3 são analisados alguns dos principais sistemas de agentes móveis. Nessa análise, dá-se ênfase aos requisitos de implementação de um agente e à forma como novos serviços podem ser oferecidos pelas agências. No Capítulo 4 são apresentados conceitos relacionados ao raciocínio lógico em IA e ao mecanismo de inferência com encadeamento progressivo. No Capítulo 5 apresenta-se a metodologia de análise e projeto de software orientado a objeto utilizada no desenvolvimento do *framework* proposto. No Capítulo 6 são definidos os requisitos que o *framework* e o mecanismo de inferência devem satisfazer para serem adequados aos agentes móveis e é proposta uma arquitetura capaz de cumpri-los. O Capítulo 7 resume os resultados da análise e do projeto do núcleo do *framework*. No Capítulo 8 são desenvolvidos os elementos que compõem os subsistemas específicos ao mecanismo concreto avaliado. No Capítulo 9 discutem-se aspectos de implementação do mecanismo de inferência segundo o *framework* proposto e os resultados da avaliação de seu desempenho em termos do tamanho de código e de dados necessários. Finalmente, no Capítulo 10 apresentam-se as conclusões gerais do trabalho e as sugestões para trabalhos futuros.

Capítulo 2

Agentes

2.1 Introdução

Desde os tempos mais remotos, a idéia de assistentes artificiais fascina a humanidade. Criaturas de ficção científica como andróides e robôs povoam a literatura e a imaginação dos seres humanos. A busca por autômatos capazes de realizar todo tipo de trabalho corriqueiro e enfadonho foi a semente de diversas linhas de pesquisas, denominadas coletivamente por Inteligência Artificial. Atualmente, com a disseminação dos computadores e expansão de suas redes, a ênfase dessas pesquisas desviou-se do hardware para o software, "dos átomos que compõem um robô mecânico para os bits que implementam um agente de software" (NEGROPONTE, 1997).

Neste capítulo, na Seção 2.2, descreve-se como o conceito de agentes, originado no ramo da Inteligência Artificial (IA), deu origem aos atuais agentes de software. Na Seção 2.3 discute-se, de uma forma geral, a definição do termo agente; enquanto na Seção 2.4 examinam-se as características dos agentes de software atuais e suas relações com a Tecnologia de Objetos. Os agentes de software móveis são especificamente tratados no Capítulo 3.

2.2 Evolução do Conceito de Agentes

Os agentes de software, que invadiram a literatura técnica a partir de 1994, surgiram a partir de pesquisas realizadas em diversas disciplinas do conhecimento humano. Dentre as disciplinas que mais contribuíram para o estado-da-arte dos atuais agentes de software estão a

Inteligência Artificial (IA) (RUSSELL; NORVIG, 1995), a Tecnologia de Orientação a Objetos (OO) (BOOCH, 1994), a Programação Concorrente (AGHA et al., 1993), os Sistemas Distribuídos (ANANDA; SRINIVASAN, 1991) e o Projeto de Interfaces Homem - Máquina (MAES, 1994).

Inicialmente, o conceito de agente surgiu na Inteligência Artificial. De fato, o propósito primeiro da IA é a construção de artefatos inteligentes, e, para alguns autores, se estes artefatos monitoram e agem sobre algum ambiente, então eles podem ser considerados agentes (RUSSELL; NORVIG, 1995). No entanto, embora o conceito de agentes estivesse presente desde o início nos trabalhos da IA, a síntese de tais entidades não foi abordada pela maioria de seus pesquisadores até o início da década de 80. No início da IA, a metáfora foi utilizada somente como justificativa para trabalhos mais concretos, com ênfase na representação do conhecimento e nos métodos de resolução de problemas, raciocínio, planejamento, aprendizado e reconhecimento de padrões. Ou seja, os pesquisadores tomaram como modelo de inteligência o comportamento individual humano, focando seus trabalhos nos componentes isolados desse comportamento, esperando que a síntese de um agente, integrado a partir desses componentes, fosse fácil de se realizar. No início da década de 70, essa hipótese estava implícita na maioria das principais pesquisas em IA. Durante esse período, a subárea da IA mais próxima dos agentes artificiais era a de planejamento (JENNINGS et al., 1998).

Em linhas gerais, o planejamento trata da obtenção da seqüência de ações necessárias para atingir um determinado objetivo. Uma vez que, em sua essência, um agente é um sistema que age em algum ambiente, existe uma estreita relação entre as pesquisas em planejamento e o estudo dos agentes. Embora a origem das pesquisas em planejamento possam ser atribuídas a Newell e Simon (1961), criadores do ***General Problem Solver*** (GPS), é mais comum associá-la ao ***STanford Research Institute Problem Solver*** (STRIPS) (FIKES; NILSSON, 1971) e a seus descendentes, como por exemplo aqueles desenvolvidos por Chapman (1987) e Wilkins (1988).

Os primeiros algoritmos de planejamento eram baseados em técnicas de **raciocínio simbólico**, de modo que as pesquisas realizadas durante a década de 70 e início da década de 80 enfocavam a representação simbólica necessária para as ações e os algoritmos de planejamento em si, com particular interesse para a demonstração da eficiência computacional desses algoritmos. Quando aplicados a exemplos relativamente simples, os algoritmos apresentavam

desempenho razoável; contudo, logo se percebeu que tais técnicas não eram adequadas para cenários realistas. Um dos principais motivos é que a decisão sobre qual ação a realizar baseava-se numa busca irrestrita pelo espaço de todas as decisões possíveis. Como esse espaço de busca cresce exponencialmente com a complexidade da tarefa a ser solucionada, essa solução torna-se inviável se o resultado deve ser obtido em um intervalo de tempo fixo. Portanto, com tais algoritmos de planejamento não era possível construir agentes capazes de responder às mudanças em seus ambientes de modo que suas respostas fossem úteis.

Essa falha aparente dos primeiros algoritmos de planejamento em aplicações reais levou muitos pesquisadores a questionarem a viabilidade do raciocínio simbólico para a tarefa de planejamento em particular e para a IA em geral. Contudo, teve o grande mérito de chamar a atenção para os aspectos relacionados ao projeto de agentes, os quais haviam sido, de certa forma, negligenciados até então.

Nos meados da década de 80, um número crescente de pesquisadores começou a questionar às hipóteses da já tradicional IA simbólica. Certamente, o mais conhecido desses críticos é Rodney Brooks, que, em uma série de artigos, apresentou suas objeções ao modelo simbólico da IA e propôs um programa de pesquisa alternativo, conhecido por diversas denominações: **IA Comportamental**, **IA Reativa** ou **IA Situada** (BROOKS, 1986, 1991a, 1991b). Nesses artigos, Brooks enfatiza diversos aspectos do comportamento inteligente que, segundo ele, foram negligenciados pela abordagem tradicional da IA. Em particular, ele sugere que o comportamento inteligente (racional) não é um atributo de sistemas "abstratos", como os demonstradores de teoremas ou os sistemas especialistas tradicionais, mas que a inteligência é o produto das interações entre um agente e o seu ambiente. Além disso, Brooks também sugere que o comportamento inteligente emerge da interação entre diversos comportamentos mais simples.

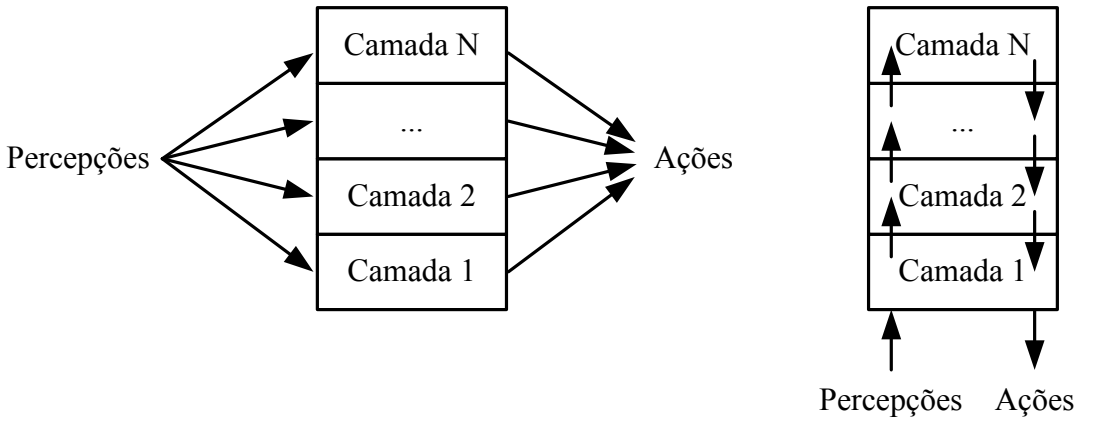
Como parte de seu programa de pesquisa, Brooks desenvolveu a **Arquitetura de Subsunção** para o controle dos agentes, a qual não empregava nenhuma representação simbólica ou técnicas de raciocínio lógico (BROOKS, 1986). Sua idéia básica é a decomposição do controle em camadas ou módulos correspondentes a comportamentos elementares. Cada módulo é um autômato de estado finito relacionado com os demais através de uma hierarquia de prioridades. Apesar de sua aparente simplicidade e de seu bom desempenho em diversas aplicações (veja, por exemplo, STEELS, 1990), existe uma série de desvantagens (veja

JENNINGS et al., 1998), dentre elas a falta de uma metodologia para construção dos agentes, a dificuldade de incorporar-se técnicas de aprendizado automático e a necessidade de uma grande quantidade de informações disponíveis no ambiente local dos agentes para que estes gerem ações aceitáveis.

Outros exemplos de arquiteturas reativas são ***Pengi*** (AGRE; CHAPMAN, 1987), ***Situated Automata*** (ROSENSCHEIN; KAELBLING, 1986) e ***Agent Network Architecture*** (MAES, 1989, 1990, 1991).

No início dos anos 90, a maioria dos pesquisadores já aceitava que as arquiteturas reativas eram adequadas somente para certos domínios de problemas. De fato, para a maioria dos problemas, nem uma arquitetura puramente deliberativa (como dos primeiros planejadores) nem uma arquitetura puramente reativa são apropriadas. Nesses casos, é necessária uma arquitetura que incorpore aspectos das duas abordagens. Como resultado dessa constatação, alguns pesquisadores começaram a investigar **arquiteturas híbridas**, que tentavam unir os melhores aspectos das abordagens deliberativa e reativa.

Tipicamente, essas arquiteturas são compostas por um certo número de camadas de software arranjadas em uma hierarquia vertical, de modo que só uma tenha acesso aos **sensores** (*sensors*) e **atuadores** (*effectors*) do agente; ou horizontal, de modo que todas tenham acesso aos sensores e atuadores do agente. A Figura 1 ilustra esses dois tipos de hierarquias.



(a) Hierarquia Horizontal (b) Hierarquia Vertical

Figura 1: Arquitetura em camadas

A maioria das implementações considera suficiente apenas três camadas. A camada de mais baixo nível na hierarquia, tipicamente reativa, toma decisões com base nos dados

brutos fornecidos por seus sensores (**nível do reflexo**), sendo implementada segundo as técnicas utilizadas por Brooks na sua arquitetura de subsunção. A camada intermediária manipula o conhecimento que o agente possui sobre seu ambiente (**nível do conhecimento**), utilizando-se, geralmente, da representação simbólica. Finalmente, a camada mais alta na hierarquia trata dos aspectos sociais do ambiente (**nível do conhecimento social**), possuindo, por exemplo, representações sobre outros agentes do ambiente. Estas três camadas interagem entre si de modo a produzirem o comportamento global do agente. Exemplos significativos desse tipo de arquitetura híbrida são ***TouringMachines*** (FERGUSON, 1992, 1995), ***InteRRaP*** (MÜLLER; PISCHEL, 1994; MÜLLER, 1997) e ***3T*** (BONASSO et al., 1996).

Uma outra abordagem para as arquiteturas híbridas inspirou-se na teoria do **Raciocínio Prático** (*Practical Reasoning*) humano; isto é, no tipo de raciocínio pragmático que nós usamos para decidir nossas ações. Tipicamente, essa é uma área de pesquisa de filósofos cognitivos, interessados em desenvolver teorias capazes de explicar o comportamento humano. Tais teorias utilizam-se da **psicologia popular**, que atribui o comportamento humano às interações entre diversas **atitudes mentais** tais como crenças, desejos, intenções, etc. As arquiteturas desenvolvidas segundo essas teorias são conhecidas por **Arquiteturas de Raciocínio Prático** (*Practical Reasoning Architectures*).

Provavelmente, a mais conhecida destas é a arquitetura ***Beliefs-Desires-Intentions*** (BDI), na qual o estado mental dos agentes é caracterizado por três atitudes: **crenças** (*beliefs)*, **desejos** (*desires*) e **intenções** (*intentions*) (BRATMAN, 1987; BRATMAN et al., 1988; RAO; GEORGEFF, 1991, 1992, 1995). Intuitivamente, as crenças correspondem às informações que o agente possui sobre seu ambiente, os desejos representam os diferentes objetivos com os quais o agente pode comprometer-se e as intenções representam os objetivos que o agente escolheu atingir. O ciclo de controle envolve, repetidamente, atualizar as crenças a partir das informações do ambiente, decidir quais opções estão disponíveis, filtrá-las para determinar as novas intenções e agir para atingi-las. O exemplo clássico de aplicação da arquitetura BDI é o *Procedural Reasoning System* (PRS) de Georgeff e Lansky (1987).

Segundo Jennings et al. (1998), a partir do início da década de 80, e portanto em paralelo com o desenvolvimento das arquiteturas reativas e híbridas, surgiram as primeiras pesquisas em IA baseadas no comportamento social humano. Essa subárea da IA, denominada **Inteligência Artificial Distribuída** (*Distributed Artificial Intelligence* - DAI) enfatiza os mo-

delos de conhecimento e as técnicas de comunicação e de raciocínio necessárias para que diversas entidades computacionais possam participar em "sociedade" na resolução de problemas (ALVARES; SICHMAN, 1997; GREEN et al., 1997).

Inicialmente, preocupou-se com os aspectos relacionados a uma sociedade de agentes capazes de interagir entre si na resolução de um **problema comum**, cuja solução estava além das capacidades individuais de cada agente. O trabalho de solução realizado por esse grupo de agentes foi denominado **Resolução Distribuída de Problemas** (GREEN et al., 1997), uma vez que, tanto os agentes quanto os dados por eles manipulados, podiam estar distribuídos lógica e geograficamente (DEMAZEAU; MÜLLER, 1990). Do ponto de vista de concepção, o desenvolvimento de tais sistemas seguia três hipóteses fundamentais:

- **hipótese da concepção centralizada**: todo o sistema, incluindo seus agentes, suas interações e sua organização, é projetado por um único projetista (ou grupo de projetistas), com o objetivo de resolver um problema específico. O objetivo do projetista é maximizar o desempenho do sistema como um todo e não dos agentes individualmente. As interações entre os agentes são guiadas por **estratégias de cooperação** baseadas no compartilhamento mútuo e irrestrito das informações;
- **hipótese dos agentes benevolentes:** os agentes jamais se questionam se devem ou não cooperar; o desejo irrestrito de cooperar é imposto por projeto;
- **hipótese do objetivo comum**: todos os agentes possuem, por definição, um objetivo comum: resolver o problema específico para o qual o sistema foi criado.

A agenda de pesquisas desse ramo inclui temas relacionados à descrição, decomposição e distribuição de tarefas, à interação e comunicação entre os agentes e a estratégias de coordenação, cooperação e resolução de conflitos. Exemplos de trabalhos nesses temas podem ser encontrados em Cammarata et al. (1983), Durfee (1988), Lesser et al., (1989) e Lesser (1991).

A partir de meados dos anos 80, um novo ramo começou a tomar forma na DAI (GREEN et al., 1997), reunindo pesquisadores interessados no comportamento de um agente **autônomo** em um **universo multiagentes** (DEMAZEAU; MÜLLER, 1990). Nesse contexto, o termo **agente** é usado para designar uma entidade que age racional e intencionalmente, segundo seus próprios objetivos e o estado atual do seu conhecimento. Por **autônomo**, entende-

se que a existência de cada agente é independente da existência dos demais. Devido à sua autonomia, cada agente é livre para executar sua tarefa individualmente ou para cooperar com os demais na realização de uma tarefa de interesse comum. O grupo de agentes pode até mesmo exibir comportamentos antagônicos. Do ponto de vista de concepção do sistema, essa linha apresenta as seguintes características (ALVARES; SICHMAN, 1997):

- não existe um problema comum específico definido por antecedência;
- os agentes são desenvolvidos por projetistas distintos e de forma a serem capazes de realizar seus processamentos independentes de uma aplicação-alvo particular;
- as interações entre os agentes são centradas, principalmente, em **estratégias de negociação**;
- o controle do sistema é implementado pelos agentes de forma totalmente descentralizada.

A agenda de pesquisas desse ramo inclui temas relacionados a estratégias genéricas de negociação que possam ser reutilizadas em diversas aplicações (veja, por exemplo, SYCARA 1988, 1990a, 1990b; ROSENSCHEIN, 1985; ROSENSCHEIN; ZLOTKIN, 1994; KRAUS, 1995; SANDHOLM; LESSER, 1995 e JENNINGS et al., 1996) e à capacidade de adaptação dos agentes face às alterações no seu ambiente (HU; WELLMAN, 1996; ZENG; SYCARA, 1997, 1998).

Devido à característica descentralizada de controle, Demazeau e Müller (1990) denominaram, inicialmente, essa abordagem **Inteligência Artificial Descentralizada**, (*Decentralized Artificial Intelligence* – DzAI) na tentativa de distingui-la da abordagem utilizada até então pela Inteligência Artificial Distribuída. No entanto, com o tempo, essa denominação caiu em desuso, sendo substituída pela expressão **Sistemas Multiagentes** (*Multi-Agent Systems* - MAS).

Mais recentemente, o termo *Sistema Multiagentes* tomou um significado mais amplo na literatura, passando a referir-se a todos os tipos de sistemas compostos por múltiplos componentes (semi-) autônomos (JENNINGS et al., 1998). Os sistemas construídos segundo as hipóteses da Resolução Distribuída de Problemas passaram a ser denominados **Sistemas Multiagentes Cooperativos**, enquanto os sistemas desenvolvidos segundo a abordagem da

Inteligência Artificial Descentralizada passaram a ser conhecidos por **Sistemas Multiagentes Competitivos**.

De acordo com Nwana (1996), a partir do início da década de 90 começou a tomar forma um movimento de pesquisa preocupado com uma classe muito mais ampla de agentes, denominados **agentes de software**, da qual os **agentes móveis** fazem parte. Até então, os pesquisadores, principalmente do ramo da Inteligência Artificial, concentravam-se no estudo dos macro-aspectos sociais e no desenvolvimento de teorias, arquiteturas e linguagens para agentes.

Porém, com essa nova onda de interesse, houve uma mudança na abordagem das pesquisas, da Inteligência Artificial para os demais ramos da Ciência da Computação, com ênfase, principalmente, em aspectos de natureza prática e comercial, como por exemplo, simplificar a complexidade do processamento distribuído e sobrepujar as limitações das atuais abordagens para a interface homem - máquina (BRADSHAW, 1997b). Os agentes passaram, então, a ser considerados como uma nova tecnologia de desenvolvimento de software.

Como conseqüência, protótipos de muitas pesquisas foram simplificados para se tornarem produtos e, como observou Foner (1993), também surgiram muitos produtos que se autodenominaram agentes ou baseados em agentes, muito embora não apresentassem nenhum aspecto que realmente os diferenciasse de um programa qualquer de software. De fato, a gama de aplicações sendo investigadas e/ou desenvolvidas é bastante ampla, variando de aplicações triviais até as moderadamente “inteligentes”.

2.3 Sobre a Definição do Termo Agente

Até o momento, não existe uma definição concisa e de consenso para o termo agente. De certa forma, a postura mercantilista adotada a partir de 1990 esvaziou o significado do termo, tornando-o um jargão extremamente difundido sobre o qual não existe concordância. Atualmente, alguns programas são denominados agentes simplesmente porque (BRADSHAW, 1997b):

- podem ser executados em uma máquina remota com data e hora marcada; ou

- são instruídos sobre suas tarefas (em geral bem simples) por meio de Linguagens *Script* ou de alto nível; ou
- abstraem ou encapsulam as diferenças na utilização de diversas fontes de informação ou serviços de processamento; ou
- agregam algum tipo primitivo de "função cognitiva"; ou
- apresentam características de inteligência distribuída; ou
- servem como mediadores entre pessoas e programas; ou
- podem migrar por "vontade própria" de um computador para outro; ou
- são vistos por seus usuários como personagens reais; ou
- são fluentes em uma linguagem de comunicação entre agentes; ou
- aos olhos de seus usuários, manifestam intencionalidade e outros aspectos de "estado mental".

Essa diversidade de enfoques e aplicações é um sinal inequívoco de que os agentes de software se firmaram como importantes elementos comerciais e de pesquisa. No entanto, o uso indiscriminado do rótulo "agente" cria uma expectativa muito elevada por parte dos potenciais usuários; expectativa essa que, uma vez não atendida, pode desgastar a área e levá-la a sofrer uma forte oposição, como aconteceu com a Inteligência Artificial nos anos 80 (JENNINGS et al., 1998).

Por isso, considera-se importante trabalhar com uma definição para o termo. Em (mais) um esforço de coleta e contribuição para a discussão do assunto, apresentam-se a seguir alguns aspectos relacionados ao tema.

De um modo geral, um **agente** pode ser uma pessoa, uma máquina, um programa de software, ou uma variedade de outras coisas. A definição básica encontrada nos dicionários é "aquele ou aquilo que age". No entanto, para o desenvolvimento de sistemas baseados em agentes, essa definição é muito geral, sendo necessário apontar-se algumas características adicionais, que permitam distingui-los de outras entidades e, até mesmo, julgar se determinada entidade pode ou não ser considerada um agente.

Apesar da divergência dos pesquisadores sobre quais características devam ser consideradas essenciais ou obrigatórias, a maioria concorda que, para serem úteis, os agentes precisam possuir pelo menos as seguintes: **autonomia**, **interatividade** e **adaptabilidade**.

2.3.1 Autonomia

Autonomia é a capacidade de iniciar e controlar sua execução sem intervenção ou solicitação externa direta. Isto é, o agente deve possui controle exclusivo sobre seu estado interno e sobre seu comportamento. A autonomia é mais bem caracterizada por níveis ou graduações do que simplesmente por "presente" ou "ausente". Existem dois aspectos envolvidos nessa definição. O primeiro deles, e o mais evidente, é determinado pela estrutura interna do agente e está relacionado com o grau de controle que ele exerce sobre suas próprias atividades. Esse aspecto é denominado **autonomia operacional** (OMG, 2000) e apresenta três graduações possíveis: **passivo**, **reativo** e **pró-ativo**.

A **passividade** corresponde à ausência da autonomia operacional. Uma entidade é passiva se executa suas atividades somente quando solicitada. A **reatividade** corresponde à habilidade de iniciar a execução de alguma atividade não apenas quando explicitamente solicitado, mas também quando um determinado evento observável ocorrer no ambiente. A **pró-atividade** corresponde à habilidade de buscar no ambiente, por iniciativa própria, por eventos e outras informações, com o objetivo de determinar-se como agir.

O segundo aspecto envolvido na definição de autonomia é mais sutil, mas extremamente importante na distinção dos agentes de outras entidades. Esse aspecto é denominado **autonomia de comportamento** (OMG, 2000). Do ponto de vista de um observador externo, um agente pode apresentar um comportamento "imprevisível", mesmo quando todas as condições iniciais forem perfeitamente conhecidas. Uma vez que o agente possua controle exclusivo sobre seu estado e sobre seu comportamento, ele não obedece "cegamente" a comandos, mas pode pedir esclarecimentos, modificar ou até mesmo recusar determinadas solicitações. Isto é, ele tem a capacidade de dizer "não".

2.3.2 Interatividade

Interatividade é a capacidade de interagir com o ambiente, com outros agentes e com seu usuário. Também pode ser expressa em graduações. A forma mais básica de interação é a troca explícita de mensagens entre entidades, denominada **comunicação direta**. Uma forma um pouco mais complexa é a **comunicação indireta**, através da percepção de eventos obser-

váveis induzidos no ambiente. Um nível ainda mais complexo de interação pode ser encontrado em sistemas multiagentes, onde os agentes podem se engajar em interações múltiplas e simultâneas, como em uma sociedade. Neste caso, os agentes devem ser capazes de **coordenar** suas atividades através de estratégias de **cooperação** e/ou **competição**.

Embora seja possível imaginar-se um agente que não interaja com nada externo, a utilidade de tal entidade para o desenvolvimento de sistemas baseados em agentes é questionável.

2.3.3 Adaptabilidade

Um agente é considerado adaptativo se é capaz de responder a outros agentes e/ou a seu ambiente, isto é, ele deve ser capaz de reagir a estímulos gerados por essas entidades. Sem essa característica a utilidade de um agente também é questionável.

A forma de reação mais simples é a **reflexiva**, o que significa responder de uma forma direta e predeterminada a um determinado evento ou sinal do ambiente. Tipicamente, esse tipo de reação pode ser expresso na forma: **QUANDO** *evento*, **SE** *condições*, **ENTÃO** *ação*. Nenhum tipo de descrição do estado atual do ambiente é mantido pelo agente.

Além dessa forma simples de reação, um agente pode utilizar **regras de inferência** para determinar sua resposta a determinado estímulo. Neste caso, diz-se que o agente pode "raciocinar". Tipicamente, esse tipo de reação mantém internamente uma descrição (parcial) do estado atual do ambiente e utiliza algum mecanismo de **encadeamento** de regras expressas na forma anterior. Embora a forma de expressar as regras seja similar, no primeiro caso, o agente reflexivo deve possuir uma regra para cada combinação {evento, condições, ação}, enquanto no segundo, o encadeamento e a manutenção da descrição do estado atual do ambiente permitem reduzir o número de regras necessárias para expressar um certo comportamento. Por exemplo, sejam as regras:

Quando *Evento1*, **se** *A* **então** *X*

Quando *Evento1*, **se** *B* **então** *Y*

Quando *Evento1*, **se** *A e B* **Então** *Z*

Quando *Evento2*, **se** *C* **então** *X*

Quando *Evento2*, **se** *D* **então** *Y*

Quando *Evento2*, **se** *C e D* **então** *Z*

Utilizando o encadeamento, estas regras podem ser simplificadas para:

Quando *Evento1*, **se** *A* **então** *X*

Quando *Evento1*, **se** *B* **então** *Y*

Quando *Evento2*, **se** *C* **então** *X*

Quando *Evento2*, **se** *D* **então** *Y*

Se X e *Y* **então** *Z*

Linguagens de representação de conhecimento e mecanismos de inferência, desenvolvidos no ramo da Inteligência Artificial, podem ser utilizados para essa tarefa.

O próximo nível de reação corresponde ao **planejamento**. Neste caso, a resposta do agente não se restringe a uma única ação, mas a uma seqüência planejada de ações. Técnicas de planejamento, desenvolvidas no ramo da Inteligência Artificial, podem ser utilizadas para essa tarefa.

A forma mais avançada de adaptação corresponde à capacidade de **aprendizado**. Neste caso, os agentes podem alterar seu comportamento com base na experiência acumulada ao longo de sua vida. Esse tipo de agente faz uso de técnicas de inteligência computacional, tais como Redes Neurais e Regras Bayesianas.

Algumas fontes (OMG, 2000) já consideram um patamar mais elevado de adaptação, correspondente à **evolução**. Nesse caso, a resposta do agente é uma função dos estímulos induzidos pelo ambiente e outros agentes ao longo de gerações sucessivas de agentes! Tais entidades fariam uso, por exemplo, de algoritmos genéticos.

Uma vez que, normalmente, as técnicas utilizadas para a implementação dos diversos níveis de adaptação estão sob o guarda-chuva da Inteligência Artificial, é comum denominar os agentes de software que possuem essa característica de **agentes inteligentes**. Deste modo, neste trabalho, o termo *inteligente* será usado como a "qualidade de possuir os elementos chaves de inteligência artificial na forma de mecanismos de inferência, planejamento, aprendizado e outras técnicas tradicionais de IA e ter seu comportamento determinado pelo estado atual de sua base de conhecimento". Portanto, a noção de inteligência usada aqui para um agente é bem diferente da utilizada para seres humanos.

Alguns pesquisadores ainda impõem que, para chamá-los de inteligentes, é necessário que eles interajam com outros agentes usando algum tipo de linguagem simbólica, por

exemplo, ***Knowledge Query and Manipulation Language*** (KQML) (FININ et al. 1994) ou ***Agent Communication Language*** (ACL) (FIPA, 1998).

Portanto, segundo uma definição minimalista, um agente pode ser considerado como "uma entidade autônoma capaz de interagir com seu ambiente". Em outras palavras, ele deve ser capaz de perceber seu ambiente através de sensores e de agir sobre ele através de atuadores.

É comum enfatizar a característica predominante de um agente acrescentando um adjetivo ao termo. Deste modo, são encontrados na literatura termos como: **agentes autônomos**, **agentes colaborativos**, **agentes inteligentes**, **agentes móveis**, etc.

Outra forma bastante comum de caracterização é classificá-los de acordo com a aplicação para a qual eles foram desenvolvidos. Por exemplo, **agentes de informação**, **agentes de interface**, etc. Essa forma dá mais ênfase ao papel desempenhado pelo agente do que à abordagem utilizada em seu projeto.

A Tabela 1 apresenta algumas propriedades adicionais normalmente atribuídas aos agentes.

Tabela 1: Algumas propriedades adicionais atribuídas aos agentes

Situado	Capacidade de interagir com o ambiente, recebendo e solicitando informações deste.
Social	Capacidade de interagir com outros agentes de forma que essas interações sejam marcadas por relações sociais de amizade e de cordialidade.
Representativo	Qualidade de agir em nome de alguém ou de algo, representando seus interesses.
Racional	Capacidade de escolher suas ações com base em suas próprias metas e no estado atual de sua base de conhecimento.
Transparente	Capacidade de fornecer um relatório de suas atividades quando solicitado.
Robusto	Capacidade de lidar com erros e informações incompletas sem degradar sua execução.

2.4 Agentes de Software

Um agente de software é um tipo mais específico de agente, que pode ser definido como "uma entidade autônoma de software capaz de interagir com seu ambiente" (OMG, 2000). Em outras palavras, são agentes implementados por software que podem interagir com outras entidades, incluindo seres humanos, máquinas e outros agentes de software. O ambiente de um agente de software pode ser um único computador ou uma rede de computadores.

Embora linguagens e ferramentas de desenvolvimento possam ser criadas especificamente para dar suporte à tecnologia de agentes de software, também se pode implementá-los usando-se linguagens e ferramentas orientadas a objetos, ou quaisquer outras linguagens, ferramentas e ambientes capazes de dar suporte a entidades de software *autônomas*, *interativas* e *adaptativas*. Ferramentas específicas são preferíveis, pois tais características seriam inerentes, não sendo necessário programá-las explicitamente. Ou seja, a tecnologia de objetos pode ser utilizada para viabilizar a tecnologia de agentes, como de fato está ocorrendo, mas, no seu estado atual, ela não suporta de forma direta e completa as características de autonomia, interatividade e de adaptação dos agentes.

Sob vários aspectos, objetos e agentes são semelhantes. De fato, pode-se considerar que, conceitualmente, agentes de software são objetos (ou coleções de objetos) ou componentes de software, visto que possuem *identidade*, *comportamento, estado* e *interfaces* por intermédio das quais podem se comunicar entre si e com outras entidades.

No entanto, embora existam similaridades, eles diferem de forma significativa. Por exemplo, com relação à *autonomia operacional*, objetos são, geralmente, *passivos* pois somente executam as atividades relacionadas às suas operações quando essas operações são explicitamente invocadas por outros objetos. Se nenhuma operação for invocada, o objeto permanece em "repouso".

As linguagens de programação orientadas a objetos podem possuir mecanismos que permitam o tratamento de eventos, possibilitando que seus objetos apresentem um certo grau de *reatividade* em seu comportamento. Esses mecanismos tornam possível que a execução de uma atividade se inicie não apenas quando um método *específico* é invocado, mas também quando um determinado evento observável ocorrer no ambiente. Nesse caso, um único método genérico é invocado, passando-se, como parâmetro de entrada, uma descrição do evento

observado. A atividade executada pelo método dependerá do tipo de específico de evento, por exemplo, a abertura ou o fechamento de uma janela.

Para que uma entidade de software apresente comportamento *pró-ativo*, ela deve ser um *processo contínuo no tempo*, ou em outras palavras, deve possuir seu próprio fluxo de controle de execução (*thread*), o que não ocorre necessariamente para todo objeto, embora existam mecanismos que permitam às linguagens de programação orientadas a objetos terem acesso aos recursos de programação concorrente.

Porém, é na *autonomia de comportamento* que as diferenças se reforçam. Um agente pode, por si só, decidir se responde ou não às mensagens e/ou solicitações de outras entidades. Isso contrasta com o comportamento básico dos objetos, que devem sempre honrar as solicitações de serviços provenientes de outros objetos (ou agentes).

Quanto à característica de **interação**, os agentes utilizam-se de **linguagens de comunicação** de alto nível. Essas linguagens podem expressar informações sobre crenças, desejos e intenções, ou seja, a atitude do agente com respeito ao que está sendo comunicado. Deste modo, os tipos de mensagens que um agente pode responder são, teoricamente, ilimitados, já que esses elementos podem ser combinados entre si. Por outro lado, objetos usam, em sua comunicação, um conjunto fixo de mensagens, definido por suas interfaces.

Embora não seja possível definir-se uma granularidade funcional capaz de diferenciar agentes de simples objetos, agentes (inteligentes) de software são, normalmente, considerados funcionalmente mais evoluídos (mais "espertos") que simples objetos por possuírem componentes complexos de raciocínio lógico e aprendizado. Dessa forma, pode-se dizer que um agente de software representa um nível de abstração mais elevado do que um objeto.

A partir deste ponto, o termo **agente** será utilizado, neste trabalho, como sinônimo de **agente de software**.

2.5 Considerações Finais

Neste capítulo apresentou-se uma revisão bibliográfica sobre a Tecnologia de Agentes, dando-se ênfase a como o conceito de agente, originário da Inteligência Artificial, evoluiu para os atuais agentes de software. Também foram introduzidos alguns conceitos básicos e a

terminologia utilizada na literatura especializada. No capítulo seguinte, esses conceitos e terminologia serão ampliados para o caso específico dos agentes móveis de software. Alguns desses conceitos serão especialmente importantes para a compreensão do *framework* proposto.

Capítulo 3

Agentes Móveis

3.1 Introdução

O objetivo deste capítulo é apresentar a terminologia e alguns dos principais conceitos utilizados na literatura sobre agentes móveis, bem como descrever alguns dos principais sistemas utilizados na sua construção, dando ênfase aos aspectos relacionados: (1) à linguagem de programação utilizada; (2) à construção de um agente, isso é, como um agente deve ser implementado para que seja reconhecido pelo sistema como um agente móvel; (3) à possibilidade de instalação de serviços adicionais em uma agência e como esses serviços podem ser acessados pelos agentes; (4) ao modo como as classes que compõem um agente móvel são "carregadas" para a memória e colocadas em execução e (5) às regras que regem a transferência dessas classes para outras agências.

Esses aspectos influenciarão a arquitetura proposta para o *Framework de Inteligência* introduzido no Capítulo 6 e desenvolvido no Capítulo 7. Essa arquitetura, além de reutilizável e extensível, deverá permitir a criação de agentes móveis inteligentes sem que seja necessário modificar os sistemas de mobilidade existentes.

3.2 Ambiente Distribuído de Agentes

Um **agente de software móvel** é um tipo específico de agente de software com a habilidade de mover-se, de forma autônoma, entre as diversas agências que compõem um **Ambiente Distribuído de Agentes** (ADA). A Figura 2 mostra, de forma genérica, o Ambiente

Distribuído de Agentes. Uma **agência** é uma localização lógica, instalada em um computador hospedeiro conectado em rede, para onde o agente pode ir e ser executado. Cada agência oferece um conjunto de serviços. Para realizar sua tarefa, um agente móvel migra de uma agência para outra, interagindo com os serviços disponíveis em cada uma delas. Portanto, um agente móvel é capaz de realizar diferentes partes de sua tarefa em diferentes nós de rede.

Por exemplo, um agente pode precisar coletar informações de diferentes bases de dados. Para tirar vantagem da velocidade das interações entre processos em um mesmo computador, em vez de usar chamadas de procedimentos remotos, o agente visita as agências instaladas nos servidores das bases de dados, uma após a outra, acessando as informações localmente. Antes de migrar para a próxima agência ele pode filtrar as informações obtidas, mantendo consigo somente aquelas de interesse.

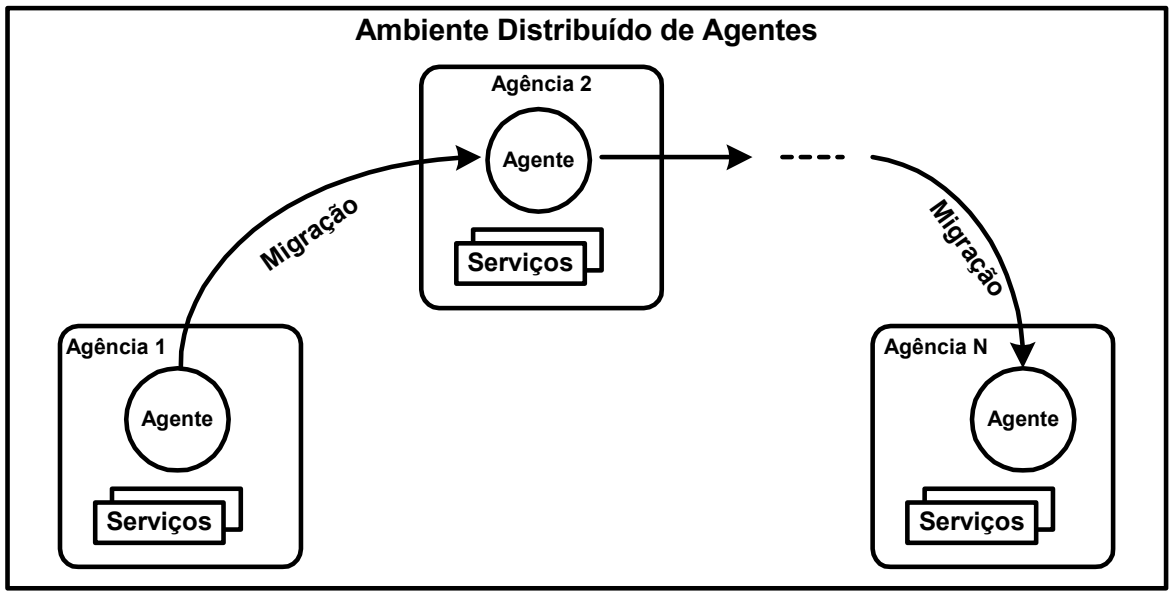


Figura 2: Ambiente Distribuído de Agentes

O conceito de agentes móveis derivou, basicamente, de duas tecnologias, provenientes do ramo dos sistemas operacionais distribuídos: a **migração de processos** e a **migração de objetos**. Inicialmente surgiu a **migração de processos**, que permite a transferência de um processo, e de todo o seu espaço de endereçamento, do computador onde ele está sendo executado, para um outro computador. Os mecanismos de migração administram os vínculos entre o processo e seu ambiente de execução, como, por exemplo, descritores de arquivos e referências a variáveis de ambiente, para permitir que o processo tenha sua execução reinicia-

da no ambiente remoto sem que a interrupção seja percebida. Na maioria dos sistemas desenvolvidos, a migração do processo ocorre de forma transparente, isto é, o programador não possui controle sobre ela e nem conhecimento de sua ocorrência. Embora a motivação inicial tenha sido o balanceamento de carga entre os diversos nós que compõem um sistema distribuído, a migração de processos também reduz o tráfego na rede, comparado à **Chamada de Procedimentos Remotos** (*Remote Procedure Call* – RPC), quando são necessárias muitas chamadas remotas para realizar uma aplicação. No entanto, esse mecanismo não possibilita o retorno dos resultados do processamento remoto para o computador de origem sem que o processo inteiro também retorne.

A **migração de objetos** tornou possível mover objetos entre espaços de endereçamento distintos, implementando uma mobilidade de granularidade mais fina que a migração de processos. Um grande número de **sistemas de objetos móveis** tornou-se popular na década de 80, entre eles o sistema ***Emerald*** (JUL et al., 1983), desenvolvido na Universidade de Washington, precursor de vários dos atuais sistemas de agentes móveis, notoriamente *Telescript* (GENERAL MAGIC, 1997), *Odyssey* (GENERAL MAGIC, 1999), *Aglets* (IBM, 2002a), *Voyager* (OBJECTSPACE, 2000) e *Concordia* (MITSUBISHI, 2000a). O *Emerald* permite a migração de objetos de qualquer granularidade, desde objetos "atômicos" até objetos compostos, mas não fornece transparência completa, uma vez que o programador pode determinar as localizações dos objetos e solicitar, explicitamente, a migração de um objeto para um nó particular.

A partir de meados da década de 90, os conceitos de agente e de objeto móvel deram origem à idéia dos agentes móveis. Tanto a migração de processos quanto a migração de objetos foram introduzidas como funcionalidades do sistema operacional para permitirem o balanceamento de carga entre os nós de redes e a otimização do desempenho do sistema como um todo. Os sistemas de agentes móveis, por outro lado, consideram um conjunto mais amplo de necessidades e requisitos, como, por exemplo, a personalização de serviços, a extensão dinâmica das funcionalidades de aplicações e o suporte à operação desconectada. Basicamente agregou-se ao objeto móvel a **autonomia de navegação**, isto é, a capacidade dos agentes decidirem, por si mesmos, com base nas tarefas que precisam realizar e no conteúdo de sua base de conhecimento, quando e para onde devem migrar. Recentemente, aspectos de comunicação de alto nível, baseados nas especificações da FIPA para uma **Linguagem de Comunicação**

entre Agentes também foram incorporados (IKV++, 2001a). No entanto, no tocante à característica de adaptabilidade, a forma considerada até agora ainda é bastante limitada, restringindo-se a simples reação direta e predeterminada a eventos específicos, apoiada nos mecanismos oferecidos pela programação procedural clássica.

3.3 Migração Fraca versus Migração Forte

Existem, basicamente, duas formas de migração: a **migração forte** e a **migração fraca** (FUGGETTA et al., 1998).

Na migração forte, o agente leva consigo, além de seu **código**, todo o **estado de execução**, caracterizado pelas informações de controle de execução, como, por exemplo, o **ponteiro de instrução** e a **pilha**; e pelos conteúdos de todas as suas estruturas internas de dados e por um conjunto de referências aos recursos utilizados. Essas informações permitem que, após a migração, o agente continue o processamento de sua tarefa no ponto exato em que ele foi interrompido. Se o agente for implementado utilizando-se a tecnologia de objetos, suas estruturas internas de dados e o seu conjunto de referências são mantidos nas variáveis de instância dos objetos que constituem o agente.

Na migração fraca, o agente leva consigo, de uma agência para outra, além do seu código, apenas alguns **dados de reinicialização**, constituídos por parte de suas estruturas internas de dados e, possivelmente, de seu conjunto de referências aos recursos utilizados. Esse conjunto de informações de reinicialização é formalmente denominado **estado de dados** do agente. As informações de controle da execução não são transportadas e o programador do agente é quem deve decidir que estruturas internas de dados e que referências devem fazer parte do estado de dados. Após ser transferido através da rede, o estado de dados é fornecido ao agente, na agência de destino, como dados de inicialização da nova execução, a qual é retomada sempre de um mesmo ponto predefinido. Normalmente, o código do agente é dividido em blocos e, a partir desse ponto comum de retomada de execução, com base nas informações contidas no estado de dados, decide-se que bloco deve ser executado na nova agência.

3.4 Java como Linguagem Preferencial de Programação

Embora existam outras linguagens para a programação de agentes móveis, atualmente, há uma acentuada preferência pela utilização da linguagem Java. Essa escolha se deve a um grande número de características não encontradas, como um todo, nas demais linguagens, que a tornam particularmente apropriada para a construção de tais sistemas. Por exemplo (SUN, 2004):

- **Java é portável**. Esse é o principal atrativo da linguagem para a construção de agentes móveis. O uso de um código intermediário interpretado (*bytecodes*) e o seu respectivo ambiente de execução (*Java Virtual Machine* – JVM) permitem que qualquer plataforma com recursos suficientes, independentemente de seu hardware e sistema operacional, possa hospedar programas em Java. Para os agentes móveis essa característica é essencial, pois transforma cada computador na rede em um hospedeiro potencial.
- **Java é robusta**. Java possui um mecanismo de tratamento de exceções que facilita a recuperação confiável em uma situação de falha. Essa característica garante que uma falha em um agente não comprometa o correto funcionamento da agência e do computador hospedeiros.
- **Java oferece suporte à programação concorrente** (*multi-threading*). Essa característica permite que cada agente possua seu próprio fluxo de controle de execução, o que garante a **autonomia operacional** de cada um deles.
- **Java oferece um mecanismo de tratamento de eventos**. O modelo de tratamento de eventos, oferecido pela linguagem, facilita a implementação da habilidade de **reatividade** dos agentes, permitindo sua resposta a eventos externos.
- **Java oferece suporte à conexão em rede**. Esse suporte inclui *sockets*, comunicação baseada em URLs e um protocolo de comunicação entre objetos distribuídos chamado **Invocação Remota de Métodos** (*Remote Method Invocation* – RMI). Essas características são importantes tanto na implementação dos serviços de comunicação entre agentes quanto na implementação dos serviços de transporte dos próprios agentes.

- **Java possui um mecanismo de serialização de objetos**. Esse mecanismo permite que as estruturas internas de dados de um agente sejam facilmente transformadas em um fluxo seqüencial de bits (**serialização**). Esse fluxo pode, por exemplo, ser transportado para outro computador através de uma conexão de rede ou armazenado em algum tipo de repositório persistente de dados, tais como bancos de dados orientados a objetos e sistemas de arquivos. Portanto, esse mecanismo facilita a implementação dos serviços de transporte e de persistência de um sistema de agentes. O Apêndice B descreve brevemente o processo de serialização.
- **Java possui um mecanismo de carregamento dinâmico de classes** (*class loader*). Esse mecanismo possibilita que o código de uma nova classe seja introduzido num processo já em execução sem que seja necessário o seu conhecimento prévio em tempo de compilação. Dessa forma, objetos dessa nova classe poderão ser instanciados no processo em andamento. Essa característica facilita a implementação do serviço de transporte de agentes. Uma vez que o código do agente e o seu estado de dados serializado tenham sido transportados para outro computador, através de uma conexão de rede, esse mecanismo poderá, a partir deles, reconstruir os objetos que compõem o agente.
- **Java permite a resolução dinâmica de referências de código** (*dynamic binding*). Essa característica permite que uma solicitação para execução de uma operação seja vinculada a uma implementação específica somente em tempo de execução. Conseqüentemente, pode-se escrever programas que esperam um objeto com uma interface em particular, sabendo-se que qualquer objeto que possua a interface correta poderá aceitar a solicitação. Além disso, ela também permite que objetos com interfaces idênticas sejam substituídos uns pelos outros (**polimorfismo**). Isso facilita tanto o desenvolvimento das agências quanto dos agentes de um sistema de agentes. Utilizando essa propriedade, o código de uma agência pode ser escrito independentemente dos códigos dos agentes que ela hospedará e, por outro lado, os códigos dos agentes poderão fazer referência a serviços oferecidos pelas agências a serem visitadas, sem um conhecimento prévio de suas implementações.

- **Java possui um mecanismo de segurança embutido**. Todo código carregado pelo mecanismo de carregamento dinâmico de classes Java está sujeito a restrições de segurança impostas por um mecanismo de segurança, que pode ser facilmente configurável para a implementação de políticas de segurança específicas. Por exemplo, enquanto agentes de um usuário podem receber permissão para escrever em arquivos, agentes de outro usuário podem receber somente permissão de leitura e os de um terceiro usuário, nenhuma permissão de acesso.
- **Java oferece suporte à localização de serviços**. Pelo menos duas tecnologias associadas a Java podem ser utilizadas para essa finalidade. A primeira é a *Java Naming and Directory Interface* (JNDI), uma API que oferece a aplicações distribuídas escritas em Java as funcionalidades de nomes e diretórios, permitindo que as mesmas compartilhem informações sobre usuários, máquinas, redes, serviços e outras aplicações (WONG et al., 1999). A segunda é o sistema de interconexão JINI. "JINI é uma infra-estrutura distribuída baseada na linguagem e no ambiente Java, que define um conjunto pequeno e simples de convenções capaz de permitir que serviços e clientes formem um sistema distribuído extremamente dinâmico" (WALDO, 2001). Como o objetivo principal da tecnologia JINI é permitir que aplicações possam descobrir serviços em uma rede, agentes que executarem em um sistema que faça uso dessa tecnologia poderão localizar os serviços desejados e, então, migrar para as agências que os oferecem.
- **Java possui alta aceitação do mercado como uma linguagem orientada a objetos de propósito geral**. Existe um forte movimento de grandes segmentos da indústria de software para o uso da linguagem; isso faz com que prolifere o número de ferramentas de alta qualidade tanto para o desenvolvimento, quanto para a depuração de aplicações.

No entanto, apesar das vantagens acima, a linguagem de programação Java padrão não oferece a possibilidade de capturar o estado de execução de um processo ou fluxo de execução (*thread*). Isso somente seria possível modificando-se a Máquina Virtual Java (*Java Virtual Machine* - JVM). Contudo, como um dos objetivos dos Sistemas de Agentes Móveis atuais é que eles sejam os mais abertos possível, no que diz respeito ao ambiente de software subjacente, esta possibilidade, normalmente, não é considerada em seus desenvolvimentos.

Portanto, os sistemas de agentes móveis, aderentes às especificações da linguagem Java estabelecidas pela Sun Microsystems, utilizam, necessariamente, a migração fraca no transporte de agentes através da rede.

3.5 Ciclo de Vida de um Agente Baseado na Migração Fraca

Esta seção descreve, de forma genérica, o ciclo de vida de um agente móvel em um Ambiente Distribuído de Agentes, construído a partir de um Sistema de Agentes Móveis baseado na migração fraca. Esse ciclo de vida tem por objetivo "simular", dentro de certos limites, o mecanismo de migração forte, usando um mecanismo de migração fraca. O ciclo de vida de um agente móvel baseado na migração fraca é o seguinte:

1) O agente é criado (instanciado) em uma agência, chamada de **agência natal**. Durante o processo de criação, o agente pode receber um conjunto de **argumentos de criação**. Imediatamente após sua criação, o agente pode executar um conjunto de **procedimentos de criação**, isto é, um conjunto de procedimentos que são realizados uma única vez, logo após a sua criação. Após a execução dos procedimentos de criação, o agente estará no **estado inativo**, o que significa que ele já está instanciado mas ainda não iniciou a execução de sua tarefa.
2) A agência natal inicia o fluxo de execução principal do agente (*thread*) em um **grupo próprio de fluxos de execução**. Nesse fluxo de execução principal, o agente pode criar fluxos de execução adicionais (processamento concorrente). Agora o agente está no **estado ativo**, isto é, o estado no qual ele está realizando a tarefa para a qual ele foi criado. É necessário criar um grupo de fluxos de execução para cada agente para garantir que todos os fluxos adicionais criados por um agente terminem quando o agente terminar sua execução.
3) Quando o agente decide continuar sua execução em uma outra agência, ele solicita que a **agência hospedeira** envie seu código e seu estado de dados para a **agência de destino**.

4) A agência hospedeira interrompe a execução do agente, parando seu fluxo de execução principal, o que coloca o agente no estado inativo.
5) A agência hospedeira serializa o estado de dados do agente, transformando todas as variáveis de instância do agente não declaradas como **transiente** em um fluxo serial de bits.
6) A agência hospedeira transfere para a agência de destino o nome da classe principal do agente, bem como a identificação de sua base de códigos e seu estado de dados serializado.
7) A agência de destino cria uma nova instância do agente e a inicializa com o estado de dados recebido da **agência remetente**. O agente estará no estado inativo. Se os códigos das classes que compõem o agente não estão presentes com antecedência na agência de destino, eles são recuperados da base de códigos a partir de sua identificação enviada pela agência remetente.
8) A agência de destino informa à agência remetente se o agente foi instanciado com sucesso.
9) Se o agente foi instanciado com sucesso na agência de destino, a agência remetente remove a instância local do agente.
10) A agência de destino invoca um método específico do agente para prepará-lo para a retomada de sua execução. Esse método, por exemplo, aloca recursos a serem utilizados pelo agente.
11) A agência de destino inicia o fluxo de execução principal do agente em um **grupo próprio de fluxos de execução**. O agente entra em seu estado ativo, iniciando a execução de sua tarefa.
12) Os passos 3 a 11 repetem-se para cada migração do agente.
13) Ao completar sua tarefa, o agente pode retornar à sua agência natal, apresentando seus resultados a seu usuário.
14) Após a apresentação dos resultados o agente pode ser destruído.

3.6 Sistemas de Agentes Móveis

A Tabela 2 apresenta um levantamento de alguns dos sistemas desenvolvidos até o momento para a construção de um ambiente distribuído de agentes (AGENTBUILDER, 2000). Alguns desses sistemas serão descritos nas subseções seguintes. O que se pretende é evidenciar alguns aspectos genéricos, isto é, que variam pouco de sistema para sistema, e que terão algum tipo de impacto sobre a arquitetura proposta para o *framework* de inteligência. Na análise desses aspectos, já citados na introdução do capítulo, dar-se-á ênfase aos sistemas mais difundidos atualmente e que continuam em evolução. Dentre eles estão o ***Aglets***, o ***Grasshopper***, o ***Voyager*** e o ***Concordia.*** Os dois primeiros serão descritos em maiores detalhes, pois apresentam documentação mais vasta e versões de avaliação (por tempo indeterminado) disponíveis na Internet. No caso do *Aglets*, até mesmo o código fonte está disponível. Os demais serão descritos brevemente.

Tabela 2: Exemplo de alguns sistemas para a construção de agentes móveis

Produto	Organização	Linguagem	Categoria
D'Agent (D'AGENT, 2000)	Universidade de Dartmouth	Tcl	Projeto
Aglets (IBM, 2002a)	IBM do Japão	Java	Produto
Ara (ARA, 2000)	Universidade de Kaiserslautern	C, C++, Tcl	Projeto
Concordia (Mitsubishi, 2000a)	Mitsubishi Electric	Java	Produto
ADK (Tryllian, 2000)	Tryllian B. V. Inc.	Java	Produto
Grasshopper (IKV++, 2001c)	IKV++ GmbH	Java	Produto
Gypsy (Gypsy, 2000)	Universidade Técnica de Viena	Java	Projeto
Java-to-go (Java-To-Go, 2000)	Universidade da Califórnia, Berkeley	Java	Projeto
JumpingBeans (Astra, 1999)	Ad Astra Enginnering Inc.	Java	Produto
Knowbot (Knowbot, 2000)	CNRI	Python	Projeto
Mole (Mole, 2000)	Universidade de Stuttgart	Java	Projeto
Odyssey (General Magic, 1999)	General Magic	Java	Produto
Tacoma (Tacoma, 2000)	Universidades de Thomsϕ e Cornell	C, C++, Tcl	Projeto
Telescript (General Magic, 1997)	General Magic	Telescript	Produto
Voyager (ObjectSpace, 2001)	ObjectSpace Inc.	Java	Produto

3.6.1 Aglets

Um ***aglet*** (*agile applet*) é um objeto Java que pode se mover de um computador para outro na Internet (IBM, 2002b). Dessa forma, um *aglet* em execução em um computador pode subitamente interromper seu processamento, mover-se para um computador remoto e lá reiniciar sua execução. Um itinerário de "viagem" é usado para especificar os destinos a serem visitados e as ações a serem executadas em cada um deles. Quando um *aglet* se move, ele leva consigo o código de seu programa (o código de suas classes) e o seu estado de dados. Os *bytecodes* e o estado de dados de um *aglet* são transformados em um fluxo de bits pelo mecanismo de serialização de Java e carregados na agência destino por um carregador de classes especialmente desenvolvido para tal. *Aglets* podem se comunicar usando a metáfora do **quadro branco**, que permite aos agentes colaborarem e trocarem informações de forma assíncrona. Trocas síncronas e assíncronas de mensagens também são suportadas. Um mecanismo de segurança, embutido no próprio sistema, garante a segurança de um computador, na eventualidade de *aglets* não confiáveis.

As APIs fornecidas pela IBM possibilitam a construção tanto dos *aglets* quanto das agências que os hospedam, bem como a introdução das funcionalidades de criação, envio e recebimento de agentes móveis em qualquer aplicação Java.

3.6.1.1 Arquitetura Geral

A arquitetura de *Aglets* consiste de duas camadas: a **camada de execução** e a **camada de comunicação**. Duas APIs definem as interfaces de acesso às funções oferecidas por cada uma delas. A Figura 3 mostra a arquitetura geral do *Aglets*.

A camada de execução implementa a API *Aglet*, descrita no item seguinte, definindo os comportamentos de seus componentes, como por exemplo `AgletProxy` e `AgletContext`. Ela fornece as funções fundamentais para a criação, gerenciamento e despacho de agentes para computadores remotos. A camada de comunicação é responsável, principalmente, por transferir um agente serializado para o seu destino e também por recebê-lo. Ela também dá suporte à comunicação entre agentes e a algumas facilidades para o seu gerenciamento.

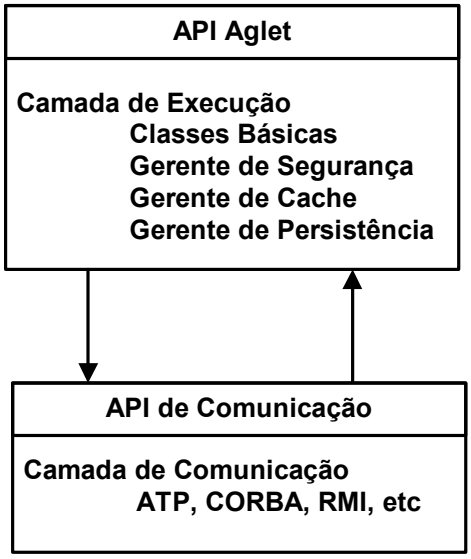


Figura 3: Arquitetura geral do *Aglets* (IBM, 2002b)

3.6.1.2 API *Aglet*

A API ***Aglet*** define as funcionalidades básicas de um agente móvel. A Figura 4 mostra as principais classes e interfaces definidas nessa API e seus relacionamentos.

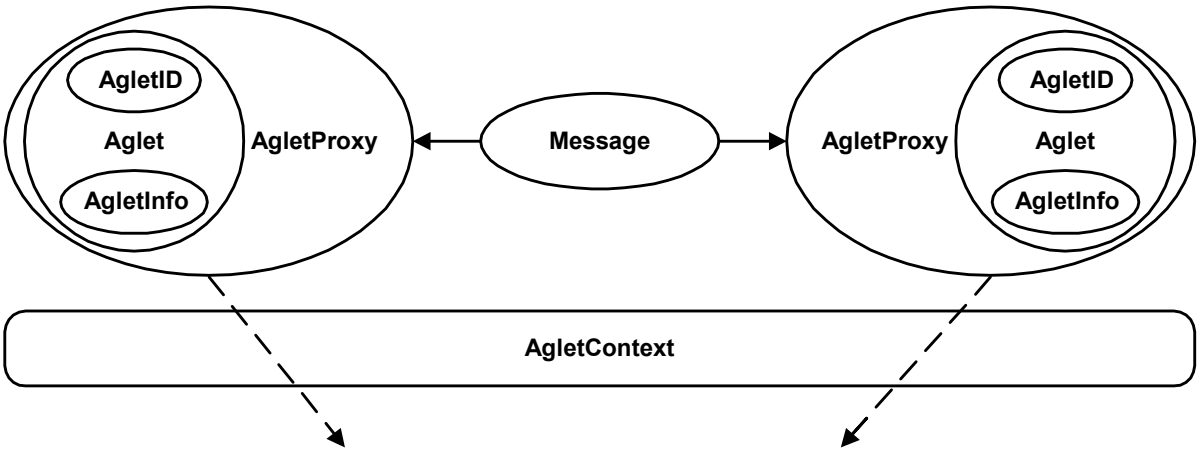


Figura 4: Principais classes e interfaces da API *Aglet* (IBM, 2002b)

A classe abstrata `Aglet` define as operações básicas utilizadas para controlar a mobilidade e o ciclo de vida dos agentes. Para que um agente seja reconhecido pelo sistema como tal ele precisa ser uma subclasse dessa classe abstrata. Durante todo o seu ciclo de vida, um *aglet* possui uma identidade única, imutável, encapsulada em um objeto da classe `AgletID`,

que pode ser obtida pela operação `Aglet.getAgletID()`. Seus atributos estáticos e dinâmicos, como, por exemplo, a data de sua criação, a URL de sua base de código, a URL de sua agência atual e o instante de sua chegada também podem ser acessados por intermédio da classe `Aglet`. Essas informações são encapsuladas em um objeto da classe `AgletInfo`, que pode ser obtido através da operação `Aglet.getAgletInfo()`.

A classe `Aglet` também possui diversos métodos públicos que devem ser invocados somente pela agência mas que, por razões de segurança, não devem ser visíveis para outros agentes. Dessa forma, qualquer *aglet* que desejar se comunicar com outro *aglet* deverá primeiro obter uma referência para um objeto que represente o *aglet* a ser contatado (objeto *proxy*) e então interagir com ele através da interface `AgletProxy`. Ao receber uma invocação, o objeto *proxy* consulta um gerente de segurança (`SecurityManager`) para determinar se o chamador possui permissão para realizá-la. A decisão de permitir ou não a invocação pode ser baseada em diversos fatores, como por exemplo a URL de origem do agente. Um outro papel importante da interface `AgletProxy` é o provimento da **transparência de localização**. Se o agente a ser contatado residir em outro computador, o *proxy* encaminha a solicitação a esse computador e a resposta ao computador local.

Aglets comunicam-se pela troca de objetos da classe `Message`. Um objeto dessa classe possui um objeto texto para especificar o tipo da mensagem e uma lista de objetos genéricos (classe `java.lang.Object`) como argumentos. Uma mensagem pode ser enviada para um *aglet* invocando-se, dentre outras, a operação `sendMessage(Message msg)` de seu objeto *proxy*.

A interface `AgletContext` fornece um meio de acesso aos recursos e informações do ambiente de execução (agência) no qual o agente se encontra. Todo *aglet* pode obter uma referência para o seu contexto de execução através da operação `Aglet.getAglet Context()` e então usá-la para obter informações locais, tais como a URL do computador hospedeiro, uma lista de representantes (objetos *proxies*) dos agentes presentes na mesma agência ou referências para objetos (não agentes) que ofereçam seus serviços localmente. Essa última possibilidade é de especial interesse. Toda agência possui uma lista de propriedades do tipo **chave - valor** onde a chave é sempre um texto e o valor uma referência para um objeto genérico. Se a agência desejar tornar disponível, para seus agentes, um serviço oferecido por determinado objeto, ela poderá registrar esse serviço na lista de propriedades adicionando

uma dupla chave – valor, onde o texto da chave corresponde ao nome do serviço (padronizado e conhecido pelos agentes) e o valor corresponde a uma referência para o objeto servidor, cuja interface também deve ser conhecida por todos os agentes que desejam utilizá-lo.

3.6.1.3 Mecanismos para Transferência e Carregamento de Classes

Num sistema de agentes móveis, as classes que compõem um agente precisam estar disponíveis nos computadores hospedeiros para que ele possa ser instanciado e iniciar sua execução. No entanto, isso não significa que essas classes devam estar previamente instaladas em todos os computadores do ADA. Portanto, tais sistemas precisam possuir facilidades para: (1) transferir o conjunto de classes que compõem o agente de um computador hospedeiro para outro e (2) carregar o código de uma classe, sob demanda, em tempo de execução.

O sistema *Aglets* suporta dois esquemas para a transferência de código. O primeiro é a transferência do código da classe junto com o agente, no momento de seu transporte; o segundo é a transferência sob demanda do código da classe, no momento de sua utilização, após o *aglet* estar em execução na nova agência. Os dois esquemas são usados de forma combinada, de acordo com a categoria a que pertençam as classes a serem transportadas.

De acordo com sua origem, as classes no sistema *Aglets* são classificadas em 4 categorias: **classes arquivadas**, cujos códigos estão presentes em um arquivo do tipo `JAR` (*Java Archive Resource*); **classes de base de código**, cujos códigos estão armazenados em um diretório específico do sistema de arquivos do computador natal do agente; **classes de sistema**, cujos códigos podem ser encontrados no computador hospedeiro, a partir do CLASSPATH local e **outras**, cujos códigos fazem parte da base de código de outros agentes. Esse último caso ocorre, por exemplo, quando um *aglet* recebe uma mensagem de outro *aglet*, contendo como argumento uma referência para um objeto cuja definição da classe está na base de código do emissor.

Cada categoria é gerenciada de modo diferente pelos mecanismos de transporte. Se as classes usadas por um agente estão todas armazenadas em um arquivo `JAR`, elas são todas transferidas junto com o agente, no momento de seu transporte. Se nenhum arquivo `JAR` for especificado, somente as classes carregadas a partir da base de código (diretório natal) do agente poderão ser transferidas junto com ele. No entanto, se alguma delas possuir o mesmo

nome de uma classe de sistema ela não será transferida pois classes de sistema nunca são transmitidas. Por outro lado, o conjunto de classes que realmente será transferido junto com o agente é determinado em tempo de execução, durante o processo de serialização. As classes de todos os objetos "visitados" durante esse processo serão coletadas e transferidas. Classes referenciadas por variáveis de instâncias com valor nulo (`null`) ou marcadas com o modificador `transient` não são transferidas. Nesse caso, se essas classes forem necessárias após o agente ter sido instanciado, os seus códigos serão obtidos, sob demanda, de acordo com as regras de carregamento descritas mais à diante. Finalmente, se um agente tentar transferir uma classe cuja base de código seja diferente da sua, isso provocará uma exceção pois, por motivos de segurança, *Aglets* não permite que um *aglet* transporte classes de duas bases de código diferentes ou extraia o código de outro agente.

Na agência destino, os códigos transportados são recuperados do fluxo serializado de bits e armazenados num sistema de *cache* exclusivo para cada agente (***cache* do agente**). No entanto, isso não garante que o código transferido será utilizado na reconstrução do agente. A escolha do código utilizado é realizada de acordo com as regras embutidas em um carregador de classe especialmente desenvolvido para o sistema. Esse carregador, denominado `AgletClassLoader`, especializa o carregador genérico `ClassLoader` presente no kit de desenvolvimento Java, que é capaz de criar, em tempo de execução, instâncias de uma nova classe a partir de seu *bytecode*. Cada *aglet* é associado a exatamente uma instância de `AgletClassLoader` e todas as classes utilizadas por ele serão gerenciadas por esse mesmo carregador. Um mesmo carregador pode gerenciar diversas instâncias *aglets*.

Embora possam existir diversas fontes para o *bytecode* de uma mesma classe, somente uma dessas fontes será escolhida pelo carregador de classes no momento da criação de uma instância da classe em questão. Cada carregador de classes mantém seu próprio cache, denominado ***cache* do carregador**, com todas as classes utilizadas por todos os agentes já carregados por ele. Quando é necessário criar um novo objeto, o carregador verifica primeiro esse cache. Se a definição da classe desejada já estiver lá, ela é carregada a partir dele. Se a definição não for encontrada em seu cache, o carregador tentará encontrá-la dentre as definições das **classes de sistema**, ou seja, dentre aquelas classes cujos caminhos para os códigos estão definidos na variável local de ambiente CLASSPATH. Se essa operação também não for bem sucedida, o carregador tentará carregar a definição da classe a partir do ***cache* do a-**

gente. Se também essa operação falhar, o carregador procurará pela definição na base de código do agente. Nesse caso, se o agente foi criado em outra agência, a definição da classe precisará ser transportada, sob demanda, até a agência atual. Uma vez que a definição da nova classe é encontrada, ela é armazenada no cache do carregador para usos futuros. Porém, se a definição também não for encontrada na base de código do agente, então uma exceção será gerada.

Esse mecanismo de *cache* de dois níveis, composto por um cache do carregador, gerenciado pelo `AgletClassLoader` e por um cache do agente, gerenciado por um `CacheManager`, tem por objetivo reduzir a necessidade de transporte do código das classes. No entanto, uma vez que um mesmo carregador não pode lidar com mais de uma definição de uma mesma classe, se uma definição mais antiga já estiver armazenada no cache do carregador, ela poderá ser utilizada, mesmo que uma nova versão esteja disponível no cache do agente. Isso pode causar problemas na deserialização dos objetos já que os dados serializados podem ser incompatíveis com a versão mais antiga da classe e, ainda pior, o comportamento da classe mais antiga pode ser totalmente diferente do da nova versão. Para evitar o problema de conflitos de versões, o sistema *Aglets* envia informações a respeito da versão de cada classe, juntamente com seu nome e código. O carregador mantém essas mesmas informações no seu cache. Tais informações são obtidas do arquivo `MANIFEST` presentes nos arquivos `JAR` ou computadas em tempo de execução para as demais categorias de classes. De posse dessas informações, a agência de destino pode descobrir quais classes e quais de suas versões são utilizadas por um *aglet*, mesmo antes de receber todo o seu código, e escolher o carregador adequado. Portanto, agentes com versões de classes diferentes são gerenciados por instâncias diferentes de `AgletClassLoader`.

3.6.2 Grasshopper

Grasshopper é o primeiro ambiente para agentes móveis aderentes ao padrão *Mobile Agent System Interoperability Facility* (MASIF), estabelecido pela OMG (IKV++, 2001b). Esse padrão, construído sobre a arquitetura CORBA, promove a integração entre o paradigma cliente/servidor tradicional e a tecnologia de agentes móveis. Seu objetivo é assegurar a interoperabilidade entre aplicações desenvolvidas a partir de ambientes de agentes distintos.

3.6.2.1 Arquitetura Geral

A Figura 5 mostra os elementos que compõem a arquitetura do *Grasshopper*. A **agência** é o componente que fornece as funcionalidades para a execução dos agentes e dos diversos serviços de suporte. Pelo menos uma agência deve estar em execução em cada computador hospedeiro que faça parte do ambiente distribuído de agentes promovido pelo *Grasshopper*.

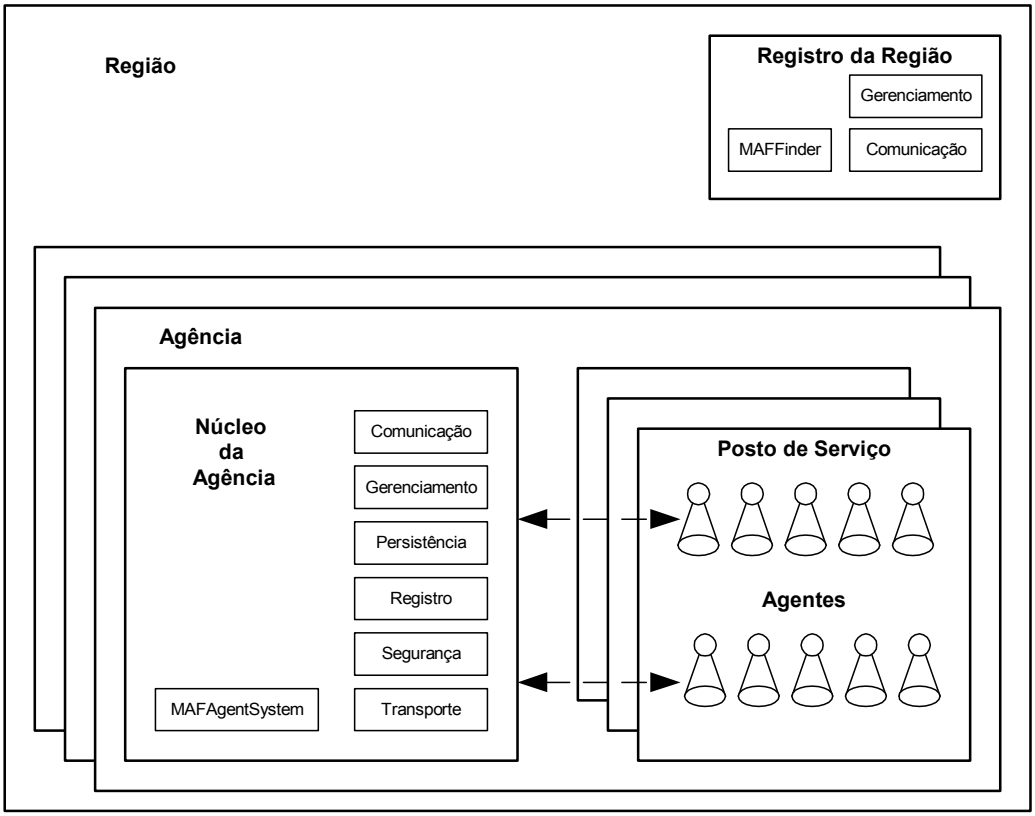


Figura 5: Arquitetura do *Grasshopper*

Uma **agência** é composta por duas partes: um **núcleo básico** (*core*) e um ou mais **postos de serviço** (*places*). Um **posto de serviço** representa um agrupamento lógico de **recursos** de uma agência, por meio do qual os agentes podem acessar serviços com características específicas, como por exemplo, comunicação segura ou negociação comercial. O **núcleo básico** oferece as funcionalidades de suporte mínimas necessárias à execução dos agentes. Os seguintes serviços são oferecidos pelo núcleo: (1) **comunicação**, responsável pelas interações locais e remotas entre os diversos elementos distribuídos; (2) **registro**, mantém informações

sobre os agentes e os postos de serviço existentes na agência; (3) **gerenciamento**, permite o monitoramento e controle de agentes e postos de serviço por partes dos usuários; (4) **segurança**, fornece mecanismos de proteção para as interações remotas e locais; (5) **persistência,** permite que as informações mantidas nos agentes e postos de serviço sejam armazenadas em um meio persistente; (6) **transporte**, construído sobre os serviços de comunicação e segurança, realiza o transporte do código e do estado de dados de um agente de uma agência para outra.

Uma **região** é um agrupamento de várias agências conectadas para um objetivo comum, oferecendo transparência de localização para seus agentes. O **registro de região** (*region registry*) fornece uma visão geral das agências, postos de serviço, e agentes presentes no contexto de uma região. A adição e remoção no registro de uma região das informações correspondentes a esses elementos são realizadas automaticamente, embora o *Grasshopper* também permita o registro manual de outros objetos, de modo a torná-los acessíveis. O ADA pode ser estabelecido sem um registro de região. No entanto, nesse caso, as agências e os agentes devem ter conhecimento de todas as informações necessárias para o estabelecimento das interações remotas, como, por exemplo, nome do computador hospedeiro, número das portas, protocolos de comunicação, etc.

No contexto do *Grasshopper*, todo agente é considerado um serviço, isto é, um componente de software que oferece certas funcionalidades para outras entidades do ADA. Basicamente dois tipos de agentes podem estar presentes em um posto de serviço: **agentes móveis** e **agentes estáticos**. Agentes com a habilidade da mobilidade podem migrar de um posto de serviço para outro, inclusive entre agências distintas, de modo a acessarem os recursos que necessitam no momento. Agentes estáticos não podem migrar e executam somente na agência na qual foram criados, podendo ser utilizados no oferecimento de serviços específicos dessa agência. Todo agente é composto por duas partes: uma parte comum e uma parte específica. A parte comum é representada pelas seguintes classes fornecidas pelo sistema: `MobileAgent` e `StationaryAgent`. A parte específica deve ser implementada pelo programador do agente na forma de uma ou mais classes Java, sendo que a classe principal deve ser uma especialização de `MobileAgent` ou de `StationaryAgent` e necessariamente implementar o método `live()`. Esse método define o comportamento ativo do agente, isso é, o fluxo de controle executado em seu próprio fluxo de controle de execução (*thread*). Quando o método `live()`

termina, o agente não é removido mas permanece na agência hospedeira como um objeto passivo.

As funcionalidades oferecidas pela agência local podem ser acessadas através da interface `IAgentSystem`, obtida por meio do método `getAgentSystem()` presente em `MobileAgent` e `StationaryAgent`. Essa referência pode ser usada para obter informações locais, tais como: a URL do computador hospedeiro, uma lista de representantes (objetos *proxies*) dos agentes presentes na mesma agência ou referências para objetos (não agentes) que oferecem seus serviços localmente. Da mesma forma, um agente pode obter uma referência para o registro de região através do método `getRegion()`, que retorna a interface `IRegion`. Além das interfaces de acesso específicas do *Grasshopper* (`IAgentSystem` e `IRegion`), as interfaces padronizadas pelo MASIF (IKV++, 2001b) também podem ser utilizadas, ou seja, `MAFAgentSystem` para as agências e `MAFFinder` para o registro de região.

O serviço de comunicação é baseado na invocação de métodos e suporta interações locais e remotas entre os componentes do *Grasshopper* (agentes, agências e registros de regiões). Também permite que aplicações externas se comuniquem com o ambiente *Grasshopper*. As invocações podem ser síncronas ou assíncronas, estáticas ou dinâmicas e dirigidas a um componente específico (*unicast*) ou a todos os componentes (*multicast*). Por meio de objetos *proxies*, os agentes podem se comunicar entre si de forma transparente quanto à localização. Isso significa que eles podem migrar durante uma sessão de comunicação enquanto o serviço redireciona tanto as invocações de métodos quanto os valores de retorno para as localizações corretas.

Através do serviço de comunicação, uma agência pode tornar disponível para seus agentes um determinado serviço na forma de um agente estático. Os métodos desse agente estático, acessíveis aos agentes clientes, devem ser declarados em uma **interface de servidor** (*server interface*). O código dessa interface é fornecido como entrada para uma ferramenta do *Grasshopper*, denominada ***Stub Generator***, que produz como saída uma classe *proxy*. Os agentes que desejam ser clientes do agente estático precisam de uma instância dessa classe *proxy* para se comunicarem com ele através do serviço de comunicação. Uma referência para essa instância pode ser obtida invocando-se o método `newInstance()` da classe `de.ikv.grasshopper.communication.ProxyGenerator`, tendo como argumento

um identificador do serviço, padronizado e conhecido pelos agentes clientes, obtido através da interface `IAgentSystem`.

3.6.2.2 Mecanismos para o Transporte e Carregamento de Classes

Para criar um agente ou reiniciá-lo após sua chegada, uma agência deve ter acesso às classes que compõem o código do agente. Para isso, a base de código do agente deve ser especificada quando da sua criação através das interfaces de programação (API) ou de usuário (UI) da agência. Se nenhuma base de código for fornecida explicitamente ou se as classes necessárias não forem encontradas na base especificada, a agência procurará por elas nos diretórios presentes na variável de ambiente CLASSPATH.

O *Grasshopper* suporta dois tipos diferentes de base de código: **sistema de arquivos** e **servidor HTTP**. No primeiro caso as classes devem ser mantidas no sistema de arquivos de um computador e no segundo, em um servidor HTTP.

Existem diversas possibilidades para viabilizar o acesso de uma agência à base de código de um agente:

- **os códigos das classes podem ser previamente armazenados em todas as prováveis agências de destino do agente**. Em cada uma dessas agências, o diretório onde as classes foram instaladas deve ser incluído no CLASSPATH. Nesse caso, se um agente migra para uma nova agência, seu código já estará lá. No entanto, uma vez carregadas, as classes presentes no CLASSPATH são mantidas no cache do sistema durante todo o tempo em que a agência estiver em execução. Isto significa que elas somente serão carregadas durante a primeira criação do agente. Se um agente do mesmo tipo for criado uma segunda vez, a agência utilizará as classes já presentes em seu cache e não acessará o sistema de arquivos, mesmo que os códigos das classes tenham sido recompilados antes da segunda criação.
- **os códigos das classes podem ser mantidos somente na agência de origem do agente**. Nesse caso, inicialmente, as classes do agente estarão armazenadas somente no sistema de arquivos da agência de origem do agente, isto é, no computador que hospeda a agência onde o agente foi criado. Se o agente migrar, cada

nova agência de destino deverá solicitar o código das classes à agência de origem. Uma vez carregadas pela agência de destino, essas classes serão mantidas num cache específico do agente (e não no cache do sistema). Portanto, se dois agentes da mesma classe migrarem para a mesma agência, esta requisitará o mesmo código duas vezes. Ou seja, o código do primeiro agente não será usado na criação do segundo.

- **o código das classes pode ser mantido somente em um servidor HTTP central**. Neste caso, mesmo a agência de origem do agente deverá solicitar as classes dessa base de código durante o processo de criação. Uma vez carregadas pela agência de destino, essas classes serão mantidas num cache específico do agente (e não no cache do sistema). Portanto, se dois agentes da mesma classe migrarem para a mesma agência, esta requisitará o mesmo código duas vezes. Ou seja, o código do primeiro agente não será usado na criação do segundo.
- **o código das classes pode ser mantido somente na agência previamente visitada pelo agente**. Em certos cenários, a agência de origem de um agente estará conectada à rede somente temporariamente. Supondo que nenhum servidor central tenha sido especificado como sendo a base de código e que o código também não tenha sido previamente armazenado no sistema de arquivos das prováveis agências de destino, o código poderá ser enviado pela agência atual para a próxima agência a ser visitada. Uma vez carregadas pela agência de destino, essas classes serão mantidas num cache específico do agente (e não no cache do sistema). Portanto, se dois agentes da mesma classe migrarem para a mesma agência, esta requisitará o mesmo código duas vezes. Ou seja, o código do primeiro agente não será usado na criação do segundo.

A seqüência na qual uma agência procurará pelo código é a seguinte:

1) Cache de sistema da agência local (que mantém as classes carregadas a partir do CLASSPATH);
2) Agência anteriormente visitada;
3) Localizações especificadas na base de código do agente (sistemas de arquivos e/ou servidores http);
4) Agência de origem.

3.6.3 Concordia

O Sistema *Concordia* é um *framework* desenvolvido pelo Mitsubishi Electric Information Technology Center America para a criação e gerenciamento de aplicações em rede baseadas no paradigma de agentes móveis (MITSUBISHI, 2000b). Utilizando o sistema *Concordia* uma aplicação pode processar dados em suas fontes, mesmo que seu usuário esteja desconectado da rede, acessar e enviar informações em um ambiente de rede heterogêneo (LANs, intranets e Internet) e utilizar-se de equipamentos de comunicação convencionais e sem-fio. Qualquer dispositivo de hardware que possua uma Máquina Virtual Java instalada pode fazer parte do Ambiente Distribuído de Agentes promovido pelo *Concordia*. Dessa forma, uma larga gama de dispositivos de hardware pode ser integrada, desde telefones "espertos" (*smartphones*) e PDAs (*Personal Digital Assistants*) a servidores "pesados" de corporações. Para que isso seja possível, o sistema é totalmente modular, o que permite sua adequação aos requisitos de memória e de processamento de um dispositivo ou aplicação em particular através da seleção dos componentes necessários.

Na sua forma mais simples, o sistema é composto por um único **Servidor *Concordia*** e um único **Agente *Concordia*** executando em um computador hospedeiro. No entanto, usualmente, um sistema é composto por vários computadores hospedeiros conectados por uma rede local ou de longa distância, cada um dos quais executando vários Servidores *Concordia* e seus respectivos agentes. O Servidor *Concordia* é o programa Java responsável por oferecer as funcionalidades do sistema em um dado hospedeiro. Seus componentes gerenciam todo o ciclo de vida dos agentes, os seus transportes e fornecem o ambiente no qual eles podem executar suas tarefas. A Figura 6 mostra os principais componentes da arquitetura do *Concordia*.

O **Gerente de Agente** fornece a infra-estrutura que permite aos agentes migrarem. Ele abstrai a interface de rede de modo que os programadores não precisem conhecer as especificidades da conexão à rede ou as suas interfaces de programação. Ele também gerencia o ciclo de vida dos agentes. Os agentes *Concordia* são criados especializando-se a classe básica `Agent`. Cada agente possui a seu próprio fluxo de controle de execução, que é interrompido antes da migração e reiniciado no Servidor *Concordia* de destino. Assim como no *Aglet*, um itinerário informa o agente sobre quais agências visitar e quais tarefas devem ser realizadas em cada uma delas. Também como nos demais sistemas, a mobilidade do estado de dados do

agente é baseada na facilidade de serialização de objetos do Java e a mobilidade do código é obtida através da especialização do mecanismo de carregamento de classes. Essa especialização transforma os arquivos contendo os *bytecodes* de todas as classes que compõem o agente em uma estrutura especial de dados capaz de ser transportada através da rede e novamente convertida, no destino, na definição de uma classe Java, a partir da qual objetos podem ser instanciados. Durante a deserialização do agente, os bytecodes do agente e de todas as suas classes relacionadas são recuperados dessa estrutura e utilizados para instanciar uma nova cópia do agente, inicializada com o respectivo estado de dados. Tanto o serviço de transporte como o serviço de comunicação entre agentes, descrito mais adiante, são baseados nos serviços de comunicação oferecidos pelo protocolo TCP/IP, não sendo necessária, portanto, nenhuma infra-estrutura adicional de conexão.

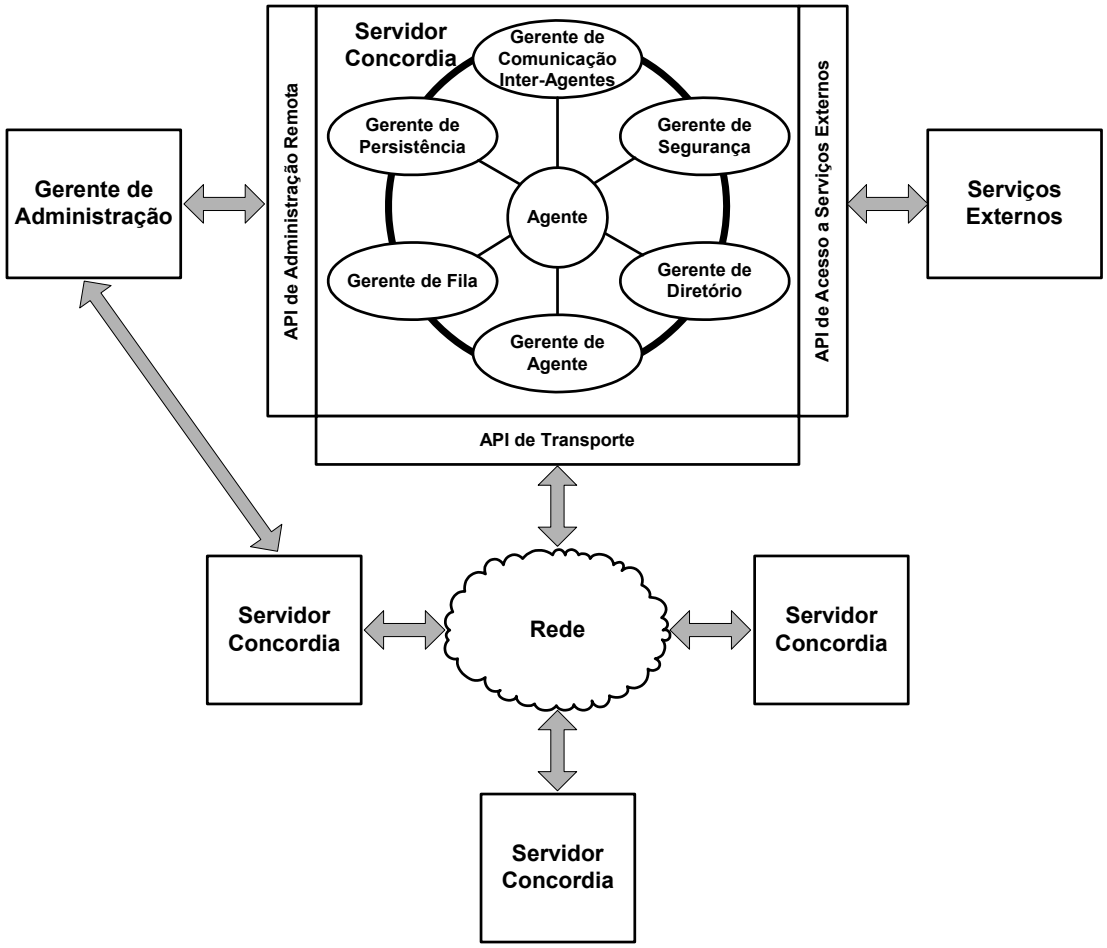


Figura 6: Arquitetura do *Concordia*

O **Gerente de Segurança** protege os recursos oferecidos pelo computador hospedeiro e assegura a integridade dos agentes e de seus dados. Enquanto estão armazenados no cliente, os códigos e dados dos agentes são protegidos contra manipulações não autorizadas através de criptografia; durante o transporte o *Concordia* usa o protocolo SSL (*Secure Socket Layer*) para protegê-los. Para proteger o servidor de atitudes maliciosas por parte dos agentes, o *Concordia* especializa o mecanismo de segurança intrínseco do Java. Cada agente recebe uma identidade, gerada a partir de informações de seu usuário. Com base nessa identidade, são estabelecidas as permissões de acesso aos recursos dos servidores. Essas permissões podem ser alteradas dinamicamente, de acordo com o acréscimo ou decréscimo de confiança no agente.

O **Gerente de Persistência** mantém o estado dos agentes e objetos em trânsito pela rede. Ele torna possível o reinício dos agentes após uma falha no servidor e/ou na rede.

O **Gerente de Comunicação entre Agentes** lida com o registro, a postagem e a notificação de eventos gerados e consumidos pelos agentes. Dois paradigmas são utilizados: eventos distribuídos assíncronos e colaboração. Duas formas de eventos distribuídos são suportadas: eventos selecionados e eventos dirigidos a grupos. No primeiro caso, os agentes selecionam quais tipos de eventos desejam receber; no segundo, todos os eventos são enviados a uma coleção de agentes, conhecido por grupo de eventos, sem nenhum tipo de filtragem. O *framework* de colaboração implementado no *Concordia* facilita a coordenação entre vários agentes que precisam trabalhar em conjunto na solução de "um problema complexo". Os agentes em uma aplicação podem formar uma ou mais unidades de colaboração, conhecida como grupos de agentes. *Concordia* oferece duas classes básicas para a implementação dos agentes colaborativos e dos grupos de agentes, `CollaboratorAgent` e `AgentGroup`, respectivamente. Os grupos de agentes são implementados na forma de objetos distribuídos que exportam uma interface, definida em `AgentGroup`, para os agentes colaborativos. Esses agentes, que estendem a classe `CollaboratorAgent`, possuem referências remotas para os objetos distribuídos que implementam `AgentGroup` e acessam suas operações através da Invocação Remota de Método do Java. As instâncias de `AgentGroup` encaminham as mensagens recebidas de seus membros para os demais participantes dos grupos.

O **Gerente de Fila** é responsável pelo escalonamento e entrega garantida dos agentes entre os diversos Servidores *Concordia*. Ele implementa um transporte confiável sobre uma

rede não confiável. O **Gerente de Diretório** fornece um serviço de nomes que permite a aplicações e agentes encontrarem serviços na rede. O **Gerente de Administração** é responsável pela administração remota de todos os servidores do ADA. Somente um gerente de administração é necessário. Uma **Biblioteca de Classes** inclui, além das classes básicas para a construção dos agentes, todas as demais APIs *Concordia*, como, por exemplo, as **APIs de Administração Remota**, **de Transporte** e **de Acesso a Serviços Externos**. Estas APIs permitem que aplicações possam receber, executar e despachar agentes móveis, além de poderem interagir diretamente com eles. Por exemplo, um desenvolvedor poderá utilizar a API de Acesso a Serviços Externos para adicionar serviços específicos de uma certa aplicação, de forma que esses não precisem migrar com os agentes. Ela também oferece uma maneira de acessar recursos externos à Máquina Virtual Java.

3.6.4 Voyager

Voyager é um Intermediador de Solicitações de Objetos (*Object Request Broker* - ORB) totalmente construído em Java e ampliado com a facilidade de agentes móveis (GLASS, 1999). Ele combina as vantagens dos agentes móveis e da invocação remota de métodos (RMI) a um suporte completo à arquitetura CORBA, incluindo ainda alguns serviços distribuídos tais como serviços de diretório, de persistência e de difusão do tipo *publish-subscribe*. Com o uso do *Voyager* é possível criar aplicações para rede que se utilizam tanto das técnicas de programação tradicionais para ambientes distribuídos quanto o paradigma de agentes móveis. Utilizando-se da sintaxe convencional da linguagem Java é possível construir objetos remotos, enviar-lhes mensagens, e movê-los entre aplicações. Um agente *Voyager* herda suas habilidades de mobilidade e pró-atividade da classe base `Agent`. Uma instância de `Agent` pode se mover por sua própria "vontade" de um computador para outro, deixando para trás, com uma "secretária", um endereço de encaminhamento, de modo que mensagens futuras possam ser enviadas para sua nova localização. Agentes especializados, chamados `Messengers`, são utilizados para entregar as mensagens. As mensagens podem ser síncronas, assíncronas unilaterais (*one-way*) ou futuras, as quais são assíncronas mas retornam um identificador que pode ser usado para recuperar um valor de retorno no futuro.

Da mesma forma que no *Aglets* e no *Concordia*, um itinerário instrui o agente sobre quais localizações ele precisa visitar e sobre quais operações ele deve realizar em cada uma delas. Também, como no *Aglets*, *Grasshopper* e *Concordia*, o *Voyager* serializa o estado do agente em um fluxo serial de bits quando o mesmo se move de uma localização para outra. O *Voyager*, igualmente, possui um gerente de segurança, que restringe as operações que um agente pode realizar.

3.7 Considerações Finais

Os principais pontos evidenciados neste capítulo são os seguintes:

A linguagem de programação mais utilizada atualmente na construção de agentes móveis é a linguagem Java.

Todo agente móvel é composto por uma parte comum e por uma parte específica. A parte comum é fornecida por uma classe abstrata própria de cada sistema de agentes móveis. A parte específica deve ser implementada pelo programador do agente na forma de uma ou mais classes, sendo que a classe principal deve ser uma especialização da classe abstrata fornecida pelo sistema de agentes móveis em uso.

Por intermédio dos métodos da classe abstrata comum, um agente pode obter uma referência para a agência e em seguida para um dos serviços oferecidos por ela.

É tarefa do programador avaliar quais variáveis de instância devem fazer parte do estado dos dados de um agente.

O tamanho do estado de dados de um agente tem um grande impacto na duração do processo de migração. Portanto, somente um conjunto mínimo de variáveis de instância deve fazer parte dele.

Se uma variável de instância estiver vinculada semanticamente a um ambiente de execução, seu valor poderá tornar-se inválido após a migração do agente. Esse problema é freqüente quando o agente utiliza serviços oferecidos pelas agências. Nesse caso, a variável em questão deve ser definida como transiente, e o agente deverá reinstanciá-la e reinicializá-la após cada migração.

Se determinada funcionalidade for comum a vários agentes, seu código pode ser previamente instalado nas agências de destino e oferecido aos agentes na forma de um serviço.

Os agentes criados por todos os sistemas de agentes móveis atuais não possuem nenhuma habilidade inerente de inteligência. Eles não possuem mecanismos de inferência, de aprendizado ou qualquer outra técnica de IA. Portanto, todos eles poderão beneficiar-se do *Framework de Inteligência* proposto neste trabalho.

Capítulo 4

Sistemas de Raciocínio Lógico

4.1 Introdução

Este capítulo apresenta os fundamentos conceituais do mecanismo de IA que será incorporado aos agentes móveis de acordo com o *framework* proposto: o mecanismo de inferência com encadeamento progressivo baseado em regras. Um agente móvel equipado com esse tipo de mecanismo é um tipo de **sistema de raciocínio lógico**. Os componentes genéricos que constituem um sistema de raciocínio, bem como seus relacionamentos, são descritos na Seção 4.2. A Seção 4.3 apresenta a **Lógica de Primeira Ordem**, a linguagem de representação de conhecimento mais estudada e mais utilizada na prática. A Seção 4.4 discute o processo de inferência nessa lógica e as simplificações utilizadas para tornar o problema tratável. A Seção 4.5 discorre sobre os aspectos práticos relativos à construção de um mecanismo de inferência com encadeamento progressivo baseado em regras. A Seção 4.6 descreve o algoritmo ***Rete***, um algoritmo bastante eficiente para tratar de uma das principais dificuldades práticas dos mecanismos de inferência com encadeamento progressivo: a procura em uma base de conhecimento por informações similares.

4.2 Definição e Componentes

Um **sistema de raciocínio lógico** é um sistema no qual o conhecimento é representado explicitamente e utilizado para deduzir novas informações. Tais sistemas possuem dois componentes principais, uma **base de conhecimento** (BC) e um **mecanismo de inferência**. A

base de conhecimento armazena **sentenças** expressas em uma **linguagem para representação do conhecimento**. Cada sentença corresponde a uma declaração sobre o universo de discurso. Dois aspectos centrais de uma linguagem possibilitam a representação do conhecimento: sua **sintaxe**, que especifica as estruturas gramaticais válidas para as sentenças, e sua **semântica**, que determina como essas sentenças se relacionam com os fatos do universo de discurso, ou seja, o significado de cada uma delas nesse universo. O mecanismo de inferência é responsável por determinar as conseqüências lógicas das sentenças presentes na BC. Esse mecanismo realiza essa tarefa através de um ou mais **procedimentos de inferência** que, sistematicamente, aplicam à base de conhecimento um conjunto de **regras de inferência**. Uma linguagem para representação de conhecimento e um conjunto de regras de inferência constituem os elementos essenciais de uma **lógica formal**, ou simplesmente de uma **lógica**.

4.3 Lógica de Primeira Ordem

A lógica para a representação do conhecimento mais estudada e utilizada até o momento é a **Lógica de Primeira Ordem**, também conhecida com **Cálculo de Predicados**. Essa lógica assume que o universo de discurso é composto por **objetos** que se **relacionam** entre si e que possuem certas **propriedades**. Ela é chamada de **primeira ordem** porque suas sentenças podem expressar declarações sobre propriedades e relações entre objetos específicos (individualização) ou entre classes inteiras de objetos (generalização), os quais representam as entidades de "primeira ordem" do universo de discurso. Por outro lado, não se pode utilizar, em uma sentença, propriedades e relacionamentos genéricos. Lógicas de ordem superior, que permitem tal generalização, possuem maior expressividade do que a lógica de primeira ordem. Contudo, até agora, ainda existe muito pouca compreensão sobre como raciocinar, de forma efetiva, com o conhecimento representado em tais lógicas, sendo o problema geral sabidamente indecidível (RUSSELL; NORVIG, 1995).

Por outro lado, as lógicas modais, utilizadas nas teorias formais de agentes, embora geralmente sejam extensões da lógica de primeira ordem, tendem a ser mais elaboradas, contendo, tipicamente, muitos operadores que interagem entre si de forma pouco perceptível. Poucos trabalhos foram realizados sobre as teorias subjacentes a essas lógicas. Até que seus

princípios gerais e suas limitações sejam perfeitamente compreendidos, a utilização de tais lógicas deverá avançar lentamente (WOOLDRIGE; JENNINGS, 1995). Além disso, a adoção dessas lógicas como linguagens de representação de conhecimento esbarra no problema da construção de algoritmos de inferência viáveis para uso em sistemas que devem operar no mundo real e sujeitos a restrições temporais. Em primeiro lugar, assim como a lógica de primeira ordem completa não é decidível, suas extensões modais também tendem a não sê-lo. Por outro lado, a construção de algoritmos tratáveis de manipulação de símbolos para essas lógicas, ou seja, algoritmos que produzam, com certeza, resultados úteis em um intervalo de tempo (fixo) aceitável também não é tarefa fácil.

Neste trabalho, decidiu-se pela utilização de uma versão restrita da lógica de primeira ordem acrescida de informações procedimentais, conhecida por **Regras Evento-Condição-Ação Situadas** (ECAS). Além de representar um bom compromisso entre compreensão, expressividade e tratabilidade, essa forma de representação é decidível e largamente empregada na prática, sendo considerada atualmente um dos pilares da ***Web Semântica*** e do **Comércio Eletrônico** (BOLEY, 2002).

As subseções seguintes descrevem a *Lógica de Primeira Ordem* completa. As restrições impostas e as informações procedimentais incluídas nas regras ECAS são discutidas nas Seções 4.4 e 4.5.

4.3.1 A Sintaxe

Objetos são representados por **termos**, que podem ser de três tipos: **constantes**, **variáveis** ou **funções**. Cada **constante** nomeia exatamente um objeto existente no universo de discurso. Uma **variável** representa um conjunto de objetos, podendo ser substituída, sempre que aparecer, por qualquer objeto desse conjunto. Cada **função** representa uma relação entre um conjunto de objetos e um único objeto. Símbolos funcionais podem ser utilizados para referir-se a objetos específicos, sem que seja necessário o uso de seus nomes. As propriedades dos objetos e suas relações são representadas por **símbolos predicativos** ou, simplesmente, **predicados**. Uma **sentença atômica**, ou simplesmente um **átomo**, pode ser construída de dois modos: primeiro, por um predicado aplicado a um ou mais termos e, segundo, por uma **igualdade** entre termos. No primeiro caso, a sentença é considerada verdadeira se a relação repre-

sentada pelo predicado for válida para os objetos representados pelos termos; no segundo, se os dois termos se referirem ao mesmo objeto. Um **literal** corresponde a uma sentença atômica ou à sua negação lógica, o primeiro chamado de **literal positivo** e o último de **literal negativo**. Uma **sentença composta** (ou **molecular** ou simplesmente um **composto**) pode ser construída através da ligação de literais por meio de **conectivos**. O uso de **quantificadores** permite expressar propriedades e relações de classes inteiras de objetos, tornando possível a declaração de sentenças genéricas. A lógica de primeira ordem leva esse nome justamente porque permite quantificar os objetos mas não os seus relacionamentos e funções. A Tabela 3 ilustra a sintaxe da lógica de primeira ordem usando a notação de Backus-Naur (*Backus-Naur Form* – BNF) (RUSSELL; NORVIG, 1995).

Tabela 3: Sintaxe da lógica de primeira ordem

Sentença	→	Átomo
	|	Composto
	|	Negação
	|	Quantificador Variável,… Variável Sentença
	|	(Sentença)
Composto	→	Sentença Conectivo Sentença
Negação	→	¬Sentença
Literal	→	Átomo | ¬Átomo
Átomo	→	Igualdade | Relação
Igualdade	→	Termo = Termo
Relação	→	Predicado(Termo,…)
Termo	→	Função(Termo,…) | Constante | Variável
Conectivo	→	∧ | ∨ | → | ↔
Quantificador	→	∀ | ∃
Constante	→	A | X1 | João | …
Variável	→	a | x | s | …
Predicado	→	Após | TemCor | chovendo | …
Função	→	Mãe | PernaEsquerdaDe | co-seno | …

Pode-se observar na Tabela 3 que os símbolos que representam constantes iniciam-se por letras maiúsculas enquanto os que representam variáveis iniciam-se por letras minúsculas. Símbolos de funções e de predicados podem começar tanto por letras maiúsculas quanto por letras minúsculas. Um termo que não contém nenhuma variável é denominado **termo básico** ou **termo fechado**. Da mesma forma, uma sentença que não contém nenhuma variável é chamada de **sentença básica** ou **sentença fechada**.

4.3.2 A Semântica

A semântica da lógica de primeira ordem é definida pelo significado atribuído pelo usuário aos símbolos de constante, de funções e de predicados que formam as sentenças atômicas, isto é, pela interpretação fornecida para esses símbolos, e pelo significado dos conectivos e quantificadores lógicos. Portanto, o significado de uma sentença composta é derivado do significado de suas partes.

Os significados dos conectivos e quantificadores presentes na Tabela 3 estão definidos na Tabela 4.

Tabela 4: Significado dos conectivos e quantificadores

Item	Significado
$\neg$	Negação
$\wedge$	Conjunção (E lógico)
$\vee$	Disjunção (OU lógico)
$\rightarrow$	Implicação (Se … então)
$\leftrightarrow$	Equivalência (Se e somente se)
$\forall$	Quantificador Universal (Para todo)
$\exists$	Quantificador Existencial (Existe)

Uma interpretação mais precisa dos conectivos pode ser dada através de suas tabelas de valores, como mostrado abaixo, onde α e β correspondem a sentenças atômicas cujos valores são dependentes da interpretação atribuídas a seus elementos, F corresponde a **falso** e V a **verdadeiro**.

Tabela 5: Tabelas de valores dos conectivos

α	β	$\neg\alpha$	$\alpha \wedge \beta$	$\alpha \vee \beta$	$\alpha \rightarrow \beta$	$\alpha \leftrightarrow \beta$
F	F	V	F	F	V	V
F	V	V	F	V	V	F
V	F	F	F	V	F	F
V	V	F	V	V	V	V

Somente a título de ilustração, apresentam-se, a seguir, alguns exemplos de declarações em lógica de primeira ordem.

"Todo humano é mortal":

$\forall x\ [Humano(x) \rightarrow Mortal(x)]$

"Alguns gatos são amarelos.":

$\exists x\ [Gato(x) \wedge Amarelo(x)]$

"Nenhuma baleia é peixe.":

$\forall x\ [Baleia(x) \rightarrow \neg Peixe(x)]$

"Todo humano é gerado por alguma mulher.":

$\forall x, \exists y\ [Humano(x) \rightarrow Mulher(y) \wedge (y = Mãe(x))]$

"Há pelo menos dois senadores.":

$\exists x,y\ [Senador(x) \wedge Senador(y) \wedge \neg(x = y)]$

4.3.3 Regras de Inferência

Uma **regra de inferência** é um dispositivo que permite a dedução de uma **conclusão** lógica a partir de um conjunto de sentenças denominadas **premissas**. Uma regra de inferência é dita **consistente** (*sound*) se produzir uma conclusão verdadeira para todos os casos nos quais as premissas forem verdadeiras.

Antes da apresentação das regras de inferência para a lógica de primeira ordem, é necessário introduzir-se alguns conceitos e notações. Geralmente, utilizam-se letras gregas (α, β, etc) para representar sentenças e o símbolo $\vdash$ para indicar dedução. Escreve-se $\Delta \vdash \gamma$ para indicar que a sentença γ é *formalmente dedutível* de um conjunto de premissas Δ. Deste modo,

a expressão $\alpha, \beta \vdash \gamma$ significa que "γ é formalmente dedutível de α e β". A notação SUBST(θ,α) denota a sentença resultante da aplicação da lista de substituição θ à sentença α. Por exemplo:

$$\text{SUBST}(\{x/\text{Paulo}, y/\text{Luís}\}, \text{Pai}(x,y)) = \text{Pai}(\text{Paulo},\text{Luís})$$

Diz-se que duas sentenças podem ser **unificadas** quando for possível encontrar uma lista de substituição θ que as torne idênticas, ou seja:

$$\text{SUBST}(\theta,\alpha) = \text{SUBST}(\theta,\beta)$$

Neste caso, θ é chamado de **unificador** das duas sentenças.

As regras de inferência mais utilizadas para dedução na lógica de primeira ordem são as seguintes:

1) **Eliminação da Conjunção**: De uma conjunção de sentenças fechadas (sem variáveis) pode-se inferir qualquer uma delas:

$$\alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_n \vdash \alpha_i$$

2) **Introdução da Conjunção**: De uma lista de sentenças fechadas pode-se inferir a conjunção das mesmas:

$$\alpha_1, \alpha_2, \ldots, \alpha_n \vdash \alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_n$$

3) **Introdução da Disjunção**: De uma sentença fechada, pode-se inferir sua disjunção com quaisquer outras sentenças fechadas:

$$\alpha_i \vdash \alpha_1 \vee \alpha_2 \vee \ldots \vee \alpha_i \vee \ldots \vee \alpha_n$$

4) **Eliminação da Dupla Negação**: Para qualquer sentença α vale:

$$\neg\neg\alpha \vdash \alpha$$

5) **Eliminação Universal**: Para qualquer sentença α, variável $\boldsymbol{v}$ e termo fechado $\boldsymbol{g}$ vale:

$$\forall v\ \alpha \vdash \text{SUBST}(\{v/g\},\alpha)$$

Por exemplo, da sentença

$$\forall x \; \texttt{Gosta(x,Banana)}$$

utilizando-se a substituição

$$\{x/Paulo\}$$

pode-se concluir

$$\texttt{Gosta(Paulo,Banana)}$$

6) **Eliminação Existencial**: Para qualquer sentença α, variável ***v*** e constante ***K*** que não apareça em nenhuma outra sentença da base de conhecimento vale:

$$\exists K \; \alpha \vdash \text{SUBST}(\{v/K\},\alpha)$$

Por exemplo, de

$$\exists x \; \texttt{Comeu(x,Banana)}$$

pode-se inferir

$$\texttt{Comeu(Paulo,Banana)}$$

desde que o símbolo `Paulo` não apareça em nenhuma outra sentença presente na base de conhecimento.

7) **Introdução Existencial**: Para qualquer sentença α, variável ***v*** que não ocorra em α e termo fechado *g* vale:

$$\alpha \vdash \exists v \; \text{SUBST}(\{g/v\}, \alpha)$$

Por exemplo, de

$$\texttt{Gosta(Paulo,Banana)}$$

infere-se que

$$\exists x \; \texttt{Gosta(x,Banana)}$$

8) **Modus Ponens**: Para as sentenças atômicas α_i, β_i e γ para as quais existe uma lista de substituição θ tal que $\text{SUBST}(\theta,\alpha_i) = \text{SUBST}(\theta,\beta_i)$ para todo *i* vale:

$$\alpha_1, \; \alpha_2, \; \ldots, \; \alpha_n, \; \beta_1 \wedge \beta_2 \wedge \ldots \wedge \beta_n \rightarrow \gamma \vdash \text{SUBST}(\theta,\gamma)$$

Por exemplo, das sentenças:

Humano(Paulo) e

$\forall x$ [Humano(x) $\rightarrow$ Mortal(x)]

pode-se inferir

Mortal(Paulo)

9) **Resolução (Forma Normal Implicativa)**: Para as sentenças atômicas $\boldsymbol{\alpha_i}$, i ≤ m, $\boldsymbol{\beta_j}$, j ≤ n, $\boldsymbol{\gamma_k}$, k ≤ p, $\boldsymbol{\lambda_l}$, l ≤ q, para os quais existe uma lista de substituição θ tal que SUBST(θ, α_i) = SUBST(θ, λ_k) vale:

$$[\alpha_1 \wedge \ldots \wedge \alpha_i \wedge \ldots \wedge \alpha_m \rightarrow \beta_1 \vee \ldots \vee \beta_n],$$

$$[\gamma_1 \wedge \ldots \wedge \gamma_p \rightarrow \lambda_1 \vee \ldots \vee \lambda_k \vee \ldots \vee \lambda_q])$$

$$\text{Subst}(\theta, [\alpha_1 \wedge \ldots \wedge \alpha_{i-1} \wedge \alpha_{i+1} \vee \ldots \wedge \alpha_m \wedge \gamma_1 \wedge \ldots \wedge \gamma_p \rightarrow$$

$$\beta_1 \vee \ldots \vee \beta_n \vee \lambda_1 \vee \ldots \vee \lambda_{k-1} \vee \lambda_{k+1} \vee \ldots \vee \lambda_q])$$

Por exemplo, das sentenças:

$\forall x$ [**Humano(x)** $\rightarrow$ Mortal(x)] e $\forall x$ [Rei(x) $\rightarrow$ **Humano(x)**]

pode-se inferir

$\forall x$ [Rei(x) $\rightarrow$ Mortal(x)].

10) **Resolução (Forma Normal Conjuntiva)**: Para os literais $\boldsymbol{\alpha_i}$, i ≤ m e $\boldsymbol{\beta_j}$, j ≤ n para os quais existe uma lista de substituição θ tal que SUBST(θ, α_i) = SUBST(θ, $\neg\beta_j$) vale:

$$[\alpha_1 \vee \ldots \vee \alpha_i \vee \ldots \vee \alpha_m], \ [\beta_1 \vee \ldots \vee \beta_j \vee \ldots \vee \beta_n]$$

$$\text{Subst}(\theta, [\alpha_1 \vee \ldots \vee \alpha_{i-1} \vee \alpha_{i+1} \vee \ldots \vee \alpha_m \vee \ldots \vee \beta_1 \vee \ldots \vee \beta_{j-1} \vee$$

$$\beta_{j+1} \vee \ldots \vee \beta_n])$$

Por exemplo, das sentenças:

$\forall x$ [$\neg$**Rei(x)** $\vee$ Humano(x)] e $\forall x$ [$\neg$Humano(x) $\vee$ **Mortal(x)**]

pode-se inferir:

$$\forall x\ [\neg Rei(x) \vee Mortal(x)].$$

Do ponto de vista computacional, a grande vantagem das regras formais de inferência é o fato de que elas podem ser utilizadas para derivar conclusões válidas independentemente do conhecimento do significado das sentenças envolvidas. Essa habilidade é essencial para o raciocínio automatizado.

4.4 Procedimentos de Inferência

Um procedimento de inferência é um processo capaz de deduzir novas sentenças através da aplicação sistemática de regras de inferência a sentenças já existentes numa base de conhecimento; ou seja, é um processo de controle da atividade de "raciocínio" em uma certa lógica.

Na verdade, um procedimento de inferência pode realizar duas coisas: dada uma base de conhecimento (BC), ele pode gerar novas sentenças que, supostamente, são conseqüências lógicas dessa base; ou, dada uma base de conhecimento e uma sentença α, ele pode verificar se α é ou não conseqüência lógica dessa base. O fato de uma sentença α ser conseqüência lógica da BC pode ser expresso por:

$$BC \models \alpha$$

Por outro lado, o fato de uma sentença α ser derivada a partir de BC pelo procedimento de inferência "P" pode ser representado por:

$$BC \vdash_P \alpha$$

Um procedimento de inferência que produz somente conclusões verdadeiras a partir de premissas também verdadeiras é dito **consistente**. Um procedimento de inferência é **completo** se ele pode derivar todas as conclusões verdadeiras decorrentes de um conjunto de premissas.

Existem dois tipos básicos de procedimentos de inferência: procedimentos **dirigidos por objetivos** e procedimentos **dirigidos por dados**. Em um procedimento dirigido por objetivos o processo de inferência é iniciado quando uma consulta é realizada à base de conhecimento, ou seja, quando se deseja saber se uma determinada sentença é ou não conseqüência lógica das sentenças presentes na base. Neste caso, o procedimento aplica as regras de inferência de modo a encontrar todas as possíveis respostas. Por outro lado, em um procedimento dirigido por dados, a inclusão de uma nova sentença à base de conhecimento dispara o processo de inferência, que aplica as regras de inferência de modo a descobrir todas as conseqüências lógicas dessa adição.

Para que um procedimento de inferência seja eficiente, ele deve fazer uso de poucas, mas poderosas, regras de inferência, que realizem em um único passo o que seria obtido com a aplicação de várias outras regras de menor poder. As regras de inferência *Resolução* e *Modus Ponens* permitem a construção de tais procedimentos, com a utilização de uma única regra.

Qualquer sentença em lógica de primeira ordem (sem igualdade) pode ser reescrita nas formas normais exigidas para a aplicação da regra *Resolução*. Em teoria, pode-se implementar um procedimento de inferência completo que aplique essa regra a uma base de conhecimento composta por sentenças escritas em lógica de primeira ordem. Contudo, essa regra de inferência não pode ser utilizada para gerar todas as conseqüências lógicas decorrentes de um conjunto de sentenças; ela somente pode ser usada para verificar se uma dada sentença é ou não conseqüência lógica desse conjunto. Em outras palavras, *Resolução* pode ser utilizada somente na construção de procedimentos completos de inferência dirigidos por objetivos. Destes, o mais conhecido é a **refutação**, também chamada de prova **por contradição** ou **prova por redução ao absurdo**. Para provar que uma sentença é verdadeira, esse procedimento acrescenta à base de conhecimento a negação da sentença e, por aplicações sucessivas da *Resolução*, tenta chegar a uma contradição. No entanto, embora se saiba que a aplicação sistemática e repetitiva dessa regra irá encontra uma prova se ela existir, não existem garantias da eficiência do processo e, pior ainda, se a prova não existir, o procedimento poderá não parar nunca. Em resumo, assim como todo procedimento de inferência para a lógica de primeira ordem completa, ele é **semidecidível** (RUSSELL; NORVIG, 1995).

A representação do conhecimento, utilizando somente um subconjunto da lógica de primeira ordem, permite a construção de procedimentos de inferência mais simples e eficientes.

A forma mais popular de representação do conhecimento em lógica de primeira ordem restrita é a **regra** (implicação) na forma de **cláusulas de Horn**. Existem várias razões para isso: elas são de fácil entendimento, mesmo por pessoas de áreas não técnicas; muitos tipos de conhecimento são, por natureza, expressos dessa forma; cada regra representa uma "parcela" de conhecimento, em grande parte independente das demais; e sua uniformidade permite a criação de mecanismos de inferência simples, eficientes e decidíveis (RUSSELL; NORVIG, 1995; RICH; KNIGHT, 1994).

Uma cláusula de Horn é uma implicação lógica com o seguinte formato:

$$P_1 \wedge P_2 \wedge P_3 \wedge \ldots \wedge P_n \rightarrow Q$$

onde $\mathbf{P_i}$ e **Q** são **sentenças atômicas**. O lado esquerdo de uma regra é conhecido como **expressão antecedente**, e seu lado direito, como **expressão conseqüente**. É interessante notar que uma sentença atômica isolada também é uma cláusula de Horn, com n = 1 e $\mathbf{P_1}$ = Verdade, o que leva a $\texttt{Verdade} \rightarrow \texttt{Q}$, o que é equivalente à **Q**.

Para esse tipo de regra existem procedimentos de inferência, baseados em *Modus Ponens*, que, além de serem decidíveis, apresentam tempo de computação polinomial (RUSSELL; NORVIG, 1995; RICH; KNIGHT, 1994). Por outro lado, nem toda base de conhecimento pode ser escrita como uma coleção de cláusulas de Horn. Em outras palavras, um procedimento de inferência que utiliza *Modus Ponens* é **incompleto**. Apesar dessa desvantagem, é enorme o número de sistemas de raciocínio lógico que utilizam procedimentos de inferência baseados na aplicação de *Modus Ponens* a uma base de conhecimento na forma de regras. Tais sistemas, que utilizam regras como forma básica de representação do conhecimento, são chamados **sistemas de raciocínio baseados em regras**.

Atualmente, a importância dos sistemas de raciocínio baseados em regras foi reafirmada com o surgimento da *Web Semântica*. Trata-se de uma evolução da Web atual na qual busca-se uma forma de expressar informações que permita que computadores compartilhem e explorem o conhecimento existente na Web, através da associação de significado aos conteúdos dos hiperdocumentos e/ou do estabelecimento de uma camada de informações que sejam compreensíveis por máquinas (BERNERS-LEE et al., 2001). Nesse contexto, tanto as regras

quanto os procedimentos de inferência tornaram-se tópicos importantes de projeto e de pesquisa (BOLEY, 2002).

Os dois procedimentos de inferência mais utilizados com *Modus Ponens* são o **encadeamento progressivo** e o **encadeamento regressivo**. O primeiro é um procedimento dirigido por dados e o segundo um procedimento dirigido por objetivos.

Dentre eles, a abordagem do encadeamento progressivo é a mais adequada para a utilização com os agentes móveis, visto que tais agentes estarão, na maior parte do tempo, distantes de seus usuários e precisarão determinar suas ações com base no estado atual de sua base de conhecimento e nas novas percepções recebidas de seus ambientes.

4.5 Mecanismos de Inferência

O mecanismo de inferência é o componente de um sistema de raciocínio lógico encarregado de executar o(s) procedimento(s) de inferência(s) sobre a base de conhecimento. Um mecanismo de inferência que utiliza um procedimento com encadeamento progressivo é chamado de **mecanismo de inferência com encadeamento progressivo**, da mesma forma, um que utiliza um procedimento com encadeamento regressivo é chamado de **mecanismo de inferência com encadeamento regressivo.**

No caso específico de um sistema de raciocínio lógico baseado em regras, a base de conhecimento é composta por um conjunto de regras, normalmente fixo, e por um conjunto de **fatos** (literais positivos sem variáveis), que varia continuamente. Uma vez que tanto o número de regras quanto o número de fatos presentes na BC pode ser grande, a forma como os mecanismos de inferência são implementados tem grande impacto sobre o desempenho do sistema como um todo.

Nem sempre os fatos e as regras estão armazenadas na BC em um formato que é adequado à manipulação por parte do mecanismo de inferência. É até mesmo desejável que o formato das sentenças na BC seja o mais independente possível do mecanismo de inferência utilizado, garantindo, desse modo, um alto grau de reutilização do conhecimento capturado. Geralmente, é necessário "traduzir" as sentenças presentes na BC para o formato interno de representação utilizado pelo mecanismo de inferência. Usualmente, esse novo formato possui

informações adicionais que tornam mais eficiente a tarefa fundamental de recuperar sentenças que satisfaçam certas condições, como, por exemplo, encontrar um literal que possa ser unificado com uma consulta, ou uma implicação que possua um certo literal na sua expressão antecedente.

Na prática, para o caso específico do mecanismo de inferência de interesse neste trabalho, ou seja, o de encadeamento progressivo, durante a inicialização do sistema, as regras são transferidas da BC para o mecanismo e compiladas para sua forma interna de representação e então armazenadas em uma **memória de regras**. Por sua vez, os fatos presentes inicialmente na BC são transferidos para a **memória de trabalho** do mecanismo.

No restante deste texto, utiliza-se, por simplicidade, somente o termo **mecanismo de inferência** ao referir-se ao mecanismo de inferência com encadeamento progressivo dirigido por regras.

4.5.1 Ciclo de Inferência

De acordo com Russell e Norvig (1995), após a inicialização, durante a etapa de execução, quando um fato é acrescentado ou removido da memória de trabalho, um **ciclo de inferência** é disparado. Um ciclo de inferência possui três fases: **fase de casamento** (*match*), **fase de resolução de conflitos** e **fase de aplicação** (da regra). Na fase de casamento, o mecanismo encontra um subconjunto de regras cujos literais na expressão antecedente sejam todos satisfeitos pelos fatos presentes na memória de trabalho. Essas regras são ditas **ativadas** e o subconjunto formado por elas é chamado de **conjunto de conflito**. Após essa fase, alguns sistemas passam direto para a fase de aplicação, executando as ações associadas a todas as regras presentes no conjunto de conflito, enquanto outros entram em uma fase de resolução de conflitos, na qual utilizam uma **estratégia de resolução de conflito** para selecionar apenas uma regra para ser executada na fase de aplicação. Em qualquer dos casos, a execução de uma ação pode disparar um novo ciclo de inferência.

4.5.2 Sensores e Atuadores

Para sistemas lógicos puros, a única ação possível é a atualização da memória de trabalho, pela adição de um novo fato ou pela remoção de um fato existente. No entanto, na prática, muitos sistemas de raciocínio baseados em regras permitem a chamada de procedimentos externos, tanto da expressão antecedente, quanto da expressão conseqüente de uma regra. Tais procedimentos correspondem, respectivamente, a **sensores** e a **atuadores**. Sensores e atuadores expandem, de forma significativa, a capacidade de tais sistemas, tornando-os capazes de monitorar e agir sobre o seu ambiente. Desse modo, são possíveis ações, tais como imprimir uma mensagem numa tela ou enviar um e-mail. Além disso, normalmente, tais sistemas operam sobre regras mais amplas que as sentenças de Horn, permitindo em sua expressão antecedente um literal negativo e em sua expressão conseqüente uma conjunção de literais. Embora, teoricamente, também seja possível a utilização de uma representação que permita disjunções de literais ("OU") na expressão antecedente, na prática isso não ocorre, pois, dependendo da implementação do mecanismo de inferência, o custo computacional, em termos de tempo e espaço de memória, para converter essa representação para a representação interna, sem as disjunções, pode ser exponencial (GROSOF; LABROU, 1999). Portanto, na prática, as regras tomam a forma:

$$P_1 \wedge P_2 \wedge P_3 \wedge \ldots \wedge P_n \rightarrow A_1 \wedge A_2 \wedge A_3 \wedge \ldots \wedge A_m$$

Na expressão anterior, cada Pi pode ser um literal positivo ou negativo e cada A_i representa uma ação que deve ser realizada quando todos os P_i forem satisfeitos.

4.5.3 Negação por Falha

A veracidade de um literal negativo é avaliada, na prática, utilizando-se a estratégia de **negação por falha**, que assume a **hipótese do mundo fechado**. Segundo essa hipótese, todas as declarações relevantes e verdadeiras estão contidas na base de conhecimento ou podem ser derivadas das declarações lá contidas. Qualquer declaração que não esteja presente pode, portanto, ser assumida como falsa. A *negação por falha*, geralmente, é indicada pelo operador "~", para diferenciá-la da negação clássica, indicada por "¬". Intuitivamente, um

literal negativo – `~Predicado(Termo,...)` – presente na expressão antecedente é considerado verdadeiro se o valor verdade de `Predicado(Termo,...)` for **falso** ou **desconhecido**. A introdução da *negação por falha* transforma o sistema de raciocínio baseado em regras em um **sistema não-monotônico**, isto é, num sistema onde a aplicação de uma regra pode impedir a aplicação posterior de uma outra regra que também poderia ter sido aplicada quando a primeira foi escolhida, impondo a necessidade do uso de técnicas de remoção de sentenças da BC, para mantê-la consistente.

4.5.4 Não Recursividade de Predicados

Na presença da *negação por falha,* normalmente é imposta uma restrição de expressividade denominada **não-recursividade de predicados** (*predicate-acyclic expressive restriction*) (GROSOF; LABROU, 1999). Segundo essa restrição, não deve existir uma dependência cíclica entre os predicados através das regras na BC, isto é, em cada regra na BC os predicados na expressão conseqüente possuem uma dependência direta dos predicados presentes na expressão antecedente. Essa dependência pode ser modelada na forma de um grafo onde cada nó representa um predicado e cada ramo uma relação de dependência. A reunião dos grafos obtidos para todas as regras na BC constitui um **Grafo de Dependência de Predicados**. Os caminhos fechados desse grafo definem quais predicados são dependentes de outros predicados. Para que a *restrição de não-recursividade* seja satisfeita, não devem existir caminhos fechados no Grafo de Dependência de Predicados. O atendimento a esta condição pode ser testado sintaticamente por algoritmos relativamente simples e com um baixo custo computacional (GROSOF et al., 1999).

4.5.5 Representação de Eventos

Além da *negação por falha*, outra extensão muito comum é a inclusão de representações de **eventos** na expressão antecedentes das regras, transformando-as de declarações do tipo "*Se* condição *então* ação", em expressões "*Quando* evento, se *condição* então *ação*".

Neste caso, as regras tomam a forma:

$$E \wedge P_1 \wedge \ldots \wedge P_m \wedge S_1 \wedge \ldots \wedge S_n \rightarrow Q_1 \wedge \ldots \wedge Q_k \wedge A_1 \wedge \ldots \wedge A_r$$

onde:

$m, n, k, r \geq 0$

$E \rightarrow$ `NomeEvento(Termo,...);`

$P_i \rightarrow$ `Predicado(Termo,...)` | `~Predicado(Termo,...);`

$Q_i \rightarrow$ `Predicado(Termo,...)` | `~Predicado(Termo,...);`

$S_i \rightarrow$ `NomeSensor(Termo,...)` | `~NomeSensor(Termo,...);`

$A_i \rightarrow$ `NomeAtuador(Termo,...)`

Algumas observações sobre a notação acima se fazem necessárias. Uma regra está associada a um único evento. Os átomos fechados que representam os eventos – `NomeEvento(Constante,...)` – são mantidos na memória de trabalho da BC somente durante o **episódio de inferência** que eles disparam. Um episódio de inferência é composto por um ou mais ciclos de inferência. Os símbolos P_i e Q_i indicam declarações lógicas puras, isto é, literais que não estão associados a nenhum procedimento externo ou a um evento. A introdução de predicados que representam eventos, sensores e atuadores impõe uma direção de uso das regras, da expressão antecedente para a expressão conseqüente. O operador "~", na expressão antecedente, indica, como já descrito, a negação por falha; porém, na expressão conseqüente, ele está sendo utilizado para indicar que o átomo ao qual ele está associado deve ser removido da BC quando todas as condições da expressão antecedente forem satisfeitas.

4.5.6 Exclusão de Funções e Número Finito de Variáveis

Para que a representação em forma de regras seja computacionalmente tratável, mesmo na presença da negação por falhas, é comum em programação lógica a introdução de duas restrições práticas (GROSOF, 1999). A primeira delas, conhecida por **Datalog**, impede a existência de símbolos funcionais como parte da lista de termos de um predicado, uma vez que eles não são essenciais para a expressividade da representação e podem introduzir recursividades. Normalmente, é simples reformular um conjunto de regras de modo a substituir

uma função lógica “F” de k (>0) termos por um predicado “FP” de k+1 termos, onde o último termo corresponde ao resultado da aplicação da função aos termos anteriores. A segunda restrição prática bastante comum é a existência de um limite superior finito para o número de variáveis em uma regra. Dadas essas duas hipóteses, o processo de inferência com regras *Evento-Condição-Ação* apresenta um tempo de computação polinomial (GROSOF, 1999).

4.5.7 Algoritmos de Unificação

O desempenho de um mecanismo de inferência também depende sobremaneira das estruturas de dados empregadas na implementação da representação interna de sua BC e da eficiência do algoritmo de casamento utilizado.

Para evidenciar a importância desses aspectos, pode-se considerar a seguinte situação: em uma possível representação interna os literais poderiam ser implementados como uma lista de *strings*, com o predicado seguido dos termos, e as regras como uma lista de literais, iniciando-se com os antecedentes e terminando com os conseqüentes. Dessa forma, as memórias de regras e de trabalho poderiam ser simplesmente vetores contendo tais representações. Neste caso, um modo imediato de tratar o problema de casamento seria utilizar um **algoritmo de unificação** (RUSSELL; NORVIG, 1995) capaz de casar dois literais, ambos podendo possuir variáveis, para se comparar todos os antecedentes de cada uma das regras com todos os fatos presentes na memória de trabalho. Essa abordagem é ineficiente por duas razões. Primeiro, o número de chamadas ao algoritmo de unificação é muito grande. Se a memória de trabalho possuir F fatos e a memória de regras R regras, cada uma com um número médio de antecedentes A, então, o algoritmo de casamento deveria realizar $RF\,(F^A-1)\,/\,(F-1)$ unificações por ciclo de inferência. Em outras palavras, a complexidade computacional seria da ordem de $O(R \cdot F^A)$ (FRIEDMAN-HILL, 2001), o que significa que, para sistemas úteis, esse número cresceria muito rapidamente. Segundo, a maioria das unificações realizadas em um ciclo de inferência apresentaria os mesmos resultados do ciclo imediatamente anterior. Isso se deve ao fato de que, para a maioria dos sistemas, embora novos fatos estejam sendo freqüentemente adicionados à memória de trabalho, e outros tantos removidos, a percentagem de alterações realizadas em um único ciclo é, geralmente, pequena. Portanto, a repetição de todas as unificações em cada ciclo é um desperdício de tempo. A utilização de um esquema de indexa-

ção mais inteligente, por exemplo, implementar as memórias de trabalho e de regras como *hash tables*, ajuda na redução do número de chamadas ao algoritmo de unificação em cada ciclo de inferência. Toda vez que o conteúdo da memória de trabalho for alterado, pode-se comparar seu novo conteúdo somente com as regras que possuem antecedentes com os mesmos predicados dos fatos adicionados ou removidos. Contudo, o número de unificações ainda poderia ser muito alto e uma porcentagem significativa ainda apresentaria os mesmos resultados do ciclo de inferência anterior.

Um algoritmo muito eficiente para tratar do problema de casamento é o algoritmo de casamento ***Rete***, introduzido por Forgy (1982) e descrito na Seção 4.6. Esse algoritmo elimina as deficiências descritas acima "lembrando-se" dos resultados das unificações realizadas em ciclos de inferência anteriores e, adicionalmente, elimina duplicações "estruturais" entre as regras. Com sua utilização, a complexidade computacional da fase de casamento em um ciclo de inferência se reduz a O(R·F·A) (FRIEDMAN-HILL, 2001).

4.5.8 Resolução de Conflitos

Com relação à fase de resolução de conflitos, existem diversas possibilidades para se decidir qual das regras de um conjunto de conflito deve ser disparada. Exemplos dessas estratégias podem ser encontrados em Russell e Norvig (1995) e em Rich e Knight (1994). Algumas possibilidades são:

- **recentidade**: disparar primeiro a regra ativada mais recentemente (*depth-first*);
- **antigüidade**: disparar as regras na ordem na qual elas foram ativadas, isto é, primeiro a mais que está no conjunto de conflito há mais tempo (*breadth-first*);
- **especificidade**: disparar primeiro a regra mais específica, isto é, a de maior número de antecedentes;
- **prioridade**: disparar as regras de acordo com uma prioridade pré-estabelecida para as ações presentes no conseqüente;
- **não-duplicação**: não disparar a mesma regra com os mesmos argumentos duas vezes.

4.6 Algoritmo de Casamento *Rete*

O algoritmo de casamento ***Rete*** possui duas fases distintas: uma de **compilação**, associada à fase de inicialização do mecanismo de inferência, e outra de **casamento** propriamente dita. Na primeira fase, o algoritmo compila o conjunto de regras presentes na BC em uma rede hierárquica de nós convenientemente interligados.

Existem três tipos básicos de nós: **nó-de-uma-entrada**, **nó-de-duas-entradas** e **nó-terminal**. Cada nó recebe sua(s) entrada(s) de seu(s) antecessor(es) e envia sua(s) saída(s) para o(s) seu(s) sucessor(es). Quando o compilador processa o antecedente de uma regra, ele começa pelos elementos estruturais de cada um de seus literais individualmente, por exemplo, seu predicado. Para cada literal do antecedente, ele determina as características desses elementos estruturais que devem ser satisfeitas por um fato (literal fechado positivo) de modo a unificá-lo com o literal em questão. Então, ele constrói uma seqüência de nós de uma entrada, cada um dos quais verifica a presença de uma dessas características.

Após tratar todos os elementos **intraliterais** do antecedente, o compilador constrói nós de duas entradas para testar as características de seus elementos **interliterais**, por exemplo, se uma mesma variável aparece em dois ou mais literais. O primeiro nó-de-duas-entradas junta as saídas das duas primeiras seqüências lineares de nós de uma entrada, o segundo nó-de-duas-entradas junta a saída do primeiro nó-de-duas-entradas com a saída da terceira seqüência linear de nós de uma entrada, e assim por diante. Cada nó-de-duas-entradas testa todas as características interliterais que dois ou mais fatos devem satisfazer para contribuir positivamente para a ativação da regra.

A título de exemplo, suponha-se que o seguinte conjunto de regras – adaptado de Russell e Norvig (1995) – seja apresentado ao algoritmo, durante a fase de compilação:

Regra 1: $A(x,y) \wedge B(y,x) \wedge C(Luís,x) \rightarrow$ adicionar $D(x,y)$

Regra 2: $A(x,y) \wedge B(y,x) \wedge D(x,x) \rightarrow$ remover $E(x,y)$

onde os quantificadores existenciais $\forall x$ e $\forall y$ foram eliminados por simplicidade.

O algoritmo irá compilá-lo na rede mostrada na Figura 7.

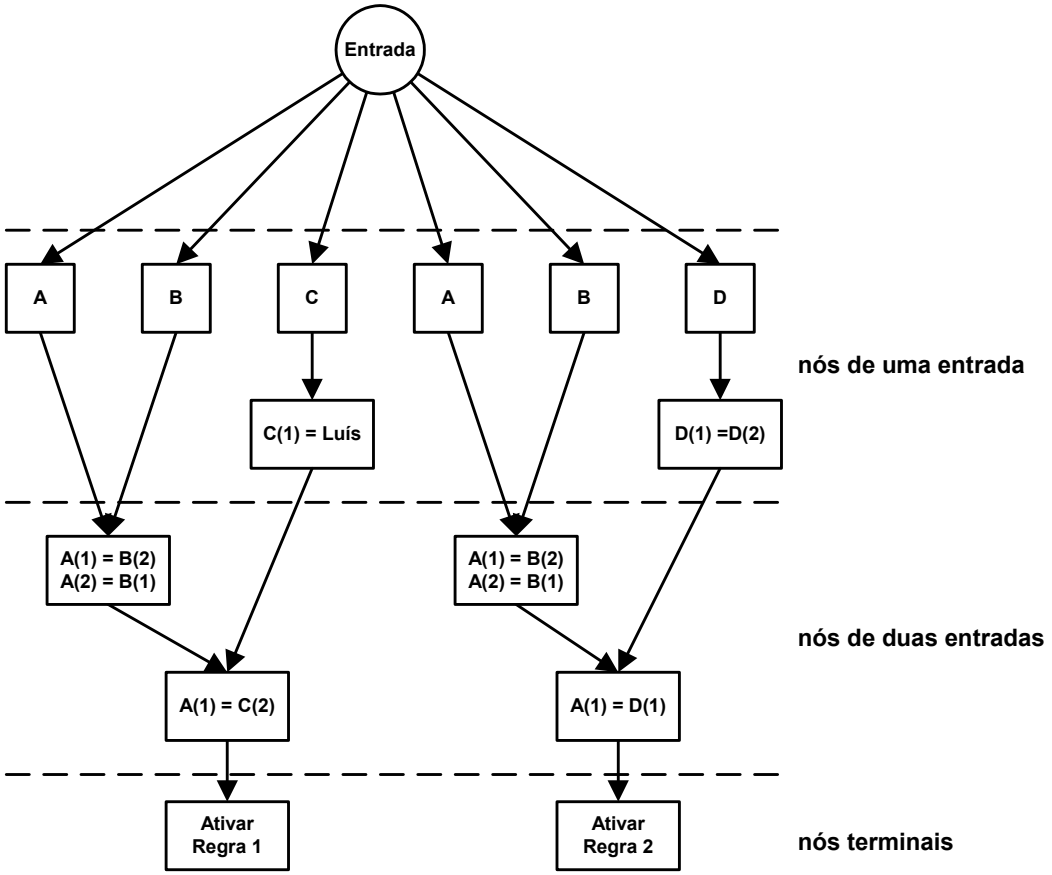


Figura 7: Uma rede de nós

Na Figura 7, os quadrados representam nós especiais de uma entrada, denominados **nós-de-entrada**, que verificam a igualdade dos predicados dos literais do antecedente. O texto no interior de cada um representa o símbolo de predicado testado. Os retângulos representam nós de uma ou duas entradas e os nós terminais. O texto no interior dos nós de uma ou duas entradas representam os testes realizados por eles. A notação `C(1) = Luís` significa que o primeiro termo do predicado `C` deve ser igual ao símbolo de constante Luís. Da mesma forma `A(1) = B(2)` indica que o primeiro termo variável do predicado `A` deve ser igual ao segundo termo do predicado `B`.

O compartilhamento das estruturas de nós entre regras distintas pode otimizar a rede produzida. Observando-se a Figura 7, percebe-se que existem seis nós de uma entrada para testar os predicados, mas somente quatro são distintos. Existem também três nós de duas entradas idênticos. Portanto, a rede pode ser otimizada de modo a compartilhá-los. A Figura 8 mostra a configuração final.

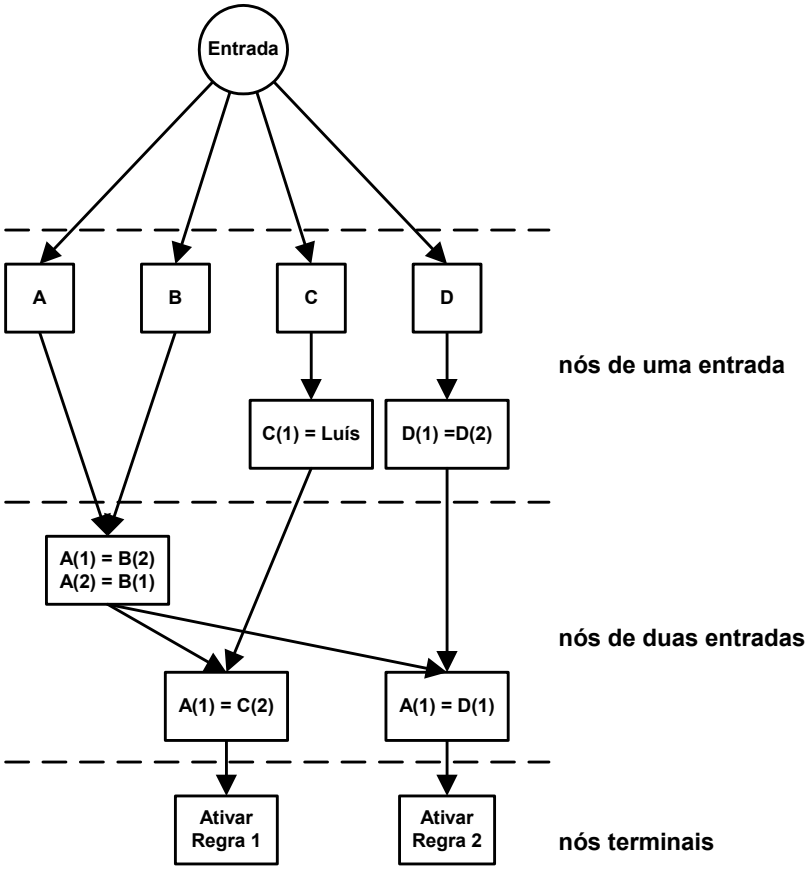


Figura 8: Rede otimizada

Na fase de casamento, a rede processa cada fato adicionado ou removido da memória de trabalho. Quando um fato é adicionado à BC, ele é apresentado a todos os nós de uma entrada presentes no topo da rede. Cada nó recebe o fato, aplica seu teste a ele e, se o resultado do teste for positivo, envia o fato para o nó inferior subseqüente. Se o teste falhar, o nó simplesmente ignora o fato apresentado. Todos os nós de uma entrada repetem esse processo. Um nó-de-duas-entradas integra os fatos provenientes de suas entradas à esquerda e à direta. Todo fato que alcançar um nó-de-duas-entradas pode, potencialmente, contribuir para a ativação de uma regra. Conseqüentemente, esse tipo de nó deve lembrar-se de todos os fatos que lhe foram apresentados ao longo dos diversos ciclos de inferência. Por essa razão, ele possui uma **memória esquerda** e uma **memória direita**. Esse nó aplica seu conjunto de testes para todas as combinações de entradas à esquerda e à direita e, se um certo par passar em todos eles, ele envia uma **conjunção** desses fatos para todos os seus nós subseqüentes. Esse esquema permite que os resultados das unificações passadas sejam lembrados ao longo dos ciclos de inferência. Quando uma conjunção de fatos atingir um dos nós terminais, ele passou por todos os

testes do antecedente de uma regra específica, a qual pode, portanto, ser acrescentada ao conjunto de conflito. Quando um fato é removido da BC, ele é processado da mesma forma. Se ele alcançar um nó-de-duas-entradas, será removido da respectiva memória, esquerda ou direita, e cada combinação sua com os elementos da outra memória, que passar em todos os testes do nó, é enviada aos nós subseqüentes para remoção. Para cada uma dessas conjunções que alcançarem um nó terminal, a regra associada é removida do conjunto de conflito.

Retornando ao exemplo, supondo-se que os seguintes fatos foram apresentados à rede: `A(Paulo,Pedro)`, `A(José,João)`, `B(João,Pedro)`, `B(Pedro,Paulo)`, `C(Luís,Pedro)`, `C(Luís,Paulo)` e `C(Luís, José)`, os conteúdos das memórias à direita e à esquerda dos nós de duas entradas seriam os mostrados na Figura 9.

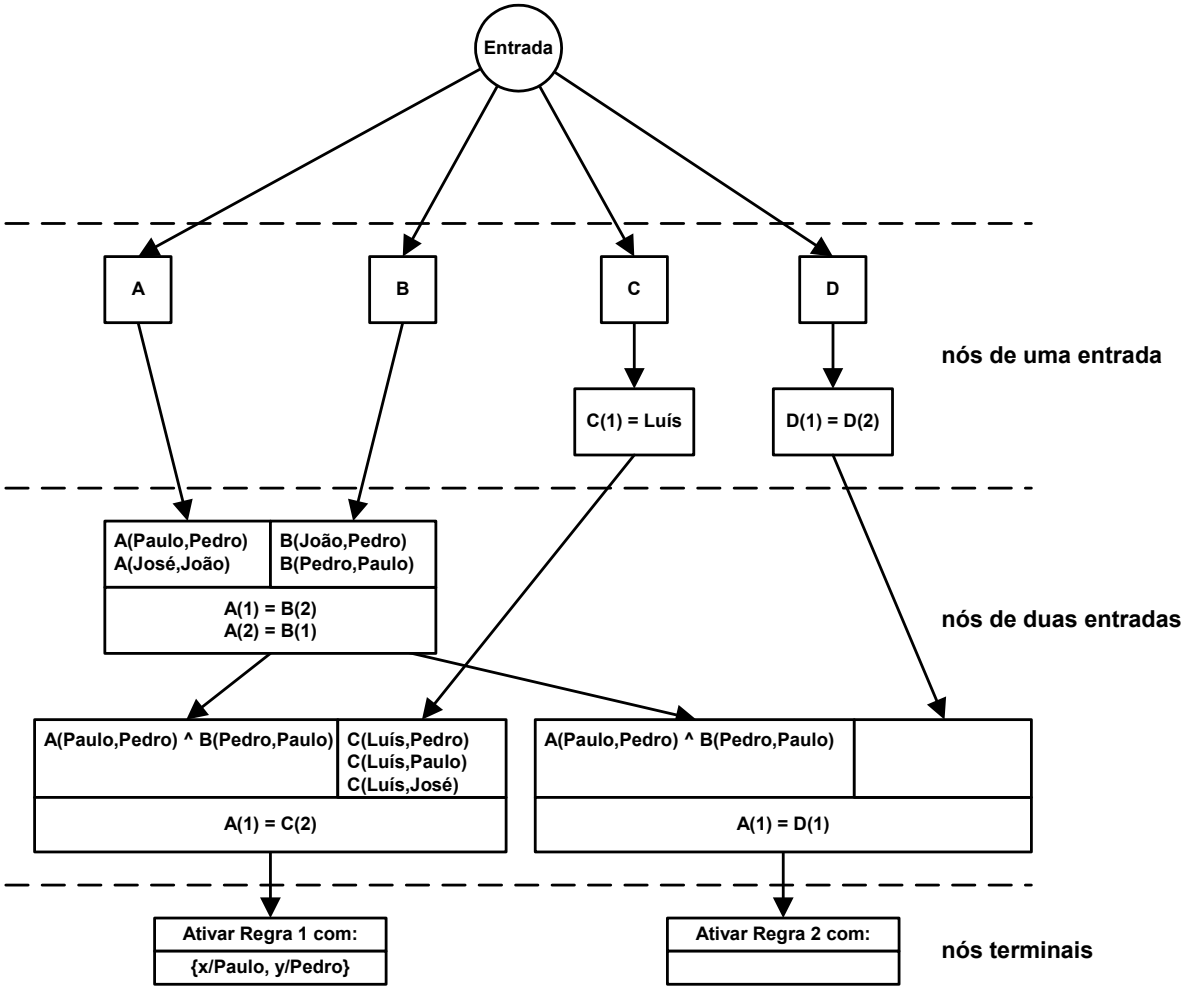


Figura 9: Conteúdos das memórias à esquerda e à direita dos nós de duas entradas

De acordo com a Figura 9, a regra 1 seria ativada com a substituição `{x/Paulo, y/Pedro}`, resultando na adição da sentença `D(Paulo,Pedro)` ao conjunto de conflito. Se essa regra for escolhida para ser disparada, esse fato será adicionado à BC e armazenado na memória à direita do nó-de-duas-entradas mais à direita na figura, o que, por sua vez provocará a ativação da regra 2 com a mesma substituição `{x/Paulo, y/Pedro}`, resultando na adição da sentença `A(Paulo,Pedro)` ao conjunto de conflito.

4.7 Considerações Finais

Neste capítulo foi mostrado que, na prática, os mecanismos de inferência com encadeamento progressivo normalmente se baseiam em procedimentos de inferência que aplicam *Modus Ponens* a uma base de conhecimento, na forma de regras de Horn, geralmente estendidas para suportar *negação por falha*, eventos, sensores e atuadores. Embora incompletos, tais procedimentos apresentam como vantagem o fato de serem decidíveis e de possuírem tempo de computação polinomial.

O desempenho de um mecanismo de inferência também depende sobremaneira das estruturas de dados empregadas na implementação da representação interna de sua base de conhecimento e na eficiência do algoritmo de casamento utilizado. Um dos algoritmos mais eficientes de casamento utilizados atualmente é o algoritmo ***Rete***.

No desenvolvimento do *framework* de inteligência para agentes móveis, especificado no Capítulo 6 e desenvolvido nos Capítulos 7 e 8, dar-se-á ênfase à redução do código e do estado de dados a serem transportados, tanto para o mecanismo de inferência em si, quanto para o algoritmo ***Rete***.

Capítulo 5

Desenvolvimento de Software Orientado a Objetos

5.1 Introdução

O objetivo deste trabalho é o desenvolvimento de um *framework* que permita, inicialmente, a inclusão de um mecanismo de inferência com encadeamento progressivo aos agentes móveis. No futuro, espera-se que outros tipos de mecanismos, como por exemplo, de inferência em lógica nebulosa e de aprendizado também sejam incorporados. Para que o *framework* seja flexível o suficiente para permitir a inclusão desses novos mecanismos e para que as quantidades de código e de dados transportados para cada um deles sejam reduzidas, é necessário que seus projetos sejam metódicos e rigorosos. As metodologias de desenvolvimento de software orientadas a objetos fornecem as ferramentas formais necessárias para esse fim.

Este capítulo tem por propósitos: (1) definir o que se entende por **desenvolvimento de software orientado a objetos**; (2) descrever o **processo de desenvolvimento** empregado na concepção do *Framework de Inteligência* para agentes móveis especificado no Capítulo 6 e projetado no Capítulo 7; e (3) explicar a **notação** (**linguagem de modelagem**) utilizada na documentação dos resultados decorrentes da aplicação do processo de desenvolvimento.

5.2 Desenvolvimento de Software Orientado a Objetos

Entende-se por **desenvolvimento de software orientado a objetos** a criação de sistemas de software flexíveis, robustos e de fácil manutenção usando a tecnologia de objetos e uma linguagem de programação orientada a objetos, como por exemplo, C++, Java, Smalltalk, etc.

A literatura sobre os fundamentos da tecnologia de objetos, tais como **classe**, **instância**, **interface**, **polimorfismo**, **encapsulamento**, **associações** e **herança** é abundante, como por exemplo, Rumbaugh et al. (1991), Jacobson et al. (1992), Coleman et al., (1994), Booch (1994) e Coad (1995). Portanto, tais conceitos não serão apresentados neste texto. Um resumo bastante abrangente pode ser encontrado em Silva (1998).

No entanto, para a criação de sistemas orientados a objetos de alta qualidade não é suficiente conhecer os conceitos fundamentais da tecnologia, utilizar uma linguagem de programação orientada a objetos e ter acesso a bibliotecas de classes bastante ricas. Para isso, também é necessário utilizar uma **metodologia de desenvolvimento orientada a objetos**, isto é, uma metodologia que dê ênfase à decomposição do espaço do problema e de sua solução por objetos e não por funções. A maioria das metodologias de desenvolvimento orientadas a objetos consiste de uma **linguagem de modelagem** e de um **processo de desenvolvimento**. (FOWLER, 2000). O **processo de desenvolvimento** corresponde à seqüência de passos a serem seguidos durante a investigação do domínio do problema e a construção de sua solução lógica. A **linguagem de modelagem** é a notação (principalmente gráfica) utilizada para expressar tanto os requisitos como as decisões tomadas durante a aplicação do processo de desenvolvimento.

5.3 Pontos de Vista, Modelos e Artefatos

Um sistema, tanto físico como de software, pode ser extremamente complexo. Para gerenciar essa complexidade e garantir o perfeito entendimento do que se deseja construir, freqüentemente é necessário observá-lo a partir de diferentes **pontos de vista**. Cada um desses

pontos de vista descreve o sistema de uma perspectiva particular, levando em consideração apenas um conjunto específico de propriedades, o que limita a quantidade de informação considerada num certo instante e, portanto, reduz a complexidade do processo de desenvolvimento. No contexto do desenvolvimento de software orientado a objetos, um **modelo** organiza, descreve e comunica os detalhes importantes do domínio do problema e do sistema a ser construído sob a perspectiva de um ponto de vista específico. Modelos são compostos por elementos coesos e fortemente relacionados, que podem ser outros modelos ou então **artefatos**, isto é, diagramas e documentos que descrevem algum aspecto específico do sistema. Cada modelo possui uma **notação associada**, na qual seus elementos e relacionamentos são expressos. Um modelo pode enfatizar informações **estáticas** ou **dinâmicas**. Um **modelo estático** descreve propriedades estruturais, enquanto um **modelo dinâmico** descreve propriedades comportamentais de um sistema. O conjunto de todos os modelos que descrevem cada um dos pontos de vista constitui o **Modelo do Sistema**.

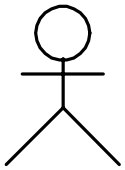
Alguns dos modelos e artefatos mais utilizados atualmente para documentar o desenvolvimento de um sistema de software orientado a objetos (LARMAN, 1998) serão apresentados na próxima seção. A notação adotada para cada um deles segue o padrão definido pela **Linguagem de Modelagem Unificada** (*Unified Modeling Language* - UML). Essa linguagem, adotada como padrão de indústria pela OMG (1999), serve para especificar, visualizar, construir e documentar os artefatos de modelagem tanto de sistemas de software como de outros tipos de sistemas. A UML tem como base uma coleção de práticas de engenharia de software com comprovada eficiência na modelagem de sistemas grandes e/ou complexos. Sua principal vantagem, além de ser um padrão aberto e não proprietário, é a sua independência tanto do processo de desenvolvimento quanto da linguagem de implementação utilizada.

A seguir são apresentados os diversos modelos segundo as notações definidas pela UML.

5.3.1 Modelo de Casos de Uso

O **modelo de casos de uso** tem por objetivo o entendimento, do ponto de vista do usuário, dos **requisitos funcionais** a serem satisfeitos pelo sistema. Ou seja, quais os modos de utilização e quais as funcionalidades desejadas para o sistema. Portanto, normalmente, ele

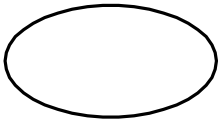
é desenvolvido com a estreita participação dos usuários finais. Embora a identificação de tais processos do domínio do problema não constitua realmente uma atividade orientada a objetos ela é uma etapa importante e amplamente praticada, uma vez que serve para balizar o desenvolvimento e definir claramente as fronteiras do sistema. Seus elementos básicos são os **atores** e os **casos de uso**. Um ator representa algo ou alguém externo que deve interagir com o sistema. A notação UML para representar um ator é a mostrada na Figura 10.



Nome do Ator

Figura 10: Notação UML para um ator

Um **caso de uso** corresponde a uma seqüência coesa de interações que ocorrem durante um "diálogo" entre um ator e o sistema. A Figura 11 mostra a notação UML para um caso de uso.



Nome do Caso de Uso

Figura 11: Notação UML para um caso de uso

Os artefatos que compõem o modelo de casos de uso são a **descrição dos casos de uso**, os **diagramas de casos de uso**, os **diagramas de estados de casos de uso** e o **esboço da interface de usuário**, descritos a seguir.

A **descrição dos casos de uso** é um documento que narra em linguagem coloquial todos os processos do domínio do problema, isto é, todos os padrões de comportamento apresentados pelo sistema. A descrição de um caso de uso pode ser realizada em diferentes níveis de detalhe e de comprometimento com decisões de projeto, dependendo da fase em que o processo de desenvolvimento se encontra. A UML não define nenhum formato específico para esse documento.

Os **diagramas de casos de uso** ilustram o conjunto de casos de uso de um sistema, explicitando seus atores e os relacionamentos entre eles. O objetivo é apresentar uma espécie de "diagrama de contexto", através do qual pode-se, rapidamente, perceber quem são os atores externos ao sistema e como eles o utilizam. Esses diagramas definem a fronteira do sistema, e portanto o que deve ser construído. A Figura 12 mostra um exemplo de diagrama de casos de uso segundo a UML.

Os **diagramas de estados de casos de uso** são diagramas opcionais, construídos normalmente para casos de uso complexos, com o objetivo de ilustrar a ordem correta aceitável de eventos externos de entrada do sistema (**eventos de sistema**). Durante as fases de projeto e implementação é necessário garantir que as ocorrências de eventos fora dessa ordem sejam rejeitadas.

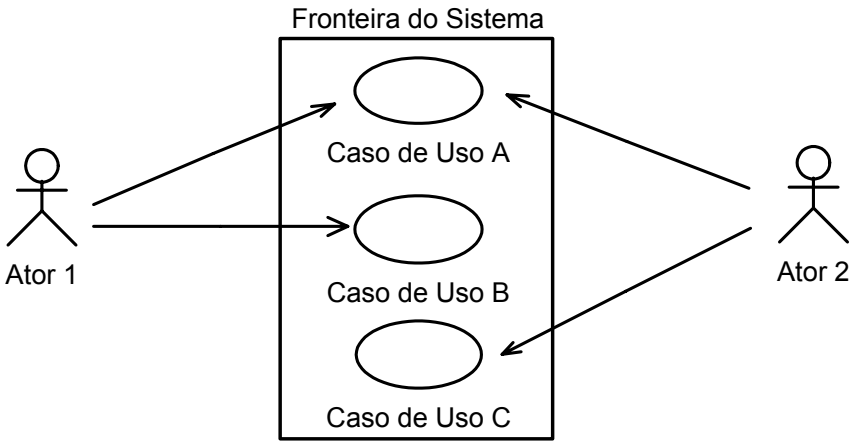


Figura 12: Exemplo de um diagrama de casos de uso

A Figura 13 mostra a notação UML para um diagrama de estados. Essa figura realça somente as características básicas desse tipo de diagrama. Um **evento** corresponde a uma ocorrência relevante que acontece em um dado instante de tempo e que pode provocar uma transição de um estado para outro. Um **estado** corresponde à condição de um certo elemento (casos de uso, sistema, conceitos, ou classes) em um dado instante de tempo. Uma **transição de estado** é um relacionamento entre dois estados que indica a mudança de um estado para outro. Uma transição de estado pode provocar a execução de uma certa ação, denominada **ação de transição**. Uma transição de estado também pode possuir uma **condição de guarda**, função booleana que deve ser satisfeita para que a transição ocorra.

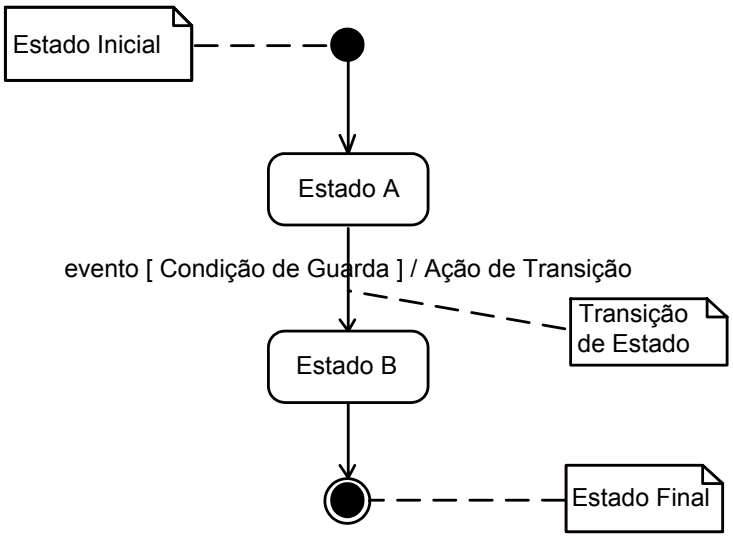


Figura 13: Notação UML para um diagrama de estados

O **esboço da interface de usuário** é criado, geralmente, para dar subsídios à descrição dos casos de uso. Algumas vezes, um ator pode representar algum outro sistema de software que deve interagir com o sistema em desenvolvimento; neste caso, pode ser necessário definir-se as interfaces com tais sistemas. A UML não define nenhum formato específico para esse documento.

5.3.2 Modelo de Conceitos

O **modelo de conceitos** é uma representação dos **conceitos** que fazem parte do **domínio do problema** no mundo real. O termo **conceito** é utilizado justamente para enfatizar que o foco está nas entidades do domínio do problema e não nas entidades de software (classes, no caso da orientação a objetos) que comporão a solução. Esse modelo é formado por um conjunto de **diagramas de conceitos**, que ilustram os conceitos, suas estruturas (atributos) e seus relacionamentos. Além de decompor o problema em unidades compreensíveis, o modelo de conceitos ajuda a tornar clara a terminologia (vocabulário) do domínio. Deste modo, ele pode ser visto como uma ferramenta de comunicação entre as partes interessadas no desenvolvimento. Sua criação depende do modelo de casos de uso e de um profundo conhecimento do domínio do problema, a partir dos quais os conceitos poderão ser identificados.

A UML não define nenhuma notação específica para a representação dos conceitos e de seus atributos e relacionamentos. No entanto, pode-se utilizar um subconjunto do seu diagrama de classes. Um conceito é representado por um retângulo com dois compartimentos. No compartimento superior indica-se o nome do conceito e no inferior seus atributos. A indicação do tipo do atributo é opcional. A Figura 14 mostra a notação utilizada para um conceito, derivada da notação UML para uma classe.

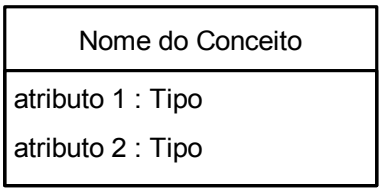


Figura 14: Notação para um conceito

Em um diagrama de conceitos, basicamente três tipos de relacionamentos podem ser representados: **associação**, **agregação** e **generalização** em um modelo de conceitos.

Uma **associação** corresponde a um relacionamento semântico entre dois ou mais conceitos. É indicada por uma linha unindo os conceitos associados e pode possuir um nome. Cada uma de suas extremidades também pode possuir um **nome** e uma indicação de **multiplicidade**. A Figura 15 ilustra a notação para o relacionamento de associação.

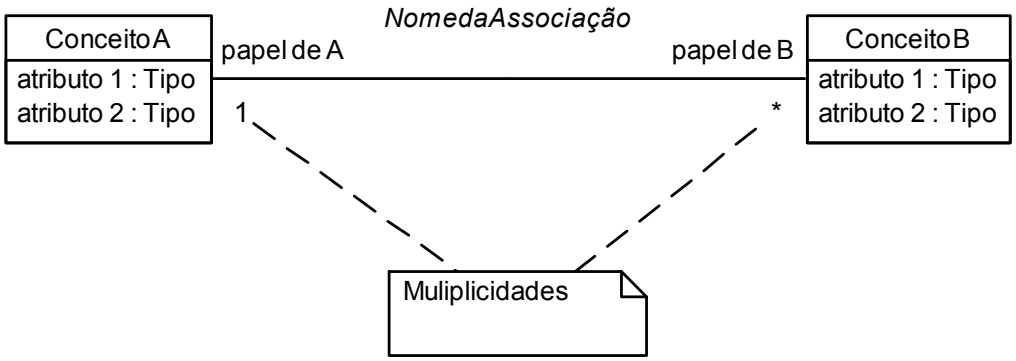


Figura 15: Notação UML para o relacionamento de associação

Os nomes das extremidades indicam o **papel** que cada um dos conceitos desempenha na associação. A multiplicidade indica quantas instâncias de um conceito se relacionam, em um dado instante de tempo, com **uma única** instância do outro conceito. Os indicadores de multiplicidade mais comuns estão mostrados na Tabela 6.

Tabela 6: Indicadores de multiplicidade mais comuns

1	Exatamente um
1..*	Um ou mais
0..*	Zero ou mais (muitos)
*	Zero ou mais (muitos)
0..1	Zero ou um
m..n	Faixa de valores (por exemplo: 4..7)
m,n	Exatamente m ou n (por exemplo, 4 ou 7)

A **agregação** é uma forma mais forte de associação que ocorre entre um todo e suas partes. É indicada por uma linha unindo os conceitos relacionados, com um losango próximo ao conceito que representa o todo. Suas extremidades também podem possuir indicadores de multiplicidade. A Figura 16 exemplifica a notação para esse tipo de relacionamento.

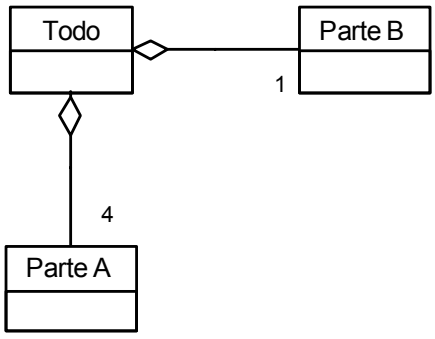


Figura 16: Notação UML para um relacionamento de agregação

A **generalização** corresponde a uma forma mais forte de associação entre um conceito mais genérico, denominado **supertipo**, e um ou mais conceitos mais específicos, denominados **subtipos**. As similaridades entre eles são indicadas no supertipo e suas especificidades nos subtipos. Portanto, trata-se de um modo de se construir classificações taxonômicas entre conceitos e de ilustrá-las em uma hierarquia de tipos. Sua identificação é útil porque permite compreender os conceitos em termos mais gerais e abstratos. Além disso, contribuem para uma "economia" de expressão, ao reduzirem a representação de informações repetidas. Na UML, o relacionamento de generalização entre elementos é indicado por uma linha com um triângulo vazado apontando do elemento mais específico para o elemento mais genérico. A Figura 17 ilustra esse tipo de relacionamento.

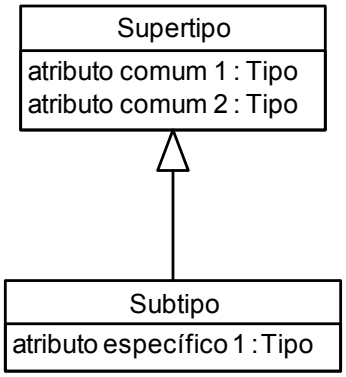


Figura 17: Notação UML para um relacionamento de generalização

Quando o número de conceitos identificados se torna muito elevado deve-se agrupá-los em módulos que permitam o gerenciamento e a visualização de alto nível do modelo de conceitos. A UML fornece um mecanismo de uso geral, denominado **pacote**, para organizar elementos em grupos. Portanto, um pacote corresponde a um conjunto de elementos de quaisquer tipos, tais como casos de uso, conceitos, subsistemas, classes, outros pacotes, etc. Graficamente, um pacote é representado por um retângulo com uma aba e um nome, como mostra a Figura 18.

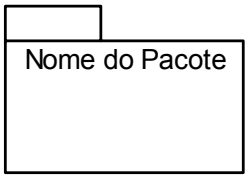


Figura 18: Notação UML para um pacote

Na UML, se um elemento de modelagem de algum modo depende de outro, essa dependência pode ser mostrada através de um **relacionamento de dependência**, indicado por uma linha tracejada com uma seta apontando do elemento dependente para o elemento do qual ele depende. A dependência entre pacotes indica que os elementos do pacote dependente, de alguma forma, têm conhecimento ou estão acoplados aos elementos do outro pacote. Tanto o conjunto de pacotes identificados para os conceitos, quanto seus relacionamentos de dependências são mostrados nos **diagramas de pacotes de conceitos**. A Figura 19 exemplifica a notação utilizada.

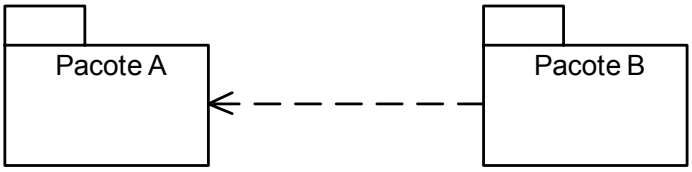


Figura 19: Exemplo de um diagrama de pacotes

5.3.3 Modelo de Comportamento Sistêmico

O **modelo de comportamento sistêmico** tem por objetivo ampliar o entendimento do comportamento do sistema como uma "caixa preta". O modelo de casos de uso sugere como os atores interagem com o sistema para terem acesso às suas funcionalidades. Durante essas interações, um ator gera eventos para o sistema (**eventos de sistema**), requisitando certas operações como resposta. Deste modo, um evento de sistema inicia uma **operação de sistema**. O modelo de comportamento sistêmico isola, ilustra e descreve todas as operações que um ator pode requisitar de um sistema. Os artefatos que compõem esse modelo são descritos a seguir.

Os **diagramas de seqüência de sistema** ilustram, para um curso particular de eventos em um caso de uso (**cenário**), os atores que interagem diretamente com o sistema e os eventos de sistema gerados por esses atores. Cada evento de sistema inicia um comportamento de resposta do sistema (operação de sistema). A notação utilizada para esses diagramas segue o padrão para **diagramas de seqüência** definidos pela UML, mostrado na Figura 20. Um diagrama de seqüência deve ser criado para cada curso típico de eventos de um caso de uso e, possivelmente, para cada um de seus cursos alternativos mais importantes.

Os **contratos para as operações de sistema** são documentos que descrevem, para cada uma das operações de sistema, quais serão as alterações no estado global do sistema quando a operação em questão for invocada. É comum que essa descrição seja feita em termos das **responsabilidades** atribuídas à operação e de suas **pré** e **pós-condições**. A UML não define nenhum formato específico para esse documento.

O diagrama de estados de sistema é um diagrama opcional que ilustra todas as transições de estado provocadas por todos os eventos externos. Ele corresponde, portanto, à união dos diagramas de estado de todos os casos de uso do sistema. É útil somente se o núme-

ro de eventos de sistema for pequeno. Utiliza-se para esse diagrama a mesma notação UML apresentada na Figura 13.

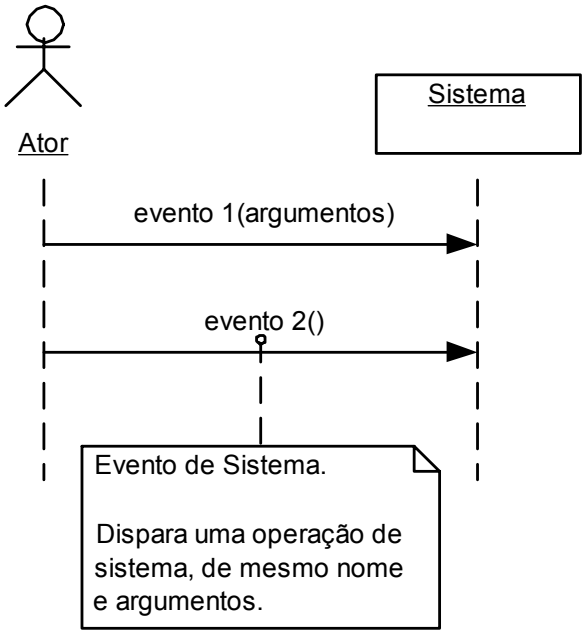


Figura 20: Notação UML para o diagrama de seqüência de sistema

5.3.4 Modelo de Comportamento de Objetos

O **modelo de comportamento de objetos** descreve o comportamento dinâmico dos objetos de software que compõem o sistema. Em um sistema orientado a objetos, as operações executadas pelo sistema são realizadas de forma colaborativa por um conjunto de objetos que se comunicam entre si através da troca de mensagens. Geralmente, uma mensagem corresponde à solicitação de um serviço (**operação pública**) oferecido por um objeto. A forma como os objetos (e as classes) interagem para realizarem cada uma das operações de sistema é ilustrada em um conjunto de **diagramas de interação**, no mínimo um para cada operação de sistema. A criação desses artefatos é uma das atividades mais importantes do desenvolvimento de software orientado a objetos, consumindo uma parcela significativa do tempo e dos esforços despendidos em todo o projeto. Durante essa atividade, as entidades de software (objetos) que comporão o sistema são identificadas, bem como as responsabilidades de cada uma delas, isto é, os serviços atribuídos a cada uma delas. A construção bem sucedida dos diagramas de interação requer a aplicação de **padrões de projeto** (princípios estruturados) e **de a-**

tribuição de responsabilidades. Essas responsabilidades são documentadas de forma textual nos **contratos para as operações das classes**. A UML não define nenhum formato específico para os contratos. Por outro lado, para os diagramas de interação existem duas notações possíveis: os **diagramas de seqüência** e os **diagramas de colaboração**.

Os **diagramas de seqüência** mostram as interações entre os objetos (instâncias de uma classe) dispostas na forma de uma seqüência temporal. A notação utilizada é idêntica à mostrada na Figura 20, com alguns detalhes adicionais. Normalmente as mensagens são numeradas e o período de tempo durante o qual um objeto está realizando uma ação, em resposta a uma mensagem recebida, seja diretamente ou através de um procedimento subordinado, é indicado pelo símbolo **foco de controle**, como ilustrado na Figura 21. De uma forma geral, o formato padrão UML para uma mensagem é o seguinte:

```
valor_de_retorno := mensagem(parâmetro:tipo):tipo_do_retorno
```

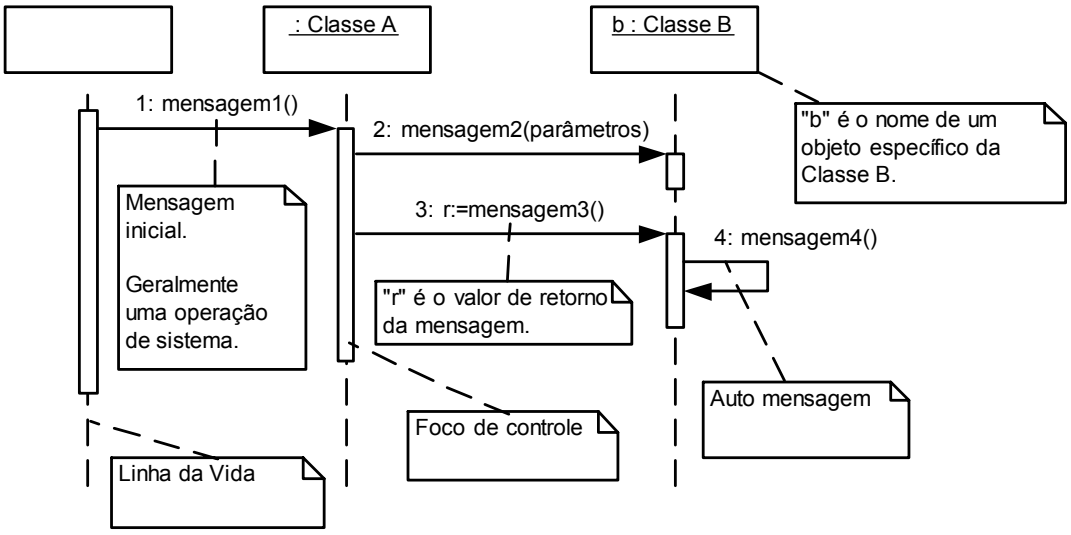


Figura 21: Notação UML para o diagrama de seqüência

Os **diagramas de colaboração** mostram as interações entre os objetos organizadas em torno de suas **ligações**. Formalmente, uma ligação é uma instância de uma associação. Nesse tipo de diagrama, a ordem temporal e os aninhamentos das mensagens são, necessariamente, identificados através de uma **numeração de seqüência**, que pode possuir vários níveis, como mostrado na Figura 22. Além disso, também é possível ilustrar mensagens condicionais, mensagens para objetos contêineres (**multiobjetos**) e mensagens para uma classe de objetos.

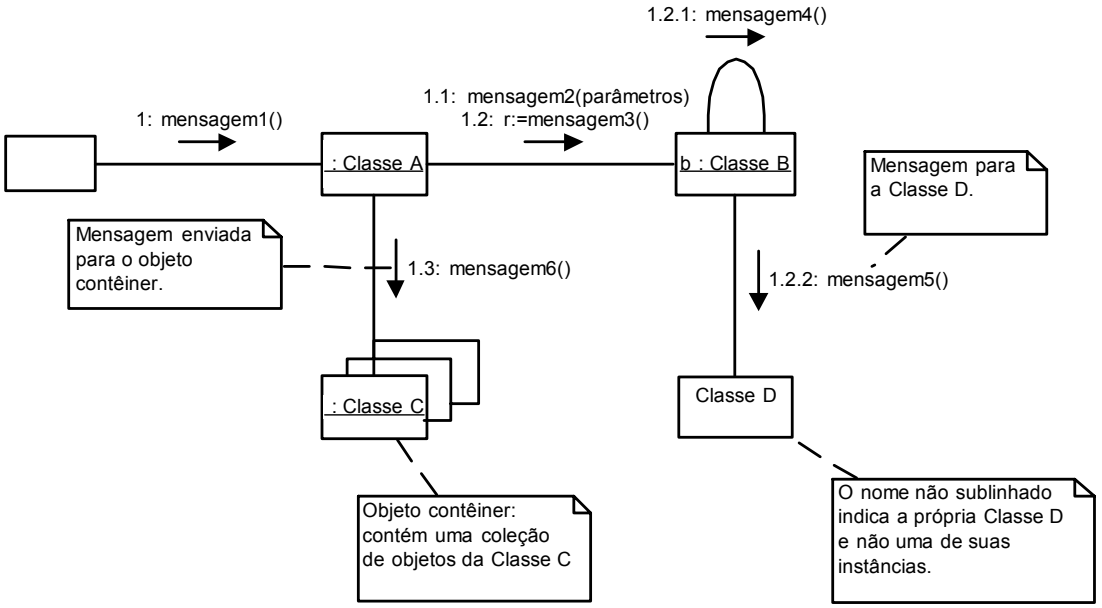


Figura 22: Notação UML para o diagrama de colaboração

Também fazem parte desse modelo os **diagramas de estados de classes**. Esses diagramas ilustram, para cada classe com comportamento complexo, o ciclo de vida de um objeto dessa classe. Tipicamente, classes com comportamento complexo são aquelas que possuem lógicas condicionais e que, portanto, reagem de forma diferente à ocorrência de um evento, dependendo de seu estado. O ciclo de vida de um objeto corresponde aos seus estados possíveis e às transições entre eles, provocadas pelas ocorrências dos eventos. Utiliza-se para esses diagramas a mesma notação UML apresentada na Figura 13.

5.3.5 Modelo de Classes

O **modelo de classes** tem por objetivo sintetizar as definições das **classes e interfaces** responsáveis pela realização das funcionalidades especificadas para o sistema, sem levar em consideração a distribuição dessas entidades pelos diversos nós computacionais (computadores) que o hospedam. Trata-se, portanto, de um modelo que descreve a **solução lógica** adotada para o sistema, sendo composto por um conjunto de **diagramas de classes**, que mostram classes, interfaces e seus relacionamentos.

A notação utilizada para esses diagramas e seus elementos é idêntica à mostrada da Figura 14 a Figura 17, com alguns detalhes adicionais. Em contraste com os diagramas de

conceitos, os diagramas de classes mostram definições de entidades de software e não conceitos do mundo real. Portanto, normalmente, acrescenta-se a esses diagramas informações relativas às **operações** a serem realizadas pelas classes e interfaces, bem como informações de **navegabilidade**.

Para acomodar as informações sobre as operações, o retângulo que representa uma classe possui três compartimentos: o superior, com nome da classe; o do meio, com seus atributos e o inferior, com as assinaturas das operações. Se o nome da classe aparecer em ***itálico***, isso indica que a classe é **abstrata**. Conforme mostrado na Figura 23, também pode-se indicar a visibilidade dos atributos e das operações através dos símbolos que os precedem.

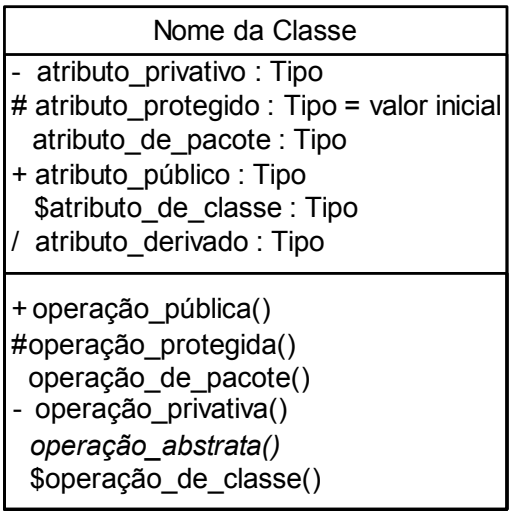


Figura 23: Notação UML para uma classe

Uma **interface** é um conjunto nomeado de operações visíveis externamente (públicas). Elas são ilustradas de forma similar às classes. Como mostrado na Figura 24, as únicas diferenças são que o nome no compartimento superior é precedido pelo estereótipo «Interface» e que, normalmente, não existe o compartimento central destinado aos atributos. Todas as operações de uma interface são, por definição, públicas.

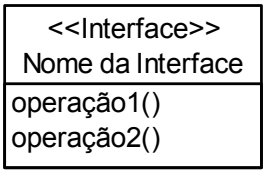


Figura 24: Notação UML para uma interface

O fato de uma classe implementar o conjunto de operações definido por uma interface é indicado por um **relacionamento de realização**, conforme mostrado na Figura 25. Quando não for necessário apresentar as operações de uma interface, ou quando essas operações já tiverem sido mostradas em outra parte do modelo, pode-se utilizar uma notação alternativa para representá-la, com o objetivo de tornar o diagrama mais conciso. Essa notação alternativa está mostrada na Figura 26, onde o pequeno círculo representa a interface e a descrição imediatamente abaixo dele, o nome da mesma.

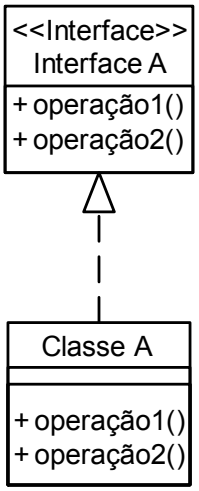


Figura 25: Notação UML para o relacionamento de realização

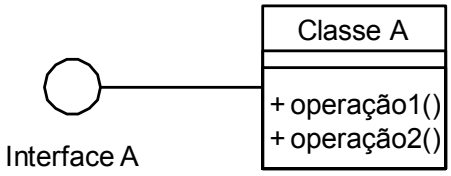


Figura 26: Notação UML alternativa para interface

Navegabilidade é uma propriedade de cada um dos extremos de uma associação ou agregação, que indica a direção na qual o relacionamento pode ser percorrido. O fato de um relacionamento ser navegável em um único sentido implica numa restrição de **visibilidade de classe**, isto é, no fato de que uma das classes do relacionamento (a **classe fonte**) tem conhecimento da existência da outra classe (a **classe alvo**), mas que o contrário não é verdadeiro. Durante a implementação, a navegabilidade é traduzida no fato da classe fonte possuir uma referência para um ou mais objetos da classe alvo. A navegabilidade é indicada nos diagramas

por uma seta decorando a extremidade do relacionamento, perto da classe alvo. A Figura 27 exemplifica um diagrama (parcial) de classes, ilustrando o conceito de navegabilidade.

Da mesma forma que para o modelo de conceitos, quando o número desses diagramas torna-se muito elevado pode-se agrupá-los em pacotes, cujos relacionamentos são mostrados nos **diagramas de pacotes de classes**. Utiliza-se para esses diagramas a mesma notação UML apresentada na Figura 19.

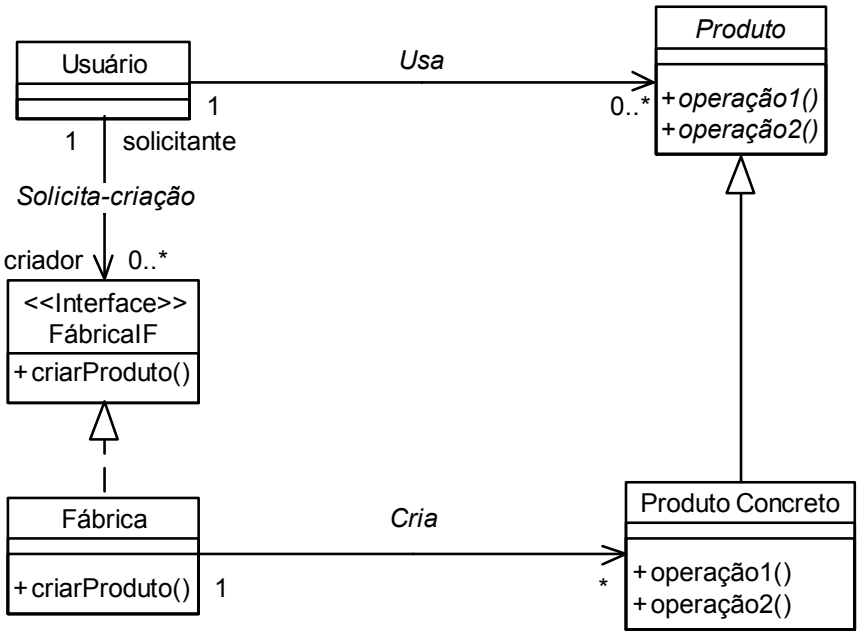


Figura 27: Notação UML para o diagrama de classes

5.3.6 Modelo Arquitetônico

O **modelo arquitetônico** descreve, através de um ou mais **diagramas de arquitetura**, a **arquitetura lógica** do sistema, isto é, sua decomposição em múltiplos subsistemas, bem como a comunicação e o acoplamento entre eles. As arquiteturas mais utilizadas atualmente são as arquiteturas multicamadas, nas quais os diversos subsistemas são distribuídos em camadas verticais de acordo com suas responsabilidades. Geralmente, utilizam-se as camadas de **apresentação** (responsável pela interface com o usuário), **lógica da aplicação** (responsável pelas regras e tarefas que governam o processo) e de **armazenamento de dados** (responsável pelos mecanismos de armazenamento persistente). Utiliza-se para o diagrama de arquitetura

lógica a mesma notação UML apresentada na Figura 19. A Figura 28 ilustra um exemplo de diagrama para uma arquitetura multicamada.

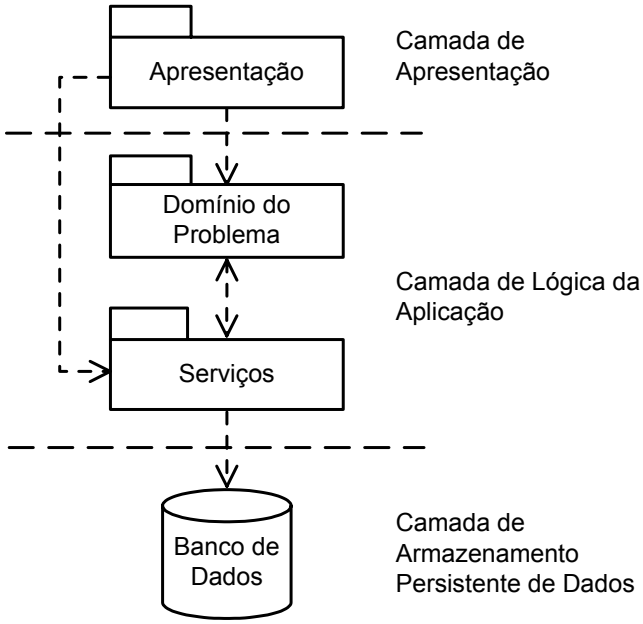


Figura 28: Camadas arquitetônicas representadas por pacotes UML

5.3.7 Modelo de Componentes

O **modelo de componentes** descreve a organização do software do sistema. É composto por um ou vários **diagramas de componentes** que ilustram as dependências de compilação e de execução entre os diversos componentes de software do sistema (arquivos fontes, executáveis e de biblioteca). É útil somente quando o número de tais componentes é elevado. A Figura 29 exemplifica a notação UML para esse tipo de diagrama.

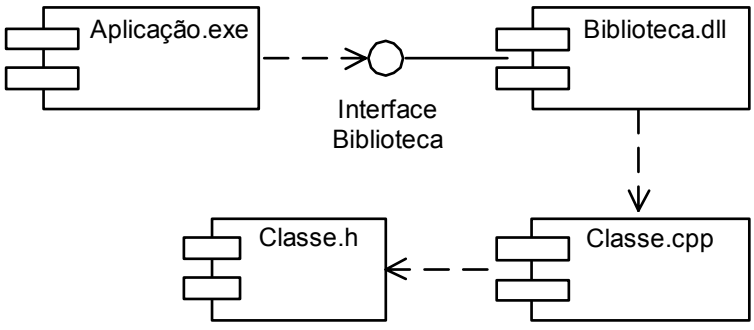


Figura 29: Notação UML para os diagramas de componentes de software

5.3.8 Modelo de Implantação

A **arquitetura lógica** do sistema pode ser implantada **fisicamente** de várias formas. Por exemplo, os subsistemas da camada de apresentação podem ser instalados no computador cliente, os subsistemas da lógica de aplicação no servidor de aplicações e os subsistemas da camada de armazenamento no servidor de dados. O **modelo de implantação** descreve a configuração dos diversos nós computacionais (computadores) que fazem parte do sistema, sendo composto por um único **diagrama de implantação**, que ilustra a alocação dos processos e componentes que fazem parte do sistema pelos diversos nós computacionais que o suportam. Se todos os processos e componentes do sistema forem executados em um único computador a criação de tal diagrama é desnecessária. A Figura 30 exemplifica a notação UML para esse tipo de diagrama.

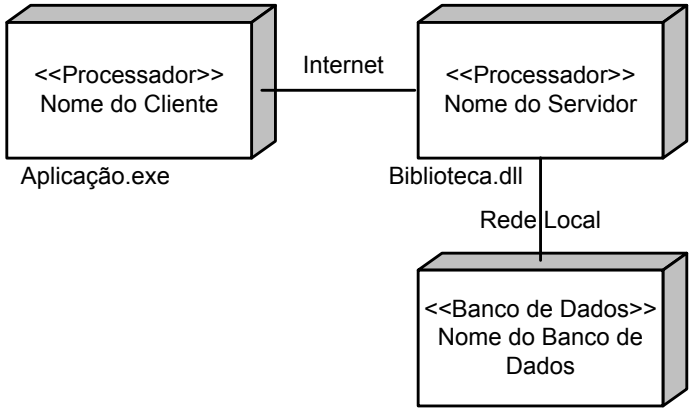


Figura 30: Notação UML para o diagrama de implantação

5.4 Processo de Desenvolvimento de Software

Os modelos e artefatos descritos na seção anterior são construídos de forma incremental ao longo de um **processo de desenvolvimento de software** e documentados utilizando-se uma **notação associada**. A seqüência exata de construção depende do processo de desenvolvimento adotado. Nesta seção descreve-se o processo de desenvolvimento utilizado neste trabalho.

Um **processo de desenvolvimento de software** é um procedimento para organizar as atividades relacionadas à criação de um sistema de software. Fundamentalmente, a descrição de um processo dessa natureza deve incluir a descrição e a seqüência das atividades necessárias para a construção dos diversos modelos e artefatos utilizados. No entanto, uma descrição mais completa deve incluir também todas as atividades relacionadas ao ciclo de vida do produto, passando, dentre outros, pela sua concepção inicial, investigação de alternativas, planejamento, gerenciamento do projeto, testes, documentação, manutenção e suporte a usuários.

Atualmente, considera-se que um bom processo de desenvolvimento (LARMAN, 1998) deve possuir as seguintes características:

- **Ser iterativo e incremental**: Dado que os sistemas de software atuais são bastante sofisticados, não é mais possível, seqüencialmente, definir-se primeiro todo o problema, projetar a solução completa, construir o software e então testá-lo no final. É necessária uma abordagem iterativa que permita um entendimento gradual do problema, através de refinamentos sucessivos, e a construção incremental de sua solução. Em cada iteração devem ser considerados subconjuntos relativamente pequenos de requisitos e o sistema cresce, portanto, de forma incremental, à medida que cada iteração é concluída.
- **Ser dirigido a casos de uso**: o processo de desenvolvimento deve ser influenciado e organizado em torno de casos de usos. Deste modo, a escolha dos requisitos que deverão ser considerados em cada ciclo de iteração será balizada pelos casos de uso identificados. Por exemplo, na iteração 1 será tratado o "caso A", na iteração 2 será tratada uma versão simplificada do "caso B", na iteração 3 o restante do "caso B", e assim por diante. As atividades em um ciclo de iteração deverão focalizar os requisitos tratados pelos casos de uso considerados. Portanto, os conceitos do domínio do problema e as classes, assim como seus relacionamentos serão identificados em um ciclo de iteração, a partir desses casos de uso específicos e, do mesmo modo, os casos de testes serão escritos para validá-los.
- **Ser centrado na arquitetura**: neste contexto, a arquitetura se refere à estrutura de alto nível de subsistemas e componentes que constituem o sistema, assim

como suas dependências, interfaces e interações. Um bom processo deve encorajar a criação antecipada de uma arquitetura geral para o sistema que, idealmente, deve ser robusta, flexível, escalável e de fácil entendimento. Seus subsistemas devem ser distribuídos em camadas, sendo que suas interfaces e interações devem ser bem definidas de modo a garantir um baixo acoplamento entre eles. Durante as fases iniciais de desenvolvimento, o foco deve estar nos subsistemas com maior grau de risco e de incerteza, que, geralmente, encontram-se na camada do domínio do problema ou na camada de serviços de alto nível. Contudo, a ênfase antecipada na arquitetura impõe que alguma atenção também deve ser dedicada às demais camadas e subsistemas, o que inclui seus componentes, suas interfaces e interações.

A seguir apresenta-se o processo utilizado no desenvolvimento do *Framework de Inteligência* para agentes móveis. Este processo, adaptado de (LARMAN, 1998), possui todas as características atribuídas anteriormente a um bom processo. Dar-se-á ênfase aos **aspectos técnicos** relativos às atividades de criação dos artefatos descritos em 5.3.

De um modo geral, como mostrado na Figura 31, o desenvolvimento orientado a objetos de uma versão de um sistema de software pode ser dividido em uma etapa de **planejamento do desenvolvimento** e em diversos **ciclos de desenvolvimento**. Ao final do último ciclo de desenvolvimento a nova versão do software é colocada em uso operacional ou, no caso de softwares comerciais, vendidas aos clientes.

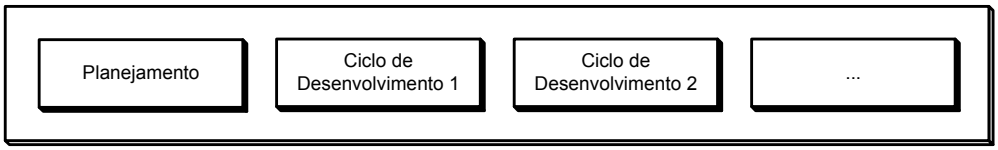


Figura 31: Desenvolvimento de uma versão de um sistema de software

5.4.1 Etapa de Planejamento

As atividades realizadas durante a etapa de planejamento do desenvolvimento são mostradas na Figura 32. A primeira atividade é a **especificação dos requisitos** que o sistema deve cumprir. A partir da especificação de requisitos e do **conhecimento do domínio do pro-**

blema, os **atores** e os **casos de uso** são identificados. A seguir, os casos de usos são descritos **de forma simplificada**, usualmente em duas ou três sentenças. Através dos **diagramas de casos de uso**, ilustram-se todos os casos de uso identificados e seus relacionamentos. Pode-se criar então uma **versão preliminar do modelo de conceitos**, com o objetivo de auxiliar no entendimento do vocabulário do domínio, especialmente no que diz respeito aos casos de uso e à especificação de requisitos. Neste ponto, um **esboço da arquitetura** imaginada para o sistema também pode ajudar, embora sua criação possa ser postergada. Os termos (nomes dos conceitos) e quaisquer informações associadas, tais como restrições e regras devem ser registradas num glossário. Essa atividade, embora apareça como a última desta etapa, normalmente é realizada em paralelo com as demais e estende-se por todo o desenvolvimento. Finalmente, os casos de uso identificados devem ser priorizados e distribuídos pelos diversos ciclos de desenvolvimento, com os de maior prioridade sendo tratados logo nas iterações iniciais. Dessa forma, os ciclos de desenvolvimento são organizados em torno dos casos de uso. Em um ciclo de desenvolvimento podem ser alocados um ou mais casos de uso, ou mesmo versões simplificadas de um caso complexo. A seqüência de desenvolvimento estabelecida deve ser documentada em um **cronograma de desenvolvimento**.

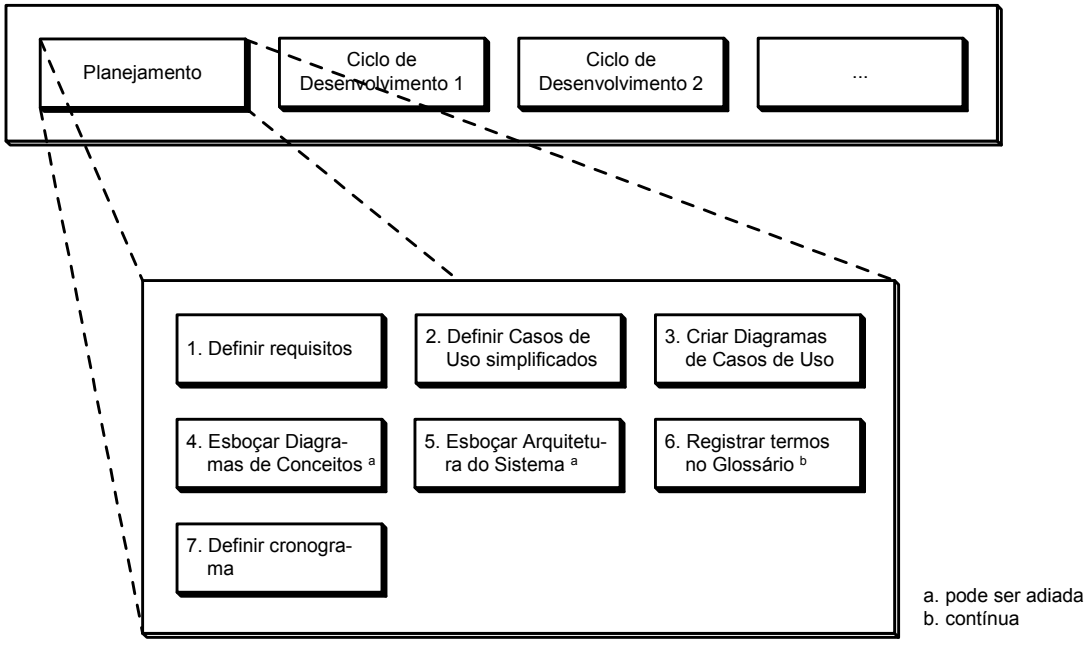


Figura 32: Atividades realizadas na etapa de planejamento

5.4.2 Ciclos de Desenvolvimento

As atividades realizadas em um ciclo de desenvolvimento são agrupadas em **fases**, conhecidas como fases de **análise**, **projeto**, **implementação**, **teste** e **sincronização de artefatos** (Figura 33). Os modelos e artefatos descritos na Seção 5.3 são construídos e refinados, de forma incremental, principalmente nas fases de análise e projeto de cada ciclo. Ao final de cada ciclo, tem-se implementado um sistema, que satisfaz os requisitos descritos pelos casos de uso atribuídos ao ciclo em questão e a todos os seus antecessores.

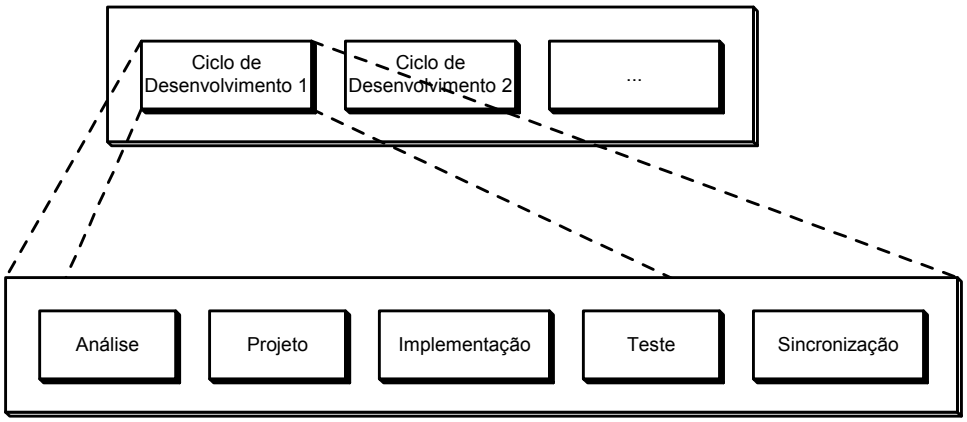


Figura 33: Fases de um ciclo de desenvolvimento

5.4.3 Fase de Análise

Durante a **fase de análise**, a ênfase está na investigação dos conceitos do domínio do problema relacionados aos casos de uso tratados no ciclo em questão. Deste modo, as atividades dessa fase concentram-se em refinar os Modelos de Casos de Uso, de Conceitos e de Comportamento do Sistema. Os casos de uso escolhidos para serem tratados durante o ciclo corrente de desenvolvimento são descritos em maiores detalhes, sem que, no entanto, sejam introduzidas informações relativas à implementação e tecnologia a ser utilizada, especialmente no que se refere à interface de usuário. Essa forma de descrição de um caso de uso, detalhada, mas sem decisões de projeto, é denominada **descrição detalhada essencial**. Os diagramas de casos de uso e os diagramas de conceitos devem ser refinados para refletirem os detalhes introduzidos nessa descrição, da mesma forma que o **glossário**. Se for o caso, os conceitos devem ser agrupados em pacotes e os relacionamentos e dependências entre os pa-

cotes documentados através dos **diagramas de pacotes de conceitos**. Tomando-se como ponto de partida principalmente o Modelo de Casos de Uso, definem-se, então, os **diagramas de seqüência de sistema** e os **contratos para suas operações**; se necessário, define-se também o **diagrama de estados de sistema**. Esses artefatos compõem o **modelo de comportamento sistêmico**, para o ciclo de desenvolvimento em questão.

A Figura 34 mostra as atividades executadas durante a fase de análise de um ciclo de desenvolvimento e a seqüência de realização proposta. Variações dessa seqüência básica também são possíveis.

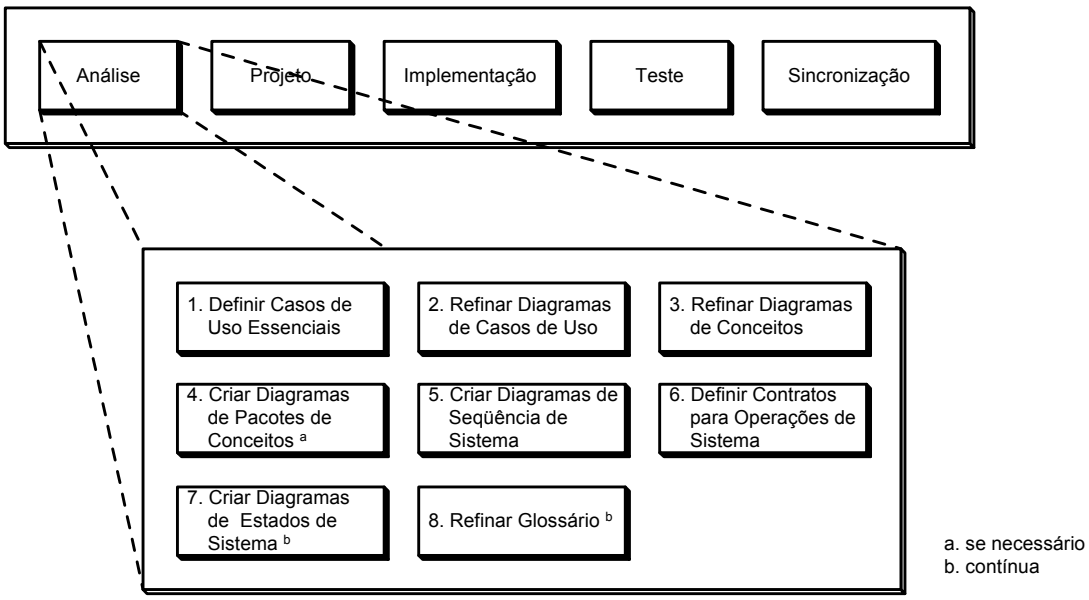


Figura 34: Atividades realizadas na fase de análise

5.4.4 Fase de Projeto

Quando os documentos de análise de um ciclo de desenvolvimento estão completos, é possível iniciar-se a fase de projeto. Através das atividades dessa fase, é construída uma solução lógica para o sistema, baseada no paradigma de orientação a objetos. Essa solução é documentada através dos **modelos de comportamento de objetos**, **de classes** e de **arquitetura do sistema**. A espinha dorsal dessa solução é a criação dos **diagramas de interação**, os quais ilustram como os objetos irão se comunicar de forma a satisfazerem o subconjunto de requisitos tratados no ciclo corrente. Seguindo a geração desses diagramas, podem-se dese-

nhar os **diagramas de classes**, os **contratos para suas operações** e, se necessário, seus **diagramas de estados**. Esses artefatos sintetizam as definições das classes e interfaces a serem implementadas. O **glossário** também deve ser atualizado com a inclusão dos novos termos identificados. Nessa fase, também podem ser definidos os **mecanismos de armazenamento persistente de dados**, suas estruturas e suas interfaces com os subsistemas do domínio do problema.

Para dar suporte à criação dos diagramas de interação, pode ser útil, embora não seja estritamente necessário, descrever os casos de uso selecionados para o ciclo de desenvolvimento corrente num formato **real** ou **concreto**. Essa **descrição detalhada real dos casos de uso**, criada a partir da descrição detalhada essencial, é realizada em termos da tecnologia concreta de entrada e saída a ser utilizada para o sistema. Por exemplo, se uma interface gráfica de usuário for utilizada, a descrição detalhada real dos casos de uso incluirá os diagramas das janelas de entrada e saída envolvidas e a discussão das interações dos atores com elementos visuais, tais como botões e caixas de entrada de texto.

A Figura 35 mostra as atividades executadas durante a fase de projeto de um ciclo de desenvolvimento e a seqüência de realização proposta. Variações dessa seqüência básica também são possíveis.

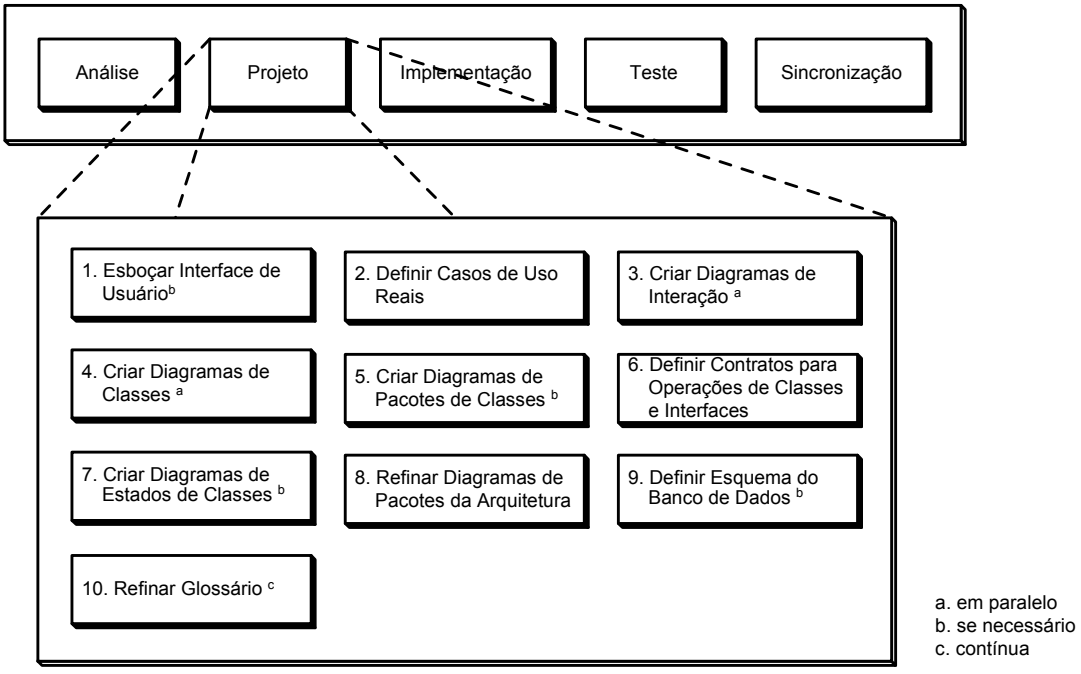


Figura 35: Atividades realizadas na fase de projeto

5.4.5 Fase de Implementação

Quando os **modelos de classes e de comportamento de objetos**, para o ciclo de desenvolvimento em progresso, estiverem completos, já existirão informações suficientes para a geração do código dos objetos da camada do domínio do problema. Desta forma, pode-se dar início à fase de implementação do sistema de software. Essa implementação, em uma linguagem de programação orientada a objetos, requer a criação dos códigos-fonte para as **assinaturas das classes e interfaces**, dos **métodos**, das **janelas de entrada e saída de dados** e das **interfaces de acesso aos mecanismos de armazenamento persistente**. Ao longo da implementação, as dependências entre os diversos elementos de software gerados e a sua distribuição pelos nós de processamento que hospedarão o sistema são documentadas através dos **diagramas de componentes** e **de distribuição**. A Figura 36 mostra a seqüência básica de execução dessas atividades.

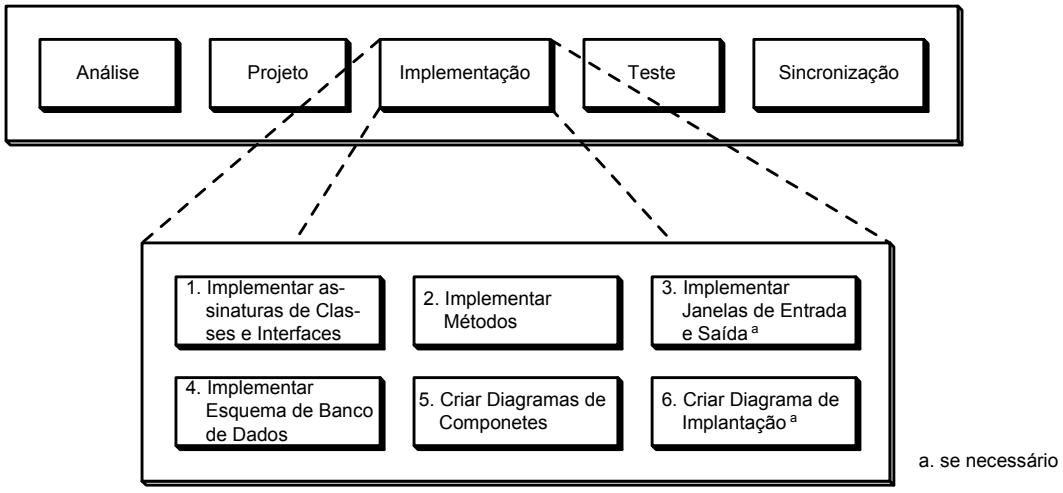


Figura 36: Atividades realizadas durante a fase de implementação

5.4.6 Fase de Testes

Embora a Figura 33 mostre a fase de teste como sendo a penúltima de um ciclo desenvolvimento, de fato, suas atividades são executadas ao longo da fase de implementação. A Figura 37 mostra as principais atividades da fase de testes. O **teste de unidade** é normalmente executado à medida que as classes, interfaces e métodos são implementados. Os **testes de**

sistema e de integração são dirigidos pelos **casos de teste** que por sua vez são criados com base nos casos de uso e nos contratos para as operações de sistema. Por outro lado, nem todos os testes apresentados nessa figura são realizados em todos os ciclos de desenvolvimento. Esse é o caso dos testes de integração, de aceitação e de documentação.

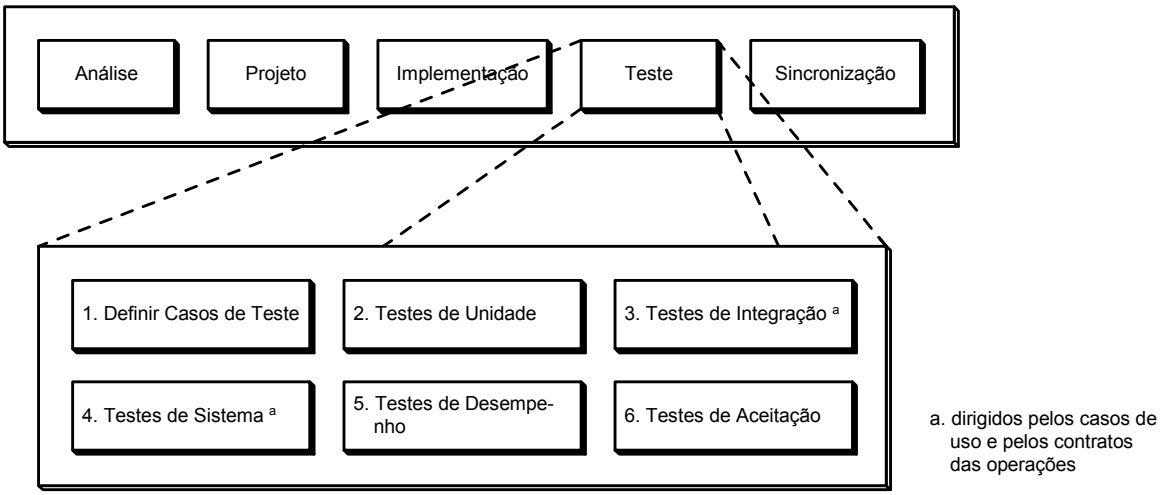


Figura 37: Atividades realizadas durante a fase de teste

5.4.7 Fase de Sincronização

Usualmente, durante as fases de implementação e testes, surgirão diversos problemas e detalhes não identificados durante as fases de análise e projeto. Como conseqüência, deve-se esperar que ocorram desvios entre o projeto original e o sistema realmente implementado. Portanto, com o objetivo de manter-se uma documentação consistente e preparar a próxima iteração, a última fase de um ciclo de desenvolvimento deve ser destinada à **sincronização dos artefatos** gerados durante a análise e projeto com o código efetivamente construído.

Finalmente, a Figura 38 (LARMAN, 1998) mostra as dependências de construção entre os principais artefatos descritos.

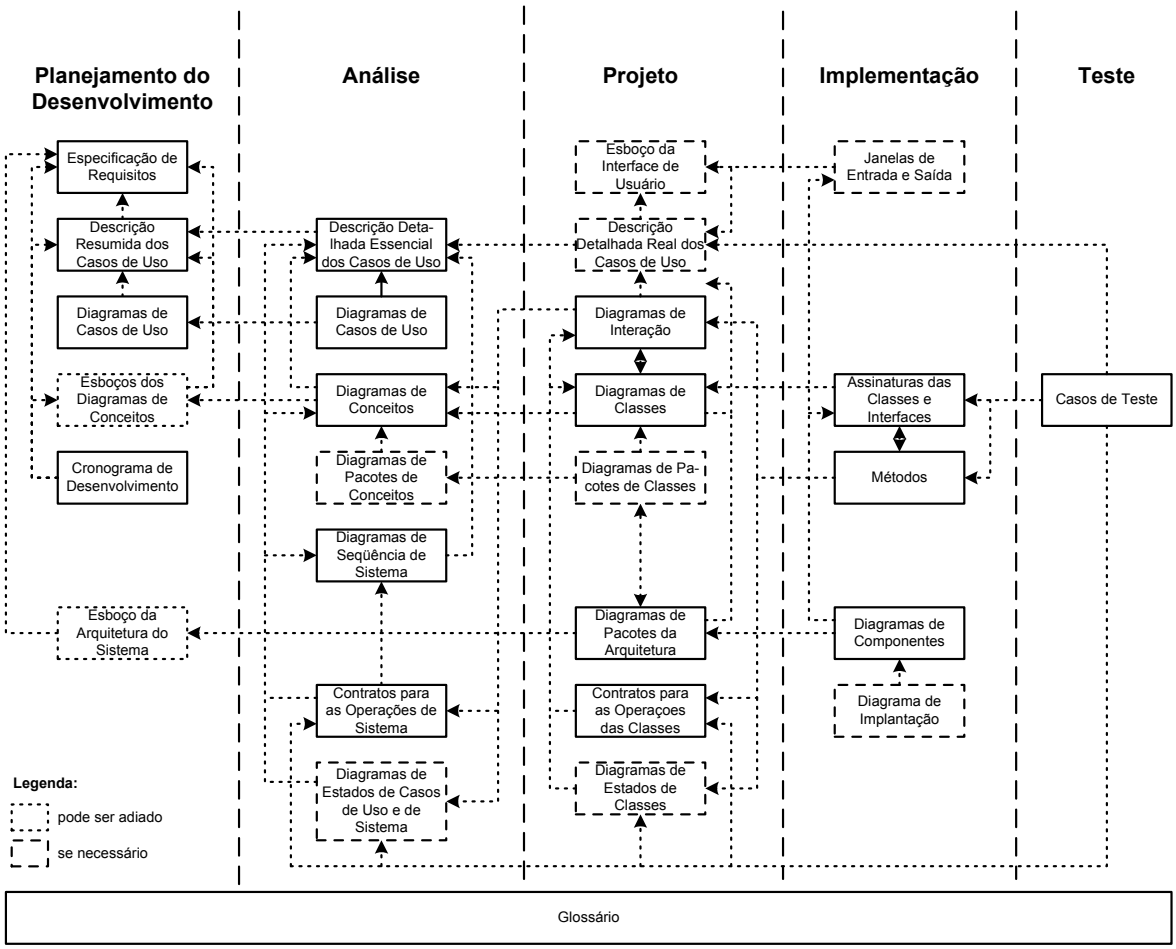


Figura 38: Dependência entre os artefatos

5.5 Considerações Finais

A notação apresentada neste capítulo será utilizada na definição da arquitetura geral proposta para o *Framework de Inteligência* realizada no Capítulo 6 e, em maior grau, na documentação do desenvolvimento de seu núcleo, apresentado no Capítulo 7, e do mecanismo concreto de inferência, descrito no Capítulo 8.

Embora existam outros processos de desenvolvimento de software propostos na literatura, como, por exemplo, o ***Fusion*** (COLEMAN, 1994) e o ***Object Models*** (COAD, 1995), até o momento, o ***Rational Unified Process*** (RUP) da Rational (2000) é o mais promissor porque além de dar grande ênfase aos aspectos de gerenciamento do trabalho em equipe é o

herdeiro direto de três processos anteriores de enorme aceitação, ***Object Modeling Techniques*** (OMT) de James Rumbaugh (RUMBAUGH et al., 1991), ***Object-Oriented Software Engineering*** (OOSE) de Ivar Jacobson (JACOBSON et al., 1992) e o método de Booch (BOOCH, 1994). No entanto, neste trabalho optou-se pela utilização do ***Recommend Models and Patterns*** (RMP) apresentado em Larman (1998) uma vez que ele pode ser considerado uma simplificação do RUP para o desenvolvimento de sistemas mais simples, que não exijam o desenvolvimento por equipes.

Capítulo 6

Um *Framework de Inteligência* para Agentes Móveis

6.1 Introdução

Neste capítulo, definem-se os requisitos que um *framework* deve apresentar para incorporar mecanismos de IA a agentes móveis e propõe-se uma arquitetura capaz de satisfazê-los. Os requisitos que o *framework* e seus mecanismos de IA agregados devem satisfazer são apresentados na Seção 6.2. A seguir, na Seção 6.3, introduz-se, em linhas gerais, a estratégia utilizada no desenvolvimento do *framework*. Na Seção 6.4 realiza-se a análise dos requisitos, apontando-se como eles serão satisfeitos e apresenta-se a arquitetura proposta para o *framework*.

6.2 Especificação dos Requisitos

Requisitos são descrições das funcionalidades e das propriedades que um sistema deve apresentar para ser útil aos seus usuários. O objetivo principal deste trabalho é o desenvolvimento de um *framework* que permita a inclusão de mecanismos "leves" de IA em agentes móveis, criados com os principais sistemas de mobilidade existentes atualmente. A meta é aumentar a adaptabilidade e capacidade de tais agentes ao realizarem tarefas em ambientes dinâmicos e altamente distribuídos. A seguir, descreve-se cada um dos requisitos estabelecidos para o *framework* e a suas justificativas.

R1. Todos os componentes do *framework* devem ser inteiramente codificados em Java.

Esse requisito é necessário porque, conforme apresentado em 3.6, os principais sistemas atuais para a construção de agentes móveis são desenvolvidos em Java, e não existem evidências de que essa tendência vá se alterar, devido à contínua evolução apresentada por essa linguagem.

Por outro lado, embora **Prolog** (*Programmation en Logique*) e **Lisp** (*LISt Processing language*) sejam as linguagens usualmente associadas à programação em IA, recentemente, muitos trabalhos comerciais nessa área têm sido implementados em C e C++ (BIGUS; BIGUS, 1998). Como uma linguagem de propósito geral, Java pode substituí-las, com as vantagens apresentadas em 3.4.

Para garantir que o *framework* de inteligência possa ser utilizado com os principais sistemas de agentes móveis, e que todos os seus componentes possam ser transportados com os agentes durante toda a sua migração ou obtidos sob demanda, quando necessários, a linguagem Java é a escolha natural.

R2. Durante sua execução, um mecanismo deve ser capaz de receber, de forma assíncrona, percepções do agente móvel sobre si mesmo e sobre seu ambiente.

Este requisito permite que o agente móvel solicite o serviço de um mecanismo toda vez que perceber a ocorrência de algum evento importante em seu ambiente ou em sua própria execução. Normalmente, um evento é qualificado por um conjunto de informações que o descreve. Uma percepção é a união do evento com seu conjunto de informações. Uma mesma percepção pode ser enviada a mais de um mecanismo. O agente deve ser capaz de decidir sobre quais eventos são importantes e para quais mecanismos as percepções devem ser enviadas, facilitando-se assim a incorporação de comportamentos reativos.

R3. Durante sua execução, um mecanismo deve ser capaz de solicitar informações adicionais ao agente.

A solicitação de informações adicionais corresponde a chamadas de procedimentos externos ao mecanismo, os quais representam sensores síncronos. Esse requisito é importante para permitir o **comportamento situado** por parte dos agentes (GROSOF et al., 1995; GROSOF, 1999). No entanto, a ausência do procedimento externo chamado não

deve bloquear a execução do mecanismo, nem provocar sua falha; o mecanismo deverá continuar sua execução simplesmente assumindo que a informação desejada não está disponível. Também deve ser transparente para o mecanismo, se o procedimento externo chamado faz parte do código do agente ou da agência hospedeira, o que facilita a transferência de procedimentos que representam sensores do código do agente para os das agências sem o reprojeto dos componentes do *framework*. Desse modo, à medida que o uso de determinado sensor se tornar ubíquo, ele poderá deixar de fazer parte do agente, passando a ser fornecido pelas agências, o que contribui para a redução do código e dos dados transportados.

R4. Como resultado de sua execução, um mecanismo deve ser capaz de sugerir uma ação ao agente.

A sugestão de ações corresponde à sugestão de chamadas de procedimentos externos ao mecanismo, os quais representam atuadores. Esse requisito, assim como R2, também é importante para garantir o comportamento situado por parte dos agentes (GROSOF et al., 1995; GROSOF, 1999). Da mesma forma que para os sensores, a não execução da ação sugerida não deve bloquear a execução ou provocar uma falha do mecanismo; também deve ser transparente para o mecanismo a localização do procedimento externo vinculado à sua sugestão.

Ao contrário dos trabalhos encontrados na literatura (IBM, 1997, 2000; BIGUS; BIGUS, 1998), considera-se conveniente que, tanto as funções de sensoriamento, como as de atuação, sejam mediadas pelo agente. Isto é, os mecanismos não devem ter acesso direto aos sensores e atuadores. Com isso, assegura-se a autonomia e liberdade de escolha do agente, que pode decidir por não avaliar o ambiente em um dado momento, ou por executar uma ação proposta por outro de seus mecanismos. Com isso, reforça-se a posição de que os mecanismos de IA devem ser considerados dispositivos auxiliares, sob o controle do agente.

R5. O *framework* deve permitir que os mecanismos sejam facilmente incorporados em um agente móvel já existente.

Não se deve impor ou assumir que a implementação do agente especialize qualquer classe específica para que ele seja capaz de utilizar um mecanismo. Esse requisito é importante porque, como descrito em 3.6, a maioria dos sistemas de agentes móveis já

impõe que a classe principal do agente especialize uma classe abstrata fornecida, para que o mesmo possa ser reconhecido e aceito pelas agências. Geralmente, essa classe abstrata define as operações básicas utilizadas para controlar a mobilidade e o ciclo de vida dos agentes, além de permitir a interação do agente móvel com seu ambiente de execução. Como apresentado em 3.6, no caso do *Aglets*, essa classe abstrata é a `Aglet;` no *Grasshopper*, a `MobileAgent`.

Deve ser possível conectar e desconectar os mecanismos dinamicamente, em tempo de execução, de modo que, através da substituição e/ou agregação de novas partes, o agente possa adaptar-se às alterações de seu ambiente, bem como transportá-las somente quando necessário.

R6. Os mecanismos não devem estar vinculados a nenhum formato não executável de representação de conhecimento.

Para uma mesma classe de mecanismo – por exemplo, de inferência ou de aprendizado – podem existir diversos formalismos de representação de conhecimento. Por sua vez, para um mesmo formalismo de representação de conhecimento, podem existir diversos formatos para expressar suas sentenças. Por exemplo, uma base de conhecimento em KIF pode ser codificada tanto no formato de texto puro, mais adequado à compreensão humana, quanto em XML (***Extended Markup Language***), mais adequado à interpretação por parte de sistemas computacionais (GROSOF; LABROU, 1999).

Um mecanismo não deve supor ou impor que o agente seja instruído em um determinado formato de codificação, pois isso limitaria sua utilização e obrigaria o usuário a se adaptar ao formato escolhido. O melhor é que exista, para cada tipo de mecanismo, um formato comum de representação *executável* – isto é, na forma de objetos de software – para o qual todos os outros possam ser traduzidos. Isso reduz os esforços de análise sintática (*parsing*) e de conversão para o formato interno de representação de cada mecanismo. Também é importante, para ampliar a possibilidade de seu reuso, que esse formato executável de representação seja puramente declarativo, não incluindo informações que o vincule a algum tipo específico de utilização. Por exemplo, no caso de um mecanismo de inferência, o formato executável escolhido para a representação de conhecimento não deve conter informações relativas ao tipo de procedimento

de inferência (encadeamento progressivo ou regressivo), ao algoritmo de casamento ou à estratégia de resolução de conflitos utilizados.

Portanto, durante sua inicialização na agência nativa, o agente deverá transferir sua base de conhecimento para os mecanismos na forma de *objetos*, criados a partir do formato externo de representação adotado.

R7. As classes que compõem o *framework* e os mecanismos agregados devem ser leves, em termos de tamanho de código e de dados.

Uma das principias vantagens atribuídas aos agentes móveis é justamente a redução do tráfego na rede. Portanto, o *framework* em si e os mecanismos agregados aos agentes através dele não devem ser pesados a ponto de dificultarem o processo de migração.

R8. As classes que compõem o *framework* e os mecanismos agregados devem ser leves em termos de tempo de execução.

Uma vez que na maior parte de seu ciclo de vida um agente móvel será executado em uma máquina que não a de seu proprietário, é importante que seu código seja eficiente, não consumindo recursos computacionais em demasia, recursos esses que normalmente serão pagos.

R9. É desejável que os formatos internos de representação de conhecimento utilizados pelos mecanismos sejam opacos.

Embora os aspectos de segurança estejam fora do escopo deste trabalho, é interessante dificultar-se ao máximo o exame, por estranhos, das informações e princípios que regem o comportamento do agente.

R10. Os mecanismos devem ser flexíveis o bastante para permitirem que seus principais algoritmos sejam independentes uns dos outros.

Esse requisito tem por objetivo facilitar a utilização de algoritmos mais recentes e/ou alternativos para as principais funcionalidades dos mecanismos, ampliando sua reutilização sem a necessidade de reprojeto. Assim, algoritmos mais adequados, em termos de custo de transporte, poderão ser utilizados, à medida que se tornarem disponíveis. Os usuários também poderão criar seus próprios componentes para as principais funcionalidades.

R11. As classes que compõem o *framework* e os mecanismos agregados não devem estar diretamente vinculadas a qualquer tipo de interface de usuário ou mecanismo de armazenamento persistente de dados.

O objetivo desse requisito é maximizar a reutilização do *framework* e dos mecanismos. A idéia é que exista uma separação lógica suficientemente nítida entre os elementos principais do *framework* e os componentes da interface de usuário (UI) e/ou de aplicação (API), de forma que a substituição de uma interface por outra mais recente, mais amigável ou mais adequada a um domínio de utilização possa ocorrer sem que seja necessário o reprojeto dos mecanismos e até mesmo a interrupção de sua execução. Essa separação também é importante pois os componentes das interfaces poderão ser deixados na agência nativa, uma vez que, geralmente, não são necessários nas demais agências visitadas.

R12. As classes que compõem o *framework* e os mecanismos não devem estar diretamente vinculadas a nenhum tipo específico de informação ou de comportamento relativos à sua aplicação cliente.

Esse requisito visa maximizar o domínio de aplicações que poderão utilizar-se do *framework*, não restringindo seu uso somente a agentes móveis.

6.3 Estratégia de Desenvolvimento

Nesta seção, apresenta-se a estratégia proposta para o desenvolvimento do *Framework de Inteligência* para agentes móveis. Um *framework* é composto por um conjunto coeso de classes e interfaces que colaboram entre si para oferecerem uma funcionalidade quase completa que pode ser personalizada ou ampliada pelo usuário para atender às suas necessidades específicas. *Frameworks* possibilitam um elevado grau de reutilização, permitindo a criação rápida de soluções a partir de componentes preexistentes. Um *framework* difere de uma simples coleção de classes e interfaces porque ele estabelece a seqüência de troca de mensagens entre esses elementos, enquanto a coleção só define a semântica das operações envolvidas.

Em linhas gerais, a estratégia utilizada no desenvolvimento será a seguinte: (1) não eliminar totalmente a programação procedural, e (2) projetar o *framework* na forma de módulos "leves" que possam ser transportados pelo agente ou "carregados" (*downloaded*) sob demanda após sua chegada numa agência. Para cada mecanismo incorporado, uma análise cuidadosa deverá ser realizada para identificar e isolar suas principais funcionalidades e determinar quais delas podem ser transferidas para as agências.

As classes responsáveis pelas funcionalidades que devem migrar serão minuciosamente projetadas de forma a reduzir o tamanho de seu código e de seus dados. Por outro lado, assume-se que as funcionalidades transferidas para as agências serão oferecidas aos agentes na forma de um **Serviço de Inteligência** (SI), cuja implementação dependerá do sistema de agentes móveis utilizado. Tarefas de sensoriamento e atuação são exemplos de serviços que poderão ser transferidos para as agências através do SI. Esses serviços poderão ser realizados, por exemplo, por agentes estáticos instalados nas agências hospedeiras. Ao iniciar sua execução em uma agência, o agente deverá negociar com ela para obter uma referência para o serviço local de inteligência. Essa abordagem possui três vantagens: primeiro, contribui para a redução da quantidade de código e de dados transportados pelo agente, ao permitir a transferência da responsabilidade pela prestação de certos serviços às agências; segundo, permite que as agências imponham suas próprias políticas de segurança; e, terceiro, garante a "autonomia" das agências, que poderão, ou não, oferecer o serviço de inteligência ao agente. Além disso, o estabelecimento de uma interface padrão para o SI (**Interface de Servidor do SI**) desacopla a troca de informações entre o AMI e as agências, no que concerne às necessidades específicas dos mecanismos de IA, do sistema de mobilidade utilizado. Desse modo, os procedimentos para "encaminhamento" das solicitações de sensoriamento e atuação poderão ser desenvolvidos de forma independente do sistema de mobilidade, garantindo o seu reuso.

A Figura 39 mostra os principais componentes do *framework* de inteligência. O **Código do Agente** representa a programação procedural que deve ser implementada pelo desenvolvedor do agente, com base na aplicação específica desejada. Embora relacionado principalmente à programação procedural, o código do agente também deverá: (1) inicializar o(s) mecanismo(s) durante o processo de criação do agente; (2) iniciar e interromper a execução do(s) mecanismo(s) durante a execução do agente; e (3) servir de "ponte" entre o(s) mecanismo(s) e o serviço de inteligência oferecido pela agência hospedeira. O **Comportamento Mó-**

vel corresponde ao comportamento herdado do sistema de agentes móveis escolhido, através de uma classe abstrata própria, como já comentado em R5. A **Interface de Usuário** é o canal de comunicação entre o agente e o seu usuário. Através dela poderão ser fornecidas as informações iniciais necessárias para a criação do agente e dos mecanismos (argumentos de criação). Por exemplo, um dos dados de criação que poderá ser fornecido através dessa interface é a localização da base de conhecimento do agente, que representa a parcela declarativa de sua programação. Usualmente, essa base inicial de conhecimento estará armazenada em um repositório persistente como, por exemplo, um sistema de arquivos ou um servidor http.

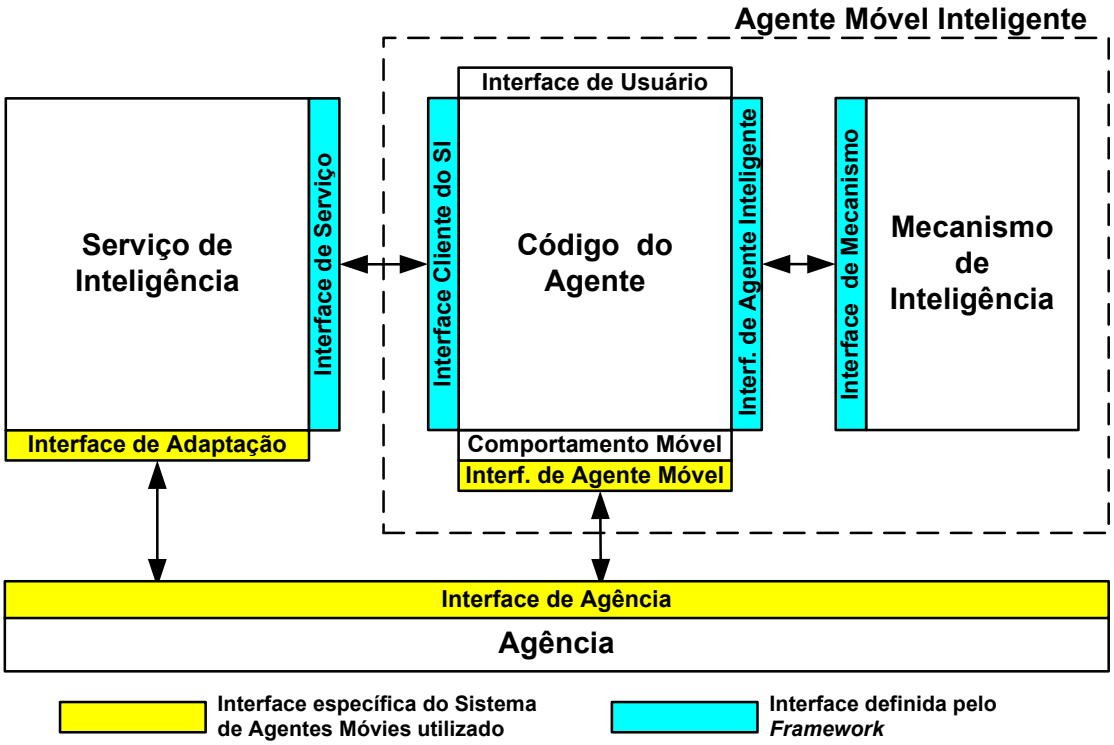


Figura 39: Principais componentes do *Framework de Inteligência*

6.4 Análise dos Requisitos

Nesta seção discute-se como os requisitos definidos em 6.2 serão satisfeitos pelo *framework* proposto.

O requisito R1 não necessita de maiores esclarecimentos.

O requisito R2 exige que o mecanismo seja visível ao agente, isto é, que o agente possua uma referência para o mecanismo.

Da mesma forma, os requisitos R3 e R4 exigem que o agente seja visível ao mecanismo, ou seja, que o mecanismo possua uma referência para o agente. No entanto, de acordo com o requisito R5, o mecanismo deve ser facilmente utilizado com os diversos sistemas de agentes móveis baseados em Java, o que significa que não deve ser necessário alterar o seu código e recompilá-lo para utilizá-lo com cada um desses sistemas. Porém, os agentes dos diversos sistemas de agentes móveis não possuem uma interface comum, o que impede que o mecanismo possa referenciá-los diretamente de modo único. Portanto, é necessária uma solução capaz de adaptar o comportamento móvel, esperado pelas agências da plataforma escolhida, ao comportamento de cliente esperado pelo mecanismo. Para resolver esse problema, propõe-se a utilização do padrão de projeto *Adapter* apresentado em GAMMA et al. (1995, p.139), em sua versão bidirecional. Pela utilização desse padrão, é possível converter a interface de uma classe já existente em outra interface, esperada por seus clientes.

A Figura 40 ilustra a aplicação do padrão *Adapter* (bidirecional) à interação *agente-mecanismo*, onde a classe `Agente Móvel Inteligente` adapta o comportamento de mobilidade herdado do sistema de mobilidade utilizado, através da classe abstrata `Agente Móvel`, ao comportamento esperado pelos mecanismos de inteligência, definido pela interface `Agente Inteligente`. Deve-se notar, ainda, que não seria possível substituir a interface `Agente Inteligente` por uma classe abstrata pois a linguagem Java não suporta herança múltipla de implementação.

O relacionamento indireto entre o agente e o mecanismo, estabelecido pela interface `Agente Inteligente`, faz com que a solução proposta também cumpra o requisito R12.

Embora cada mecanismo específico possua seu próprio comportamento, definido por seu conjunto de operações, algumas partes desse comportamento serão comuns a todos eles, como por exemplo, o registro de cliente (agente), o controle do fluxo (*thread*) de execução, a recepção de percepções e a interação com o agente para solicitar um sensoriamento ou atuação. Para evitar a duplicação desnecessária de código e de dados, pode-se utilizar o padrão de projeto *Template Method* (GAMMA et al., 1995, p.325).

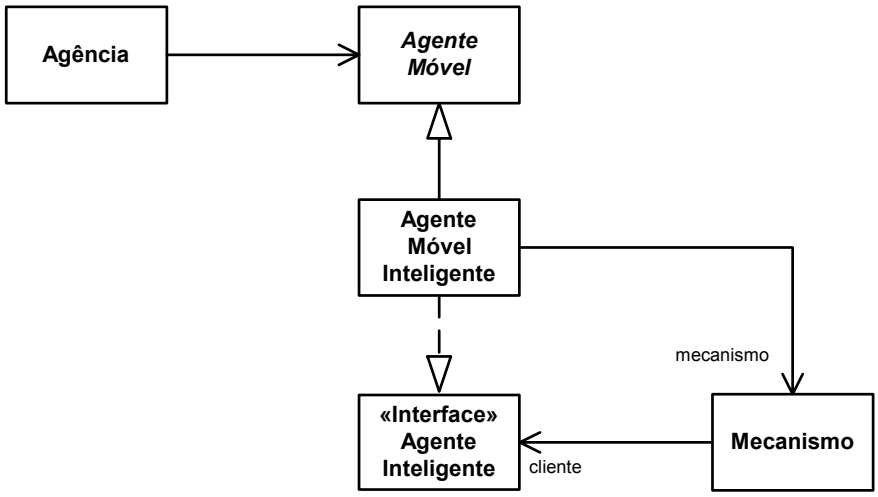


Figura 40: Padrão de projeto *Adapter* aplicado à interação *agente - mecanismo*

Como mostrado na Figura 41, o padrão *Template Method* define o esqueleto de um algoritmo em um método (`métodoGabarito`) e posterga a definição de alguns de seus passos para as subclasses (operações primitivas abstratas). Portanto, esse padrão permite que as subclasses definam certos passos do algoritmo sem mudar a estrutura do mesmo.

Dessa forma, a parcela invariante do comportamento dos mecanismos pode ser concentrada numa classe abstrata, denominada `Mecanismo Abstrato`, que também define a assinatura de todas as operações primitivas abstratas que um `Mecanismo Concreto` deve implementar para especializar o comportamento padrão. A classe `Mecanismo Concreto` também poderá acrescentar suas próprias operações.

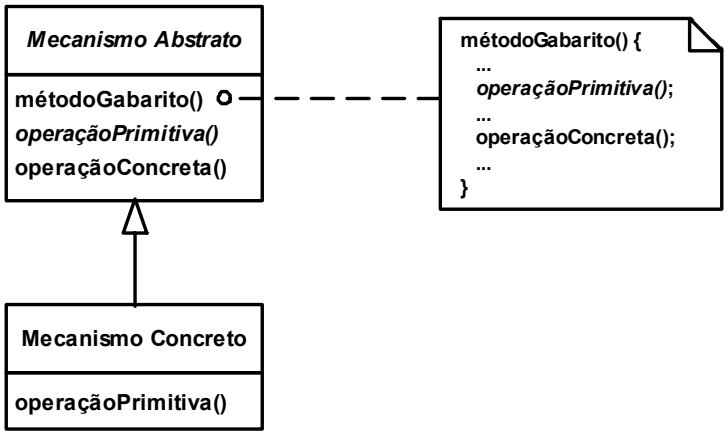


Figura 41: Padrão de projeto *Template Method* aplicado ao relacionamento *Mecanismo Abstrato – Mecanismo Concreto*

A Figura 42 mostra como fica a interação *agente - mecanismos* com a introdução do padrão *Template Method*. O agente utilizará as operações próprias do `Mecanismo Concreto` para inicializá-lo. Durante a inicialização, na agência de origem, o `Mecanismo Concreto` deverá ser informado sobre sua base inicial de conhecimento e sobre todos os componentes que se fizerem necessários ao seu funcionamento. Ainda nessa fase, o agente deverá registrar-se como cliente do mecanismo, utilizando uma operação definida pelo `Mecanismo Abstrato`. Isso é necessário para que o agente seja visível ao mecanismo. Após a inicialização, o agente poderá iniciar e interromper a execução do `Mecanismo Concreto` através de operações fornecidas pelo `Mecanismo Abstrato` exclusivamente para essas finalidades. Quando o mecanismo estiver em execução, o agente poderá enviar suas percepções ao `Mecanismo Concreto` (R2) através de operações que este herdou do `Mecanismo Abstrato`. Cada percepção será composta por um conjunto de fatos, que poderão conter informações sobre o estado interno do agente ou sobre o estado de seu ambiente (agência). As percepções relativas ao estado interno do agente serão geradas pelo código (procedural) do agente. As percepções sobre o estado do ambiente serão enviadas pelo serviço de inteligência local ao código (procedural) do agente, que as transferirá para o mecanismo. Essa "ponte", realizada pelo código procedural, é importante, pois permite que o agente selecione, caso esteja utilizando mais de um mecanismo de IA, para qual deles as percepções serão enviadas.

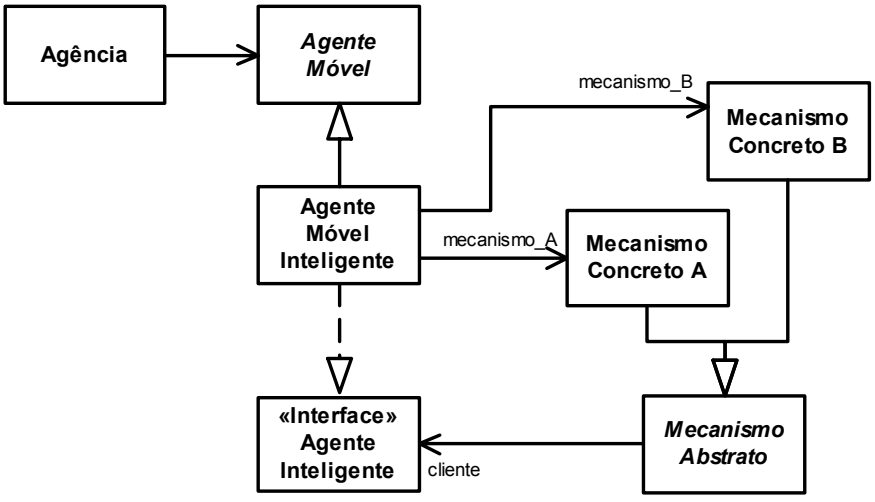


Figura 42: Padrão de projeto *Template Method* aplicado aos mecanismos

Durante sua execução, o mecanismo poderá solicitar, através de operações oferecidas pela interface `Agente Inteligente`, informações adicionais para completar sua tarefa (R3)

e/ou sugerir a execução de uma ação (R4). O agente poderá honrar uma solicitação de informação adicional de três formas distintas: (1) ele mesmo poderá possuir e fornecer a informação desejada; (2) ele poderá solicitá-la a outro mecanismo de IA que também esteja sendo utilizado ou, então, (3) poderá solicitá-la à agência hospedeira, através do serviço de inteligência local. Do mesmo modo, a execução de uma ação, sugerida ao agente pelo mecanismo, poderá ser realizada pelo próprio agente, por um outro mecanismo de IA ou pelo SI da agência hospedeira.

Portanto, das descrições anteriores, fica claro que o agente deve ser visível ao serviço de inteligência e vice-versa. Para manter o código procedural do agente desacoplado da implementação concreta do serviço de inteligência e vice-versa, além de adaptar o comportamento pré-existente do sistema de mobilidade ao novo comportamento do serviço de inteligência esperado pelos AMIs, utiliza-se, novamente, o padrão de projeto *Adapter* (GAMMA et al., 1995), conforme mostrado na Figura 43. Nessa figura, a classe `Serviço de Inteligência`, adapta o comportamento da agência ao comportamento esperado pelos AMIs.

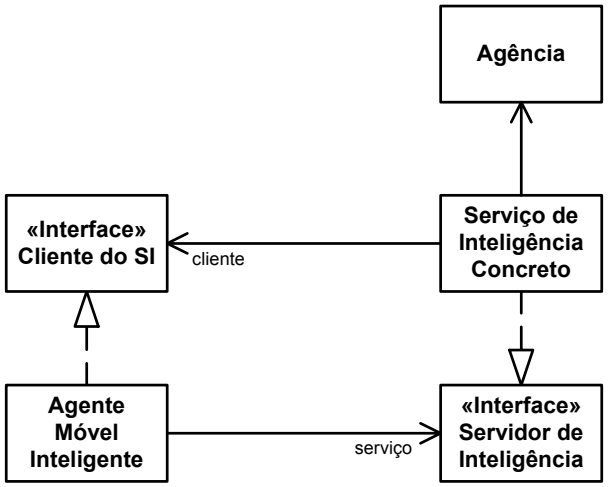


Figura 43: Padrão de projeto *Adapter* aplicado à interação Agente - *Serviço de Inteligência*

Através das operações da interface `Servidor de Inteligência`, o agente poderá registrar-se como cliente do `Serviço de Inteligência Concreto` da agência hospedeira e solicitar informações adicionais ou a execução de uma determinada ação. Por meio das operações da interface `Cliente do SI`, o serviço de inteligência poderá notificar ao código

procedural do agente a ocorrência de eventos no domínio da agência hospedeira. Normalmente, o monitoramento de um evento específico, por parte do serviço de inteligência, é requisitado antecipadamente, pelo próprio agente.

Para satisfazer o requisito R6, propõe-se que, para cada tipo de mecanismo, seja definido um conjunto de **Interfaces de Representação de Conhecimento** que especifiquem um grupo bem definido e coeso de operações que **objetos "puros" de conhecimento** deverão implementar. Trata-se, portanto, da aplicação do princípio de projeto *Interface* - "*programe para uma interface e não para uma implementação*" - descrito em GAMMA et al. (1995, p.17-18), que reduz as dependências entre subsistemas. As operações presentes nas interfaces propostas devem permitir o acesso aos componentes constitutivos de cada uma das entidades de conhecimento presentes no formalismo de representação adotado. Além disso, tais interfaces não devem possuir nenhuma característica que as vincule a um tipo específico de mecanismo; por isso os objetos criados a partir delas são chamados de "puros". *Informações operacionais*, que especifiquem como o conhecimento representado por esses objetos será utilizado pelos mecanismos deverão ser acrescentadas como adornos às entidades básicas definidas por essas interfaces.

Por exemplo, para o caso do mecanismo de inferência, devem ser criadas interfaces para todos os conceitos da Lógica de Primeira Ordem que se apliquem à representação na forma de regras, como, por exemplo: *variável*, *constante*, *termo*, *predicado, literal* e *regra*. As operações definidas nessas interfaces devem permitir o acesso aos componentes constitutivos de cada um desses conceitos, tais como o predicado e a lista de termos de um literal e todos os literais que compõem as expressões antecedente e conseqüente de uma regra. Por outro lado, a informação de que um predicado representa um *evento*, um *sensor* ou um *atuador*, deverá ser acrescentada como adorno à interface associada ao predicado.

Embora possa parecer pouco produtivo transferir para o desenvolvedor do agente a responsabilidade pela implementação dos objetos de conhecimento necessários à tradução da base de conhecimento para o formato executável, a flexibilidade de tal enfoque supera o trabalho adicional pois, uma vez criados, esses objetos são facilmente reutilizados. Além disso, a quantidade de código e de dados transportados pelo agente também pode ser reduzida, visto que as tarefas de análise sintática (*parsing*) e de tradução são realizadas fora do mecanismo de inferência, podendo ser delegadas a um componente específico que não precisará, obrigatori-

amente, ser transportado. Portanto, o agente necessitará de uma referência para o objeto tradutor somente na fase de inicialização.

Generalizando essa idéia, pode-se estabelecer a seguinte diretriz de projeto: *todos os procedimentos necessários somente na fase de inicialização do agente, na sua agência nativa, deverão ser encapsulados em um único objeto, ao qual o agente delega essas tarefas e que não precisará ser transportado*.

Como mostrado na Figura 44, o relacionamento entre o agente e o objeto iniciador segue o princípio de projeto *Delegação* documentado em GAMMA et al. (1995, p.20).

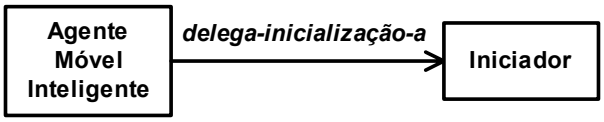


Figura 44: Princípio de projeto *Delegação* aplicado ao relacionamento agente - iniciador

Especificamente, no caso do mecanismo de inferência, esse é um ponto no qual as ferramentas e bibliotecas de classes analisadas – JESS (Friedman-Hill, 2001), ABLE (IBM, 2000), CIAgent (BIGUS; BIGUS, 1998) e ABE (IBM, 1997) – falham: primeiro, ao vincularem as interfaces de seus componentes a formatos externos de representação não executáveis e, segundo, ao incorporarem, de forma inseparável, as tarefas de análise sintática e de tradução desses formatos aos componentes responsáveis pela inferência.

A aderência aos requisitos R7, R8, R9 e R10 está estreitamente relacionada aos mecanismos concretos. No que se segue serão apresentadas algumas diretrizes gerais a serem seguidas para atendê-los.

Para satisfazer R7, a representação de conhecimento utilizada internamente deve ser compacta. Informações compartilhadas por várias sentenças devem ser representadas uma única vez, para reduzir o tamanho do código e dados. Deve existir uma preocupação especial com a *implementação*, evitando-se a migração desnecessária de código e dados em tempo de execução. No que se refere à redução de dados, o padrão de projeto *Flyweight* (GAMMA et al., 1995, p.195) deverá ser utilizado. Esse padrão, mostrado na Figura 45, tem por objetivo, justamente, usar o compartilhamento para suportar eficientemente grandes quantidades de objetos de granularidade fina (*flyweights*), garantindo uma economia de armazenamento, e por conseqüência do custo de migração.

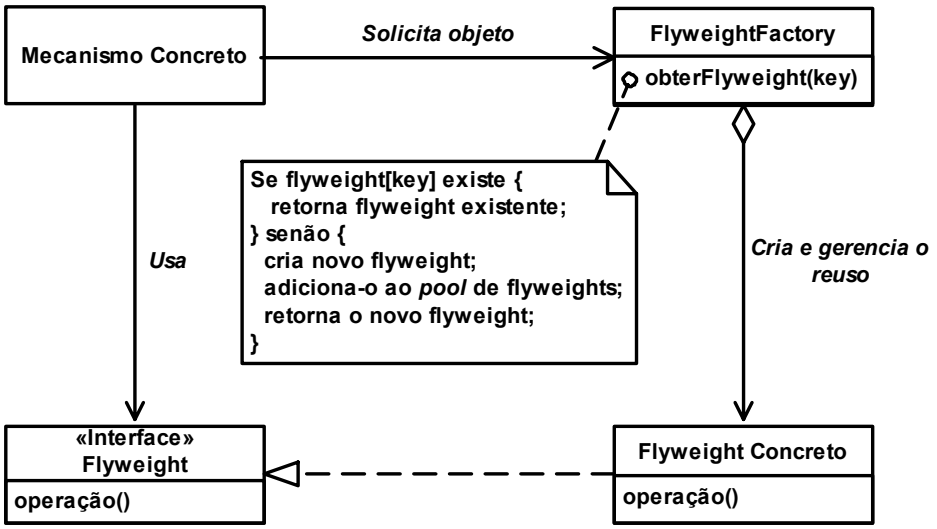


Figura 45: Padrão de projeto *Flyweight* aplicado aos objetos compartilháveis de um mecanismo

Para atender R8, os algoritmos utilizados pelos mecanismos devem ser computacionalmente eficientes (tratáveis). Para satisfazer R9 devem ser evitados, nas representações internas de conhecimento, por exemplo, descrições e/ou comentários codificados no formato de texto puro. Para atender R10, as principais funcionalidades dos mecanismos devem ser isoladas e atribuídas a componentes distintos, com interfaces bem definidas. O padrão de projeto *Strategy* (GAMMA et al., 1995, p.305) é adequado a este propósito, pois permite definir uma família de algoritmos, encapsular cada um deles e torná-los intercambiáveis, independentemente dos clientes que os utilizam. A Figura 46 exemplifica o uso desse padrão no contexto de um mecanismo concreto.

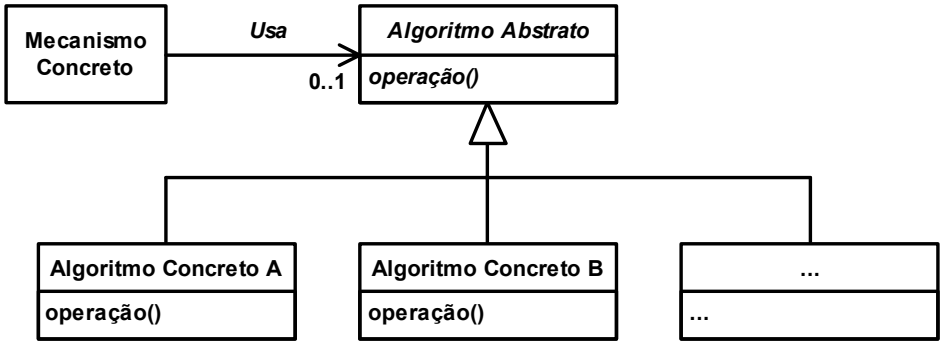


Figura 46: Padrão de projeto *Strategy* aplicado às funcionalidades intercambiáveis de um mecanismo

No caso específico do mecanismo de inferência, a utilização do algoritmo de casamento ***Rete*** (FORGY, 1982) descrito em 4.6, não só contribui para aliviar o custo computacional do processo (R7), ao recordar-se dos resultados das unificações realizadas nos ciclos de inferência anteriores, como também elimina a duplicidade de estruturas de dados entre regras (Friedman-Hill, 2001), atendendo a R8. Uma vez que as regras são compiladas em uma rede de nós, ele também satisfaz o requisito R9. Para satisfazer R10, as principais funcionalidades do mecanismo foram atribuídas a componentes distintos, como mostra a Figura 47.

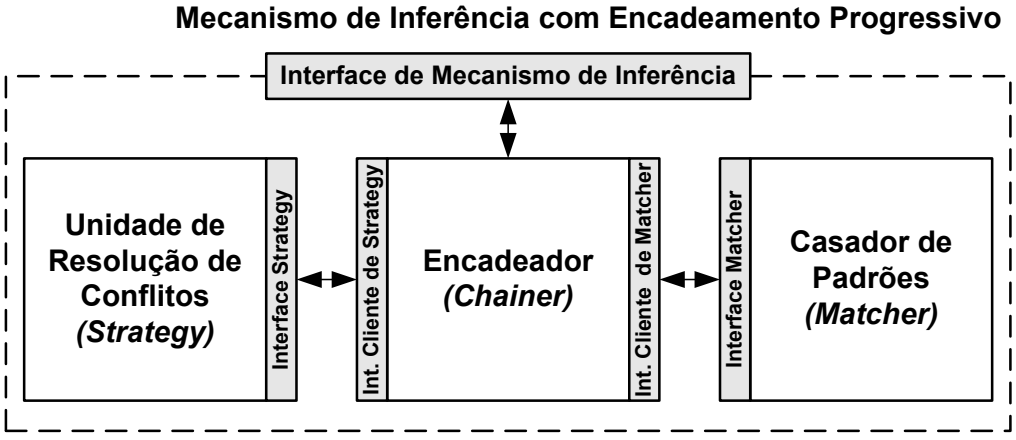


Figura 47: Principais componentes do *Mecanismo de Inferência com Encadeamento Progressivo*

O *Encadeador* encapsula o *algoritmo de controle da inferência com encadeamento progressivo*, o *Casador de Padrões* encapsula o *algoritmo de casamento* e a *Unidade de Resolução de Conflitos* encapsula a *estratégia de resolução de conflitos*. O agente utiliza as operações da *Interface de Mecanismo de Inferência* para inicializar e controlar a execução do processo de inferência. Os componentes *Casador de Padrões* e *Unidade de Resolução de Conflitos* devem ser fornecidos ao mecanismo durante o processo de criação do agente. O *Encadeador* utilizará as operações das interfaces desses componentes para iniciá-los e para solicitar seus serviços durante um ciclo de inferência.

A separação da lógica de controle da inferência dos algoritmos de casamento e de resolução de conflitos não só satisfaz R10 como também contribui para a redução do código e dos dados transportados pelo agente (R7), permitindo diversas combinações. Por exemplo, por ser genérico, o *Encadeador* poderá ser oferecido pela agência hospedeira, enquanto que o agente leva consigo somente os seus próprios *Casador de Padrões* e a *Unidade de Resolução*

de Conflitos. Contudo, se uma agência não oferecer o *Encadeador*, ele poderá ser "transferido" sob demanda, da agência nativa do agente.

Como forma de reduzir ainda mais a quantidade de código e de dados transportados, o *Casador de Padrões* também pode ser decomposto em componentes mais especializados, como mostra a Figura 48.

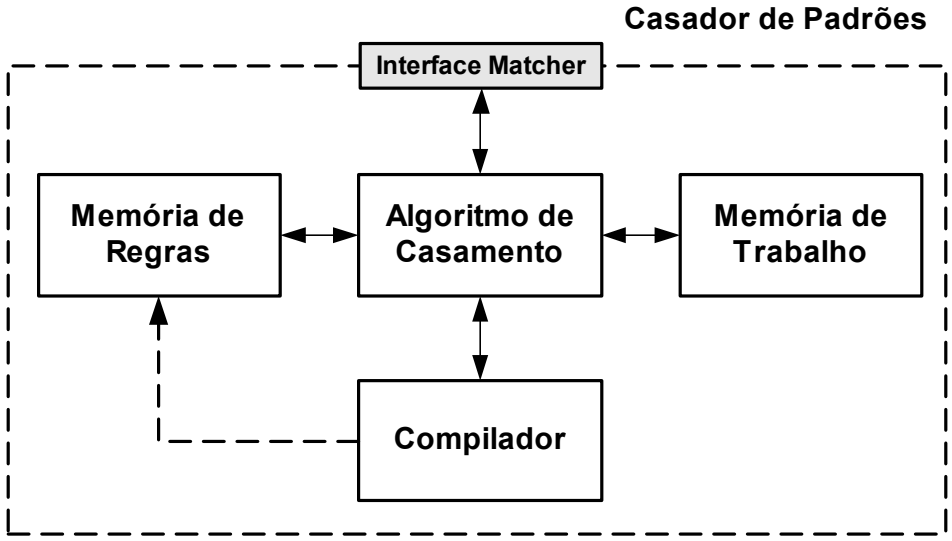


Figura 48: Principais componentes de um *Casador de Padrões*

O *Compilador* encapsula o código que traduz as regras e os fatos transferidos para o mecanismo de inferência no formato de objetos de conhecimento para o formato interno de representação utilizado pelo *Casador de Padrões*, de modo a construir a *Memória de Regras e a Memória de Trabalho*. O *Algoritmo de Casamento* é responsável por identificar, quando solicitado, o conjunto de regras presentes na *Memória de Regras* que se casam com os fatos presentes na *Memória de Trabalho*. Somente a interface `Matcher` do *Casador de Padrões* deverá ser especificada, uma vez que as interfaces dos outros componentes dependem do algoritmo de casamento escolhido. Uma vez que o *Encadeador* (do mecanismo de inferência) não tem acesso direto aos componentes do *Casador*, a interface `Matcher` representa uma "fachada" para a utilização de todos eles. A Figura 48 também ilustra uma utilização do padrão de projeto *Façade* (GAMMA et al., 1995, p.185).

Isolar e atribuir as principais funcionalidades do *Casador de Padrões* a componentes distintos, também reduz a quantidade de código e de dados transportados de duas formas: primeiro, o código e os dados do *Compilador* não precisam migrar, visto que sua funcionali-

dade só é necessária durante o processo de criação do agente, quando as regras na base de conhecimento deverão ser convertidas para o formato interno de representação utilizado pelo *Casador*; segundo, facilita a identificação das operações e dos atributos atribuídos ao *Algoritmo de Casamento*, à *Memória de Regras* e à *Memória de Trabalho* estritamente relacionados com a fase de casamento. Dessa forma, por exemplo, as estruturas de dados da *Memória de Regras* poderão ser mais facilmente projetadas para representarem, uma única vez, as informações compartilhadas por diversas regras. Como forma de reduzir a quantidade de informação presente na *Memória de Trabalho*, suas estruturas de dados deverão ser projetadas de forma a remover, automaticamente, fatos que sejam exclusivamente dependentes da agência hospedeira corrente. Para tanto, durante o processo de compilação, todos os antecedentes de uma regra, que representem tais fatos, deverão ser marcados pelo *Compilador* como **transientes**.

Para satisfazer R11, propõe-se que os elementos do *framework* sejam agrupados em subsistemas relacionados de acordo com a arquitetura clássica de 3 camadas (LARMAN, 1998). Essa arquitetura desvincula os subsistemas do domínio do problema dos subsistemas de interface de usuário e de armazenamento persistente, atendendo R11.

6.5 Arquitetura do *Framework de Inteligência*

A Figura 49 apresenta a arquitetura proposta para o *Framework de Inteligência*, os subsistemas identificados e as relações entre eles. Existem quatro tipos de subsistemas: (1) subsistemas cujos elementos (classes e interfaces) são fornecidos pelo sistema de agentes móveis adotado, por exemplo, o *Aglets* ou o *Grasshopper*: *Plataforma de Mobilidade* e *Comportamento Móvel*; (2) subsistemas que representam o núcleo do *Framework de Inteligência*, cujos elementos são desenvolvidos neste trabalho: *Mecanismo Abstrato* e *Interfaces do Serviço de Inteligência*; (3) subsistemas específicos de cada mecanismo concreto adicionado ao *framework*: *Interface de Representação de Conhecimento* e *Mecanismo Concreto*; e (4) subsistemas cujos elementos deverão ser desenvolvidos pelo usuário do *framework*, para personalizar o agente e as agências: *Lógica do Agente*, *Interface de Usuário* e *Serviço de Inteligência*.

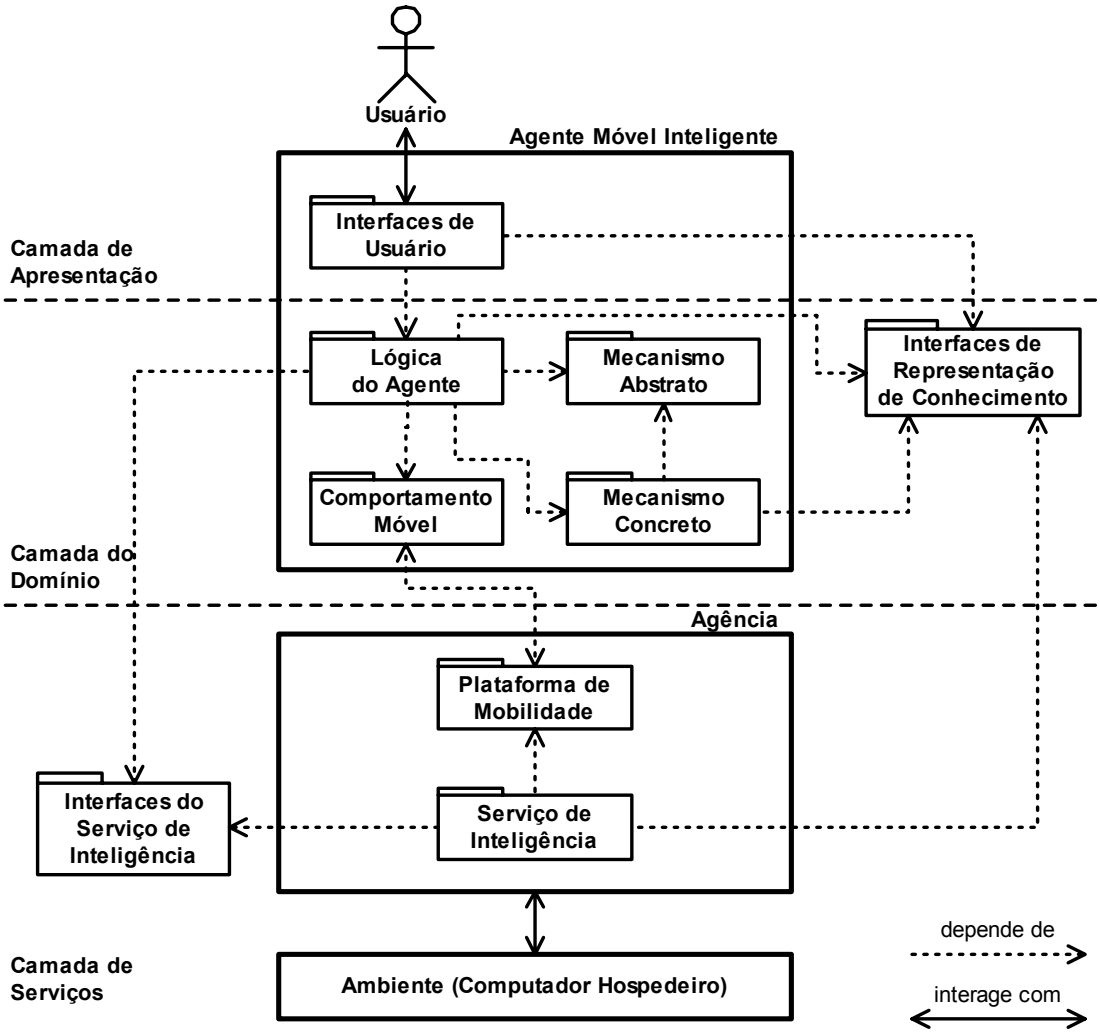


Figura 49: Arquitetura proposta para o *Framework de Inteligência*

O subsistema *Comportamento Móvel* contém as classes necessárias para construir um agente móvel de acordo com um sistema de agentes móveis escolhido. O subsistema *Plataforma de Mobilidade* reúne as interfaces que o sistema de mobilidade escolhido disponibiliza para que a interação *agente móvel - agência* seja possível. O subsistema *Interfaces de Representação de Conhecimento* agrupa as interfaces que os objetos "puros" de conhecimento devem implementar. Com o objetivo de tornar a utilização do *framework* mais imediata, esse subsistema também contém uma implementação básica dessas interfaces. O subsistema *Interfaces do Serviço de Inteligência* agrupa as interfaces que um SI concreto deve implementar, de acordo com o sistema de agentes móveis utilizado, de forma a garantir as interações entre ele e o AMI. Vale notar que os componentes de *Interfaces do Serviço de Inteligência* não dependem dos elementos de *Interfaces de Representação de Conhecimento*. Isto é importante

para que os elementos do primeiro possam ser especificados de forma genérica, independente dos objetos de conhecimento utilizados pelos diversos mecanismos concretos. O subsistema *Serviço de Inteligência* representa um serviço de inteligência concreto. O subsistema *Lógica do Agente* contém as classes e interfaces, que o desenvolvedor do agente deve implementar, correspondentes à programação procedural do mesmo, responsável, por exemplo, pela inicialização do agente. O subsistema *Mecanismo Abstrato* define um pequeno conjunto de classes e interfaces que permitem ao agente configurar, controlar a execução e enviar percepções aos mecanismos concretos. O subsistema *Mecanismo Concreto* reúne os componentes que implementam um mecanismo de inteligência específico. Para cada mecanismo concreto agregado ao *framework*, um novo subsistema *Mecanismo Concreto* deve ser adicionado. Tanto o subsistema *Mecanismo Abstrato* quanto o *Mecanismo Concreto* não dependem dos subsistemas relativos à mobilidade.

Os componentes dos subsistemas *Interfaces do Serviço de Inteligência*, *Interfaces de Representação de Conhecimento* e *Serviço de Inteligência* devem estar disponíveis em todas as agências do *Ambiente Distribuído de Agentes* que possuam um SI instalado.

6.6 Considerações Finais

Neste capítulo foram definidos os requisitos que o *framework* deve satisfazer para incorporar mecanismos de IA a agentes móveis. Os requisitos propostos representam um compromisso entre flexibilidade e austeridade de código e dados. Da análise desses requisitos, identificaram-se alguns padrões de projeto que devem ser usados no desenvolvimento do *framework* para que seja possível atendê-los. Durante a análise também se propôs que (1) para reduzir a quantidade de código transportado e garantir a segurança da agência, parte das funcionalidades do *framework* (sensoriamento e atuação) pode ser transferida para as agências, na forma de um serviço de inteligência; (2) a troca de conhecimento entre o agente e os mecanismos de inteligência deve ser realizada na forma de objetos de conhecimento, que implementam um conjunto de interfaces de representação de conhecimento.

Finalmente, foi definida uma arquitetura para o *framework*. Cada subsistema dessa arquitetura representa uma especialização importante e, como eles são fracamente acoplados

entre si, especialistas em um subsistema poderão desenvolver novos componentes sem que para isso precisem ter conhecimentos sobre como desenvolver componentes de outros subsistemas. Por exemplo, especialistas em IA poderão se concentrar no desenvolvimento dos mecanismos de inteligência, enquanto especialistas em mobilidade poderão tratar somente dos aspectos de migração; especialistas em interface homem-máquina poderão se concentrar nos aspectos relativos à camada de apresentação, especialistas na lógica da aplicação poderão se preocupar somente com a criação de sensores e atuadores; especialistas na lógica do negócio poderão se concentrar na criação das regras do negócio enquanto especialistas em *engenharia de conhecimento* poderão se ater na criação, análise sintática e tradução de e para os objetos de representação de conhecimento.

Os elementos dos subsistemas que compõem o núcleo do *framework* proposto são desenvolvidos no Capítulo 7, de acordo com o processo descrito no Capítulo 5. No Capítulo 8 são apresentados os elementos que compõem os subsistemas específicos ao mecanismo concreto avaliado.

Capítulo 7

Desenvolvimento do Núcleo do

Framework de Inteligência

7.1 Introdução

Este capítulo apresenta o desenvolvimento orientado a objetos do *Framework de Inteligência* delineado no Capítulo 6. Uma vez que este trabalho não tem por objetivo ilustrar a aplicação do processo de desenvolvimento de software orientado a objetos descrita no Capítulo 5, apresentam-se somente os modelos resultantes de sua utilização. Para cada subsistema presente na arquitetura proposta no Capítulo 6, são apresentados os modelos pertinentes.

Para evitar uma composição desnecessária de idiomas que, fatalmente, ocorreria na fase de implementação, visto que as expressões e classes nativas da linguagem Java possuem denominação em inglês, decidiu-se nomear as classes e interfaces do *Framework de Inteligência*, bem como seus atributos e operações, nesse mesmo idioma.

A Seção 7.2 introduz os casos de uso identificados e a Seção 7.3 descreve a seqüência de desenvolvimento adotada. As Seções 7.4 e 7.5 apresentam os demais modelos construídos para os subsistemas que constituem o núcleo do *framework* proposto. Especificamente, a Seção 7.4 trata dos modelos para o subsistema *Mecanismo Abstrato* e a Seção 7.5, dos modelos para o subsistema *Interfaces do Serviço de Inteligência*.

7.2 Modelo de Casos de Uso

Para melhorar a compreensão sobre a utilização e as funcionalidades desejadas para o *Framework de Inteligência*, foi utilizada a técnica dos casos de uso descrita em 5.3.1. Conforme mostrado na Figura 50, os seguintes casos de uso foram identificados: *Configurar Mecanismo* e *Processar Percepção*.

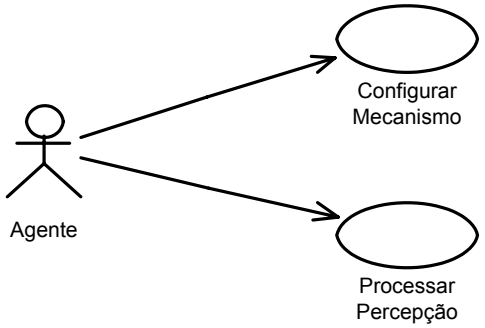


Figura 50: Diagrama de casos de uso para o *Framework de Inteligência*

Configurar Mecanismo corresponde ao diálogo entre o agente e um mecanismo, no qual o Agente define os algoritmos a serem utilizados, bem como informa sua base inicial de conhecimento.

Processar Percepção corresponde ao diálogo entre o agente e um mecanismo, no qual o Agente informa sobre uma nova percepção (conjunto de novos fatos) e solicita que o mecanismo derive as conseqüências lógicas decorrentes.

A seguir apresentam-se as descrições essenciais detalhadas de cada um desses casos de usos, explicitando-se seus cursos básicos e alternativos de eventos, quando existirem.

Caso de Uso: ***Configurar Mecanismo***

Curso básico de eventos:

Ações do Agente	**Resposta do Mecanismo**
1. Este caso de uso começa quando o agente, em sua inicialização, decide configurar um mecanismo para uso futuro.	
2. O agente informa ao mecanismo os seguintes elementos (em qualquer ordem):	3. Entra em estado de espera de execução (mecanismo configurado).

2.1. os algoritmos a serem utilizados;

2.2. sua base de conhecimento (BC), na forma de objetos de conhecimento.

Com base nessa descrição, o diagrama de estados mostrado na Figura 51 pode ser identificado para o caso de uso em questão:

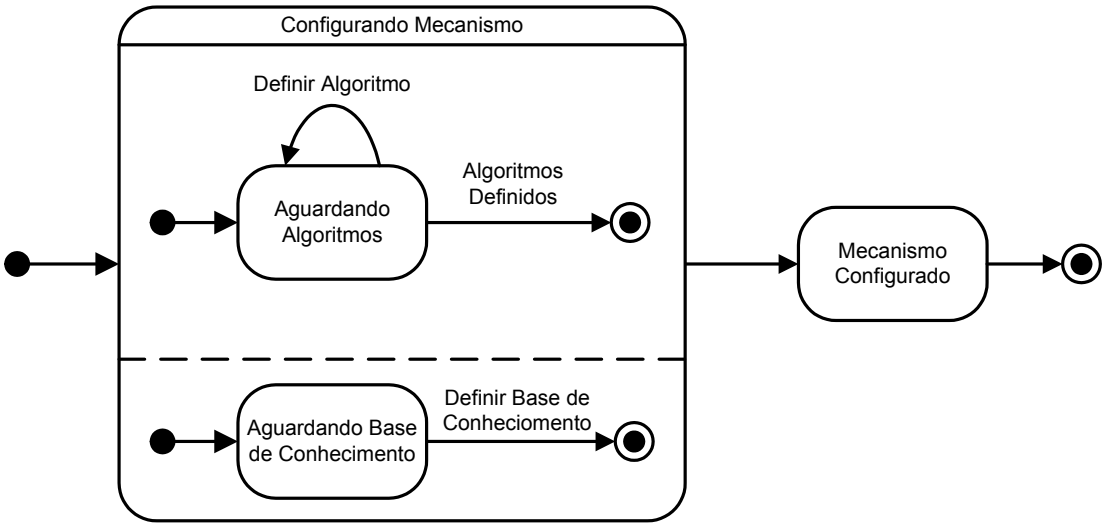


Figura 51: Diagrama de estados para *Configurar Mecanismo*

Caso de Uso: ***Processar Percepção***

Curso básico de eventos

Ações do Agente	**Resposta do Mecanismo**
1. Este caso de uso começa quando o agente deseja utilizar um mecanismo para deduzir as conseqüências lógicas de suas percepções à cerca de seu ambiente.	
2. Durante sua execução, o agente inicia a execução do mecanismo.	3. Se a BC ainda não foi tratada: 3.1. Traduz os objetos de conhecimento presentes na BC para sua representação interna. 3.2. Armazena esses objetos de conhecimento traduzidos em sua BC interna. 4. Deduz todas as conseqüências lógicas decorrentes dos fatos iniciais presentes na sua BC interna.

5. Ao longo de sua execução, o agente informa ao mecanismo suas percepções a respeito de seu ambiente.

6. Acrescenta cada nova percepção a uma fila de percepções a serem tratadas.

7. Para cada percepção presente na fila o mecanismo faz o seguinte:

   7.1. Converte a percepção em um conjunto de sentenças expressas na sua forma interna de representação de conhecimento;

   7.2. Adiciona as sentenças que representam a percepção à sua BC interna.

   7.3. Deduz todas as conseqüências lógicas decorrentes do estado atual de sua BC interna. Durante esse processo o mecanismo pode:

      7.3.1. Adicionar novas sentenças à sua BC interna (conclusões);

      7.3.2. Remover sentenças de sua BC interna (conclusões);

      7.3.3. Solicitar informações adicionais ao agente (sensoriamento).

      7.3.4. Sugerir que o agente execute alguma ação sobre o ambiente (atuação).

8. Ao receber uma solicitação de informações adicionais, o agente pode:

   8.1. Fornecê-las por si mesmo;

   8.2. Encaminhar a solicitação para algum de seus outros mecanismos, ou

   8.3. Encaminhá-la a um serviço de inteligência oferecido pela agência hospedeira.

9. Ao receber uma solicitação para executar uma ação, o agente pode:

   9.1. Executá-la por si mesmo;

   9.2. Encaminhar a solicitação para algum de seus outros mecanismos, ou

   9.3. Encaminhá-la a um serviço de inteligência oferecido pela agência hospedeira

10. Quando todas as conseqüências decorrentes da percepção em tratamento forem deduzi-

	das, informa ao agente que a percepção foi processada.
11. Quando julgar conveniente, o agente encerra o processamento das percepções.	12. Encerra sua execução e entra em estado de espera de nova execução. A partir desse momento deixa de aceitar e de processar novas percepções.

Cursos Excepcionais:

Passo 3: Se a base de conhecimento ainda não foi especificada o mecanismo deve indicar um erro de configuração.

Passo 4: Se algum algoritmo, necessário para o processamento das percepções, ainda não foi especificado o mecanismo deve indicar um erro de configuração.

7.3 Seqüência de Desenvolvimento

O desenvolvimento do *Framework de Inteligência* foi realizado em cinco ciclos de desenvolvimento, centrados nos casos de uso descritos anteriormente.

No primeiro ciclo, desenvolveram-se os elementos que compõem o núcleo do *framework* proposto, isto é, os subsistemas *Mecanismo Abstrato* e *Interfaces do Serviço de Inteligência*. Foram tratadas as operações genéricas relacionadas aos casos de uso *Configurar Mecanismo* e *Processar Percepção*.

No segundo ciclo tratou-se das operações específicas ao *Mecanismo Concreto* de Inferência, relacionadas ao caso de uso *Configurar Mecanismo*, e a uma primeira versão de *Processar Percepção*. Nessa primeira versão as seguintes simplificações foram adotadas: durante o processo de inferência o mecanismo não podia solicitar informações adicionais ao agente e as conclusões decorrentes desse processo levavam somente à adição ou remoção de fatos da memória de trabalho. Ou seja, os requisitos de sensoriamento e atuação não foram levados em conta. Além disso, nos antecedentes e nos conseqüentes das regras só eram aceitos literais positivos.

No terceiro ciclo dedicou-se à implementação de uma segunda versão de *Processar Percepção*, na qual incluiu-se a funcionalidade de sugestão de ações e a possibilidade do predicado de um literal representar um evento.

No quarto ciclo de desenvolvimento agregou-se a capacidade do mecanismo, durante o processo de inferência, solicitar informações adicionais aos agentes (sensores) e, finalmente, no quinto ciclo permitiu-se a ocorrência de literais negativos nas expressões antecedentes das regras, através da estratégia de *negação por falha*, bem como relacionamentos de igualdade entre dois termos.

Este capítulo apresenta somente os resultados do primeiro ciclo de desenvolvimento. Os resultados dos demais ciclos de desenvolvimento estão apresentados no Capítulo 8.

7.4 Subsistema Mecanismo Abstrato

Nesta seção apresentam-se os diversos modelos que descrevem a análise e o projeto do subsistema *Mecanismo Abstrato* do *Framework de Inteligência* proposto.

7.4.1 Modelo de Conceitos

Com base nas descrições dos casos de uso, identificou-se o modelo de conceitos mostrado na Figura 52.

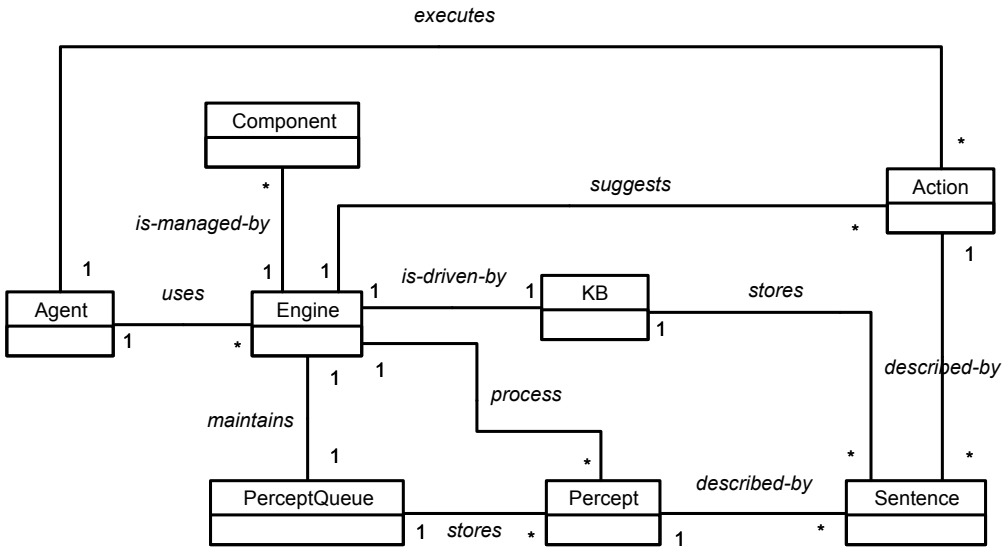


Figura 52: Modelo de conceitos para o mecanismo abstrato

7.4.2 Modelo de Comportamento Sistêmico

Uma vez que o agente também é um sistema de software, o modelo de comportamento sistêmico, criado a partir das descrições dos casos de uso, ajuda a definir não só o conjunto de operações que o mecanismo abstrato (servidor) deve apresentar, mas também o conjunto de operações que o agente (cliente) deve oferecer para que ambos possam interagir. Esses conjuntos de operações correspondem às interfaces de programação de aplicações (API) desses componentes, e representam os *serviços* que cada um oferece. As operações que um servidor pode invocar de seu cliente são chamadas *operações de retorno* (*callbacks*), e invertem os papéis de cliente e servidor.

As operações que o mecanismo abstrato deve oferecer ao agente foram identificadas a partir das descrições dos casos de uso *Configurar Mecanismo* e *Processar Percepção*. Somente em *Processar Percepção* surge a necessidade das operações de retorno, ou seja, dos serviços que o agente deve oferecer ao mecanismo. Os detalhes de cada um desses conjuntos de operações são descritos a seguir.

7.4.2.1 Operações oferecidas pelo mecanismo abstrato

Da análise dos casos de uso *Configurar Mecanismo* foram identificadas as operações de sistema mostradas na Figura 53.

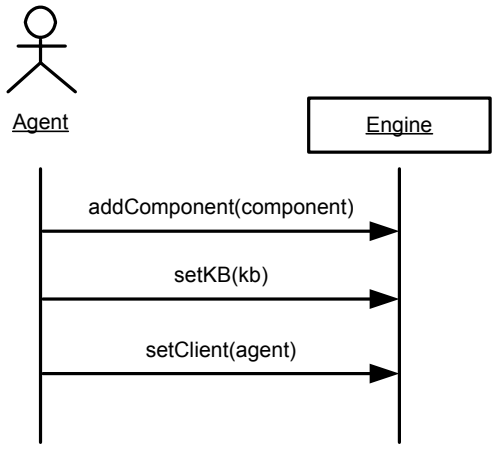


Figura 53: Operações para a configuração dos mecanismos

A operação `addComponent(component)` permite informar o mecanismo sobre os componentes que implementam os algoritmos a serem utilizados. O parâmetro `component` corresponde, portanto, a uma referência a objetos que implementam tais algoritmos, devendo ser fornecidos pelo desenvolvedor do agente. Para que o mecanismo possa interagir com esses objetos, é necessário que eles apresentem interfaces conhecidas dos mecanismos concretos que os utilizam. As operações que compõem a interface de `Component` são descritas em 7.4.3.10.

A base de conhecimento do agente é transferida ao mecanismo através da operação `setKB(kb)`, onde `kb` é uma referência a um objeto que implementa a BC. Embora não esteja explícito na descrição do caso de uso *Processar Percepção*, para que um mecanismo possa invocar as operações de retorno oferecidas pelo agente, é necessário que ele possua uma referência para o objeto agente que o está utilizando. Esse é o propósito da operação `setClient(agent)`, fornecer ao mecanismo uma referência para o seu cliente.

As operações de sistema identificadas a partir de *Processar Percepção* estão mostradas na Figura 54.

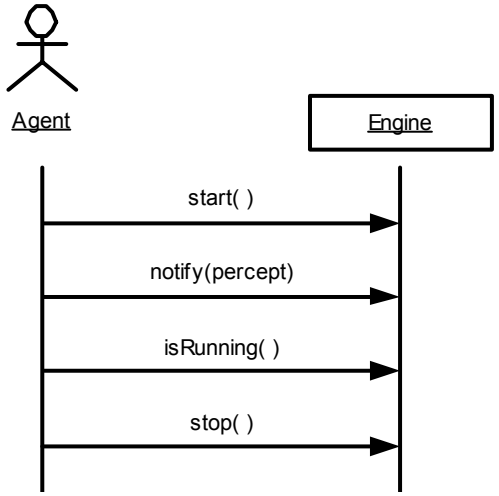


Figura 54: Operações para o controle da execução dos mecanismos

A execução do mecanismo é iniciada através da operação `start()`. Ao ser colocado em execução, o mecanismo realiza o processamento de todas as percepções recebidas através de cada nova invocação de `notify(percept)`. O parâmetro de entrada dessa operação, `percept`, contém um conjunto de sentenças que descreve a nova percepção, um nome (`name`), uma referência para o objeto que a gerou (`source`) e a data em que ela foi gerada (`da-`

`te`). Os atributos `source`, `name`, e `date` podem ser utilizados tanto pelo agente quanto pelo mecanismo, para filtrar e/ou priorizar as percepções. O processamento de percepções é encerrado através da operação `stop()`. Por sua vez, `isRunning()` é utilizada pelo agente para verificar se um mecanismo está em execução.

Ao iniciar sua execução em uma agência, o agente deve iniciar o mecanismo através de `start()` e, antes de migrar para uma outra agência, deve encerrá-lo usando `stop()`. Para possibilitar a execução de procedimentos imediatamente anteriores e posteriores a uma migração, foram incluídas em `Engine` duas operações específicas para essa finalidade: `beforeMove()` e `afterMove()`. Essas operações devem ser explicitamente invocadas pelo agente após `stop()` e antes de `start()`, respectivamente. Por exemplo, `beforeMove()` e `afterMove()` podem ser especializadas por um mecanismo concreto para introduzir um esquema de serialização de dados que utilize compressão ou para eliminar informações transientes. A Figura 55 ilustra essas operações.

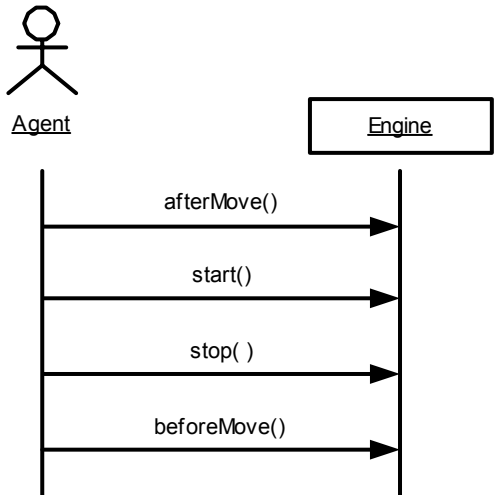


Figura 55: Operações para o controle da mobilidade dos mecanismos

7.4.2.2 Operações oferecidas pelo agente

Durante o processamento de uma percepção, um mecanismo pode solicitar informações adicionais ao agente. Ele também pode, como conseqüência de seu processamento, sugerir ao agente a execução de alguma ação. Ou seja, pode ser necessário que o agente realize

algum serviço para o mecanismo, como conseqüência de `notify()`. Esses serviços são executados pelas operações de retorno mostradas na Figura 56.

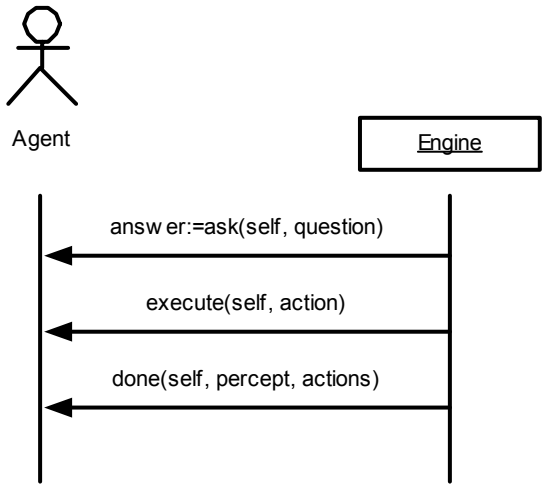


Figura 56: Operações que o agente oferece ao mecanismo abstrato

As semânticas das operações mostradas na Figura 56 são descritas a seguir. A operação `ask()` é utilizada pelo mecanismo para solicitar informações adicionais ao agente. Seu parâmetro de entrada, `question`, corresponde à sentença que representa essa indagação; seu parâmetro de retorno, `answer`, corresponde a uma lista de sentenças que representam as respostas do agente. Se o agente não possuir a informação solicitada, `answer` será uma lista vazia. A operação `execute()` é utilizada pelo mecanismo para sugerir ações ao agente, durante o processamento de uma percepção. Seu parâmetro de entrada, `action`, representa a ação sugerida. A operação `done()` é necessária para informar ao agente que uma certa percepção (`percept`) já foi completamente processada; o parâmetro de entrada `actions` corresponde à lista de ações que foram sugeridas ao agente durante esse processamento. É necessário identificar, através do parâmetro `percept`, a percepção que foi processada, uma vez que o agente pode ter enviado várias percepções ao mecanismo antes mesmo da primeira ter sido completamente avaliada.

É importante que o mecanismo se identifique, através do parâmetro de entrada `self`, ao invocar cada uma das operações descritas acima, uma vez que o agente pode ter enviado a mesma percepção a outros mecanismos.

Para honrar as invocações de `ask()`, `execute()` e `done()` o agente poderá recorrer a outros mecanismo de IA ou a algum SI presente na agência hospedeira, conforme descrito em 7.5.

A Figura 57 resume as operações de sistema identificadas, tanto para o mecanismo abstrato quanto para o agente. Vale notar que, nessa figura, nem `Engine` nem `Agent` correspondem a especificações de entidades de software. Eles são simplesmente tipos artificiais criados para ilustrar e documentar o conjunto de operações identificado.

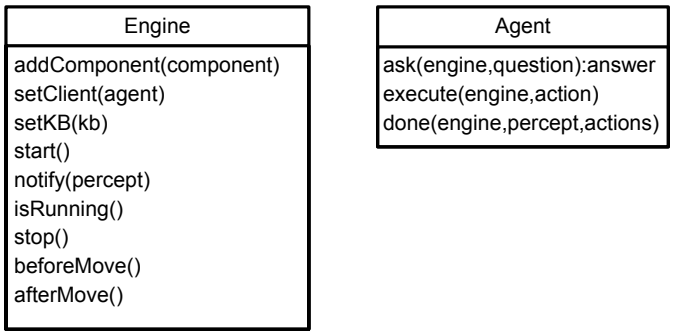


Figura 57: Operações de sistema identificadas

7.4.3 Modelo de Comportamento dos Objetos

O comportamento associado a uma operação de sistema é realizado através da interação entre um conjunto de objetos. Os objetos que tomam parte desse trabalho colaborativo são identificados, inicialmente, com base no modelo de conceitos. A interação inicia-se quando o objeto responsável pelo tratamento de uma operação recebe uma mensagem solicitando que a operação em questão seja executada. Para cada operação de sistema é necessário desenvolver-se um *diagrama de interação*, mostrando como os objetos envolvidos interagem para realizá-la. Durante esse processo, novas operações poderão ser identificadas.

A responsabilidade de lidar com as operações identificadas em 7.4.2 foram atribuídas, no caso do mecanismo, a uma classe abstrata denominada `Engine` e, no caso do agente, a uma interface chamada `EClient` (Figura 58). Todo mecanismo que venha a ser incluído ao *framework* deverá herdar os comportamentos definidos em `Engine`. Da mesma forma, todo agente que desejar utilizar esses mecanismos deverá implementar métodos para as operações definidas em `EClient`.

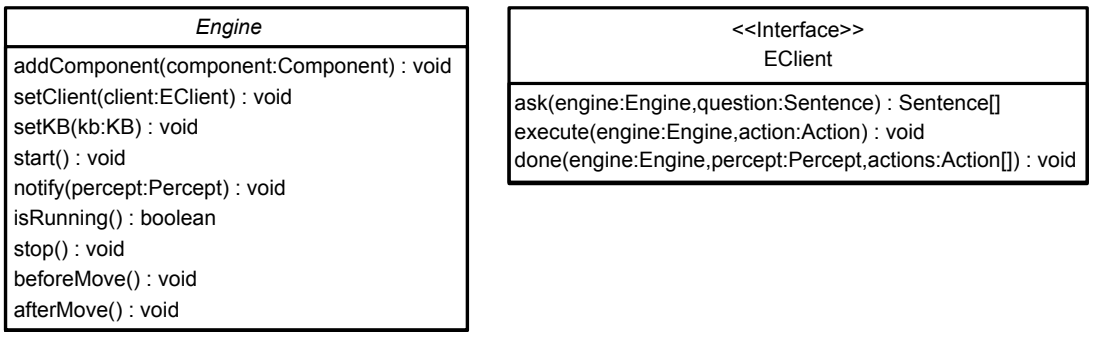


Figura 58: Operações atribuídas a `Engine` e `EClient`

Para cada operação não abstrata de `Engine`, é necessário desenvolver-se um *diagrama de interação*, mostrando como os objetos interagem para realizá-la. No caso de `EClient`, por tratar-se de uma interface, as implementações de suas operações serão específicas de cada agente.

As subseções seguintes mostram os diagramas de interação para as operações concretas de `Engine`.

7.4.3.1 Diagrama de interação para `Engine::setClient()`

O diagrama de interação para `Engine::setClient(…)` é bastante simples, como mostra a Figura 59. Ao receber a invocação `setClient(agent)` o mecanismo abstrato simplesmente armazena a referência para seu cliente em uma variável de instância (`client`), não enviando nenhuma mensagem a outros objetos. Para que o agente possa ser passado como argumento de entrada para `setClient(…)` ele precisa implementar a interface `EClient`.

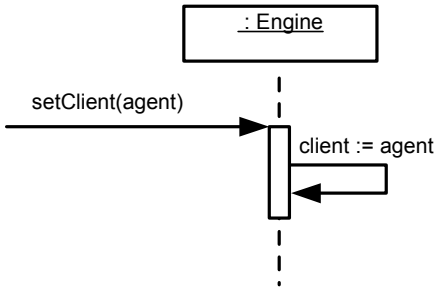


Figura 59: Diagrama de interação para `Engine::setClient()`

7.4.3.2 Diagrama de interação para `Engine::setKB()`

O diagrama de interação para `Engine::setKB(…)` é tão simples quanto o diagrama de `Engine::setClient(…)`., e por isso não será mostrado. Ao receber a invocação `setKB(kb)` o mecanismo abstrato simplesmente armazena uma referência para a base de conhecimento a ser utilizada em uma variável de instância (`kb`), não enviando nenhuma mensagem a outros objetos. O objeto passado como argumento de `setKB(…)`, isto é, o objeto que representa a base de conhecimento que se quer utilizar, deve implementar a interface `KB`, cujas operações são descritas em 7.4.4.

7.4.3.3 Diagrama de interação para `Engine::addComponent()`

Como mostra a Figura 60, ao receber a invocação `addComponent(c)`, o mecanismo abstrato simplesmente transfere uma referência para si mesmo ao novo componente, utilizando a operação `setEngine(engine)`, e, em seguida, adiciona-o a sua coleção de componentes.

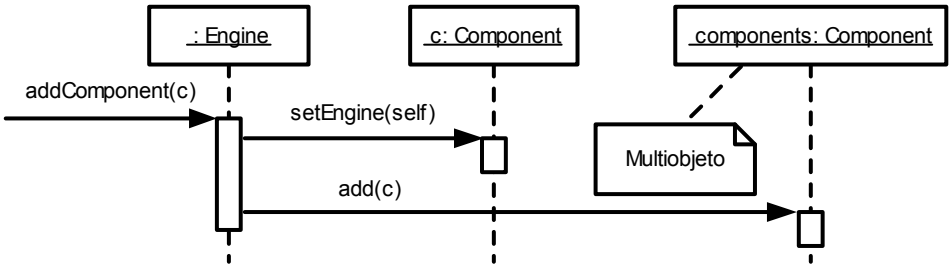


Figura 60: Diagrama de interação para `Engine::addComponent()`

7.4.3.4 Diagrama de interação para `Engine::beforeMove()`

Como mostra a Figura 61, ao receber a invocação `beforeMove()` o mecanismo abstrato primeiro ajusta o tamanho da coleção de componentes, para reduzir a quantidade de dados a serem transportados, e, então, repassa a invocação a `beforeMove()` a todos eles.

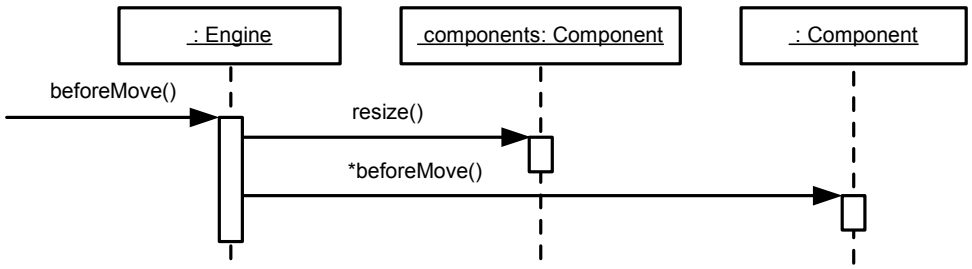


Figura 61: Diagrama de interação para `Engine::beforeMove()`

7.4.3.5 Diagrama de interação para `Engine::afterMove()`

Ao receber a invocação `afterMove()` o mecanismo abstrato repassa essa invocação a todos os seus componentes, como mostrado na Figura 62, e envia a cada um deles a sua própria referência, utilizando-se da operação `setEngine(engine)`. Isso pode parecer inócuo, já que o mesmo também foi realizado em `addComponent()`, no entanto, optou-se por fazê-lo novamente aqui para que a referência ao mecanismo, armazenada em cada componente, possa ser feita *transiente*, reduzindo-se, assim, a quantidade de dados a ser transportada em uma migração.

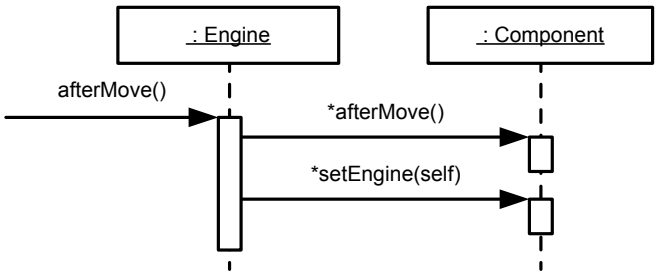


Figura 62: Diagrama de interação para `Engine::afterMove()`

7.4.3.6 Diagrama de interação para `Engine::start()`

Como mostrado na Figura 63, ao receber a invocação `start()`, o mecanismo abstrato cria seu próprio fluxo de controle de execução (*thread of control*), mas não o inicia imediatamente. Antes, a variável de instância booleana `running` recebe o valor `TRUE` e uma percepção inicial, denominada `StartingEngine`, é gerada e adicionada, por meio de `notify()`, à fila de percepções a serem tratadas. Essa percepção inicial será a primeira a ser processada

pelo mecanismo concreto, podendo ser usada para disparar procedimentos específicos de inicialização. O digrama de interação para `notify()` é descrito em 7.4.3.7.

Após a notificação de `StartingEngine`, o fluxo de controle de execução de `Engine` é iniciado. O fato de que cada mecanismo possui seu próprio fluxo de controle de execução permite que o agente possua diversos mecanismos em execução simultânea.

Ainda com base na Figura 63, percebe-se que uma nova operação, denominada `run()`, foi acrescentada ao mecanismo abstrato. Essa operação não deve ser visível aos clientes do mecanismo, devendo, portanto, ser uma operação privativa (`private`). O digrama de interação para `run()` é descrito em 7.4.3.10.

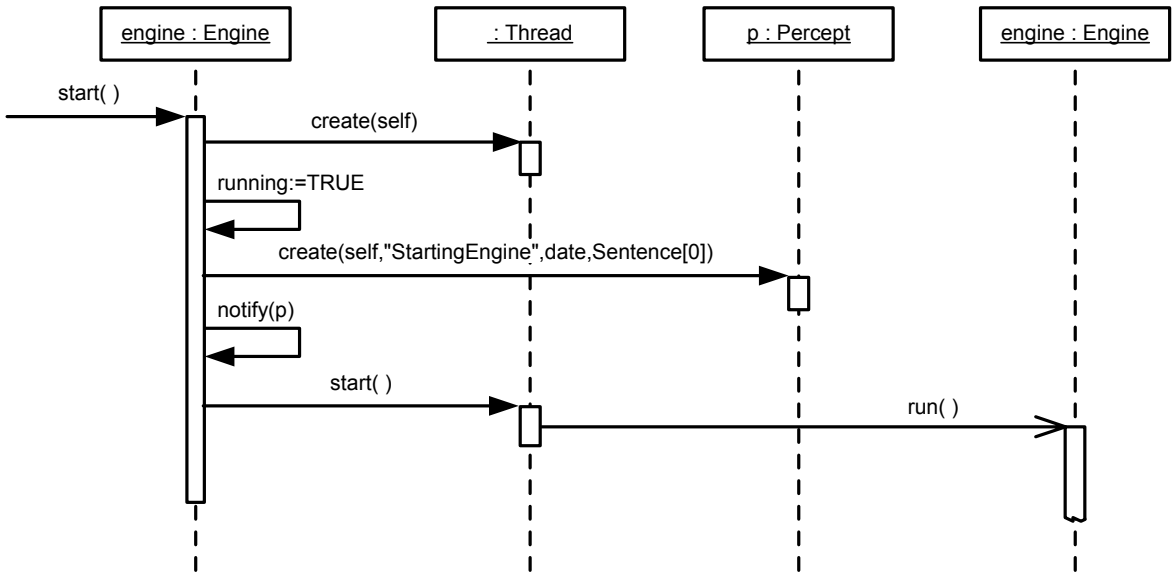


Figura 63: Diagrama de interação para `Engine::start()`

7.4.3.7 Diagrama de interação para `Engine::notify()`

Ao receber a invocação de `notify(percept)`, o mecanismo abstrato simplesmente armazena o objeto `percept` numa fila do tipo FIFO (*first-in-first-out*) para posterior processamento (Figura 64). Uma percepção somente é adicionada à fila se o mecanismo já estiver em execução. A fila de percepções é implementada por um objeto da classe `PerceptQueue`, criado somente na primeira invocação de `notify()`. O adiamento da criação do objeto que implementa a fila de percepções é um exemplo de aplicação do princípio de projeto *Instanciação Tardia* (*Lazy Instantiation*), descrito por Grand (1998).

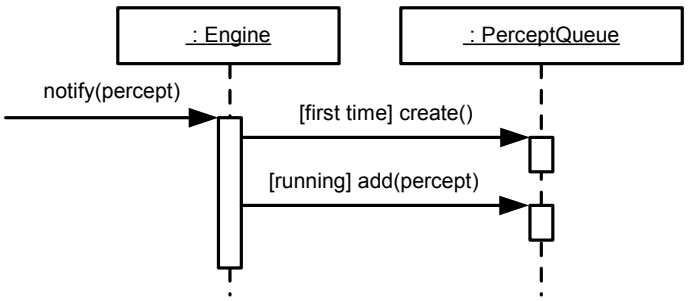


Figura 64: Diagrama de interação para `Engine::notify()`

7.4.3.8 Diagrama de interação para `Engine::isRunning()`

Não é necessário um diagrama de interação para `isRunning()`, posto que essa operação simplesmente retorna o valor da variável de instância `running`.

7.4.3.9 Diagrama de interação para `Engine::stop()`

Ao receber a invocação de `stop()`, o mecanismo abstrato cria uma percepção final, denominada `StoppingEngine` e a adiciona, por meio de `notify()`, à fila de percepções a serem tratadas. Essa percepção final será a última a ser processada pelo mecanismo concreto, podendo ser usada para disparar procedimentos específicos de finalização. Finalmente, a variável de instância booleana `running` recebe o valor `FALSE` para indicar que a execução foi encerrada. A operação `run()` utiliza essa informação para interromper o fluxo de execução de `Engine`. A Figura 65 mostra o diagrama de interação para `Engine::stop()`.

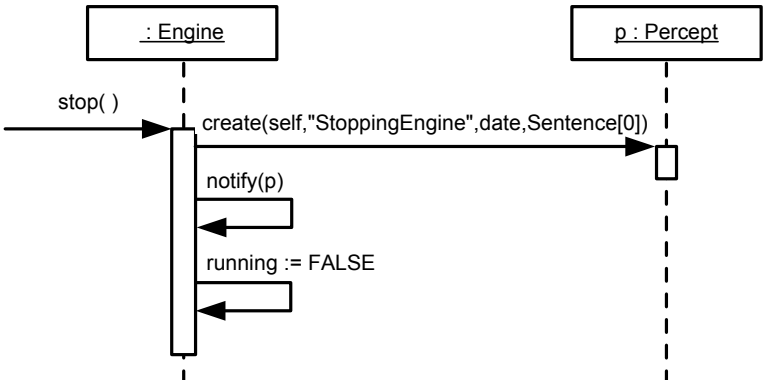


Figura 65: Diagrama de interação para `Engine::stop()`

7.4.3.10 Diagrama de interação para `Engine::run()`

Como descrito em 7.4.3.2, para que `Engine` possua seu próprio fluxo de controle de execução ele deve implementar ou transferir a responsabilidade pela implementação da operação `run()` para suas subclasses, isto é, para os mecanismos concretos. No caso, adota-se uma solução intermediária, utilizando-se o padrão de projeto *Template Method* (GAMMA et al., 1995, p.325). Assim, em `run()` define-se o "esqueleto" genérico do processo de execução e postergam-se alguns passos mais específicos desse processamento para os mecanismos concretos. A Figura 66 mostra esse "esqueleto".

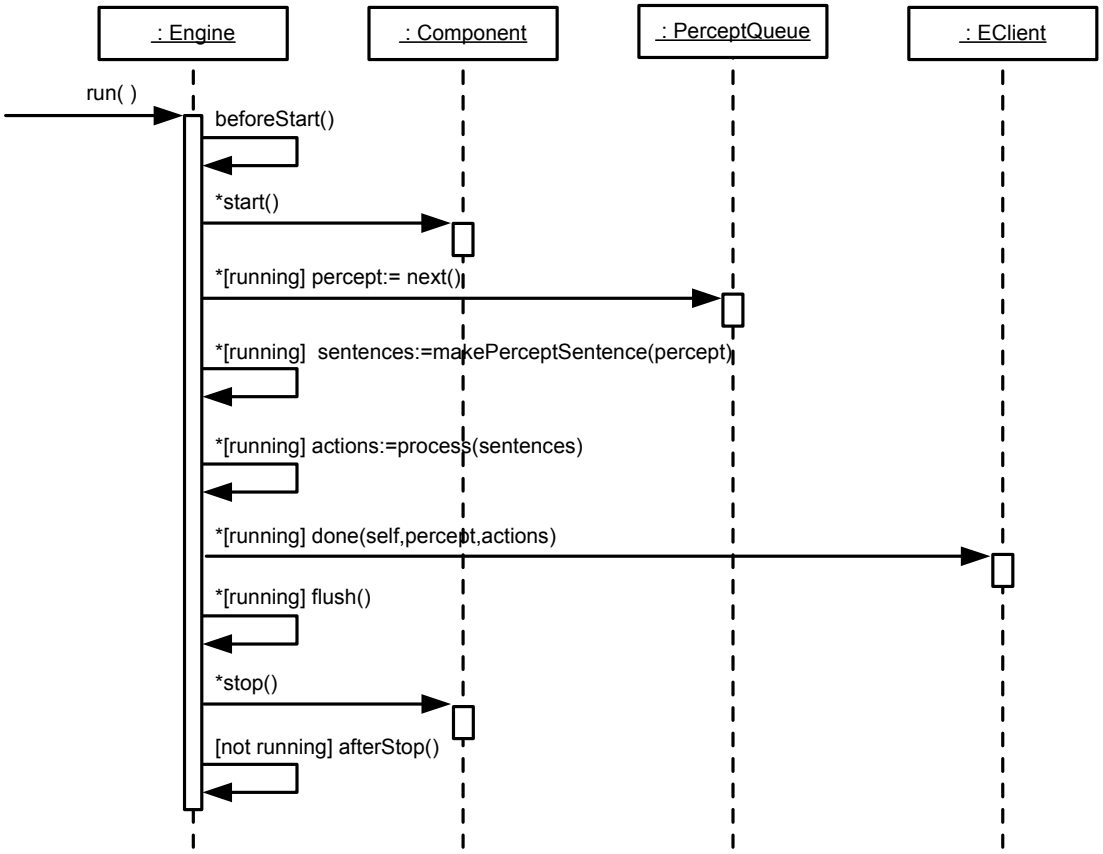


Figura 66: Diagrama de interação para `Engine::run()`

Para permitir que cada mecanismo concreto possa executar procedimentos específicos imediatamente antes do início de seu fluxo de execução e imediatamente após o encerramento do mesmo, foram adicionadas a `Engine` duas operações específicas para esse fim: `beforeStart()` e `afterStop()`. Essas operações são invocadas *automaticamente* pelo

mecanismo abstrato quando da chamada de `run()`. Por exemplo, a operação `beforeStart()` pode ser especializada por um mecanismo concreto para verificar se todos os componentes necessários à sua execução já foram especificados; enquanto `afterStop()`, para liberar algum recurso vinculado ao mecanismo.

Após a chamada de `beforeStart()`, o mecanismo inicia a execução de cada um de seus componentes através de `start()`. Entre as chamadas de `start()` e de `stop()`, enquanto `running=TRUE`, o mecanismo permanecerá num laço executando os seguintes passos: `next()` retira da fila de percepções a próxima percepção a ser processada; `makePerceptSentence()` traduz a percepção em um conjunto de sentenças expressas em uma linguagem de representação que o mecanismo concreto sabe como processar; `process()` realiza o processamento da percepção, produzindo um conjunto de ações como resposta; `Eclient:done()` informa o agente sobre o resultado do processamento e, finalmente, através de `flush()` o mecanismo concreto pode, por exemplo, liberar recursos alocados durante o episódio de processamento da percepção em questão.

Em síntese, o "esqueleto" estabelecido por `run()` impõe a necessidade das seguintes operações adicionais: `beforeStart()`, `afterStop()`, `flush()`, `process()` e `makePerceptSentence()`. Para que essas operações não sejam visíveis aos clientes, elas são definidas como operações protegidas (`protected`). As primeiras três operações são operações concretas que, assim como `afterMove()`, simplesmente repassam as mesmas chamadas a todos os componentes do mecanismo. Caso seja necessário, um mecanismo concreto poderá especializá-las. As duas últimas operações são definidas como operações abstratas. Portanto, cada mecanismo concreto deverá implementá-las de acordo com suas necessidades.

Finalmente, também foi incluída em `Engine` a operação `getClient()`, que retorna uma referência do tipo `EClient` para o cliente do mecanismo. Esta operação é necessária caso os componentes do mecanismo precisem invocar diretamente as operações oferecidas pelo cliente.

7.4.4 Modelo de Classes

Com base nos diagramas de interação descritos em 7.4.3, foram identificadas as operações para a classe abstrata `Engine` e para a interface `Component`, mostradas na Figura 67. Também se pôde construir o diagrama de classes do *Subsistema Mecanismo Abstrato*, ilustrado na Figura 68.

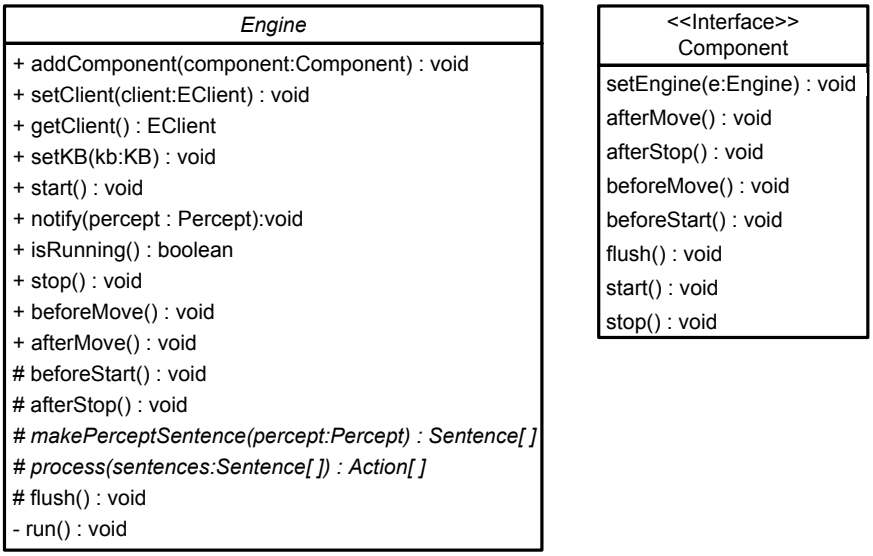


Figura 67: Operações de `Engine` e `Component`

A partir do diagrama da Figura 68, pode-se observar que foram criadas interfaces para os elementos de software que serão trocados entre os diversos subsistemas, especificamente: `KB`, `Percept`, `Sentence` e `Action`. O uso de interfaces permite o desenvolvimento em separado dos diversos subsistemas, promovendo um fraco acoplamento entre eles.

A interface `KB` define um iterador sobre uma base de conhecimento genérica, especificando operações que permitem acessar seqüencialmente as sentenças presentes nessa base (`next()`), controlar a iteração sobre o seu conteúdo (`hasNext()`) e determinar o número total de sentenças que ela contém (`size()`). Uma base de conhecimento concreta deverá realizar essa interface para que possa ser utilizada por `Engine`. Trata-se, portanto, de uma aplicação do padrão de projeto *Iterator* (GAMMA et al., 1995, p.257).

A interface `Percept` define as operações que permitem identificar e processar uma percepção. As percepções geradas pelo SI e pelo próprio agente deverão realizar essa interfa-

ce. Ela também desacopla os subsistemas *Mecanismo Abstrato* e *Interface do Serviço de Inteligência.*

A interface `Sentence` tem por objetivo desacoplar os subsistemas *Mecanismo Abstrato*, *Interface de Representação de Conhecimento* e *Interface do Serviço de Inteligência.* Ela define uma única operação, `language()`, que permite recuperar uma descrição do tipo de representação utilizada. As classes que representarem sentenças em qualquer tipo de linguagem de representação de conhecimento deverão realizar essa interface.

A interface `Action` tem por objetivo desacoplar os subsistemas *Mecanismo Abstrato* e *Interface do Serviço de Inteligência.* Ela define duas operações: `type()`, que retorna o tipo de ação que o objeto representa, e `sentence()`, que retorna uma descrição da ação em alguma linguagem de representação de conhecimento.

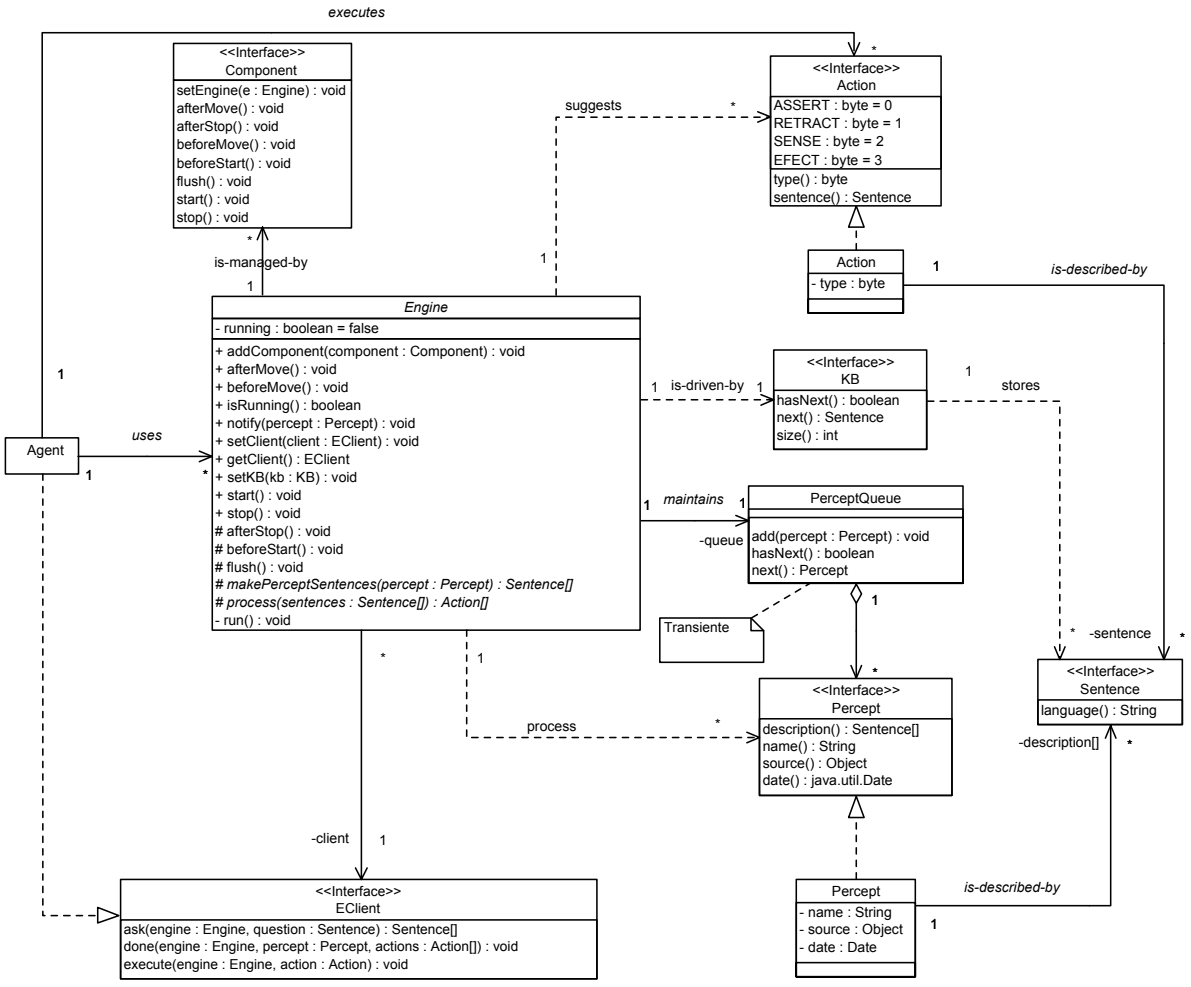


Figura 68: Diagrama de classes para o *Subsistema Mecanismo Abstrato*

Além das operações identificadas para `PerceptQueue`, a partir dos diagramas de seqüência mostrados nos itens 7.4.3.8 e 7.4.3.10, também foi incluída nessa classe a operação `hasNext()`, para auxiliar no controle da iteração sobre os elementos da fila de percepções. A referência de `Engine` para `PerceptQueue` foi marcada como `transient`, de modo que quando o mecanismo for serializado, antes da migração do agente, o objeto `queue` dessa classe não será serializado e, por conseguinte, não será transportado. Isso, além de aliviar o *estado de dados* transportado, também garante que percepções percebidas em uma agência não serão processadas em outra.

Dos elementos mostrados no diagrama da Figura 68, somente a classe abstrata `Engine` e as interfaces `Percept`, `Action`, `Sentence` e `EClient` são visívies externamente ao subsistema.

7.5 Subsistema Interfaces do Serviço de Inteligência

Como descrito no Capítulo 6, para honrar os pedidos de informações adicionais ou de execução de ações, o agente pode depender de um serviço de inteligência oferecido pela agência. Internamente, uma agência pode possuir diversas implementações do serviço de inteligência, por exemplo, uma para cada domínio de ontologia. No entanto, do ponto de vista externo, todos esses serviços devem ser acessados através de uma única interface, a qual foi atribuído o nome de *Interface de Servidor do Serviço de Inteligência*. As operações presentes nessa interface estão sintetizadas na Figura 69.

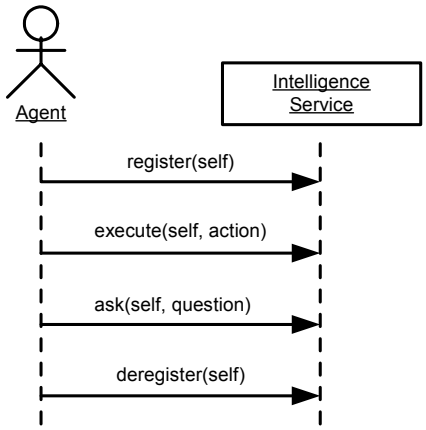


Figura 69: Operações presentes na *Interface de Servidor do Serviço de Inteligência*

As semânticas das operações `execute()` e `ask()` são idênticas às semânticas das operações de mesmo nome oferecidas por `EClient`. A operação `register()` é utilizada pelo agente para cadastrar-se como consumidor das percepções geradas pelo *Serviço de Inteligência*. A operação `deregister()` retira o agente da lista de consumidores de percepções.

Toda vez que ocorrer um evento em seu domínio, o *Serviço de Inteligência* deve informar seus clientes sobre essa ocorrência através da operação de retorno `notify(percept)`, oferecida por uma *Interface de Cliente do Serviço de Inteligência*, conforme mostrado na Figura 70.

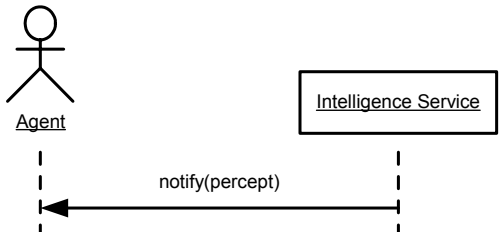


Figura 70: Operações presentes na *Interface de Cliente do Serviço de Inteligência*

7.5.1 Modelo de Classes

A Figura 71 mostra o modelo de classes para o *Serviço de Inteligência*, derivado dos diagramas de interação apresentados nas duas figuras anteriores.

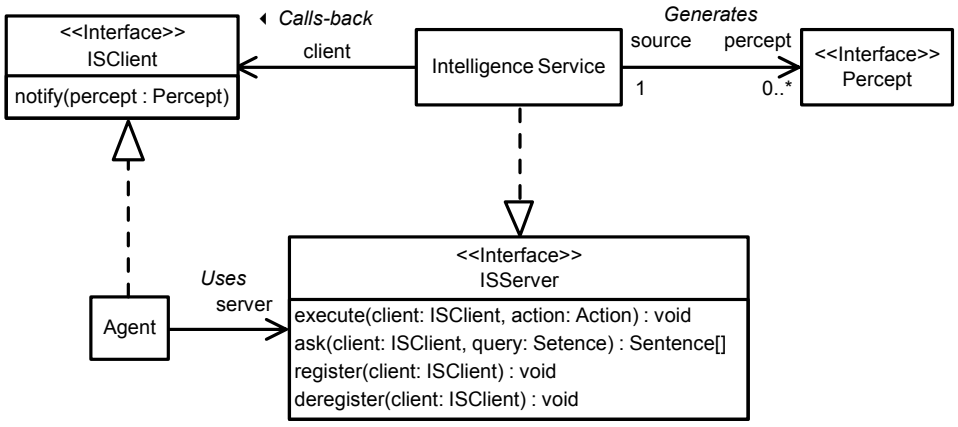


Figura 71: Diagrama de classes para o *Subsistema Serviço de Inteligência*

O agente deve implementar a interface `ISClient`, que contém a única operação de retorno que o *Serviço de Inteligência* espera de seus clientes. O argumento de entrada dessa operação deve ser um objeto de uma classe que implemente a interface `Percept`. Por sua vez, todo *Serviço de Inteligência* concreto deve implementar a interface `ISServer`, que reúne todas as operações que um agente espera dele.

7.6 Considerações Finais

Neste capítulo apresentou-se o desenvolvimento orientado a objetos dos subsistemas centrais do *Framework de Inteligência* especificado no Capítulo 6, isto é, os *Subsistemas Mecanismo Abstrato* e *Interfaces do Serviço de Inteligência*. As classes e interfaces que compõem o subsistema *Mecanismo Abstrato* são utilizadas pelo agente para configurar e controlar a execução dos mecanismos concretos. As classes e interfaces do subsistema *Interfaces do Serviço de Inteligência* são utilizadas na interação entre o agente e o serviço de inteligência concreto oferecido pela agência.

Em síntese, as ações tomadas neste capítulo para reduzir a quantidade de código e dados transportada pelo AMI foram as seguintes: (1) definição das interfaces do *Serviço de Inteligência*, que permitem a transferências de funcionalidades de sensoriamento e atuação para as agências e (2) a utilização do padrão de projeto *Method Template* para evitar a duplicação desnecessária de código e dados na implementação dos mecanismos concretos, com a conseqüente introdução da classe abstrata `Engine`, onde se concentrou a parcela invariante do comportamento desses mecanismos.

Capítulo 8

Desenvolvimento do Mecanismo Concreto de Inferência

8.1 Introdução

Neste capítulo são descritos os modelos para os subsistemas específicos ao mecanismo concreto avaliado, isto é, ao *Mecanismo de Inferência com Encadeamento Progressivo Baseado em Regras*. Na Seção 8.2 discutem-se os modelos para o subsistema *Interfaces de Representação de Conhecimento*. Na Seção 8.3 são apresentados os modelos para o *Mecanismo de Inferência* em si. Na Seção 8.4 são construídos os modelos para o *Rete*, o algoritmo de casamento utilizado. Finalmente, na Seção 8.5, tecem-se alguns comentários sobre a estratégia de resolução de conflitos utilizada como *default* pelo mecanismo de inferência desenvolvido.

8.2 Subsistema Interfaces de Representação de Conhecimento

O desenvolvimento do *Mecanismo de Inferência* começou pela definição das classes e interfaces que compõem o subsistema *Interfaces de Representação de Conhecimento*. A principal razão para isso é o fato de que esse subsistema é um subsistema *não funcional*, pois seus componentes não oferecem, de fato, serviços a um cliente, mas sim representam "peças" a partir das quais uma base de conhecimento pode ser construída. Portanto, trata-se de um

subsistema básico, que não depende dos demais subsistemas do mecanismo concreto, os quais, por sua vez, dependem direta ou indiretamente dos componentes oferecidos por ele. As operações apresentadas pelas interfaces de cada um de seus elementos correspondem unicamente a *operações de acesso*, que permitem recuperar os atributos que uma implementação concreta dessas interfaces deve possuir. Em outras palavras, essas operações permitem recuperar as "partes" que compõem cada um dos elementos utilizados na representação de conhecimento. Essas "partes" são criadas por *fábricas* (GAMMA et al., 1995, p.87), a partir da forma de representação que for mais conveniente para o usuário, por exemplo, sentenças em ASCII puro ou em BRML (*Business Rules Markup Language*) (GROSOF; LABROU, 1999).

Por se trata de um subsistema não-funcional, somente os modelos de conceitos e de classes fazem sentido para esse subsistema.

8.2.1 Modelo de Conceitos

O modelo de conceitos foi criado diretamente da descrição dos elementos sintáticos da lógica de primeira ordem apresentados em 4.3. A Figura 72 apresenta o único diagrama de conceitos que compõe esse modelo.

8.2.2 Modelo de Classes

O modelo de classes foi obtido a partir do modelo de conceitos. Algumas simplificações foram efetuadas com o objetivo de minimizar o número de interfaces que fazem parte do subsistema. Deve-se ter em mente que o modelo de conceitos tem por função o perfeito entendimento dos elementos do domínio do problema e que, por outro lado, o modelo de classes especifica as entidades de software que deverão ser implementadas. Portanto, a escolha dos elementos e dos relacionamentos que compõem o modelo de classes deve ser realizada de forma "austera". A Figura 73 apresenta essas interfaces, explicitando suas operações e seus relacionamentos.

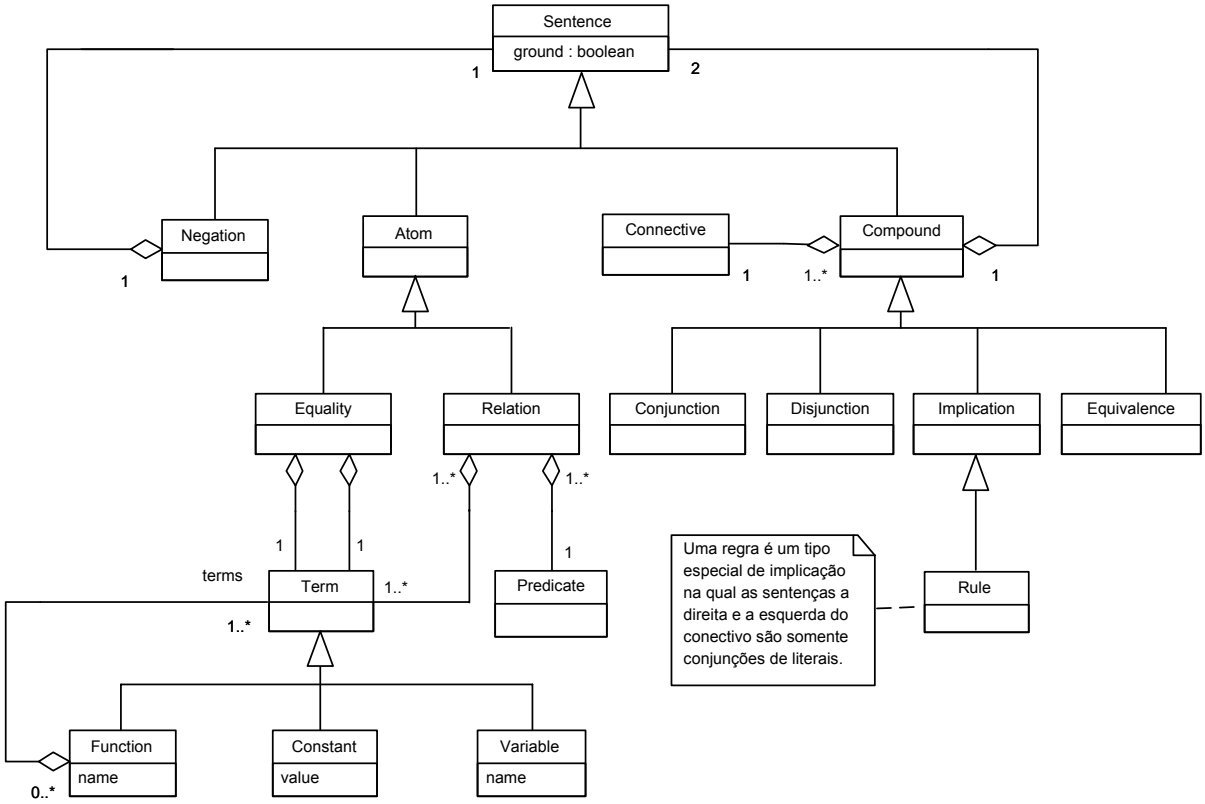


Figura 72: Diagrama de conceitos para o *Subsistema Interfaces de Representação de Conhecimento*

Observa-se, na Figura 73, que os conceitos `Negation`, `Compound`, `Connective`, `Conjunction`, `Disjunction`, `Equivalence`, `Implication`, `Atom`, `Equality`, `Relation` e `Function` não possuem correspondentes no modelo de classes. Por outro lado, foi acrescentada uma nova interface: `Literal`. A interface `Sentence` é a definida no *Subsistema Mecanismo Abstrato*.

Embora uma regra seja um tipo especial de `Compound` (`Implication`), decidiu-se, por tratar-se de foco primário do desenvolvimento, manter-se somente uma interface própria para ela (`Rule`). Desse modo, pôde-se eliminar os conceitos `Compound` e `Implication` e nomear as operações de `Rule` de uma forma mais expressiva para seus usuários. Através dessas operações, tem-se acesso direto às suas expressões antecedente e conseqüente, sem permitir que o seu conteúdo seja alterado, o que caracteriza uma aplicação do padrão de projeto *Immutable* (GRAND, 1998, p. 67). Com relação aos conectivos, assume-se que as expressões antecedente e conseqüente de `Rule` estão, implicitamente, ligadas pelo conectivo *implica* ($\rightarrow$) e que os diversos literais que as compõem estão relacionados pelo conectivo *E* ($\wedge$).

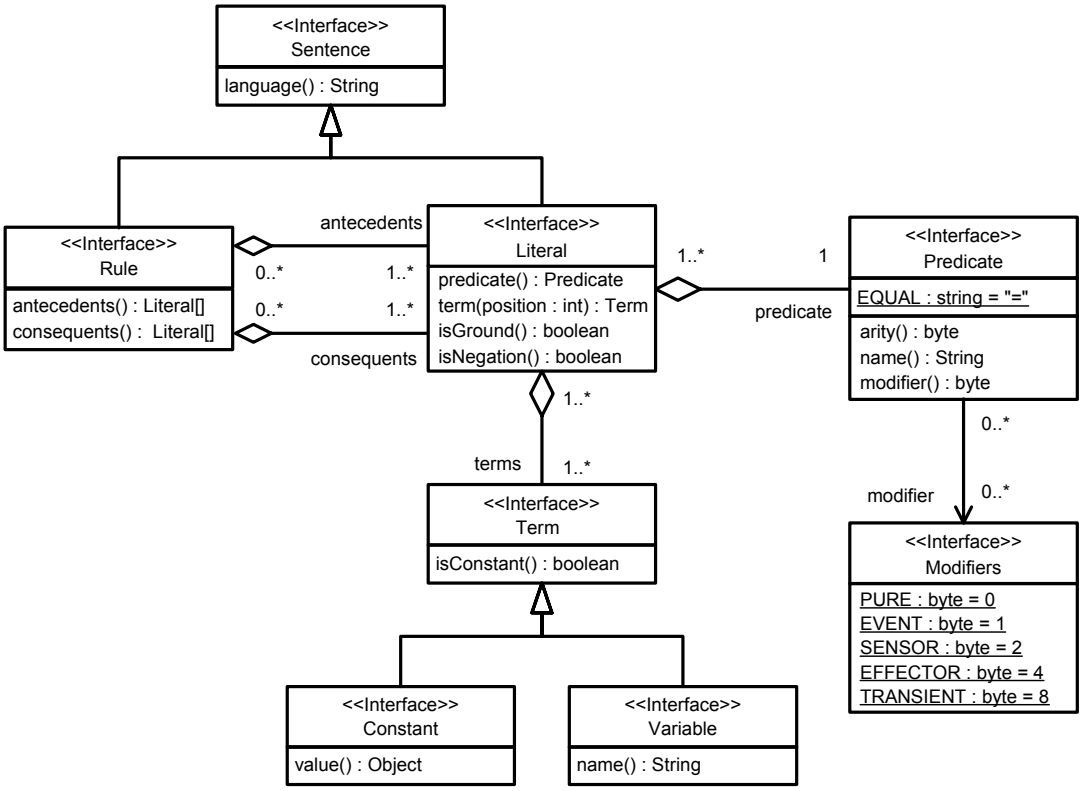


Figura 73: Classes e Interfaces de uso geral para representação de conhecimento

Com a eliminação de `Negation` e a generalização de `Sentence` proveniente do *Subsistema Mecanismo Abstrato*, os atributos `negation` e `ground` foram deixados a cargo da implementação da interface `Literal`, que desempenha um papel mais amplo que o conceito `Atom`, também eliminado. O papel que o conceito `Atom` desempenharia no software é exercido por um `Literal` positivo. A igualdade representada por `Equality` foi substituída por um *predicado reservado* denominado `EQUAL`, definido na interface `Predicate`. Os conceitos componentes de `Relation`, isto é, `Term` e `Predicate`, foram transferidos para `Literal`. A interface `Predicate` , além de permitir o acesso ao seu nome, através da operação `name()`, também permite recuperar, por meio de `arity()`, o número de termos a que o predicado está associado, e possíveis modificadores de seu significado, através de `modifiers()`. Para reduzir a quantidade de dados a serem transportados para cada predicado, o número de termos e o número que representa o modificador foram escolhidos como sendo do tipo *byte*. A definição dos possíveis modificadores foi estabelecida em uma interface denominada `Modifiers`, acrescentada ao *Subsistema Mecanismo Abstrato* por tratar-se de informação útil para qualquer tipo de mecanismo. Os modificadores possíveis são `EVENT` (`1`),

SENSOR (2), EFFECTOR (4) e TRANSIENT (8). Seus valores foram escolhidos de modo a permitirem a composição de modificadores.

O conceito Function foi eliminado devido a restrição **datalog** (GROSOF, 1999), já descrita em 4.5.6. O conceito Constant foi mantido; porém o parâmetro de retorno de sua única operação, value(), foi feito o mais genérico possível (Object), para que pudesse comportar os diversos tipos de constantes. Dessa forma, quando uma instância da classe Constant corresponder a um símbolo, o parâmetro de retorno deverá ser do tipo java.lang.String, o mesmo ocorrendo quando a instância representar um texto. Se a instância de Constant corresponder a um número inteiro, o parâmetro de retorno deverá ser do tipo java.lang.Integer e no caso de um número real, java.lang.Double. A classe do usuário que implementa a interface Constant deve impor essas restrições. O conceito Variable também foi mantido, como o parâmetro de retorno de sua única operação, name(), igual a java.lang.String.

8.3 Subsistema Mecanismo de Inferência

Nesta seção apresentam-se os diversos modelos que representam o resultado da especialização do *Mecanismo Abstrato* descrito em 7.4 em um *Mecanismo de Inferência com Encadeamento Progressivo Baseado em Regras*.

8.3.1 Modelo de Conceitos

Com base na descrição do *Mecanismo de Inferência com Encadeamento Progressivo* realizada em 4.5 identificou-se o modelo de conceitos mostrado na Figura 74.

Nesse modelo, o conjunto de conflito (Conflict Set) é composto por uma série de ativações (Activation), que representam os conseqüentes das regras cujos antecedentes foram todos satisfeitos pelos fatos presentes na memória de trabalho do Matcher (Working Memory), com suas varáveis já eliminadas. Strategy representa a estratégia de resolução de conflitos utilizada.

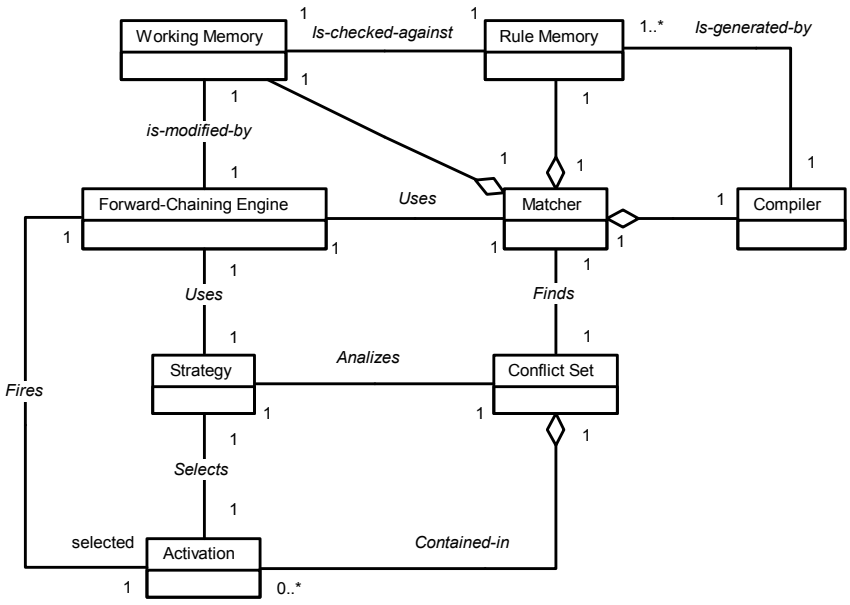


Figura 74: Modelo de conceitos para o mecanismo de inferência

8.3.2 Modelo de Comportamento dos Objetos

Nesta subseção são apresentados os diagramas de interação para as operações abstratas de `Engine` cujos comportamentos devem ser definidos especificamente para cada mecanismo concreto desenvolvido. Também são apresentados os diagramas de interação para operações concretas de `Engine` cujos comportamentos precisaram ser especializados para se adaptarem ao *Mecanismo de Inferência com Encadeamento Progressivo*.

8.3.2.1 Diagrama de interação para `FCEngine::setKB()`

A operação concreta `Engine::setKB()` foi especializada simplesmente para que, ao ser definida a base de conhecimento a ser utilizada, uma variável booleana denominada `compiled` fosse ajustada para `false`, indicando que essa base ainda não fora compilada para a forma interna de representação utilizada pelo casador de padrões do Mecanismo de Inferência. Após isso, a operação `setKB()` de `Engine` é chamada normalmente.

8.3.2.2 Diagrama de interação para `FCEngine::addComponent()`

De acordo com o exposto em 6.5, os objetos que implementam os algoritmos de casamento e de resolução de conflitos do Mecanismo de Inferência devem possuir interfaces bem definidas e conhecidas. Essas interfaces foram denominadas, respectivamente, `Matcher` e `Strategy`.

A operação concreta `Engine::addComponent()` foi especializada simplesmente para que, ao ser adicionado um novo componente, fosse realizado um teste para verificar se esse componente era do tipo `Matcher` ou `Strategy`. Variáveis de instâncias mais específicas do que `Component` também são definidas para esses componentes, de modo que, nos demais métodos de `FCEngine`, as referências a eles fosse mais natural, isto é, de acordo com os conceitos específicos de tal mecanismo. A Figura 75 mostra o diagrama de interação para `FCEngine::addComponent()`.

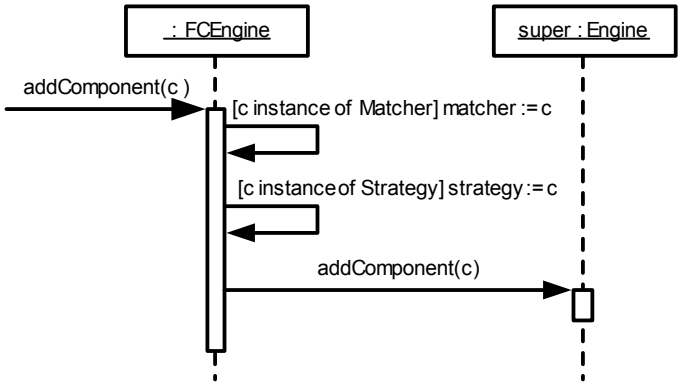


Figura 75: Diagrama de interação para `FCEngine::addComponent()`

As operações que fazem parte das interfaces `Matcher` e `Strategy` foram identificadas a partir dos diagramas de interação para as operações de `FCEngine` descritos a seguir.

8.3.2.3 Diagrama de interação para `FCEngine::beforeStart()`

A operação concreta `Engine::beforeStart()` foi especializada simplesmente para que, antes de iniciar a execução do Mecanismo de Inferência, se a base de conhecimento ainda não tivesse sido traduzida para a representação interna do *Casador de Padrões*

(`compiled=false`) utilizado, ela o fosse, através de `Matcher::assert(kb)`. A Figura 76 mostra o diagrama de interação para `FCEngine::beforeStart()`.

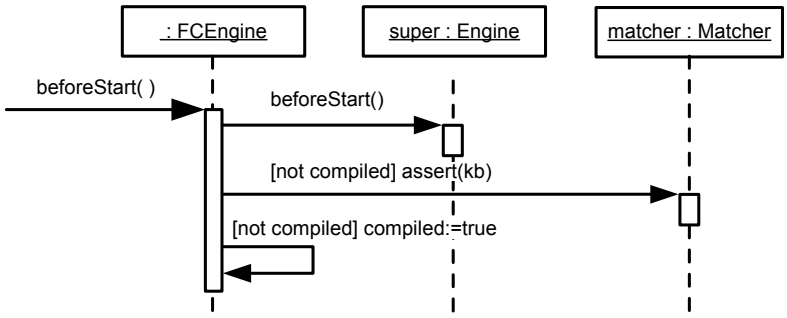


Figura 76: Diagrama de interação para `FCEngine::beforeStart()`

8.3.2.4 Diagrama de interação para `FCEngine::makePerceptSentence()`

A operação `FCEngine::makePerceptSentence()` é responsável por converter a descrição de uma percepção, expressa na forma de um conjunto de objetos que implementam a interface `Sentence`, em um conjunto de objetos do tipo `Literal`, introduzido no *Subsistema de Interfaces de Representação de Conhecimento*. A Figura 77 mostra o diagrama de interação para essa operação.

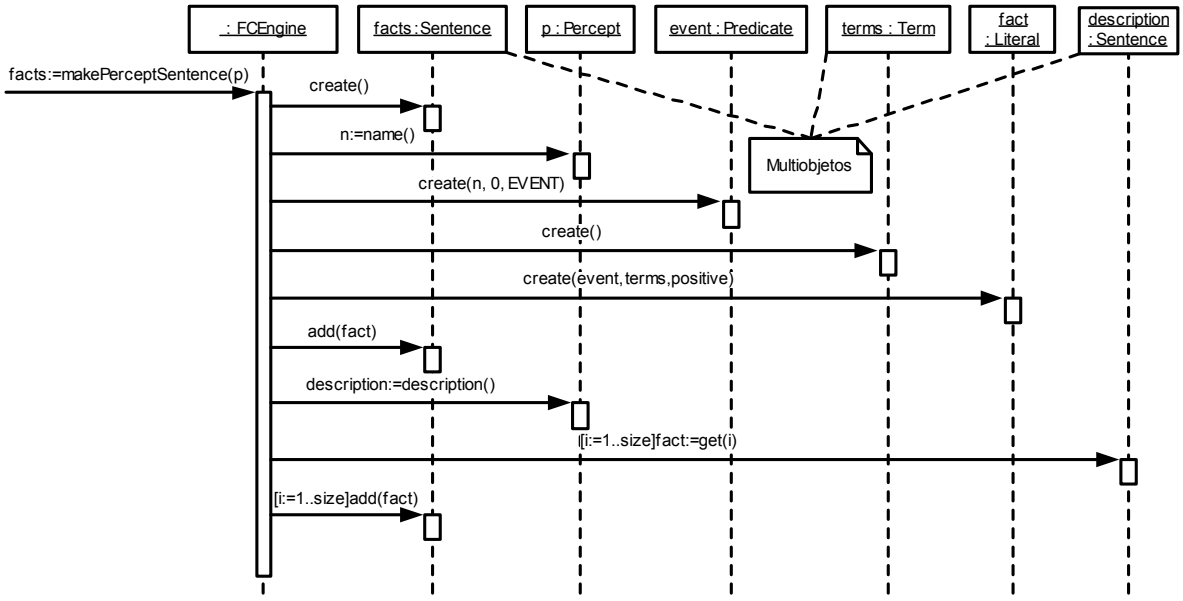


Figura 77: Diagrama de interação para `FCEngine::makePerceptSentence()`

Objetos do tipo `Literal` são tratáveis pelo Mecanismo de Inferência. Essa tradução é importante para que não sejam armazenados na base de conhecimento interna do mecanismo concreto objetos cujas implementações sejam desconhecidas e que possam, portanto, onerar a migração do agente com o transporte de dados desnecessários e/ou até mesmo impedi-la, por não serem serializáveis. Além disso, para cada percepção, essa operação também cria um literal positivo do tipo `Event` para representar a ocorrência da percepção em si. O predicado desse literal é feito idêntico ao nome da percepção e sua lista de termos é vazia.

8.3.2.5 Diagrama de interação para `FCEngine::process()`

A operação `FCEngine::process()` é responsável pelo processamento de um único episódio de inferência, realizando o controle da atividade de encadeamento progressivo. A Figura 78 mostra o diagrama de interação para essa operação.

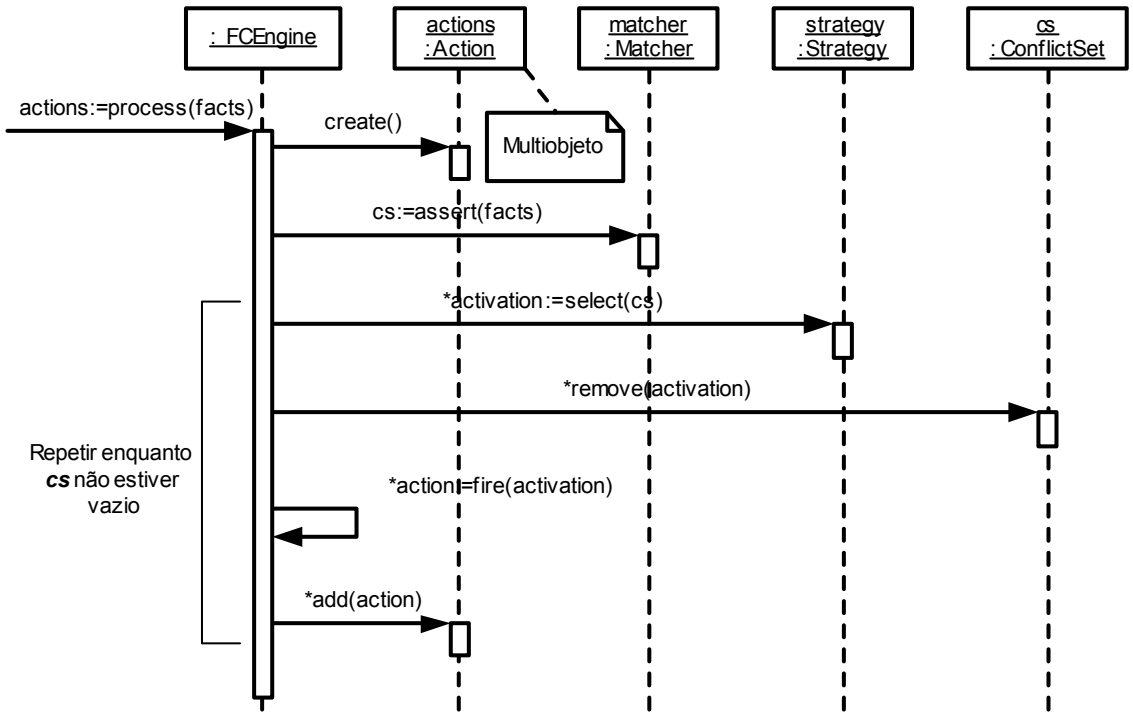


Figura 78: Diagrama de interação para `FCEngine::process()`

Inicialmente `FCEngine::process()` cria um objeto contêiner para armazenar as ações executadas durante o episódio corrente (`actions`) e, em seguida, informa o `Matcher`

sobre os novos fatos a serem processados (`Matcher::assert(facts)`), recebendo como retorno uma referência para o conjunto de conflitos (`cs`) gerado pelo algoritmo de casamento. Esse conjunto é composto por uma coleção de ativações (`Activation`), cada uma delas representando uma conclusão.

Então, enquanto o conjunto de conflitos não estiver vazio, ela realiza as seguintes tarefas: solicita que o algoritmo de resolução de conflitos selecione uma das ativações (`Strategy::select()`); remove a ativação escolhida do conjunto de conflitos (`ConflictSet::remove()`), disparando-a em seguida (`FCEngine::fire()`) e atualiza a coleção de ações já executadas (`actions`), retornada por `process()`. A operação `fire()` é privativa de `FCEngine`.

8.3.2.6 Diagrama de interação para `FCEngine::fire()`

Esta operação é responsável por executar a ação associada à ativação selecionada pelo algoritmo de resolução de conflitos. A mostra Figura 79 o diagrama de interação para a operação `FCEngine::fire()`.

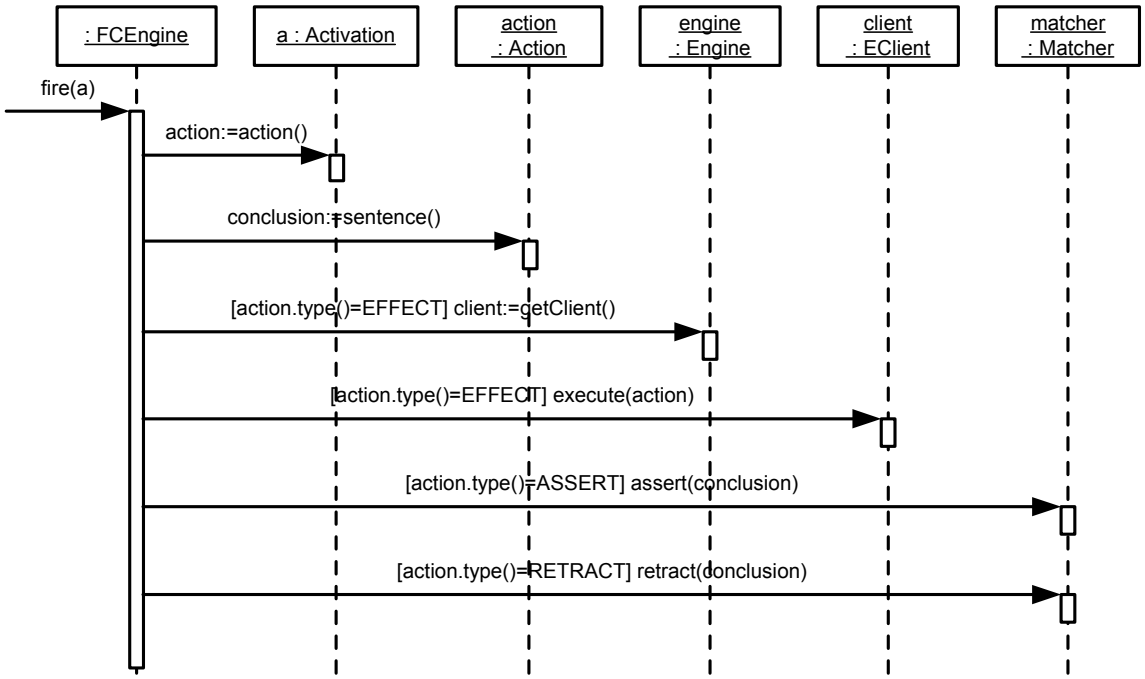


Figura 79: Diagrama de interação para `FCEngine::fire()`

Ao ser invocada, `FCEngine::fire()` obtém a ação correspondente à conclusão inferida, presente na ativação recebida como argumento de entrada (`Activation::action()`). A seguir, processa a ação, com base em seu tipo: se a ação for do tipo `EFFECT`, a sua execução é sugerida ao cliente do mecanismo através de `EClient.execute(action)`; se a ação for do tipo `ASSERT`, o literal associado a ela é simplesmente adicionado à memória de trabalho por meio de `Matcher::assert()`; finalmente, se a ação for do tipo `RETRACT`, o literal associado é removido da memória de trabalho através de `Matcher::retract()`.

8.3.3 Modelo de Classes

A Figura 80 mostra o modelo de classes para o mecanismo de inferência, obtido dos diversos diagramas de interação apresentados na subseção anterior. Ele sintetiza as especificações das entidades de software envolvidas na realização das funcionalidades do mecanismo.

`Matcher` e `Strategy` foram mantidos na forma de interfaces porque os objetos que oferecem essas funcionalidades poderão ser criados pelo desenvolvedor do agente, segundo os algoritmos que ele desejar utilizar para essas funcionalidades.

Três operações que não aparecem em nenhum dos diagramas de interação foram acrescentadas à interface `ConflictSet`: `add(activation)`, `activation(i)` e `size()`. A primeira é necessária para que o objeto que realiza a interface `Matcher` possa acrescentar ativações ao conjunto de conflito e as duas últimas, para que o objeto que realiza a interface `Strategy` possa ter acesso às ativações contidas em conjunto de conflitos, de forma a selecionar uma delas.

Da mesma forma, também foi adicionada à interface `Activation` a operação `premises(i)`. Assumiu-se que uma ativação deveria ser composta pela conclusão deduzida pelo algoritmo de inferência e pelo conjunto de fatos (premissas) que levaram a essa conclusão. Isso porque algumas estratégias de resolução de conflitos podem utilizar tanto o número de premissas (p. ex., *especificidade*) quanto a sua natureza (p.ex. *não-duplicação*) para decidir qual regra deve ser disparada. Portanto, a operação introduzida tem por objetivo permitir que

o objeto responsável pela estratégia de resolução de conflitos tenha acesso a essas informações.

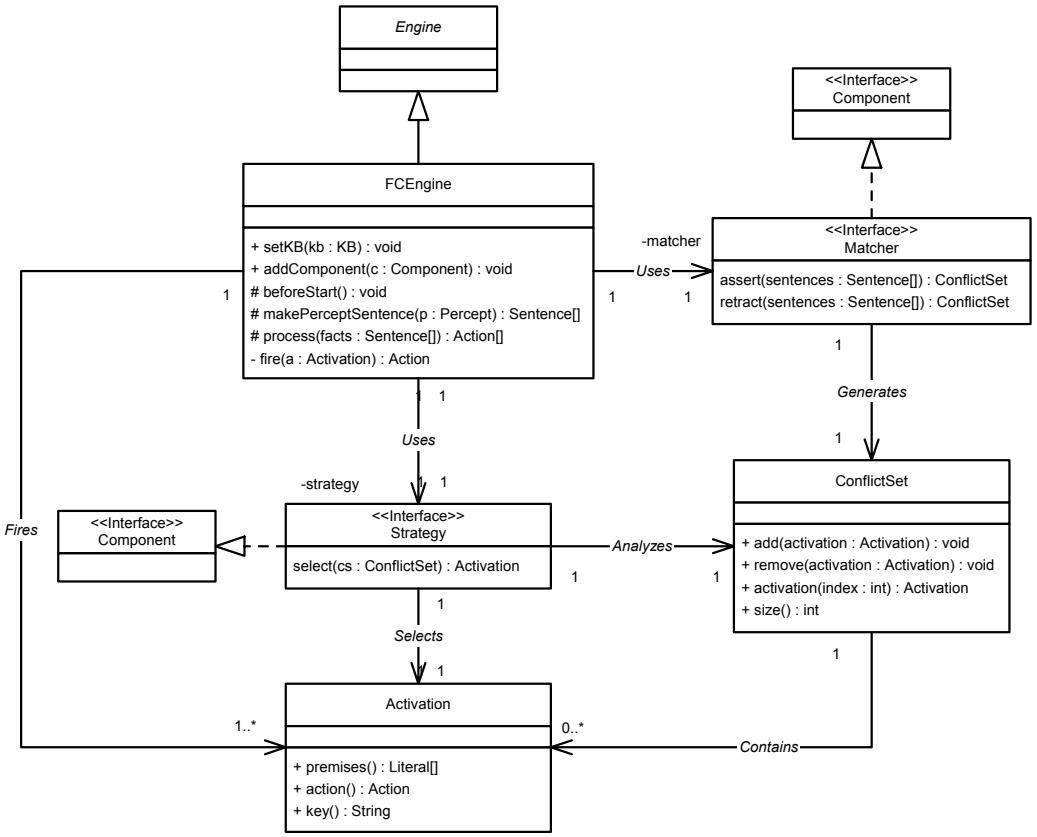


Figura 80: Diagrama de classes para o *Subsistema Mecanismo de Inferência*

8.4 Subsistema Rete

Esta seção tem por objetivo discutir os aspectos a serem considerados no desenvolvimento de um *casador de padrões* de modo a reduzir a quantidade de código e de dados transportados pelo agente. A discussão é realizada através da apresentação dos modelos que descrevem o desenvolvimento de um *casador* segundo o algoritmo de casamento *Rete*, descrito em 4.6. A inclusão da implementação de um algoritmo de casamento específico ao *Framework de Inteligência* proposto facilita o desenvolvimento de aplicações baseadas nessa arquitetura.

Conforme já afirmado, o algoritmo de casamento possui influência preponderante sobre o tamanho da memória de regras transportada pelo agente e sobre sua eficiência computacional. Portanto, o desenvolvimento orientado a objetos de um *casador* deve ser cuidadoso.

Uma das formas de reduzir a quantidade de código e de dados que o agente precisa transportar é isolar as responsabilidades pelas fases de *compilação* e *casamento* e atribuí-las a objetos ou grupos de objetos distintos. O objeto ou grupo de objetos responsável pela compilação da base de regras e fatos para o formato interno de representação utilizado pelo *casador* é necessário somente durante a inicialização do agente, que ocorre em sua agência nativa e, portanto, não precisam ser transportados por ele. Por outro lado, o objeto ou o grupo de objetos responsável pela fase de casamento é necessário somente quando o agente estiver realizando sua tarefa, ou seja, durante sua execução. Portanto, tais objetos precisam ser transportados de uma agência para a outra e o custo de transmiti-los deve ser reduzido. Para que o código dessa parte transportável seja "leve", seus objetos devem possuir um número reduzido de variáveis (atributos) e de métodos (operações), além, é claro, de uma implementação "enxuta" para os métodos existentes.

Especificamente para o algoritmo *Rete*, as responsabilidades pelas duas fases (*compilação* e *casamento*) foram isoladas e atribuídas a objetos distintos. O primeiro deles, `ReteCompiler`, é responsável pela construção da estrutura de nós composta pelos objetos correspondentes aos nós de uma entrada, de duas entradas e aos nós terminais. O segundo, `Rete`, é responsável pela fase de casamento. Tanto `ReteCompiler` quanto `Rete` realizam suas tarefas com a cooperação de outros objetos. O `ReteCompiler` e os objetos que cooperam com ele na construção da rede de nós não são necessários durante a execução do agente que, durante sua migração, leva consigo somente essa estrutura, mantida no objeto `Rete`. Todas as variáveis e métodos não-específicos da fase de casamento, foram eliminados das classes que compõem essa estrutura (os nós), ou seja, os objetos que representam os diversos tipos de nós encapsulam somente variáveis e métodos referentes à fase de casamento do algoritmo.

As classes desenvolvidas tanto para a fase de compilação quanto para a fase de casamento do *Rete* foram agrupadas em um novo subsistema, específico ao Mecanismo de Inferência Progressiva, denominado *Subsistema Rete*.

Os aspectos de mobilidade considerados no desenvolvimento dessas classes são detalhados na subseção seguinte, através da apresentação dos principais diagramas de interação para seus objetos.

8.4.1 Modelo de Comportamento dos Objetos

Esta subseção apresenta os principais diagramas de interação desenvolvidos para o *Subsistema Rete*. São mostrados apenas os diagramas que ilustram as decisões de projeto visando à redução do código e dos dados transportados pelos agentes. Os demais diagramas que compõem o *Modelo de Comportamento dos Objetos* são apresentados no Apêndice A. Todos os diagramas desse modelo têm origem nos diagramas para as duas operações definidas na interface `Matcher`: `assert(sentences)` e `retract(sentences)`.

8.4.1.1 Diagrama de interação para `Rete::assert(sentences)`

A Figura 81 mostra o diagrama de interação para essa operação. Basicamente, ao receber a primeira invocação `assert(sentences)`, o *Rete* cria um objeto da classe `ConflictSet`, que representa o conjunto de conflito, e, então, analisa cada uma das sentenças que compõem o conjunto de sentenças de entrada, verificando se essa sentença é do tipo `Rule` ou `Literal`. Cada um desses tipos é processado de forma diferente, através de `assert(rule)` ou `assert(fact)`, onde `fact` é do tipo `Literal`. Nas demais invocações de `assert(sentences)`, somente a análise e o processamento das sentenças são realizados.

A operação `assert(rule)` dá início à seqüência de operações realizadas durante a compilação da *Base de Conhecimento* para o formato interno do casador. A operação `assert(fact)` inicia a seqüência de operações do casamento de padrões propriamente dito, preenchendo o objeto que representa o conjunto de conflito (`cs`) com os literais que representam as conclusões das regras cujas premissas foram todas satisfeitas.

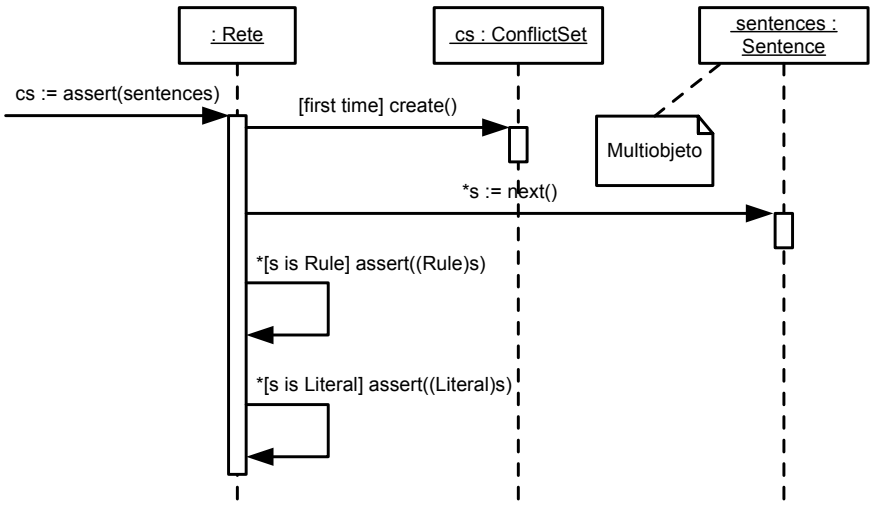


Figura 81: Diagrama de interação para `Rete::assert()`

8.4.1.2 Diagrama de interação para `Rete::assert(rule)`

Quando a operação `assert(rule)` é invocada, fundamentalmente, o objeto `rete` delega o serviço de criação da estrutura de nós a um objeto da classe `ReteCompiler`. O usuário do casador tem visibilidade somente para o objeto da classe `Rete`, o que significa que ele ignora a existência do objeto da classe `ReteCompiler`. Portanto, o relacionamento entre o objeto `Rete` e o objeto `ReteCompiler` segue o princípio de projeto *Delegação* descrito por GAMMA et al. (1995, p.20), sendo que todo ciclo de vida de uma instância de `ReteCompiler` é gerenciado por uma instância de `Rete`. A Figura 82 mostra o diagrama de interação para `assert(rule)`.

Durante a primeira invocação de `assert(rule)`, uma instância de `ReteCompiler` é criada. Nas demais invocações, essa instância é somente utilizada, através da operação `ReteCompiler::assert(rule)`, que sempre retorna o nó de entrada (raiz) da rede (`entry`).

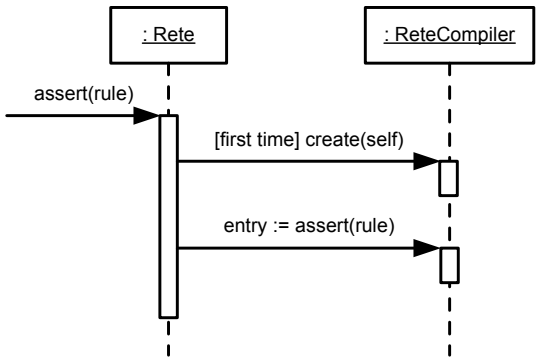


Figura 82: Diagrama de interação para `Rete::assert(rule)`

8.4.1.3 Diagrama de interação para `ReteCompiler::create(rete)`

A análise das ações executadas pela operação `ReteCompiler::create(rete)` é importante para a compreensão da estratégia utilizada na redução da quantidade de dados transportada pelo agente móvel. A Figura 83 apresenta o diagrama de interação para essa operação.

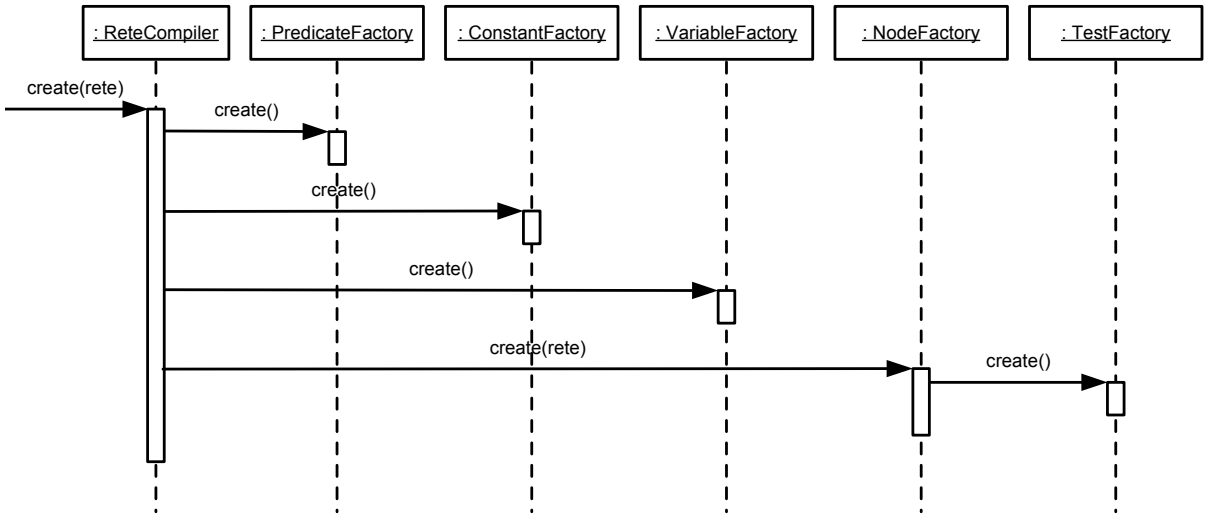


Figura 83: Diagrama de interação para `ReteCompiler::create(rete)`

Com o objetivo de reduzir o número de objetos criados e referenciados por um agente, durante o processo de construção da rede de nós, o compilador deve preocupar-se com a reutilização de objetos correspondentes a predicados, constantes, variáveis, nós e testes idênticos. A responsabilidade pela reutilização desses objetos foi dividida entre cinco classes: `PredicateFatory`, `ConstantFactory`, `VariableFactory`, `NodeFactory` e

`TestFactory`. Essas classes são implementações do padrão de projeto *Flyweight* (GAMMA et al., 1995, p.195).

Cada *Flyweight* (GAMMA et al., 1995, p. 195) mantém uma tabela do tipo **chave - valor**, onde os valores são referências para os respectivos produtos (os *flyweights*) já criados e as chaves são construídas a partir das características de cada produto, de forma a identificá-los de modo único. Toda vez que a criação de um produto é solicitada, a fábrica, antes de criá-lo, verifica em sua tabela se um produto idêntico já foi criado. Caso afirmativo, ela retorna uma referência a esse produto; caso contrário, ela cria um novo produto e o acrescenta na tabela.

O diagrama da Figura 84 mostra um diagrama de interação genérico para uma operação que solicita a criação de um produto a uma fábrica.

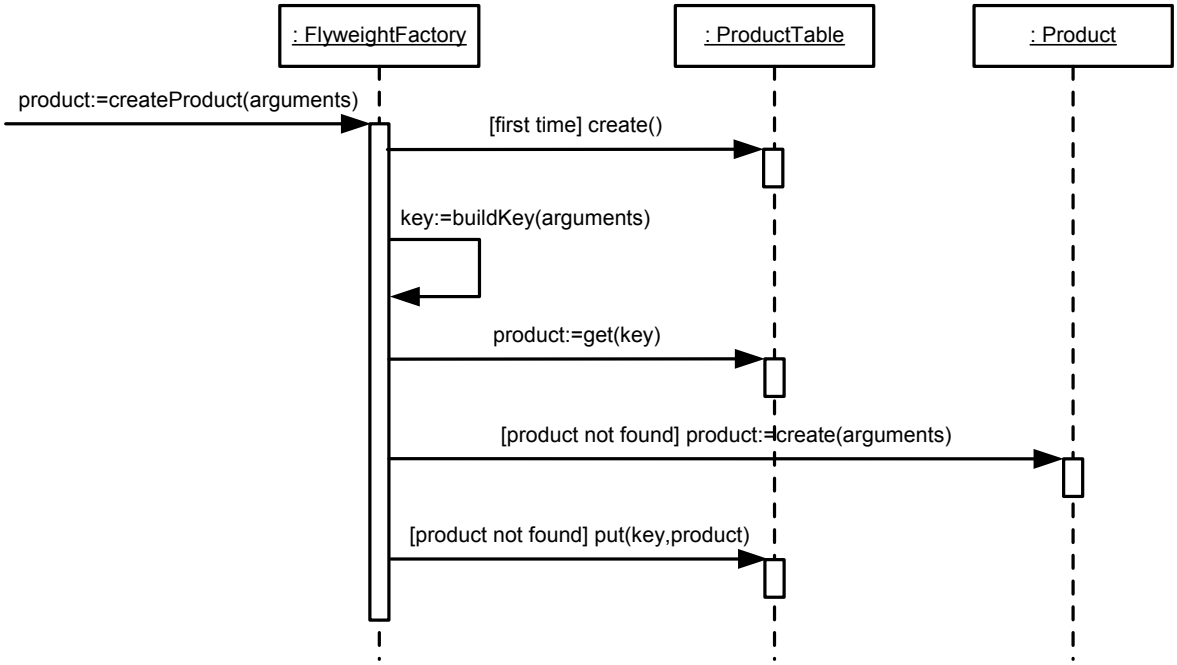


Figura 84: Diagrama de interação para `Factory::createProduct()`

Os argumentos passados como parâmetros de entrada para `createProduct()` são as "partes" a partir das quais o produto será criado. Por exemplo, no caso de um predicado, esses parâmetros seriam o seu nome, o número de seus termos e seus modificadores.

Tanto `NodeFactory` quanto `TestFactory` são implementações de *fábricas parametrizadas* (GAMMA et al., 1995, p. 110), isto é, fábricas que criam vários tipos de produtos, nós e testes, respectivamente, em função dos parâmetros passados para os métodos fábricas. O tipo de nó ou de teste a ser criado é definido em função da lista de parâmetros de entrada. No

entanto, para que esses produtos sejam tratados de modo uniforme durante a fase de casamento, todos os tipos de nós criados por uma fábrica `NodeFactory` são subclasse de uma classe abstrata denominada `Node`, que reúne atributos e operações comuns a todos eles. Da mesma forma, todos os tipos de testes criados por uma fábrica `TestFactory` implementam uma interface denominada `Test`.

A chave criada por `buildKey()` a partir dos argumentos de entrada deve identificar de modo único o produto. Por exemplo, para um nó-de-uma-entrada, as informações que o definem de modo único são o seu nó antecessor e o teste que ele deve realizar (predicado, número de termos, valor da constante ou igualdade de variáveis intraliterais); para um nó-de-duas-entradas, são suficientes as informações sobre quem são seus antecessores à direita e à esquerda. Para os testes, as seguintes informações são usadas na criação da chave: para um teste de predicado, a única informação necessária é o predicado em si; para um teste de número de termos, o próprio número de termos; para um teste de valor de constante, o valor da constante e o índice do termo que deve possuir esse valor; para um teste de igualdade entre variáveis intraliterais, o índice da primeira ocorrência da variável na lista de termos e o índice do termo de sua próxima ocorrência; para um teste de igualdade entre variáveis interliterais, o par formado pelos índices do literal e de seu termo, onde a variável aparece pela primeira vez na expressão antecedente da regra e o par formado pelos índices do literal e de seu termo de sua próxima ocorrência.

Finalmente, para que o objeto `compiler` não seja transportado quando da migração do agente, a variável de `Rete` que mantém uma referência para `compiler` é marcada com o modificador `transient`, da linguagem Java. Objetos referenciados por variáveis com esse tipo de modificador não são serializados.

8.4.1.4 Diagrama de interação para `ReteCompiler::assert(rule)`

O diagrama de interação para `ReteCompiler::assert(rule)` mostrado na Figura 85 ilustra a seqüência de operações realizadas durante a tradução de uma regra, na forma de objetos de conhecimento, para a representação interna do `Rete`, isto é, em uma estrutura de nós interligados.

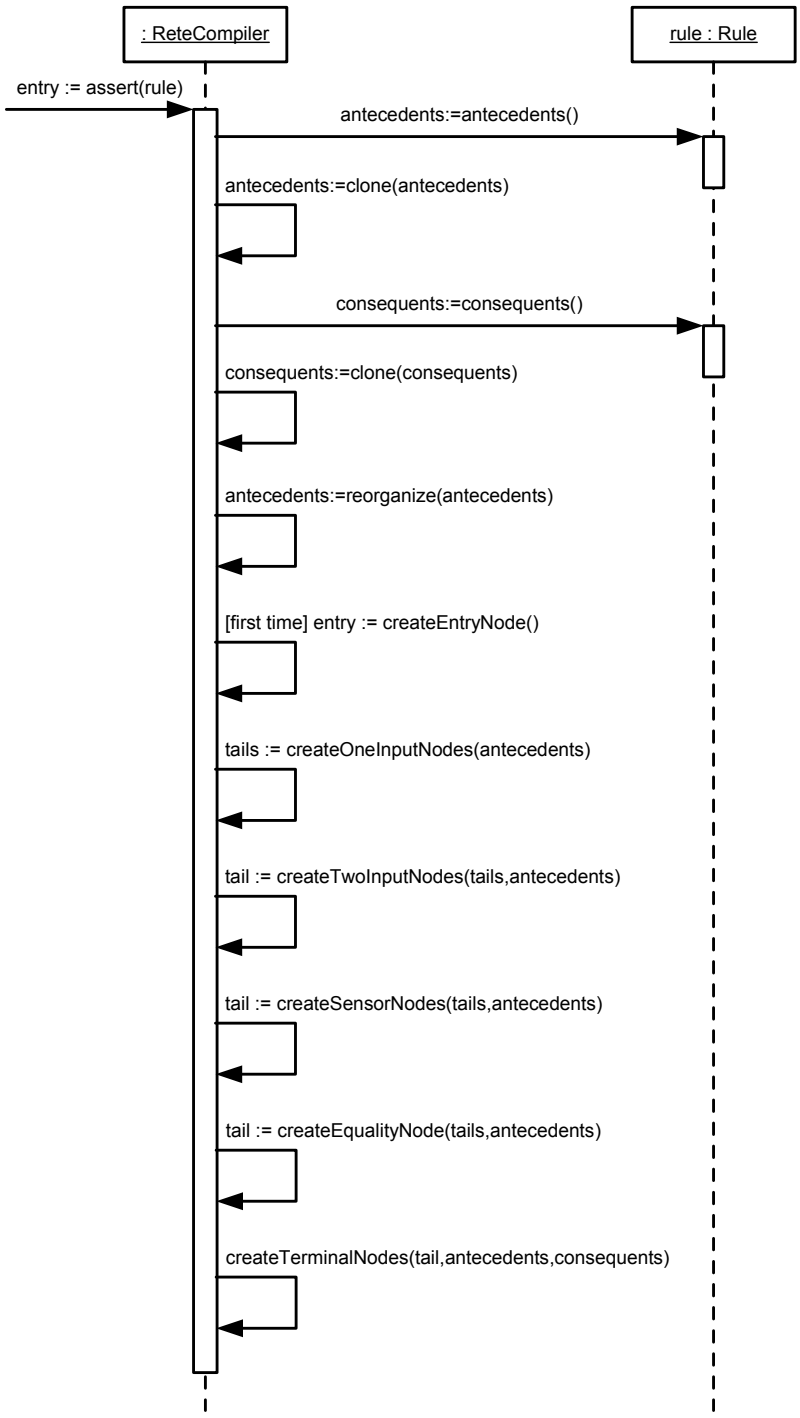


Figura 85: Diagrama de interação para `ReteCompiler::assert(rule)`

Inicialmente é realizada uma clonagem de cada literal presente nas expressões antecedentes e conseqüentes da regra (`clone(...)`). Isso evita que referências para as implementações das *Interfaces de Representação do Conhecimento* desenvolvidas pelo usuário do agente fiquem armazenadas na *Memória de Regras* do *Rete*. A manutenção de referências a tais im-

plementações poderia fazer com que códigos e dados desnecessários fossem transportados. Dois exemplos de dados desnecessários que, normalmente, são acrescentados a essas implementações de usuários são seqüências de caracteres representando o nome da regra e algum comentário elucidativo sobre ela.

No pior caso, a manutenção de referências a implementações do usuário poderia até mesmo impedir a migração do agente, caso algum de seus componentes não pudessem ser serializados para o transporte. Portanto, a operação `clone(…)` realiza uma "tradução" de tais objetos para uma implementação minimalista das *Interfaces de Representação do Conhecimento*, interna ao *Subsistema Mecanismo de Inferência*.

Após a clonagem, os literais presentes na expressão antecedente da regra são reorganizados (`reorganize(…)`) com o objetivo de possibilitar um maior reaproveitamento dos nós-de-duas-entradas, o que ajuda a reduzir o tamanho da estrutura de nós. Isolar as tarefas de reorganização das regras em um único método facilita a utilização e avaliação de novos algoritmos para essa finalidade.

Basicamente, `reorganize(…)` identifica e separa os literais presentes na expressão antecedente de uma regra de acordo com três tipos de predicados: predicados correspondentes a sensores, predicados que estabelecem a *desigualdade* entre variáveis intra e interliterais e predicados que não se enquadram em nenhuma das duas categorias anteriores, chamados de *predicados puros*.

A necessidade de um predicado para representar a desigualdade entre variáveis intra e interliterais pode ser entendida considerando-se, por exemplo, a seguinte regra:

$$A(x,x) \wedge B(y,Paulo) \wedge C(Paulo,y) \rightarrow D(x,y)$$

Essa regra seria ativada tanto para $x = y$ quanto para $x \neq y$. Caso seja necessário, por algum motivo, impor somente a última situação, é preciso que o compilador seja informado de alguma forma. Para tanto, estabeleceu-se um *predicado reservado*, `EQUAL`, com a intenção de usá-lo nessas situações específicas, em conjunto com o indicador de negação por falha (~). Dessa forma, para ser ativada somente quando $x \neq y$, a regra deveria ser rescrita como:

$$A(x,x) \wedge B(y,Paulo) \wedge C(Paulo,y) \wedge \sim Equal(x,y) \rightarrow D(x,y)$$

Os literais cujos predicados são puros são colocados no início da expressão antecedente, em ordem alfabética crescente dos nomes dos predicados. Esses literais dão origem aos nós de uma e de duas entradas do algoritmo *Rete* clássico descrito em 4.6.

Quanto aos literais cujos predicados correspondem a sensores, para que o maior número possível de variáveis já tenha seus valores determinados no momento da chamada do procedimento externo associado, independentemente de suas posições na regra original, o compilador os reordena de modo que eles fiquem o mais à direita possível na expressão antecedente, após os literais de predicados puros. Desse modo, quando as consultas externas representadas por esses sensores forem realizadas, somente um pequeno número de variáveis precisará ter seus valores determinados. A responsabilidade da consulta externa indicada pelos predicados do tipo `Sensor` foi atribuída a um novo tipo de nó, denominado `SensorNode`.

Os literais cujos predicados correspondem a *desigualdades* entre variáveis intra e interliterais são colocados após os sensores, no final da expressão antecedente. A tarefa de realizar os testes de desigualdade foi atribuída a nós-de-uma-entrada situados imediatamente antes dos nós terminais. Dessa forma, evita-se que referências a objetos que representam esses testes sejam replicadas em todos os nós terminais cujas estruturas de nós precedentes sejam idênticas.

Após a reorganização dos literais presentes na expressão antecedente e conseqüente da regra, esses elementos são traduzidos na estrutura de nós correspondente, através dos métodos `createEntryNode()`, `createOneInputNodes()`, `createTwoInputNodes()`, `createSensorNodes()`, `createEqualityNode()` e `createTerminalNodes()`.

Os diagramas de interação para esses métodos, bem como para `clone(…)`, `reorganize(…)` e todos os demais métodos introduzidos a partir desses, estão apresentados no Apêndice A. Esses diagramas dão origem a todas as operações presentes nas diversas classes de objetos que representam as fábricas *flyweights* (GAMMA et al., 1995, p. 195) no modelo de classes do *Subsistema Rete* apresentado em 8.4.2.

A Figura 86 mostra uma estrutura típica de um plano da rede, construída por `ReteCompiler::assert(rule)`, correspondente a uma única regra, incluindo os nós sensores (`SensorNode`) e os nós-de-uma-entrada para testar as desigualdades intra e interliterais.

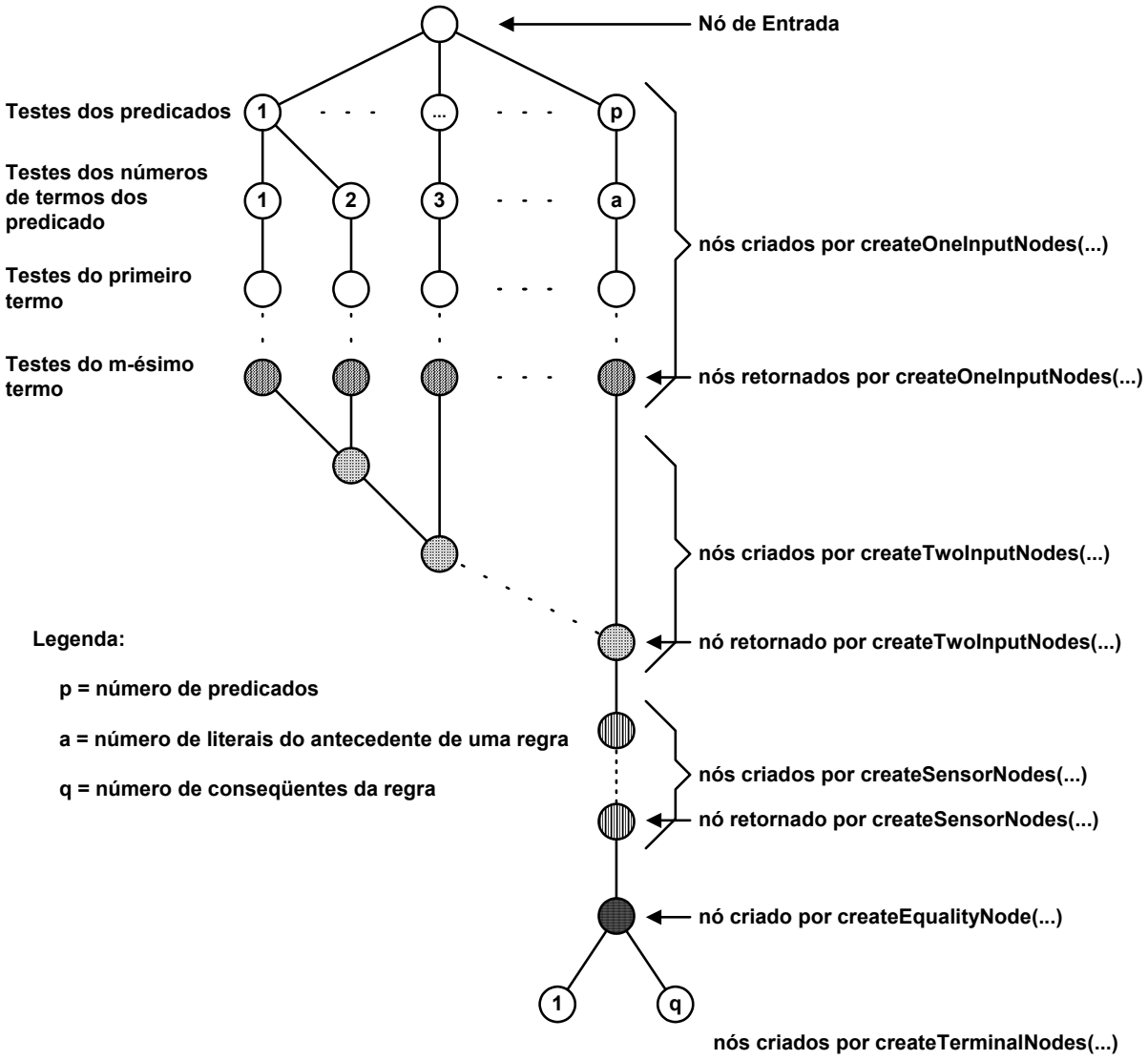


Figura 86: Estrutura típica de nós construída por `ReteCompiler::assert(rule)`

8.4.1.5 Diagrama de interação para `Rete::assert(fact)`

A operação `Rete::assert(fact)` é invocada durante a fase de casamento, quando um fato é adicionado à memória de trabalho, disparando o processo de casamento. A Figura 87 mostra a seqüência de interações para essa operação.

Inicialmente, verifica-se se o predicado associado ao literal `fact` existe na tabela de predicados criados por `PredicateFactory` durante a fase de compilação. Se esse predicado

não estiver na tabela, isso significa que ele não aparece na expressão antecedente de nenhuma regra e, portanto, não é necessário prosseguir com o processo de casamento.

Se o predicado está presente na tabela de `PredicateFactory`, verifica-se então se `fact` é um literal positivo ou negativo. Caso `fact` seja positivo, `addSentence(…)` o envia aos nós da rede para que os testes pertinentes sejam aplicados. Se `fact` passar por todos os testes, ele será armazenado na memória de pelo menos um nó-de-duas-entradas. Caso `fact` seja negativo, `removeSentence(…)` também o envia aos nós da rede para que os testes pertinentes sejam aplicados. Porém, se `fact` passar por todos os testes, ele e todas as sentenças em que ele aparece na expressão antecedente das regras serão removidas das memórias dos nós-de-duas-entradas, devido à utilização da estratégia de *negação por falha*.

Os diagramas de interação para `addSentence(…)` e `removeSentence(…)`, assim como os de todos os demais métodos introduzidos a partir desses, estão apresentados no Apêndice A. Esses diagramas dão origem a todas as operações presentes nas diversas classes de objetos que representam nós e testes no modelo de classes do *Subsistema Rete* apresentado em 8.4.2.

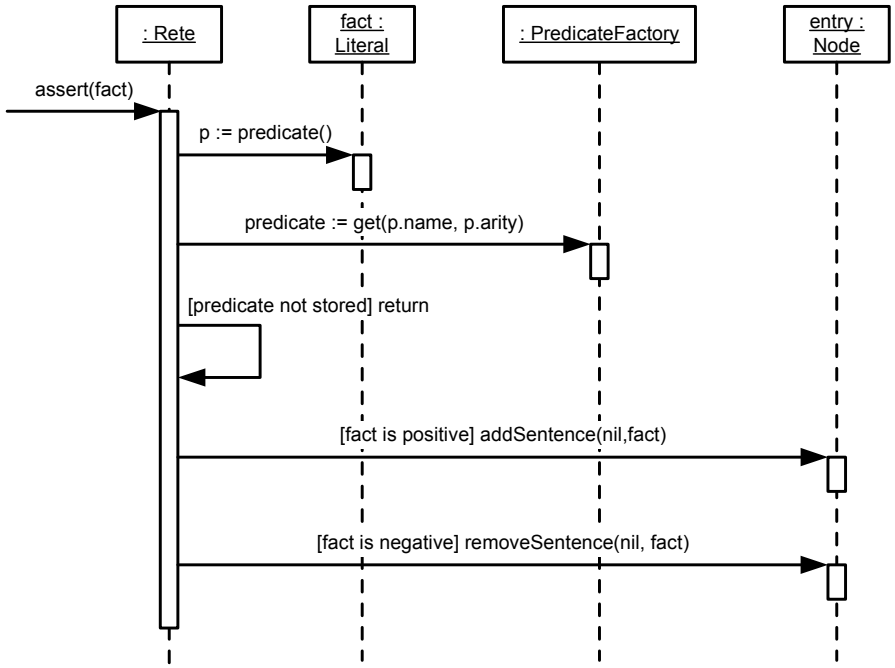


Figura 87: Diagrama de interação para `Rete::assert(fact)`

É importante ressaltar que o objeto `fact` pode ter sido gerado pelo serviço de inteligência da agência hospedeira e que não é conveniente manter-se na rede uma referência direta

para essa implementação local. Se isso ocorresse, durante o percurso de seu itinerário, o código do agente iria crescendo devido as diferentes implementações que cada agência visitada pode possuir para as *Interfaces de Representação do Conhecimento*. No pior caso, a manutenção de uma referência para uma implementação local poderia até mesmo impedir a migração do agente, pois alguns sistemas, como por exemplo o *Aglets*, não permitem que classes que não foram criadas a partir da base de código do agente sejam transportadas.

Portanto, para evitar uma tentativa de transporte dessas diversas implementações, quando um fato passa por todos os testes e deve ser adicionado à memória de um nó-de-duas-entradas, ele é "traduzido" por esse nó para uma implementação minimalista das *Interfaces de Representação do Conhecimento*. Nesse processo de tradução, as fábricas *flyweights* `ConstantFactory` e `PredicateFactory` são utilizadas. `PredicateFactory` é utilizada simplesmente como uma fornecedora de implementações de objetos `Predicate` já criados durante a fase de compilação das regras por `ReteCompiler`. Maiores detalhes sobre essa "tradução" são apresentados em 8.4.2.3.

8.4.1.6 Diagrama de interação para `Rete::retract(sentences)`

Como mostrado na Figura 88, o diagrama de interação para essa operação é muito parecido com o diagrama de `Rete::assert(sentences)`, dispensando comentários.

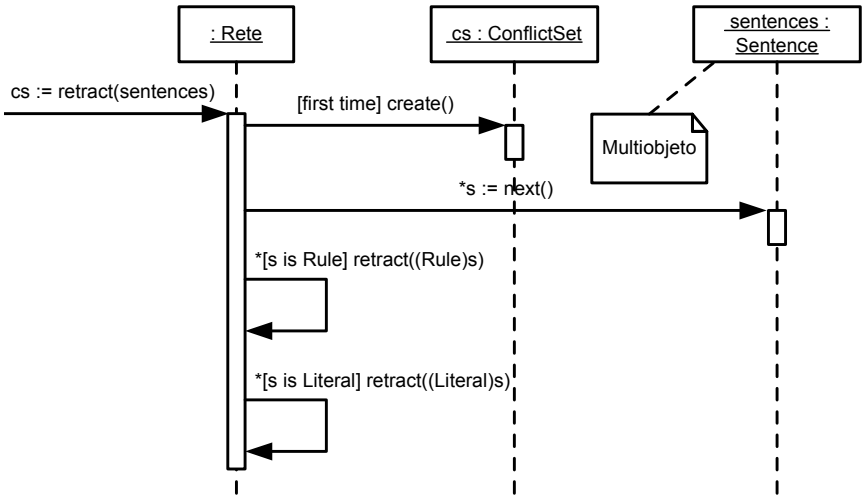


Figura 88: Diagrama de interação para `Rete::retract(sentences)`

8.4.1.7 Diagrama de interação para `Rete::retract(rule)`

O diagrama de interação para essa operação, mostrado na Figura 89, também é muito parecido com o diagrama de `Rete::assert(rule)`, dispensando comentários.

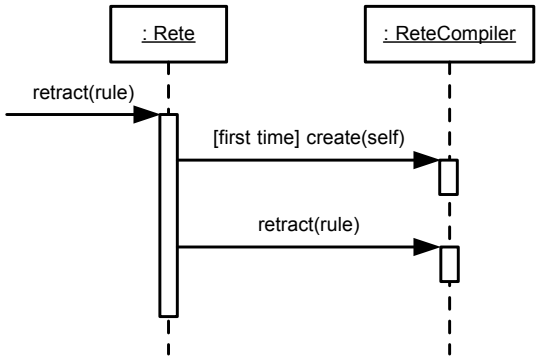


Figura 89: Diagrama de interação para `Rete::retract(rule)`

8.4.1.8 Diagrama de interação para `ReteCompiler::retract(rule)`

O diagrama de interação para `ReteCompiler::retract(rule)`, mostrado na Figura 90, ilustra a seqüência de operações realizadas durante a remoção de uma regra da Memória de Regras (estrutura de nós). Deve-se notar que essa não é uma operação normalmente invocada enquanto o agente está em execução. A necessidade de remover uma regra durante o estado ativo do ciclo de vida de um agente móvel pode implicar na necessidade de transporte dos códigos e dos dados das classes envolvidas com a fase de compilação do algoritmo *Rete*, isto é, `ReteCompiler` e de todas as classes das quais ela depende.

O processo de remoção de nós realizado por `deleteOneInputNodes(…)`, `deleteTwoInputNodes(…)`, `deleteSensorNodes(…)`, `deleteEqualityNode(…)` e `deleteTerminalNodes(…)`, basicamente é o seguinte: os nós correspondentes aos diversos literais da expressão antecedente da regra são percorridos do topo para os nós terminais, verificando-se o número de nós sucessores de cada um. Se o número de sucessores de um nó é igual a 1, esse nó é colocado em uma lista de exclusão, caso contrário, esse nó e todos os seus antecessores são removidos da lista de exclusão. Quando a análise de todos os nós terminais na regra em questão é concluída, os nós presentes na lista são de fato removidos.

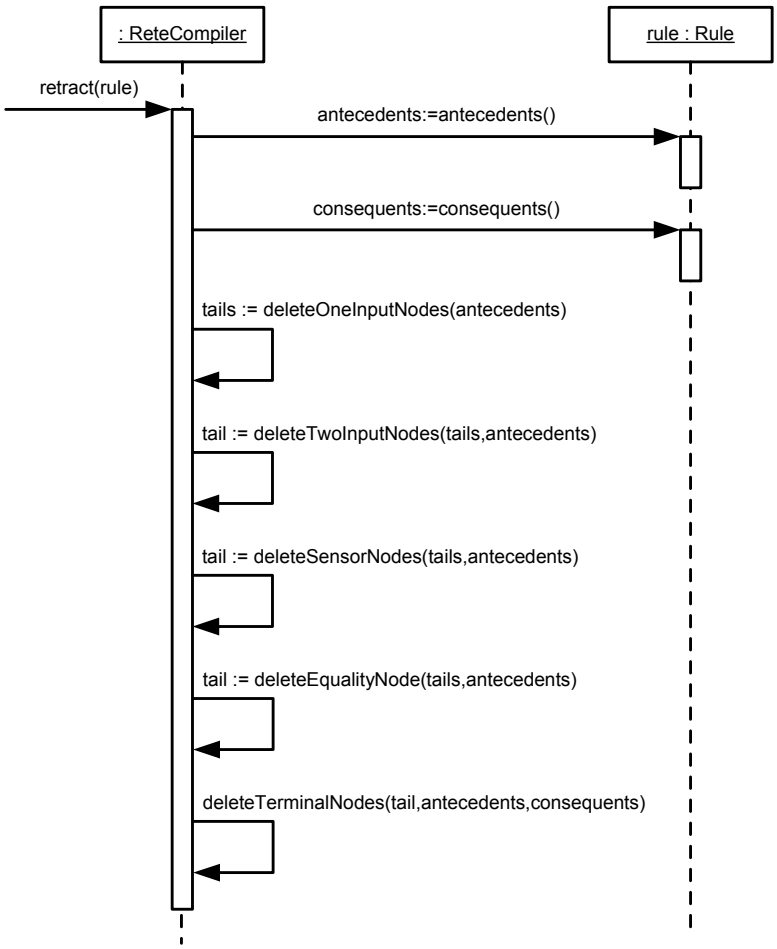


Figura 90: Diagrama de interação para `ReteCompiler::retract(rule)`

8.4.1.9 Diagrama de interação para `Rete::retract(fact)`

Da mesma forma que para `Rete::retract(sentences)`, o diagrama de interação para a operação `Rete::retract(fact)`, mostrado na Figura 91, é muito parecido com o diagrama de `Rete::assert(fact)`, sendo digna de nota somente a inversão nas condições a serem satisfeitas para as chamadas de `addSentence(…)` e de `removeSentence(…)`.

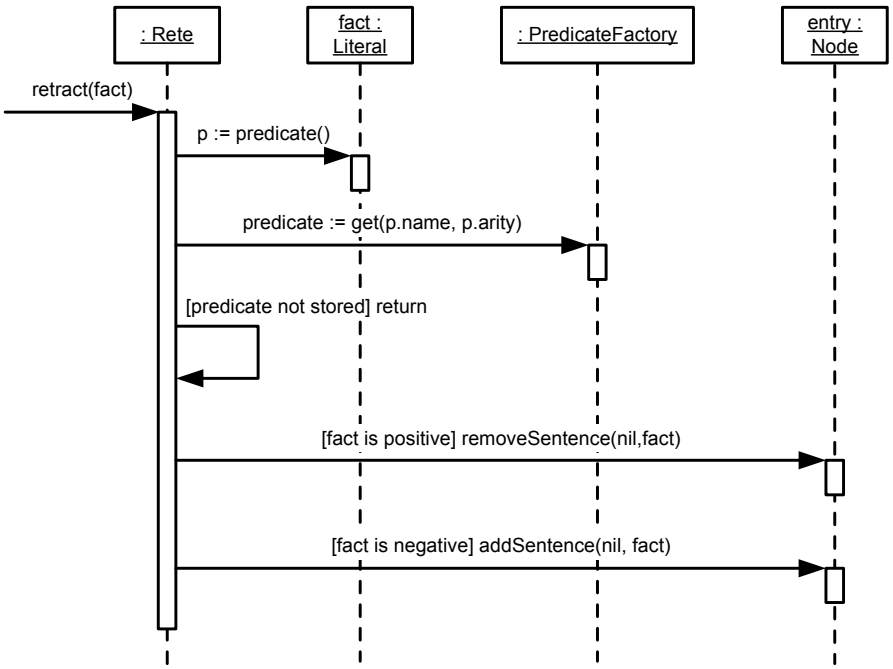


Figura 91: Diagrama de interação para `Rete::retract(fact)`

8.4.2 Modelo de Classes para o Rete

O modelo de classes para o *Rete*, criado a partir dos diagramas de comportamentos de seus objetos descritos em 8.4.1 e no Apêndice A, é composto pelos diagramas apresentados nos itens seguintes.

8.4.2.1 Diagrama Principal

O diagrama principal para o *Subsistema Rete*, mostrado na Figura 92, dá ênfase às classes `Rete` e `ReteCompiler` , explicitando as operações a elas atribuídas. Nota-se que, por uma questão de flexibilidade e coesão funcional, adotou-se o princípio de projeto *Delegação* (GAMMA et al., 1995, p.20) entre as classes `Rete` e `Node`, que representa o nó de entrada (raiz) da estruturas de nós. Dessa forma, evitou-se o relacionamento de generalização entre elas, o que levaria à inclusão em `Rete` de operações que não seriam compatíveis com suas responsabilidades de *casador de padrões*.

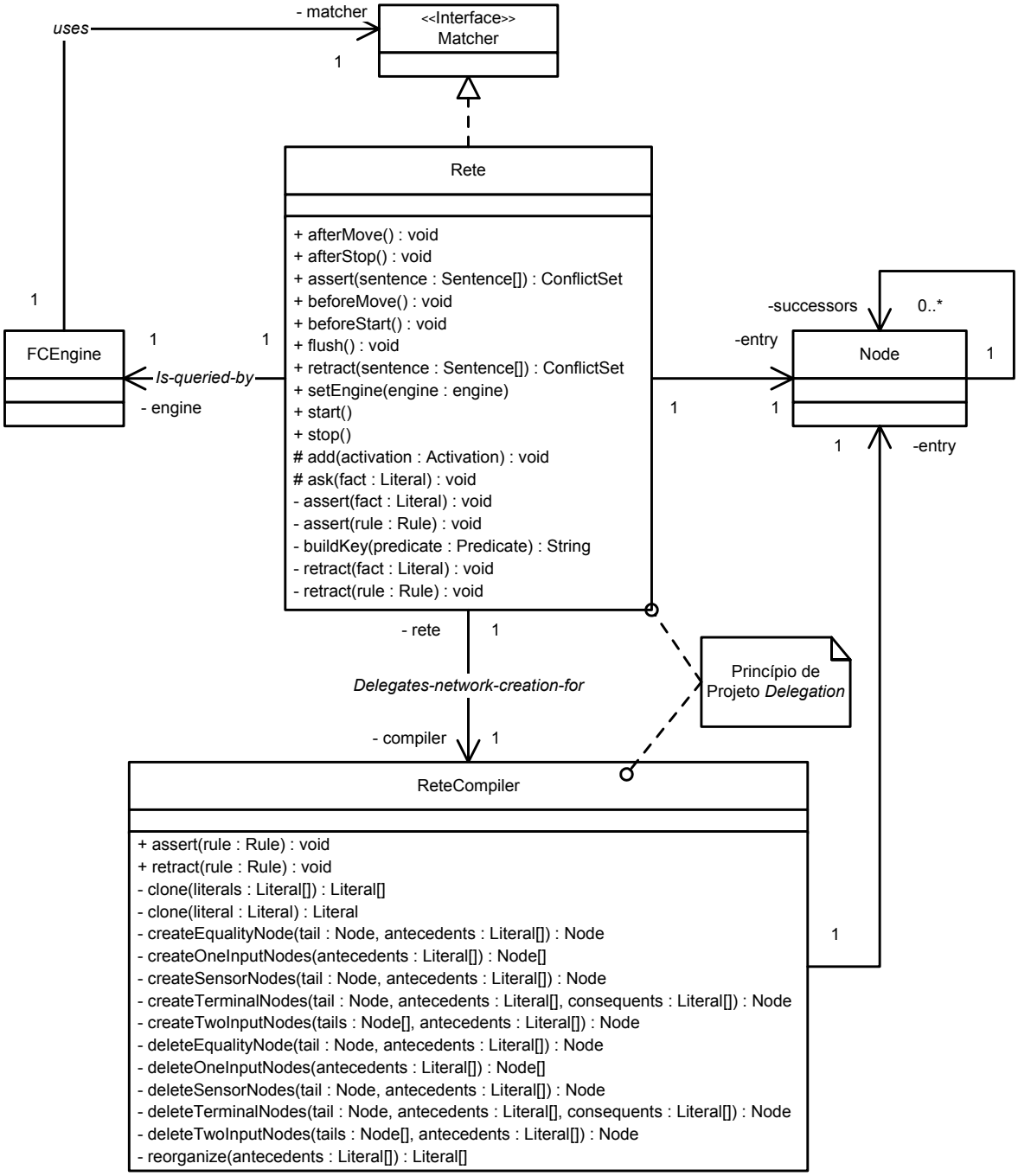


Figura 92: Diagrama de classes para o *Subsistema Rete*: `Rete` e `ReteCompiler`

8.4.2.2 Diagramas para as Fábricas *Flyweights*

Os diagramas da Figura 93 a Figura 97 destacam as classes responsáveis pelas fábricas *flyweights*, explicitando seus clientes, seus produtos e as operações a elas atribuídas.

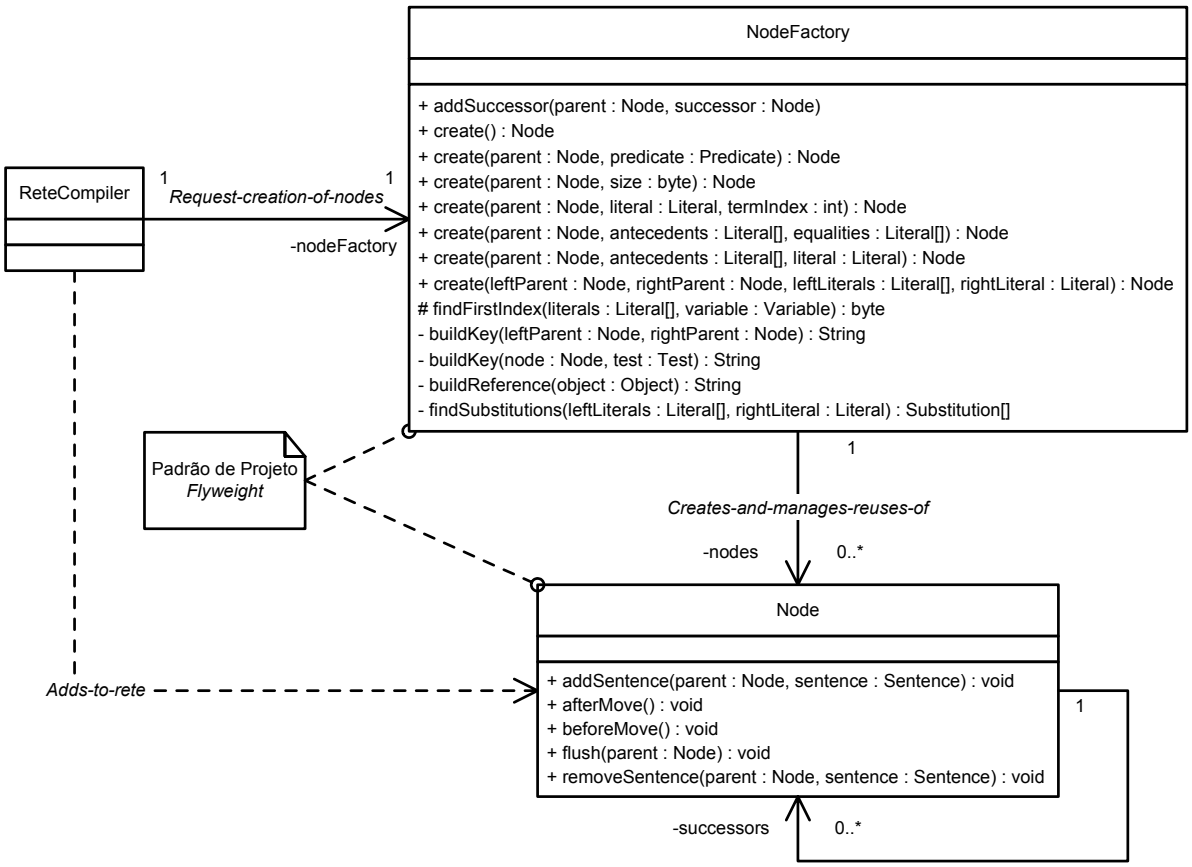


Figura 93: Diagrama de classes para o *Subsistema Rete*: `NodeFactory`

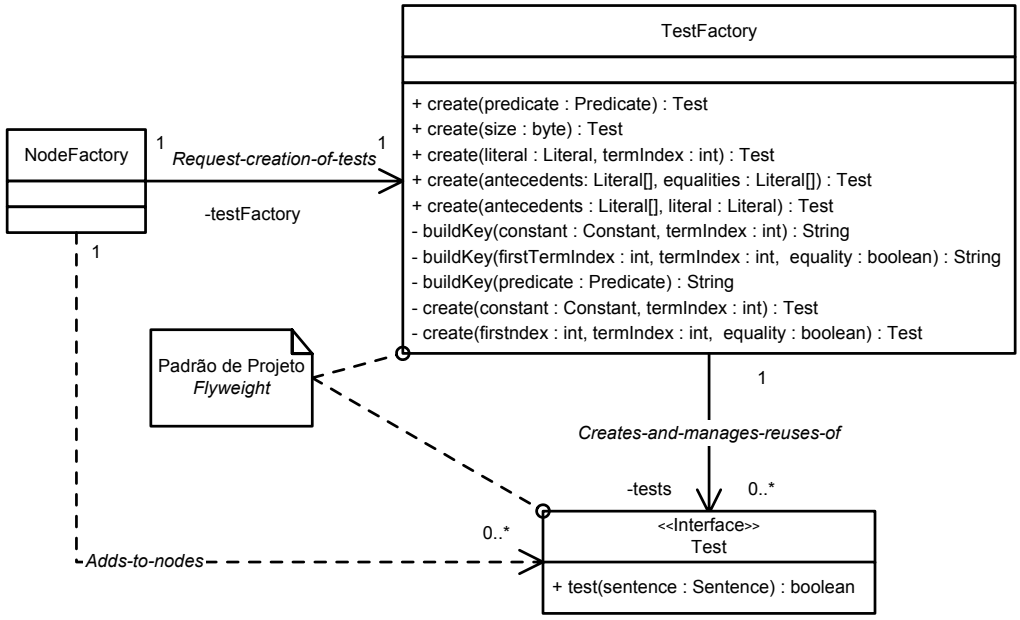


Figura 94: Diagrama de classes para o *Subsistema Rete*: `TestFactory`

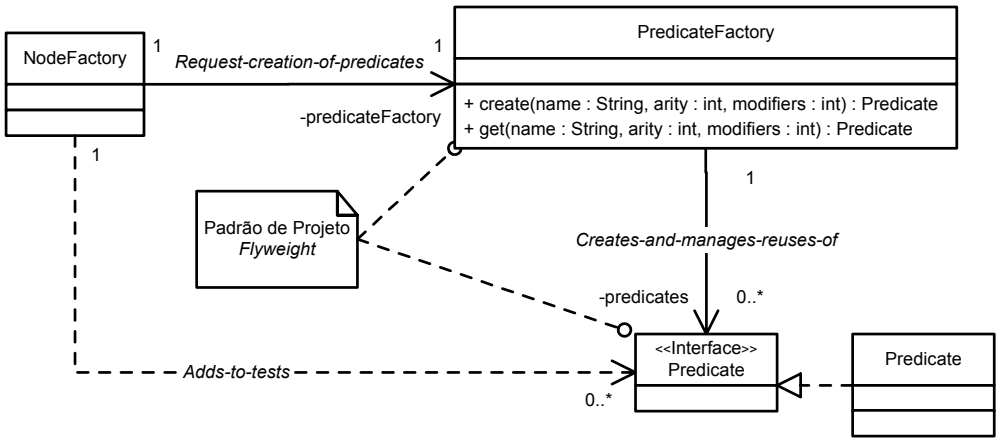


Figura 95: Diagrama de classes para o *Subsistema Rete*: `PredicateFactory`

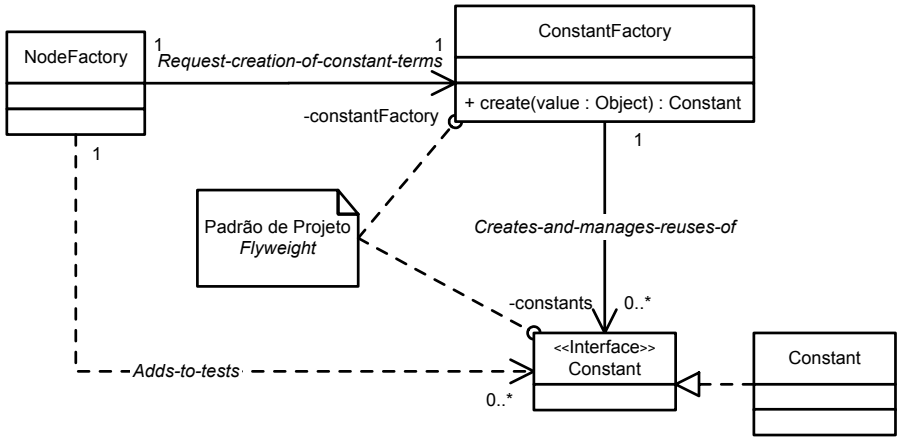


Figura 96: Diagrama de classes para o *Subsistema Rete*: `ConstantFactory`

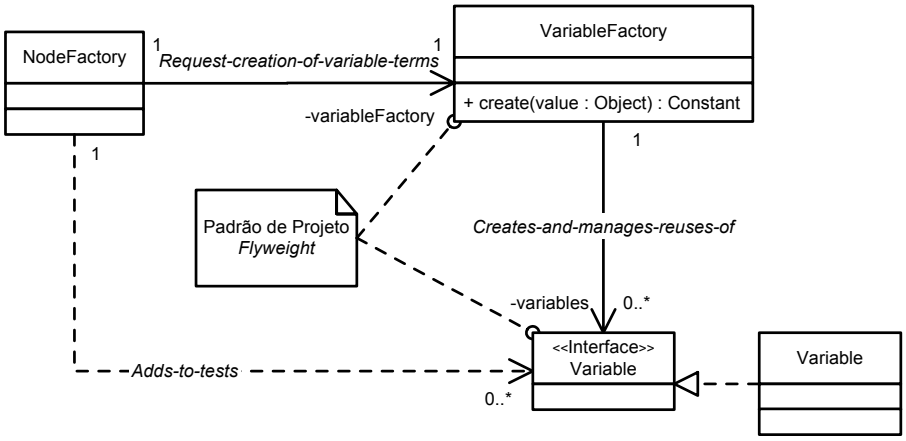


Figura 97: Diagrama de classes para o *Subsistema Rete*: `VariableFactory`

8.4.2.3 Diagramas para a Hierarquia de Nós

A hierarquia de classes projetada para representar os diversos tipos de nós existentes em uma estrutura *Rete* é apresentada no diagrama da Figura 98. Nesse diagrama, nas classes `Node1`, `Node2`, `SensorNode` e `Terminal`, são indicadas somente as operações herdadas da classe `Node` cujos comportamentos foram especializados por cada uma delas.

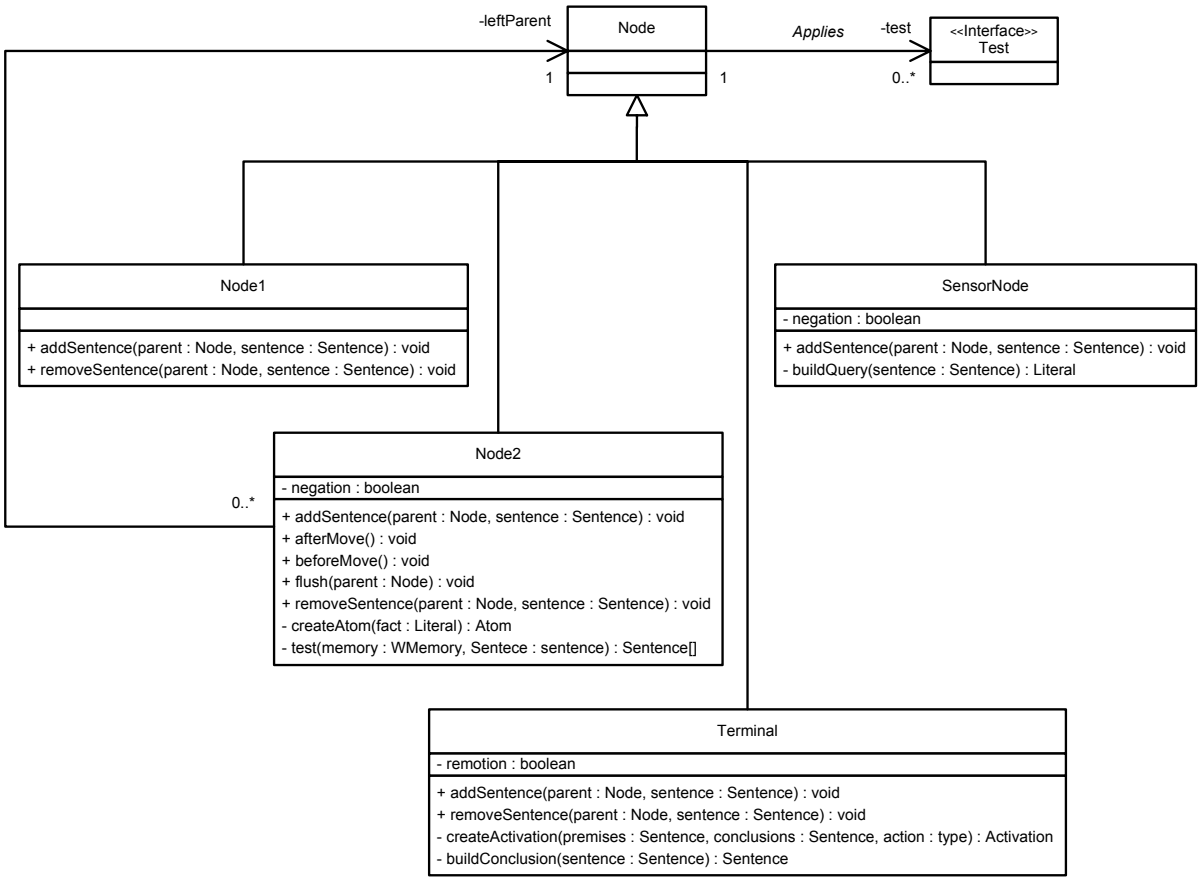


Figura 98: Diagrama de classes para o *Subsistema Rete*: hierarquia de `Node`

Nos objetos da classe `Node2`, os parâmetros de entrada das operações `addSentence()` e `removeSentence()` podem ser tanto um único literal positivo fechado, isto é, um átomo fechado, quanto uma conjunção de literais positivos fechados. Para que fosse possível passar, de maneira uniforme, esses dois tipos distintos de parâmetros para os métodos de `Node2` que implementam essas operações, utilizou-se o padrão de projeto *Composite* (GAMMA et al., 1995), resultando na introdução das classes `Conjunct`, `Atom` e `Conjunction` ao pacote do *Rete*. Além disso, como esses tipos de nós memorizam as sen-

tenças que recebem de seus antecessores, também foram incluídas nesse pacote as classes `LeftMemory` e `RightMemory`, utilizadas para representar as memórias à esquerda e à direita de `Node2`. A união das memórias à direita e à esquerda de todos os nós-de-duas-entradas constitui a *Memória de Trabalho* do *casador*. Os relacionamentos entre essas classes e interfaces estão mostrado na Figura 99.

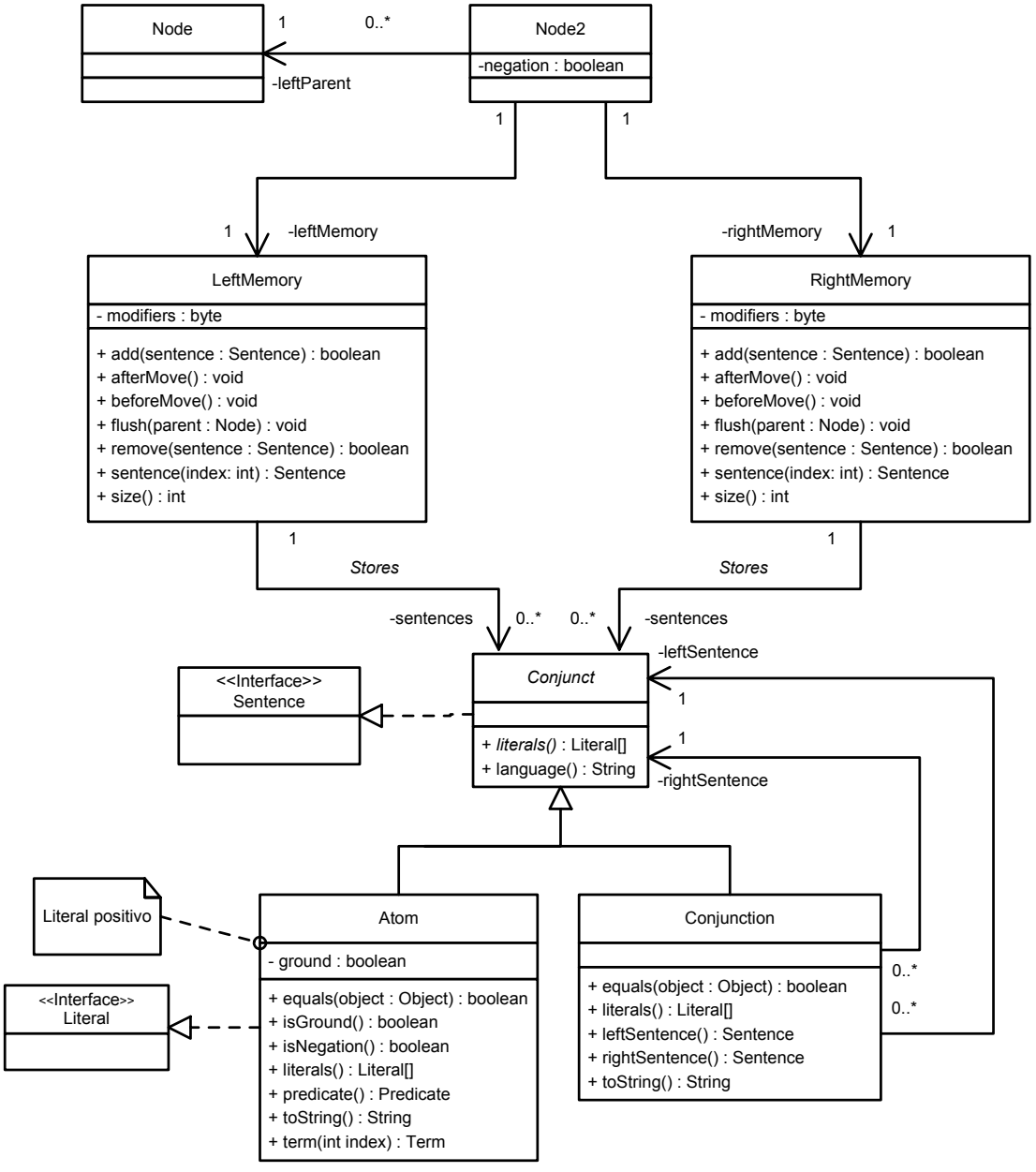


Figura 99: Diagrama de classes para o *Subsistema Rete*: Memórias de `Node2`

Como mostrado na figura acima, a classe abstrata `Conjunct` implementa a interface `Sentence`, sendo especializada tanto por `Conjunction` quanto por `Atom`, que, portanto, podem ser passadas com parâmetro de entrada para as operações `addSentence()` e `removeSentence()`.

Todos os nós-de-duas-entradas com pelo menos um nó-de-uma-entrada como antecedente recebem, através de `addSentence()` ou `removeSentence()`, objetos do tipo `Literal`. Esses objetos satisfazem todos os testes aplicados pelos nós antecessores, devendo, portanto, ser armazenados ou removidos da memória à esquerda ou à direita do nó-de-duas-entradas em questão. Antes de armazená-los, o nó-de-duas-entradas "traduz" esses objetos para instâncias da classe `Atom`, evitando que referências a implementações de usuário e de agências visitadas permaneçam em seu interior. Nessa tradução são utilizados os serviços das fábricas flyweights descritas em 8.4.2.2.

Continuando o processo, todas conjunções formadas pelas sentenças presentes nas memórias à esquerda e à direita, que passarem pelos testes aplicados pelo nó-de-duas-entradas, são encaminhadas ao nó-de-duas-entradas seguinte. Portanto, cada entrada das memórias esquerda e direita de um nó-de-duas-entradas corresponde a um literal fechado positivo (`Atom`) ou a uma conjunção de literais fechados positivos (`Conjunction`) ou, de forma mais genérica a uma sentença, visto que ambos, através de `Conjuct`, implementam a interface `Sentence`.

Para reduzir a quantidade de dados, relativa à *memória de trabalho*, a ser serializada e transportada, a classe `Conjunction` foi projetada de modo a manter em seu interior referências a somente duas outras sentenças (`leftSentence` e `rightSentence`). Desse modo, cada entrada de memória mantém, no máximo, referência a dois outros objetos, das classes `Atom` ou `Conjunction`, numa tipo de *lista ligada*. Isso evita a repetição desnecessária de referências entre as memórias dos diversos nós-de-duas-entradas, o que fatalmente ocorreria caso cada entrada fosse implementada como um arranjo (*array*) de referências para objetos da classe `Atom` somente.

O tratamento das sentenças recebidas por uma instância de `Node2` depende, além do tipo de ação pretendida (adição ou remoção), da memória à qual a ação se aplica (à esquerda ou à direita) e do tipo de literal que a instância de `Node2` representa (positivo ou negativo).

A informação sobre a ação pretendida vem da própria operação solicitada: `addSentence()` ou `removeSentence()`. A informação sobre em qual memória armazenar, ou de qual memória remover a sentença, é obtida da comparação entre uma referência ao nó remetente (`parent`), passada a `addSentence()` e a `removeSentence()` como parâmetro de entrada, e uma referência ao nó antecessor esquerdo (`leftParent`), transferida a instância de `Node2` durante sua criação. O tipo de literal que a instância de `Node2` representa é mantido em seu atributo `negation`.

No atributo `modifiers` as instâncias de `LeftMemory` e `RightMemory` mantêm a informação de que a memória em questão contém sentenças (literais ou conjunções) associadas a predicados que representam eventos (`EVENT`) ou que são transientes (`TRANSIENT`). Memórias associadas ao modificador `EVENT` têm todo o seu conteúdo removido após cada episódio de inferência. O conteúdo das memórias associadas ao modificador `TRANSIENT` é removido antes de cada migração do agente.

A Figura 100 mostra detalhes dos relacionamentos da classe `SensorNode`.

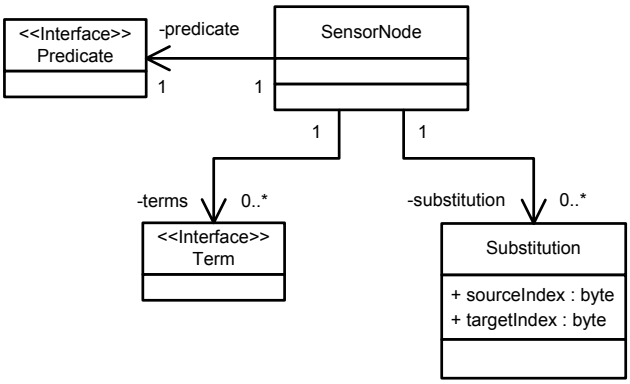


Figura 100: Diagrama de classes para o *Subsistema Rete*: `SensorNode`

Sempre que uma instância de `SensorNode` recebe uma conjunção de literais, proveniente de seu nó antecessor, uma solicitação de informação adicional é disparada. Essa solicitação é construída eliminando-se o maior número possível de variáveis do literal correspondente ao sensor, com o auxílio da conjunção recebida. Os índices das variáveis cujos valores devem ser obtidos a partir da conjunção são determinados durante a fase de compilação e transferidos para o nó sensor na forma de uma coleção de substituições (`substitutions`). Cada substituição (`substitution`) corresponde a uma variável que deve ser eliminada.

Instâncias de `SensorNode` também possuem referências a uma lista de termos (`terms`), onde são mantidas as constantes pré-existentes no literal sensor (antes das substituições), e a uma instância de `Predicate`, que representa o tipo de solicitação a ser realizada. Essas informações são utilizadas na criação de um objeto da classe `Atom`, correspondente à solicitação formulada, que é enviado ao agente através da operação `EClient::ask()`. As variáveis existentes nessa instância de `Atom` no momento da solicitação devem ser resolvidas pelo próprio agente, por outro mecanismo de IA, ou pela agência hospedeira.

O parâmetro de retorno de `EClient::ask()` é uma lista de fatos, isto é, uma lista de literais sem variáveis. As conjunções formadas por cada fato dessa lista com a sentença recebida inicialmente pela instância de `SensorNode` são adicionadas aos nós sucessores. Uma lista vazia significa que a condição testada pelo sensor não é satisfeita e que, portanto, nenhuma sentença deverá ser enviada, impedindo a ativação da regra.

A Figura 101 mostra detalhes dos relacionamentos da classe `Terminal`. Objetos da classe `Terminal` são responsáveis por construírem uma conclusão através da eliminação das variáveis da expressão conseqüente da regra, a partir da sentença fechada (expressão antecedente) recebida de seu nó antecessor e de uma coleção de substituições (`substitutions`), determinadas durante a fase de compilação. Cada substituição (`substitution`) corresponde a uma variável que deve ser eliminada da expressão conseqüente. Cada conclusão, instância de `Atom`, é encapsulada em uma instância de `Activation` e adicionada ao conjunto de conflito do mecanismo de inferência.

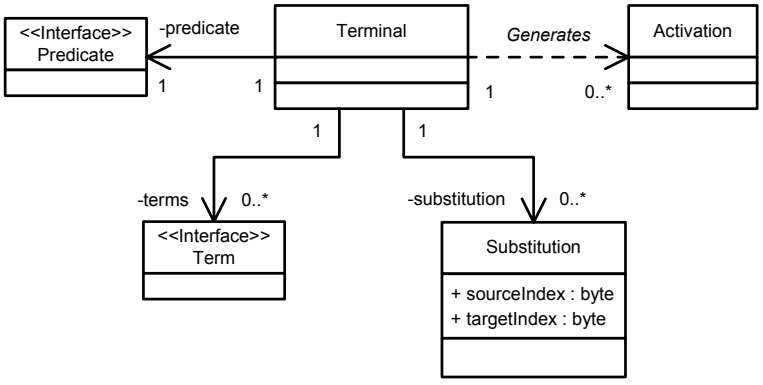


Figura 101: Diagrama de classes para o *Subsistema Rete*: `Terminal`

De modo semelhante a instâncias de `SensorNode`, instâncias de `Terminal` também possuem referências a uma lista de termos (`terms`), onde são mantidas as constantes pré-existentes no literal conclusão (antes das substituições), e a uma instância de `Predicate`.

8.4.2.4 Diagramas para a Hierarquia de Testes

O diagrama da Figura 102 ilustra a interface `Test` e a hierarquia de classes que a implementam, projetadas para realizarem os diversos tipos de testes aplicados pelos nós do algoritmo *Rete*. A interface `Test` oferece uma única operação, necessária somente durante a fase de casamento, `test(sentence)`, cujo parâmetro de retorno booleano apresenta o valor `true` se a sentença de entrada (`sentence`) passar pelo teste que o objeto representa.

É digna de nota a utilização, novamente, do padrão de projeto *Composite* (GAMMA et al., 1995, p.163) no relacionamento entre a interface `Test` e a classe `TestComposite`, projetada para ser utilizada pelos nós-de-duas-entradas e pelos nós-de-uma-entrada que testam a *desigualdade* entre variáveis intra e interliterais. Durante a fase de compilação da rede, quando esses nós são criados, eles recebem uma referência para um objeto que implementa a interface `Test`. Esse objeto é responsável por realizar, de forma transparente para os nós, todos os testes que precisam ser aplicados por eles às sentenças recebidas de seus antecessores. Portanto, esse objeto representa uma composição de testes que deve se comportar como um único teste. Por isso, utilizou-se o padrão de projeto *Composite* (GAMMA et al., 1995, p.163) onde o objeto que representa a composição de teste é uma instancia da classe `TestComposite`, que, além de implementar a interface `Test`, também corresponde a uma agregação de vários objetos do tipo `Test`. Com a utilização desse padrão, os diversos tipos de nós existentes na rede tratam todos os testes de uma maneira uniforme, ignorando a diferença entre testes individuais ou composições de testes.

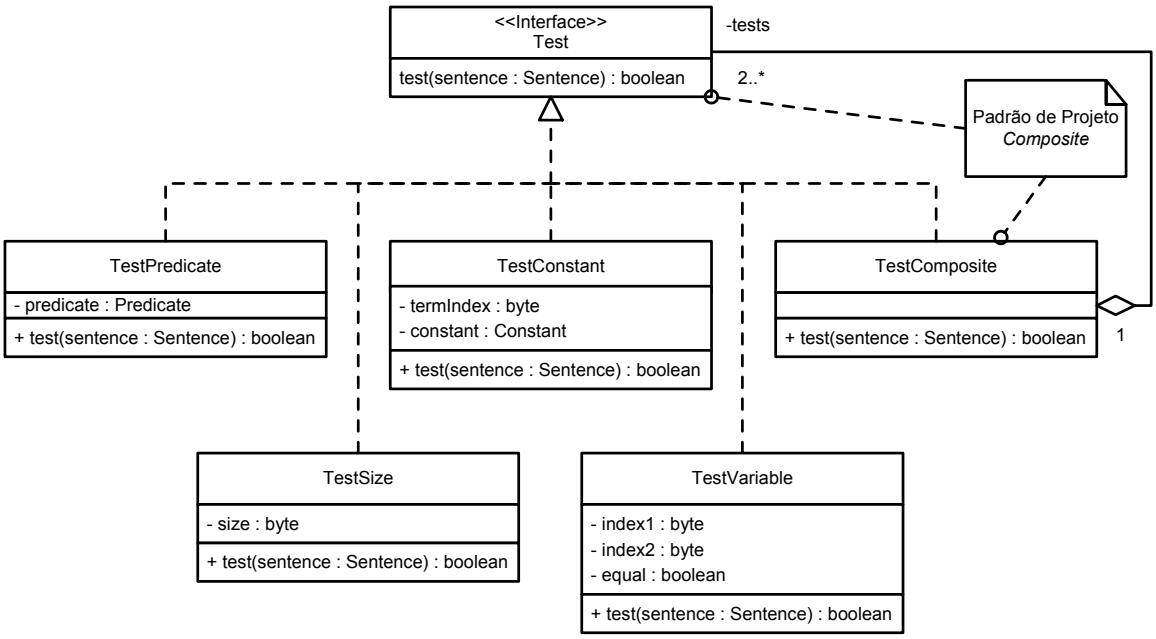


Figura 102: Diagrama de classes para o Subsistema *Rete*: hierarquia de `Test`

8.5 Subsistema Strategy

Para facilitar o desenvolvimento de aplicações baseadas no mecanismo de inferência concreto desenvolvido, inclui-se no *Subsistema Strategy* apenas uma implementação da estratégia de resolução de conflito *breadth-first*, que dispara as regras na ordem na qual elas foram ativadas, isto é, primeiro a que está no conjunto de conflito há mais tempo. Estratégias mais elaboradas, como as citadas em 4.5.8, podem ser incluídas, se necessário, pelo usuário.

8.6 Considerações Finais

Neste capítulo apresentou-se o desenvolvimento orientado a objetos dos subsistemas que compõem o *Mecanismo de Inferência com Encadeamento Progressivo Baseado em Regras*, incluído no *Framework de Inteligência* proposto.

O *Subsistema Interfaces de Representação de Conhecimento* contém um conjunto de interfaces independentes do tipo de mecanismo de inferência utilizado, isto é, encadeamento progressivo ou encadeamento regressivo. Durante a configuração do agente, a base de conhecimento que irá guiar sua execução, geralmente armazenada em um repositório persistente, deve ser convertida, se necessário, para essa representação. Por outro lado, ao ser transferida para o mecanismo de inferência, essa representação, na forma de objetos de conhecimento, é convertida para a representação interna do *casador* em uso. Para ampliar o poder de representação das *Interfaces de Representação de Conhecimento*, incluiu-se um *predicado reservado*, `EQUAL`, com a intenção de usá-lo em situações específicas, em conjunto com a negação por falha (~), para indicar a igualdade ou, principalmente, a desigualdade entre duas variáveis presentes em literais distintos. Como forma de reduzir a quantidade de código e de dados transportados pelo agente, propõe-se que a tradução da base de conhecimento armazenada em repositório para a forma de objetos de conhecimento seja realizada por um objeto específico, denominado, por exemplo, *iniciador*, que não precisa ser transportado, uma vez que a tradução em questão só ocorre na agência natal do agente.

O *Subsistema Mecanismo de Inferência* inclui as classes e interfaces necessárias à especialização do *Mecanismo Abstrato*, descrito em 7.4, em um *Mecanismo de Inferência com Encadeamento Progressivo Baseado em Regras*. Esse subsistema inclui, além de suas próprias classes, dois outros subsistemas: o *Subsistema Rete* e o *Subsistema Strategy*.

Com o intuito de reduzir a quantidade de código e de dados transportados pelo agente foram incluídas nas classes do *Subsistema Mecanismo de Inferência*, implementações minimalistas das interfaces das *Interfaces de Representação de Conhecimento*. Desse modo, tanto o mecanismo de inferência como seus componentes podem converte objetos de conhecimentos oriundos de implementações desenvolvidas pelo usuário do agente ou pelos responsáveis pelas agências em objetos dessas classes minimalistas. Isso evita que referências para implementações externas das *Interfaces de Representação do Conhecimento* fiquem armazenadas, por exemplo, na *Memória de Regras* do *Rete*. A manutenção de referências a tais implementações poderia fazer com que códigos e dados desnecessários fossem transportados. Dois exemplos de dados desnecessários que, normalmente, são acrescentados a implementações externas são seqüências de caracteres representando o nome da regra e algum comentário elucidativo sobre ela. No pior caso, a manutenção de referências a implementações externas

poderia até mesmo impedir a migração do agente, caso algum de seus componentes não pudessem ser serializados para o transporte.

O *Subsistema Rete* acrescenta ao *Framework de Inteligência* a implementação de um *casador de padrões* desenvolvido de acordo com o algoritmo *Rete*. O usuário do agente pode utilizar esse *casador* ou qualquer outro que desejar. Durante o desenvolvimento das classes e interfaces que compõem o *Subsistema Rete*, foi dada atenção especial aos aspectos de redução da quantidade de código e de dados a ser transportada. As funcionalidades do *casador* foram decompostas em dois grupos de classes e interfaces, um responsável pela compilação dos objetos de conhecimento na estrutura de nós e testes (`ReteCompiler`) e outro, que utiliza essa estrutura, responsável pela lógica do casamento (`Rete`).

O primeiro grupo de classes só é necessário no momento da configuração do mecanismo, durante o processo de instanciação do agente e, portanto, não precisa ser transportado, o que reduz a quantidade de código e de dados que deve migrar. No segundo grupo, todas as variáveis e métodos que não eram específicos da fase de casamento foram eliminados das classes que compõem a estrutura. Para as variáveis que permaneceram, sempre que possível, optou-se pelos tipos primitivos representados pelo menor número de bytes. Por exemplo, sempre que possível utilizou-se o tipo `byte` no lugar do tipo `integer`, visto que o primeiro é representado por apenas 1 byte enquanto a representação do segundo requer 4 bytes. Em relação ao código das classes do segundo grupo, preferiu-se sobrecarregar as assinaturas de seus construtores, passando-se vários parâmetros de entrada, evitando-se assim um processamento mais intenso no "corpo" desses métodos, o que também reduz o código dessas classes.

Como o número de nós, testes, predicados e constantes presentes na rede pode ser grande, deu-se ênfase aos padrões de projeto que evitam a replicação desnecessária de instâncias idênticas das classes que representam essas entidades, em especial ao padrão de projeto *Flyweight*.

No *Subsistema Strategy*, incluiu-se apenas uma implementação da estratégia de resolução de conflito *breadth-first*.

Capítulo 9

Resultados

9.1 Introdução

Neste capítulo, apresenta-se uma avaliação de desempenho do *framework* proposto, em termos da quantidade de código e de dados a serem transportados, e explicitam-se algumas decisões de implementação que afetam esses valores. É apresentado ainda um cenário de uso que evidencia as potencialidades do *framework* e o uso dos modificadores de sentenças introduzidos na sua concepção.

9.2 Implementação

O *framework* projetado nos capítulos anteriores foi implementado utilizando-se o *Java 2 Software Developer's Kit* (J2SDK) versão 1.4.2 da Sun Microsystems (SUN, 2004). Todas as classes e interfaces foram feitas totalmente serializáveis. Agentes móveis inteligentes foram criados com os sistemas *Aglets* (IBM, 2002a) e *Grasshopper* (IKV++, 2001c). Esses agentes foram colocados para migrar entre 4 microcomputadores do Laboratório de Tecnologia da Informação do Departamento de Engenharia Elétrica da FEB - UNESP. Essas máquinas eram compatíveis com o padrão IBM PC, estavam equipadas com o sistema operacional Windows 2000 da Microsoft e interconectadas por uma rede local Ethernet de 100Mbps. Foram realizados testes para se verificar a correta mobilidade de todas as classes e interfaces do *framework* e os custos de migração do código e do estado de dados do mecanismo de inferência. Os resultados obtidos, relativos a esses custos, são descritos na próxima seção.

Alguns comentários, relativos à utilização do *framework*, também são importantes. O desenvolvedor do agente pode implementar seu próprio algoritmo de casamento (`Matcher`) ou utilizar o algoritmo oferecido pelo *framework* (`Rete`). Da mesma forma, ele também pode implementar sua própria estratégia de resolução de conflitos (`Strategy`) ou utilizar uma estratégia fornecida. Atualmente, somente uma implementação de `Strategy` foi incluída no pacote, aquela que seleciona a ativação mais antiga do conjunto de conflito (`LessRecentStrategy`).

O desenvolvedor do agente também deve escrever código para: (1) inicializar o mecanismo de inferência durante o processo de criação do agente; (2) iniciar e terminar a execução do mecanismo de inferência durante a execução do agente; (3) executar os procedimentos de sensoriamento e atuação exclusivos do agente e (4) encaminhar, quando necessário, as solicitações do mecanismo de inferência ao serviço de inteligência oferecido pela agência hospedeira. Porém, uma vez escrito, todo esse código pode ser completamente reutilizado em outros agentes que utilizem o *framework*. Como os sensores e os atuadores utilizados são os únicos procedimentos que poderão variar de agente para agente, em função das regras presentes na base de conhecimento, o ideal, para facilitar a reutilização do código, é que esse seja construído de forma a carregar dinamicamente as classes que implementam tais métodos.

Por sua vez, o desenvolvedor responsável por uma agência deve implementar o serviço de inteligência de sua agência e instalá-lo adequadamente, por exemplo, na forma de um agente estático.

9.3 Avaliação do Desempenho

Dois aspectos são importantes para a avaliação de desempenho do *framework* desenvolvido: a quantidade de código e a quantidade de dados (estado de dados) a serem transportados pelo agente. As subseções seguintes discutem esses aspectos.

9.3.1 Avaliação da Quantidade de Código

As tabelas a seguir mostram os tamanhos, em bytes, das classes que compõem os diversos subsistemas do *framework*.

Tabela 7: Custo do código do subsistema *Mecanismo Abstrato*

Componente	Tamanho (bytes)
`Action.class` (interface)	299
`Component.class` (interface)	309
`Eclient.class` (interface)	437
`Engine.class`	5.911
`Modifiers.class` (interface)	256
`Percept.class` (interface)	257
`PerceptQueue.class`	895
`Sentence.class` (interface)	153
Subtotal	**8.517**
`KB.class` (interface)	208
Total	**8.725**

O componente `KB.class` foi contabilizado à parte pois só é necessário na fase de inicialização do mecanismo e, portanto, não é transportado durante a migração do agente.

Tabela 8: Custo do código do subsistema *Interfaces do Serviço de Inteligência*

Componente	Tamanho (bytes)
`ISClient.class` (interface)	141
`ISServer.class` (interface)	470
Total	**611**

Tabela 9: Custo do código do subsistema *Interfaces de Representação de Conhecimento*

Componente	Tamanho (bytes)
`Constant.class` (interface)	374
`Literal.class` (interface)	336
`Predicate.class` (interface)	275
`Sentence.class` (interface)	214
`Term.class` (interface)	149
`Variable.class` (interface)	186
Subtotal	**1.534**
`Rule.class` (interface)	245
Total	**1.779**

O componente `Rule.class` foi contabilizado à parte pois só é necessário na fase de inicialização do mecanismo e, portanto, não é transportado durante a migração do agente.

Tabela 10: Custo do código do subsistema *Mecanismo de Inferência*

Componente	Tamanho (bytes)
`Activation.class`	724
`ConflictSet.class`	1.219
`Action.class`	567
`Constant.class`	1653
`Literal.class`	1.531
`Predicate.class`	1.343
`Variable.class`	704
`FCEngine.class`	4.320
`Matcher.class` (interface)	270
`Strategy.class` (interface)	272
Total	**12.603**

Tabela 11: Custo do código do subsistema *Rete*

Componente	Tamanho (bytes)
`Atom.class`	2.850
`Conjunct.class`	386
`Conjunction.class`	1.948
`ConstantFactory.class`	1.671
`PredicateFactory.class`	2.187
`Node.class`	1.405
`Node1.class`	1.448
`Node2.class`	5.170
`LeftMemory.class`	2.611
`RightMemory.class`	2.441
`Rete.class`	4.157
`SensorNode.class`	3.378
`Substitution.class`	276
`Terminal.class`	4.120
`Test.class` (interface)	186
`TestComposite.class`	461
`TestConstant.class`	1.258
`TestPredicate.class`	1.135
`TestVariable.class`	1.782
Subtotal	**38.870**
`ReteCompiler.class`	6.658
`NodeFactory.class`	5.914
`TestFactory.class`	3.815
`VariableFactory.class`	704
Total	**55.961**

Os componentes `ReteCompiler.class`, `NodeFactory.class`, `PredicateFactory.class`, `TestFactory.class` e `VariableFactory.class` foram contabilizados à parte pois só são necessários na fase de inicialização do mecanismo, quando da compilação da base de conhecimento na estrutura de nós do *Rete* e, portanto, não são transportados durante a migração do agente.

Tabela 12: Custo do código do subsistema *Strategy*

Componente	Tamanho (bytes)
`LessRecentStrategy.class`	861
Total	**861**

Em resumo, com base nas tabelas anteriores, a quantidade de código que precisa ser transportada é de apenas 62.996 bytes. Se já houver um SI instalado nas agências hospedeiras, os subsistemas *Interfaces do Serviço de Inteligência* e *Interfaces de Representação de Conhecimento* já estarão instalados, o que reduz a quantidade de código para 60.851 bytes. Caso o código seja acondicionado de forma compactada num arquivo JAR (*Java Archive Resource*), esse valor cai para 38.607 bytes, como mostra a Tabela 13. Note-se, no entanto, que a maioria das plataformas de agentes móveis possui algum mecanismo de *cache*, de modo que, caso o agente retorne a uma agência já visitada, as classes já estarão disponíveis, não havendo a necessidade de transportá-las.

Tabela 13: Tamanho do código transportado na forma de arquivo JAR

Subsistema	Não-compactado (bytes)	Compactado (bytes)
MECANISMO ABSTRATO	8.517	4.417
Interfaces do Serviço de Inteligência	611	328
Interfaces de Representação de Conhecimento	1.534	1.071
Mecanismo de Inferência	12.603	6.095
Rete	38.870	16.780
Strategy	861	359
Manifest.mf[1]		41
Informações sobre a compactação		9.516
Total	**62.996**	**38.607**

[1] O arquivo `manifest.mf` é um arquivo especial que contém informações sobre os arquivos `*.class` encapsulados no arquivo JAR.

9.3.2 Avaliação da Quantidade de Dados

A quantidade de dados transportada depende do estado de dados dos objetos criados a partir das classes dos diversos subsistemas. Em função da implementação realizada, as únicas classes que produzirão objetos com estado de dados são as classes `Engine`, do *Mecanismo Abstrato*; `FCEngine`, `Predicate` e `Constant`, do *Mecanismo de Inferência;* e todas as classes do subsistema *Rete* que tomam parte da fase de execução do mecanismo. A quantidade de dados dessas últimas é composta pelos dados da lógica de controle do casamento e pelos dados presentes nas suas memórias de regras e de trabalho, que são, por sua vez, dependentes do número de regras e de fatos presentes na base de conhecimento do agente. Portanto, a quantidade total de dados a ser transportada é dependente da aplicação pretendida. No entanto, é possível avaliar os custos unitários de serialização dos componentes que constituem essas memórias e utilizá-los na avaliação do custo de serialização de uma base de conhecimento qualquer. Para isso, foram realizadas avaliações dos custos de serialização de uma instância dos objetos em questão.

Os custos de serialização dos dados dos diversos tipos de nós e de testes que podem estar presentes numa estrutura do `Rete` representam os custos unitários dos componentes que constituem a *memória de regras* do `Rete`. Por sua vez, o custo de serialização da *memória de trabalho* do `Rete` é composto pelos custos individuais dos diversos objetos das classes `Atom` e `Conjunction`, presentes nas memórias esquerdas e direitas dos diversos nós-de-duas-entradas existentes numa rede. Os objetos da classe `Atom` presentes nesses nós representam fatos atômicos, enquanto os objetos da classe `Conjunction` representam fatos compostos, relacionados pelo conectivo lógico **E**.

O mecanismo de serialização e deserialização da linguagem Java chama, respectivamente, os métodos `writeObject(...)` e `readObject(...)` de cada objeto a ser serializado ou deserializado. O método `writeObject(...)` transforma o estado de dados do objeto em um fluxo de bits, de acordo com o protocolo de serialização (SUN, 2004) descrito no Apêndice B. O método `readObject(...)` realiza a operação inversa. Embora existam implementações *default* para esses métodos, eles podem ser reescritos pelo desenvolvedor das classes.

Como pode ser observado do Apêndice B, basicamente, para cada objeto serializado, três tipos de informações são incluídas no fluxo de bits: a descrição da classe a que o objeto pertence, a descrição de todos os seus campos (tipo e nome) e os valores desses campos (estado de dados). As duas primeiras informações são representadas uma única vez no fluxo, independentemente do número de objetos da mesma classe a serem serializados. Após a serialização do primeiro objeto, todas as demais fazem referência à descrição já incluída. A informação relativa aos valores dos campos deve estar presente para cada uma das instâncias.

Na implementação do *framework*, todas as classes cujos estados de dados estariam presentes no fluxo serial de bits tiveram seus métodos `writeObject(...)` e `readObject(...)` reescritos, de modo a excluir do fluxo as descrições das classes contêineres utilizadas, por exemplo, nas memórias dos nós-de-duas-entradas e as descrições dos campos. Isso reduziu a quantidade de dados transportados e facilitou a avaliação do custo dessas informações para as classes, visto que a ordem de serialização foi definida pelo desenvolvedor. As tabelas a seguir resumem os custos encontrados.

Tabela 14: Custo dos dados do subsistema *Mecanismo Abstrato*

Componente	Descrição da Classe (bytes)	Dados (bytes)
`Engine`	42	0

Tabela 15: Custo dos dados do subsistema *Mecanismo de Inferência*

Componente	Descrição da Classe (bytes)	Dados (bytes)
`FCEngine`	64	26
`Predicate`	70	14+ L
`Constant` (do tipo string)	69	13+ L
`Constant` (do tipo real)	69	18
`Constant` (do tipo inteiro)	69	18

L: número de letras do predicado ou da constante do tipo string. Letras acentuadas contam como 2 bytes

Tabela 16: Custo dos dados do subsistema *Rete* – controle do casamento

Componente	Descrição da Classe (bytes)	Dados (bytes)
`Rete`	167	38

Tabela 17: Custo dos dados do subsistema *Rete* – memória de regras

Componente	Descrição da Classe (bytes)	Dados (bytes)
`Node`	48	5 + 5S
`Node1`	53	9 + 5T + 5S
`Node2`	53	24 + 5T + 5S
`SensorNode`	58	$24 + 2T_M + 3C - V + 5S$
`Terminal`	56	$24 + 3T_M + 2C$
`TestPredicate`	57	7
`TestConstant`	56	10
`TestVariable`	56	7

S: número de nós sucessores
T: número de testes executados pelo nó
T_M: número de termos do predicado sensor ou terminal
C: número de constantes do perdicado sensor ou terminal
V: número de variáveis livres do predicado sensor, ie, que devem ser valoradas pelo sensor

Tabela 18: Custo dos dados do subsistema *Rete* – memória de trabalho

Componente	Descrição da Classe (bytes)	Dados (bytes)
`LeftMemory + RightMemory`	0	$5S_T$
`Atom`	104	$11 + 5T_M$
`Conjunction`	59	12

T_M: número de termos do átomo
S_T: número total de sentenças armazenadas pelos nós-de-duas-entradas (à esquerda + à direita)

Tabela 19: Custo dos dados do subsistema *Strategy*

Componente	Descrição da Classe (bytes)	Dados (bytes)
`LessRecentStrategy`	66	1

A título de exemplo de avaliação do custo de serialização, seja o seguinte conjunto de regras, já apresentado em 4.6:

Regra 1: `A(x,y) ∧ B(y,x) ∧ C(Luís,x) → adicionar D(x,y)`

Regra 2: `A(x,y) ∧ B(y,x) ∧ D(x,x) → remover E(x,y)`

Para esse conjunto, a Figura 103 mostra a rede de nós criada por `ReteCompiler` e o conteúdo das memórias dos nós-de-duas-entradas após a inserção dos fatos `A(Paulo,Pedro)`, `A(José,João)`, `B(João,Pedro)`, `B(Pedro,Paulo)`, `C(Luís,Pedro)`, `C(Luís,Paulo)` e `C(Luís,José)`.

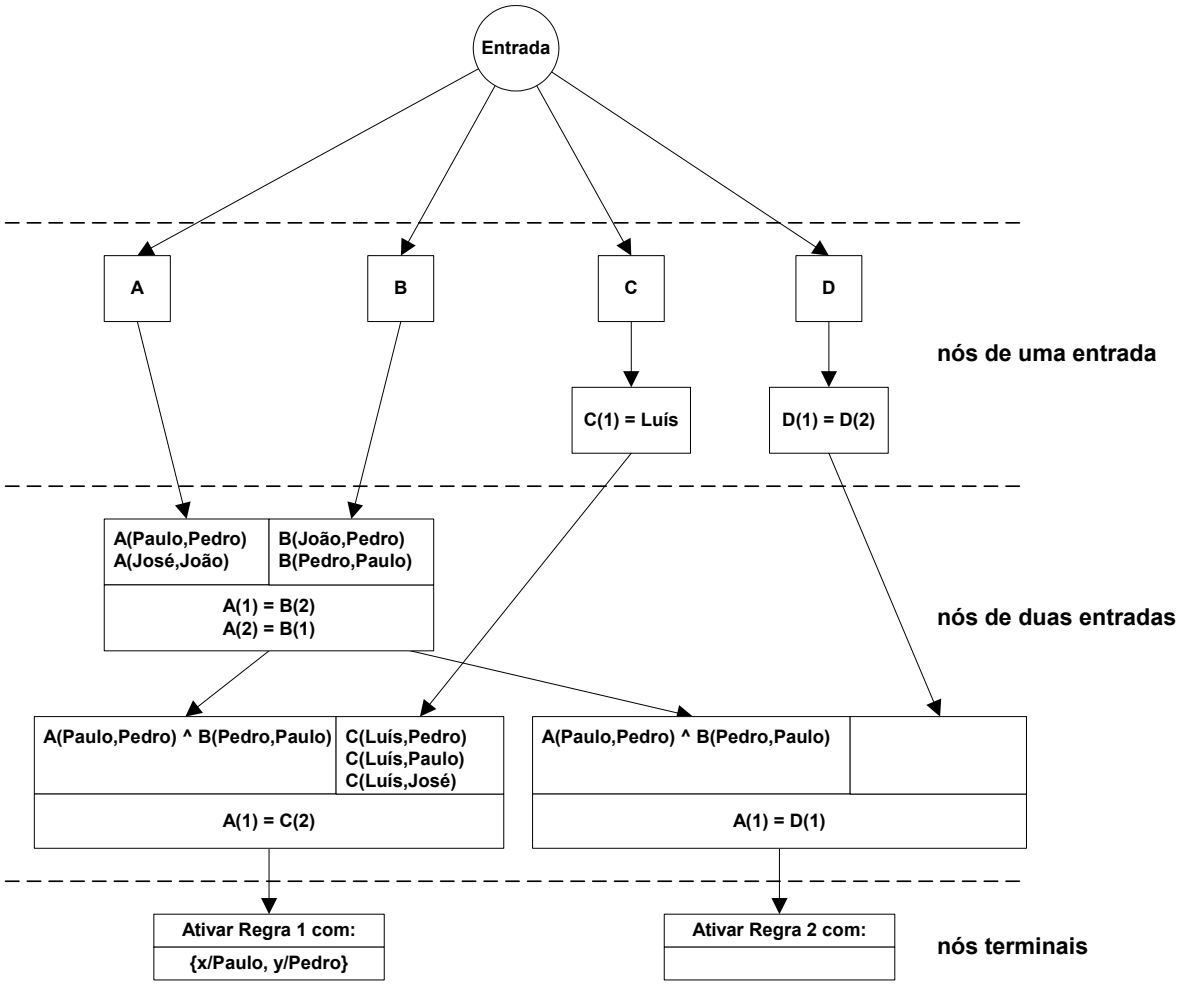


Figura 103: Exemplo de uma rede

Para um mecanismo de inferência com essa rede, os custos de serialização dos dados envolvidos são os mostrados na Tabela 20.

Tabela 20: Custo dos dados do mecanismo de inferência com a rede exemplo

Componente		Custo de Serialização (bytes)	Total
Engine		42	42
FCEgine		64 + 26	90
LessRecentStrategy		66 + 1	67
Rete		167 + 38	205
Predicate	A	70 + 14 + 1	85
	B	14 + 1	15
	C	14 + 1	15
	D	14 + 1	15
	E	14 + 1	15
Constant	João	69 + 13 + 5	87
	José	13 + 5	18
	Luís	13 + 5	18
	Paulo	13 + 5	18
	Pedro	13 + 5	18
Memória de Regras			
Node	Entrada	48 + 5 + 5x4	73
Node1	A	53 + 9 + 5x1 + 5x1	72
	B	9 + 5x1 + 5x1	19
	C	9 + 5x1 + 5x1	19
	D	9 + 5x1 + 5x1	19
	C(1) = Luís	9 + 5x1 + 5x1	19
	D(1) = D(2)	9 + 5x1 + 5x1	19
Node2	A ∧ B	53 + 24 + 5x2 + 5x2	97
	A ∧ B ∧ C	24 + 5x1 + 5x1	34
	A ∧ B ∧ D	24 + 5x1 + 5x1	34
SensorNode	-	0	0
Terminal	D	56 + 21 + 3x2 + 5x0	83
	E	21 + 3x2 + 5x0	27
TestPredicate	A	57 + 7	64
	B	7	7
	C	7	7
	D	7	7
TestConstant	C(1) = Luís	56 + 10	66
TestVariable	D(1) = D(2)	56 + 7	63
	A(1) = B(2)	7	7
	A(2) = B(1)	7	7
	A(1) = C(2)	7	7
	A(1) = D(1)	7	7
Memória de Trabalho			
LeftMemory + RightMemory		5x9	45
Atom	A(Paulo,Pedro)	104 + 11 + 5x2	125
	A(José,João)	11 + 5x2	21
	B(João,Pedro)	11 + 5x2	21
	B(Pedro,Paulo)	11 + 5x2	21
	C(Luís,Pedro)	11 + 5x2	21
	C(Luís,Paulo)	11 + 5x2	21
	C(Luís,José)	11 + 5x2	21
Conjunction	A(Paulo,Pedro) ∧ B(Pedro,Paulo)	59 + 12	71
Custo Total da Serialização			**1832**

A redução de código e de dados, obtidas com a abordagem utilizada, fica clara quando se compara o sistema desenvolvido com o JESS, conforme apresentado na Tabela 21.

Tabela 21: Comparação entre o *framework* desenvolvido e o JESS (em bytes)

Subsistema	*Framework*	JESS
Código (Engine + Strategy + Rete)	62.996	452.758
Dados (Engine + Strategy + Rete)	404	19.009
Dados Memória de Regras (do exemplo)	989	3.991
Dados Memória de Trabalho (do exemplo)	439	1.527
Total	**64.828**	**477.285**

9.4 Cenário de Utilização

Esta seção descreve o cenário de teste criado para evidenciar as potencialidades do *framework* desenvolvido e o uso dos modificadores introduzidos durante sua concepção.

9.4.1 Descrição

A plataforma de suporte a agentes móveis *Aglets* (IBM, 2002b) foi instalada em 4 microcomputadores, como mostrado na Figura 104. Essas máquinas eram compatíveis com o padrão IBM PC, estavam equipadas com o sistema operacional Windows 2000 da Microsoft e interconectadas por uma rede local Ethernet de 100Mbps. Em cada um dos microcomputadores que compunham o *Ambiente Distribuído de Agentes* (ADA) foi armazenada uma página XML (W3C, 2004) contendo informações sobre publicações de alguns pesquisadores (`c:\public\library.xml`). Um exemplo do conteúdo dessa página é mostrado na Figura 105 e o DTD (*Document Type Definitions*) (W3C, 2004) utilizado em sua construção, na Figura 106.

Um agente móvel, munido do mecanismo de inferência com encadeamento progressivo desenvolvido no Capítulo 8, foi criado para executar a tarefa de percorrer as agências do ADA, buscando pelas publicações realizadas durante o ano de 2003. Para possibilitar a execução dessa tarefa, também foram desenvolvidos sensores e atuadores específicos. Em espe-

cial, foi criado um sensor capaz de analisar as informações contidas no arquivo `library.xml`. A base de conhecimento empregada para essa tarefa está mostrada na Figura 107, na sua forma `infix`. Essa base foi armazenada, na agência natal do agente, num arquivo denominado `rulebase.isb`.

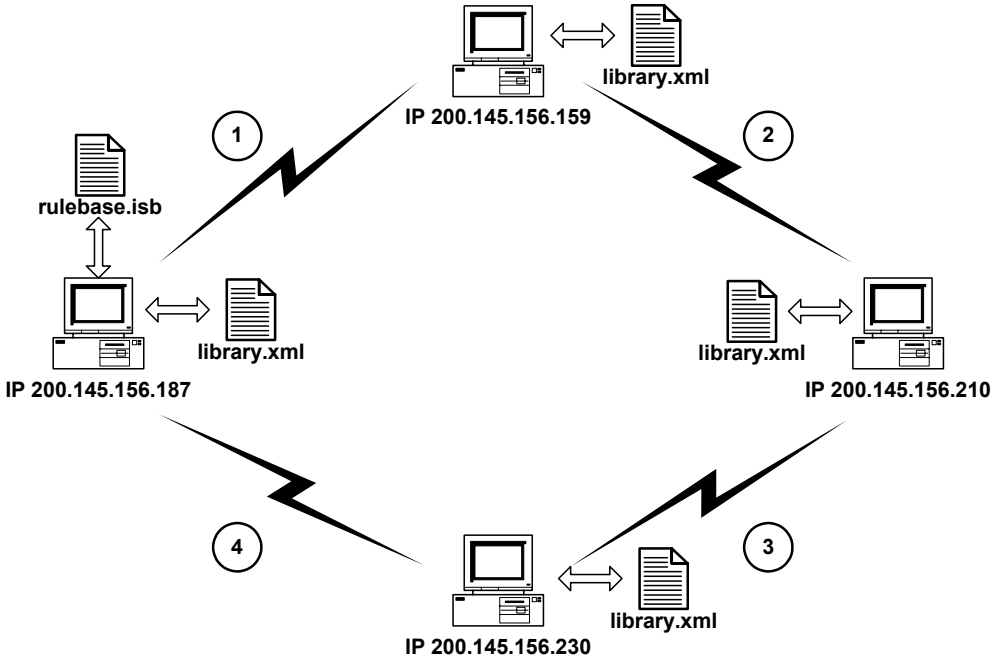


Figura 104: Configuração utilizada no cenário de teste

Basicamente, os comentários presentes incluídos na Figura 107 são suficientes para o entendimento de cada uma das regras. No entanto, merece nota a forma de identificar as agências instaladas em cada um dos microcomputadores do ambiente. Essa identificação aparece, especialmente, nas regras 9, 10, 11 e 12, que definem o itinerário do agente, e utiliza o protocolo de transferência de agentes "atp" (*Agent Transfer Protocol*). Esse protocolo é específico da plataforma *Aglets*; caso fosse utilizada outra plataforma, esse protocolo deveria ser alterado adequadamente. No lugar do IP das máquinas em questão, também se poderia usar o nome completo de cada uma delas; por exemplo, o IP 200.145.156.187 poderia ser substituído por `davinci.dee.bauru.unesp.br`.

As descrições e justificativas dos predicados marcados como eventos, sensores, atuadores e transientes na Figura 107 serão realizadas no item seguinte.

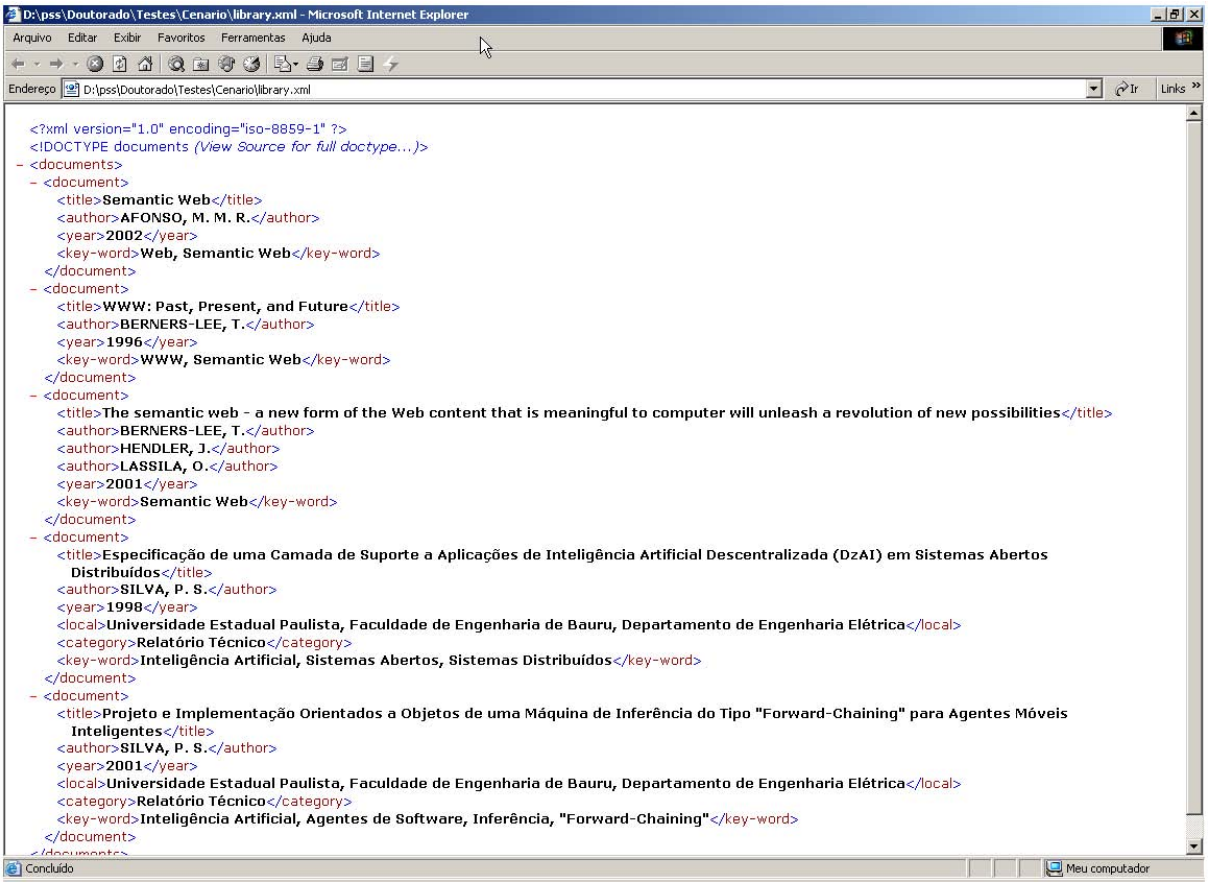


Figura 105: Exemplo de conteúdo da página XML com informações de publicações

```
<!--Arquivo DTD para o arquivo library.xml-->
<!ELEMENT documents (document*)>
<!ELEMENT document (title, author*, year, local*, editor*, number*, volume*, category*, key-word*, abstract*)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (#PCDATA)>
<!ELEMENT year (#PCDATA)>
<!ELEMENT local (#PCDATA)>
<!ELEMENT editor (#PCDATA)>
<!ELEMENT number (#PCDATA)>
<!ELEMENT volume (#PCDATA)>
<!ELEMENT category (#PCDATA)>
<!ELEMENT key-word (#PCDATA)>
<!ELEMENT abstract (#PCDATA)>
```

Figura 106: DTD utilizado pelo arquivo da Figura 105 (`library.dtd`)

Base de conhecimento para o cenário de teste

Regra 1: Mostra janela de diálogo do agente assim que o mecanismo é iniciado.
startingEngine() -> display()

Regra 2: Se a busca terminou imprime o resultado, se houver.
startingEngine() & finished() & document(x,y,z) -> save("c:/public/resultados.txt",x,y,z)

Regra 3: Se a busca terminou, pára o agente ao chegar na agência natal.
startingEngine() & finished() -> stop()

Regra 4: Ao iniciar o agente em uma agencia, obter o endereço da agencia.
startingEngine() & getAddress(x) & ~finished() -> atAgency(x)

Regra 5: Se o arquivo library.xml existe, abra-o para pesquisa.
atAgency(x) & existFile("c:/public/ library.xml") -> openFile("c:/public/library.xml")

Regra 6: Se o arquivo library.xml não existe, termina o processamento na agência hospedeira.
atAgency(x) & ~existFile("c:/public/library.xml") -> done(x)

Regra 7: Procura por documentos cujo ano de publicação é 2003.
atAgency(x) & fileOpenned("c:/public/library.xml") & find(author, title, "2003") -> document(author, title, "2003") & done(x)

Regra 8: Se não encontrar documentos publicado 2003, termina a busca na agência corrente.
atAgency(x) & fileOpenned("c:/public/library.xml") & ~find(author, title, "2003") -> done(x)

Regra 9: Ao terminar na Agência 1, vá para Agência 2.
done("atp://200.145.156.187") -> move("atp://200.145.156.159")

Regra 10: Ao terminar na Agência 2, vá para a Agencia 3.
done("atp://200.145.156.159") -> move("atp://200.145.156.210")

Regra 11: Ao terminar na Agência 3, vá para a Agencia 4.
done("atp://200.145.156.210") -> move("atp://200.145.156.230")

Regra 12: Ao terminar na Agência 4, vá para a agencia natal.
done("atp://200.145.156.230") -> move("atp://200.145.156.187")

Regra 13: Se a execução está sendo encerrada na última agência, indica que a busca terminou.
stoppingEngine() & done("atp://200.145.156.230")-> finished()

EVENT: startingEngine
EVENT: stoppingEngine
SENSOR: existFile: c:/pss/documentos/doutorado/testes/adapters/page.util.adapters.FileSystem#existFile
SENSOR: find: c:/pss/documentos/doutorado/testes/adapters/page.util.adapters.XMLParser#find
SENSOR: getAddress: c:/pss/documentos/doutorado/testes/adapters/page.util.adapters.Transport#getAddress
EFFECTOR: move: c:/pss/documentos/doutorado/testes/adapters/page.util.adapters.Transport#move
EFFECTOR: stop: c:/pss/documentos/doutorado/testes/adapters/page.util.adapters.Transport#stop
EFFECTOR: openFile: c:/pss/documentos/doutorado/testes/adapters/page.util.adapters.XMLParser#openFile
EFFECTOR: save: c:/pss/documentos/doutorado/testes/adapters/page.util.adapters.FileSystem#save
EFFECTOR: display: c:/pss/documentos/doutorado/testes/adapters/page.util.adapters.reasoning.Display#show
TRANSIENT: atAgency
TRANSIENT: fileOpenned
TRANSIENT: done

Figura 107: Base de conhecimento expressa na forma `infix` (`rulebase.isb`)

9.4.2 Execução

O agente móvel, ao ser iniciado em sua agência natal (microcomputador 1, IP 200.145.156.187) lê a base de conhecimento armazenada em `rulebase.isb`, traduz as regras e fatos expressos na forma `infix` para objetos de representação de conhecimento, cujas interfaces foram definidas em 8.2, e coloca sua máquina de inferência com encadeamento progressivo em execução. Então, seguindo as orientações codificadas nas regras mostradas na Figura 107, o agente percorre as agências que compõem o ADA, buscando, nos arquivos `c:\public\library.xml` de cada uma delas pelas publicações realizadas durante o ano de 2003.

A Figura 108 mostra a janela de controle da agência *Aglet* instalada na máquina `davinci` (200.145.146.187), porta 4434, com o agente `Finder` instanciado mas ainda não executando a sua tarefa.

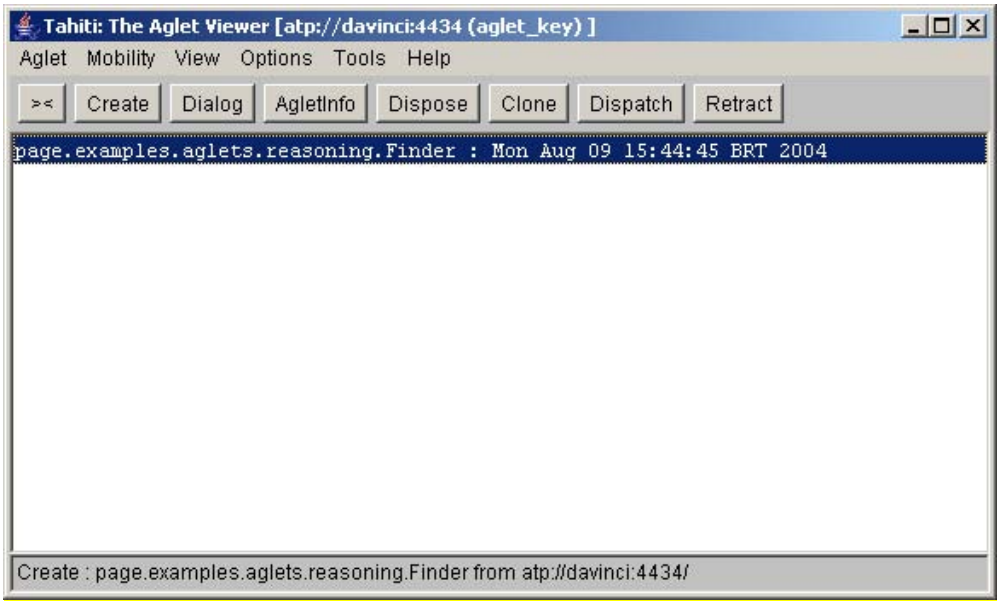


Figura 108: Janela de controle da agência *Aglets* instalada em 200.145.156.187

As figuras a seguir mostram as janelas de diálogo do agente `Finder` em cada uma das agências visitadas.

Ao ser iniciado na sua agência natal (200.145.156.187), o agente abre a janela de diálogo mostrada na Figura 109 assim que o mecanismo de inferência é colocado em execução

(Regra 1). O evento `startingEngine` é gerado pelo código herdado do mecanismo abstrato, toda vez que um mecanismo concreto é iniciado pela invocação da operação `start()`.

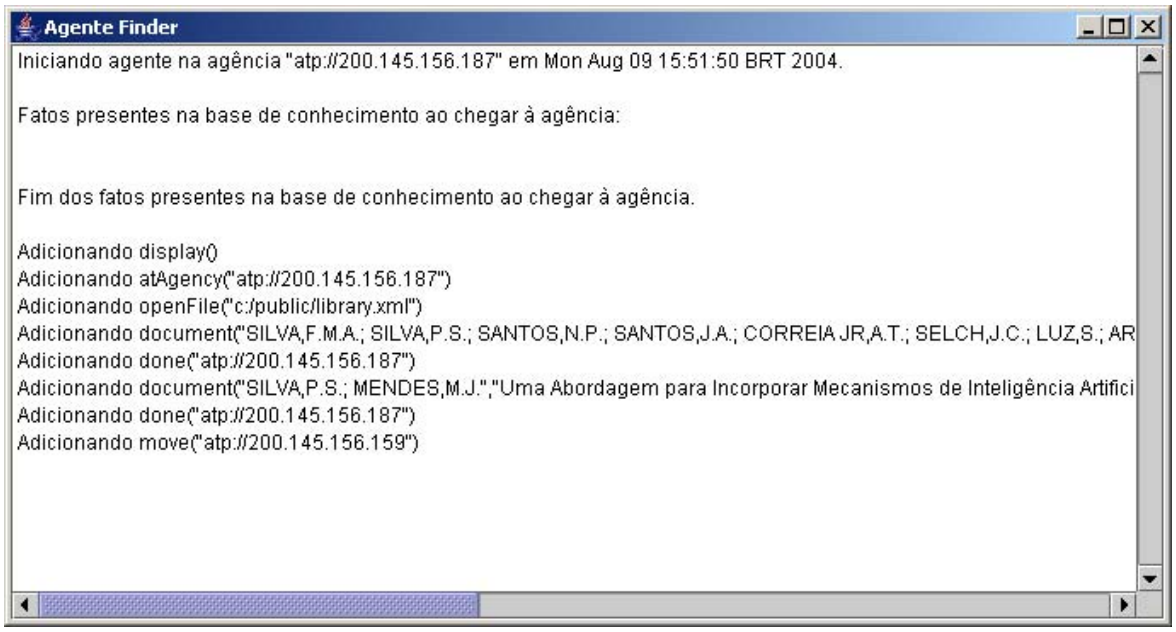


Figura 109: Janela de diálogo mostrada na agência 200.145.156.187

De acordo com a Figura 109, o agente móvel inicia a sua execução na agência natal (200.145.156.187) sem nenhum fato em sua base de conhecimentos, isto é, a base possui somente as regras. A seguir, as regras 4, 5 e 7 são disparadas, essa última duas vezes, pois foram encontradas as seguintes publicações com data de 2003:

- `Document("SILVA,F.M.A.; SILVA,P.S.; SANTOS,N.P.; SANTOS,J.A.; CORREIA JR,A.T.; SELCH,J.C.; LUZ,S.; ARGOLLO,S.L.", "Executing Workflows with Mobile Agents from UML Specifications. In: V WORKSHOP DE COMUNICAÇÃO SEM FIO E COMPUTAÇÃO MÓVEL – CSF2003","2003")` e
- `Document("SILVA,P.S.; MENDES,M.J.", "Uma Abordagem para Incorporar Mecanismos de Inteligência Artificial a Agentes Móveis. In: 21. SIMPÓSIO BRASILEIRO DE REDES DE COMPUTADORES.", "2003")`

As regras 1 e 4 são ativadas no primeiro ciclo de inferência do episódio de inferência iniciado pela ocorrência do evento `startingEngine`. A regra 4 é ativada porque o fato

`finished`, que indica que a tarefa foi concluída, ainda não está presente na memória de trabalho. Pelo mesmo motivo as regras 2 e 3 não ativadas. A ativação da regra 1 é colocada no conjunto de conflitos antes da ativação da regra 4, porque essa última aparece depois daquela na base de conhecimento. Como a estratégia de resolução de conflitos utilizada (*breath-first*) dispara as regras na ordem em que elas foram ativadas, a regra 1 é disparada antes da regra 4. Ao ser executada, a regra 4 adiciona o fato `atAgency` à memória de trabalho, iniciando o segundo ciclo de inferência do episódio de inferência originado por `startingEngine`. Nesse segundo ciclo, somente a regra 5 é ativada e, portanto, disparada, pois o sensor `existFile("c:/public/library.xml")` informa que o arquivo `library.xml` está presente no diretório especificado. Como o atuador `openFile`, presente no conseqüente da regra 5, tem sucesso ao tentar abrir o arquivo `library.xml`, ele gera uma percepção em cujo conjunto de fatos existe o fato `fileOpenned("c:/public/library.xml")`. A adição desse fato à memória de trabalho produz uma ativação da regra 7 para cada instância de `document(author,title,"2003")` retornada por `find(author,title,"2003")`. Assim que a primeira ativação da regra 7 é disparada, o fato `done("atp://200.145.156.187")` é adicionado à memória de trabalho, produzindo a ativação da regra 9. Porém, devido à estratégia de resolução de conflitos escolhida, a ativação da regra 9 somente será executada após a execução de todas as demais ativações da regra 7.

Finalmente, quando a regra 9 é disparada, o agente é enviado para a próxima agência (200.145.156.159). Como parte das tarefas realizadas durante a execução do atuador associado ao predicado `move`, o mecanismo é interrompido, pela chamada da operação `stop()`, gerando o evento `stoppingEngine`. Como o fato `done("atp://200.145.156.230")` não está presente na memória de trabalho, a regra 13 não é ativada.

Ao chegar à agência 200.145.156.159, o agente é colocado em execução e uma nova janela de diálogo é aberta, como mostrado na Figura 110.

De acordo com a Figura 110, o agente inicia a sua execução na agência 200.145.156.159 somente com os dois fatos relativos ao predicado `document` em sua memória de trabalho. Os outros fatos, adicionados durante a sua estadia na agência 200.145.156.187, isto é, `display()`, `atAgency("atp://200.145.156.187")`, `openFile("c:/public/library.xml")`, `fileOpenned("c:/public/library.xml")`, `done("atp://200.145.156.187")` e `move("atp://200.145.156.159")`, não estão mais

presentes, pois foram eliminados antes da migração do agente. Isso ocorre porque os predicados `atAgency` , `fileOpenned` e `done` foram marcados com o modificador `TRANSIENT` enquanto `display`, `move` e `openFile`, com o modificador `EFFECTOR` (ver Figura 107). Fatos presentes na base de conhecimento, cujos predicados estão marcados com esses modificadores, bem como com os modificadores `EVENT` e `SENSOR`, nunca são transportados de uma agência para outra.

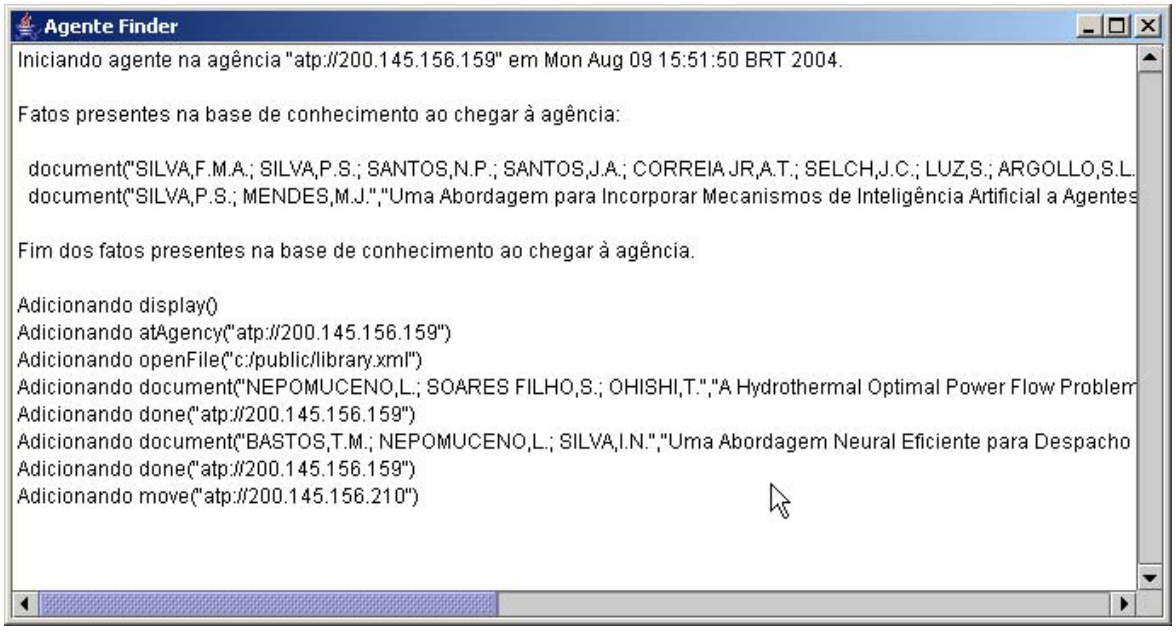


Figura 110: Janela de diálogo mostrada na agência 200.145.156.159

Quando o mecanismo de inferência é iniciado na agência 200.145.156.159, as regras 1, 4, 5 e 7 são disparadas, essa última duas vezes, pois foram, novamente, encontradas mais duas publicações com data de 2003:

- `Document("NEPOMUCENO,L.; SOARES FILHO,S.; OHISHI,T.", "A Hydrothermal Optimal Power Flow Problem Solved by Lagrangian Relaxation Aprooach. International Journal of Computer Research", "2003")` e
- `Document("BASTOS,T.M.; NEPOMUCENO,L.; SILVA,I.N.", "Uma Abordagem Neural Eficiente para Despacho Econômico com Representação do Sistema de Transmissão. Revista de Iniciação Científica, São Carlos.", "2003"`

Finalmente, quando a regra 10 é disparada, o agente é enviado para a próxima agência (200.145.156.210). Ao chegar à agência 200.145.156.210, o agente é colocado em execução e uma nova janela de diálogo é aberta, como mostrado na Figura 111.

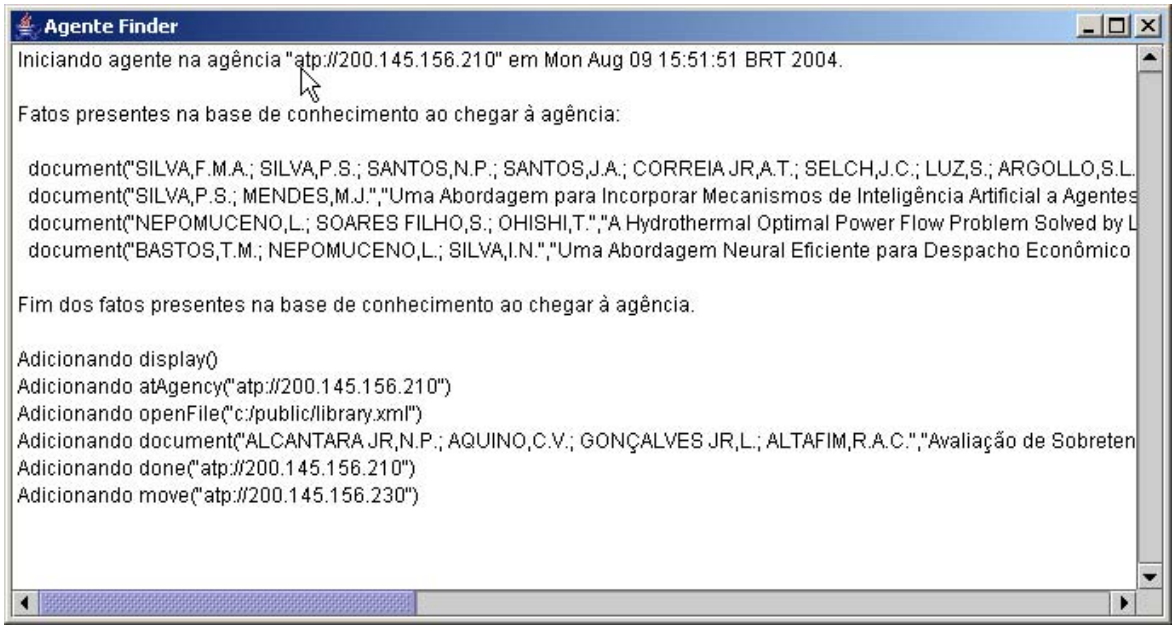


Figura 111: Janela de diálogo mostrada na agência 200.145.156.210

De acordo com a Figura 111, o agente inicia a sua execução, novamente, somente com os fatos relativos ao predicado `document` em sua memória de trabalho. Os demais fatos adicionados durante a sua estadia na agência 200.145.156.159 não estão mais presentes, pois foram eliminados antes da migração do agente, uma vez que seus predicados foram marcados com um dos modificadores `EVENT`, `TRANSIENT`, `EFFECTOR` ou `SENSOR`.

Quando o mecanismo de inferência é iniciado, as regras 1, 4, 5 e 7 são disparadas, essa última uma única vez, pois, agora, foi encontrada somente uma publicação com data de 2003:

- `Document("ALCANTARA JR,N.P.; AQUINO,C.V.; GONÇALVES JR,L.; ALTAFIM,R.A.C.", "Avaliação de Sobretensões em Equipamentos Elétricos e Linhas de Transmissão e Coordenação do Isolamento por Simulações Computacionais. In: SEMINÁRIO NACIONAL DE PRODUÇÃO E TRANSMISSÃO DE ENERGIA ELÉTRICA", "2003")`

Finalmente, quando a regra 11 é disparada, o agente é enviado para a próxima agência (200.145.156.230). Ao chegar à agência 200.145.156.230, o agente é colocado em execução e uma nova janela de diálogo é aberta, como mostrado na Figura 112.

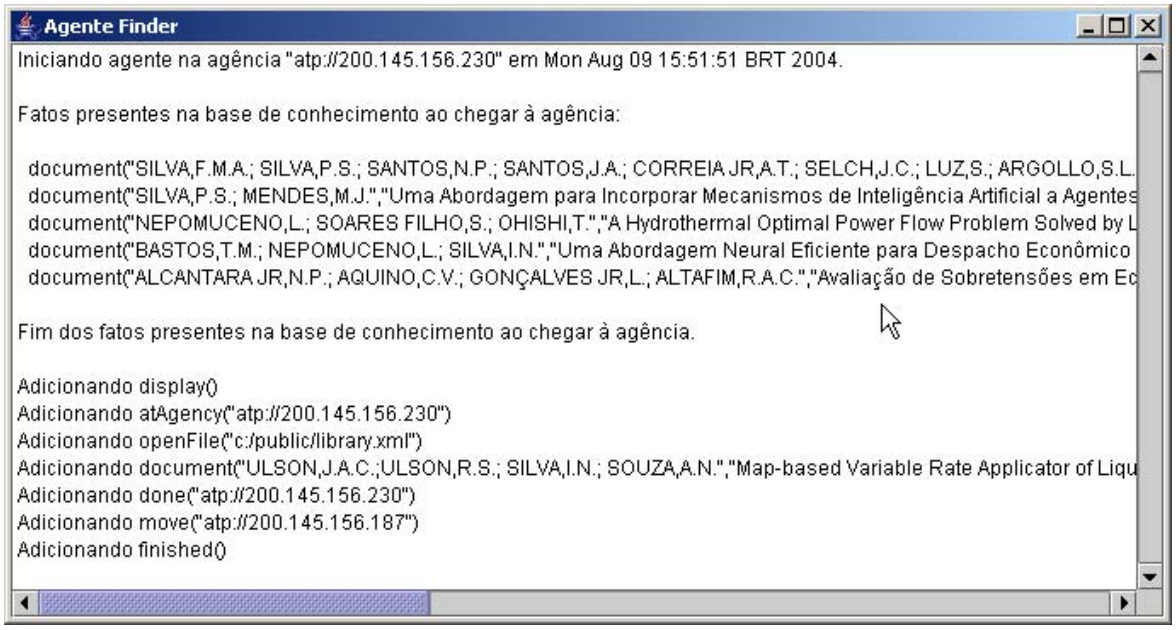


Figura 112: Janela de diálogo mostrada na agência 200.145.156.230

De acordo com a Figura 112, o agente inicia a sua execução, novamente, somente com os fatos relativos ao predicado `document` em sua memória de trabalho.

Quando o mecanismo de inferência é iniciado, as regras 1, 4, 5 e 7 são disparadas, essa última uma única vez, pois, novamente, foi encontrada somente uma publicação com data de 2003:

- `Document("ULSON,J.A.C.;ULSON,R.S.; SILVA,I.N.; SOUZA,A.N." "Map-based Variable Rate Applicator of Liquid Fertilizers Based on Artificial Neural Networks. WSEAS Transaction on Systems." "2003")`

Finalmente, quando a regra 12 é disparada, o agente é enviado de volta para a sua agência natal (200.145.156.187). Quando o mecanismo é interrompido, como parte das tarefas realizadas pelo atuador associado ao predicado `move`, o evento `stoppingEngine` é gerado pela operação `stop()`, provocando o disparo da regra 13, visto que agora `done("atp://200.145.156.230")` está na memória de trabalho. A execução da regra 13

adiciona o fato `finished` a memória de trabalho. Esse fato tem por finalidade indicar ao agente, quando esse iniciar sua próxima execução na agência natal, que a tarefa foi cumprida.

A adição de `finished`, após a chamada de `stop()`, só é possível porque o fluxo de controle de execução do mecanismo só é realmente interrompido após o processamento completo da última percepção recebida (`stoppingEngine`).

Ao chegar à sua agência natal o agente é colocado em execução e uma nova janela de diálogo é aberta, como mostrado na Figura 113. De acordo com essa figura, uma vez que a tarefa foi cumprida, as regras 2 e 3 são ativadas. A primeira salva o resultado da busca em um arquivo denominado `c:/public/resultados.txt`, enquanto a segunda interrompe definitivamente a execução do agente (`stop()`). Devido à estratégia de resolução de conflitos escolhida, a regra 2 é disparada antes da regra 3, o que garante o armazenamento em arquivo dos dados colhidos pelo agente. O conteúdo final do arquivo de resultados está mostrado na Figura 114.

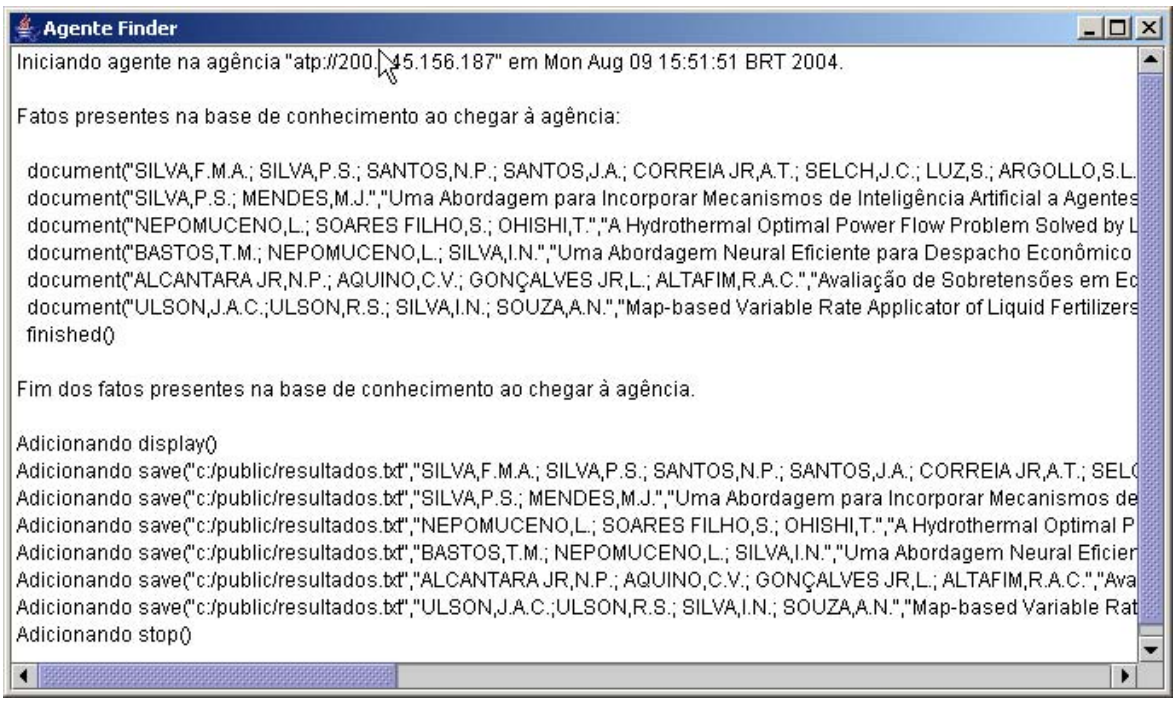


Figura 113: Janela de diálogo mostrada ao voltar para a agência 200.145.156.187

"SILVA,F.M.A.; SILVA,P.S.; SANTOS,N.P.; SANTOS,J.A.; CORREIA JR,A.T.; SELCH,J.C.; LUZ,S.; ARGOLLO,S.L." "Executing Workflows with Mobile Agents from UML Specifications. In: V WORKSHOP DE COMUNICAÇÃO SEM FIO E COMPUTAÇÃO MÓVEL - CSF2003" "2003"

"SILVA,P.S.; MENDES,M.J." "Uma Abordagem para Incorporar Mecanismos de Inteligência Artificial a Agentes Móveis. In: 21. SIMPÓSIO BRASILEIRO DE REDES DE COMPUTADORES." "2003"

"NEPOMUCENO,L.; SOARES FILHO,S.; OHISHI,T." "A Hydrothermal Optimal Power Flow Problem Solved by Lagrangian Relaxation Approach. International Journal of Computer Research" "2003"

"BASTOS,T.M.; NEPOMUCENO,L.; SILVA,I.N." "Uma Abordagem Neural Eficiente para Despacho Econômico com Representação do Sistema de Transmissão. Revista de Iniciação Científica, São Carlos." "2003"

"ALCANTARA JR,N.P.; AQUINO,C.V.; GONÇALVES JR,L.; ALTAFIM,R.A.C." "Avaliação de Sobretensões em Equipamentos Elétricos e Linhas de Transmissão e Coordenação do Isolamento por Simulações Computacionais. In: SEMINÁRIO NACIONAL DE PRODUÇÃO E TRANSMISSÃO DE ENERGIA ELÉTRICA" "2003"

Figura 114: Conteúdo do arquivo de resultados `results.txt`

9.4.3 Discussão

Embora simples, esse cenário evidencia a flexibilidade obtida pela utilização do mecanismo de inferência desenvolvido em conjunto com a linguagem XML da *Web Semântica* (BERNERS-LEE et al., 2001).

Para tornar a sua aplicação ainda mais geral, o cenário poderia ser modificado de forma que os endereços das agências a serem visitadas não fossem representados explicitamente nas regras. Para tanto, o agente deveria migrar, inicialmente, para uma agência "catálogo" que disponibilizaria os endereços das demais agências em função do tipo de busca que o agente estaria realizando. Essa agência catálogo seria uma espécie de páginas amarelas. Além disso, ao chegar em uma agência, o agente também poderia obter o diretório e o nome do arquivo onde as publicações estão armazenadas através de uma consulta direta à agência hospedeira.

Essas modificações tornariam o cenário mais realista, evitando a representação explícita de endereços (URLs) no corpo das regras, o que tornaria a base de conhecimento facilmente reutilizável caso uma outra plataforma de agentes móveis precisasse ser utilizada, visto que cada plataforma pode usar formas diferentes para a representação do endereço das agências.

9.5 Considerações Finais

Este capítulo apresentou uma avaliação de desempenho do *framework* proposto, em termos da quantidade de código e de dados a serem transportados. Também foram explicitadas algumas decisões de implementação que afetam esses valores, por exemplo, a decisão de se re-escrever os métodos `writeObject(...)` e `readObject(...)` para se excluir do fluxo de bits serializados as descrições das classes contêineres, utilizadas nas memórias dos nós-de-duas-entradas, e as descrições dos campos. Foi mostrado ainda um cenário de uso que, embora simples, evidencia suas potencialidades e o uso dos modificadores de sentenças introduzidos na concepção do *framework*.

O principal ponto a se ressaltar é a enorme redução da quantidade de códigos e dados transportada obtida com a abordagem utilizada, quando comparada ao pacote de software em Java mais difundido para a realização de inferência progressiva (JESS). Para a base de conhecimento apresentada na Seção 9.3, o custo do *framework* proposto representa apenas 13,6% do que seria obtido com o JESS.

Capítulo 10

Conclusões

10.1 Introdução

Este trabalho apresentou o desenvolvimento de um *framework* para a integração de técnicas de Inteligência Artificial a agentes móveis construídos com os principais sistemas de mobilidade contemporâneos. Como exemplo de utilização do *framework*, desenvolveu-se um mecanismo de inferência baseado em regras com encadeamento progressivo. Neste capítulo são reunidas as conclusões gerais delineadas ao longo deste documento e apresentadas sugestões para trabalhos futuros.

10.2 Síntese do Trabalho

Do levantamento bibliográfico realizado sobre os atuais sistemas de agentes móveis (Capítulo 3), concluiu-se que os mesmos não contemplam nenhuma habilidade inerente de inteligência. Isto é, esses sistemas não dão suporte à construção de agentes móveis munidos de mecanismos de inferência, de aprendizado ou qualquer outra técnica de IA. Portanto, todos eles podem beneficiar-se de um *Framework de Inteligência*, isto é, uma arquitetura e um conjunto colaborativo e adaptável de classes e interfaces, que permita incorporar aos agentes móveis tais mecanismos. A idéia é que o *framework* proposto, além de "leve", deveria ser suficientemente flexível para que, no futuro, outros mecanismos, como, por exemplo, de inferência em lógica nebulosa e de aprendizado, também possam ser incorporados, ampliando ainda mais a autonomia e a capacidade dos agentes móveis na realização de tarefas em ambientes

dinâmicos e distribuídos, sem reduzir sua mobilidade e aceitabilidade por parte de seus potenciais usuários.

Uma metodologia de desenvolvimento de software orientada a objetos (LARMAN, 1998) e vários padrões de projeto (GAMMA et al., 1995) foram utilizados visando à obtenção de um acoplamento "fraco" entre os diversos componentes e subsistemas do *framework* e a ampliação da possibilidade de extensão e reuso dos elementos identificados. Para tanto, os principais componentes desse *framework* foram isolados, generalizados e implementados como componentes dinâmicos. Essa abordagem aumenta a probabilidade de utilização dos componentes do *framework* e contribui para a redução da quantidade de código e de dados transportados pelo agente.

No Capítulo 6 foram definidos os requisitos que esse *framework* deve satisfazer. Os requisitos propostos representam um compromisso entre flexibilidade e austeridade de código e dados. Da análise desses requisitos, identificaram-se alguns padrões de projetos que deviam ser usados no desenvolvimento do *framework* para que seja possível atendê-los. Especificamente, para cumprir os requisitos especificados, foram utilizados os princípios de projeto *Interface* e *Delegation* e os padrões de projetos *Adapter*, *Template Method*, *Flyweight*, *Strategy* e *Façade*. Durante a análise também se propôs que (1) para reduzir a quantidade de código transportado e garantir a segurança da agência, parte das funcionalidades do *framework* (sensores e atuadores) pode ser transferida para as agências, na forma de um serviço de inteligência; (2) a troca de conhecimento entre o agente e os mecanismos de inteligência deve ser realizada na forma de objetos de conhecimento, que implementam um conjunto de interfaces de representação de conhecimento.

Além disso, também foi definida uma arquitetura para o *framework*. Cada subsistema dessa arquitetura representa uma especialização importante e, como eles são fracamente acoplados entre si, especialistas em um subsistema podem desenvolver novos componentes sem que para isso precisem ter conhecimentos sobre como desenvolver componentes de outros subsistemas. Por exemplo, especialistas em IA poderão se concentrar no desenvolvimento dos mecanismos de inteligência, enquanto especialistas em mobilidade poderão tratar somente dos aspectos de migração; especialistas em interface homem-máquina poderão se concentrar nos aspectos relativos à camada de apresentação, especialistas na lógica da aplicação poderão se preocupar somente com a criação de sensores e atuadores; especialistas na lógica do negócio

poderão se concentrar na criação das regras do negócio enquanto especialistas em *engenharia de conhecimento* poderão se ater na criação, análise sintática e tradução de e para os objetos de representação de conhecimento.

No Capítulo 7 realizou-se o desenvolvimento dos subsistemas básicos do *Framework de Inteligência* usando o paradigma de orientação a objetos, ou seja, dos *Subsistemas Mecanismo Abstrato* e *Interfaces do Serviço de Inteligência*. As classes e interfaces que compõem o subsistema *Mecanismo Abstrato* são utilizadas pelo agente para configurar e controlar a execução dos mecanismos concretos. As classes e interfaces do subsistema *Interfaces do Serviço de Inteligência* são utilizadas na interação entre o agente e o serviço de inteligência concreto oferecido pela agência. Como parte desse núcleo básico foram introduzidos modificadores sentenciais (`EVENT`, `SENSOR`, `EFFECTOR` e `TRANSIENT`) capazes de modificar a forma como os mecanismos devem lidar com determinadas sentenças. A definição dos possíveis modificadores foi estabelecida em uma interface denominada `Modifiers`.

No Capítulo 8 apresentou-se o desenvolvimento de um mecanismo de inteligência específico: o *Mecanismo de Inferência com Encadeamento Progressivo Baseado em Regras*. A escolha de tal mecanismo foi baseada nos seguintes fatores: (1) ele representa uma das técnicas mais conhecidas e estabelecidas de IA; (2) de acordo com diversos pesquisadores (IBM, 1997), esse tipo de mecanismo é requisito primário para a maioria dos sistemas inteligentes, podendo ser usado na configuração e no controle de outros mecanismos – portanto, seu desenvolvimento potencializa futuras incorporações; (3) os agentes móveis enfrentarão, com maior freqüência, a necessidade de selecionar suas ações com base tanto no estado atual de sua base de conhecimento quanto em suas percepções sobre seu ambiente – justamente o tipo de tarefa para a qual um procedimento de inferência com encadeamento progressivo é o mais adequado; (4) vários tipos de conhecimento, como, por exemplo, políticas corporativas de negócios e de segurança são, por natureza, expressas em regras; (5) atualmente o paradigma simbólico ainda é a abordagem predominante em IA. Acredita-se que, provavelmente, esse cenário ainda perdurará por algum tempo (IBM, 2000). Existem várias razões para isso. Talvez a mais importante seja que muitas técnicas de IA simbólica, como os sistemas baseados em regras, levam consigo uma metodologia e uma tecnologia associada que estão se tornando familiares aos pesquisadores e engenheiros dos demais ramos da Ciência da Computação. Por

exemplo, atualmente, as regras tornaram-se conceitos de primeira classe no desenvolvimento da *Web Semântica* (BERNERS-LEE et al., 2001).

No desenvolvimento desse mecanismo de inferência, definiu-se, inicialmente, um subsistema composto somente por um conjunto de interfaces para a representação de conhecimento na forma de regras. As interfaces propostas são independentes do tipo de encadeamento utilizado pelo mecanismo de inferência, isto é, encadeamento progressivo ou encadeamento regressivo. Esse subsistema foi denominado *Subsistema Interfaces de Representação de Conhecimento*. Durante a configuração do agente, a base de conhecimento que irá guiar sua execução, geralmente armazenada em um repositório persistente, deve ser convertida, se necessário, para essa representação. Por outro lado, ao ser transferida para o mecanismo de inferência, essa representação, na forma de objetos de conhecimento, é convertida para a representação interna do *casador* em uso. Para ampliar o poder representacional das *Interfaces de Representação de Conhecimento*, incluiu-se um *predicado reservado*, `EQUAL`, com a intenção de usá-lo em situações específicas, em conjunto com a negação por falha (~), para indicar a igualdade ou, principalmente, a desigualdade entre duas variáveis presentes em literais distintos.

Um subsistema, denominado *Subsistema Mecanismo de Inferência*, foi definido para incluir as classes e interfaces necessárias à especialização dos elementos do *Mecanismo Abstrato* em um *Mecanismo de Inferência com Encadeamento Progressivo Baseado em Regras*. Esse subsistema inclui, além de suas próprias classes, dois outros: o *Subsistema Rete* e o *Subsistema Strategy*. As interfaces dos diversos componentes do mecanismo de inferência e de seus subsistemas foram definidas de forma cuidadosa, de modo a possibilitar que o desenvolvedor do agente utilize outros componentes para essas funções, além daqueles oferecidos.

O *Subsistema Rete* acrescenta ao *Framework de Inteligência* a implementação de um *casador de padrões* desenvolvido de acordo com o algoritmo *Rete*. Durante o desenvolvimento das classes e interfaces que compõem o *Subsistema Rete*, foi dada atenção especial aos aspectos de redução da quantidade de código e de dados a ser transportada. As funcionalidades do *casador* foram decompostas em dois grupos de classes e interfaces, um responsável pela compilação dos objetos de conhecimento na estrutura de nós e testes (`ReteCompiler`) e outro, que utiliza essa estrutura, responsável pela lógica do casamento (`Rete`). O primeiro grupo só é necessário no momento da configuração do mecanismo, durante o processo de instancia-

ção do agente e, portanto, não precisa ser transportado, o que reduz a quantidade de código e dados que deve migrar. No segundo grupo, todas as variáveis e métodos que não são específicos da fase de casamento foram eliminados das classes que compõem a estrutura. Como o número de nós, testes, predicados e constantes presentes na rede pode ser grande, deu-se ênfase aos padrões de projeto que evitassem a replicação desnecessária de instâncias idênticas das classes que representam essas entidades. Especificamente, para evitar as replicações, utilizou-se o padrão *Composite* em conjunto com o padrão *Flyweight*. No *Subsistema Strategy*, incluiu-se apenas uma implementação da estratégia de resolução de conflito *breadth-first*.

No Capítulo 9 apresentou-se uma avaliação de desempenho do *framework* proposto, em termos da quantidade de código e de dados a serem transportados. Foram explicitadas algumas decisões de implementação que afetam esses valores, por exemplo, a decisão de se re-escrever os métodos `writeObject(...)` e `readObject(...)` para se excluir do fluxo de bits serializados as descrições das classes contêineres, utilizadas nas memórias dos nós-de-duas-entradas, e as descrições dos campos. Foi mostrado ainda um cenário de uso que, embora simples, evidencia suas potencialidades e o uso dos modificadores de sentenças introduzidos na concepção do *framework*.

Como resultado da utilização de uma metodologia de desenvolvimento cuidadosa e de padrões de projeto adequados, obteve-se a implementação de um *framework* e um mecanismo de inferência onde a quantidade de código e de dados transportada é compatível com a capacidade de transporte dos atuais sistemas de agentes móveis.

O ganho obtido com a abordagem adotada torna-se evidente quando se compara a quantidade de códigos e dados da proposta apresentada neste trabalho com a do pacote de software em Java mais difundido para a realização de inferência progressiva (JESS). Para a base de conhecimento apresentada na Seção 9.3, o custo do *framework* proposto representa apenas 13,6% do que seria obtido com o JESS.

A partir dos agentes móveis inteligentes criados para testar o *framework*, concluiu-se que uma das vantagens do uso da programação declarativa em tais agentes é a simplificação do processo de simulação da migração forte por meio da migração fraca. Essa simplificação ocorre porque não é necessário dividir o código do agente em blocos e manter-se a informação de qual bloco executar, quando o agente chega em uma determinada agência.

10.3 Contribuições

As contribuições do presente trabalho podem ser resumidas da seguinte forma:

- Desenvolvimento de um *framework* de inteligência "leve" para agentes móveis, capaz de estender os principais sistemas de agentes móveis contemporâneos;
- Proposta de um modelo orientado a objetos para os conceitos da lógica de primeira ordem;
- Proposição de um conjunto de classes e interfaces para troca de conhecimento na forma de regras de produção;
- Proposta de uma forma de marcação para sentenças que não precisam ser transportadas de uma agência para outra (sentenças transientes);
- Proposta de uma forma de marcação para transformar regras de produção em regras Evento-Condição-Ação Situadas;
- Proposta de um modelo orientado a objetos para um mecanismo de inferência com encadeamento progressivo;
- Definição de um conjunto de diretrizes a serem adotadas no desenvolvimento de outros mecanismos de IA para agentes móveis.

As principais diretrizes de projeto, derivadas da experiência obtida com o desenvolvimento do *framework* e do mecanismo de inferência são as seguintes:

- Utilizar o padrão de projeto *Adapter* para incorporar aos agentes móveis as técnicas de IA fornecidas na forma de mecanismos de inteligência. O uso desse padrão permite a construção de mecanismos independentes da plataforma de mobilidade escolhida.
- Utilizar o padrão de projeto *Template Method* para concentrar o comportamento comum a vários mecanismos em uma única superclasse. A fatoração do comportamento comum evita a replicação desnecessária de código e de

dados. O comportamento específico de cada mecanismo deve ser, então, definido em subclasses.

- Utilizar o padrão de projeto *Façade* e o princípio de projeto *Interface* para reduzir a comunicação e as dependências entre os subsistemas que compõem o *framework* e seus mecanismos. Isso promove um acoplamento fraco entre eles, possibilitando o desenvolvimento em separado de cada um e ampliando a probabilidade de extensão e reuso desses elementos.
- Utilizar o padrão de projeto *Strategy* para tornar os mecanismos flexíveis, permitindo que seus principais algoritmos sejam independentes uns dos outros. Esse padrão permite definir uma família de algoritmos, encapsular cada um deles e torná-los intercambiáveis, independentemente dos clientes que os utilizam. Para aplicá-lo, as principais funcionalidades dos mecanismos devem ser isoladas e atribuídas a componentes distintos com interfaces bem definidas.
- Separar os procedimentos utilizados somente na fase de inicialização, tanto do agente quanto de seus mecanismos, dos procedimentos utilizados na fase de execução dos mesmos e implementá-los em objetos separados. O relacionamento entre o agente, ou seus mecanismos, com os objetos responsáveis pela inicialização deve seguir o princípio de projeto *Delegation*. Dessa forma, tais objetos poderão ser marcados como transientes, evitando-se, assim, que sejam transportados durante a migração. Caso essas funcionalidades se façam necessárias, sob certas circunstâncias, em outras agências elas poderão ser transferidas sob demanda.
- Utilizar o padrão de projeto *Flyweight* para dar suporte ao compartilhamento de grandes quantidades de objetos de granularidade fina, garantindo assim uma economia de armazenamento, e por conseqüência do custo de migração. Os objetos compartilhados devem ser criados por *fábricas flyweight parametrizadas*. Essa diretriz é especialmente útil na implementação orientada a objetos das bases de conhecimentos dos mecanismos. De certa forma, a utilização das fábricas *flyweights* também ajuda a garantir a opacidade da BC, melhorando a sua segurança, pois os objetos que representam suas sentenças

possuirão somente ponteiros (referências) para objetos de granularidade mais fina, criados pelas fábricas, o que dificulta a interpretação.

- Durante a migração, os objetos são copiados por valor. Portanto, as *fábricas flyweight* não devem utilizar referências como chaves para identificar os objetos já criados e armazenados em suas tabelas *hash*. A chave deve ser criada com base no estado intrínseco do objeto, isto é, nas informações que ele mantém como atributos.
- O agente e seus mecanismos não devem manter referências a implementações das *Interfaces de Representação do Conhecimento* desenvolvidas por seu usuário ou recebidas das agências que o mesmo visita. A manutenção de referências a tais implementações pode fazer com que códigos e dados desnecessários sejam transportados ou, no pior caso, até mesmo impedir a migração do agente, se algum de seus componentes não puder ser serializado para o transporte. Portanto, é necessário realizar uma "tradução" de tais objetos para uma implementação minimalista das *Interfaces de Representação do Conhecimento*, desenvolvida pelos criadores dos mecanismos.
- O padrão de projeto *Immutable* deve ser utilizado na criação das *Interfaces de Representação do Conhecimento*. Isto é, essas interfaces devem possuir somente operações de acesso aos elementos constitutivos das sentenças, sem permitir que os mesmos sejam alterados. Essa necessidade decorre do fato de que referências a objetos de conhecimento criados pelos mecanismos, além de ficarem armazenados na BC do agente, também podem ser transferidas para as agências, por exemplo, através dos atuadores, e, portanto, não devem correr o risco de serem alterados por elas.

10.4 Trabalhos Futuros

Como trabalhos futuros, vislumbra-se o seguinte:

- Avaliar a escalabilidade da implementação realizada para o algoritmo *Rete* por meio de uma análise de complexidade espacial relacionando o número de regras na memória de regras, o número médio de antecedentes em cada regra e o número de fatos na memória de trabalho com a ordem de grandeza (em bytes) da base de conhecimento transportada pelo agente.
- Ampliar a expressividade das regras manipuladas pelo mecanismo de inferência com encadeamento progressivo através da implementação de *regras default priorizadas* propostas por Grosof (1997, 1999). Essas regras fornecem um método para resolver conflitos que surgem durante a autoria, atualização e junção de bases de conhecimentos. Isso é especialmente útil para a criação de agentes inteligentes baseados em regras por autores não-técnicos responsáveis por criar regras para aplicações comerciais, tais como filtragem personalizada de informações e *workflow*.
- Desenvolvimento de recursos para facilitar o uso do *framework*, por exemplo, um editor de regras com interface gráfica de usuário capaz de permitir a definição de predicados representando eventos, sensores, atuadores e informações transientes e a vinculação desses predicados a procedimentos externos, nos casos aplicáveis.
- Construção de outros tipos de mecanismos, por exemplo, para raciocínio probabilístico, para raciocínio em lógica nebulosa e de aprendizado de máquina, tais como redes neurais e sistemas classificadores baseados em técnicas de algoritmos genéticos. O estudo de novos mecanismos de inferência pode levar ao estabelecimento de uma nova classe abstrata denominada, por exemplo, `InferenceEngine`, onde as operações comuns a todos esses mecanis-

mos poderão ser definidas. Como já mencionado, a fatoração de operações comuns ajuda na redução do código e dados.

- Desenvolvimento de cenários de aplicações específicos, baseados em agentes móveis, construídos em conjunto com o *framework* desenvolvido. Em especial, propõe-se o uso desses agentes em cenários de coleta de informações na *World-Wide-Web*, na *Web Semântica* e em aplicações de *workflows*.
- Avaliar a utilidade do *framework* em áreas em que o uso de inferência em conjunto com a mobilidade está sendo largamente considerado como, por exemplo, nas áreas de *Mobility Aware* e *Context Aware* (BROY, 2004).

10.5 Trabalhos Publicados pelo Autor

O desenvolvimento deste trabalho possibilitou a publicação dos seguintes artigos, relacionados ao tema pesquisado:

SILVA, Paulo Sérgio da; MENDES, Manuel de Jesus. Uma Abordagem para Incorporar Mecanismos de Inteligência Artificial a Agentes Móveis. In: SIMPÓSIO BRASILEIRO DE REDES DE COMPUTADORES, 21. (SBRC2003), 2003, Natal. **Anais...** Natal: UFRN/DIMAp, 2003, v. 2, p. 837-852.

SILVA, Paulo Sérgio da; MENDES, Manuel de Jesus. A Framework for Adding Computational Intelligence to Mobile Agents. In: AISB SYMPOSIUM ON SOFTWARE MOBILITY AND ADAPTIVE BEHAVIOUR, 2001, Heslington, York, UK. **Proceedings…** Heslington, York, UK: SSAISB, 2001. p. 114-121.

SILVA, Paulo Sérgio da; MENDES, Manuel de Jesus. Reducing the Gap Between Mobile and Intelligent Software Agents. In: 5TH WORLD MULTICONFERENCE ON SYSTEMICS, CYBERNETICS AND INFORMATICS (SCI), 2001, Orlando, Florida, USA. **Proceedings of the Information System Development Session**. Orlando, USA: IIIS & IEEE Computer Society, 2001. v. 1, p. 13-18.

SILVA, Paulo Sérgio da; MENDES, Manuel de Jesus. An Architecture for Embedding Rule-Based Intelligence into Mobile Agents. In: II WORKSHOP ON ADVANCES & TRENDS IN ARTIFICIAL INTELLIGENCE (ATAI 2001), 2001, Punta Arenas, Chile. **Electronic Proceedings (CD-ROM)**. Punta Arenas, Chile: JCCC & IEEE Computer Society, 2001.

SILVA, Paulo Sérgio da; MENDES, Manuel de Jesus. Uma Infra-Estrutura para Incorporar Inteligência Computacional a Agentes Móveis. **SBPN Scientific Journal**, São Paulo, v. 5, n. 1, p. 178-180, 2001.

MENDES, M.; FALSARELLA, O.; FONTES, I.; KRAUSE, S.; LOYOLLA, W.; MENDEZ, C.; SILVA, P. S.; TOBAR, C. Architectural Considerations about Open Distributed Agent Support Platforms. In: THIRD INTERNATIONAL SYMPOSIUM ON AUTONOMOUS DECENTRALIZED SYSTEMS (ISADS 97), 1997, Berlin, Germany. **Proceedings…** Berlin, Germany, 1997. p. 359-366.

FALSARELA, O.; MENDEZ, C.; FONTES, I.; SILVA, P.S.; LOYOLLA, W.; MENDES, M. Um Serviço de Mobilidade para Agentes Baseados em Plataformas CORBA. In: **Revista do Instituto de Informática**. Campinas:Puccamp, 1996. v. 4, n. 2, p. 58-66.

Referências Bibliográficas

AGENTBUILDER (2000). **Agent Construction Tools**. Disponível em: <http://www. agent-builder.com/AgentTools/>. Acesso em: 2 out. 2000.

AGHA, G.; WEGNER, P.; YONEZAWA, A. (Eds.) (1993). **Research Directions in Concurrent Object-Oriented Programming**. Cambridge, MA: The MIT Press.

AGRE, P.; CHAPMAN, D. (1987). Pengi: An implementation of the theory of activity. In: NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE (AAAI-87), 6, 1987, Seattle. **Proceedings…** Seattle:AAAI Press. p. 268-272.

ALVARES, L. O.; SICHMAN, J. S. (1997). Introdução aos Sistemas Multiagentes. In: CONGRESSO DA SOCIEDADE BRASILEIRO DE COMPUTAÇÃO, 17, 1997, Brasília. MEDEIROS, C. M. B. (Ed.) **Anais …** Brasília: UnB. p. 1-37.

ANANDA, A. L., SRINIVASAN, B. (Eds.) (1991). **Distributed Computing Systems: Concepts and Structures**. Los Alamitos, CA: IEEE Computer Society Press.

APPLEBY, S.; STEWARD, S. (1994). Mobile Software Agents for Control in Telecommunications Networks. **BT Technological Journal**, v.12, n. 2, p.104-113, April.

ARA (2000). **The Ara Project Home Page**. Department of Computer Science, University of Kaiserslauter. Disponível em: <http://wwwagss.informatik.uni-kl.de/Projekte/Ara/ index_e.html>. Acesso em 2 out. 2000.

ASSIS SILVA, F. M. (1999). **A Transaction Model based on Mobile Agents**. PhD Thesis. Technical University of Berlin.

ASTRA (2000). **Jumping Beans White Paper**. Ad Astra Enginnering Inc., October 1999.Disponível em <http://www.Jumpingbeans.com/>. Acesso em: 2 out. 2000.

BERNERS-LEE, T.; HENDLER, J.; LASSILA, O. (2001). The Web Semantic. **Scientific American**, p. 29-37, May.

BIGUS, J. P.; BIGUS, J. (1998). **Constructing intelligent agents with Java: a programmer's guide to smarter applications**. New York, NY: John Wiley & Sons.

BOLEY, H. (2002). **Rule Markup Language**. RuleML Homepage. Disponível em <http://www.dfki.uni-kl.de/ruleml/>. Acesso em: 23 mar. 2002.

BONASSO, R. P.; KORTENKAMP, D.; MILLER, D. P.; SLACK, M. (1996). Experiences with an architecture for intelligent, reactive agents. In: WOOLDRIDGE, M.; MÜLLER J. P.; TAMBE, M. (Ed.) **Intelligent Agents: II, Lecture Notes in Artificial Intelligence**, Berlin Heidelberg: Springer-Verlag, n. 1037, p. 261-276.

BOOCH, G. (1994) **Object-Oriented Analysis and Design**, **2nd Edition**. Redwood City, CA.: Benjamin-Cummings.

BRADSHAW, J. M. (1997). An Introduction to Software Agents. In: BRADSHAW, J. M. (Ed.) **Software Agents**. Cambridge, MA: The MIT Press.

BRADSHAW, J. M. (Ed.) (1997). **Software Agents**. Cambridge, MA: MIT Press.

BRATMAN, M. E. (1987). **Intention, Plans, and Practical Reasoning**. Cambridge, MA: Harvard University Press.

BRATMAN, M. E.; ISRAEL, D. J.; POLLACK M. E. (1988) Plans and resource-bounded practical reasoning. **Computational Intelligence**, v. 4, p. 349-355.

BROOKS, R. A. (1986). A robust layered control system for a mobile robot. **IEEE Journal of Robotics and Automation**,v. 2. n. 1, p. 14-23.

BROOKS, R. A. (1991a). Intelligence without reason. In: INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE (IJCAI-91*)*, 12, 1991, Sidney. **Proceedings…** p. 569-595.

BROOKS, R. A. (1991b). Intelligence without representation. **Artificial Intelligence**, n. 47, p. 139-159.

BROY, M. (2004). **Competence Center Mobility & Context Aware Architecture Home Page**. Technische Universität München - Fakultät für Informatik. Disponível em: http://wwwbroy.informatik.tu-muenchen.de/research/mobility/index_en.shtml. Acesso em: 10 nov. 2004.

CAMMARATA, S.; McARTHUR, D.; STEEB, R. (1983). Strategies of cooperation in distributed problem solving. In: INTERNATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE (IJCAI-83), 8, Karlsruhe, 1983. **Proceedings…** Karlsruhe: University of Karlsruhe. p. 767-770.

CARDELLI, L. (1995). A Language with Distributed Scope. **Computing Systems**, v. 8, n. 1, p. 27-59, Jan.

CETUS (2000). **Distributed Objects & Components: Mobile Agents**. Disponível em: <http://cetus-links.org/oo_mobile_agents.html>. Acesso em: 31 maio 2000.

CHAPMAN, D. (1987).Planning for conjunctive goals. **Artificial Intelligence**, n. 32, p. 333-378.

CHESS, D. M.; GROSOF, B.; HARRISON, C.; LEVINE, D.; PARRIS, C.; TSUDIK, G. (1995). Itinerant Agents for Mobile Computing. **IEEE Personal Communications Magazine**, v. 2, n. 5, p. 34-49, Oct.

CHESS, D. M.; HARRISON, C. G.; KERSHENBAUM, A. (1994). **Mobile Agents: Are they a good idea?**, IBM Research Report, RC 19887, T. J. Watson Research Center, Oct.

COAD, P. (1995). **Object Models: Strategies, Patterns and Applications**. Englewood Cliffs, NJ: Prentice-Hall.

COLLEMAN, D.; ARNOLD, P.; BODOFF, S.; DOLLIN, C.; GILCHRIST, H.; HAYES, F.; JEREMAES, P. (1994). **Object-Oriented Development: The Fusion Method**. Englewood Cliffs, NJ: Prentice-Hall.

D'AGENT (2000). **D'Agent Home Page**. Center for Mobile Computing (CMC), Dartmouth College. Disponível em: <http://agent.cs.dartmouth.edu/>. Acesso em: 31 maio 2000.

DEMAZEAU, Y.; MÜLLER, J. P. (1990). Decentralized Artificial Intelligence. In: EUROPEAN WORK-SHOP ON MODELLING AUTONOMOUS AGENTS AND MULTI-AGENTS WORLDS (MAAMAW-89*)*, 1, 1989, Cambridge, England. DEMAZEAU, Y.; MÜLLER, J.P. (Ed.) **Decentralized AI - Proceedings…** Cambridge, England: Elsevier. p. 3-13.

DURFEE, E. H. (1988). **Coordination of Distributed Problem Solvers**. Boston, MA: Kluwer Academic Publishers.

FERGUSON, I. A. (1992). **Touring Machines: An Architecture for Dynamic, Rational Mobile Agents**. Ph.D. Thesis - Clare Hall, University of Cambridge, UK.

FERGUSON, I. A. (1995). Integrated control and coordinated behaviour: A case for agent models. In: WOOLDRIDGE, M.; JENNINGS, N. R. (Ed.) **Intelligent Agents: Theories, Architectures, and Languages, Lecture Notes in Artificial Intelligence**, Berlin Heidelberg: Springer-Verlag, n. 890, p. 261-276.

FIKES, R. E.; NILSSON, N. (1971). STRIPS: A new Approach to the application of theorem proving to problem solving. **Artificial Intelligence**, v. 5, n. 2, p. 189-208.

FININ, T., FRITZSON, R., MCKAY, D., MCENTIRE, R. (1994). KQML as an Agent Communication Language. In: INTERNATIONAL CONFERENCE ON INFORMATION AND KNOWLEDGE MANAGEMENT (CIKM'94), 3, 1994, Gaithersburg, Maryland. **Proceedings…** Maryland: ACM Press. p. 456-463.

FININ, T.; FRITZSON, R. (1994). **KQML - A Language and Protocol for Knowledge and Information Exchange**. Technical Report CS-94-02. Computer Science Department, University of Maryland. Disponível em: http://www.cs.umbc.edu/kqml/papers/kbkshtml/kbks.html>. Acesso em: 9 nov. 1995.

FONER, L. (1993). **What's an Agent, Anyway? A Sociological Case Study**. Agents Memo 93-01, Cambridge, MA: MIT Media Lab., 1993. Disponível em: <http://foner.www.media.mit.edu/people/foner/Julia/>. Acesso em: 1 dez. 1997.

FORGY, C. L. (1982). Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. **Artificial Intelligence**, v. 19, p. 17-37.

FOUNDATION FOR INTELLIGENT PHYSICAL AGENTS (FIPA) (1998). **FIPA'98 Draft Specifications**, Switzerland. Disponível em: <http://drogo.cselt.stet.it/fipa/spec/fipa98/ fipa98.htm>. Acesso em: 3 ago. 1998.

FRIEDMAN-HILL, E. J. (2001). **Jess, The Java System Shell Version 6.03a**. Report SAND98-8206, Livermore, CA: Distributed Computing Systems, Sandia National Laboratories, Livermore, CA. Disponível em: <http://herzberg.ca.sandia.gov/jess/ readme.html>. Acesso em: 15 fev. 2001.

FUGGETA, A.; PICCO, G. P.; VIGNA, G. (1998). Understanding Code Mobility. **IEEE Trans. on Software Engineering**, v. 24, n. 5, p. 342-361.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. (1995). **Design Patterns: Elements of Reusable Object-Oriented Software**. New York: Addison Wesley.

GENERAL MAGIC (1997). **Telescript Home Page**. Disponível em: < http://www.genmagic.com/Telescript/>. Acesso em: 15 maio 1997.

GENERAL MAGIC (1999). **Odyssey Home Page**. Disponível em: <http://genmagic.com/technology/odyssey.html>. Acesso em: 6 out. 1999.

GENESERETH, M. R. (1998). **Knowledge Interchange Format**. Draft proposed American National Standard (dpANS) NCITS.T2/98-004. Disponível em: <http://logic.stanford.edu/kif/dpans.html>. Acesso em: 2 fev. 1999.

GEORGEFF, M. P.; LANSKY, A. L. (1987). Reactive reasoning and planning. In: NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE (AAAI-87), 6, 1987, Seattle. **Proceedings…** Seattle: AAAI Press. p. 677-682.

GLASS, G. (2000). **Overview of Voyager: ObjectSpace's Product Family for State-of-the-Art Distributed Computing.** 1999. Disponível em: <http://www.objectspace.com/ products/voyager/>. Acesso em: 6 out. 2000.

GRAND, M. (1998). **Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML**. New York, NY: John Wiley & Sons. v. 1.

GREEN, S.; HURST, L.; NANGLE, B.; CUNNINGHAM, P.; SOMERS, F.; EVANS, R. (1997). **Software Agents: A review.** Dublin: Trinity College, May. Disponível em <http://ww.cs.tcd.ie/Breanda.Nangle/iag.html>. Acesso em: 10 ago. 1997.

GROSOF, B. N. (1997). **Courteous Logic Programs: Prioritized Conflict Handling for Rules**. Research Report. Yorktown Heights: IBM T. J. Watson Research Center, Sept. Disponível em: <http://www.research.ibm.com/rules/paps/rc20836.pdf>. Acesso em: 9 dez. 1999. Acesso em: 9 dez. 1999.

GROSOF, B. N. (1999). **Business Rules for Electronic Commerce: Interoperability and Conflict Handling**. Palestra. Hawthorne: IBM T. J. Watson Research Center, Sept. Disponível em: <http://www.research.ibm.com/rules/paps/talk_proj_ovw.pdf>. Acesso em: 9 dez. 1999.

GROSOF, B. N.; LABROU, Y. (1999). An Approach to using XML and a Rule-based Content Language with an Agent Communication Language. In: WORKSHOP ON AGENT COMMUNICATION LANGUAGES (ACL-99), INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE (IJCAI-99), Stockholm. **Proceedings…** Stockholm, 1999a. Disponível em <http://www.research.ibm.com/rules/paps/ rc21491.ps> Acesso em: 9 dez. 1999.

GROSOF, B. N.; LABROU, Y.; CHAN, H. Y. (1999). A Declarative Approach to Business Rules in Contracts: Courteous Logic Programs in XML. In: ACM CONFERENCE ON ELECTRONIC COMMERCE (EC-99), 1, Denver. **Proceedings…** New York: ACM Press, 1999. Disponível em: <http://www.research.ibm.com/rules/paps/ ec99_proceedings.pdf> Acesso em: 9 dez. 1999.

GROSOF, B. N.; LEVINE, D. W.; CHAN, H. Y.; PARRIS, C. J.; AUERBACH, J. S. (1995). Reusable Architecture for Embedding Rule-based Intelligence in Information Agents. In: WORKSHOP ON INTELLIGENT INFORMATION AGENTS, ACM CONFERENCE ON INFORMATION AND KNOWLEDGE MANAGEMENT (CIKM-95), Baltimore. **Proceedings…** Baltimore: University of Maryland, 1995. Disponível em <http://www. research.ibm.com/iagents/paps/rc20305.ps>. Acesso em: 9 dez. 1999.

GYPSY (2000). **The Gypsy Project Home Page**. Distributed Systems Group, Technical University of Vienna**.** Disponível em: <http://www.infosys.tuwien.ac.at/Gypsy/homepage.html>. Acesso em 2 oct. 2000.

HU, J.; WELLMAN, M. P. (1996). Self-fulfilling bias in multiagent learning. In: INTERNATIONAL CONFERENCE ON MULTIAGENT SYSTEMS (ICMAS-96), 2, Kyoto, 1996. **Proceedings…** Kyoto. p. 118-125.

INFORMATIONS- UND KOMMUNIKATIONSTECNOLOGIE (IKV++) (2001a). **Grasshopper 2.1 Users' Guide**. Disponível em: <http://www.ikv.de/products/grasshopper/>. Acesso em: 27 nov. 2001.

INFORMATIONS- UND KOMMUNIKATIONSTECNOLOGIE (IKV++) (2001b). **Grasshopper 2.1 Programmers' Guide**. Disponível em: <http://www.ikv.de/products/ grasshopper/>. Acesso em: 27 nov. 2001.

INFORMATIONS- UND KOMMUNIKATIONSTECNOLOGIE (IKV++) (2001c). **Grasshopper Home Page**. Disponível em: <http://www.ikv.de/products/ grasshopper/>. Acesso em: 27 nov. 2001.

INTERNATIONAL BUSINESS MACHINE (IBM) (1997). **Overview of IBM Agent Building Environment Developer's Toolkit Level 6**. Intelligent Agent Center of Competence.

Disponível em: <http://www. networking.ibm.com/iag/iaghome.html>. Acesso em: 25 jun. 1997.

INTERNATIONAL BUSINESS MACHINE (IBM) (2000). **Overview of the Agent Building and Learning Environment**. ABLE Home Page. Disponível em: <http://www.alphaworks. ibm.com/>. Acesso em: 8 set. 2000.

INTERNATIONAL BUSINESS MACHINE (IBM) (2002a). **Aglets Home Page.** Tokyo Research Laboratory. Disponível em: <http://www.trl.ibm.com/aglets/>. Acesso em: 8 jun. 2002.

INTERNATIONAL BUSINESS MACHINE (IBM) (2002b). **Aglets Specification 1.1**. Tokyo Research Laboratory, July 2002b. Disponível em: < http://www.trl.ibm.com/aglets/ >. Acesso em: 8 jun. 2002.

ISMAIL, L.; HAGIMONT, D. (1999). A Performance Evaluation of the Mobile Agent Paradigm. In: CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, SYSTEMS, LANGUAGES, AND APPLICATIONS (OOPSLA'99), Denver, 1999. **Proceedings…** Denver. p. 306-313.

JACOBSON, I.; CHRISTERSON, M.; JONSSON, P.; OVERGAARD, G. (1992). **Object-Oriented Software Engineering: A Use Case Driven Approach**. Reading, MA: Addison-Wesley.

JAVA-TO-GO (2000). **Java-to-go Home Page**. Department of Electrical Engineering and Computer Sciences, University of California at Berkely. Disponível em <http://ptolemy.eecs. berkeley.edu/~wli/group/java2go/java-to-go.html>. Acesso em: 2 out. 2000.

JENNINGS, N. R.; FARATIN, P.; JOHNSON, M. J.; NORMAN, T. J.; O'BRIEN, P.; WIEGAND, M. E. (1996). Agent Based Business Process Management. **International Journal of Cooperative Information Systems**, v. 5, n. 2, p. 105-130.

JENNINGS, N. R.; SYCARA, K., WOOLDRIDGE, M. (1998). A Roadmap of Agent Research and Development. **Autonomous Agents and Multi-Agent Systems**, n. 1, p. 7-38.

JUL, E. LEVY, H.; HUTCHINSON, N.; BLACK, A. (1988). Fine-grained mobility in the Emerald system. **ACM Transactions on Computer Systems**, v. 6, n. 1, p. 109-133, Feb. Apud Wong et al. (1999).

KNOWBOT (2000). **Knowbot System Software Home Page**. Corporation for National Research Initiatives (CNRI). Disponível em: <http://www.cnri.reston.va.us/home/koe/index. html>. Acesso em: 2 out. 2000.

LARMAN, C. (1998). **Applying UML and patterns: an introduction to object-oriented analysis and design**. Upper Saddle River, NJ: Prentice-Hall.

LESSER, V. R. (1991). A retrospective view of FA/C distributed problem solving. **IEEE Transactions on System, Man, and Cybernetics**, v. 21, n. 6, p. 1347-1362, Dec.

LESSER, V. R.; DURFEE, E. H.; CORKILL, D. D. (1989). Trends in cooperative distributed problem solving. **IEEE Transactions on Knowledge and Data Engineering**, v. 1, n. 1, p. 63-83.

MAES, P. (1989). The dynamics of action selection. In: INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE (IJCAI-89), 11, 1989, Detroit. **Proceedings…** Detroit. p. 991-997.

MAES, P. (1990). Situated agents can have goals, In: MAES, P. (Ed.) **Designing Autonomous Agents**. Cambridge, MA: The MIT Press. p. 49-70.

MAES, P. (1991). The agent networks architecture (ANA). **SIGART Bulletin**, v. 2, n. 4, p. 115-120.

MAES, P. (1994). Agents that Reduce Work and Information Overload. **Communications of the ACM**, v. 37, n. 7, p. 30-40.

MENDEZ, C. A. (2000). **Uma abordagem para o transporte de agentes móveis em ambientes abertos, heterogêneos e distribuídos**. Tese (Doutorado em Engenharia Elétrica e Computação) – Faculdade de Engenharia Elétrica e de Computação, Universidade Estadual de Campinas, Campinas.

MENDEZ, C. A.; MENDES, M. J. (1999). Agent Migration Issues in CORBA Platforms. In: THIRD INTERNATIONAL SYMPOSIUM ON AUTONOMOUS DECENTRALIZED SYSTEMS (ISADS`99), 5, 1999, Tokyo. **Proceedings…** Tokyo.

MINAR, N.; KRAMER, K. H.; MAES, P. (1999). Cooperating Mobile Agents for Mapping Networks. In: HUNGARIAN NATIONAL CONFERENCE ON AGENT BASED COMPUTATION, 1, 1999. **Proceedings…** Disponível em: <http://nelson.www.media.mit.edu/people/nelson/research/routes-coopagents/routes-coopagents-a4.ps.gz>. Acesso em: .31 maio 2000.

MITSUBISHI (1998). **Mobile Agent Computing: A White Paper.** Electric Information Technology Center American. Horizon Systems Laboratory, Jan. Disponível em: <http://www.merl.com/projects/ concordia/>. Acesso em: 5 fev. 1999.

MITSUBISHI (2000a). **Concordia Home Page**. Electric Information Technology Center American. Horizon Systems Laboratory. Disponível em: < http://www.merl.com/projects/ concordia/>. Acesso em: 6 out. 2000.

MITSUBISHI (2000b). **Concordia Technology - At a Glance**. Electric Information Technology Center American. Horizon Systems Laboratory. Disponível em: <http://www.merl.com/projects/concordia/>. Acesso em: 6 out. 2000.

MOLE (2000). **Mole Home Page**. Institute of Parallel and Distributed High Performance Systems, University of Stuttgart. Disponível em <http://mole.informatik.uni-stuttgart.de/papers.html>. Acesso em: 31 maio 2000.

MÜLLER, J. P. (1997). **The Design of Intelligent Agents**. Berlin Heidelberg: Springer-Verlag.

MÜLLER, J. P.; PISCHEL, M. (1994). Modelling interacting agents in dynamic environments. In: EUROPEAN CONFERENCE ON ARTIFICIAL INTELLIGENCE (ECAI-94*)*, 11, 1994, Amsterdam. **Proceedings…** p. 709-713.

NEGROPONTE, N. (1997). Agents: From Direct Manipulation to Delegation. In: BRADSHAW J. M. (Ed.) **Software Agents**. Cambridge, MA: MIT Press.

NEWELL, A.; SIMON, H. A. (1961). GPS: A program that simulates human thought. In: OLDENBOURG, R. (Ed.) **Lernede Automaten**, KG.

NWANA, H. S. (1996). Software Agents: An Overview. **Knowledge Engineering Review**, v. 11, n. 3, p. 1-40, Sept.

NWANA, H. S.; NDUMU, D. T. (1999a). Agents of Change in Future Communications Systems. In: HAYZELDEN, A. L. G.; BIGHAM, J. (Ed.) **Software Agents for Future Communication Systems**. Berlin Heidelberg: Springer-Verlag.

NWANA, H. S.; NDUMU, D. T. (1999b). A Perspective on Software Agents Research. **Knowledge Engineering Review**, v. 12, n. 2, p. 1-18, Jan. Disponível em: <http://www.labs.bt. com/ projects/agents/ publish/papers/hn-dn-ker99.zip>. Acesso em: 5 jun. 2000.

OBJECT MANAGEMENT GROUP (OMG) (1997). **Mobile Agents Facility (MASIF)**. Document orbos/97-10-05, May. Disponível em: <http://www.omg.org/cgi-bin/doc/orbos/97-10-05>. Acesso em: 6 jan. 1998.

OBJECT MANAGEMENT GROUP (OMG) (1999). **Unified Modeling Language Specification – Version 1.3**. Document ad/99-06-08, June. Disponível em: <www.rational.com/ uml/resources/ documentation/ media/ad99-06-08-pdf>. Acesso em 29 nov. 2000.

OBJECT MANAGEMENT GROUP (OMG) (2000). Agent Working Group. **Agent Technology Green Paper – Version 1.00**. Document agent/2000-09-01, Sept. Disponível em: <http://www.objs.com/ agent/agents_Green_Paper_v100.doc>. Acesso em 29 nov. 2000.

OBJECTSPACE (2000). **Voyager Home Page**. Disponível em: <http://www.objectspace.com/products/voyager/>. Acesso em: 6 out. 2000.

OUSTERHOUT, J. (1995). **Tcl and the Tk Toolkit**. New York, NY: Addison Wesley.

PEINE, H.; STOLPMANN, T. (1997). The Architecture of the Ara Platform for Mobile Agents. In: ROTHERMEL, K.; POPESCU-ZELETIN, R. (Eds.) **Mobile Agents: 1st International Workshop MA'97, Lecture Notes in Artificial Intelligence**, Berlin Heidelberg: Springer-Verlag, n. 1219.

PLU, M. (1995). Software Agents in Telecommunications Network Environments. In: **UNICOM Seminar on Intelligent Agents and their Business Applications**, London, November 1995. p. 225-243. Apud Nwana e Ndumu (1999a).

RAO, A. S.; GEORGEFF, M. P. (1991). Modeling rational agents within a BDI-architecture. In: KNOWLEDGE REPRESENTATION AND REASONING (KR&R-91), San Mateo, 1991. FIKES, R.; SANDEWALL, E. (Ed.) **Proceedings…** San Mateo: Morgan Kaufmann Publishers. p. 473-484.

RAO, A. S.; GEORGEFF, M. P. (1992). An abstract architecture for rational agents. In: KNOWLEDGE REPRESENTATION AND REASONING (KR&R-92), 1992. RICH, C.; SWARTOUT, W.; NEBEL, B. (Ed.) **Proceedings…** p. 439-449.

RAO, A. S.; GEORGEFF, M. P. (1995). BDI Agents: From Theory to Practice. In: INTERNATIONAL CONFERENCE ON MULTIAGENT SYSTEMS *(*ICMAS-95), 1, San Francisco, 1995. **Proceedings…** San Francisco. p. 312-319.

RATIONAL SOFTWARE CORPORATION (2000). **Rational Unified Process White Paper: Best Practices for Software Development Teams.** Disponível em: <http://www. rational.com/sitewide/support/whitepapers/dynamic.jtmpl>. Acesso em: 24 ago. 2000.

RICH, E.; KNIGHT, K. (1994) **Inteligência Artificial**. São Paulo, SP: Makron Books do Brasil.

RIELY, J.; HENNESSY, M. (1999). Trust and Partial Typing in Open Systems of Mobile Agents. In: ACM SIGPLAN-SIGACT SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES, 26, 1999. **Proceedings…**ACM Press.

ROSENSCHEIN, J. S. (1985). **Rational Interaction: Cooperation Among Intelligent Agents**. Ph. D Thesis, Computer Science Department, Stanford University, Stanford.

ROSENSCHEIN, J. S.; KAELBLING, L. P. (1986). The synthesis of digital machines with provable epistemic properties. In: CONFERENCE ON THEORETICAL ASPECTS OF REASONING ABOUT KNOWLEDGE, 1986., San Mateo. HALPERN, J. Y. (Ed.) **Proceedings…** San Mateo: Morgan Kaufmann. p. 83-98.

ROSENSCHEIN, J. S.; ZLOTKIN, G. (1994). **Rules of Encounter: Designing Conventions for Automated Negotiation among Computers**. Boston, MA: The MIT Press.

RUMBAUGH, J.; BLAHA, M.; PREMERLANI, W.; EDDY, F.; LORENSON, W. (1991) **Object-Oriented Modeling and Design**. Englewood Cliffs, NJ: Prentice-Hall.

RUSSEL, S.; NORVIG, P. (1995). **Artificial Intelligence: A Modern Approach**. Englewood Cliffs, NJ: Prentice-Hall.

SANDHOLM T.; LESSER, V. (1995). Issues in automated negotiation and electronic commerce: extending the contract net protocol. In: INTERNATIONAL CONFERENCE ON

MULTIAGENT SYSTEMS (ICMAS-95), 1995, San Francisco. **Proceedings…** San Francisco. p. 328-335.

SILVA, L. M.; SOARES, G.; MARTINS, P.; BATISTA, V.; SANTOS, L. (1999). The Performance of Mobile Agent Platforms. In: INTERNATIONAL SYMPOSIUM ON AGENT SYSTEMS AND APPLICATIONS, 1, INTERNATIONAL SYMPOSIUM ON MOBILE AGENTS, 3, (ASAMA'99), 1999, Palm Springs. **Proceedings…** Palm Springs, Poster Session.

SILVA, P. S. da. (1998). **Especificação de uma Camada de Suporte a Aplicações de Inteligência Artificial Descentralizada (DzAI) em Sistemas Abertos Distribuídos**. Relatório Trienal de Pesquisa. Bauru: Departamento de Engenharia Elétrica, Faculdade de Engenharia de Bauru, Universidade Estadual Paulista.

STEELS, L. (1990). Cooperation between distributed agents through self-organization. In: EUROPEAN WORKSHOP ON MODELLING AUTONOMOUS AGENTS IN A MULTI-AGENT WORD (MAAMAW-89), 1, 1989, Amsterdam. DEMAZEAU, Y.; MÜLLER, J. P. (Ed.) **Decentralized AI – Proceedings…** Amsterdam: Elsevier. p. 175-196.

SUN MICROSYSTEMS (2004). **Java 2 DK, Standard Edition Documentation Version 1.4.2**. Disponível em: <http://java.sun.com/j2se/1.4.2/download.html#docs>. Acesso em: 8 maio 2004.

SYCARA, K. P. (1988). Resolving goal conflicts via negotiation. In: NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE (AAAI-88), 7, 1988, St. Paul. **Proceedings…** St. Paul. p. 245-250.

SYCARA, K. P. (1990a). Negotiation planning: An AI approach. **European Journal of Operational Research**, v. 46, p. 216-234.

SYCARA, K. P. (1990b). Persuasive argumentation in negotiation. **Theory and Decision**, v. 28, p. 203-242.

TACOMA (2000). **The TACOMA Project Home Page**. Department of Computer Science, University of Tromsø, Norway. Disponível em: <http://www.cs.uit.no/forskning/DOS/ Tacoma/>. Acesso em 27 out. 2000.

THOMSEN, B.; LETH, L.; PRASAD, S.; KUO, T. M.; KRAMER, A.; KNABE, F. C.; GIACALONE, A. (1993). **Facile Antigua Release programming guide**. Tech. Report. ECRC-93-20, European Computer Industry Research Centre, Munich, Germany, Dec. Apud Fuggetta (1998).

TRYLLIAN (2000). **Tryllian's Agent Development Kit (ADK) Home Page**. Tryllian B. V. Disponível em: <http://www.tryllian.com/>. Acesso em: 2 out. 2000.

VAN ROSSUM, G.; DRAKE, F. L. (2000). **Python Tutorial**. PythonLabs. Disponível em <http://www. python.org/doc/current/tut/tut.html>. Acesso em 19 abr. 2000.

W3C (2004). **Extensible Markup Language (XML) 1.1**. W3C Recommendation, 4 Feb. 2004. Disponível em: < http://www.w3.org/TR/2004/REC-xml11-20040204/>. Acesso em: 31 mar. 2004.

WALDO, J. (2001). Mobile Code, Distributed Computing, and Agents. **IEEE Intelligent Systems**, March/April. p.10- 12.

WHITE, J. E. (1997). Telescript Technology: Mobile Agents**.** In: BRADSHAW, J. M. (Ed.) **Software Agents**. Cambridge, MA: The MIT Press. p. 437-472.

WILKINS, D. (1988). **Practical Planning: Extending the Classical AI Planning Paradigm**. San Mateo, CA: Morgan Kaufmann.

WONG, D., PACIOREK, N., MOORE, D. (1999). Java-based Mobile Agents. **Communications of the ACM,** v. 2, n. 3, p. 92-102, March.

WOOLDRIDGE, M.; JENNINGS, N. R. (Ed.) (1995). Intelligent Agents: Theories, Architectures, and Languages**. Lecture Notes in Artificial Intelligence**, Berlin Heidelberg: Springer-Verlag, n. 890, p. 261-276.

ZENG, D.; SYCARA, K. (1997). Benefits of learning in negotiation. In: NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE (AAAI97), 14, 1997, Rhode Island, 1997. **Proceedings…** Rhode Island. p. 36-41.

ZENG, D.; SYCARA, K. (1998). Bayesian learning in negotiation. **International Journal of Human-Computer Studies**, n. 48, p. 125-141.

Apêndice A

Diagramas de Interação para o Subsistema Rete

A.1 Diagramas de Interação para `ReteCompiler`

Nas subseções seguintes são apresentados os diagramas de interação para as operações introduzidas na Figura 85. Nesses diagramas, as notas explicativas têm por objetivo somente esclarecer como os diversos laços de iterações estão aninhados e não ilustrar como o código foi realmente implementado.

A.1.1 Operação `reorganize(…)`

A operação `ReteCompiler::reorganize(…)` é responsável por reordenar os literais presentes na expressão antecedente de uma regra, colocando-os na seguinte ordem: literais puros, sensores e igualdades. A Figura 115 mostra o diagrama de interação para o método que implementa essa operação.

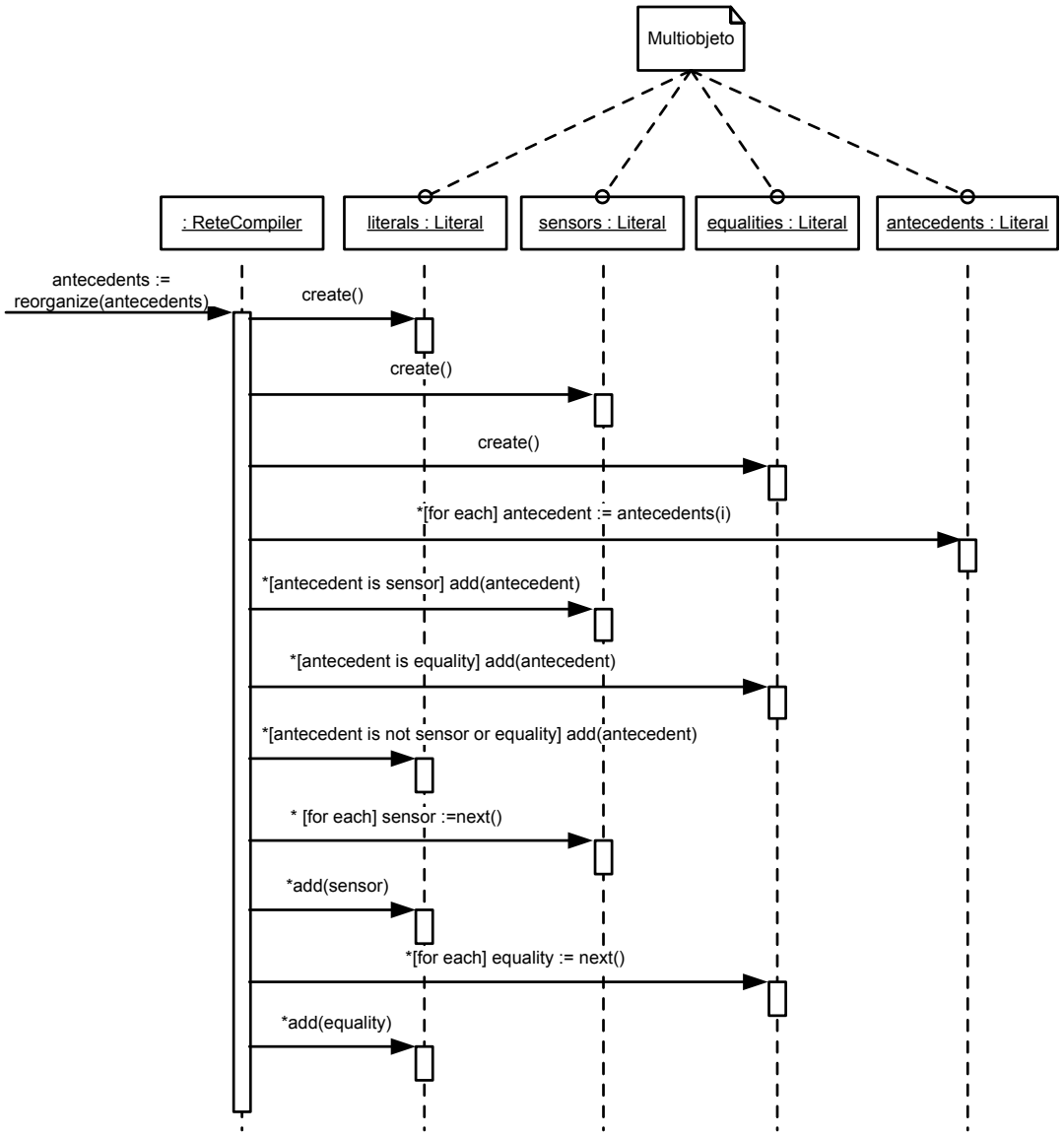


Figura 115: Diagrama de interação para `ReteCompiler::reorganize()`

A.1.2 Operação `createOneInputNodes(…)`

A operação `ReteCompiler::createOneInputNodes(…)` é responsável pela criação de todos os nós-de-uma-entrada presentes na rede de nós que representa as regras de uma base de conhecimento. A Figura 116 mostra o diagrama de interação para o método que implementa essa operação.

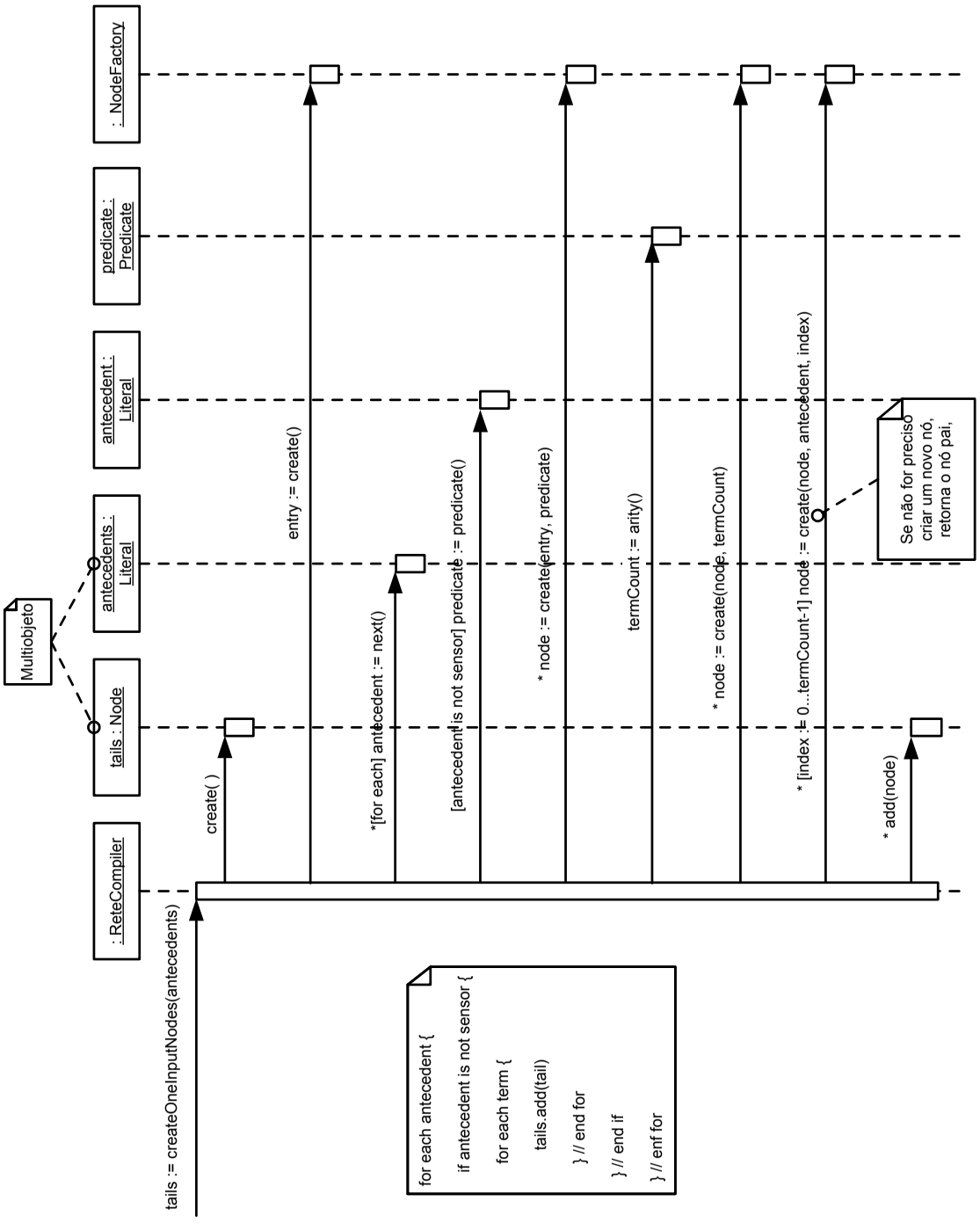


Figura 116: Diagrama de interação para `ReteCompiler::createOneInputNodes()`

A.1.3 Operação `createTwoInputNodes(…)`

A operação `ReteCompiler::createTwoInputNodes(…)` é responsável pela criação de todos os nós-de-duas-entradas presentes na rede de nós que representa as regras de uma base de conhecimento. A Figura 117 mostra o diagrama de interação para o método que implementa essa operação.

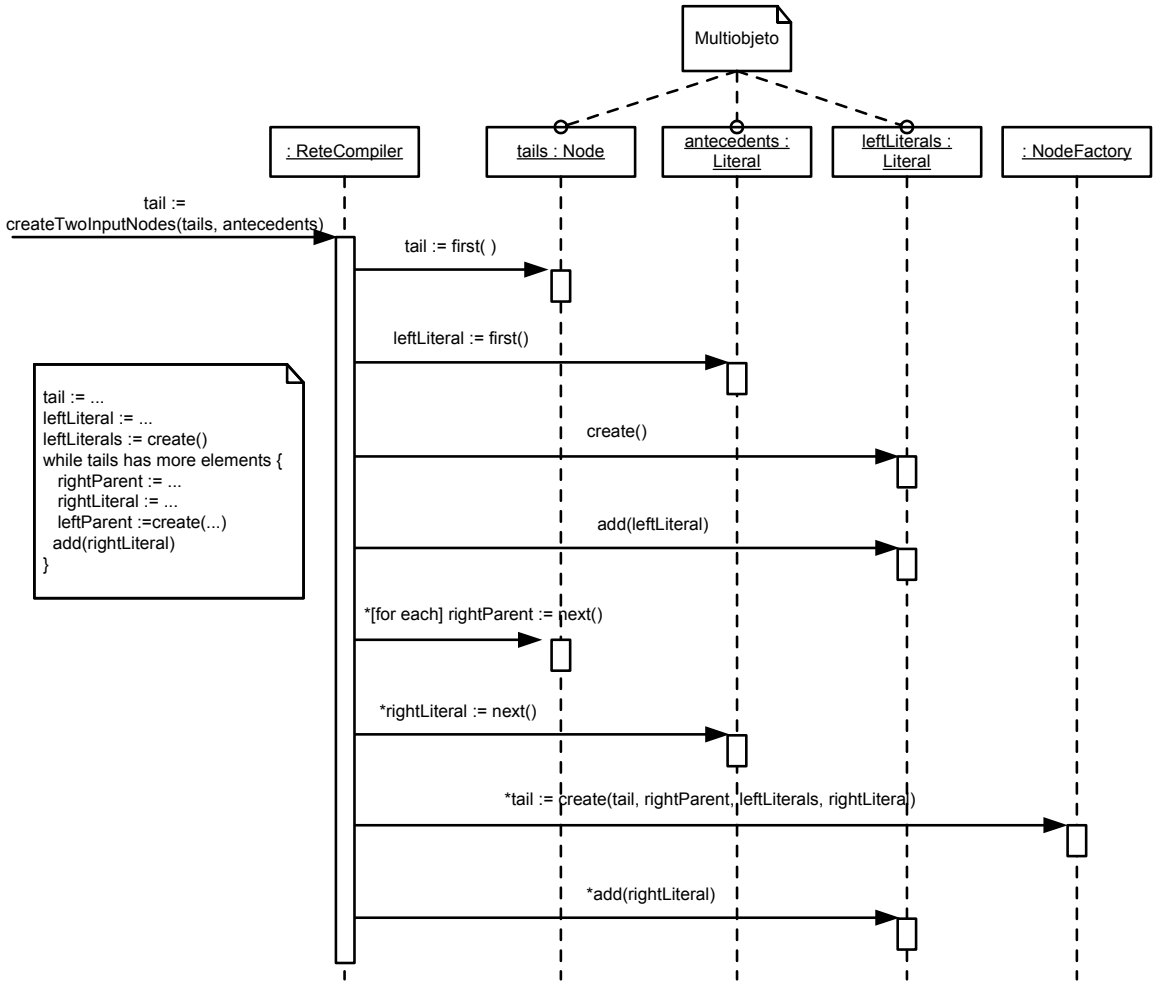


Figura 117: Diagrama de interação para `ReteCompiler::createTwoInputNodes()`

A.1.4 Operação `createSensorNodes(…)`

A operação `ReteCompiler::createSensorNodes(…)` é responsável pela criação de todos os nós-sensores presentes na rede de nós que representa as regras de uma base de

conhecimento. A Figura 118 mostra o diagrama de interação para o método que implementa essa operação.

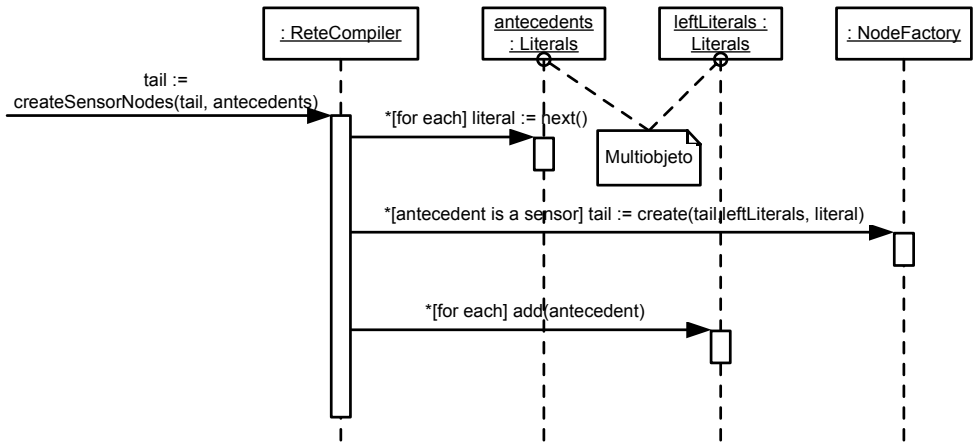


Figura 118: Diagrama de interação para `ReteCompiler::createSensorNodes()`

A.1.5 Operação `createEqualityNode(…)`

A operação `ReteCompiler::createEqualityNode(…)` é responsável pela criação de todos os nós responsáveis por testarem as igualdades/desigualdades entre variáveis intra e interliterais presentes na rede de nós que representa as regras de uma base de conhecimento. A Figura 119 mostra o diagrama de interação para o método que implementa essa operação.

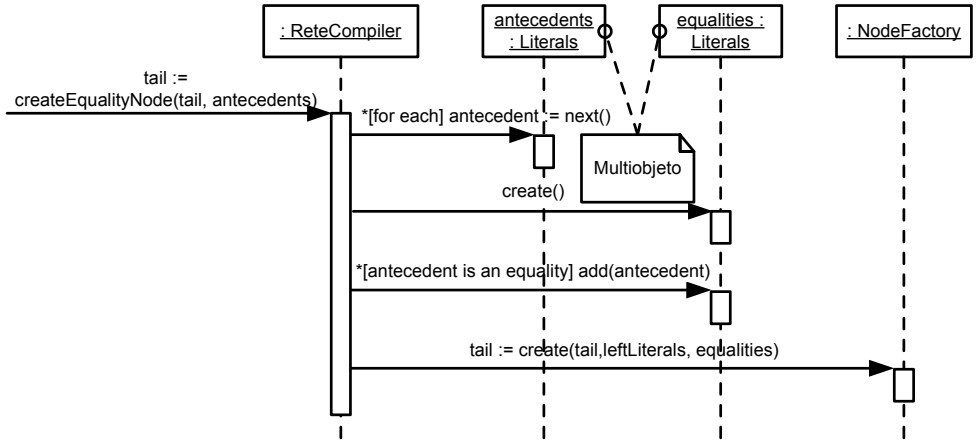


Figura 119: Diagrama de interação para `ReteCompiler::createEqualityNode()`

A.1.6 Operação `createTerminalNodes(…)`

A operação `ReteCompiler::createTerminalNodes(…)` é responsável pela criação de todos os nós-terminais presentes na rede de nós que representa as regras de uma base de conhecimento. A Figura 120 mostra o diagrama de interação para o método que implementa essa operação.

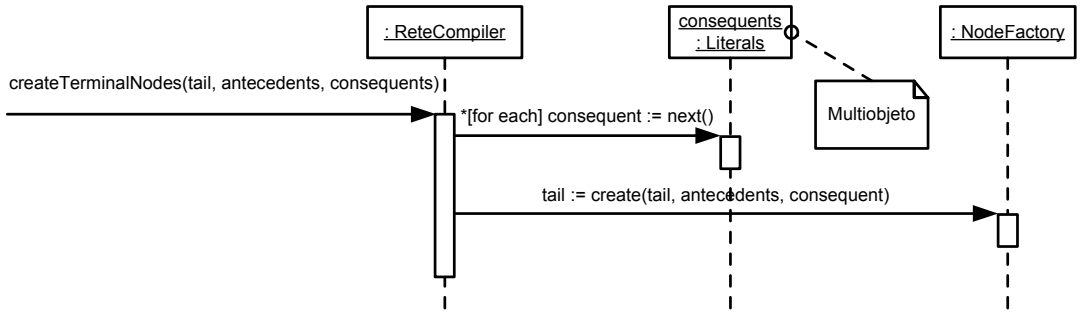


Figura 120: Diagrama de interação para `ReteCompiler::createTerminalNodes()`

A.2 Diagramas de Interação para `NodeFactory`

A.2.1 Operação `create()`

A operação pública `NodeFactory::create()` é responsável pela criação do único nó de entrada (raiz) da estrutura de nós que representa as regras de uma base de conhecimento. Ao ser invocada, retorna uma instância de `Node`.

A.2.2 Operação `create(parent,predicate)`

A operação pública `NodeFactory::create(parent,predicate)` é responsável pela criação dos nós-de-uma-entrada que testam os predicados dos literais apresentados à rede, durante a fase de casamento, para adição ou remoção da *memória de trabalho*. A Figura 121 mostra o diagrama de interação para o método que implementa essa operação.

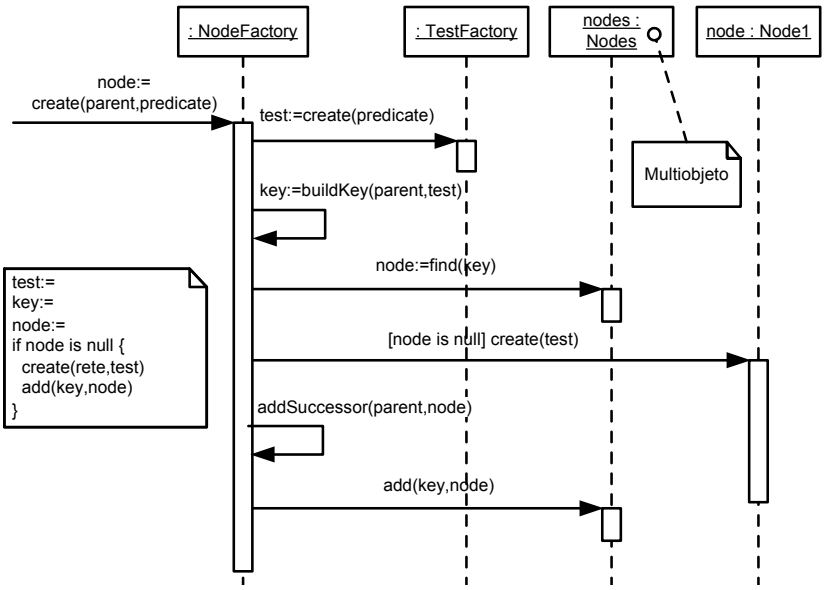


Figura 121: Diagrama de interação para `NodeFactory::create(parent,predicate)`

A.2.3 Operação `create(parent,size)`

A operação pública `NodeFactory::create(parent,size)` é responsável pela criação dos nós-de-uma-entrada que testam o número de termos dos literais apresentados à rede, durante a fase de casamento, para adição ou remoção da *memória de trabalho*. A Figura 122 mostra o diagrama de interação para o método que implementa essa operação.

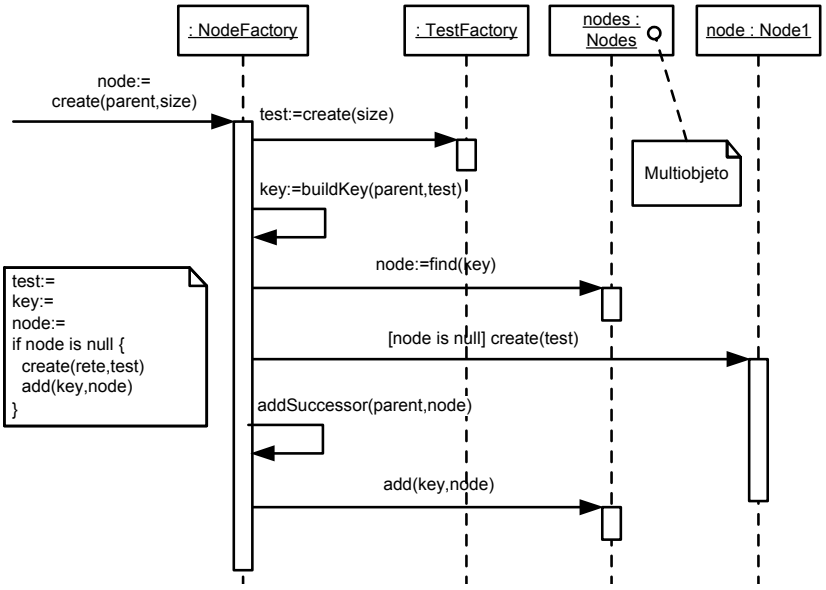


Figura 122: Diagrama de interação para `NodeFactory::create(parent,size)`

A.2.4 Operação `create(parent,literal,termIndex)`

A operação pública `NodeFactory::create(parent,literal,termIndex)` é responsável pela criação dos nós-de-uma-entrada que testam as características *intraliteral* (constantes e variáveis) dos literais apresentados à rede, durante a fase de casamento, para adição ou remoção da *memória de trabalho*. A Figura 123 mostra o diagrama de interação para o método que implementa essa operação.

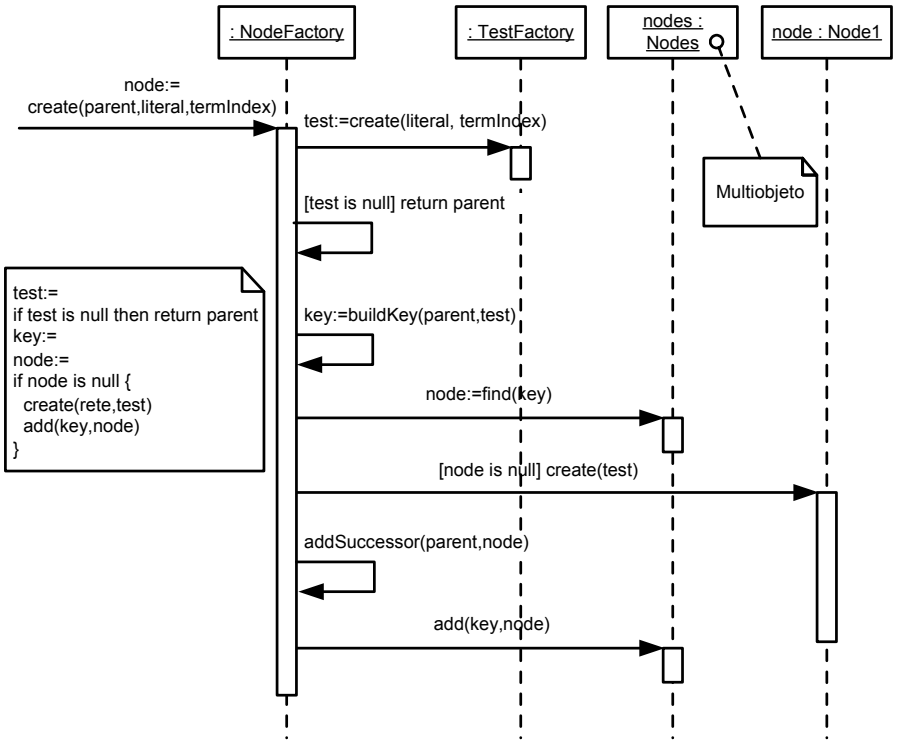


Figura 123: Diagrama de interação para `NodeFactory::create(parent,literal,termIndex)`

Alguns comentários sobre esse diagrama são importantes. Se nenhum teste for necessário para o termo em questão, nenhum novo nó será criado e o parâmetro de retorno da operação será o próprio nó pai. O parâmetro de retorno (`node`) é do tipo abstrato `Node`, do qual todos os nós são subclasses.

A.2.5 Operação `create(leftParent,rightParent,leftLiterals, rightLiteral)`

A operação pública `NodeFactory::create(parent,leftParent,right-Parent,antecedents,literal)` é responsável pela criação dos nós-de-duas-entradas que testam as características *interliterais* (constantes e variáveis) dos literais apresentados à rede, durante a fase de casamento, para adição ou remoção da *memória de trabalho*. A Figura 124 mostra o diagrama de interação para o método que implementa essa operação.

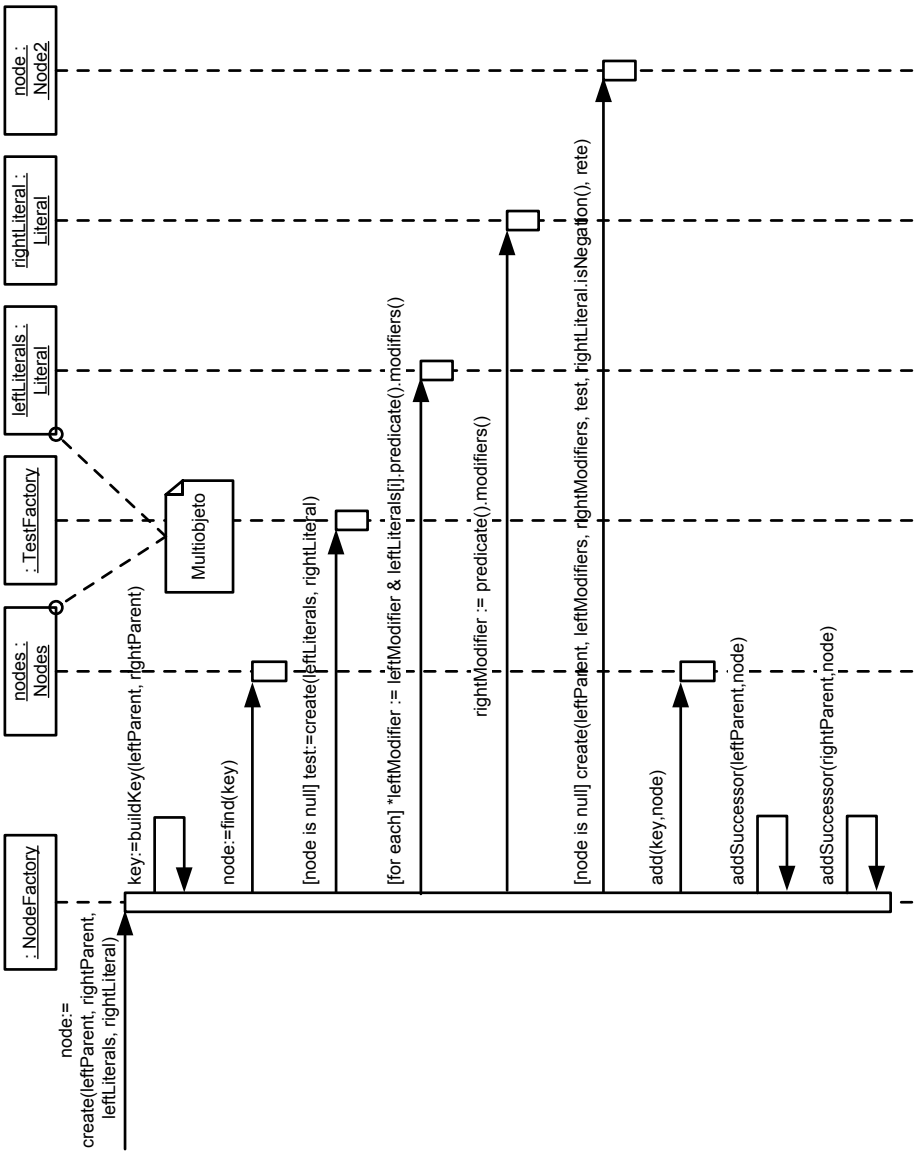


Figura 124: Diagrama de interação para `NodeFactory::create(leftParent,rightParent,leftLiterals, rightLiteral)`

A.2.6 Operação `create(parent,leftLiterals,equalities)`

A operação pública `NodeFactory::create(parent,leftParent,equalities)` é responsável pela criação dos nós-de-uma-entrada que testam as igualdades/desigualdades entre variáveis intra e interliterais dos literais apresentados à rede, durante a fase de casamento, para adição ou remoção da *memória de trabalho*. A Figura 125 mostra o diagrama de interação para o método que implementa essa operação.

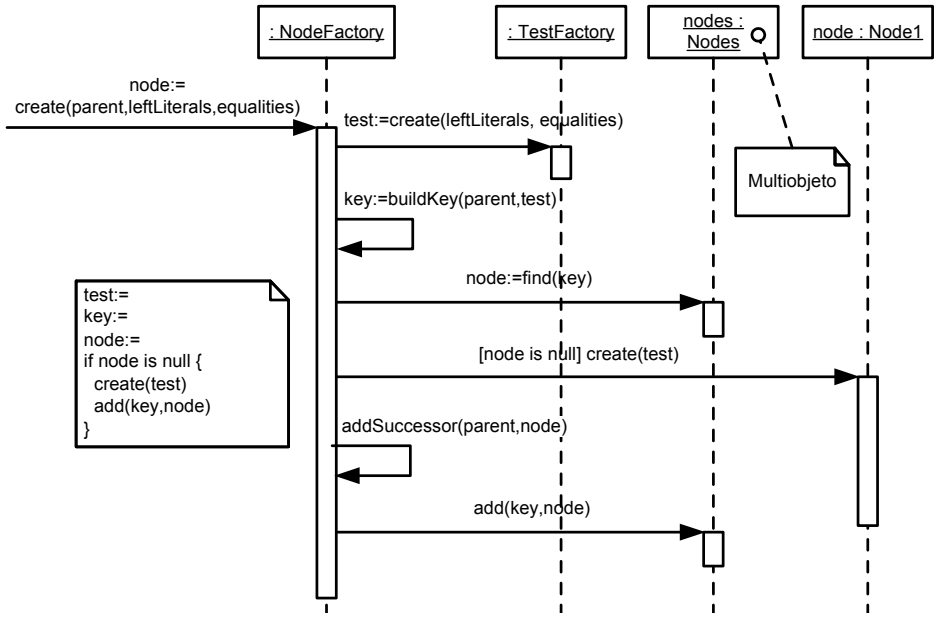


Figura 125: Diagrama de interação para `NodeFactory::create(parent,leftLiterals,equalities)`

A.2.7 Operação `create(parent,leftLiterals,literal)`

A operação pública `NodeFactory::create(parent,leftParent,literal)` é responsável pela criação dos nós sensores e nós terminais. Se o parâmetro de entrada literal for um sensor, será criada uma instância da classe `SensorNode`, caso contrário, a instância criada será da classe `Terminal`. A Figura 126 mostra o diagrama de interação para o método que implementa essa operação.

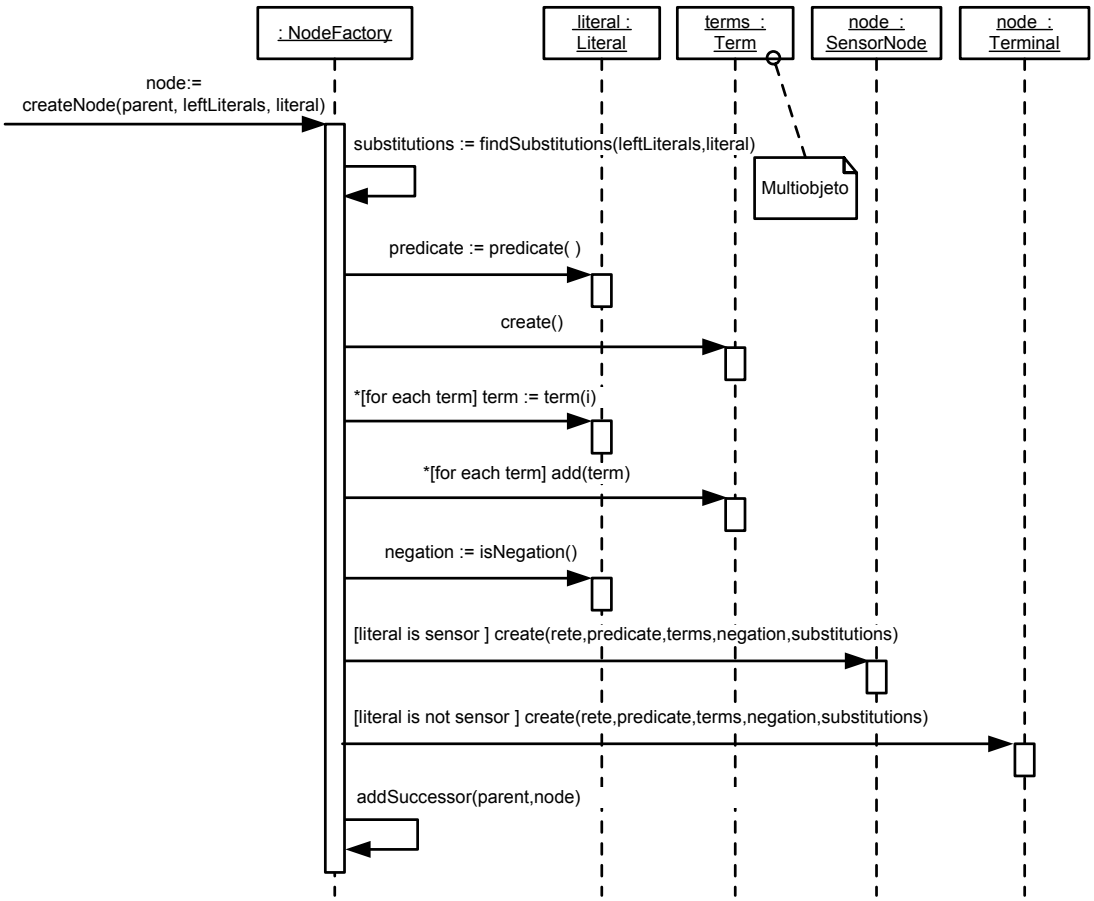


Figura 126: Diagrama de interação para `NodeFactory::create(parent,leftLiterals,literal)`

Preferiu-se sobrecarregar os construtores de `SensorNode` e `Terminal` com vários parâmetros de entrada do que passar somente o parâmetro `literal` como entrada e derivar os elementos `predicate`, `terms` e `negation` no "corpo" de seus construtores. Com isso, os códigos das classes `SensorNode` e `Terminal` são reduzidos, o que é importante.

A.3 Diagramas de Interação para `TestFactory`

Todas as operações `TestFactory::create(…)` retornam um objeto que implementam a interface `Test`. Essa interface oferece somente a operação `test(sentence)`, necessária durante a fase de *casamento*, cujo parâmetro de retorno booleano apresenta o valor `true` se a sentença de entrada (`sentence`) passar pelo teste que o objeto representa.

A.3.1 Operação `create(predicate)`

A operação pública `TestFactory::create(predicate)` é responsável pela criação dos objetos que testam os predicados dos literais apresentados à estrutura de nós durante a fase de casamento do algoritmo *Rete*. A Figura 127 mostra o diagrama de interação para o método que implementa essa operação.

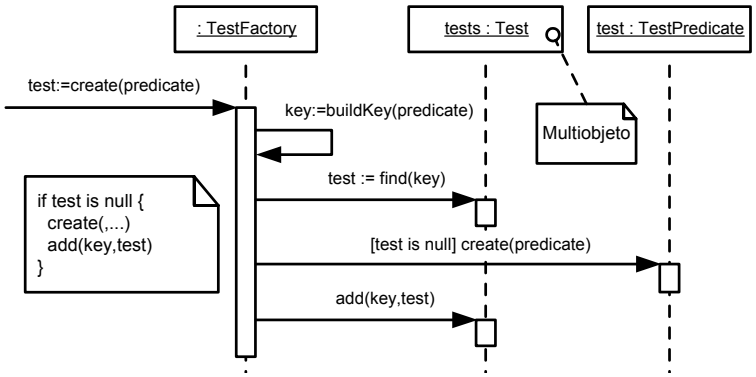


Figura 127: Diagrama de interação para `TestFactory::create(predicate)`

A.3.2 Operação `create(size)`

A operação pública `TestFactory::create(size)` é responsável pela criação dos objetos que testam o número de termos dos literais apresentados à estrutura de nós durante a fase de casamento do algoritmo *Rete*. A Figura 128 mostra o diagrama de interação para o método que implementa essa operação.

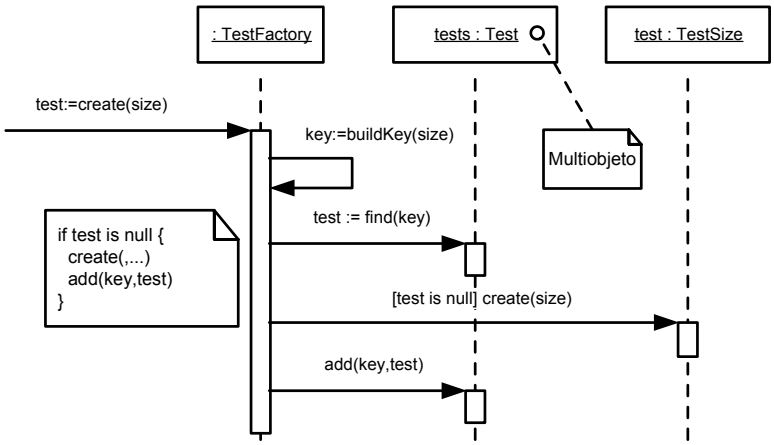


Figura 128: Diagrama de interação para `TestFactory::create(size)`

A.3.3 Operação `create(literal,termIndex)`

A operação pública `TestFactory::create(size)` é responsável pela criação dos objetos que testam a presença de constantes e variáveis nos literais apresentados à estrutura de nós durante a fase de casamento do algoritmo *Rete*. A Figura 129 mostra o diagrama de interação para o método que implementa essa operação.

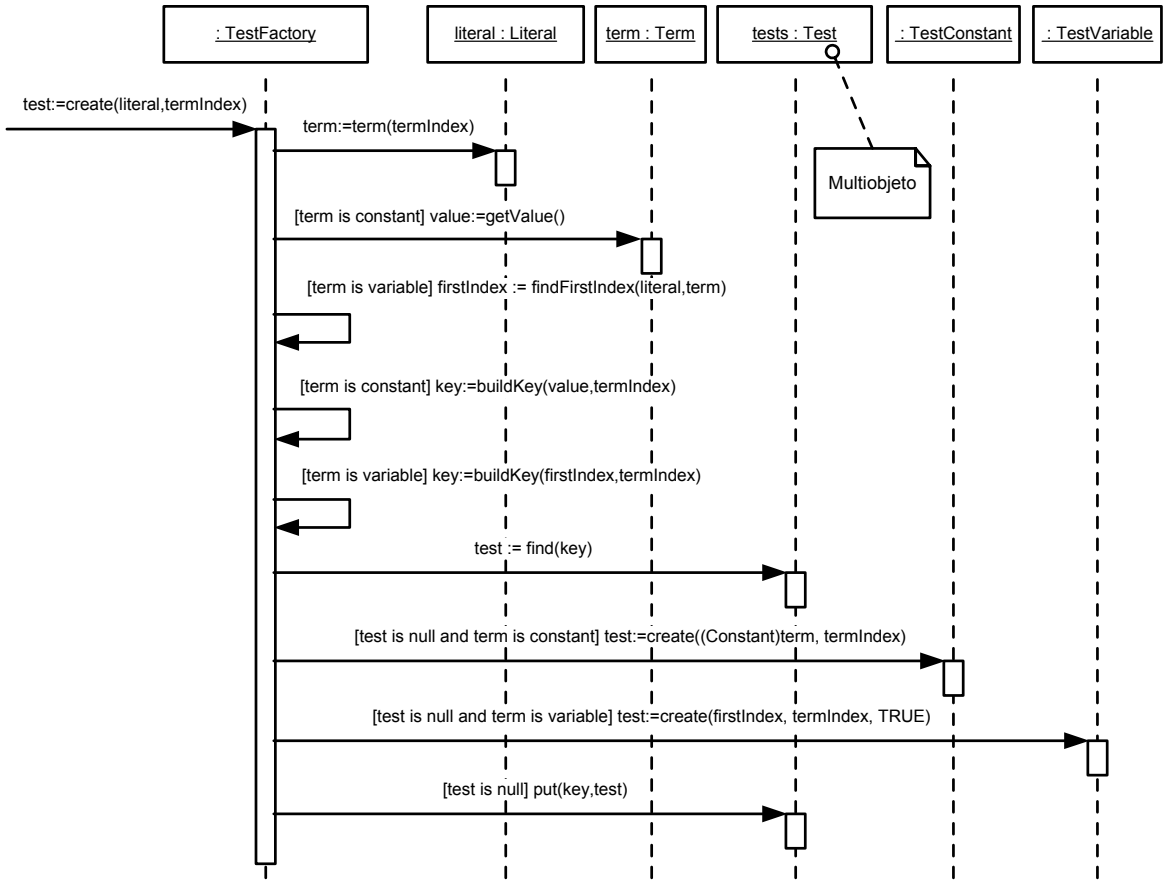


Figura 129: Diagrama de interação para `TestFactory::create(literal,termIndex)`

A.3.4 Operação `create(constant,termIndex)`

A operação privativa `TestFactory::create(value,termIndex)` é responsável pela criação dos objetos que testam a presença de uma constante específica em uma dada posição da lista de termos dos literais apresentados à estrutura de nós durante a fase de casamen-

to do algoritmo *Rete*. A Figura 130 mostra o diagrama de interação para o método que implementa essa operação.

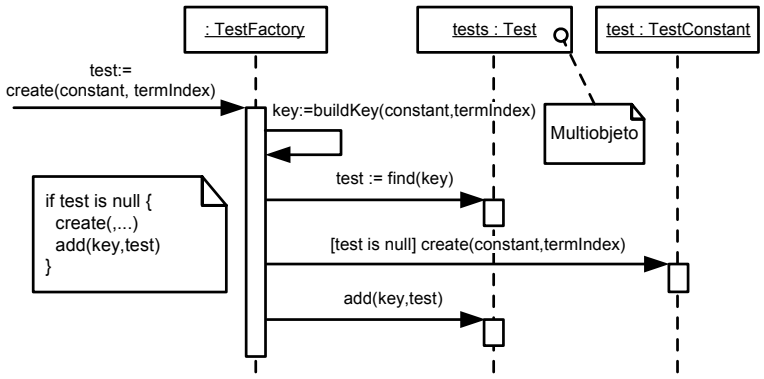


Figura 130: Diagrama de interação para `TestFactory::createTest(constant,termIndex)`

A.3.5 Operação `create(firstIndex,index,equality)`

A operação privativa `TestFactory::create(firstIndex,index, equality)` é responsável pela criação dos objetos que testam a igualdade/desigualdade entre as variáveis presentes em literais distintos ou não, apresentados à estrutura de nós durante a fase de casamento do algoritmo *Rete*. A Figura 131 mostra o diagrama de interação para o método que implementa essa operação.

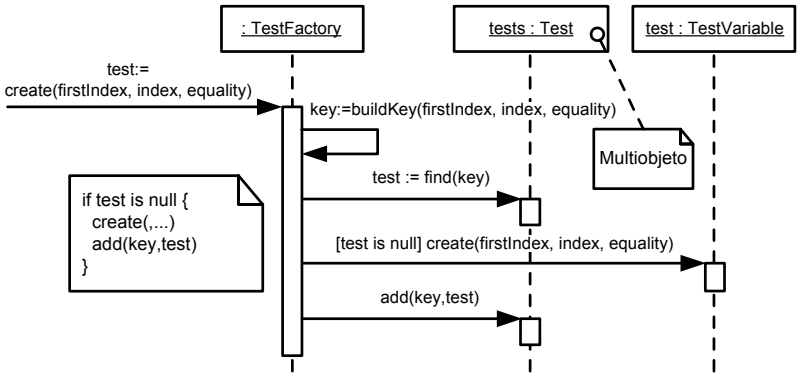


Figura 131: Diagrama de interação para `TestFactory::create(firstIndex,index,equality)`

A.3.6 Operação `create(leftLiterals,rightLiteral)`

A operação pública `TestFactory::create(leftLiterals, rightLiteral)` é responsável pela criação dos objetos compostos que realizam todos os testes aplicados pelos nós-de-duas-entradas, durante a fase de casamento do algoritmo *Rete*, entre as novas sentenças a serem armazenadas ou removidas da *memória de trabalho* e as já armazenadas. A Figura 132 mostra o diagrama de interação para o método que implementa essa operação.

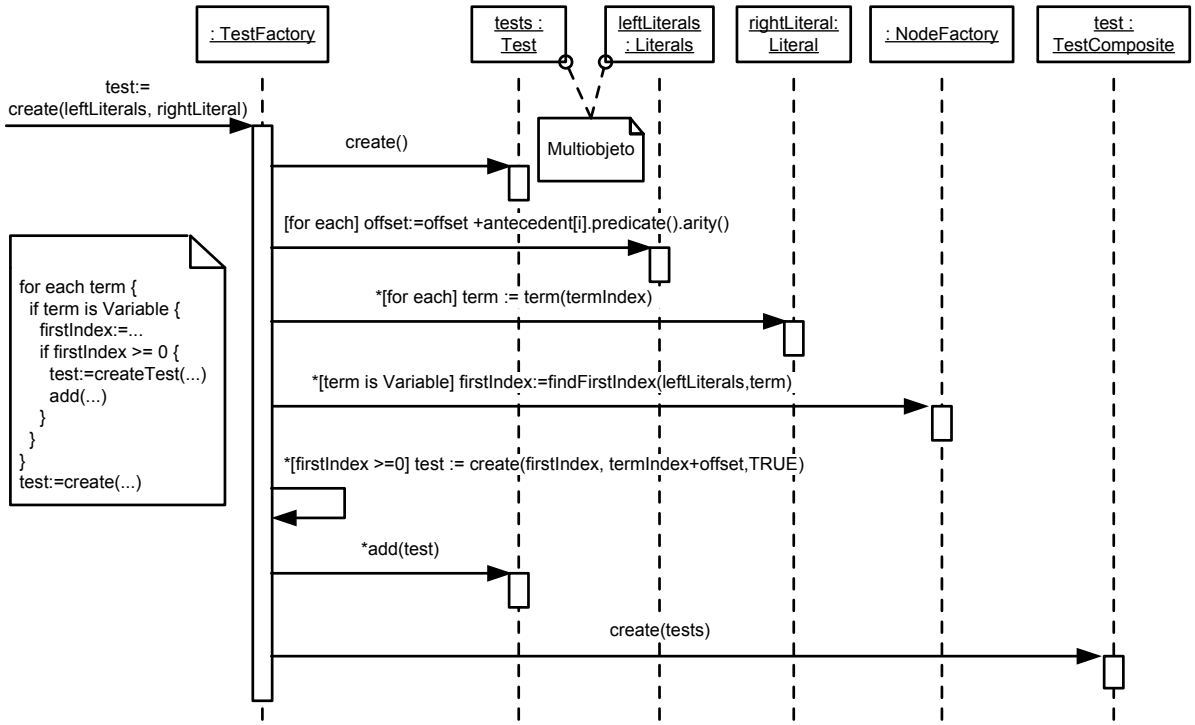


Figura 132: Diagrama de interação para `TestFactory::create(leftLiterals,rightLiteral)`

A.3.7 Operação `create(leftLiterals,equalities)`

A operação pública `TestFactory::create(leftParent,equalities)` é responsável pela criação dos objetos compostos que testam, durante a fase de casamento do algoritmo *Rete*, as igualdades/desigualdades entre variáveis intra e interliterais especificadas pelo predicado reservado `EQUAL`. A Figura 133 mostra o diagrama de interação para o método que implementa essa operação.

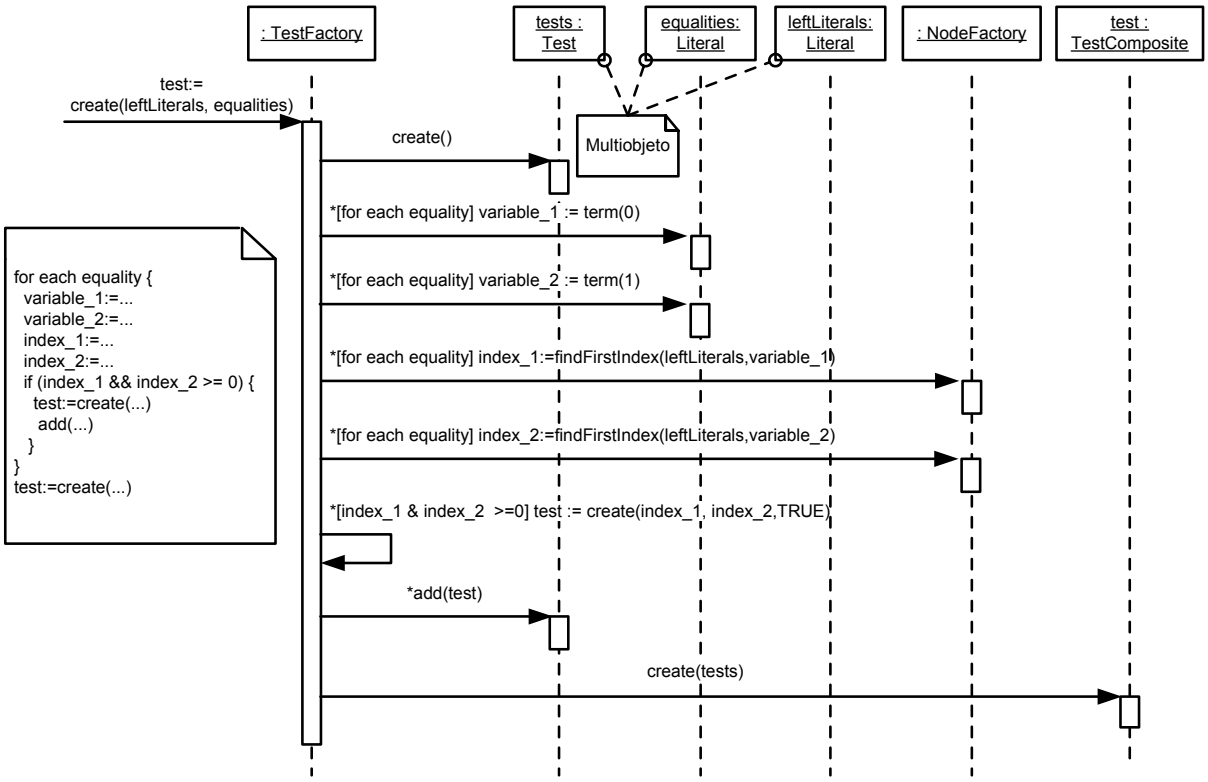


Figura 133: Diagrama de interação para `TestFactory::create(leftLiterals,equalities)`

A.4 Diagramas de Interação para `Node`

Os processamentos realizados por `addSentence(…)` e `removeSentence(…)` são específicos para cada tipo de nó. `Node` implementa somente os comportamentos por omissão, que devem ser especializados pelas suas subclasses.

A.4.1 Operação `addSentence(…)`

A operação `addSentence()` simplesmente envia seu argumento de entrada (`ground_sentence`) aos seus nós sucessores. O argumento `ground_sentence` pode ser tanto um único literal fechado (`Atom`) quanto uma conjunção de literais fechados (`Conjunction`). A Figura 134 ilustra esse comportamento.

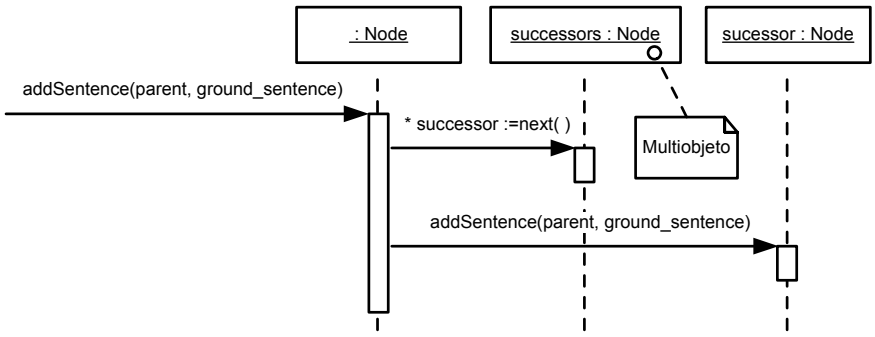


Figura 134. Diagrama de interação para `Node::addSentence()`

A.4.2 Operação `removeSentence(…)`

A operação `removeSentence()` simplesmente envia seu argumento de entrada (`ground_sentence`) aos seus nós sucessores. O argumento `ground_sentence` pode ser tanto um único literal fechado (`Atom`) quanto uma conjunção de literais fechados (`Conjunction`). A Figura 135 ilustra esse comportamento.

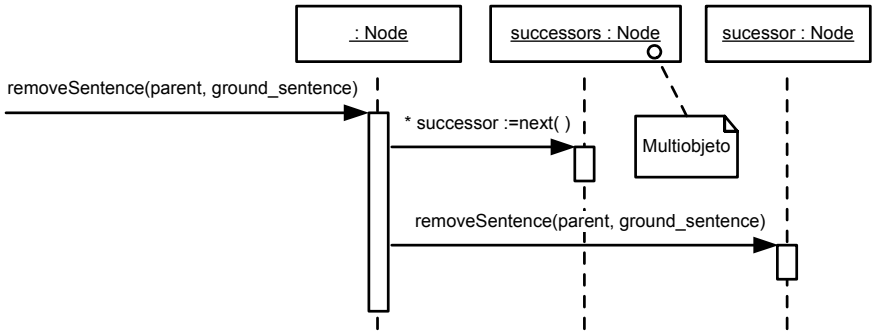


Figura 135. Diagrama de interação para `Node::removeSentence()`

A.5 Diagramas de Interação para `Node1`

As subseções seguintes ilustram os comportamentos de `addSentence(…)` e `removeSentence(…)` para os nós de uma entrada (`Node1`). Os diagramas apresentados nes-

sas subseções ilustram o fato de que os nós-de-uma-entrada somente enviam para seus sucessores sentenças fechadas que tenham passado nos seus respectivos testes.

A.5.1 Operação `addSentence(…)`

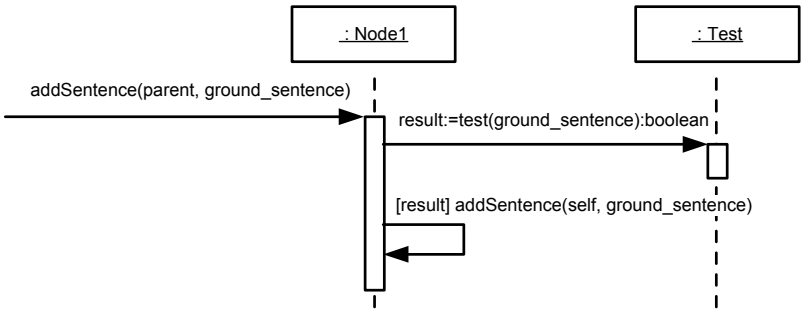


Figura 136: Diagrama de interação para `Node1::addSentence()`

A.5.2 Operação `removeSentence(…)`

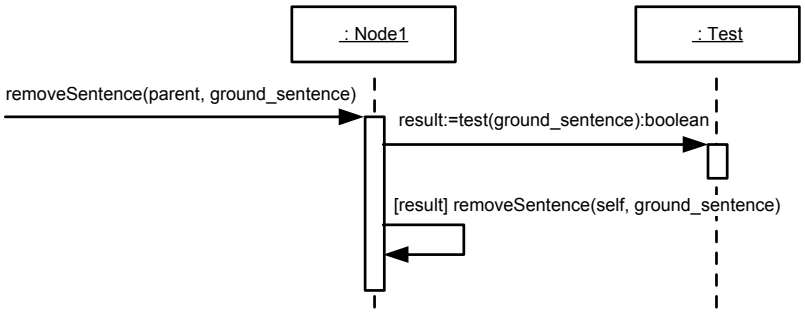


Figura 137: Diagrama de interação para `Node1::removeSentence()`

A.6 Diagramas de Interação para `Node2`

Para os nós-de-duas-entradas (`Node2`), os comportamentos de `addSentence(…)` e `removeSentence(…)` são mais complexos que para os nós-de-uma-entrada. O tratamento de cada sentença recebida por esses nós depende do nó antecessor remetente (à esquerda -

`leftParent` - ou à direita - `rightParent`), do tipo de ação pretendida (adição ou remoção da sentença) e do tipo de literal que ele representa (positivo ou negativo). Essa última condição é testada no corpo das operações `add(…)` e `remove(…)` das instâncias de `LeftMemory` e `RightMemory`.

A.6.1 Operação `addSentence(…)`

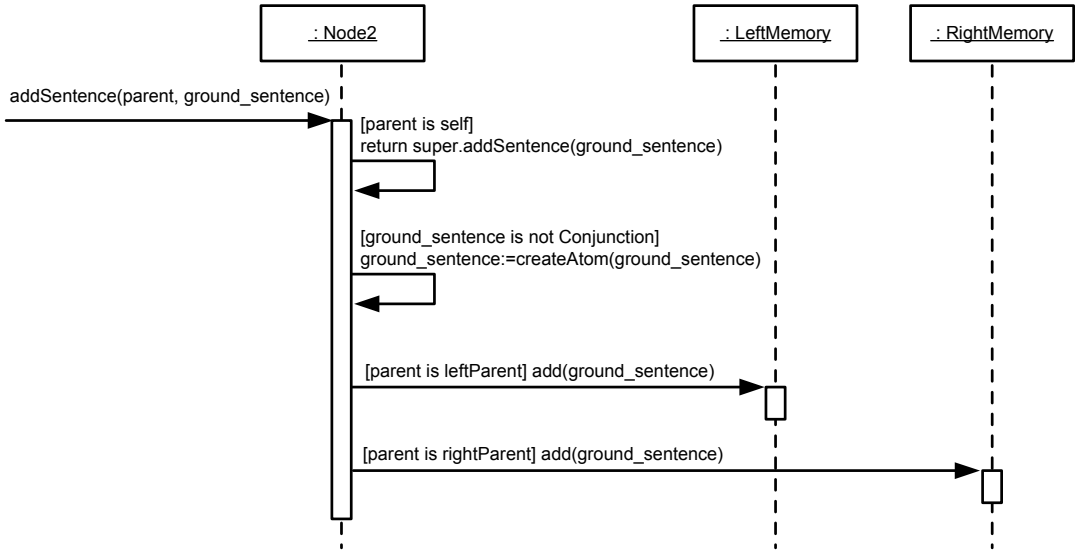


Figura 138: Diagrama de interação para `Node2::addSentence()`

A.6.2 Operação `removeSentence(…)`

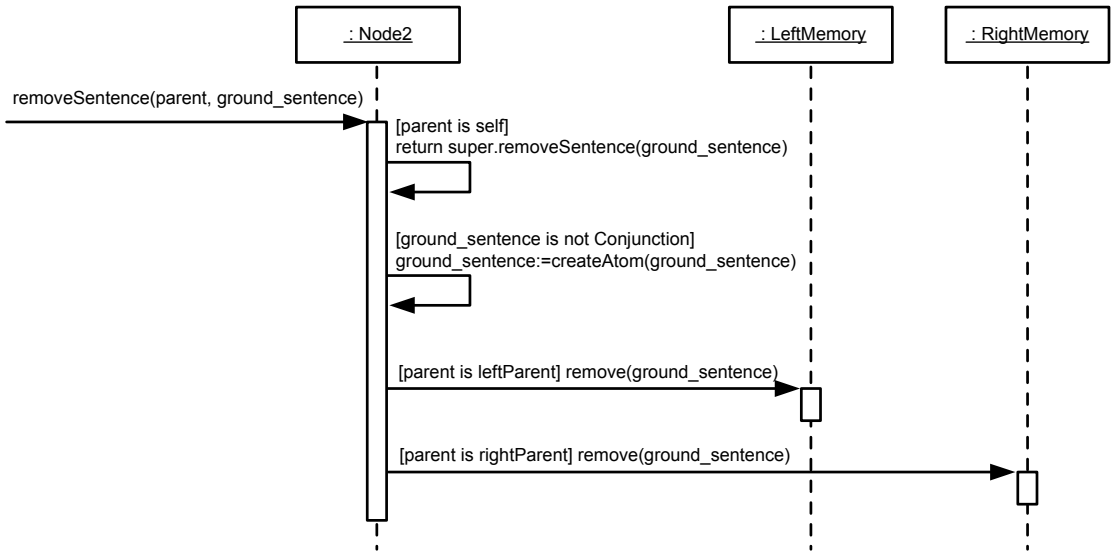


Figura 139: Diagrama de interação para `Node2::removeSentence()`

A.7 Diagramas de Interação para `LeftMemory`

A.7.1 Operação `add(…)`

A Figura 140 ilustra o caso em que o nó antecessor a esquerda envia uma sentenças para adição.

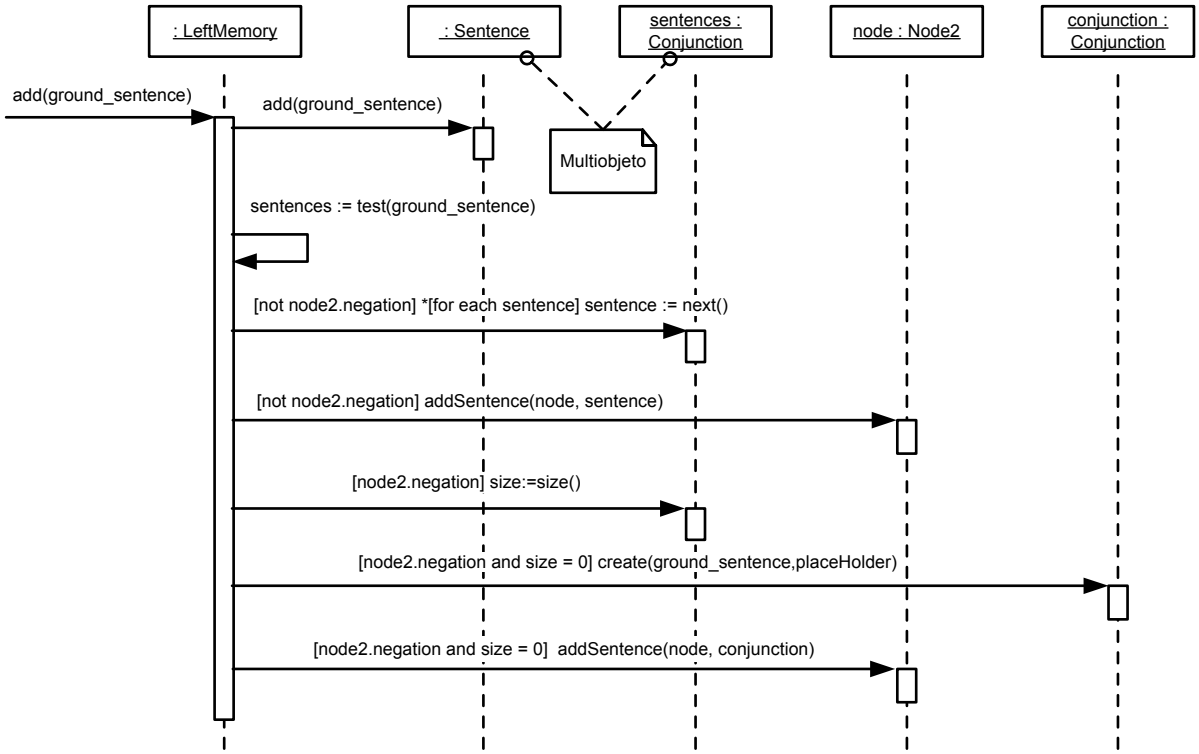


Figura 140: Diagrama de interação para `LeftMemory::add()`

Sentenças recebidas do antecessor à esquerda (`leftParent`) são armazenadas na memória esquerda do nó (`leftMemory`) e testadas contra todas as sentenças recebidas anteriormente do antecessor à direita, armazenadas na memória direita (`rightMemory`). Isso é realizado pelo método privativo `LeftMemory::test()`. Esse método retorna uma coleção de conjunções (`sentences`) constituídas pela sentença esquerda (`ground_sentence`) e por cada uma das sentenças presentes na memória direita que passaram em todos os testes de responsabilidade do nó em questão. Caso o literal correspondente à sentença direita seja positivo, cada sentença dessa coleção (`sentence`) é enviada a todos os nós sucessores para adição.

Caso o literal correspondente à sentença direita seja negativo o comportamento será um pouco mais complexo devido ao mecanismo de *negação por falha*. Se a sentença adicionada à memória esquerda não passar nos testes com nenhuma das sentenças presentes na memória direita (`size = 0`) a condição de negação da sentença direita é satisfeita. Nessa situação uma única conjunção, composta pela sentença esquerda (`ground_sentence`) e por um "marcador" (`placeHolder`), é adicionada aos nós sucessores. O "marcador" tem por função indicar que a negação foi satisfeita.

A.7.2 Operação `remove(...)`

A Figura 141 ilustra o caso em que o nó antecessor a esquerda envia uma sentenças para remoção.

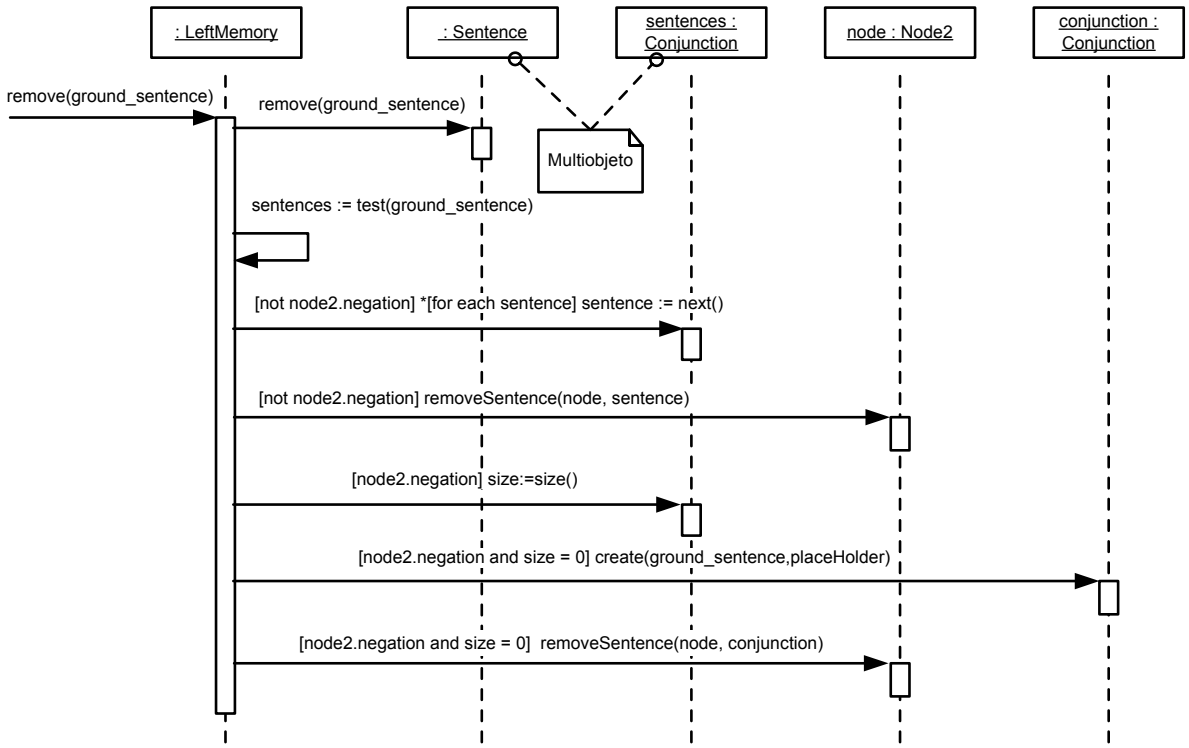


Figura 141: Diagrama de interação para `LeftMemory::remove()`

A sentença recebida do antecessor a esquerda é removida da memória e testada contra todas as sentenças recebidas anteriormente do antecessor à direita. Caso o literal correspondente a sentença direita seja positivo, toda conjunção que passou nos testes do nó é envia-

da a todos os sucessores para remoção. Caso o literal seja negativo e a sentença removida não passar em nenhum dos testes (`size = 0`), a condição de negação da sentença direita é satisfeita, indicando que uma conjunção composta pela sentença esquerda e pelo "marcador" `negation` foi adicionada anteriormente aos nós sucessores. Essa conjunção é então removida de todos eles.

A.8 Diagramas de Interação para `RightMemory`

A.8.1 Operação `add(…)`

A Figura 142 ilustra o caso em que o nó antecessor a direita envia uma sentenças para adição.

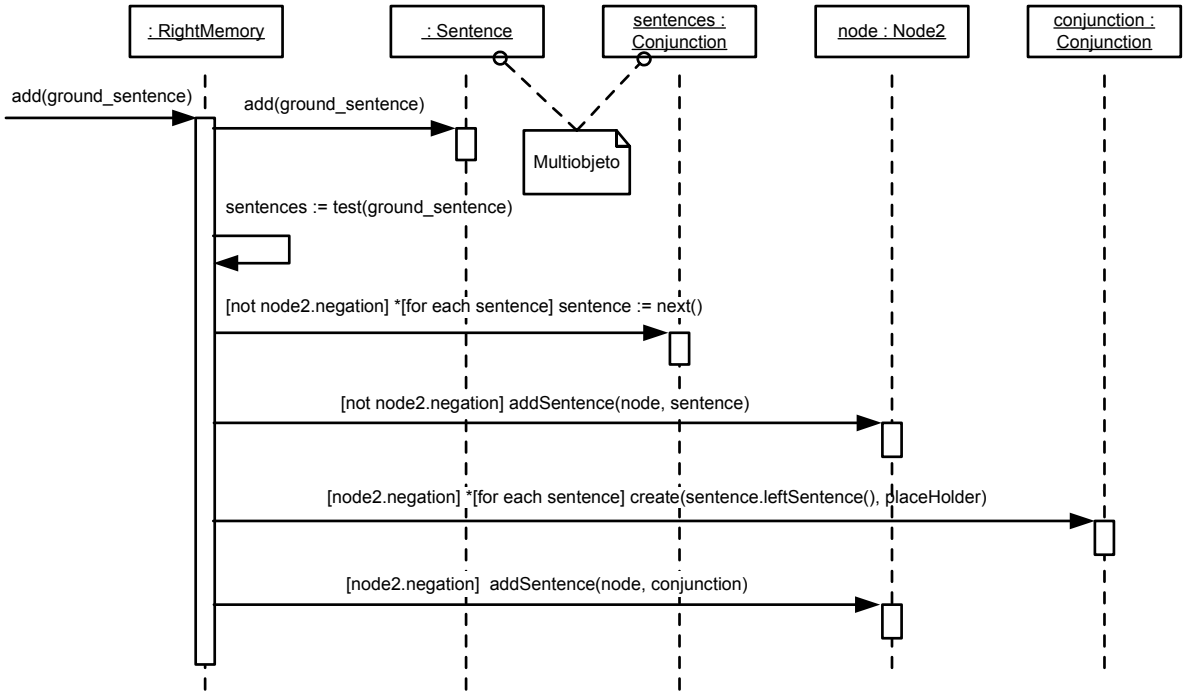


Figura 142: Diagrama de interação para `RightMemory::add()`

Sentenças recebidas do antecessor à direita (`rightParent`) são armazenadas na memória direita do nó (`rightMemory`) e testadas contra todas as sentenças recebidas anteri-

ormente do antecessor à esquerda, armazenadas na memória esquerda (`leftMemory`). Isso é realizado pelo método privativo `RightMemory::test()`. Esse método retorna uma coleção com todas as conjunções (`sentences`) constituídas por cada uma das sentenças presentes na memória esquerda que passaram em todos os testes de responsabilidade do nó em questão e pela sentença direita (`ground_sentence`). Caso o literal correspondente ao antecessor a direita seja positivo, cada sentença dessa coleção (`sentence`) é enviada a todos os nós sucessores para adição. Caso contrário, se o literal correspondente ao antecessor a direita for negativo, toda conjunção que passar nos testes do nó corresponde a uma sentença para a qual a condição de negação por falha da sentença direita não é satisfeita. Nessa situação, as conjunções compostas por cada uma dessas sentenças (`leftSentence`) com o "marcador" (`placeHolder`) devem ser removidas dos nós sucessores.

A.8.2 Operação `remove(...)`

A Figura 143 ilustra o caso em que o nó antecessor a direita envia uma sentenças para remoção. A sentença recebida do antecessor a direita é removida da memória e testada contra todas as sentenças recebidas anteriormente do antecessor à esquerda. Caso o literal correspondente à sentença direita seja positivo, toda conjunção que passou nos testes do nó é enviada a todos os sucessores para remoção. Caso o literal seja negativo, para toda sentença a esquerda que não passar nos testes com as demais sentenças da memória direita, uma conjunção composta por ela e pelo "marcador" `placeHolder` é adicionada aos nós sucessores pois, nesse caso, a sentença removida era a única que impedia a negação por falha.

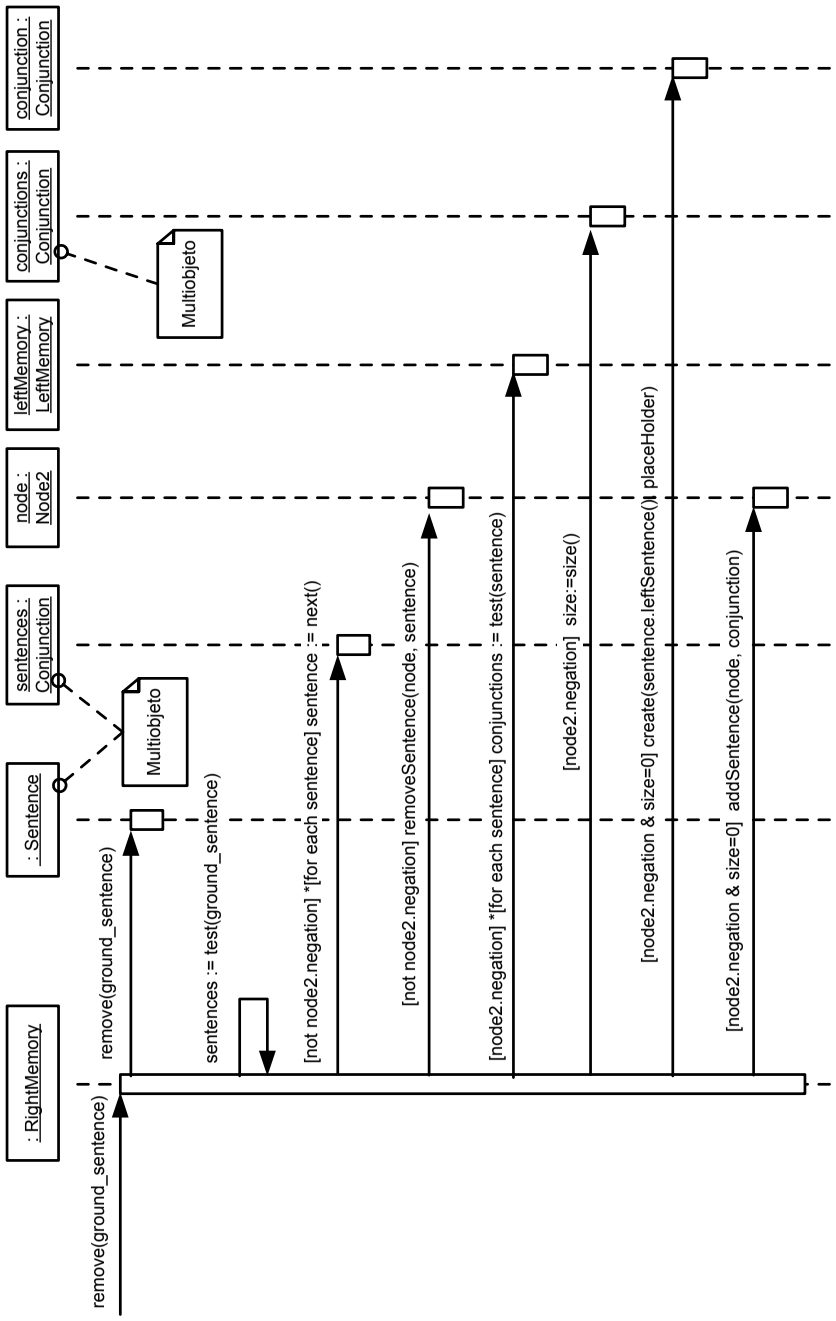


Figura 143: Diagrama de interação para `RightMemory::remove()`

A.9 Diagramas de Interação para `SensorNode`

A classe `SensorNode` somente especializa a operação `Node::ddSentence()`.

A.9.1 Operação `addSentence(…)`

A Figura 144 mostra o diagrama de interação para a especialização da operação `SensorNode::addSentence(…)`.

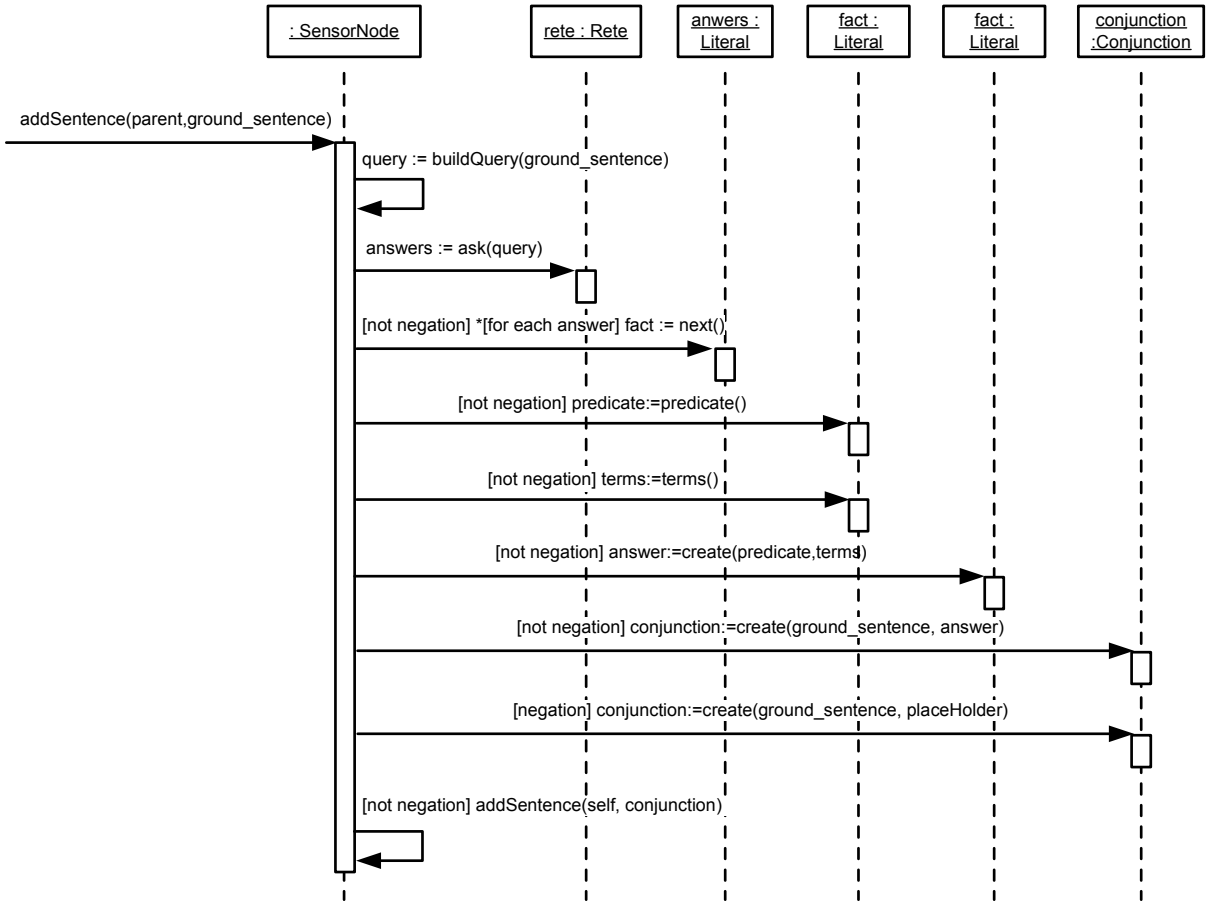


Figura 144: Diagrama de interação para `SensorNode::addSentence()`

Uma solicitação de informação adicional é disparada sempre que um `SensorNode` receber uma conjunção de fatos provenientes de seu antecessor (`ground_sentence`). Inicialmente, cria-se uma pergunta (`query`) que é, indiretamente, enviada ao cliente do mecanismo de inferência através de `Rete::ask(…)`, a qual retorna uma lista de fatos, isto é, uma lista de literais sem nem uma variável. As conjunções de cada fato dessa lista com a sentença recebida são adicionadas aos nós sucessores. Uma lista vazia significa que a condição testada pelo sensor não é satisfeita e que, portanto, nenhuma sentença deverá ser enviada, impedindo a ativação da regra.

A.9.2 Operação `buildQuery(...)`

A Figura 145 ilustra a operação privativa `SensorNode::buildQuery(...)`, responsável pela formulação da pergunta a ser enviada ao agente.

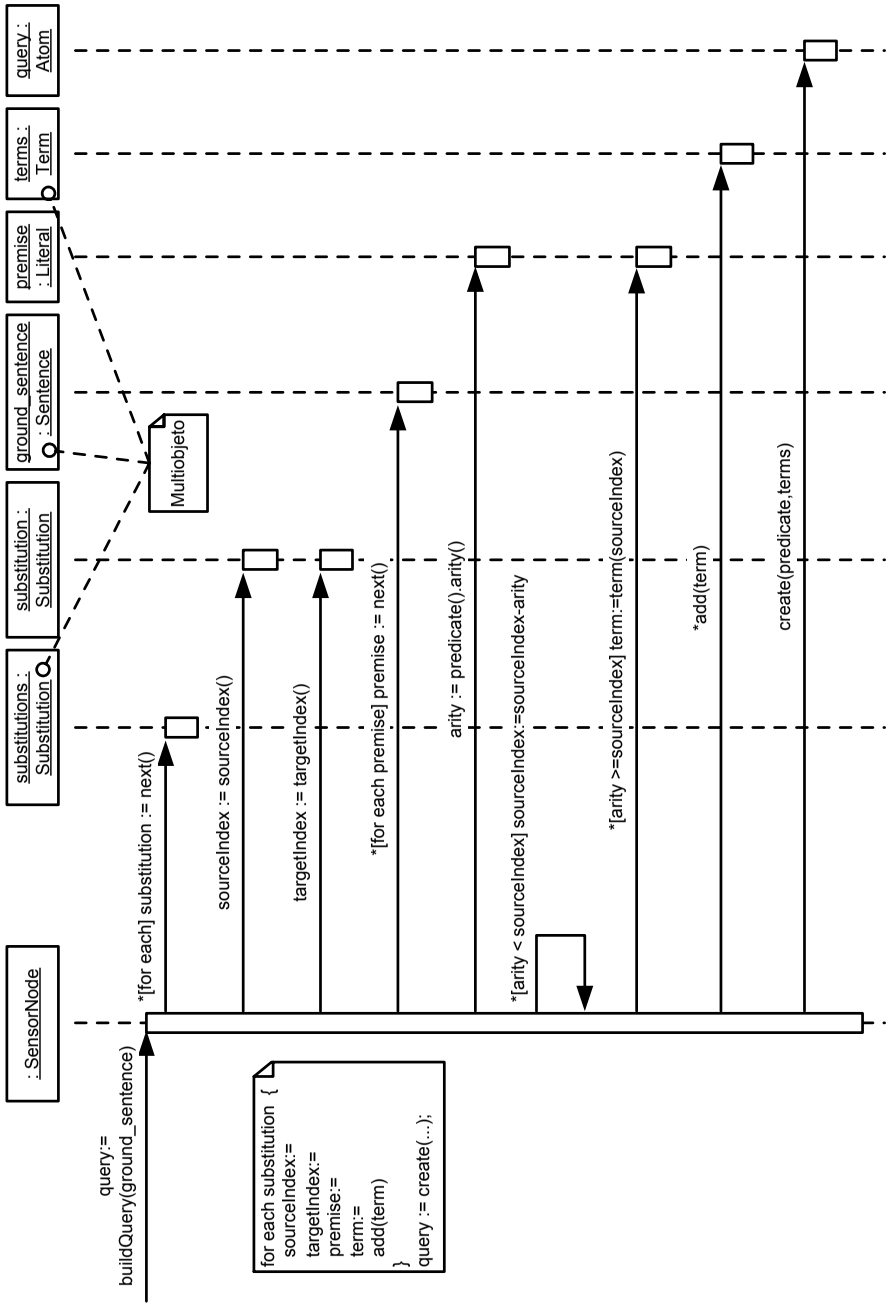


Figura 145: Diagrama de interação para `SensorNode::buildQuery()`

A solicitação é construída eliminando-se o maior número possível de variáveis do literal correspondente ao sensor, com o auxílio da sentença recebida. Os índices das variáveis cujos valores podem ser determinados a partir da sentença fechada foram determinados durante a fase de compilação e transferidos para o nó sensor na forma de uma coleção de substituições (`substitutions`). Cada substituição (`substitution`) corresponde a uma variável que pode ser eliminada. Dessa forma, a pergunta formulada é um objeto da classe `Atom` com um certo número de variáveis não resolvidas.

A.10 Diagramas de Interação para `Terminal`

A classe `TerminalNode` somente especializa as operações `Node::ddSentence()` e `Node::removeSentence()`.

A.10.1 Operação `addSentence(…)`

A Figura 146 ilustra o comportamento de `Terminal::addSentence(…)`.

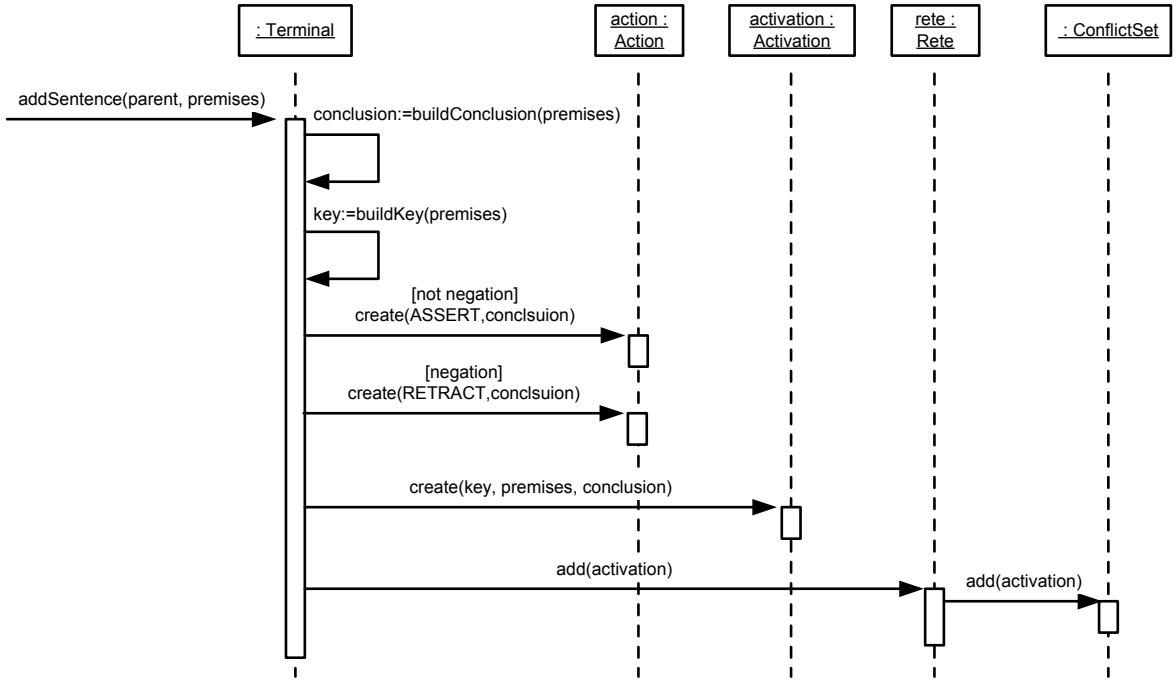


Figura 146: Diagrama de interação para `Terminal::addSentence()`

A.10.2 Operação `removeSentence(…)`

A Figura 147 ilustra o comportamento de `Terminal::removeSentence(…)`.

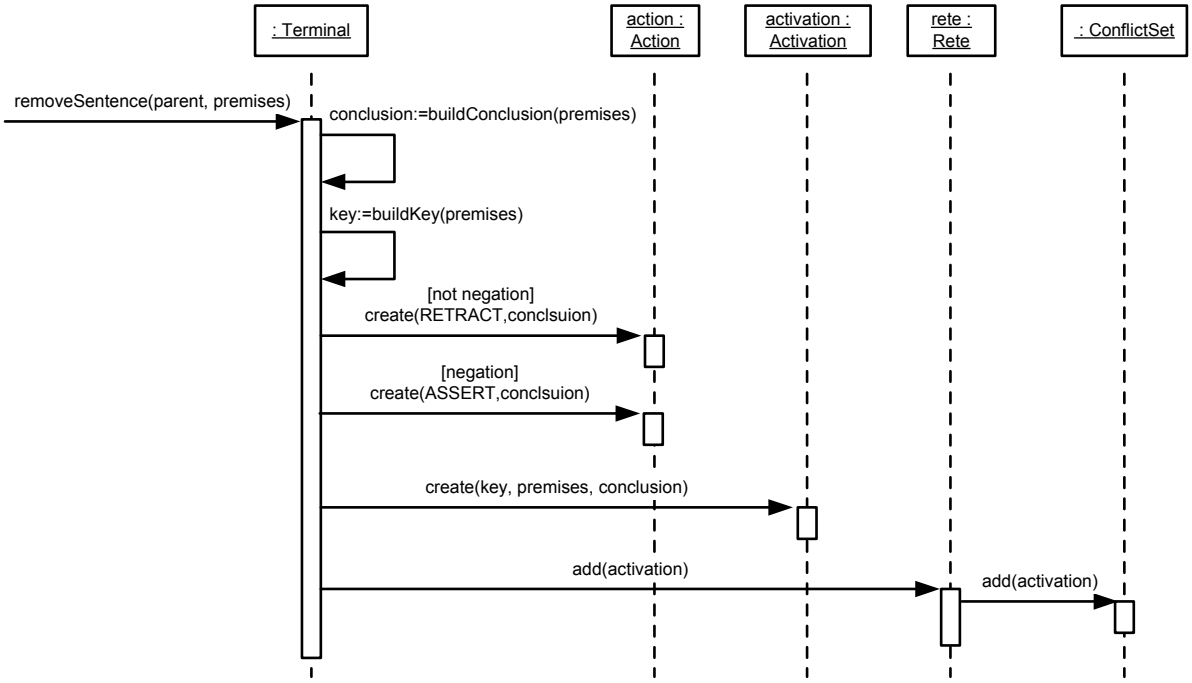


Figura 147: Diagrama de interação para `Terminal::removeSentence()`

A.10.3 Operação `buildConclusion(…)`

A operação privativa `Terminal::buildConclusion(…)` é responsável por eliminar as variáveis da expressão conseqüente da regra a partir da sentença fechada recebida (expressão antecedente) e de uma coleção de substituições (`substitutions`), determinadas durante a fase de compilação.

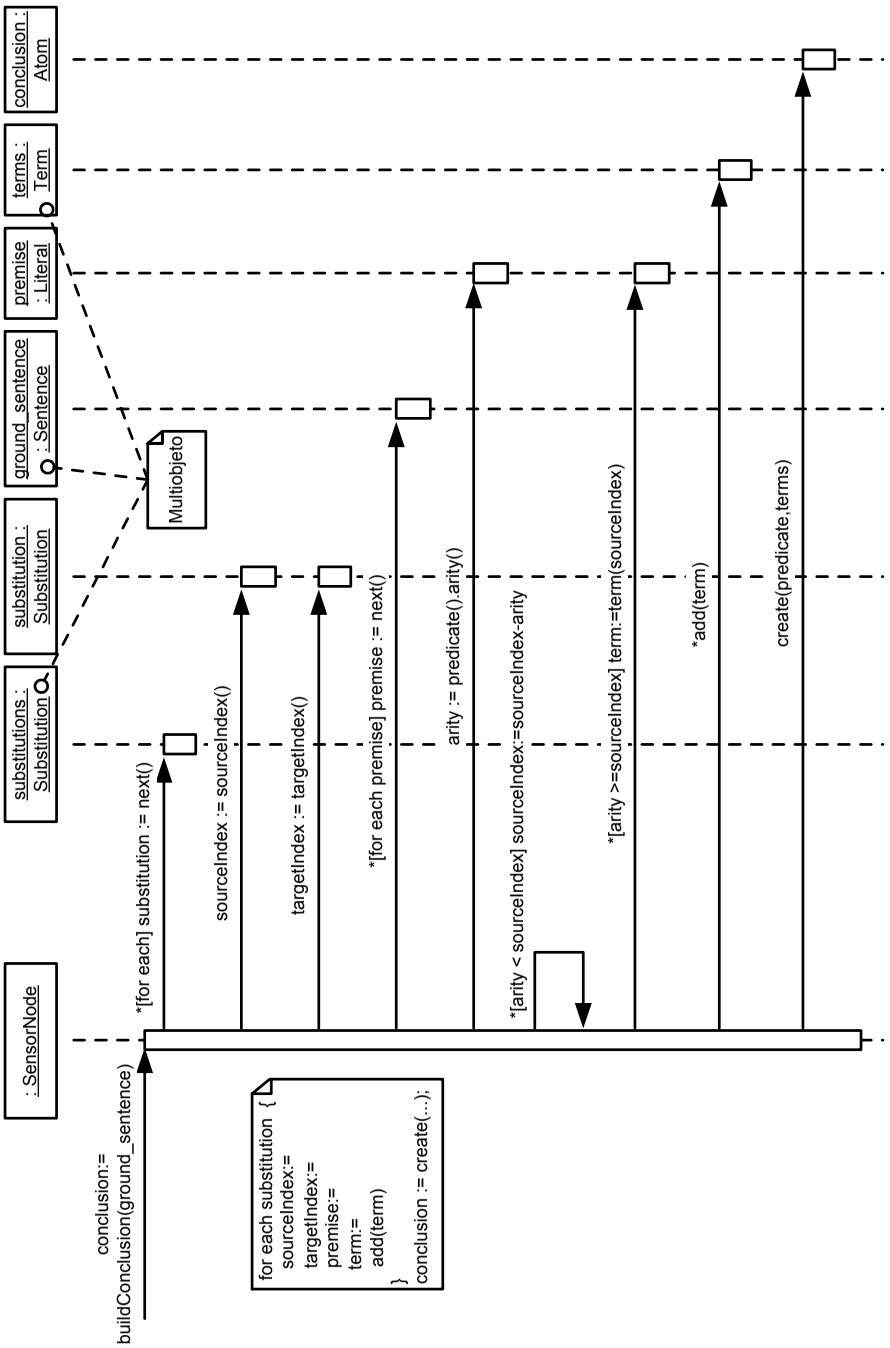


Figura 148: Diagrama de interação para `Terminal::buildConclusion()`

Apêndice B

Protocolo de Serialização da Linguagem Java

B.1 Introdução

A serialização de objetos produz e consome um fluxo (*stream*) de bits que contém um ou mais tipos primitivos e objetos (SUN, 2004). Os objetos inseridos no fluxo, por sua vez, podem se referir a outros objetos, os quais também serão representados no fluxo. A serialização de objetos produz apenas um formato de fluxo que codifica e armazena os objetos contidos.

Todo objeto que age como um container implementa uma interface que permite que os tipos primitivos e seus objetos componentes sejam armazenados e recuperados a partir dele. Essas interfaces são as interfaces `ObjectOutput` and `ObjectInput` , as quais:

- Fornecem um fluxo de bits para a escrita e a leitura dos objetos e tipos primitivos;
- Tratam as solicitações de inclusão (escrita) dos tipos primitivos e objetos no fluxo;
- Tratam as solicitações de recuperação (leitura) dos tipos primitivos e objetos do fluxo.

Todo objeto a ser armazenado no fluxo deve, explicitamente, permitir o seu armazenamento e implementar o protocolo necessário para salvá-lo e recuperá-lo do fluxo. Esse pro-

tocolo permite que o objeto container solicite que os objetos componentes salvem e recuperem seus estados.

Para ser armazenado em fluxo de objetos, um objeto deve implementar a interface `Serializable` ou a interface `Externalizable` (SUN, 2004):

- Para uma classe `Serializable`, a serialização de objetos pode, automaticamente, salvar e recuperar os campos de cada classe um objeto e, também automaticamente, lidar com classes que evoluem pela adição de campos ou supertipos. A classe serializável pode declarar quais dos seus campos podem ser salvos e recuperados e escrever ou ler valores e objetos opcionais
- Para uma classe `Externalizable`, a serialização de objetos delega à classe o controle completo sobre seu formato externo e sobre como os estado de seus supertipos serão salvos e recuperados.

B.2 Notação

A notação utilizada na descrição do protocolo de serialização e deserialização utilizado pela linguagem Java é apresentada na Tabela 22.

Tabela 22: Notação utilizada na descrição do protocolo de serialização da linguagem Java

Notação	Significado
`token: tipo`	O `token` é do `tipo` especificado.
`(tipo) token`	O `token` é do `tipo` especificado.
`token[n]`	Indica um número (`n`) predefinido de ocorrências do `token`, isto é, um *array*.
`(tipo)<param>`	Indica que um parâmetro cujo valor é do `tipo` especificado deve ser lido do fluxo de bits. Essa notação é geralmente usada para indicar o tamanho de um array no fluxo.
`$command`	Indica que um comando deve ser executado pelo serializador ou deserializador sem incluir ou retirar nada do fluxo de bits.
`0xValor`	Prefixo (`0x`) que indica que o `valor` é hexadecimal.
|	"ou"
&	"e"

B.3 Regras da Gramática de Serialização

Um fluxo serializado é representado por qualquer fluxo de bits que satisfaça a regra ***stream***, descrita abaixo.

stream: STREAM_MAGIC STREAM_VERSION contents

contents: content | contents content

content: object | blockdata

object: newObject | newClass | newArray | newString | newClassDesc | prevObject | TC_NULL | exception | TC_RESET

newObject: TC_NEWOBJECT | classDesc $newHandle classData[]

classDesc: newClassDesc | TC_NULL | (classDef)prevObject

newClassDesc: TC_NEWCLASS className serialVersionUID $newHandle classDescInfo

className: utf

serialVersionUID: (long)

$newHandle: o próximo número na seqüência é atribuído ao objeto que está sendo serializado ou deserializado,mas não é incluído no fluxo de bits.

classDescInfo: classDescFlags fields classAnnotation superClassDesc

classDescFlags: SC_SERIALIZABLE | SC_SERIALIZABLE & SC_WRITE_METHOD | SC_EXTERNALIZABLE | SC_EXTERNALIZABLE & SC_BLOCDATA

fields: (short)<count> fieldDesc[count]

fieldDesc: primitiveDesc | objectDesc

primitiveDesc: prim_typecode fieldName

prim_typecode: TC_BYTE | TC_CHAR | TC_SHORT | TC_INT | TC_LONG | TC_FLOAT | TC_DOUBLE | TC_BOOLEAN

fieldName: utf

objectDesc: object_typecode fieldname fieldType

object_typecode: TC_ARRAY | TC_OBJECT

fieldType: (string)fieldTypeBytes

fieldTypeBytes: object_typecode typeName TC_ENDFIELDTYPE

typeName: byte[]

classAnnotation: TC_ENDBLOCKDATA | contents TC_ENDBLOCKDATA

superClassDesc: classDesc

prevObject: TC_REFERENCE (int)Handle

classData: nowrclass | wrclass objectAnnotation | externalContents | objectAnnotation

nowrclass: values

wrclass: values

values: value | values value

value: (byte) | (char) | (short) | (int) | (long) | (float) | (double)

objectAnnotation: TC_ENDBLOCKDATA | contents TC_ENDBLOCKDATA

string: newString | (string)prevObject

newString: TC_NEWSTRING $newHandle utf | TC_NEWLONGSTRING $newHandle long_utf

utf: (short)<length> byte[length]

long_utf: (int)<length> byte[length]

newClass: TC_NEWCLASS classdesc $newHandle

newArray: TC_NEWARRAY classDesc $newhandle (int)<size> value[size]

exception: TC_EXCEPTION $reset (Throwable)object $reset

$reset: o conjunto de objetos conhecidos é descartado, de modo que os objetos da exceção não se sobreponham aos objetos enviados anteriormente ou aos objetos que possam ser enviados após a exceção.

externalContent: (byte) | object

externalContents: externalContent | externalContents externalContent

blockData: blockDataShort | blockDataLong

blockDataShort: TC_BLOCKDATA (unsigned byte)<size> (byte)[size]

blockDataLong: TC_BLOCKDATALONG (int)<size> (byte)[size]

STREAM_MAGIC: 0xACED

STREAM_VERSION: 0x0005

TC_NULL: 0x70

TC_RESET: 0x79

TC_NEWOBJECT: 0x73

TC_NEWCLASSDESC: 0x72

TC_BLOCKDATA: 0x77

TC_ENDBLOCKDATA: 0x78

TC_ENDFIELDTYPE: 0x3B

TC_BLOCKDATALONG: 0x7A

TC_REFERENCE: 0x71

TC_NEWCLASS: 0x76

TC_NEWARRAY: 0x75

TC_EXCEPTION: 0x7B

TC_NEWSTRING: 0x74

TC_NEWLONGSTRING: 0x7C

TC_BYTE: 0x42

TC_CHAR: 0x43

TC_SHORT: 0x53

TC_INT: 0x49

TC_LONG: 0x4A

TC_FLOAT: 0x46

TC_DOUBLE: 0x44

TC_BOOLEAN: 0x5A

TC_ARRAY: 0x5B

TC_OBJECT: 0x4C

SC_SERIALIZABLE: 0x02

SC_WRITE_METHOD: 0x01

SC_EXTERNALIZABLE: 0x04

SC_BLOCKDATA: 0x08