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

UM FRAMEWORK PARA AGENTES ADAPTATIVOS NA WEB SEMÂNTICA

DANIEL DA SILVA ELIAN

Porto Alegre
2008

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

UM FRAMEWORK PARA AGENTES ADAPTATIVOS NA WEB SEMÂNTICA

DANIEL DA SILVA ELIAN

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre, pelo programa de Pós-Graduação em Ciência da Computação da Faculdade de Informática, Pontifícia Universidade Católica do Rio Grande do Sul.

Orientador: Prof. Dr. Marcelo Blois Ribeiro

Porto Alegre
2008

Pontifícia Universidade Católica do Rio Grande do Sul

Dados Internacionais de Catalogação na Publicação (CIP)

E42f Elian, Daniel da Silva.
Um framework para agentes adaptativos na web semântica / Daniel da Silva Elian. – Porto Alegre, 2008.
135 p.

Diss. (Mestrado) – Fac. de Informática, PUCRS.
Orientador: Prof. Dr. Marcelo Blois Ribeiro

1. Informática. 2. Framework. 3. Sistemas Multiagentes. 4. Web Semântica. I. Ribeiro, Marcelo Blois. II. Título.

CDD 006.3

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

PUCRS
Campus Central
Av. Ipiranga, 6681 - prédio 16 - CEP 90619-900
Porto Alegre - RS - Brasil
Fone: +55 (51) 3320-3544 - Fax: +55 (51) 3320-3548
Email: bceadm@pucrs.br
www.pucrs.br/biblioteca

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

TERMO DE APRESENTAÇÃO DE DISSERTAÇÃO DE MESTRADO

Dissertação intitulada **"Um Framework para Agentes Adaptativos na Web Semântica"**, apresentada por Daniel da Silva Elian, como parte dos requisitos para obtenção do grau de Mestre em Ciência da Computação, Sistemas de Informação, aprovada em 29/01/08 pela Comissão Examinadora:

Prof. Dr. Marcelo Blois Ribeiro – Orientador — PPGCC/PUCRS

Prof. Dr. Ricardo Melo Bastos – PPGCC/PUCRS

Prof. Dr. Ricardo Choren Noya – IME-RJ

Homologada em 15/12/2008, conforme Ata No. 25 pela Comissão Coordenadora.

Prof. Dr. Fernando Gehm Moraes
Coordenador.

Campus Central
Av. Ipiranga, 6681 – P32 – sala 507 – CEP: 90619-900
Fone: (51) 3320-3611 – Fax (51) 3320-3621
E-mail: ppgcc@inf.pucrs.br
www.pucrs.br/facin/pos

Dedico este trabalho à minha mãe Suleima, à minha falecida avó Maria e, principalmente, ao meu falecido avô Ody, grande inspiração para a busca do conhecimento, da ética, da justiça e da honestidade.

Daniel da Silva Elian

AGRADECIMENTOS

Aos meus professores, principalmente ao meu orientador, Prof. Dr. Marcelo Blois Ribeiro, por todos os ensinamentos, críticas e conselhos.

Aos membros da banca, professores Ricardo Choren Noya e Ricardo Melo Bastos, por aceitarem o convite de participação na avaliação deste trabalho. Não posso deixar de mencionar a professora Vera Lúcia Strube de Lima, por me acompanhar desde o começo do mestrado. Infelizmente, ela não pôde participar na minha defesa por compromissos de trabalho.

Aos colegas do CDPe e aos membros do grupo de pesquisa ISEG, por terem compartilhado muitos momentos, tanto agradáveis quanto desanimadores. Também aos colegas do mestrado, participantes na concretização deste trabalho.

Ao convênio Dell-PUCRS, por viabilizar a bolsa de estudos para o mestrado.

A todos os funcionários da PUCRS, pela postura e profissionalismo.

Aos meus amigos, pelo companheirismo e motivação.

A todos os familiares, pelo carinho e amor.

À minha namorada Letícia, pela paciência, amor e motivação. Também soube dar liberdade quando precisei me dedicar com mais afinco aos estudos, principalmente por eu ser aquariano. Não posso esquecer os seus familiares, igualmente incentivadores.

Principalmente a três pessoas: aos meus falecidos avós Ody e Maria, por tudo aquilo que carrego sobre conhecer uma vida construída para o bem; e à minha mãe Suleima, pelas mesmas qualidades, além do amor, dedicação e paciência sentidos até o final deste trabalho.

A todas as pessoas não citadas explicitamente que, de alguma forma, direta ou indiretamente, possibilitaram a conclusão deste trabalho.

RESUMO

As arquiteturas atuais para agentes adaptativos não apresentam uma preocupação explícita para com as exigências da Web Semântica. Essas exigências se referem ao uso de linguagens para se expressar metainformação processável por máquina. Além disso, essas arquiteturas não se mostram genéricas o suficiente para permitirem que componentes básicos, presentes na maioria dos processos adaptativos, sejam utilizados em diferentes domínios e resoluções de problemas, conforme a necessidade de aplicação de Sistemas Multiagentes. Quando se fala em arquitetura genérica, quer-se dizer que a mesma é capaz de alterar o seu processo de adaptação, e não somente a estrutura do agente. Quanto aos processos adaptativos, afirma-se serem todas as ações necessárias para se modificar a estrutura de um agente. Dado ao exposto, o presente trabalho visa apresentar o desenvolvimento de um *framework* que permita a adaptação da estrutura e do comportamento de agentes de software na Web Semântica, de acordo com o contexto composto por propriedades descritas, principalmente, por meio de ontologias. Adicionalmente, esse *framework* possibilita a abstração da arquitetura específica utilizada na construção de um agente, sendo necessário apenas o acoplamento entre o agente e o *framework* de adaptação.

Palavras-chave — Agentes de software, agentes adaptativos, Web Semântica, Sistemas Multiagentes, ontologias.

ABSTRACT

The current architectures for adaptive agents do not present an explicit concern to the demands of Semantic Web. These demands refer to the use of languages to represent processable metainformation by machines. Furthermore, these architectures are not sufficiently generic to allow basic components, found in most of the adaptive methods, to be used in different domains and problem resolutions, according to the needs of Multiagent Systems appliance. A generic architecture is capable of change its adaptation process, not only the agent structure. In relation to the adaptive processes, they are the all necessary actions to modify the structure of an agent. For these reasons, the actual work aims to present the development of a framework that enables the adaptation of structure and behavior of software agents in the Semantic Web, in accordance with context composed by properties described, mostly, via ontologies. Moreover, this framework makes possible the abstraction of particular architecture used in the construction of an agent, being necessary just the linkage between agent and adaption framework.

Key-words — Software agents, adaptive agents, Semantic Web, Multiagent Systems, ontologies.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

AOP – Programação Orientada a Aspectos

IA – Inteligência Artificial

OWL – Ontology Web Language

RDF – Resource Description Framework

RDFS – RDF Schema

SMA – Sistema Multiagente

XML – eXtensible Markup Language

W3C – World Wide Web Consortium

Web – World Wide Web

WS – Web Semântica

SUMÁRIO

1 INTRODUÇÃO

Os atuais sistemas computacionais se mostram cada vez mais complexos. Inseridos em ambientes distribuídos, de larga escala, dinâmicos, abertos e heterogêneos, esses sistemas vêm ganhando maior importância e influenciando parte da vida diária do homem, na medida em que ficam mais poderosos e mais conectados entre si por meio de redes locais e de longa distância, além da interação humana por meio de interfaces de usuário [WEI01]. Um dos mais proeminentes exemplos de tais sistemas é a Internet. Por ser tão complexa, ainda não é possível caracterizá-la completamente e descrevê-la com precisão. Com controles cada vez mais descentralizados, seus componentes parecem atuar como indivíduos, demandando atributos como autonomia, racionalidade e inteligência. Assim, o paradigma de software baseado em agentes é apropriado para a exploração das possibilidades surgidas nela, como afirmado por [WOO02]. Os sistemas em que esse paradigma é aplicado são comumente conhecidos por Sistemas Multiagentes (SMAs).

Com o surgimento da *World Wide Web* (Web), a Internet se popularizou. Na Web, uma quantidade enorme de informação é construída, crescendo exponencialmente [FEN05]. Por não haver um padrão amplamente utilizado para a construção dessa informação, torna-se difícil processá-la para a obtenção de conhecimento relevante. Nesse cenário, [ANT04] cita a necessidade dos dados serem acompanhados de suas semânticas. Diferentemente da compreensão humana, os sistemas precisam de semânticas fornecidas de maneira processável por máquina. Como conseqüência, verifica-se a demanda pela Web Semântica (WS), ou seja, especificamente uma Web de informação processável por máquina, na qual o significado é bem definido por padrões. Ela é a realização de um aspecto da Web que foi parte das esperanças e sonhos originais de 1989, visionados por Tim Berners-Lee.

Sabendo-se que a WS possui as mesmas características dos sistemas citados anteriormente, nota-se que a utilização de SMAs sobre ela é vantajosa. Além disso, sabendo-se que os futuros sistemas não serão apenas inteligentes, mas também adaptativos – capazes de melhorar o seu funcionamento de maneira autônoma e em tempo de execução –, não basta construir agentes sem os mesmos serem considerados artefatos dinâmicos. Assim, os agentes precisarão ser capazes de se adaptar à natureza dinâmica da WS. Como a adaptação automatizada é uma opção possível somente se uma estrutura suficiente for fornecida, a proposta de uma arquitetura para o desenvolvimento de agentes adaptativos na WS é bastante propícia.

Desde a metade da década de 1990, [WEI95] afirma que o tópico de adaptação e aprendizado em SMAs tem adquirido uma atenção crescente na Inteligência Artificial (IA). Seguindo essa tendência, o presente trabalho expõe o desenvolvimento de um *framework* que permite a adaptação da estrutura e do comportamento de agentes de software na WS, de acordo com o contexto composto por propriedades descritas, principalmente, por meio de ontologias.

1.1 Questão de Pesquisa

Com base na constatação da necessidade de uma arquitetura para o desenvolvimento de agentes adaptativos, emerge a questão de pesquisa que guia este trabalho: **"É possível definir uma arquitetura que permita a adaptação estrutural e comportamental de agentes de software na Web Semântica de acordo com o contexto composto por propriedades descritas, principalmente, por meio de ontologias?"**.

1.2 Objetivo Geral

O objetivo geral desta pesquisa é propor e aplicar uma arquitetura para agentes adaptativos na Web Semântica, utilizando uma plataforma de desenvolvimento de SMA.

1.3 Objetivos Específicos

- Aprofundar o estudo teórico sobre o padrão da Web Semântica.
- Aprofundar o estudo teórico sobre adaptação em agentes.
- Avaliar as propostas de agentes adaptativos como apoio ao desenvolvimento da nova arquitetura.
- Identificar os elementos que devem ser considerados para a proposta da nova arquitetura.
- Propor uma arquitetura para agentes adaptativos na Web Semântica.
- Implementar o *framework* proposto junto a uma plataforma de desenvolvimento de SMA e executar um cenário de aplicação.
- Avaliar os resultados.

1.4 Estrutura do Trabalho

Este trabalho está organizado da seguinte forma:

- O segundo capítulo apresenta a fundamentação teórica referente a: agentes de software, Sistemas Multiagentes, desenvolvimento de SMAs, Web Semântica e adaptação.
- O terceiro capítulo descreve a adaptação em agentes de software, conceituando-a, caracterizando o problema que motivou esta pesquisa, e os trabalhos relacionados.
- O quarto capítulo explana sobre o modelo de arquitetura para a adaptação em agentes de software, mostrando a análise de diversos modelos de adaptação que permitiram a identificação dos elementos básicos de adaptação, além da descrição do Modelo Conceitual.
- O quinto capítulo apresenta a implementação do *framework* e a exemplificação de utilização do mesmo por meio de um cenário de aplicação.
- O sexto capítulo esclarece as considerações finais e os trabalhos futuros.
- As últimas seções constituem as referências pesquisadas e os apêndices.

2 FUNDAMENTAÇÃO TEÓRICA

A compreensão dos fundamentos teóricos e das tecnologias básicas que apóiam a arquitetura proposta é necessária para o correto entendimento da mesma e dos trabalhos relacionados. Sendo assim, nas seções 2.1 e 2.2, apresentam-se os conceitos e características de agentes de software e de SMAs. Na seção 2.3, são citadas quatro plataformas e, dentre elas, explicada a plataforma *SemantiCore* [RIB07], pois esta é utilizada para o desenvolvimento dos agentes presentes no exemplo de aplicação do *framework*. Por fim, na seção 2.4, explica-se a Web Semântica, sendo focado o uso de ontologias para a manipulação de informação com anotações semânticas dentro dos agentes.

2.1 Agentes de Software

Durante as quase seis décadas da IA, os agentes computacionais têm aparecido como um tópico ativo de exploração. O conceito de agente pode ser buscado em 1973, quando Carl Hewitt, Peter Bishop e Richard Steiger propuseram o ACTOR [HEW73], um agente ativo que executa um papel no momento correto, de acordo com um script. Posteriormente, em 1977, nas suas pesquisas dentro da área de Inteligência Artificial Distribuída, Hewitt explicou que os atores eram objetos de execução concorrente, autocontidos e interativos [HEW77]. Recentemente, uma definição interessante apresentada por [WOO02], afirma que o agente é um sistema computacional inserido em um ambiente, capaz de atingir os objetivos planejados por meio de ações autônomas nesse ambiente.

[WEI95] afirma que existe uma discussão considerável e uma controvérsia proveitosa sobre as propriedades de um agente, com a seguinte questão central: quais são as propriedades que permitem a um objeto, como um programa ou um robô industrial, ser um agente? Formando as interseções das diversas respostas que têm sido dadas a essa questão, uma obtém algo como a "essência" das propriedades principais, que implica na definição concisa de um agente como um objeto que, não importando a maneira, é inteligente e autônomo:

- Habilidades perceptivas, cognitivas e eficazes.
- Habilidades comunicativas e sociais.
- Autonomia.

Adicionalmente, à interseção das diversas respostas, pode-se complementar a conceituação de um agente afirmando ser ele uma entidade de software que:

- É autônoma: um agente age de forma autônoma em relação a outras entidades de um sistema, ou seja, o seu fluxo de execução é independente.
- Atua em um ambiente: o conceito de ambiente relaciona-se fortemente à existência de vários agentes atuando em um sistema, além da capacidade do agente de perceber os acontecimentos ao seu redor.
- É inteligente: um agente é capaz de atuar conforme os objetivos definidos e de acordo com o conhecimento relativo ao seu ambiente e às suas ações.
- Possui um modelo limitado do mundo: um agente possui apenas o modelo de mundo necessário para o desempenho de suas tarefas e para o alcance de seus objetivos.

Resumidamente, [RIB02] afirma que um agente é uma entidade de software que, a partir de informações sentidas no ambiente, captadas através da interação direta com outros agentes de software ou humanos, ou geradas a partir dos mecanismos dedutivos internos ao agente, atua em um ambiente buscando o alcance de seus objetivos.

2.2 Sistemas Multiagentes

[WEI95] define o SMA como um sistema computacional composto por diversos agentes capazes de interagir mutuamente e com o ambiente. Posteriormente, [WEI01] o conceitua como um método conveniente para o tratamento de agentes, permitindo manuseá-los coletivamente, com maior facilidade, como uma sociedade, tornando possível a afirmativa de que esse método é o melhor caminho para caracterizar ou desenvolver sistemas computacionais distribuídos. Apesar disso, existem momentos em que os SMAs não são a melhor solução para um problema. Sendo assim, [JEN96] justifica que um domínio passível de aplicação dessa tecnologia deve possuir as seguintes características:

- Distribuição intrínseca de dados, capacidade de resolução de problemas e responsabilidades.
- Autonomia em suas subpartes, conservando a estrutura organizacional.
- Complexidade nas interações, exigindo negociação e cooperação.
- Diligência, devido à possibilidade de mudanças dinâmicas em tempo real no ambiente.

Quanto às características, afirma-se que os SMAs:

- Provêem uma infra-estrutura especificando protocolos de interação e comunicação.
- São tipicamente abertos.
- Possuem controle distribuído.
- São constituídos por dados descentralizados.
- Necessitam de computação assíncrona.
- São compostos por agentes que têm apenas uma informação incompleta e são restringidos por suas capacidades.
- São compostos por agentes autônomos e distribuídos, podendo apresentar interesses próprios ou comportamento cooperativo.

2.3 Desenvolvimento de Sistemas Multiagentes

Sabendo-se que as técnicas convencionais de Engenharia de Software não podem ser diretamente aplicadas à especificação de SMAs, vêm sendo propostas arquiteturas que incorporam conceitos nativos de agentes em seus modelos. Apesar de essas arquiteturas permitirem o avanço na criação de plataformas para a implementação de SMAs, estas últimas não são completas para todos os domínios de aplicação existentes.

Para tornar completa uma plataforma de implementação de SMA, é necessário o suporte ao desenvolvimento interno dos agentes e à criação da infra-estrutura de atuação dos mesmos em sua organização (parte externa ao agente). Dentre as plataformas disponíveis na atualidade, tem-se: *MadKit* [MAD07], JADE [JAD07], *OpenCybele* [OPC04] e *SemantiCore*. É importante salientar que nenhuma dessas plataformas oferece suporte total ao desenvolvimento de agentes.

Outra questão relevante é sobre a escolha do *SemantiCore* para a implementação dos agentes. A arquitetura proposta neste trabalho utiliza tecnologias da WS, e o *SemantiCore* é um *framework* que oferece artefatos em alto nível para o desenvolvimento de aplicações na mesma.

2.3.1 SemantiCore

[RIB04] apresenta o *SemantiCore* como uma camada de abstração baseada em agentes para o desenvolvimento de aplicações na WS. Essa camada é estruturada como um *framework*

sobre plataformas de distribuição computacional, ocultando os detalhes da construção de aplicações distribuídas, fornecendo as primitivas de serviços básicos, e permitindo a definição interna do agente. Tudo isso reduzindo o tempo de implementação dos desenvolvedores de SMAs. O *SemantiCore* surgiu a partir de uma extensão da arquitetura *Web Life* [RIB02], e, atualmente, encontra-se disponível na versão 2008.

O *framework SemantiCore* é dividido em dois modelos: o modelo do agente (*SemanticAgent*), responsável pelas definições internas dos agentes, e o modelo do domínio semântico, responsável pela definição da composição do domínio e suas entidades administrativas. Os dois modelos dispõem de pontos de flexibilidade (*hotspots*), permitindo a associação de diferentes padrões, protocolos e tecnologias.

O modelo do agente possui uma estrutura orientada a componentes, os quais contribuem para uma parte essencial do funcionamento do agente, agregando todos os aspectos necessários à sua implementação. É possível simplificar a arquitetura do agente com a retirada de um ou mais componentes não relacionados ao cumprimento das suas tarefas. Os quatro componentes básicos do agente são:

- Sensorial: permite a recuperação de objetos a partir do ambiente. O componente armazena os diversos sensores, cada um capturando um tipo diferente de objeto, e verifica se algum deles deve ser ativado pelo recebimento de alguma mensagem do ambiente. Se um ou mais sensores são ativados, os objetos são enviados a outros componentes, para processamento. Um tipo pré-definido existente na plataforma é o *OWLSensor*, que captura objetos OWL[1].
- Decisório: encapsula o mecanismo de tomada de decisão do agente, que é um ponto de flexibilidade. Mesmo assim, a plataforma disponibiliza uma integração nativa com o *Jena* [JEN07], possibilitando o uso de máquinas de inferência nesse componente. A saída gerada por ele deve ser uma instância de uma ação (*Action*), que também é um ponto de flexibilidade. As ações mapeiam todos os possíveis comandos que um agente deve entender para trabalhar apropriadamente.
- Executor: encapsula os planos de ação que são executados pelo agente, podendo trabalhar com o mecanismo de *workflow*[2].
- Efetuador: encapsula os dados recebidos dos outros componentes em objetos a serem transmitidos no ambiente. Toda a publicação de um objeto no ambiente requer um

[1] A linguagem OWL, para a descrição de ontologias, é explicada na seção 2.4.1.
[2] Seqüência de passos necessários para se atingir a automação de processos de negócio, conforme um conjunto de regras definidas.

efetuador apropriado no agente. Um tipo pré-definido existente na plataforma é o *OWLEffector*, que encapsula as mensagens em um formato OWL.

Um agente precisa estar situado em um ambiente para poder atuar. No *SemantiCore*, esse ambiente é denominado domínio semântico, e este requer um domínio Web para operar. Como ilustrado na Figura 2.1, cada domínio semântico é composto por algumas entidades administrativas, como o Controlador de Domínio (*Domain Controller*) e o Gerente de Ambiente (*Environment Manager*). O Controlador de Domínio é responsável pelo registro dos agentes no ambiente, pela recepção de agentes móveis provenientes de outros domínios e, pela manutenção e execução de aspectos relacionados à segurança. O Gerente de Ambiente representa uma ponte entre o domínio semântico do *SemantiCore* e os domínios Web convencionais.

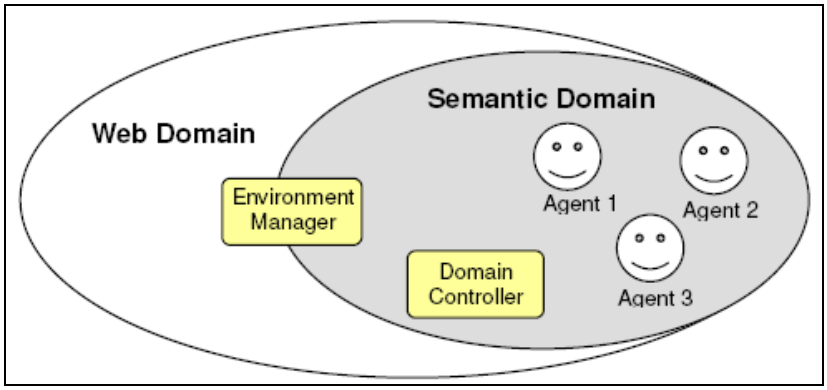

Figura 2.1 – Modelo de domínio do *SemantiCore* [ESC06].

2.4 Web Semântica

Tim Berners-Lee argumenta em [LEE95] que escreveu um programa chamado Enquire, em 1980, considerado por ele o antecessor da Web. Seu propósito era manter o curso da complexa rede de relacionamentos acerca de pessoas, programas, máquinas e idéias, apenas para uso próprio. Em 1989, propôs a Web, como uma extensão da sua ferramenta pessoal. Ela foi projetada para ser um espaço universal de informação. Assim, deveria ser possível conectar qualquer pedaço de informação para ser acessado usando redes. Esta universalidade foi proposta para (i) permitir às pessoas trabalharem melhor em conjunto, e (ii) passar aos computadores qualquer atividade possível de ser reduzida a um processo racional, auxiliando no tratamento de grandes quantidades de dados.

A Web vista atualmente é apenas uma parte do plano original. Pensava-se em editores de hipertexto intuitivos, e não tão simples como os existentes no presente. Os editores atuais

impossibilitam que a Web seja utilizada como um meio plenamente colaborativo. Imaginava-se, também, o controle de acesso e ferramentas de arquivamento tão fáceis de operar quanto os navegadores. Contudo, ainda existem deficiências para o alcance dessas metas, que se somam aos limites impostos pela falta de suporte das máquinas ao que pode ser feito com informação. Mecanismos de busca perdem-se no meio de tanta documentação impossível de ser diferenciada. Assim, [FEN05] afirma que é preciso informação sobre informação – metadados – para auxiliar na organização da segunda.

Paralela aos metadados, a necessidade por padrões comuns capazes de, no futuro, definir regras que permitam aos computadores conversar sobre um assunto facultado, exigiu a criação do *World Wide Web Consortium* (W3C) [W3C06]. O W3C é um local onde as organizações associadas têm a Web como parte crucial para a sobrevivência de seus negócios. Busca-se, com ele, desenvolver protocolos capazes de unificar os programas por meio de uma mesma linguagem, alcançando-se, assim, a chave para o desenvolvimento da Web.

Alcançar os objetivos da WS, segundo [FEN05], requer:

- Desenvolvimento de linguagens para expressar metainformação processável por máquina para documentos, e desenvolvimento de terminologias usando essas linguagens e as disponibilizando na Web.
- Desenvolvimento de ferramentas e novas arquiteturas que usam tais linguagens e metodologias, fornecendo suporte na busca, acesso, apresentação e manutenção das fontes de informação.
- Realização de aplicações que fornecem um novo nível de serviço para os usuários humanos da WS.

Os princípios fundamentais para a construção da WS, definidos por [KOI01], são implementados nas camadas das tecnologias da Web e dos padrões. Observam-se essas camadas na Figura 2.2. Dentre elas, para o escopo do *framework* proposto, são estudadas as camadas relacionadas aos dados e às regras. Esses dados são descritos por meio de ontologias. Por meio destas, é possível representar o domínio de forma semântica. Na Ciência da Computação, uma ontologia é um modelo de dados que representa um domínio, e é usado para se pensar logicamente sobre os objetos nesse domínio e sobre as relações entre eles. Para [GRU93], ela nada mais é do que uma especificação da conceituação. [BOR97] complementa a definição de Gruber, afirmando que uma ontologia é uma especificação formal explícita de uma conceituação compartilhada. “Conceituação” se refere a um modelo abstrato de algum

fenômeno no mundo que identifica os conceitos relevantes desse fenômeno. “Explícita” significa que os tipos de conceitos usados e as restrições nos seus usos são definidos explicitamente. “Formal” se refere ao fato que uma ontologia deve ser processável por máquina.

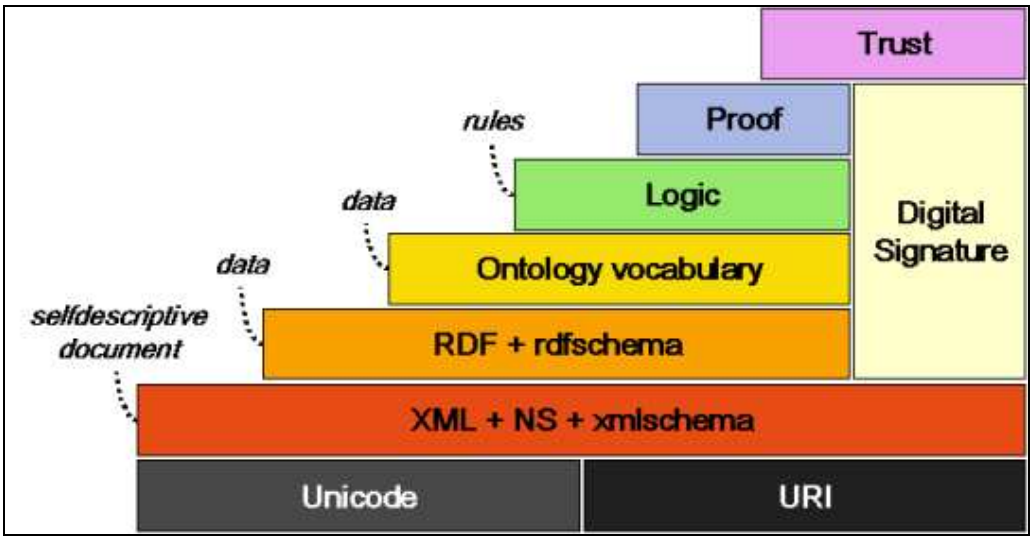

Figura 2.2 – O enfoque em camadas da Web Semântica [KOI01].

Segundo [GRU96] e [NOY01], as ontologias, na sua maioria, descrevem os seguintes componentes:

- Indivíduos: ou instâncias, são os componentes básicos. Podem incluir objetos concretos, como pessoas e animais, além de objetos abstratos, como números e palavras.
- Classes: ou conceitos, são grupos, conjuntos ou coleções de objetos abstratos. Podem conter indivíduos, outras classes, ou uma combinação de ambos.
- Atributos: descrevem objetos. Cada atributo tem, pelo menos, um nome e um valor.
- Relacionamentos: ou relações, são atributos cujo valor é outro objeto. Descrevem a correspondência entre os objetos na ontologia.

2.4.1 Linguagens para a Descrição de Ontologias

Na utilização de ontologias, [LEM07] afirma que alguns aspectos devem ser observados. O primeiro se refere à criação das ontologias. Aconselha-se a utilização de uma ontologia já existente para um determinado domínio. Encontrada uma ontologia previamente construída para a área de interesse, o esforço inicial é para o entendimento da mesma,

acrescentando-se os conceitos e as relações pertinentes ao domínio em questão. O uso de ferramentas para a criação de ontologias também deve ser observado.

O segundo aspecto se refere ao modo de representação da ontologia. Existem, atualmente, diversas maneiras de se representar as ontologias por meio da utilização de linguagens de marcação. De acordo com [ANT04], as mais importantes linguagens de descrição de ontologias disponíveis são:

- XML – *eXtensible Markup Language*: provê uma sintaxe superficial para documentos estruturados e tem restrições quanto à representação do significado dos documentos.
- RDF – *Resource Description Framework*: é um modelo de dados com semântica simples para descrever objetos (recursos) e relacionamentos entre eles.
- RDFS – RDF *Schema*: é uma linguagem de descrição de vocabulários para descrever propriedades e classes de recursos RDF.
- OWL – *Web Ontology Language*: utiliza lógica descritiva para a explicitação de conhecimento. Permite descrever propriedades e classes, assim como relações entre as classes (como *disjointness*), cardinalidade (como "exatamente um"), igualdade, características das propriedades (como simetria) e classes enumeráveis.

Levando-se em consideração os objetivos deste trabalho, são explicadas com mais detalhes as linguagens RDF e OWL.

2.4.1.1 RDF

O RDF é um *framework* mantido pelo W3C para representar informações na Web. Complementando, é um modelo de dados básico, do tipo entidade-relacionamento, para se escrever declarações sobre objetos Web (recursos). Seus conceitos fundamentais, segundo [ANT04], são os seguintes:

- Recurso (*Resource*): qualquer coisa sobre o que se possa falar. Todo recurso tem uma URI (*Universal Resource Identifier*) associada. Uma URI pode ser uma URL (*Unified Resource Locator*) ou qualquer outro tipo de identificador único [URI94].
- Propriedade (*Property*): tipo especial de recurso. As propriedades descrevem relações entre os recursos e são identificadas por URIs.
- Declaração (*Statement*): é uma tripla do tipo sujeito-predicado-objeto, constituído por um recurso, uma propriedade e um valor. Valores podem ser recursos ou literais.

A Figura 2.3 apresenta um grafo RDF como exemplo. O grafo demonstra um documento com o título "*RDF/XML Syntax Specification (Revised)*", além da página principal e o nome completo do seu editor. Os nodos são representados por elipses, os predicados por arcos, e os literais por retângulos.

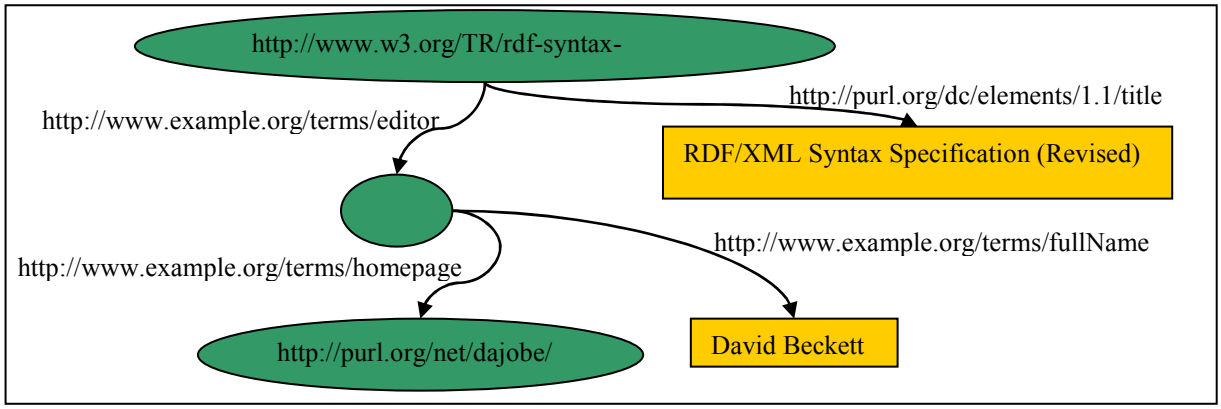

Figura 2.3 – Exemplo de grafo RDF [RDG04].

A Figura 2.4 expõe o código representando o documento RDF/XML completo, serializado do grafo da Figura 2.3.

```
1.  <?xml version="1.0"?>
2.  <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3.         xmlns:dc="http://purl.org/dc/elements/1.1/"
4.         xmlns:ex="http://example.org/stuff/1.0/">
5.    <rdf:Description rdf:about="http://www.w3.org/TR/rdf-syntax-grammar"
6.                     dc:title="RDF/XML Syntax Specification (Revised)">
7.      <ex:editor>
8.        <rdf:Description ex:fullName="David Beckett">
9.          <ex:homePage rdf:resource="http://purl.org/net/dajobe/" />
10.       </rdf:Description>
11.     </ex:editor>
12.   </rdf:Description>
13. </rdf:RDF>
```

Figura 2.4 – Exemplo de código RDF [RDG04].

A primeira linha do código contém a declaração XML, indicando que o documento está em conformidade com a especificação 1.0 da XML. O elemento base de um documento RDF, o *rdf:RDF*, é definido entre as linhas 2 e 13. A linha 2 contém a referência para o *namespace* RDF, representado por *xmlns:rdf*, enquanto as linhas 3 e 4 referenciam outros *namespaces*, específicos do exemplo. Esses *namespaces* definem os vocabulários utilizados. O elemento *rdf:Description*, na linha 5, identifica um recurso que tem seus metadados descritos no local indicado pelo atributo *about*. A linha 6 possui o atributo *title* que descreve o título do recurso. A linha 7 tem o atributo *editor*, definido até a linha 11, contendo outro recurso, presente na linha 8. Este é descrito pelos atributos *fullname* e *homePage*, estando o

último localizado na linha 9. Ainda nessa linha, *resource* define que *homePage* também é recurso, e indica o local do mesmo.

O RDF, por ser genérico, não faz associações sobre domínios específicos de aplicação. Para isso, usa-se o RDFS, especificando a semântica do domínio. Segundo [DAC03], o RDFS é um conjunto simples de classes e propriedades RDF para a definição de novos vocabulários RDF. Os elementos básicos de um RDFS são:

- Classes: definem tipos de objetos. Semelhantemente às linguagens orientadas a objetos, uma classe é definida como um grupo de elementos com características comuns.
- Propriedades: são definidas globalmente, ou seja, não são encapsuladas como atributos nas definições das classes. Existem propriedades para a definição de relações e para a definição de restrições de propriedades.

O RDF e o RDFS permitem a representação de alguns conhecimentos ontológicos, mas possuem várias limitações. Assim, o W3C recomenda o uso da linguagem OWL para a construção de ontologias.

2.4.1.2 OWL

A linguagem OWL, segundo [OWL04], é projetada para ser usada quando a informação contida em um documento precisa ser processada por aplicações, contrariando as situações em que o conteúdo necessita ser apenas apresentado aos humanos. Ela oferece mais facilidades para expressar significados e semânticas do que XML, RDF e RDFS, indo além dessas linguagens na habilidade em representar conteúdos interpretados por máquina na Web.

Comparando as linguagens de ontologias existentes com a OWL, verifica-se que esta adiciona mais vocabulário para descrever propriedades e classes: relações entre classes, cardinalidade, igualdade, tipagem mais rica de propriedades, características de propriedades, classes enumeradas, entre outros. Ela dispõe de três sublinguagens, com níveis crescentes de expressividade:

- OWL *Lite*: dá suporte àqueles usuários que, inicialmente, necessitam de uma classificação hierárquica e restrições simples.
- OWL DL: dá suporte àqueles usuários que buscam a máxima expressividade enquanto se provê a completude computacional, ou seja, garante-se que todas as conclusões são computáveis, e que buscam decidibilidade, isto é, todas as computações terminam em um

tempo finito. Ela contém todas as construções da linguagem OWL, mas só podem ser usadas sob certas restrições.

- OWL *Full*: destina-se aos usuários que buscam a máxima expressividade e a liberdade sintática do RDF, sem nenhuma garantia computacional. Possibilita uma ontologia para intensificar o significado de vocabulários pré-definidos, como o RDF e a OWL.

O exemplo demonstrando um trecho de código OWL, apresentado na Figura 2.5, é descrito a seguir. É definida uma ontologia de "vinhos" que importa uma ontologia de "alimentos", sendo declarado um tipo de vinho chamado *Vintage*.

```
1.  <?xml version="1.0"?>
2.  <rdf:RDF ... >
3.    <owl:Ontology rdf:about="">
4.      <rdfs:comment>An example OWL ontology</rdfs:comment>
5.        <owl:priorVersion>
6.            <owl:Ontology rdf:about="http://www.w3.org/TR/2003/CR-owl-
                                guide-20030818/wine"/>
7.        </owl:priorVersion>
8.        <owl:imports rdf:resource="http://www.w3.org/TR/2003/PR-owl-
                               guide-20031209/food"/>
9.            ...
10.     <rdfs:label>Wine Ontology</rdfs:label>
11.   </owl:Ontology>

12.         ...

13.   <owl:Class rdf:ID="Vintage">
14.     <rdfs:subClassOf>
15.       <owl:Restriction>
16.         <owl:onProperty rdf:resource="#hasVintageYear"/>
17.         <owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">1
18.         </owl:cardinality>
19.       </owl:Restriction>
20.     </rdfs:subClassOf>
21.   </owl:Class>

22.         ...

23. </rdf:RDF>
```

Figura 2.5 – Exemplo de código OWL [OWL04].

As linhas 1 e 2 contêm os mesmos elementos explicados no exemplo de RDF, mostrando que o elemento *rdf:RDF* é finalizado na linha 23. Entre as linhas 3 e 11 é definido o cabeçalho de ontologia pelo elemento *owl:Ontology*, indicando que o bloco descreve a ontologia corrente. A linha 4 define um comentário. O elemento *owl:priorVersion*, definido entre as linhas 5 e 7, contém uma referência para outra ontologia, identificando que a mesma é uma versão de maior prioridade. Na linha 8, o elemento *owl:imports* é declarado para referenciar outra ontologia OWL contendo definições, cujo o significado é considerado parte do significado da ontologia que a importa. O elemento *rdfs:label*, na linha 10, rotula o recurso

com um nome para a compreensão humana. Entre as linhas 13 e 21 é declarada uma classe OWL, em um bloco contendo os axiomas dela. O atributo *rdf:ID*, na linha 13, define o identificador de um elemento, neste exemplo, chamado *Vintage*. A propriedade *rdfs:subClassOf*, definida entre as linhas 14 e 20, indica que o recurso é uma subclasse de uma classe anônima, pois não existe referência à classe pai. Entre as linhas 15 e 19 é declarado o elemento *owl:Restriction*, um tipo especial de axioma de classe, especificando uma classe anônima abrangendo todos os indivíduos que satisfazem uma restrição. A linha 16 relaciona a restrição a uma propriedade específica por meio do elemento *owl:onProperty*. Por fim, as linhas 17 e 18 apresentam a restrição de cardinalidade *owl:cardinality*, relacionando uma classe de restrição a um valor de dados *nonNegativeInteger*, pertencente ao intervalo de tipos de dados do XML *Schema* [XML06]. O valor "1" indica que a classe em questão contém todos os indivíduos que possuem, semanticamente, um valor distinto.

2.4.2 Inferência em Ontologias

Para se recuperar o conhecimento presente em uma ontologia, de acordo com a sua semântica, é necessária a utilização de um mecanismo de inferência. Por meio de regras de inferência, consegue-se derivar novos fatos baseados em fatos existentes, ou seja, inferem-se novas informações.

O mecanismo *Pellet* [PEL07] é uma das diversas ferramentas utilizadas para o auxílio computacional ao processo de inferência. Originalmente desenvolvido no laboratório *Mindswap* da Universidade de Maryland, o *Pellet* é um mecanismo de inferência de código aberto para a linguagem OWL DL, implementado em *Java* [JAV07]. Dentre as suas características, destacam-se:

- Suporte à inferência com toda a expressividade da OWL DL. Atualmente, vem sendo estendido para suportar a OWL 1.1.
- Fornece diversas maneiras de acesso (interfaces) às suas capacidades de inferência, como *OWLSight* [SIG07], *Jena*, *Manchester* OWL-API [API07] e SWOOP [SWO07].
- Inferências por tipos de dados de XML *Schemas*.
- Suporte a regras.
- Inferência incremental.
- Análise, reparação e depuração de ontologias.

2.5 Considerações sobre o Capítulo

Ao longo deste capítulo, foram apresentados conceitos importantes para a compreensão e construção do *framework* proposto. É necessário entender as definições de agentes de software e de SMAs porque a arquitetura é aplicada dentro do paradigma de programação orientado a agentes. Quanto ao desenvolvimento de SMAs, o *framework SemantiCore* é introduzido para antecipar o entendimento da sua estrutura antes do exemplo de aplicação da arquitetura.

Na elucidação da WS, o uso de ontologias para a representação semântica de conhecimento foi explicado porque a arquitetura proposta as utiliza na descrição das informações de contexto. Sobre este último, sua definição será apresentada mais adiante. Quanto às linguagens para a descrição de ontologias, enfatizou-se a OWL por ser recomendada pelo W3C. A ferramenta *Pellet*, para a inferência em ontologias, foi escolhida por ser compatível com diversos outros mecanismos de inferência.

3 ADAPTAÇÃO EM AGENTES DE SOFTWARE

Observando as afirmações apresentadas no capítulo anterior, sobre a conceituação de um agente de software, verifica-se que em nenhum momento o mesmo é caracterizado como uma entidade constituída, em sua forma nativa, por capacidades adaptativas. Quando se fala em uma entidade inteligente, sustentar a idéia de atuar conforme os objetivos definidos e de acordo com o conhecimento relativo ao seu ambiente e às suas ações poderia remeter ao conceito de adaptação, pois o agente estaria modificando, e assim adaptando, o seu comportamento, conforme a manipulação dos seus objetivos e conhecimentos. Mas, seguindo o curso da discussão acerca da definição de agentes de software, o conceito de adaptação deixa de ser um consenso entre os pesquisadores na área de SMAs.

3.1 Conceituação de Adaptação em Agentes

Inicialmente, pode-se apresentar uma definição sucinta e clara, defendida por [MAE94], que afirma ser adaptativo o agente capaz de se aprimorar ao longo do tempo, isto é, quando o agente se torna melhor no alcance de seus objetivos por meio da experiência. Além disso, é possível notar a existência de uma seqüência contínua de maneiras pelas quais um agente pode ser adaptativo, desde ser capaz de se adaptar flexivelmente a mudanças pequenas e de curto prazo no ambiente, até lidar com mudanças mais significantes e de longa duração, ou seja, sendo capaz de mudar e melhorar o seu comportamento ao longo do tempo.

Para [HUH98], a adaptação é uma das importantes propriedades de agentes e, para possuí-la, estes precisam ser persistentes e capazes de aprender. [WEI95] vai mais além, destacando que observar as noções de adaptação e aprendizado em SMAs, com um olhar mais profundo, permite verificar que não existe uma distinção explícita entre elas. Ao contrário, pode-se assumir que a "adaptação" é coberta pelo "aprendizado". Sendo assim, o conceito de aprendizado deveria ser entendido como o afirmado por [RUS03], no qual as percepções não deveriam ser usadas apenas para agir, mas também para melhorar a habilidade do agente nas suas ações futuras. O aprendizado ocorreria conforme o agente observasse suas interações com o mundo e seus próprios processos de tomada de decisão.

Na visão de [FER99], existe uma adaptação estrutural e comportamental de uma sociedade de agentes. Para realizá-la, é preciso entendê-la sob dois pontos de vista: ou como uma característica individual dos agentes – falando-se de aprendizado – ou como um processo

coletivo colocando mecanismos reprodutivos em ação, chamado evolução. Continuando o raciocínio sobre adaptação estrutural, é importante apresentar a definição de [SPL03], afirmando que o comportamento reativo de um agente é algumas vezes apelidado de adaptativo, quando um objetivo ou plano é abandonado e outro com melhor ajuste à situação corrente é adicionado. Mas se defende que a adaptação de um agente ocorre com mudanças estruturais, incluindo conhecimento e fatos disponíveis. Além disso, uma assistência externa pode ser requisitada para se executar as modificações necessárias, como, por exemplo, uma *factory*[3] de agentes. Diz-se que um processo de adaptação tem um escopo, e este define até que ponto são adaptadas as partes de um agente. Assim, na pesquisa de adaptação de [SPL03], configuram-se três escopos: adaptação de conhecimento e fatos; adaptação da interface de um agente, normalmente relacionada com a adaptação da interface do agente à plataforma corrente; e adaptação de uma funcionalidade do agente, não comumente disponível.

Por fim, [IMA04] apresenta três categorias de adaptação, baseadas no relacionamento entre a adaptação interna e o comportamento externo do agente. São elas: (i) adaptação interna – quando os sistemas internos usados por um agente são adaptativos, mas suas ações externas não refletem qualquer comportamento adaptativo; (ii) adaptação externa – quando os sistemas internos não são adaptativos, mas as ações externas do agente refletem o comportamento adaptativo, e; (iii) adaptação completa – quando os sistemas internos são adaptativos e as conseqüências das adaptações são refletidas nas ações externas do agente. Assim, os agentes adaptativos são considerados sistemas ou máquinas que utilizam metodologias computacionais inferenciais ou complexas, para modificar os parâmetros de controle, bases de conhecimento, metodologias de resolução de problemas, cursos de ações, ou outros objetos, a fim de cumprir um conjunto de tarefas que é de interesse do usuário.

Analisando as diversas definições de agentes adaptativos e considerando o escopo deste trabalho, caracterizou-se a adaptação como: **um processo individual de um agente capaz de se aprimorar ao longo do tempo, por meio de pequenas modificações estruturais e comportamentais, manipulando conhecimento, e não se desvinculando do seu objetivo final de execução**. A afirmação "pequenas modificações" refere-se a alterar o agente de maneira a não o transformar em outro, mantendo uma quantidade mínima de elementos que o caracterizam como o mesmo agente original. "Modificação estrutural" refere-se às alterações das partes internas do agente, como regras, ontologias, planos, crenças, entre outras. "Modificação comportamental" refere-se às alterações no conjunto de reações que se podem observar no agente, estando este em seu ambiente, e em dadas circunstâncias.

[3] Padrões de projeto são explicados em [GAM95].

3.2 Problema

Atualmente, como afirmado por [JUA03], os SMAs são constituídos por inteligência e adaptabilidade, inseridos em ambientes abertos. Sistemas inteligentes executam tarefas que necessitam de grande quantidade de conhecimento e raciocínio sobre este último. Enquanto isso, sistemas adaptativos precisam sentir as mudanças na sua utilização e no seu ambiente, alterando o seu comportamento em tempo de execução para a melhoria de resultados.

Assim, sabe-se que o problema existente na construção de agentes adaptativos se refere a: **Como permitir a um agente em um ambiente como a Web Semântica – aberto, diverso, e baseado em ontologias para a representação de informação – executar e se aprimorar ao longo de muito tempo, sendo útil e buscando alcançar os seus objetivos?**

Desse problema, compreende-se que um agente deve ser capaz de manipular conhecimento para se adaptar às mudanças, sem precisar parar a sua execução. Nesse sentido, o *framework* proposto permite que um agente manipule conhecimento por meio de ontologias para se adaptar conforme as informações do contexto de execução e as políticas de adaptação definidas. Os conceitos de contexto de execução e política de adaptação são explicados no Capítulo 4.

A seguir, destacam-se as principais características (benefícios) do *framework* relacionadas ao tratamento do problema existente:

- Existem diversas arquiteturas para agentes de software que introduzem estratégias específicas de adaptação. Levando-se em conta a utilização de ontologias para a manipulação de conhecimento, o *framework* permite acoplar essas diversas estratégias. Exemplos dessas estratégias aparecem na seção 3.3, de trabalhos relacionados.
- Como o *framework* abstrai a arquitetura estrutural do agente, é possível inseri-lo, virtualmente, em qualquer tipo de arquitetura. Para isso ocorrer, basta acoplar o *framework* ao agente.
- As plataformas disponíveis para o desenvolvimento de SMAs não oferecem um suporte nativo à definição interna dos agentes, principalmente em relação à utilização de arquiteturas que permitam a adaptação estrutural e comportamental dos mesmos. O *framework* aparece na tentativa de suprir essa deficiência, podendo ser utilizado diretamente nos agentes, ou integrado a essas plataformas. A integração do *framework* às plataformas não é escopo deste trabalho.

- Quando não se quer instanciar o *framework* na criação de agentes adaptativos, é possível utilizar o seu Modelo Conceitual como um guia para o desenvolvimento desses agentes, principalmente por ele conter os principais conceitos encontrados nos diversos processos de adaptação pesquisados.

Objetivando apresentar o estado da arte em adaptação de agentes de software, a próxima seção apresenta alguns trabalhos relacionados ao *framework* aqui proposto.

3.3 Trabalhos Relacionados

Na literatura, encontraram-se diversas arquiteturas de adaptação em agentes de software, mas apenas algumas são descritas aqui. Todas elas apresentam, dentre os seus objetivos, uma maneira de melhorar a execução dos agentes dinamicamente. Apesar disso, mostraram-se aplicadas para a resolução de problemas dentro de domínios específicos. Assim, as arquiteturas foram analisadas principalmente para (i) o melhor entendimento sobre o conceito de adaptação, e para (ii) facilitar a definição dos componentes básicos de adaptação.

Esta seção apresenta apenas uma parte da extensa quantidade de trabalhos relacionados à construção de agentes adaptativos, deixando-se a discussão sobre a definição dos componentes básicos de adaptação para o Capítulo 4, mais precisamente na seção 4.1, sobre a análise dos modelos de adaptação.

Avaliando o enfoque adaptativo, [SCH05] apresenta uma comparação entre duas populações, ou sociedades, de agentes de comércio, uma estática e outra adaptativa. A primeira teria segmentações de dados classificados como conjuntos de treinamento e teste, caracterizando a evolução estática. Enquanto que a segunda teria seus agentes evoluindo indefinidamente, sem uma parada após a fase de aprendizado, caracterizando a capacidade de adaptação. Concluída a comparação, verificou-se que a abordagem adaptativa oferece vantagens sobre a alternativa estática para otimizar as estratégias de negócios ou melhorar as populações usando análise técnica. Uma representação geral de como a população de agentes evolui é mostrada na Figura 3.1.

Em uma proposta com idéias evolucionárias, [CIC99] afirma que a adaptação eficiente é constituída por predição, controle e *feedback*. É apresentado um *framework* unificado, no qual lógica *fuzzy*[4] e computação genética[5] são inseridas como meios sinérgicos para a

[4] A lógica *fuzzy*, ou lógica difusa, é uma generalização da lógica booleana, pois admite valores lógicos intermediários entre a falsidade e a verdade. Como existem várias formas de se implementar um modelo *fuzzy*, a lógica difusa deve ser vista mais como uma área de pesquisa

definição de agentes que aprendem. Por meio de uma implementação em *Java*, um ecossistema simples é testado, confirmando a habilidade de aprendizagem dos agentes.

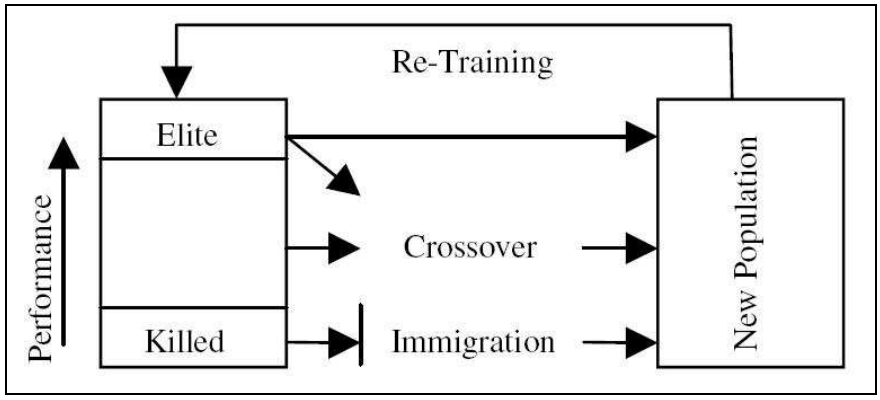

Figura 3.1 – Evolução das populações de agentes [SCH05].

Quanto a propostas utilizando ontologias, [ERD03] introduz um *framework* desenvolvido para a instanciação de agentes adaptativos. O termo adaptativo é utilizado para agentes que adaptam seu conhecimento interno e comportamento quando uma ontologia é mudada ou uma nova é adicionada em tempo de execução no sistema. O *framework* apresenta uma camada adicional contendo as ações que possibilitam aos agentes instanciados se adaptarem dinamicamente às ontologias substituídas no ambiente. A Figura 3.2 apresenta a arquitetura em camadas. Na opinião do autor, seu *framework* é essencial para o agente inserido na Web Semântica. Questões relativas aos conceitos de ontologias e Web Semântica podem ser vistos no Capítulo 2.

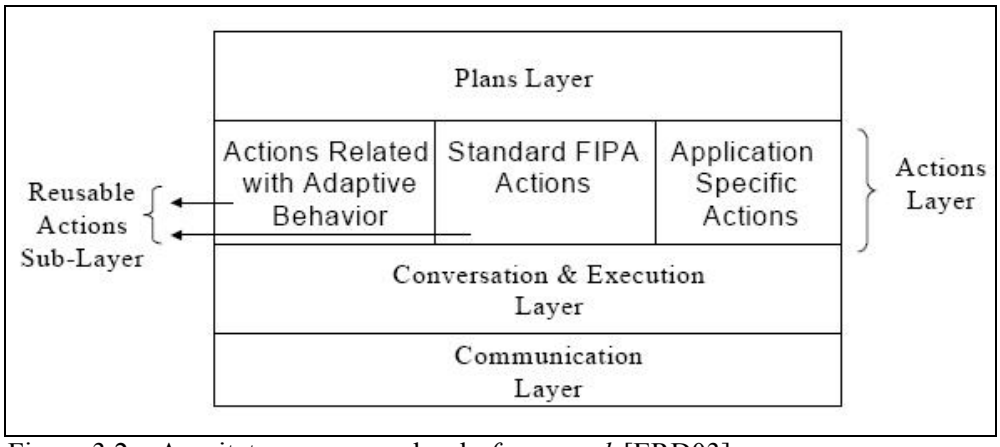

Figura 3.2 – Arquitetura em camadas do *framework* [ERD03].

sobre o tratamento da incerteza, ou uma família de modelos matemáticos dedicados ao tratamento da incerteza, do que uma lógica propriamente dita.

[5] A computação genética é caracterizada pelo uso de materiais genéticos e princípios biológicos de reprodução de proteína no projeto de sistemas computacionais.

Ainda no estudo de ontologias, [AMA06] propõe a arquitetura GAMA, para auto-adaptação de agentes móveis em ambientes computacionais difusos. Os ambientes de execução dos agentes e as estruturas destes últimos usam ontologias. Um processo de raciocínio é realizado sobre essas ontologias a fim de possibilitar a tomada de decisão sobre a reconfiguração solicitada, para ajustar a estrutura do agente – adicionando, substituindo, ou removendo componentes – ao seu novo contexto de execução. A Figura 3.3 apresenta a arquitetura GAMA.

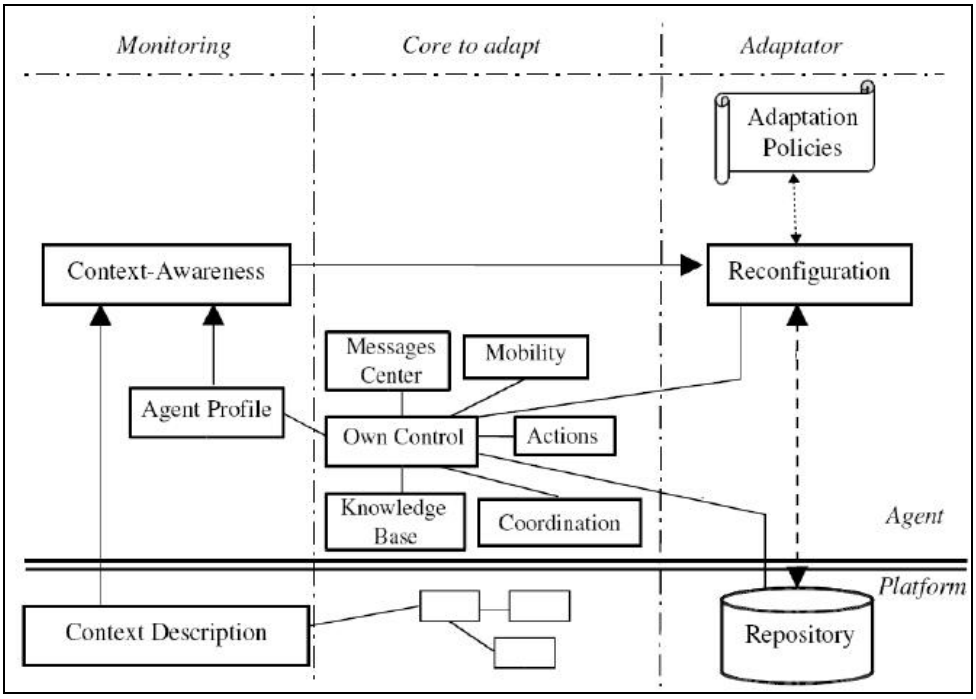

Figura 3.3 – A arquitetura GAMA [AMA06].

[YAM01] apresenta um *framework* de aprendizado por reforço multiagente, para adquirir comportamento adaptativo, pela geração e coordenação de cada objetivo de aprendizado de maneira interativa entre os agentes. Os objetivos de aprendizado são tratados como sinais de reforço que podem ser comunicados entre os agentes, e são propostas regras de motivação para integrar esses sinais dentro de um valor de recompensa.

Outra abordagem interessante pode ser vista quando [LAI97] introduz uma arquitetura que caracteriza o processo de adaptação em três níveis de conhecimento e controle. Esses níveis são: o nível de reação para a resposta reativa, um nível deliberativo para o comportamento dirigido a objetivos, e uma camada de reflexão para a deliberação de planos e decomposição de problemas. Essa abordagem demonstra adaptação nos níveis de reação e deliberativo. No primeiro, a teoria de domínio é modificada e estendida para determinar as

ações necessárias às situações similares. No segundo, o agente usa o conhecimento reflexivo para atualizar seu curso de ação. Os três níveis são mostrados na Figura 3.4.

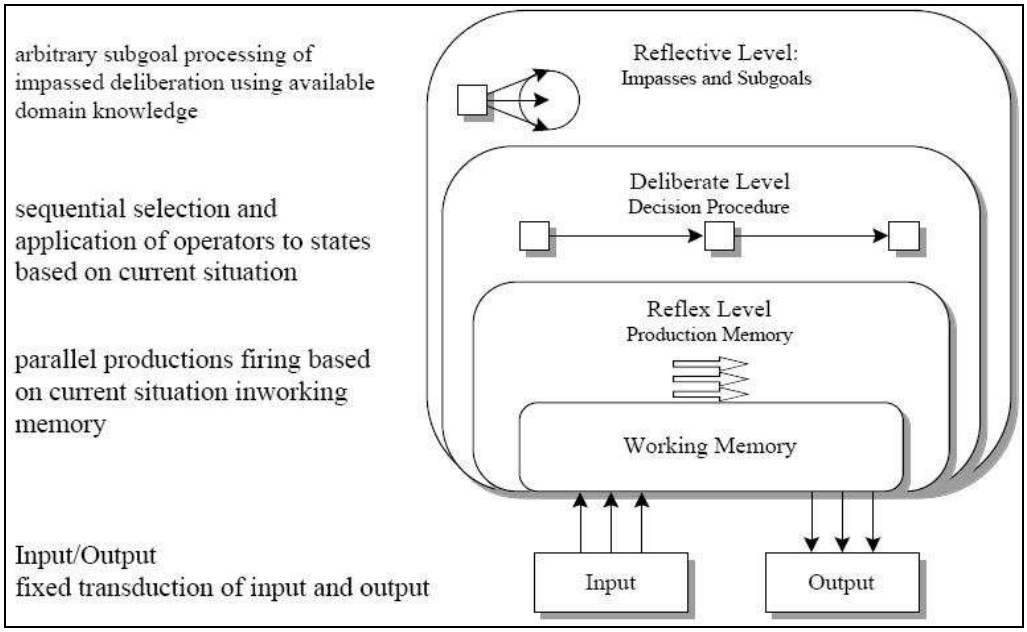

Figura 3.4 – Os três níveis da arquitetura [LAI97].

Aproximando as abordagens da vida humana, [JAR05] expõe que na psicologia a adaptação é um processo dinâmico no qual o comportamento e os mecanismos fisiológicos de um indivíduo mudam continuamente para ajustá-lo às variações do ambiente. É possível determinar que o processo de aprendizado, de um ponto de vista psicológico, preocupa-se com dois aspectos: um aspecto individual, em que a aquisição de conhecimento é feita sozinha, e um aspecto coletivo, em que o aprendizado é fortemente influenciado pelas interações dos indivíduos. Com isso, é apresentado um modelo de comunicação e trocas baseadas em protocolo para um agente adaptativo. Desenvolvem-se três passos sucessivos, necessários para um agente se integrar em um novo ambiente: o primeiro passo, de "escuta passiva" para todas as mensagens trocadas entre agentes; o segundo passo, de questionamento aos outros agentes sobre o significado das palavras ouvidas durante a primeira fase, e; o terceiro e último passo, de aprendizado e aprimoramento da nova ontologia do agente.

[WIK04] propõe uma arquitetura de agente adaptativo inspirada pelo comportamento humano, psicologia e ciência do cérebro – esta é um ramo da neurociência[6]. Seu objetivo principal é permitir ao agente aprender o comportamento exigido para a dinâmica do ambiente, ao invés de selecionar uma ação de uma lista pré-definida. A arquitetura proposta possui quatro camadas (também visualizadas na Figura 3.5):

[6] A neurociência é uma prática interdisciplinar resultante da interação de disciplinas biológicas que estudam o sistema nervoso, especialmente a anatomia e a fisiologia do cérebro humano.

- Herança: geralmente as ações primitivas tomadas para a sobrevivência surgem como neurônios pré-interligados, e esse fenômeno é capturado nesta camada.
- Treinamento: o aprendizado obtido por meio de educação ou treinamento é mapeado por esta camada, construída sobre a anterior. Como a evolução do cérebro é alcançada por meio de informação definida geneticamente e de aprendizado, esta camada e a de herança modelam o processo dessa evolução.
- Experiência: com maturidade, os humanos ganham experiência sobre o treinamento, o qual permite a tomada de decisão adaptativa. Isso é denotado por esta camada, e ela faz o agente mais adaptativo para o ambiente.
- Imprevisão: com as três camadas anteriores, o agente se torna capaz de reagir à natureza "inesperada" de um ambiente dinâmico. Isso é mapeado por esta última camada, e ela, com o passar do tempo, pode ser absorvida pela camada de experiência.

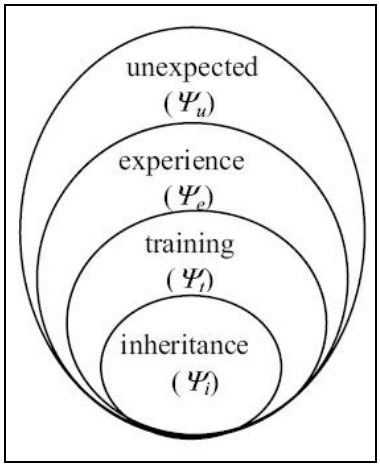

Figura 3.5 – Arquitetura de quatro camadas [WIK04].

[LER03] descreve um mecanismo geral para a adaptação em SMAs nos quais os agentes modificam seus comportamentos baseados na memória de eventos passados. Essas mudanças de comportamento podem ser obtidas pelas dinâmicas do ambiente ou aparecerem como reação às ações de outros agentes. A memória é utilizada para estimar o estado global do sistema a partir de observações individuais dos agentes, permitindo que estes ajustem suas ações. Apresenta-se também um modelo matemático das dinâmicas do comportamento coletivo em tais sistemas, que é utilizado para se estudar a formação de coalizão adaptativa em mercados eletrônicos.

Por fim, [LER07] propõe um modelo de agente adaptativo construído a partir de componentes reusáveis com alto grau de modularidade, que implementa mecanismos não-funcionais, tais como comunicação, mobilidade ou habilidades de adaptação. Afirma-se que a

adaptação dinâmica dos mecanismos internos de agentes móveis é necessária para a eficiência dos mesmos. Para implementar esses mecanismos, são usados, por um lado, os princípios de separação de interesses, a separação de níveis e a modularidade, e por outro lado, as tecnologias baseadas em componentes para o alcance de reusabilidade, de configuração e suporte à adaptação dinâmica.

3.4 Considerações sobre o Capítulo

Neste capítulo, conceituou-se a adaptação em agentes de software, apresentou-se o problema relativo a agentes adaptativos, e se descreveu alguns trabalhos relacionados aos diversos tipos de processos adaptativos em agentes. Verificou-se nas pesquisas analisadas que a adaptação existe quando algum mecanismo possibilita o aprimoramento das características de um agente, não se rejeitando a chance desse agente ter seu funcionamento prejudicado. Muitos autores também defendem a relação direta entre adaptação e aprendizado, demonstrando a importância do uso de conhecimento para a realização de qualquer tarefa.

Um ponto visivelmente divergente entre as pesquisas é a definição de adaptação estrutural e comportamental. Torna-se difícil compreender como uma mudança estrutural não altera o comportamento do agente, ou como é possível alterar a conduta do agente sem haver uma modificação em alguma estrutura do mesmo, por menor que seja essa estrutura. Assim, é coerente afirmar que uma adaptação estrutural não altere o resultado de uma tarefa do agente, mas só o fato de mudá-lo internamente para melhorar a sua execução caracteriza uma mudança comportamental visivelmente imperceptível.

Quanto aos trabalhos relacionados, observa-se que todos buscam solucionar problemas dentro de domínios específicos. Os agentes possuem processos fixos de adaptação, como a utilização de, por exemplo, lógica *fuzzy*, computação genética e atualização de ontologias. A adaptação está sempre sobre a estrutura do agente, e não sobre o próprio processo adaptativo. Quanto às propostas e limitações de cada trabalho, a Tabela 3.1 complementa o conteúdo exposto ao longo deste capítulo.

Quando se utiliza o termo “processo de adaptação”, quer-se dizer todas as ações necessárias para se modificar a estrutura de um agente. Por exemplo, pensando-se em um processo fixo que funcione com lógica *fuzzy*, todas as vezes que um agente precisa modificar a sua estrutura, ele a modifica por meio da aplicação da lógica *fuzzy* sobre o seu conhecimento. Agora, se no mesmo agente é inserida a adaptabilidade para os processos, em um momento a lógica *fuzzy* pode ser utilizada para a adaptação, conforme o seu

conhecimento, mas em outro momento o agente pode se beneficiar da computação genética, permitindo-se outros níveis de modificação estrutural e comportamental.

TABELA 3.1 – PROPOSTAS E LIMITAÇÕES DOS TRABALHOS RELACIONADOS

Artigo	Proposta	Limitações
[SCH05]	Comprova que uma sociedade com agentes adaptativos obtém melhores resultados na execução de atividades, comparados a agentes estáticos.	Foca-se apenas na otimização de estratégias de comércio ou de populações usando análise técnica.
[CIC99]	Descreve um modelo de computação concorrente massiva no qual a plataforma multiagente adapta o comportamento global por meio de lógica *fuzzy* e computação genética.	Os agentes só aprendem para adaptar por meio de lógica fuzzy e computação genética.
[ERD03]	Os agentes adaptam o seu conhecimento interno e comportamento quando uma ontologia é modificada ou uma nova ontologia é adicionada no sistema em tempo de execução.	Os agentes só adaptam por meio de modificações nas ontologias.
[AMA06]	Os agentes se adaptam estruturalmente por meio de inferência nas ontologias que descrevem o contexto de execução dos mesmos, detectando significantes mudanças no ambiente.	A adaptação ocorre apenas com a adição, substituição ou remoção de componentes constituintes do agente.
[YAM01]	Os agentes se adaptam por meio de objetivos de aprendizado representados como sinais de reforço, e por regras de motivação, incentivando a cooperação.	A adaptação ocorre apenas com a utilização de valores de recompensa, faltando uma representação mais completa de conhecimento.
[LAI97]	Os agentes possuem uma arquitetura dividida em três níveis para a adaptação, resolvendo problemas de aprendizado.	A adaptação ocorre apenas com a correção de erros no conhecimento do agente, relativo à teoria do domínio.

TABELA 3.1 – PROPOSTAS E LIMITAÇÕES DOS TRABALHOS RELACIONADOS

[JAR05]	Os agentes aprendem e se adaptam por meio de interações e comunicação em uma linguagem natural entre eles.	A adaptação ocorre apenas com a aquisição de novas palavras para as ontologias dos agentes.
[WIK04]	Os agentes possuem uma arquitetura dividida em quatro camadas e se adaptam por meio de uma Rede Neural Artificial Evolucionária.	A adaptação ocorre apenas com o uso de uma rede neural.
[LER03]	Os agentes se adaptam modificando o comportamento baseado na memória dos eventos passados.	A construção dos agentes precisa obedecer à propriedade de Markov, isto é, apenas o estado atual determina o estado futuro.
[LER07]	Os agentes se adaptam pela mudança dinâmica e autônoma dos seus componentes para se adequarem ao contexto de execução.	A adaptação ocorre apenas com a mudança de componentes.

4 MODELO DE ARQUITETURA PARA ADAPTAÇÃO EM AGENTES

Sabe-se da necessidade dos sistemas adaptativos precisarem sentir as mudanças na sua utilização e no seu ambiente, alterando o seu comportamento em tempo de execução para a melhoria de resultados, e assim permitindo aos agentes a manipulação de conhecimento para se adaptarem às mudanças sem precisarem parar a sua execução. Com isso, o presente trabalho propõe um *framework* que possibilita a adaptação estrutural e comportamental de agentes de software na Web Semântica, de acordo com o contexto composto por propriedades descritas, principalmente, por meio de ontologias.

A arquitetura proposta conta com os elementos básicos de adaptação identificados nos diversos processos de adaptação pesquisados, com pequenas variações na nomenclatura. Além desses elementos, foi introduzido o conceito de **núcleo de adaptação**, ou *AdaptionCore*, como no modelo a ser visto posteriormente, relacionado às funções de integração entre todas as partes do *framework*.

Para uma melhor compreensão das partes do *framework*, o mesmo foi dividido em cinco pacotes conceituais, apresentados aqui em ordem alfabética: *assessment* (avaliação), *core* (núcleo), *knowledge* (conhecimento), *percept* (percepção) e *point* (ponto). Posteriormente, na seção 4.2, apresenta-se a constituição de cada pacote. A Figura 4.1 ilustra a organização dos pacotes – o diagrama segue uma notação UML.

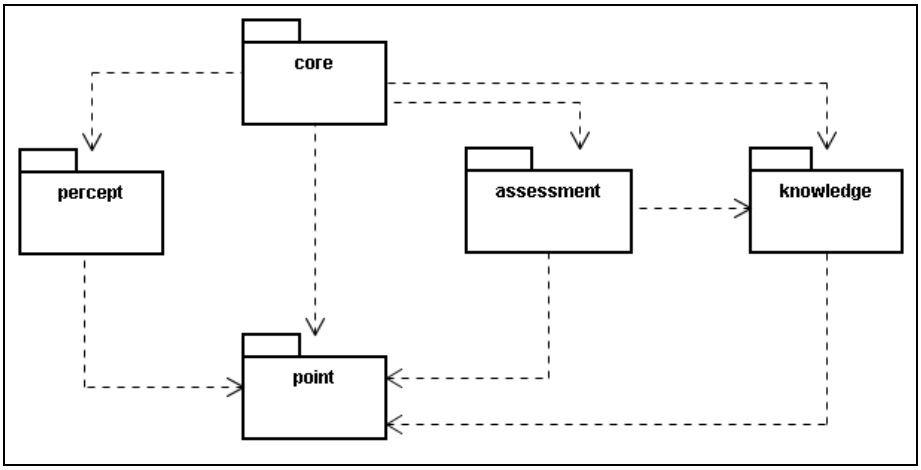

Figura 4.1 – Diagrama de Pacotes do Modelo Conceitual do *framework*.

É relevante expor o porquê de se utilizar conjuntamente os conceitos de *framework* e arquitetura neste trabalho. Quando se iniciou a construção do Modelo Conceitual, apresentado posteriormente, objetivava-se validá-lo como uma arquitetura que incorpora capacidades adaptativas a agentes de software. No desenvolvimento do trabalho, avaliou-se apenas o funcionamento de um *framework* que implementa a maior parte dessa possível arquitetura,

apresentando-se o primeiro passo para alcançá-la no futuro. Assim, somente após a validação final de que toda a estrutura proposta é uma solução genérica para agentes adaptativos, será possível defini-la como uma arquitetura.

Sobre a estruturação deste capítulo, a seção 4.1 mostra como os elementos básicos de adaptação foram identificados. Em seguida, a reunião e relacionamento desses elementos são explicados pela descrição do Modelo Conceitual, bem como as suas funções em relação aos pacotes definidos.

4.1 Análise dos Modelos de Adaptação

Analisando o conhecimento compilado dos diversos trabalhos em agentes adaptativos, buscou-se definir elementos básicos que caracterizam o processo de adaptação em agentes de software. Dessa análise, sete elementos, tratados aqui como conceitos, surgiram com maior importância: **avaliação**, **conhecimento**, **contexto**, **percepção**, **política**, **ponto de adaptação** e **restrição**. O uso de cada elemento nos trabalhos relacionados é comparado com o uso definido para o *framework* proposto e, ao final desta seção, a Tabela 4.1 reúne essas comparações.

Diversos sistemas possuem estimativas de resultados por meio da utilização de valores. A qualificação dos resultados correntes é importante tanto para a organização de informações quanto para a melhoria de resultados futuros, por meio da identificação de problemas. Saber avaliar as conseqüências de uma tomada de decisão pode ser crucial para se manter o funcionamento de um software. Como exemplo de trabalho pesquisado, [IBA98] mostra um sistema no qual os agentes se adaptam por meio de algoritmo genético e rede neural. A configuração dos agentes em relação à população é avaliada pela interação entre o ambiente e as características genéticas, enquanto a configuração em relação ao indivíduo é avaliada pelo algoritmo de rede neural. As avaliações definem a escolha das melhores configurações a serem mantidas. Assim, conclui-se que o conceito de **avaliação** é relevante para a arquitetura de adaptação.

O **conhecimento** é outro conceito importante para a arquitetura de adaptação. Segundo [FER99], o conhecimento é toda a informação necessária a um ser humano (ou máquina), organizada de uma maneira que permita o cumprimento de uma tarefa considerada complexa. Um agente precisa armazenar todas as informações relevantes para conseguir se adaptar. Como sempre existe um mínimo de informação relevante que precisa ser processado pelo agente, é evidente que o conhecimento é conceito fundamental. Como exemplo de trabalho

analisado, [WIK04] apresenta uma arquitetura de agente adaptativo com quatro camadas de conhecimento, como visto anteriormente nos trabalhos relacionados (seção 3.3).

O contexto, segundo [GAR05], é qualquer informação sobre as circunstâncias, objetos, ou condições pelas quais um agente está rodeado, que são relevantes para a interação entre o agente e o ambiente computacional. Como exemplo de trabalho analisado, e também presente na seção 3.3, [AMA06] afirma ser o contexto uma informação sobre o ambiente de execução corrente de um agente móvel. Portanto, considerou-se o conceito de **contexto** para a arquitetura de adaptação.

Todo agente, como mostrado por Wooldridge em [WOO02], possui um sensor de entrada que possibilita a percepção de características do ambiente. Como exemplo de trabalho analisado, [RUS96] define seus agentes como programas que podem sentir o estado da rede na qual estão se movendo, em um contexto de agentes móveis. Assim, tornou-se clara a importância do conceito de **percepção** para a arquitetura de adaptação.

Qualquer tipo de processo precisa ter uma maneira de ser executado, ou seja, é necessário saber como realizá-lo. Como exemplo, [AMA05] mostra que o método para substituir componentes em um agente é determinado por algumas políticas de adaptação. Disso, percebe-se a necessidade do conceito de **política** de adaptação para a arquitetura, determinando como uma parte do agente pode ser adaptada.

O conceito de **ponto de adaptação** é importante para indicar exatamente a parte do agente que pode sofrer mudanças, conforme as informações de contexto, políticas e restrições. A definição do termo foi inspirada na técnica de Programação Orientada a Aspectos (AOP). A AOP[7] utiliza o conceito de ponto de execução, ou *join point*, para especificar em qual lugar de uma aplicação um código deve ser executado conforme um interesse definido. Como exemplo, não relacionado diretamente a agentes adaptativos, [VAY05] chama de ponto de adaptação as instruções presentes em locais no fluxo de controle de aplicações em que o comportamento de um componente pode ser modificado para tratar as mudanças no ambiente.

Por fim, no desenvolvimento de qualquer sistema, modelo ou arquitetura, encontram-se limitações que restringem o escopo de aplicação dos mesmos. Igualmente, um agente possui limitações nas suas partes constituintes que impossibilitam certas adaptações. Como exemplo, apresentado também nos trabalhos relacionados, [LER07] expõe com clareza as limitações dos agentes dentro da sua arquitetura. Desse modo, o conceito de **restrição** é importante para a arquitetura de adaptação.

[7] A AOP pode ser estudada e compreendida em documentos como [KIC97] e [MIL04].

TABELA 4.1 – COMPARAÇÃO ENTRE O USO DOS ELEMENTOS BÁSICOS DE ADAPTAÇÃO PRESENTES NOS TRABALHOS PESQUISADOS E O USO NO FRAMEWORK PROPOSTO

Elemento básico	Artigo	Uso do elemento no artigo	Uso do elemento no framework proposto	Relação entre os usos
Avaliação	[IBA98]	A avaliação é utilizada para se escolher as melhores configurações alcançadas nos agentes e propagá-las nas gerações seguintes.	A avaliação indica o valor de satisfação alcançado por uma adaptação.	Ambos são utilizados para garantir a permanência das melhores adaptações.
Conhecimento	[WIK04]	O conhecimento é o conjunto de dados interno ao agente e necessário para o aprendizado sobre o ambiente, permitindo a adaptação.	O conhecimento é composto pelas propriedades internas do agente e do ambiente, necessárias para a adaptação.	Ambos são utilizados para caracterizar o estado interno do agente e do ambiente.
Contexto	[AMA06]	O contexto é qualquer informação que pode ser usada para caracterizar a situação de uma entidade – entidade é uma pessoa, local ou objeto considerado relevante para a interação entre o usuário e a aplicação. Existem três níveis de contexto: físico, social e de usuário.	O contexto é qualquer informação sobre as circunstâncias, objetos, ou condições pelas quais um agente está rodeado, além das suas propriedades internas, que são relevantes para a interação entre o agente e o ambiente computacional.	Para o *framework*, não existe diferenciação entre as propriedades (informações) do ambiente e as propriedades internas do agente.
Percepção	[RUS96]	A percepção do agente é obtida por meio de sensores que permitem descobrir informações importantes sobre o ambiente, estabelecendo o estado externo. O foco está sobre três componentes do estado do ambiente: hardware, software e outros agentes.	A percepção do agente é obtida por meio de sensores que captam as propriedades internas e do ambiente.	Para o *framework*, a percepção envolve também sensores que captam as mudanças internas do agente.
Política	[AMA05]	A política é utilizada para definir uma maneira de selecionar e ligar ao agente os componentes a serem substituídos.	A política define como uma parte do agente pode ser adaptada.	Para o *framework*, a política define os passos de adaptação para qualquer parte do agente, não somente aplicada a componentes.

TABELA 4.1 – COMPARAÇÃO ENTRE O USO DOS ELEMENTOS BÁSICOS DE ADAPTAÇÃO PRESENTES NOS TRABALHOS PESQUISADOS E O USO NO FRAMEWORK PROPOSTO

Ponto de adaptação	[VAY05]	O ponto de adaptação é composto pelas instruções presentes em locais no fluxo de controle de aplicações em que o comportamento de um componente pode ser modificado para tratar as mudanças no ambiente.	O ponto de adaptação indica exatamente a parte do agente que pode sofrer mudanças.	Para o *framework*, as instruções presentes no ponto de adaptação são as políticas.
Restrição	[LER07]	Na conceituação de um estilo arquitetural, o artigo expõe a idéia de restrição. Assim, um estilo arquitetural seria um conjunto de regras que especifica como um sistema é composto, definindo as regras e restrições de composição e evolução que impõem os limites de adaptação.	Um agente possui limitações nas suas partes constituintes que impossibilitam a ocorrência de certas adaptações. As restrições especificam essas limitações.	Ambos são utilizados para restringir certos tipos de adaptação.

4.2 Descrição do Modelo Conceitual

Analisados e definidos os conceitos básicos para uma arquitetura de agentes adaptativos, partiu-se para a construção do Modelo Conceitual, apresentado na Figura 4.3 – a representação do modelo segue uma notação UML.

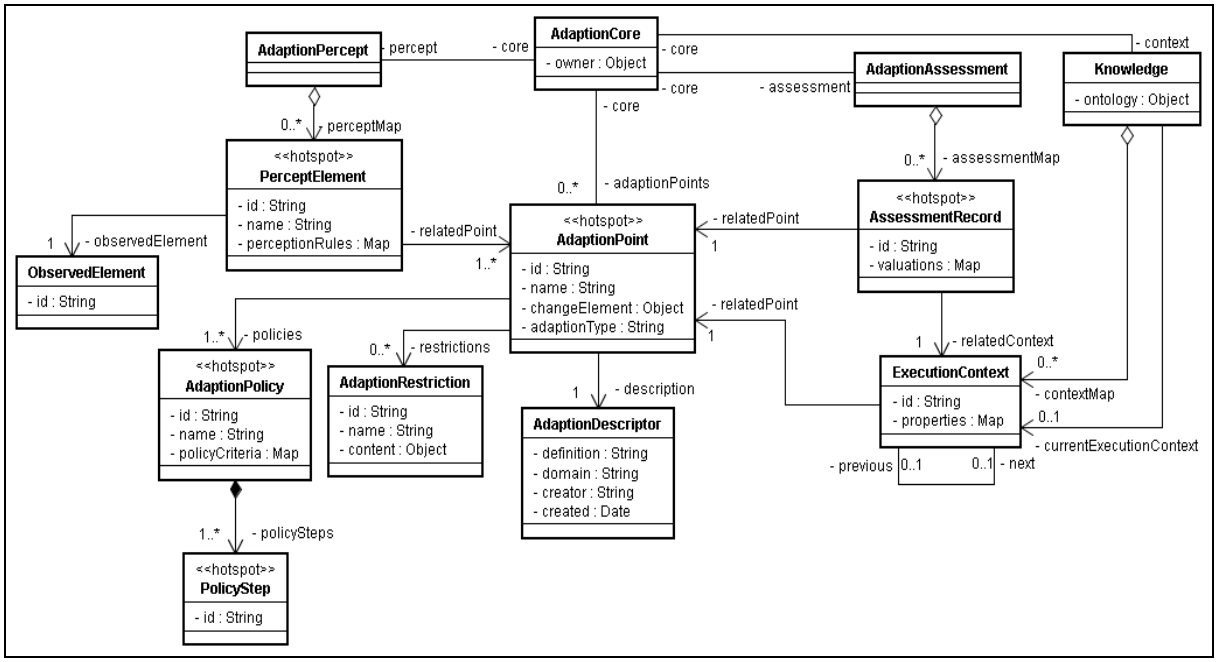

Figura 4.2 – Modelo Conceitual do *framework* proposto.

Observando a Figura 4.3, é possível identificar os sete conceitos descritos na seção 4.1, mais as classes adicionais que complementam o modelo. Percebe-se a utilização do estereótipo *hotspot* nas classes que são pontos de flexibilidade, ou seja, nas classes que podem ser estendidas por quem implementar agentes utilizando o *framework*. Destaca-se que os conceitos de contexto e de conhecimento utilizam estruturas ontológicas para as suas descrições. Essas estruturas são indispensáveis para a permanência dos agentes na Web Semântica. O contexto pode utilizar ontologias nas propriedades (atributo *properties*), enquanto o conhecimento agrega os diferentes contextos de execução em uma única ontologia (atributo *ontology*). Adicionalmente, o conceito de restrição também é constituído por uma ontologia que o descreve. Outras partes do *framework* poderiam se beneficiar do uso de ontologias, mas dependeriam da escolha do programador.

É preciso esclarecer que, além da descrição presente neste capítulo sobre o Modelo Conceitual, explicações adicionais, necessárias para um melhor entendimento do mesmo, aparecem no cenário de aplicação do *framework*, no Capítulo 5. Outro ponto importante, antes da descrição de cada classe e relacionamento, diz respeito aos tipos de arquitetura de agentes. Aqui, generaliza-se o Modelo Conceitual de um componente de um agente para guiar o desenvolvimento voltado à adaptação, e não definir um modelo de uma arquitetura que resolva um problema dentro de um domínio específico. Assim, quando o *framework* proposto for utilizado, tipos específicos de arquiteturas[8], como discutidos em [MAE94] ou classificados em [FER99], podem ser aplicados dentro dos conceitos definidos neste modelo.

4.2.1 Classes e Relações do Modelo Conceitual

Analisando novamente a Figura 4.1, e completando os pacotes com as classes definidas, chega-se ao Diagrama de Pacotes presente na Figura 4.4 (sem as relações entre as classes e seguindo uma notação UML).

A classe ***AdaptionCore*** (Núcleo de Adaptação) tem a função de manter referências às classes de agregação do modelo, criando um elo importante para a troca de informação, além de permitir a correta execução do *framework* quando implementado. Ela está diretamente associada às classes *AdaptionPercept*, *AdaptionPoint*, *AdaptionAssessment* e *Knowledge*, pelos respectivos papéis: *percept* (percepção), *adaptionPoints* (pontos de adaptação),

[8] [MAE94] expõe as características das arquiteturas de agente na seção 4 do seu artigo, intitulada *Characteristics of Agent Architectures*. Enquanto [FER99] apresenta diversos tipos de arquiteturas na seção 3.6 do seu livro, intitulada *Individual organisations*.

assessment (avaliação) e *context* (contexto). O seu atributo *owner* (proprietário) é uma referência ao agente com o qual o *framework* está integrado.

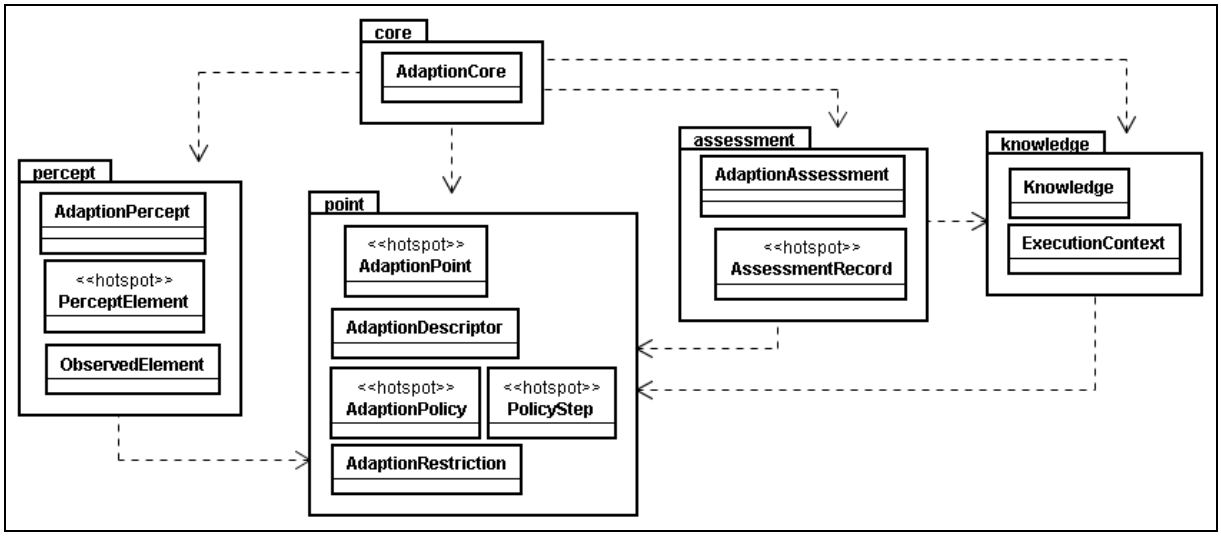

Figura 4.3 – Diagrama de Pacotes do Modelo Conceitual com as classes definidas.

A classe ***AdaptionPercept*** (Percepção de Adaptação) é uma agregação de objetos do tipo *PerceptElement*, e essa agregação é referenciada pelo papel *perceptMap* (mapa de percepções). Cada ***PerceptElement*** (Elemento de Percepção), que é um ponto de flexibilidade, associa-se a uma ou mais instâncias da classe *AdaptionPoint*, por meio do papel *relatedPoint* (ponto relacionado). Ele é um elemento que monitora uma parte do agente, por meio do papel *observedElement* (elemento observado), não podendo ser confundido com a parte do agente que sofre a adaptação. O atributo *perceptionRules* (regras de percepção) contém uma coleção de objetos e guarda as regras que, dependendo dos eventos detectados advindos do elemento observado, sinalizam o início de um processo de adaptação. Ainda existem os atributos de identificação do elemento de percepção: o *id* (identificador) e o *name* (nome).

O elemento observado é representado pela classe ***ObservedElement*** (Elemento Observado) e possui apenas um atributo de identificação, o *id*. O modelo mostra que cada *PerceptElement* só pode ter referência a uma instância de *ObservedElement*. Assim, cada elemento de percepção só observa um recurso no agente que está suscetível a sofrer uma adaptação.

A classe ***AdaptionPolicy*** (Política de Adaptação) é referenciada pela classe *AdaptionPoint* e define como será realizada a adaptação. A política, que é um ponto de flexibilidade, possui os atributos de identificação *id* e *name*, critérios para a utilização representados pelo atributo *policyCriteria* (critérios de política), e seus passos de execução

representados pelo papel *policySteps* (passos da política). Os critérios definem quais propriedades do contexto de execução devem ser consideradas para a aplicação da política, enquanto os passos definem as atividades que devem ser executadas durante o processo de adaptação.

A classe ***PolicyStep*** (Passo de Política) é um ponto de flexibilidade que define um passo de execução da política que a referencia, e é constituída por um atributo de identificação representado por *id*. Observa-se uma agregação por composição entre ela e a classe *AdaptionPolicy*, significando que passos de políticas não existem sem estas últimas. Além disso, qualquer política precisa ter, pelo menos, um passo de execução para estar completa.

A classe ***AdaptionRestriction*** (Restrição de Adaptação) é referenciada pela classe *AdaptionPoint*. Ela representa uma restrição para um ponto de adaptação, ou seja, limita uma possível adaptação por meio de informações descritivas. Essas informações podem ser construídas por meio de ontologias e estariam presentes no atributo *content* (conteúdo). Adicionalmente, a restrição possui os atributos de identificação *id* e *name*.

A classe ***Knowledge*** (Conhecimento) é uma agregação de objetos do tipo *ExecutionContext*, e essa agregação é representada pelo papel *contextMap* (mapa de contextos). Encontra-se ainda outra associação entre essas duas classes citadas, por meio do papel *currentExecutionContext* (contexto de execução corrente), indicando o contexto de execução criado para o processo completo de uma única adaptação. Seu atributo *ontology* representa a ontologia construída conforme o encadeamento dos diversos contextos de execução, e ela é explicada no Capítulo 5. Nota-se que a classe *AdaptionCore* referencia a classe *Knowledge* por meio do papel *context* (contexto). Isso ocorre porque neste trabalho o conceito de conhecimento está intimamente ligado ao conceito de contexto. Assim, as propriedades internas do agente e do ambiente são o conhecimento necessário para o *framework* efetuar as adaptações.

Cada instância da classe ***ExecutionContext*** (Contexto de Execução) contém tanto propriedades internas do agente quanto propriedades do ambiente no qual o agente está inserido, ambas representadas por um único atributo chamado *properties* (propriedades). As propriedades têm duas funções principais: (i) definir a escolha de uma política para a adaptação, e (ii) caracterizar o processo de adaptação para um ponto de adaptação específico. Essa caracterização é possível porque a classe *ExecutionContext* referencia a *AdaptionPoint* por meio do papel *relatedPoint*, e ela auxilia na avaliação das adaptações ocorridas e permite recuperar informações para uma análise futura, como uma espécie de histórico. Para este

último funcionar, a classe tem um auto-relacionamento, representado pelos papéis *previous* (anterior) e *next* (posterior), que encadeia os contextos de execução. Adicionalmente, a classe é composta por um atributo de identificação *id*.

A classe ***AdaptionAssessment*** (Avaliação de Adaptação) é uma agregação de objetos do tipo *AssessmentRecord*, e essa agregação é referenciada pelo papel *assessmentMap* (mapa de avaliações). Cada ***AssessmentRecord*** (Registro de Avaliação) é um ponto de flexibilidade e indica o valor de satisfação alcançado por uma adaptação aplicada sobre um ponto de adaptação, conforme os elementos observados, políticas, restrições e contexto de execução. A associação entre a classe *AssessmentRecord* e o ponto de adaptação existe por meio do papel *relatedPoint*, enquanto a associação daquela com o contexto de execução existe por meio do papel *relatedContext* (contexto relacionado). O registro de avaliação ainda é composto por um atributo de identificação *id*, e pelas avaliações dos pontos de adaptação, representadas pelo atributo *valuations* (avaliações).

A classe ***AdaptionPoint*** (Ponto de Adaptação) é um ponto de flexibilidade e tem a função de representar a parte do agente que sofre uma adaptação, por meio do atributo *changeElement* (elemento de mudança). Ela possui os atributos identificadores *id* e *name*, e um tipo de adaptação representado pelo atributo *adaptionType*, servindo apenas como um classificador para uma possível recuperação de informações. A classe *AdaptionPoint* ainda contém uma associação com a classe *AdaptionDescriptor*, por meio do papel *description*.

A classe ***AdaptionDescriptor*** (Descritor de Adaptação) fornece mais informações sobre o ponto de adaptação, contendo os seguintes atributos: *definition* – permite acrescentar uma definição descritiva sobre o ponto de adaptação; *domain* – permite determinar o domínio de utilização do ponto de adaptação; *creator* – permite especificar quem cria ou o que inicia o ponto de adaptação; *created* – permite armazenar a data de instanciação do ponto de adaptação. Essas informações podem ser consideradas metadados, e foram inspiradas no material desenvolvido pela organização *The Dublin Core Metadata Initiative*[9], dedicada a promover a adoção de padrões de metadados interoperáveis para a descrição de recursos. A utilização da classe descritora foi baseada em [LEM07].

Observando novamente a questão das avaliações das adaptações, é importante salientar que a utilização dessas juntamente com os contextos de execução, estes compostos por propriedades descritas, principalmente, por meio de ontologias, permitiria, por exemplo, a compilação das informações das constantes adaptações para a geração de conhecimento novo. Mas esse estudo não é contemplado por este trabalho.

[9] http://www.dublincore.org/

4.3 Considerações sobre o Capítulo

Este capítulo apresentou os aspectos gerais do modelo de arquitetura para a adaptação de agentes de software, desenvolvido e aplicado neste trabalho. Na descrição, buscou-se elucidar os conceitos dos elementos básicos encontrados nos diversos processos de adaptação pesquisados, possibilitando a construção do *framework* proposto. Cada elemento foi explicado por meio das classes definidas, e estas foram detalhadas em relação às suas funções, atributos e relacionamentos. Deixou-se o aprofundamento de uso dessas classes para o Capítulo 5, evitando-se a confusão entre os aspectos conceituais e os aspectos de implementação.

A análise dos modelos de adaptação é importante para se compreender como foram encontrados esses elementos básicos, justificando a criação das classes presentes no Modelo Conceitual. Adicionalmente, a análise complementa a descrição de alguns dos trabalhos relacionados discutidos no Capítulo 3, além de apresentar outras pesquisas.

Como o Modelo Conceitual foi detalhado apenas em relação às funcionalidades individuais de cada classe, é indispensável o esclarecimento do fluxo de execução do *framework*. Assim, o próximo capítulo utilizará a explicação de implementação do *framework* para também descrever esse fluxo.

Uma questão importante, observada da análise do Modelo Conceitual, é a ausência explícita da representação dos objetivos dos agentes. Seria interessante poder avaliar as adaptações associadas aos objetivos, e garantir a correta execução dos agentes. Em contrapartida, as plataformas para desenvolvimento de sistemas multiagentes – como JADE e *SemantiCore* – não suportam a modelagem dos objetivos. No *framework*, o contexto de execução pode armazenar os objetivos por meio das propriedades internas do agente.

5 PROTÓTIPO E IMPLEMENTAÇÃO DO FRAMEWORK USANDO UM CENÁRIO DE APLICAÇÃO

Finalizada a conceituação de adaptação em agentes de software e explicado o Modelo Conceitual do *framework* proposto, iniciou-se a sua implementação e o desenvolvimento de um cenário de aplicação para exemplificar a sua utilização. Essa fase caracterizou-se pela execução de três atividades principais:

- A implementação do *framework* proposto, utilizando-se a linguagem de programação *Java*.
- A caracterização e o desenvolvimento do cenário de aplicação.
- A implementação da simulação do cenário, utilizando-se a plataforma *SemantiCore*.

Neste capítulo, a implementação é explicada tentando-se seguir a mesma seqüência de apresentação existente no Capítulo 4, especificamente na seção 4.2, na qual é descrito o Modelo Conceitual do *framework*. Com isso, busca-se facilitar a identificação entre a descrição conceitual e a de implementação. A seqüência não é idêntica porque as partes pertencentes aos mesmos pacotes são agrupadas, diferenciando-se da ordem apresentada no capítulo anterior.

5.1 O Protótipo

Iniciada a implementação do protótipo, optou-se por manter uma estrutura de pacotes semelhante à definida no Modelo Conceitual, apenas com algumas complementações necessárias. Assim, os cinco pacotes definidos anteriormente permanecem, e são eles, para relembrar: *assessment*, *core*, *knowledge*, *percept* e *point*. Dentro de cada um deles estão inseridos três novos pacotes: *model* (modelo), *impl* (implementação), e *hotspot* (ponto de flexibilidade). Os pacotes mantidos contêm os classificadores[10] definidos como interface (todos são iniciados pela letra ‘i’ em maiúscula), garantindo que algumas propriedades sejam obrigatoriamente implementadas para a construção do protótipo. O pacote *model* contém as classes abstratas (todas são iniciadas pela palavra *Abstract*) com a maioria dos atributos e métodos básicos especificados. O pacote *impl* contém as classes que implementam as abstratas, além de classes adicionais que auxiliam o funcionamento das primeiras. Por fim, o

[10] O termo classificador é utilizado exatamente como definido na linguagem UML e pesquisado em [PIL05], ou seja, o elemento básico representante de um grupo de coisas com propriedades em comum. Assim, neste trabalho, quando se fala em classificadores, quer-se dizer interfaces e classes.

pacote *hotspot* contém um segundo nível de classes abstratas, quando não é desenvolvida a implementação, permitindo a extensão do *framework* para a construção de aplicações específicas.

Adicionalmente aos pacotes citados, mais três estão criados na implementação. O pacote *starter* (iniciador) contém a classe de definições e a classe que inicia a execução do *framework*. O pacote *exceptions* (exceções) contém apenas uma classe que trata as possíveis exceções durante o início de execução do *framework*. Finalmente, o pacote *general* (geral) é dividido em outros pacotes e contém classes para carregar arquivos de configuração e manipular ontologias. Todos os pacotes mencionados até aqui são explicados detalhadamente no decorrer desta seção, concluindo-se com a ilustração de um fluxo completo de funcionamento do *framework*.

Antes de se explicar cada pacote individualmente, convém explicitar três questões importantes. A primeira se relaciona a quais partes do Modelo Conceitual não foram implementadas a ponto de funcionarem no protótipo. Os classificadores relativos às restrições de adaptação, dentro do pacote *point*, tiveram suas estruturas mínimas construídas, sendo elas: *IAdaptionRestriction*, *AbstractAdaptionRestriction* e *AdaptionRestriction*. Semelhantemente, o pacote *assessment* não é funcional dentro do protótipo. Mas, independentemente da falta de funcionalidade, toda a implementação é explicada neste capítulo.

A segunda questão é relativa à existência de dois arquivos de configuração do *framework*. Um deles é o arquivo *adaptioncomponent.properties*, contendo as localizações das classes implementadas e as propriedades para a execução do sistema. O outro arquivo é o *adaptionconfig.xml*, contendo as definições dos *hotspots* implementados. Ambos são mais bem explicados na seção 5.3.

A terceira e última questão diz respeito à apresentação dos diagramas de classes de implementação dos pacotes detalhados. Alguns atributos e métodos estão suprimidos por não demonstrarem relevância significativa na elucidação do *framework*. Além disso, como em alguns casos as figuras dos diagramas aparecem menores do que o desejável, todas elas foram adicionadas ao Apêndice A. Sem distinção, todos os diagramas de implementação seguem uma notação UML.

Principiando o detalhamento, a Figura 5.1 apresenta o pacote *core*. Vê-se a interface *IAdaptionCore* e mais duas classes. A classe *AbstractAdaptionCore*, como é uma *thread*[11], realiza a interface *java.lang.Runnable*. Ela referencia a classe de definições do *framework* por meio do atributo *adaptionDefinitions*, e também as classes que lêem as informações dos

[11] *Thread* é parte de um programa que pode executar concorrentemente com outra parte [DEI04].

arquivos de configuração, descritos em XML, por meio dos atributos *loaderDOM4J* e *loaderSAX*. Adicionalmente, ela se relaciona com as classes *AbstractAdaptionPercept* (atributo *percept*), *AbstractKnowledge* (atributo *context*), *AbstractAdaptionAssessment* (atributo *assessment*), e, por meio de um mapa, com a classe *AbstractAdaptionPoint* (atributo *adaptionPoints*). O método *changeObservedElement()* é utilizado para sinalizar o *framework* quando um elemento observado sofre uma alteração, esta decorrente de alguma informação importante que deve ser considerada para uma possível adaptação. Esse método necessita do nome do elemento de percepção que monitora o elemento observado, de um objeto contendo a informação relevante, do tipo de objeto (se é uma ontologia ou apenas uma string simples), e, se a informação for uma ontologia, do *namespace* da mesma.

A classe *AdaptionCore* possui o método *start()* que inicia a execução da *thread*, possibilitada pelo método *run()*. O método *internalSetup()* lê as definições do *framework* por meio da classe *AdaptionDefinitions* e carrega as informações do arquivo de configuração *adaptionconfig.xml*. O método *verifyPerceptElementsUpdated()* monitora os elementos de percepção que têm seus elementos observados modificados, criando ou atualizando o contexto de execução corrente com a informação relevante para uma possível adaptação. O método *verifyAdaptionPointsChanged()* monitora quais pontos de adaptação estão em execução, sinalizando ao contexto que deve ser escolhida uma política de adaptação para ser aplicada sobre aqueles pontos, permitindo concluir o processo de adaptação.

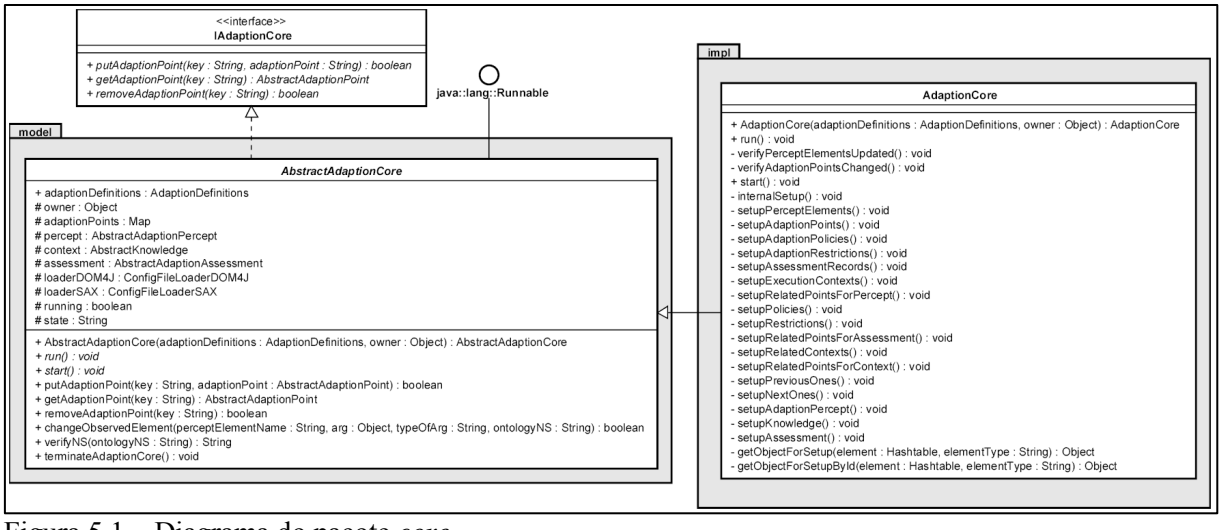

Figura 5.1 – Diagrama do pacote *core*.

O pacote *percept*, mostrado na Figura 5.2, contém três interfaces: *IAdaptionPercept*, *IPerceptElement*, e *IObservedElement*.

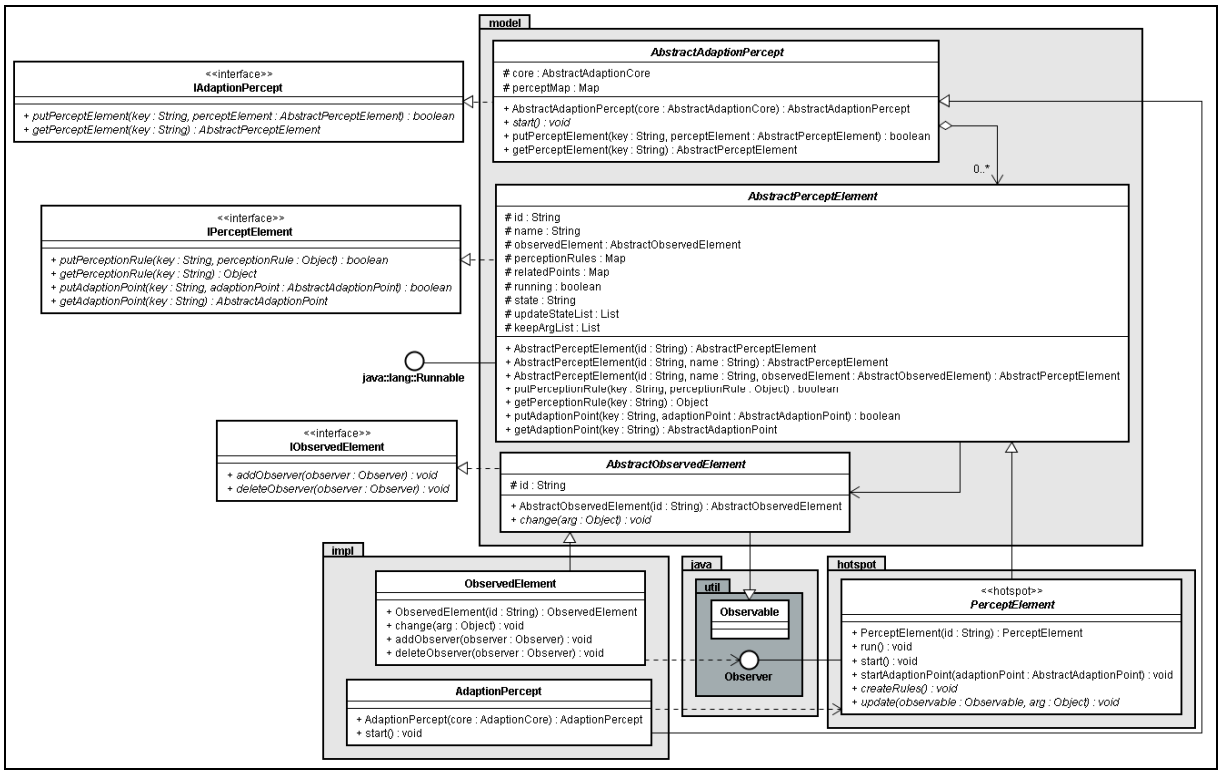

Figura 5.2 – Diagrama do pacote *percept*.

A classe *AbstractAdaptionPercept* possui uma referência à classe *AbstractAdaptionCore* (atributo *core*), e um mapa de classes *AbstractPerceptElement* (atributo *perceptMap*). Ela trabalha, basicamente, como uma coleção de elementos de percepção. A classe *AdaptionPercept* possui apenas a implementação do método *start()*, encarregado de iniciar a execução de cada elemento de percepção. A classe *AbstractPerceptElement*, como é uma *thread*, realiza a interface *java.lang.Runnable*, sendo composta, principalmente, por um mapa de regras de percepção (atributo *perceptionRules*), um mapa de pontos de adaptação (atributo *relatedPoints*), e uma referência a um elemento observado (atributo *observedElement*). Ela ainda mantém a lista de estados de mudança (atributo *updateStateList*) para os momentos nos quais o elemento de percepção é sinalizado, e a lista das informações passadas ao elemento de percepção (atributo *keepArgList*) em cada sinalização.

A classe *PerceptElement* realiza a interface *Observer*, sendo esta explicada logo adiante. Ela é definida como um *hotspot* e, quando especializada, obriga a implementação do método *createRules()*, para a criação de regras que podem iniciar um processo de adaptação, e do método *update()*, para receber a sinalização de mudança de estado do elemento observado e iniciar a execução do ponto de adaptação relacionado. Ela possui ainda o método *start()*, com a função de iniciar a execução do elemento de percepção.

Neste momento, antes de se explicar as últimas classes do pacote *percept*, convém esclarecer o termo *Observer*, citado anteriormente. Foi necessário utilizar um recurso da linguagem *Java* semelhante ao padrão de projeto *Observer* [GAM95], com o intuito de permitir a sinalização ao elemento de percepção por meio de um objeto observável. Assim, quando alguma informação relevante em um agente precisa ser passada a um elemento de percepção, este mesmo é notificado por uma mudança no estado do elemento observado. Por isso, a classe *PerceptElement* realiza a interface *Observer*, e o que é definido como *Subject* no padrão de projeto (como uma interface), é definido em *Java* como uma classe *Observable*.

A classe *AbstractObservedElement* é um *Observable*. A classe *ObservedElement* implementa o método *change()* utilizado para modificar o estado do elemento observado e para notificar os elementos de percepção, além de implementar os métodos para adicionar e remover os elementos de percepção como observadores.

O pacote *point*, presente na Figura 5.3, contém o maior número de interfaces e classes do *framework*. Suas interfaces são: *IAdaptionPolicy*, *IPolicyStep*, *IAdaptionRestriction*, *IAdaptionPoint* e *IAdaptionDescriptor*. A segunda interface não possui assinaturas, mas foi mantida para possíveis complementações no futuro.

A classe *AbstractAdaptionPolicy* possui um mapa de critérios (atributo *policyCriteria*) e um mapa de passos da política (atributo *policySteps*). A classe *AdaptionPolicy* implementa o método *fromFileToRules()*, com a função de carregar regras de inferência – quando estão salvas em arquivo – para a escolha da política em função das propriedades do contexto corrente de execução. Ela é definida como um *hotspot* e, quando especializada, obriga a implementação do método *createCriteria()*, para a criação dos critérios de escolha da política, e do método *createSteps()*, para a criação dos passos da política para a adaptação. Os tipos de critérios para a escolha da política são apresentados durante a explicação do pacote *knowledge*, mais adiante. A classe *AbstractPolicyStep* apenas implementa as assinaturas definidas na interface. Com simplicidade semelhante, a classe *PolicyStep* possui apenas o seu método construtor. Ela é definida como um *hotspot*, mas, quando especializada, não obriga a implementação de método algum.

A classe *AbstractAdaptionRestriction* apenas implementa as assinaturas definidas na interface. A classe *AdaptionRestriction* contém apenas o seu método construtor. Como o conceito de restrição foi implementado sem funcionalidade no protótipo, os classificadores que o representam possuem estruturas extremamente simplificadas.

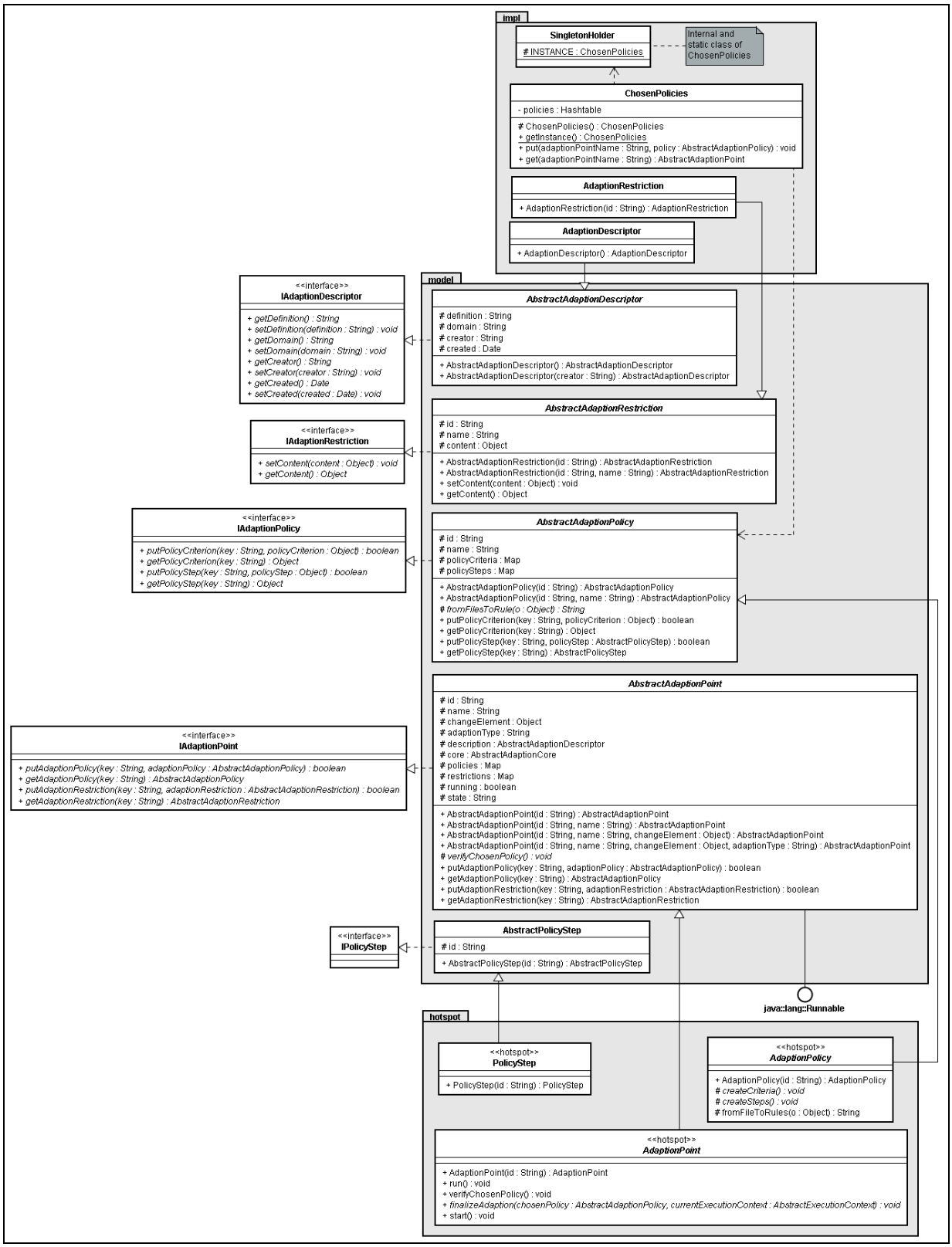

Figura 5.3 – Diagrama do pacote *point*.

A classe *AbstractAdaptionPoint*, como é uma *thread*, realiza a interface *java.lang.Runnable*. Ela referencia o elemento de mudança (atributo *changeElement*), o descritor de adaptação (atributo *description*), e o núcleo de adaptação (atributo *core*). Ela

ainda classifica o tipo de adaptação por meio do atributo *adaptionType*, e contém dois mapas, um de políticas de adaptação (atributo *policies*) e outro de restrições de adaptação (atributo *restrictions*). A classe *AdaptionPoint* implementa o método *verifyChosenPolicy()*, encarregado de repassar ao método *finalizeAdaption()* a política de adaptação escolhida e o contexto corrente de execução. O método *verifyChosenPolicy()* utiliza a classe *ChosenPolicies*, explicada adiante, para recuperar a política escolhida relacionada ao ponto de adaptação. A classe *AdaptionPoint* é definida como um *hotspot* e, quando especializada, exige a implementação do método *finalizeAdaption()*, há pouco citado, responsável por efetuar a adaptação sobre o elemento de mudança. Adicionalmente, ela possui o método *start()*, utilizado para iniciar a execução do ponto de adaptação.

Para a compreensão da classe *ChosenPolicies*, é preciso entender o uso do padrão de projeto *Singleton* [GAM95]. O uso desse padrão permite a uma classe ter apenas uma instância dentro do sistema, fornecendo um ponto global de acesso a ela. Assim, a classe *ChosenPolicies* possui um método construtor protegido e um método estático (*getInstance()*) que retorna a única instância dela. Este último método utiliza uma classe interna, definida aqui como *SingletonHolder*, composta por apenas uma constante que recebe essa instância. A classe *ChosenPolicies* ainda possui o mapa de políticas escolhidas (atributo *policies*).

Concluindo-se a explicação do pacote *point*, tem-se ainda a classe *AbstractAdaptionDescriptor*, que apenas implementa as assinaturas dos métodos da interface. A classe *AdaptionDescriptor* possui somente o seu método construtor.

O pacote *knowledge*, visto na Figura 5.4, é composto pelas interfaces *IKnowledge* e *IExecutionContext*, e por mais cinco classes. A segunda interface possui as assinaturas para inserir e recuperar as propriedades de contexto, informações indispensáveis para o *framework* efetuar as adaptações.

A classe *AbstractKnowledge* possui, principalmente, os seguintes atributos: *core*, *contextMap*, *currentExecutionContext* e *ontology*. O primeiro atributo é uma referência à classe *AbstractAdaptionCore*. O segundo atributo é um mapa de contextos de execução. O terceiro atributo representa o contexto de execução corrente. Por fim, o atributo *ontology* guarda a representação ontológica de todos os contextos de execução correntes agrupados. Quando se unifica as informações dos diversos contextos em uma única ontologia, tenta-se facilitar a manipulação e distribuição do conhecimento armazenado durante a execução de um agente.

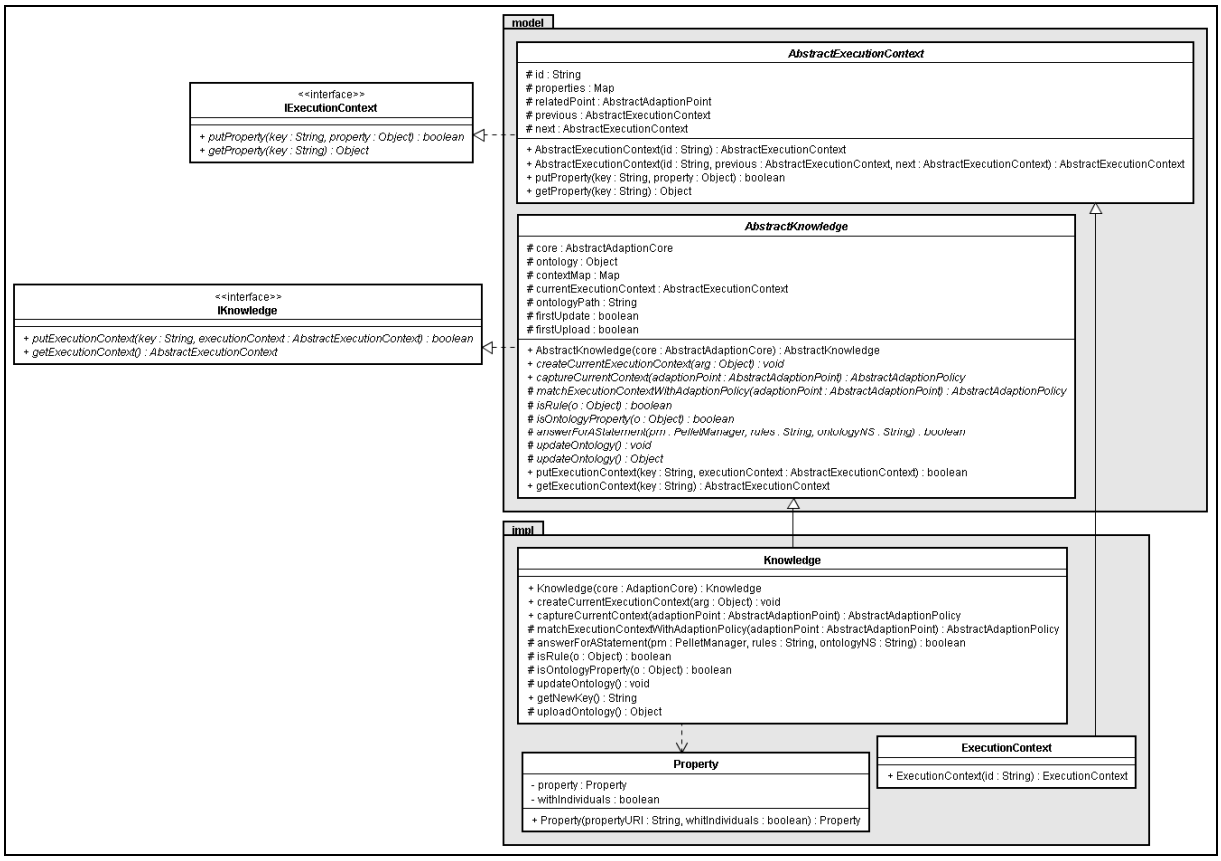

Figura 5.4 – Diagrama do pacote *knowledge*.

Para a compreensão dessa estrutura ontológica, observa-se a Figura 5.5.

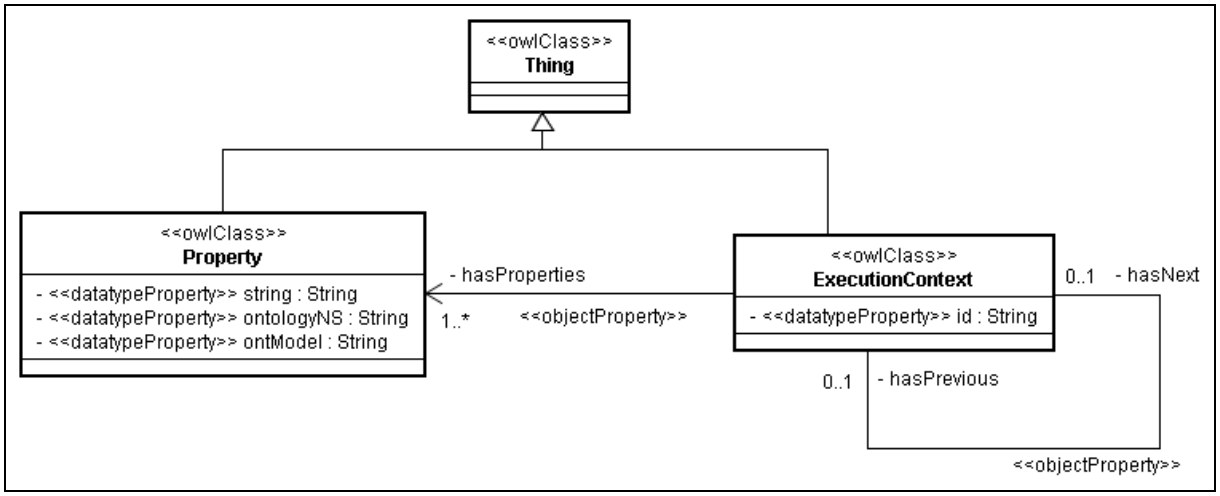

Figura 5.5 – Estrutura ontológica dos contextos de execução encadeados como conhecimento.

Na Figura 5.5, verifica-se que toda a classe OWL construída em uma ontologia é uma especialização da classe *Thing*. A classe OWL *Property* representa uma propriedade (do agente ou do ambiente) do contexto de execução, como explicada no Modelo Conceitual, e contém três propriedades explicadas mais adiante, durante o detalhamento da classe *Knowledge* (não confundir uma "propriedade" de uma classe OWL – semelhante a um

atributo de uma classe UML –, com uma "propriedade" do contexto de execução). A classe OWL *ExecutionContext* representa o contexto de execução, contendo uma ou mais propriedades de contexto (propriedade *hasProperties*), um identificador (propriedade *id*), zero ou um contexto de execução anterior (propriedade *hasPrevious*), e zero ou um contexto de execução posterior (propriedade *hasNext*). As propriedades *hasPrevious* e *hasNext* permitem encadear os contextos de execução.

A classe *Knowledge* implementa os diversos métodos abstratos da superclasse. O primeiro deles é o método *createCurrentExecutionContext()*, encarregado de criar um novo contexto de execução corrente, ou atualizar o contexto corrente existente, inserindo uma propriedade e encadeando corretamente esse contexto com os demais. O método *captureCurrentContext()* tem três funções: relaciona o contexto de execução corrente ao ponto de adaptação adequado, retorna a política escolhida pelo método *matchExecutionContextWithAdaptionPolicy()*, e adiciona esse contexto ao mapa de contextos. O método *matchExecutionContextWithAdaptionPolicy()* compara cada critério de escolha das políticas definidas (os valores daquele atributo *policyCriteria* citado durante a explicação do pacote *point*, especificamente na classe *AbstractAdaptionPolicy*) com as propriedades do contexto de execução corrente, permitindo a escolha de uma dessas políticas para a execução do processo de adaptação.

Atualmente, existem três tipos de critérios para a escolha de uma política de adaptação, como mostra a Tabela 5.1. Analisando a tabela, é possível entender as três propriedades definidas na classe OWL *Property*, vista na Figura 5.5. Quando uma propriedade do contexto é representada por uma string simples, a propriedade *string* permite guardar essa propriedade. Agora, quando uma propriedade do contexto é representada por uma ontologia, utiliza-se a propriedade *ontologyNS* para manter o *namespace* da ontologia, e a propriedade *ontModel* para armazenar todo o código OWL da ontologia em questão.

Voltando-se à explicação da classe *Knowledge*, têm-se ainda três métodos utilizados durante a comparação dos critérios de escolha da política de adaptação com as propriedades do contexto de execução corrente. O primeiro é o método *answerForAStatement()*, usado para a inferência da propriedade, por meio do critério que é uma regra. O segundo é o método *isRule()*, usado para averiguar se o critério de escolha é realmente uma regra de inferência. O último é o método *isOntologyProperty()*, usado para averiguar se o critério é realmente uma propriedade de ontologia. Além desses três métodos, ainda é importante explicar mais dois. O método *updateOntology()* é utilizado para atualizar a ontologia que representa todos os

contextos de execução encadeados. Quanto ao método *uploadOntology*(), é utilizado para carregar em memória essa ontologia construída.

TABELA 5.1 – TIPOS DE CRITÉRIOS PARA A ESCOLHA DE UMA POLÍTICA DE ADAPTAÇÃO

Tipos de critérios	Uso	Tipo de propriedade	Comparação
String simples	É utilizado para ser comparado a uma propriedade que também é uma string simples.	Representada por uma string simples.	Se as duas strings forem iguais, o critério é satisfeito.
Regra de inferência	É utilizado para se inferir uma resposta verdadeira com base na propriedade.	Representada por uma ontologia.	Se a inferência resulta em verdade, o critério é satisfeito.
Propriedade de ontologia	É utilizado para verificar se existem ou não indivíduos (instâncias) em uma propriedade. Necessita-se criá-lo como uma instância da classe *Property*[12] do *framework*.	Representada por uma ontologia.	Se for pretendido encontrar indivíduos, e isso ocorrer, o critério é satisfeito. Se não for pretendido encontrar indivíduos, e isso ocorrer, o critério é satisfeito.

A classe *AbstractExecutionContext* possui, principalmente, os seguintes atributos: *properties* (representa o mapa de propriedades do contexto de execução), *relatedPoint* (referencia o ponto de adaptação relacionado ao contexto de execução corrente), *previous* (referencia o contexto de execução anterior ao corrente), e *next* (referencia o contexto de execução posterior ao corrente). É importante relembrar que existem dois tipos de propriedades do contexto de execução: a propriedade que é uma string simples, e a definida por meio de uma estrutura ontológica. A classe *ExecutionContext* possui, atualmente, apenas o seu método construtor.

Para finalizar a explicação do pacote *knowledge*, ainda é preciso apresentar a classe *Property*. Ela é necessária, como observado na Tabela 5.1, para permitir a comparação entre um critério de escolha de uma política de adaptação do tipo "propriedade de ontologia" com uma propriedade do contexto de execução representada por uma ontologia. Assim, ela possui o atributo *property*, representando uma propriedade de uma ontologia, e o atributo *withIndividuals*, um *boolean* que diferencia a busca ou não de indivíduos dentro dessa ontologia. A classe *Property* precisa ser instanciada durante a criação dos critérios de escolha de uma política, no método implementado *createCriteria()* de uma classe que especializa o *hotspot AdaptionPolicy*, do pacote *point*.

[12] Não confundir com a classe OWL *Property* da ontologia que representa o encadeamento dos contextos de execução.

O pacote *assessment*, presente na Figura 5.6, é composto pelas interfaces *IAdaptionAssessment* e *IAssessmentRecord*, e por mais quatro classes. Ressalta-se que este pacote é implementado sem execução funcional dentro do protótipo.

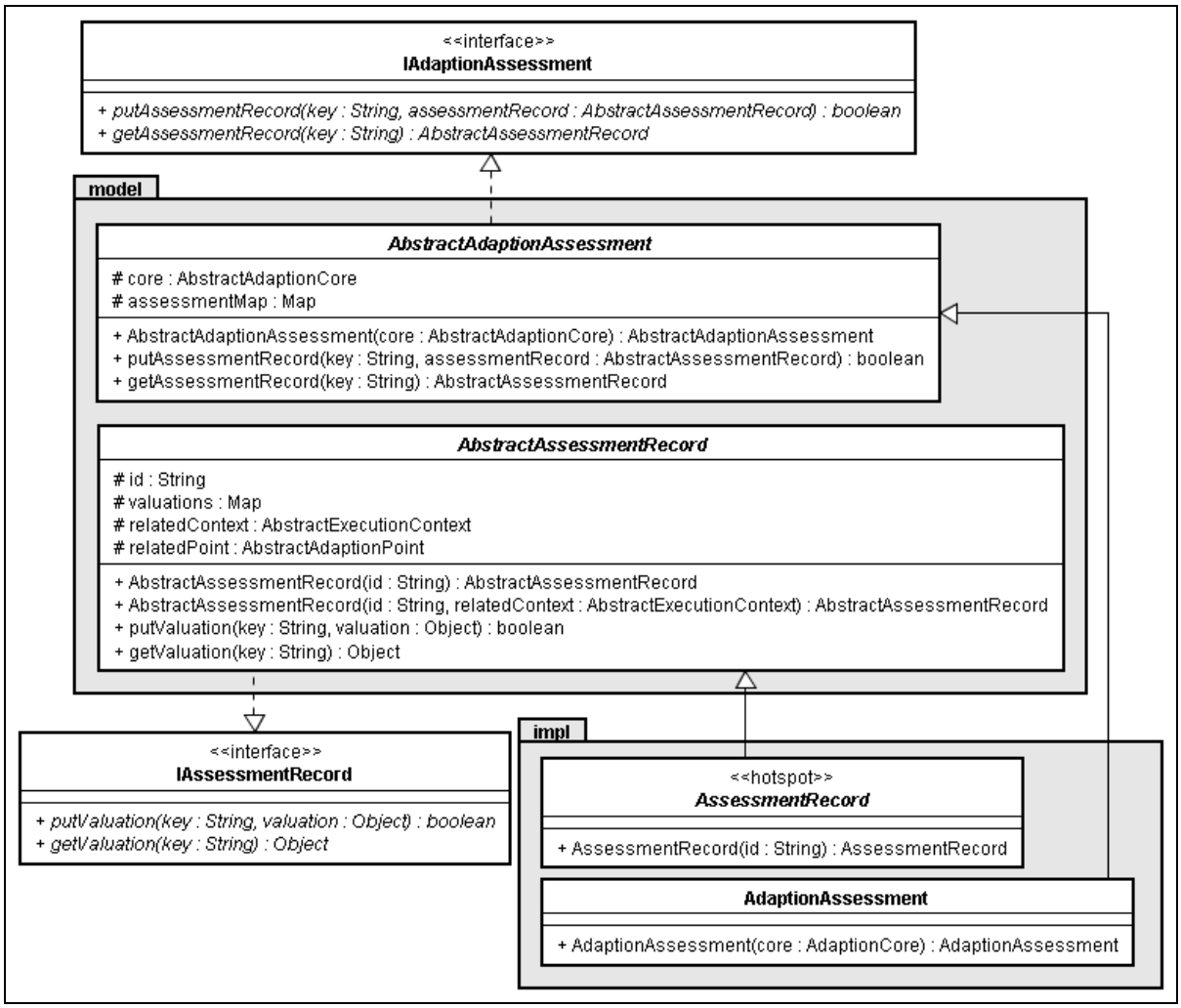

Figura 5.6 – Diagrama do pacote *assessment*.

A classe *AbstractAdaptionAssessment* contém dois atributos: *core* (referência ao núcleo de adaptação) e *assessmentMap* (mapa de registros de avaliação). A classe *AdaptionAssessment* possui apenas o seu método construtor. Quanto à classe *AbstractAssessmentRecord*, ela contém, principalmente, os seguintes atributos: *valuations* (mapa de avaliações), *relatedContext* (referência ao contexto de execução relacionado), e *relatedPoint* (referência ao ponto de adaptação relacionado). Por fim, tem-se a classe *AssessmentRecord*, possuindo apenas o seu método construtor. Ela é definida como um *hotspot*, mas como não é funcional dentro do protótipo, ainda não pode ser estendida.

Detalhados os pacotes diretamente mapeados do Modelo Conceitual, é preciso explicar os pacotes restantes do protótipo. São eles: *starter*, *exception* e *general*. O pacote

starter, presente na Figura 5.7, é necessário para iniciar a execução do *framework*, constituído pelas classes *AdaptionStarter* e *AdaptionDefinitions*. A primeira referencia a segunda e também a classe *AdaptionCore*, possuindo os seguintes métodos: *main()*, para iniciar o *framework* a partir de uma linha de comando; o método privado *start()*, chamado pelo *main()*, com a função de executar o núcleo de adaptação; e o método público *start()*, chamado a partir de um agente, com a mesma finalidade do método privado *start()*. A classe *AdaptionDefinitions* tem as funções de carregar as informações do arquivo de propriedades *adaptioncomponent.properties* mantendo as instâncias em tabelas *hash*, e de gerar automaticamente os valores dos identificadores (atributos *ids*) dessas instâncias.

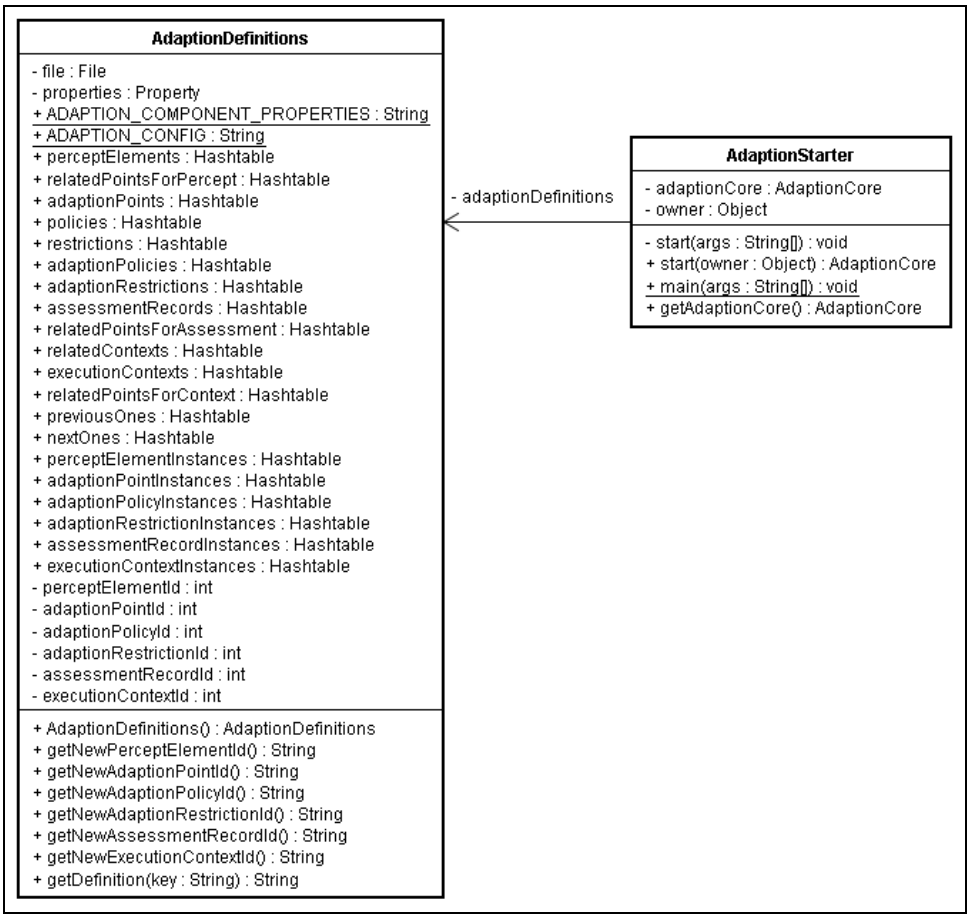

Figura 5.7 – Diagrama do pacote *starter*.

Atualmente, o pacote *exceptions* possui apenas a classe *StartException* (estende a classe *java.lang.Throwable*), responsável por gerar uma exceção se uma execução do *framework* inicia a partir de um agente sem a passagem do objeto que o representa (definido como *owner*).

Por fim, o pacote *general* é composto por quatro classes, vistas na Figura 5.8: *ConfigFileLoaderDOM4J*, *ConfigFileLoaderSAX*, *Ontology* e *PelletManager*. A primeira tem

a função de carregar as informações contidas no arquivo de configuração *adaptionconfig.xml*, utilizando o *parser* DOM4J[13], capaz de analisar arquivos escritos na linguagem XML. A segunda classe tem a mesma função da primeira, utilizando o *parser* SAX[14], mas, para o protótipo, sua implementação não está finalizada. A classe *Ontology* é utilizada quando o método *changeObservedElement()* da classe *AbstractAdaptionCore* (pacote *core*, explicado anteriormente) recebe uma informação representada por um objeto descrito por meio de uma ontologia. Ela possui o atributo *ontModel*, para armazenar toda a estrutura ontológica, e o atributo *ontologyNS*, utilizado para guardar o *namespace* dessa ontologia.

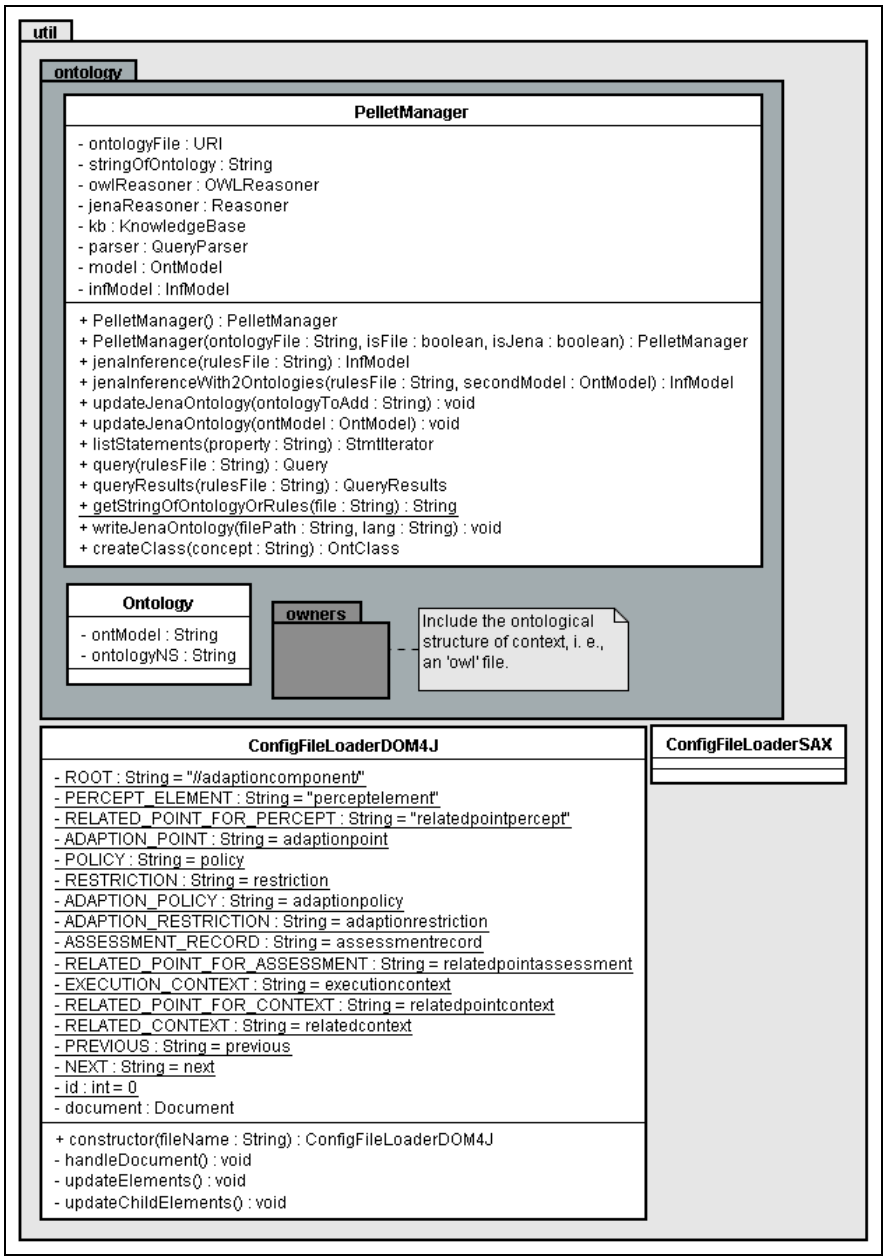

Figura 5.8 – Diagrama do pacote *general*.

[13] Para mais informações sobre o *parser* DOM4J, consultar a página http://www.dom4j.org/
[14] Para mais informações sobre o *parser* SAX, consultar a página http://www.saxproject.org/

Quanto à classe *PelletManager*, ela tem a função de facilitar o uso do mecanismo *Pellet*. Seus principais métodos são:

- Construtor: permite carregar uma ontologia a partir de uma string ou de um arquivo, além de aceitar tanto o modelo de ontologia processável pelo *Jena* quanto pela OWL API.
- *jenaInference()*: permite realizar uma inferência com base em regras processáveis pelo *Jena*.
- *jenaInferenceWith2Ontologies()*: permite realizar uma inferência sobre duas ontologias com base em regras processáveis pelo *Jena*.
- *updateJenaOntology()*: permite atualizar uma ontologia processável pelo *Jena*. Um dos métodos permite receber o modelo em forma de string, o outro por meio de um *OntModel*[JEN07].
- *listStatements()*: permite recuperar as expressões resultantes de uma inferência processável pelo *Jena*, para uma propriedade em particular.
- *getStringOfOntologyOrRules()*: permite transformar em string um arquivo de ontologia ou de regras.
- *writeJenaOntology()*: permite salvar em arquivo uma estrutura ontológica processável pelo *Jena*.

É importante esclarecer que dentro do pacote *general* existe um subpacote chamado *util.ontology.owners*, contendo o arquivo *context.owl*, que descreve a estrutura ontológica de todos os contextos de execução correntes agrupados (explicado no detalhamento do pacote *knowledge* e na Figura 5.5). A codificação dessa ontologia, na linguagem OWL, pode ser vista no Apêndice B.

Concluindo e complementando a elucidação da implementação do protótipo, busca-se, de maneira sintetizada e com o auxílio da Figura 5.9, caracterizar o funcionamento do *framework* a partir de uma visão geral do que se apresenta desenvolvido. Notam-se, observando essa figura, a supressão dos atributos e métodos das classes, além de alguns relacionamentos, permitindo uma maior clareza para o entendimento do diagrama.

O *framework* é iniciado pela classe *AdaptionStarter*, instanciando a classe *AdaptionCore* e a classe *AdaptionDefinitions*. A classe *AdaptionCore* instancia as classes definidas no arquivo *adaptionconfig.xml*, juntamente com a classe *ConfigFileLoaderDOM4J*. A seguir, a classe *AdaptionCore* repassa o fluxo de execução para a classe *AdaptionPercept*, que inicia a execução dos elementos de percepção (as instâncias que estendem a classe *PerceptElement*).

Figura 5.9 – Diagrama geral de implementação do *framework* de adaptação.

Quando alguma informação, definida como importante para uma possível adaptação, é repassada à classe *AdaptionCore*, a classe *ObservedElement* relacionada é sinalizada, notificando a classe *PerceptElement* que a observa. Essa notificação faz a classe

PerceptElement iniciar a execução do ponto de adaptação (*AdaptionPoint*) relacionado a uma regra de percepção casada – se nenhuma regra é casada, nenhum ponto de adaptação é iniciado. A classe *AdaptionCore* identifica uma mudança em uma classe *PerceptElement*, criando um contexto de execução corrente (*ExecutionContext*).

Enquanto o contexto de execução corrente é criado por meio da classe *Knowledge*, o ponto de adaptação é iniciado por meio da classe *PerceptElement*. Durante a execução do ponto de adaptação, as políticas escolhidas são sempre verificadas para uma possível adaptação. Novamente, a classe *AdaptionCore* entra em ação, identificando o início de um ponto de adaptação e repassando o fluxo de execução para a classe *Knowledge*, para a escolha de uma política de adaptação pertencente ao ponto de adaptação em questão. Essa escolha é realizada conforme as propriedades de contexto recuperadas durante a execução atual. Escolhida a política, ela é salva na classe *ChosenPolicies*.

Com a política armazenada nessa classe, a classe *AdaptionPoint* a captura e instancia os seus passos, que permitem adaptar o elemento de mudança. Assim, fecha-se um ciclo de adaptação dentro do *framework* proposto.

Compreendido o funcionamento do protótipo, segue-se com a definição de um cenário para a aplicação do *framework*.

5.2 O Cenário de Aplicação

Com a finalidade de exemplificar algumas das funcionalidades disponibilizadas pelo *framework* proposto, esta seção apresenta a descrição de um cenário a ser aplicado para a execução do protótipo desenvolvido. Mas, antes da descrição desse cenário, é necessário justificar a sua escolha. Assim, introduzem-se os conceitos e teoria, para depois se detalhar o funcionamento do exemplo.

A popularização de sistemas embarcados de navegação – GPS[15], por exemplo – e a contínua melhoria de periféricos portáteis – como o PDA[16] ou o telefone celular – permitirão o surgimento de novas aplicações, que poderão obter benefícios dos recursos da Web Semântica quando conectados à Internet e interconectados de uma maneira *ad hoc*. Analisando essas possibilidades, surgem projetos interessados em reproduzir esse tipo de cenário. Um deles é o projeto *Urban Vehicular Grid* [MES07].

[15] GPS é a abreviatura para Sistema de Posicionamento Global, ou seja, a navegação por meio de uma rede universal de satélites.
[16] PDA é a abreviatura para Assistente Digital Pessoal, ou seja, um pequeno computador portátil que oferece ferramentas para a organização de informações pessoais.

Nesse projeto, busca-se utilizar redes *ad hoc* – pequenas redes nas quais alguns dispositivos são partes delas apenas durante a sessão de comunicação, ou enquanto permanecem próximos o suficiente para trocarem informações – como uma extensão da infra-estrutura fixa de telecomunicação, focando-se no ambiente de transporte urbano. Cada automóvel seria equipado com alguns processos básicos de informação e com capacidades de comunicação por rede, não sendo apenas um nodo final de rede, mas tornando-se parte da infra-estrutura. O acoplamento de redes *ad hoc* nos automóveis, com a infra-estrutura já existente de redes *mesh* – redes nas quais cada nodo está conectado a um ou mais nodos, possibilitando a transmissão de mensagens entre eles por diferentes caminhos –, resultaria em redução de custos, aumento de desempenho e maior resiliência a falhas.

Esse novo paradigma de rede oportunista, na qual a rede *ad hoc* encontra a infra-estrutura de rede *mesh*, modificará a maneira como as redes irão operar e também as otimizarão, enquanto for focada, principalmente, a alocação de recursos. A rede oportunista alcançará os recursos para as necessidades da conectividade de comunicação, conduzindo a um *grid* veicular urbano ubíquo, extremamente flexível e robusto, tornando-se o "sistema nervoso" das comunicações urbanas.

As comunicações sem fio se tornarão, finalmente, uma parte dominante dos subsistemas eletrônicos veiculares. Elas fornecem duas funções importantes que as tornam uma tecnologia de escolha para consumidores e para aplicações de transporte: a mobilidade e a infra-estrutura barata. O suporte à mobilidade é fator chave em qualquer sistema de comunicação veicular. Esse tipo de sistema deve suportar conexões a qualquer momento e em qualquer lugar, para realizar a visão de futuro almejada ao sistema de transporte. Ademais, a comunicação veicular sem fio pode permitir a navegação, informação do motorista, comunicação eletrônica, entretenimento, e melhoria na segurança. Ao mesmo tempo, as comunicações veiculares podem ter preços acessíveis.

Retornando ao projeto, ele apresenta algumas aplicações interessantes dentro dessa visão de futuro. Mas, como exemplo para este trabalho, é dado foco à aplicação de redes automóvel para automóvel, na qual as informações são propagadas entre os automóveis. A Figura 5.10 ilustra, resumidamente, um possível cenário nesse contexto. Imagina-se uma situação em que uma ambulância precisa se deslocar rapidamente para prestar socorro. Um agente de software, em execução na ambulância, é responsável por definir a melhor rota até o local do socorro e comunicar os carros a frente que devem seguir na pista da direita, desobstruindo a passagem da ambulância.

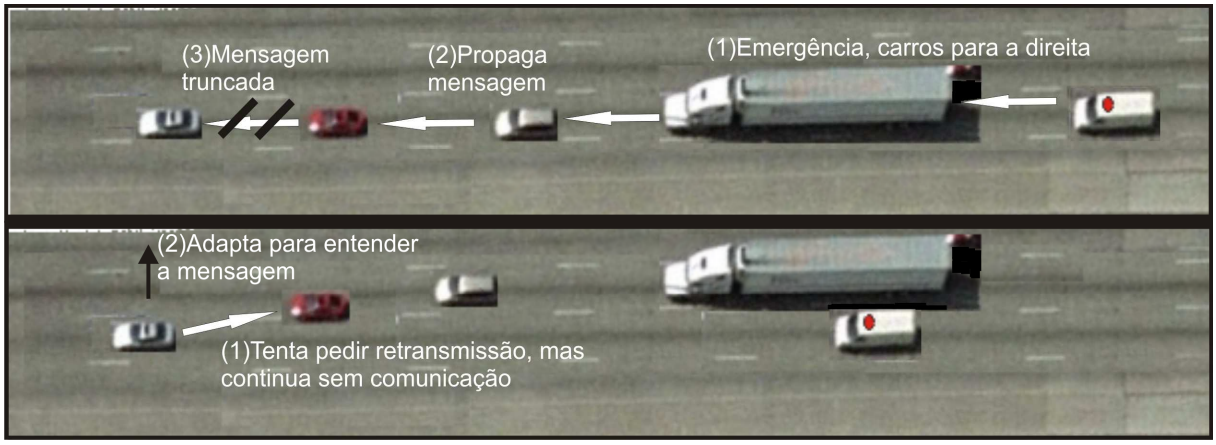

Figura 5.10 – Ilustração do cenário de aplicação.

Pelo conceito de redes *ad hoc*, os agentes dos carros não suficientemente próximos da ambulância precisam receber a informação propagada pelos agentes dos carros em localização intermediária. Por alguma falha de comunicação, o agente de um carro mais distante recebe uma mensagem de emergência truncada, do carro mais próximo da ambulância. O agente até poderia tentar uma nova comunicação, mas a falha permanece. Portanto, outra ação deve ser executada.

Enquanto o motorista do carro mais distante não percebe a aproximação da ambulância, o agente do seu carro identifica que a mensagem truncada é de emergência, precisando ainda descobrir o significado do conteúdo da mesma para tomar alguma decisão. Um sensor de hardware no carro detecta que os carros logo atrás estão se movendo para a pista da direita (este tipo de detecção não envolve a troca de mensagens entre os carros). Com o pedaço preservado da mensagem que a define como de emergência, e com a informação da mudança de pista dos carros logo atrás, o agente consegue modificar suas próprias regras e inferir que uma ambulância pede passagem.

Infelizmente, essa adaptação não é finalizada a tempo do agente avisar o motorista sobre a passagem da ambulância. Apesar disso, a mudança das regras para a inferência se mantém armazenada. Assim, quando em outra ocasião o mesmo erro de falha na comunicação acontece, o agente é capaz de enviar um aviso para o painel do carro informando ao motorista que tente permanecer na pista da direita, pois é provável que uma ambulância esteja se aproximando. Desconsidera-se que os sensores de hardware consigam detectar exatamente em qual pista o carro está. Desta maneira, quando se fala em permanecer na pista da direita, pode-se também significar que o motorista deva mudar de pista.

Por fim, naquela outra ocasião, o motorista guia o carro para a pista da direita, permitindo a passagem da ambulância. Ele até poderia informar ao agente, por meio do painel, que o mesmo acertou na tomada de decisão, após verificar que realmente uma

ambulância iria passar. Imaginando-se todo esse sistema conectado à Internet, e utilizando ontologias, o controle de tráfego atingiria os resultados, afirmados anteriormente, das redes *ad hoc*. A conexão poderia abranger, além dos automóveis, outros elementos como semáforos, câmeras, postes de iluminação, e assim por diante. Problemas como acidentes e congestionamentos poderiam ser evitados.

Dado o cenário, esse exemplo mostra a adaptação estrutural do agente por meio de uma informação nova (movimentação dos carros), permitindo a modificação das regras do agente. Essa informação precisou ser identificada e comparada com outras informações do contexto corrente de execução, presentes nas ontologias processadas pelo agente. Isso é diferente do agente "sentir" uma situação diretamente inteligível e tomar uma decisão.

Nota-se, como exposto no parágrafo anterior, a afirmação de um cenário que apresenta uma adaptação estrutural do agente, não sendo citada a adaptação comportamental. Isso ocorre porque o carro que permite a passagem, após adaptar, não altera o seu conjunto de reações observáveis. Assim, o cenário não contempla integralmente o conceito de adaptação definido neste trabalho.

Explicado o cenário de aplicação para a instanciação do *framework* proposto, a próxima seção apresenta o detalhamento de implementação da simulação, construída sobre esse cenário.

5.3 A Simulação do Cenário

Iniciado o desenvolvimento da simulação do cenário descrito, foi necessária a realização de três atividades principais: (i) a criação de agentes, utilizando a plataforma *SemantiCore*, de acordo com o cenário de aplicação; (ii) a instanciação do *framework* proposto, para o seu funcionamento sobre esses agentes, e; (iii) a execução da plataforma *SemantiCore*.

Na implementação da simulação, bem como do *framework*, além da plataforma *SemantiCore*, utilizaram-se as seguintes tecnologias: a linguagem de programação *Java*; o ambiente de desenvolvimento integrado *Eclipse*[17], para a implementação com a linguagem *Java*; a linguagem OWL para a descrição de ontologias; o mecanismo *Pellet*, incluindo sua integração à API *Jena*, para a inferência em ontologias; e a plataforma *Protégé*[18], para a criação e manipulação de ontologias.

[17] Informações sobre o IDE *Eclipse* podem ser encontradas na página *http://www.eclipse.org*.
[18] Informações sobre a plataforma *Protégé* podem ser encontradas na página *http://protege.stanford.edu/*.

Antes do detalhamento de cada atividade principal da simulação, é conveniente observar a estrutura final alcançada com a implementação, presente na Figura 5.11.

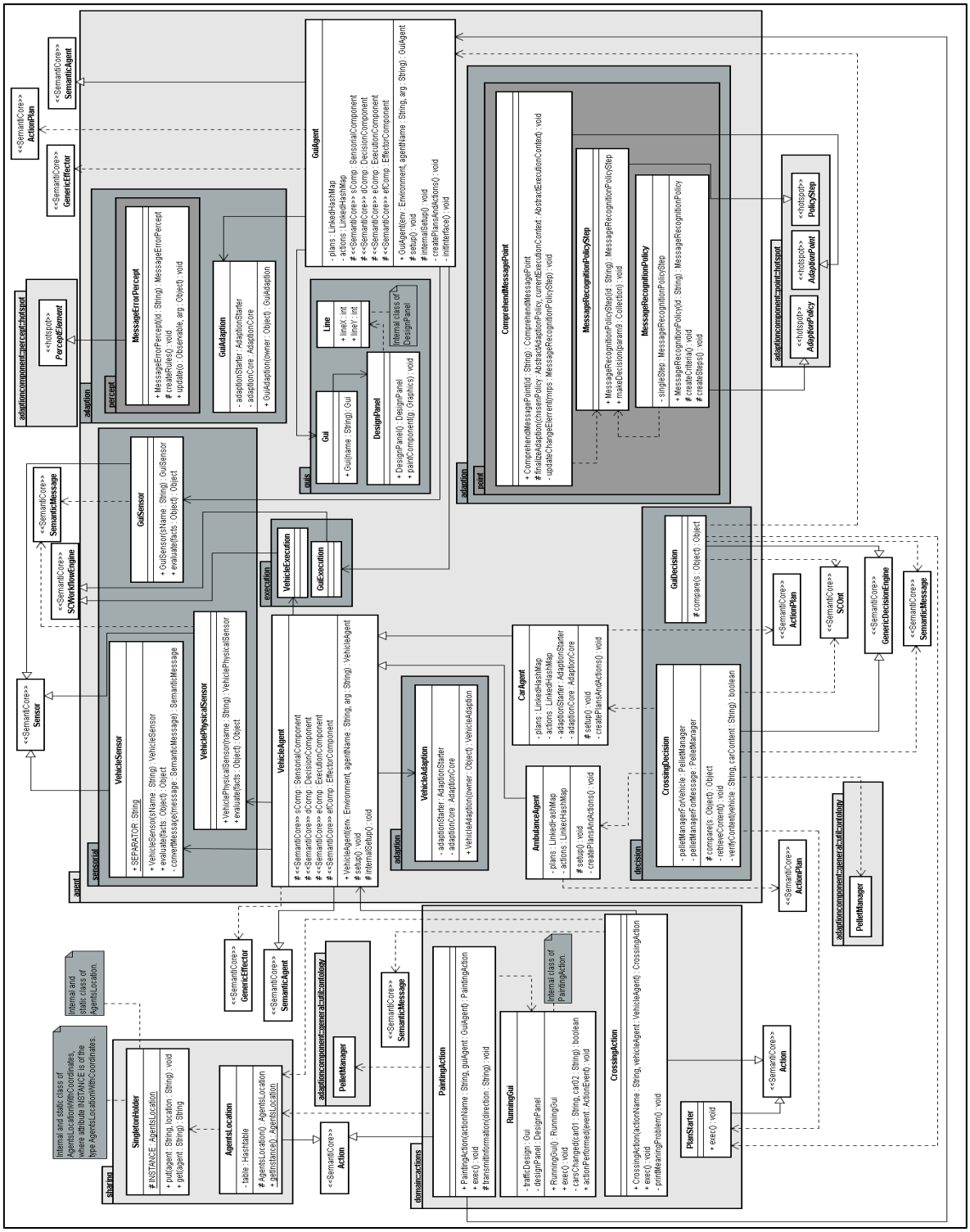

Figura 5.11 – Diagrama geral de implementação da simulação.

Semelhantemente, como aconteceu na ilustração de implementação do *framework* proposto, foi preciso suprimir alguns atributos e métodos das classes, além de alguns relacionamentos, para se poder entender com maior clareza o diagrama.

5.3.1 A Criação dos Agentes

Enquanto é explicado o desenvolvimento dos agentes no *SemantiCore*, este último também é detalhado, para facilitar a compreensão de uso da plataforma. Inicialmente, criaram-se dois tipos de agentes: um representando os veículos, por meio da classe *VehicleAgent*; e outro representando a interface gráfica que permite visualizar a animação da simulação, por meio da classe *GuiAgent*. Para a criação de agentes no *SemantiCore*, é preciso estender a classe *semanticore...SemanticAgent*, implementando os seus métodos construtor e *setup()*. Este último método é utilizado para se inserir todo o código de inicialização do agente. Na simulação, ainda foi implementado o método *internalSetup()*, sendo responsável pela definição da ordem de comunicação interna entre cada componente do agente.

Como existem dois tipos de veículos no exemplo, criaram-se duas classes que estendem a classe *VehicleAgent*: a classe *AmbulanceAgent*, representando as ambulâncias, e a classe *CarAgent*, representando os demais veículos (carros). Com essas duas extensões, deixou-se a implementação do método *setup()* da classe *VehicleAgent* para as suas especializações. Além disso, em cada especialização, bem como para o agente de interface gráfica, foram adicionadas as extensões dos componentes do *SemantiCore*, necessários para o funcionamento dos agentes. Os objetivos desses componentes (sensorial, decisório, executor e efetuador) foram explicados na seção 2.3.1.

Para os agentes representando os veículos, definiram-se dois tipos de componentes sensoriais que especializam a classe *semanticore...Sensor*: a classe *VehicleSensor* e a classe *VehiclePhysicalSensor*. A primeira representa a captura de mensagens de comunicação entre os veículos, enquanto a segunda representa o sensor físico que detecta o movimento desses veículos. É importante notar que todas as mensagens trocadas são instâncias da classe *semanticore...SemanticMessage*. O sensor *VehicleSensor* pode receber mensagens da plataforma *SemantiCore*, repassando-as diretamente para o componente decisório, ou pode receber mensagens de agentes, precisando converter, previamente, o conteúdo descrito por meio de ontologias.

O sensor *VehiclePhysicalSensor* recebe mensagens apenas do agente de interface gráfica, simulando a detecção de movimento dos veículos. Para o agente de interface gráfica,

foi definido o seu sensor por meio da classe *GuiSensor*, que apenas repassa para o componente decisório mensagens advindas do *SemantiCore*. Nas classes que representam os componentes sensoriais, é necessário implementar os métodos construtor e *evaluate()*. Neste último, é inserido o código que trata os objetos (mensagens) recuperados do ambiente.

Quanto ao componente decisório, para os agentes que representam os veículos se criou apenas uma classe que especializa a classe *semanticore...GenericDecisionEngine*, a classe *CrossingDecision*. Esta recebe as mensagens do componente sensorial e, conforme o conteúdo das mesmas, inicia os planos. Esse conteúdo é verificado por meio do método *verifyContent()*, no qual são utilizadas as regras de inferências para a tomada de decisão dos agentes. Essas regras são explicadas na seção 5.3.3, além de estarem presentes no Apêndice V. Para o agente de interface gráfica, criou-se a classe *GuiDecision*. A única função dessa classe é receber a mensagem do componente sensorial, advinda do *SemantiCore*, e iniciar o plano que exibe a simulação na tela. Nas classes que representam os componentes decisórios, especificamente o tipo *semanticore...GenericDecisionEngine*, é necessário implementar o método *compare()*, sem obrigatoriedade, permitindo a tomada de decisão do agente.

Em relação ao componente executor, os agentes que representam os veículos também utilizaram apenas uma classe, chamada *VehicleExecution*, que especializa a classe *semanticore...SCWorkflowEngine*. No caso do agente de interface gráfica, criou-se a classe *GuiExecution*. Nas classes que representam os componentes executores, especificamente o tipo *semanticore...SCWorkflowEngine*, não é necessário implementar os seus métodos. Por isso, na simulação, elas se encontram vazias.

O último componente a ser citado, o efetuador, não precisou ser estendido. Na simulação, os agentes instanciam, dentro do método *internalSetup()*, o componente efetuador do tipo *semanticore...GenericEffector*, responsável apenas por publicar as mensagens trocadas no ambiente.

Os agentes no *SemantiCore*, além dos componentes citados, precisam de planos e ações para funcionarem. Sendo assim, para cada agente foi criado o método *createPlansAndActions()*, com a função de adicionar planos e ações a eles. As instâncias da classe *semanticore...ActionPlan* representam os planos, enquanto as instâncias da classe *semanticore...Action* representam as ações. A adição de ações aos planos é feita por meio do método *addAction()* da classe *semanticore...ActionPlan*. Agora, para adicionar os planos aos agentes, utiliza-se o método *addActionPlan()* da classe *semantciore...SemanticAgent*. Os agentes dos veículos necessitam da ação definida pela classe *CrossingAction*, que estende a classe *semanticore...Action*, com as seguintes funções: (i) transmitir mensagens entre os

agentes, e (ii) atualizar as localizações dos mesmos. De maneira semelhante, o agente de interface gráfica necessita da ação definida pela classe *PaintingAction*, com as seguintes funções: (i) transmitir mensagens para os sensores do tipo *VehiclePhysicalSensor*, e (ii) repassar à classe *DesignPanel* as atualizações das localizações dos agentes de veículos. Nas classes que representam as ações, é necessário implementar um dos seus métodos construtores e também o método *exec()*, responsável pelo código que torna a ação funcional.

Voltando à explicação do agente *GuiAgent*, é preciso esclarecer a sua estrutura para a exibição da simulação. Criou-se o método *initInterface()*, chamado dentro do método *setup()*, para a instanciação da janela da aplicação, definida pela classe *Gui*. Essa classe constrói o painel que apresenta a animação do trânsito, definido pela classe *DesignPanel*. Quando as ações dos agentes (classes *CrossingAction* e *PaintingAction*) manipulam as informações de localização dos veículos, por meio da classe *AgentsLocation*, elas atualizam a exibição da animação por meio do método *setLanes()* da classe *DesignPanel*.

A classe *AgentsLocation* funciona de maneira análoga à classe *ChosenPolicies*, explicada na seção 5.1, construída seguindo o padrão de projeto *Singleton*. Assim, ela possui um método construtor protegido e um método estático (*getInstance()*) que retorna a sua única instância. Este último método utiliza uma classe interna, definida aqui como *SingletonHolder*, composta por apenas uma constante que recebe essa instância. A classe *AgentsLocation* ainda possui uma tabela *hash* contendo as localizações dos veículos (atributo *table*).

5.3.2 A Instanciação do Framework para a Simulação

Construídos os agentes, seguiu-se a implementação com a instanciação do *framework* proposto, por meio da extensão dos seus *hotspots*. Inicialmente, como a simulação exige a adaptação do carro mais distante da ambulância que se aproxima, optou-se por integrar o *framework* a partir da classe *CarAgent*. Ela instancia a classe *AdaptionStarter*, que retorna a instância do núcleo de adaptação (classe *AdaptionCore*). É importante guardar a referência do núcleo de adaptação porque é por meio dele que se consegue notificar a mudança de estado de um elemento observado.

Quando o *framework* inicia, lêem-se dois arquivos, citados anteriormente na seção 5.1: *adaptioncomponent.properties* e *adaptionconfig.xml*. O primeiro é dividido em duas partes, uma contendo as propriedades que não podem ser modificadas, e a outra definindo três tipos de propriedades de execução (para o protótipo, estas últimas propriedades também não influenciam a sua execução). São elas:

- *reasoner.type*: define se o mecanismo de inferência a ser utilizado é diretamente o *Pellet* ou a sua API para o *Jena* – respectivamente, os valores *pellet* ou *jena*.
- *no.internal*: define se a execução do *framework* se dá a partir de uma linha de comando ou de dentro de um agente – respectivamente, os valores *true* ou *false*.
- *no.owner*: define se é passado ou não o objeto que inicia a execução do *framework* – respectivamente, os valores *false* ou *true*.

O segundo arquivo – *adaptionconfig.xml* – contém as definições dos *hotspots* implementados, que são explicados nesta seção. Atualmente, têm-se os seguintes elementos:

- <*adaptioncomponent*></*adaptioncomponent*>: delimita o início e o fim do arquivo.
- <*perceptelement*>: permite instanciar um elemento de percepção. É preciso definir um nome (atributo *name*) e a classe que o implementa (atributo *class*).
- <*relatedpointpercept*>: é um elemento interno ao <*perceptelement*>, permitindo relacionar este último a um ponto de adaptação. É preciso definir um nome (atributo *name*).
- <*adaptionpoint*>: permite instanciar um ponto de adaptação. É preciso definir um nome (atributo *name*) e a classe que o implementa (atributo *class*).
- <*policy*>: é um elemento interno ao <*adaptionpoint*>, permitindo relacionar este último a uma política de adaptação. É preciso definir um nome (atributo *name*).
- <*adaptionpolicy*>: permite instanciar uma política de adaptação. É preciso definir um nome (atributo *name*) e a classe que o implementa (atributo *class*).

Ambas as classes de configuração estão exemplificadas no Apêndice C. Nota-se, no arquivo *adaptionconfig.xml*, um atributo *arg* dentro dos seus elementos, atualmente sem funcionalidade, mas com a futura função de repassar argumentos às classes implementadas.

Definido o local de integração do *framework* dentro da simulação, observou-se a melhor localização para as notificações dos elementos observados. Assim, optou-se por inseri-las na classe *CrossingDecision*, pois todas as informações (mensagens) recebidas pelos agentes precisam passar pela classe que especializa um tipo de componente decisório do *SemantiCore*, possibilitando a tomada de decisão. As estruturas ontológicas das mensagens trocadas entre os veículos, das mensagens armazenadas nos veículos, e das informações dos sensores físicos, podem ser vistas na seção 5.3.3. Além disso, a codificação dessas estruturas, na linguagem OWL, encontra-se no Apêndice D.

O elemento de percepção da simulação, chamado *messageerrorpercept*, é implementado pela classe *MessageErrorPercept*. Ela possui apenas uma regra (*rule1*) composta pela string *comprehendMessage* (mesmo nome do ponto de adaptação da simulação). Quando chega uma informação da classe *CrossingDecision* com o conteúdo *message misunderstood*, dispara-se a regra *rule1*, conseqüentemente, iniciando o ponto de adaptação associado.

O ponto de adaptação da simulação, chamado *comprehendMessage*, é implementado pela classe *ComprehendMessagePoint*. Quando ela é instanciada, são definidos o seu tipo e as informações do seu descritor: tipo – *RuleAdaption* (adaptação de regra); definição – *Update a rule* (atualiza uma regra); domínio – *Message Error* (erro de mensagem); criador – *SemanicAgent* (agente do *SemantiCore*). Após a verificação de uma política, escolhida por meio da classe *Knowledge*, para esse ponto de adaptação, capturam-se e executam-se os passos dessa política.

A política de adaptação presente na simulação, chamada *messageRecognitionPolicy*, é implementada pela classe *MessageRecognitionPolicy*. Nela, adicionam-se três critérios para a sua escolha: *criterion01*, contendo uma string com o conteúdo *message misunderstood*; *criterion02*, contendo uma regra de inferência que permite verificar se uma mensagem é de emergência; e *criterion03*, contendo uma instância da classe *Property* (explicada na seção 5.1) para verificar se falta um indivíduo dentro da ontologia da mensagem trocada entre os veículos. A regra de inferência do segundo critério é explicada na seção 5.3.3, e também está presente no Apêndice E. Ainda é preciso criar os passos da política, mas apenas um é implementado na simulação.

O passo da política *messageRecognitionPolicy*, chamado *singleStep*, é implementado pela classe *MessageRecognitionPolicyStep*. Como o *framework* não exige a implementação de métodos específicos para os passos de políticas, o método *makeDecision()*, desenvolvido apenas para essa simulação, recebe a informação de movimentação dos veículos e guarda as regras que podem ser adicionadas para a adaptação final do elemento de mudança no agente. O formato final das regras de inferência adaptadas, utilizadas sobre as ontologias das mensagens trocadas entre os veículos, é explicado na seção 5.3.3, e também está presente no Apêndice E.

5.3.3 Executando o Protótipo

Finalizada a implementação dos agentes, e acoplado o *framework* aos mesmos, iniciou-se a execução do protótipo. Precisou-se configurar a plataforma *SemantiCore* para a execução da simulação com o cenário de aplicação. No *SemantiCore*, é necessário ajustar os arquivos de configuração para o seu perfeito funcionamento. São eles:

- *semanticoreconfig.xml*: permite definir se a interface de usuário da plataforma é visível, e também definir os agentes a serem instanciados (por meio do nome, da classe de implementação, e de algum argumento a ser repassado).
- *semanticoreinstantiation.xml*: permite definir as classes que implementam os mecanismos dos componentes decisório, executor e efetuador.
- *semanticore.properties*: permite definir opções para a interface de usuário (arquivo não modificado para a simulação).

Esses arquivos, bem como as suas explicações, podem ser vistos no Apêndice F. Após o ajuste dos arquivos, consegue-se executar a plataforma. Para isso, chama-se o método *main()* da classe *semanticore.domain.SemantiCore*. Como argumentos para esse método, na simulação, devem-se passar: o valor de controle *main*, o número da porta de conexão local, e nome do domínio do *SemantiCore*. Como exemplo, tem-se a seguinte expressão de argumentos:

main 10000 traffic

Iniciada a plataforma, aparecem as interfaces gráficas da mesma (Figura 5.12) e da simulação (Figura 5.13).

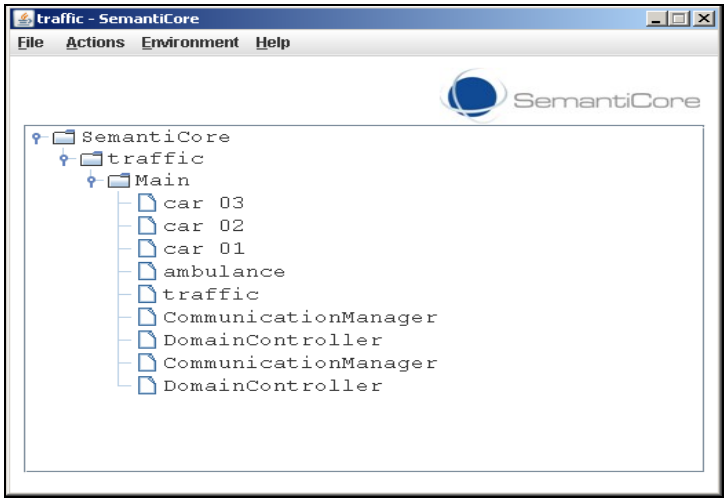

Figura 5.12 – Tela inicial do *SemantiCore* para a simulação.

Na Figura 5.12, percebe-se a presença de nove agentes em execução, dentro do domínio *traffic*, dos quais cinco fazem parte da simulação. Esses cinco agentes são: *car 01*, *car 02*, *car 03*, *ambulance* e *traffic*. Os três primeiros são instâncias da classe *CarAgent*, enquanto o agente *ambulance* é uma instância da classe *AmbulanceAgent*. O último agente, *traffic*, é aquele responsável pela exibição da simulação, sendo uma instância da classe *GuiAgent*.

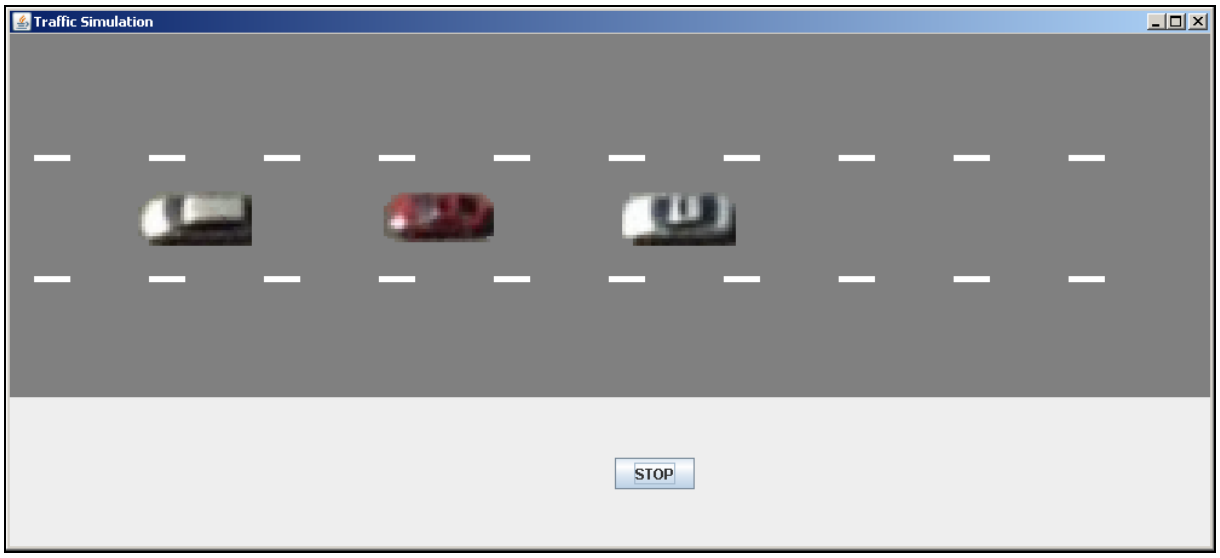

Figura 5.13 – Tela inicial da simulação.

A Figura 5.13 apresenta a tela inicial da simulação – *Traffic Simulation*. Nota-se a presença de três carros em uma estrada, além do botão STOP, utilizado para encerrar a execução da interface gráfica.

O primeiro passo a ser tomado para executar a interface gráfica é enviar uma mensagem para o agente *traffic*. Na janela principal do *SemantiCore*, seleciona-se o menu *Actions*, e depois o item *Send Message*. Na janela *Input*, insere-se o nome do agente (*traffic*) no campo *To ?*, clicando-se, em seguida, no botão OK (Figura 5.14).

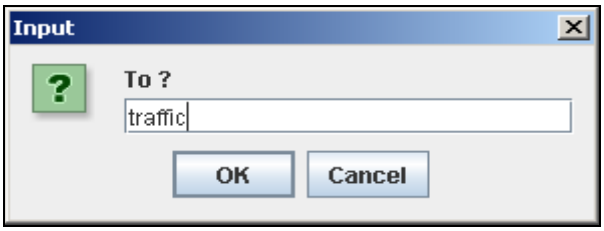

Figura 5.14 – Escolhendo o agente *traffic* para o envio da primeira mensagem.

Novamente, surge a janela *Input*, requisitando o conteúdo da mensagem. Insere-se a palavra *start* no campo *Content ?* e, na seqüência, clica-se no botão OK (Figura 5.15). Assim, a simulação começa a ser animada.

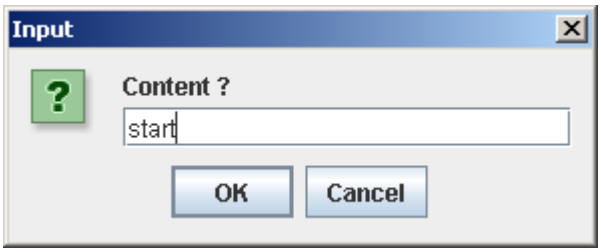

Figura 5.15 – Inserindo o conteúdo da mensagem para o agente *traffic*.

Com a interface gráfica funcionando, ainda é preciso enviar uma mensagem para o agente *ambulance*. Assim, volta-se na janela principal do *SemantiCore*, seleciona-se o menu *Actions*, e depois o item *Send Message*. Na janela *Input*, insere-se a palavra *ambulance* no campo *To ?*, clicando-se, em seguida, no botão OK (Figura 5.16).

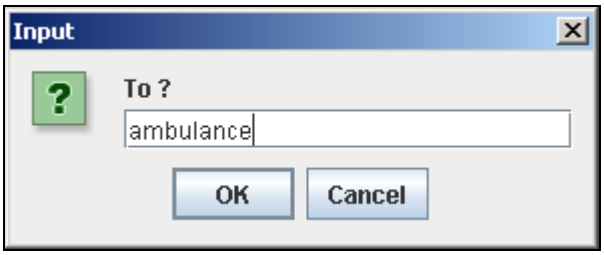

Figura 5.16 – Escolhendo o agente *ambulance* para o envio da segunda mensagem.

Por fim, insere-se o conteúdo da mensagem no campo *Content ?* da nova janela *Input* e, na seqüência, clica-se no botão OK (Figura 5.17). Esse conteúdo pode receber dois valores diferentes:

- *start right*: opta-se por apresentar a simulação com a requisição da ambulância pedindo para os veículos se manterem na faixa da direita.
- *start left*: opta-se por apresentar a simulação com a requisição da ambulância pedindo para os veículos se manterem na faixa da esquerda.

Enviada a mensagem para o agente *ambulance*, ele constrói uma nova mensagem contendo as informações sobre a necessidade da ambulância em ultrapassar os carros devido a uma emergência. Essa nova mensagem é repassada entre os carros e é descrita por uma estrutura ontológica, presente na Figura 5.18.

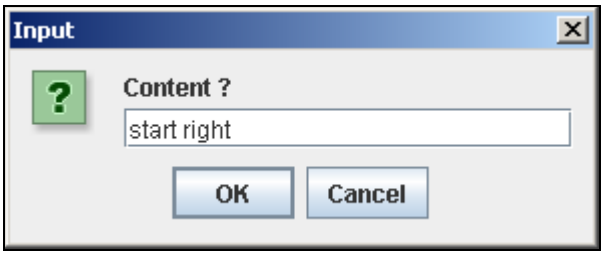

Figura 5.17 – Inserindo o conteúdo (exemplo para a direita) da mensagem para o agente *ambulance*.

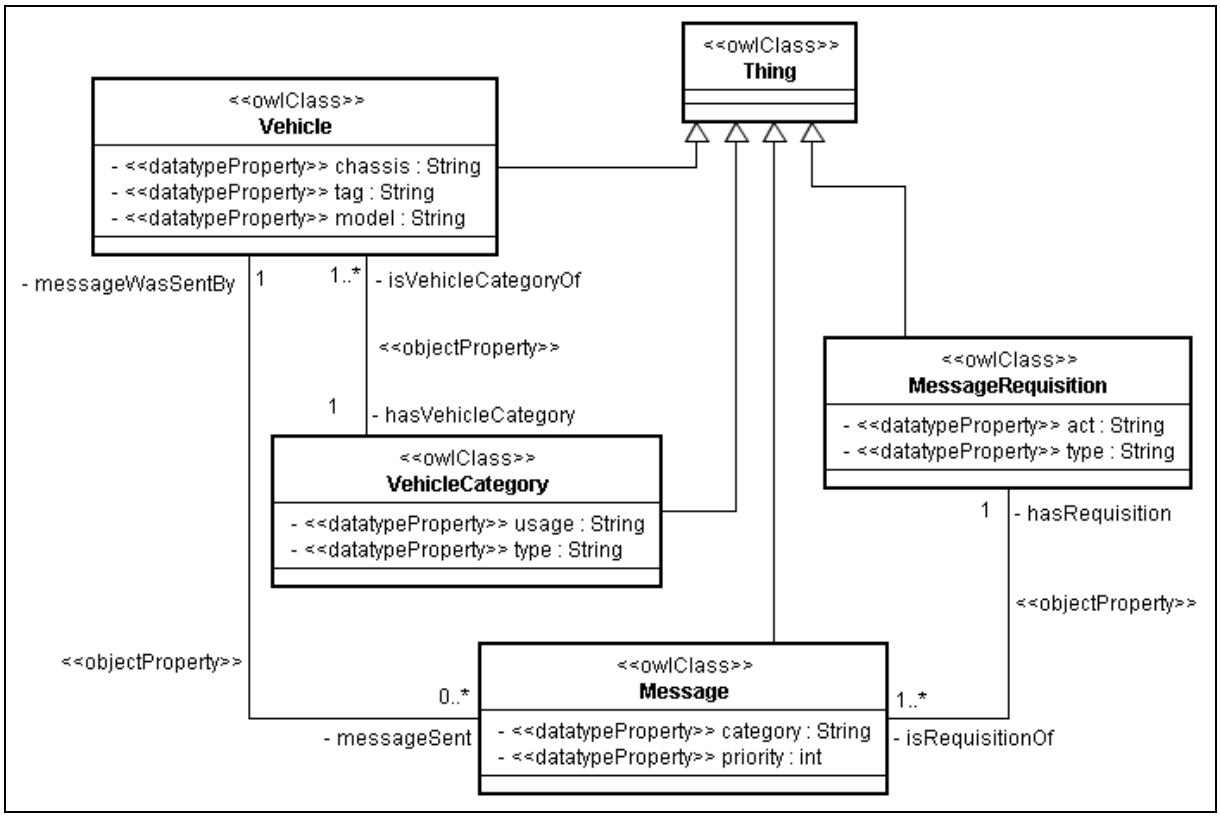

Figura 5.18 – Estrutura ontológica das mensagens trocadas entre os veículos.

Observando a Figura 5.18, é possível notar que as mensagens identificam o veículo emissor da mesma, por meio da classe OWL *Vehicle*. O veículo possui um chassi (propriedade *chassis*), uma placa (propriedade *tag*), um modelo (propriedade *model*), uma categoria (propriedade *hasVehicleCategory*), e uma mensagem enviada (propriedade *messageSent*). A classe OWL *Message* identifica a categoria da mensagem (propriedade *category*), a prioridade de importância (propriedade *priority*), o seu emissor (propriedade *messageWasSentBy*), e a sua requisição (propriedade *hasRequisition*). A categoria do veículo é identificada pela classe OWL *VehicleCategory*, possuindo o seu uso (propriedade *usage*, como ambulância ou de passeio), um tipo (propriedade *type*, como categoria E), e o veículo em questão (propriedade *isVehicleCategoryOf*). A requisição da mensagem é descrita pela classe OWL *MessageRequisition*, possuindo um ato (propriedade *act*, como pedir para um

veículo trocar de pista), um tipo (propriedade *type*, como emergência), e a mensagem em questão (propriedade *isRequisitionOf*).

Compreendida a estrutura ontológica da mensagem criada pelo agente *ambulance*, o seu conteúdo, para o exemplo no qual os carros precisam tomar a pista da direita, é apresentado na Figura 5.19.

```
1.    (ns:message rdf:type ns:Message)
2.    (ns:message ns:hasRequisition ns:messageRequisition)
3.    (ns:message ns:messageWasSentBy ns:ambulance)
4.    (ns:message ns:category "emergency")
5.    (ns:message ns:priority "1")
6.    (ns:messageRequisition rdf:type ns:MessageRequisition)
7.    (ns:messageRequisition ns:isRequisitionOf ns:message)
8.    (ns:messageRequisition ns:act "go right")
9.    (ns:messageRequisition ns:type "crossing")
10.   (ns:ambulance rdf:type ns:Vehicle)
11.   (ns:ambulance ns:hasVehicleCategory ns:eCategory)
12.   (ns:ambulance ns:messageSent ns:message)
13.   (ns:ambulance ns:chassis "04")
14.   (ns:ambulance ns:model "M4")
15.   (ns:ambulance ns:tag "IAT0404")
16.   (ns:eCategory rdf:type ns:VehicleCategory)
17.   (ns:eCategory ns:isVehicleCategoryOf ns:ambulance)
18.   (ns:eCategory ns:type "E")
19.   (ns:eCategory ns:usage "ambulance")
```

Figura 5.19 – Conteúdo da mensagem criada pelo agente *ambulance*, para a requisição de troca de pista à direita.

Na Figura 5.19, o conteúdo da mensagem pode ser lido, resumidamente, da seguinte maneira: a mensagem (*message* – linhas 1 a 5) de emergência (categoria *emergency* – linha 4) com prioridade alta (valor "1" – linha 5) contém uma requisição (*messageRequisition* – linhas 6 a 9) de ultrapassagem (tipo *crossing* – linha 9) para os carros tomarem a pista da direita (ação *go right* – linha 8); enviada pelo veículo (*ambulance* – linhas 10 a 15) de chassi "04" (linha 13), modelo "M4" (linha 14) e placa "IAT0404" (linha 15); de categoria (*eCategory* – linhas 16 a 19) tipo "E" (linha 18) e utilizado como ambulância (*ambulance* – linha 19).

Após construir a mensagem, o agente *ambulance* a repassa para o carro mais próximo, representado pelo agente *car 01*. A Figura 5.20 ilustra a transmissão da mensagem entre esses agentes.

Explicando o tratamento da mensagem recebida (válido para todos os veículos), o agente *car 01* captura-a pelo sensor *VehicleSensor* e a encaminha para o componente decisório *CrossingDecision*. Essa mensagem é armazenada no agente e, como ela é compreendida, a ação *CrossingAction* repassa-a para o agente *car 02*, enquanto o *car 01* troca de pista para possibilitar a passagem da ambulância (Figura 5.21).

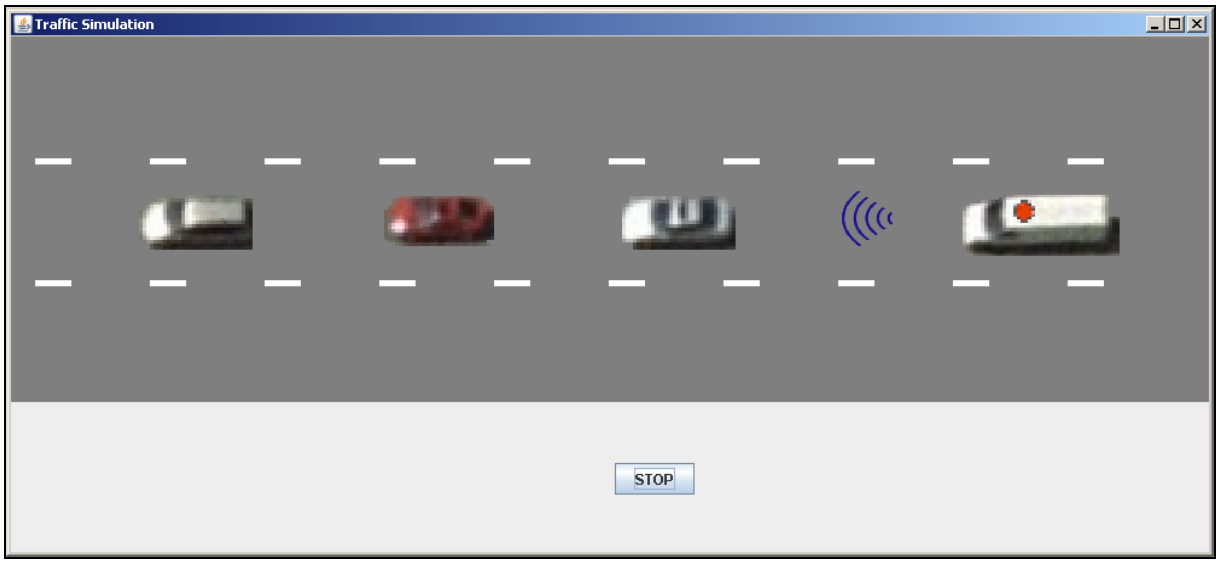

Figura 5.20 – Transmissão de mensagem entre os agentes *ambulance* e *car 01*.

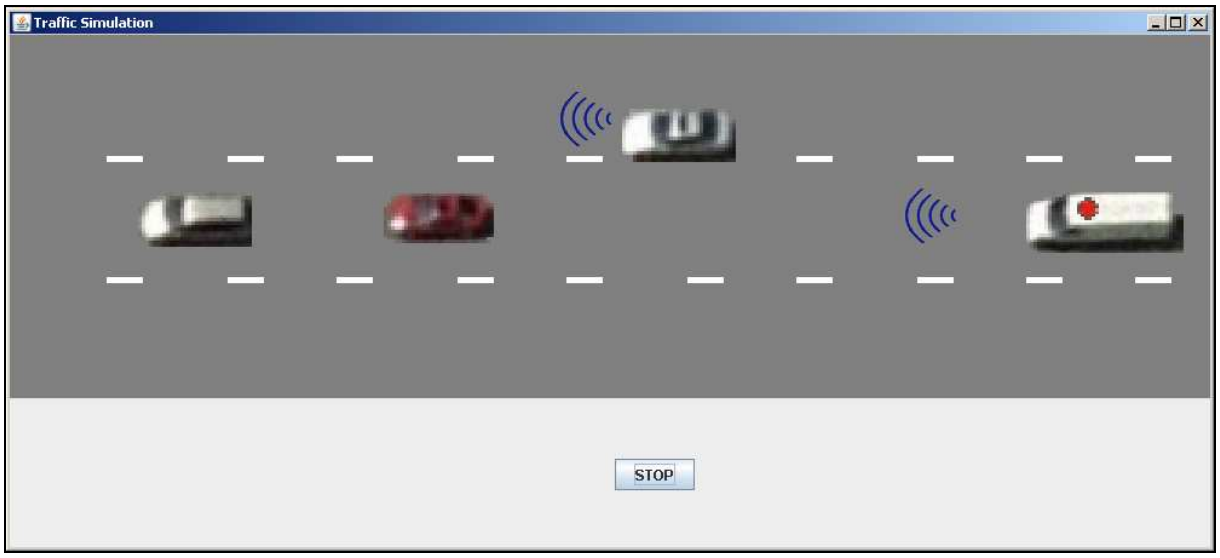

Figura 5.21 – Agente *car 01* trocando de pista e transmitindo a mensagem para o agente *car 02*.

A compreensão da mensagem é feita por meio de inferência sobre a sua ontologia, e existem regras para essa inferência. Criaram-se duas regras para os veículos inferirem sobre as mensagens trocadas: uma para a requisição da ambulância de passagem pela esquerda, e outra de passagem pela direita. A Figura 5.22 apresenta a regra que infere a requisição de passagem pela direita. Todas as regras usadas na simulação estão presentes no Apêndice E.

Essa regra (*rule1*) pode ser lida da seguinte forma: quando uma mensagem (*?message*) tem uma categoria (*?category*) e uma requisição (*?requisition*); e essa requisição é de um tipo (*?type*) e possui um ato (*?act*); e a categoria (*?category*) for uma emergência (igual a *emergency*), o tipo (*?type*) for uma ambulância pedindo para ultrapassar (igual a *crossing*), e o ato (*?act*) for uma ação para os carros se manterem na pista da direita (igual a *go right*); então,

consegue-se inferir que da mensagem (*?message*) é formada uma decisão (*tf:decision*) para se manter na pista da direita (valor *right*).

```
[rule1:
       (?message rdf:type tf:Message)
       (?message tf:category ?category)
       (?message tf:hasRequisition ?requisition)
       (?requisition tf:type ?type)
       (?requisition tf:act ?act)
       equal (?category, "emergency")
       equal (?type, "crossing")
       equal (?act, "go right")

       -> (?message tf:decision "right")]
```

Figura 5.22 – Uma das regras de inferência para as mensagens trocadas entre os veículos.

O armazenamento das mensagens recebidas é feito dentro da classe de tomada de decisão – *CrossingDecision*. A Figura 5.23 apresenta a estrutura ontológica para esse armazenamento.

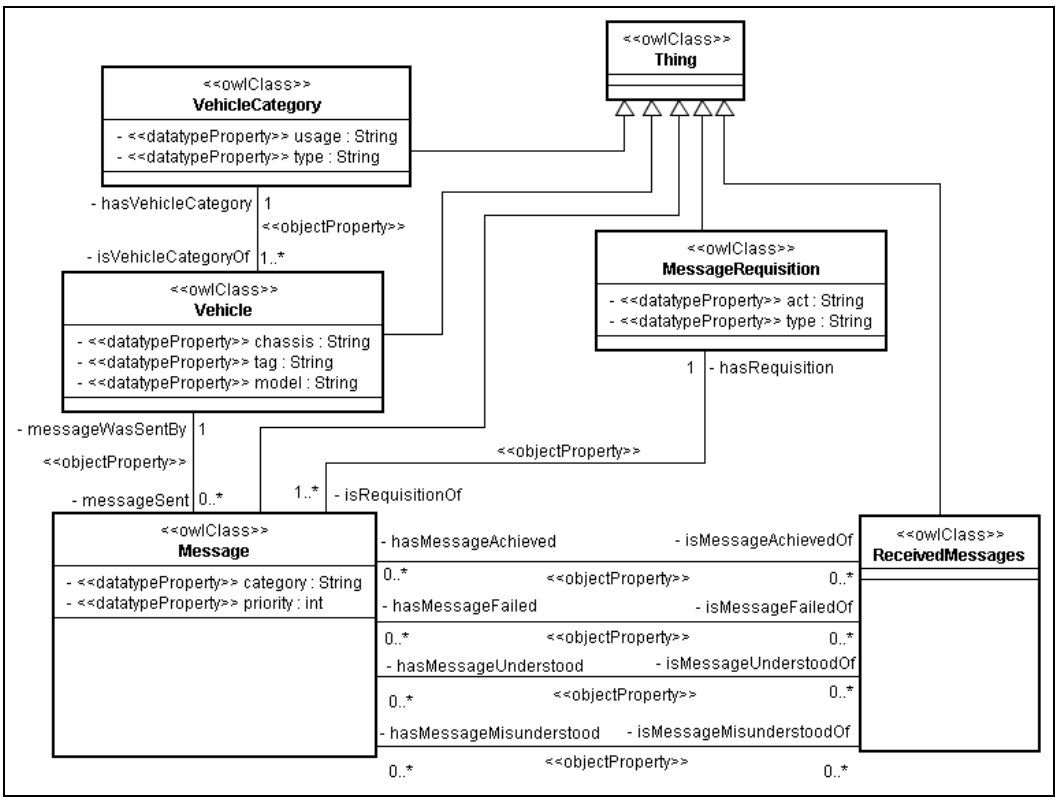

Figura 5.23 – Estrutura ontológica das mensagens armazenadas nos veículos.

Analisando a Figura 5.23, nota-se a sua semelhança com a Figura 5.18. Além das mesmas classes OWL presentes na estrutura ontológica das mensagens trocadas entre os veículos, surge apenas uma nova classe OWL, a *ReceivedMessages*. Ela armazena os estados

alcançados pelo processamento das mensagens recebidas: mensagem cumprida (propriedade *hasMessageAchieved*), mensagem falhada (propriedade *hasMessageFailed*), mensagem compreendida (propriedade *hasMessageUnderstood*), e mensagem não compreendida (propriedade *hasMessageMisunderstood*). Com isso, a classe OWL *Message* recebe novas propriedades para o relacionamento com a classe OWL *ReceivedMessages*: *isMessageAchievedOf*, *isMessageFailedOf*, *isMessageUnderstoodOf*, e *isMessageMisunderstoodOf*.

O processo de troca de mensagens entre os agentes *car 02* e *car 03* é semelhante ao processo de troca executado entre os agentes *car 01* e *car 02*, excetuando-se a estrutura ontológica das mensagens. Como a simulação reproduz um truncamento da mensagem recebida pelo agente *car 03*, devido a uma falha de comunicação, sua ontologia não apresenta a classe OWL *MessageRequisition* (Figura 5.24).

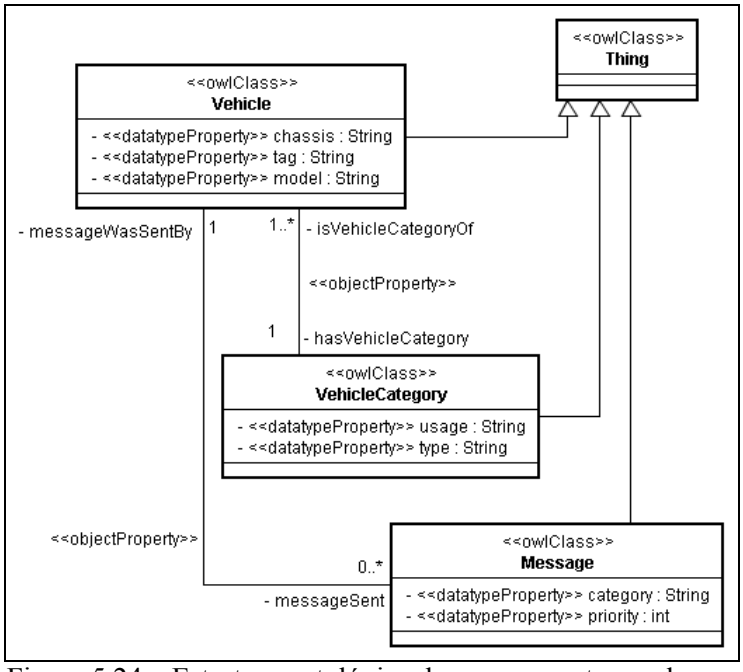

Figura 5.24 – Estrutura ontológica da mensagem truncada.

O agente *car 03* recebe a mensagem transmitida pelo agente *car 02* (Figura 5.25 – o agente *car 02* também troca de pista para dar passagem à ambulância) e, no momento em que não consegue inferir uma tomada de decisão por problemas na mensagem, o *framework* é sinalizado, tentando-se efetuar uma adaptação. A Figura 5.26 apresenta o conteúdo da mensagem truncada.

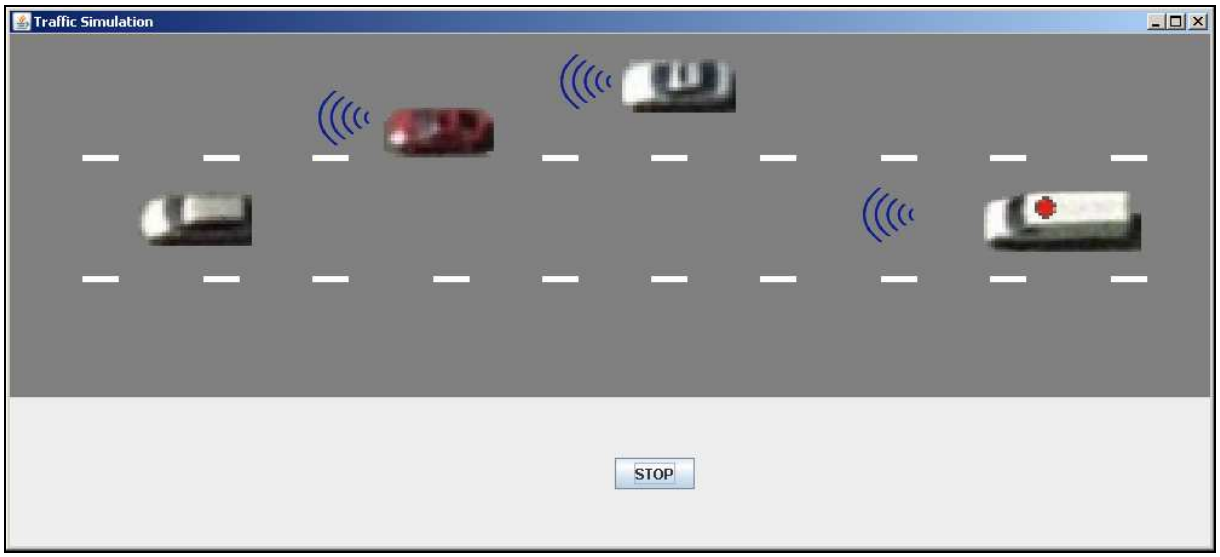

Figura 5.25 – Agente *car 02* trocando de pista e transmitindo a mensagem para o agente *car 03*.

```
1.    (ns:message rdf:type ns:Message)
2.    (ns:message ns:hasRequisition ns: )
3.    (ns:message ns:messageWasSentBy ns:ambulance)
4.    (ns:message ns:category "emergency")
5.    (ns:message ns:priority "1")
6.    (ns:ambulance rdf:type ns:Vehicle)
7.    (ns:ambulance ns:hasVehicleCategory ns:eCategory)
8.    (ns:ambulance ns:messageSent ns:message)
9.    (ns:ambulance ns:chassis "04")
10.   (ns:ambulance ns:model "M4")
11.   (ns:ambulance ns:tag "IAT0404")
12.   (ns:eCategory rdf:type ns:VehicleCategory)
13.   (ns:eCategory ns:isVehicleCategoryOf ns:ambulance)
14.   (ns:eCategory ns:type "E")
15.   (ns:eCategory ns:usage "ambulance")
```

Figura 5.26 – Conteúdo da mensagem truncada.

Na Figura 5.26, nota-se que o conteúdo da mensagem é semelhante ao apresentado na Figura 5.19, mas como a mensagem está incompleta, falta um indivíduo da classe OWL *MessageRequisition*. Conseqüentemente, a propriedade *hasRequisition* (linha 2) do indivíduo *message* se encontra vazia.

A sinalização ao *framework* para iniciar a adaptação é capturada pelo elemento de percepção *messageerrorpercept*. Ele recebe uma informação da classe *CrossingDecision* com o conteúdo *message misunderstood*, casando-a com a regra de percepção *rule1*. Essa regra dispara o início do ponto de adaptação *comprehendMessage*. O núcleo de adaptação detecta a mudança de estado do elemento de percepção e sinaliza a classe *Knowledge*, para a criação de um contexto de execução corrente. As propriedades desse contexto são utilizadas para se escolher uma política de adaptação. Na simulação, a política *messageRecognitionPolicy* é escolhida com base nos três critérios citados na seção 5.3.2. Dentre eles, o critério *criterion02*

contém uma regra de inferência que está presente na Figura 5.27. Essa regra permite verificar se uma mensagem é de emergência.

```
[rule1:
       (?message rdf:type tf:Message)
       (?message tf:category ?category)
       equal (?category, "emergency")
       -> (?message tf:match_result "true")]
```

Figura 5.27 – Regra de inferência para o segundo critério de escolha de política.

A regra da Figura 5.27 (*rule1*) pode ser lida da seguinte forma: quando uma mensagem (*?message*) tem uma categoria (*?category*), e essa categoria for uma emergência (igual a *emergency*); então, consegue-se inferir que da mensagem (*?message*) se chega a um resultado (*tf:match_result*) verdadeiro (valor *true*).

Escolhida a política de adaptação, o ponto de adaptação executa o único passo dela, chamado *singleStep*. Nesse passo, uma nova informação é utilizada: a movimentação dos veículos capturada pelo sensor físico *VehiclePhysicalSensor*. Esse sensor captura a movimentação e informa o agente por meio de uma mensagem, composta por uma estrutura ontológica simples, apresentada na Figura 5.28.

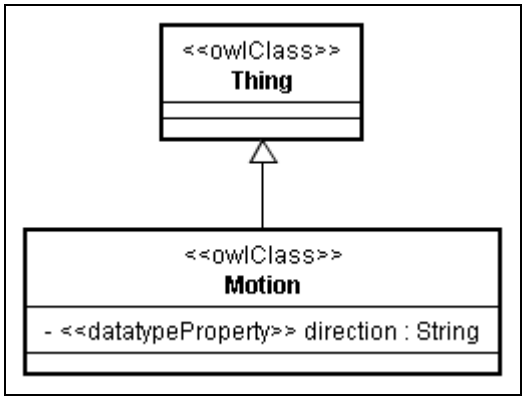

Figura 5.28 – Estrutura ontológica das mensagens advindas do sensor físico.

Examinando-se a Figura 5.28, nota-se apenas a existência da classe OWL *Motion*, com a propriedade *direction*, responsável por informar a direção de movimentação dos veículos detectados pelo sensor físico.

Além de receber a informação de movimentação dos veículos, o passo *singleStep* guarda as regras de inferência que são adicionadas durante a adaptação final do elemento de mudança do agente, capazes de possibilitar uma nova tomada de decisão. Assim, quando o ponto de adaptação atualiza as regras existentes para as mensagens trocadas entre os veículos,

criam-se duas novas. A Figura 5.29 apresenta uma das novas regras criadas, referente à inferência para a passagem da ambulância pela esquerda.

```
[rule3:
       (?message rdf:type tf:Message)
       (?message tf:category ?category)
       (?motion rdf:type ps:Motion)
       (?motion ps:direction ?direction)
       equal (?category, "emergency")
       equal (?direction, "left")

       -> (?message tf:decision "left")]
```

Figura 5.29 – Uma das regras de inferência criadas após a adaptação realizada pelo *framework*.

A regra da Figura 5.29 (*rule3*) pode ser lida da seguinte forma: quando uma mensagem (*?message*) tem uma categoria (*?category*), e uma informação do sensor físico (*?motion*) indica uma direção (*?direction*); e a categoria (*?category*) for uma emergência (igual a *emergency*), e a direção (*?direction*) indicar que os carros se movem para a pista da esquerda (igual a *left*); então, consegue-se inferir que da mensagem (*?message*) é formada uma decisão (*tf:decision*) para se manter na pista da esquerda (valor *left*).

Observando-se a nova regra de inferência, uma dúvida pode surgir: como é possível o agente inferir sobre duas ontologias diferentes – das mensagens trocadas entre os veículos e das mensagens advindas do sensor físico – utilizando regras que combinam informações das duas ontologias? A resposta é simples. Antes da adaptação, o agente só levava em consideração a inferência sobre as mensagens trocadas entre os veículos. Após a adaptação, o agente passa a inferir sobre uma ontologia que é a combinação das duas anteriores, permitindo o uso de novas regras que contemplam as informações de ambas.

Na simulação, o ciclo de adaptação executado pelo *framework* está completo. Infelizmente, segundo a descrição do cenário de aplicação, o agente *car 03* não consegue se adaptar a tempo de mudar de pista. Assim, a passagem sem interrupção da ambulância não é alcançada. A Figura 5.30 apresenta o bloqueio do terceiro carro, enquanto a Figura 5.31 mostra o desvio da ambulância.

Quando em um segundo momento ocorre o mesmo problema de falha na comunicação, o agente *car 03*, mantendo as novas regras de inferência, pode tomar a decisão de trocar de pista sem atrapalhar a passagem da ambulância. A Figura 5.32 ilustra essa situação.

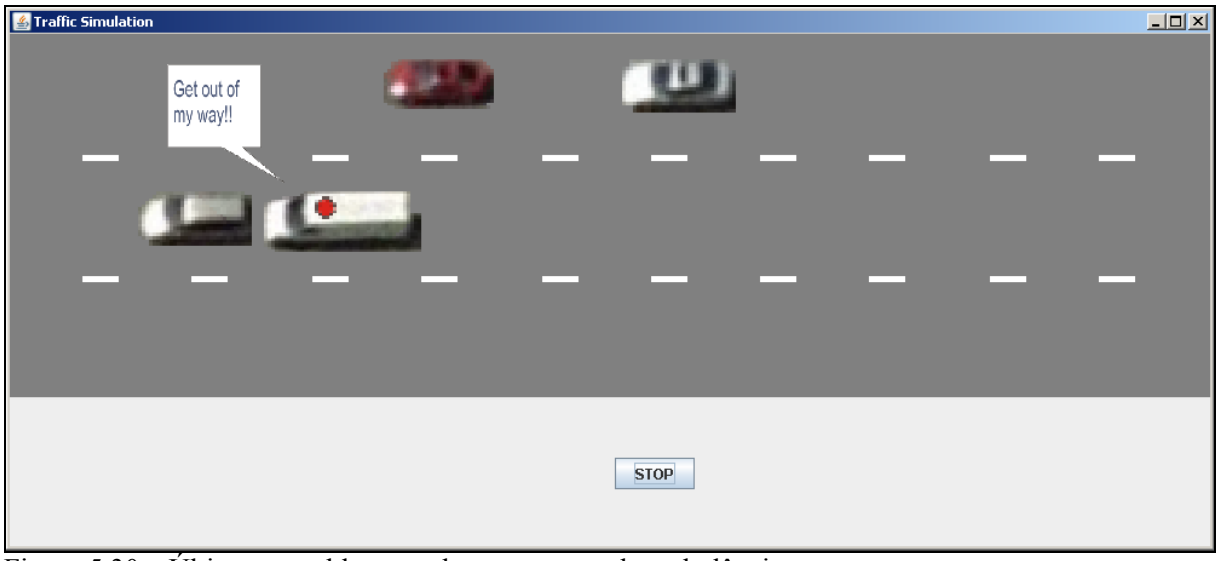

Figura 5.30 – Último carro bloqueando a passagem da ambulância.

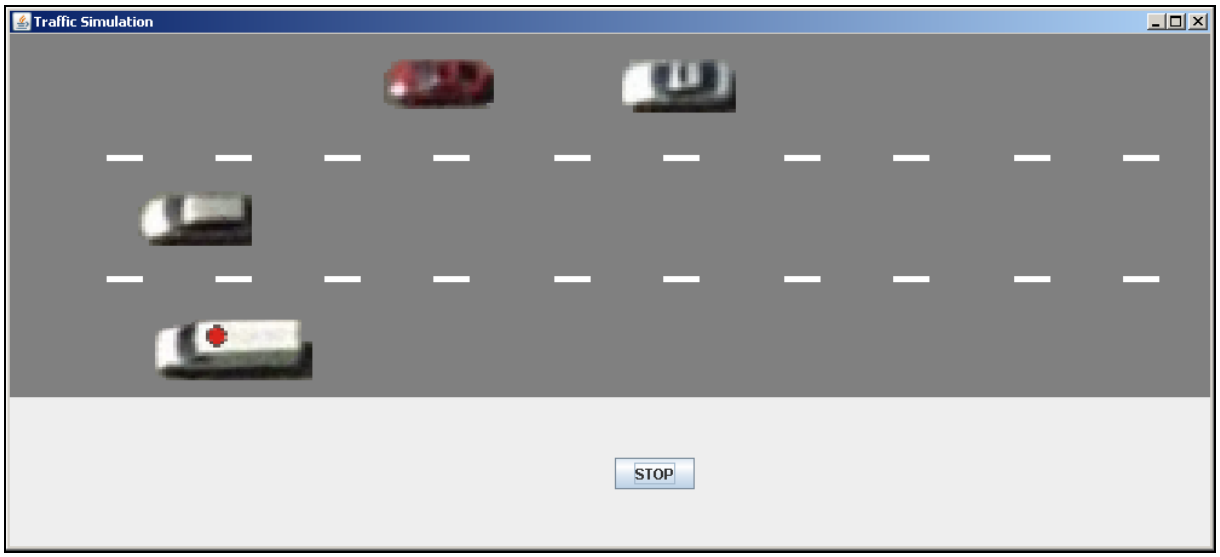

Figura 5.31 – Ambulância desviando do último carro.

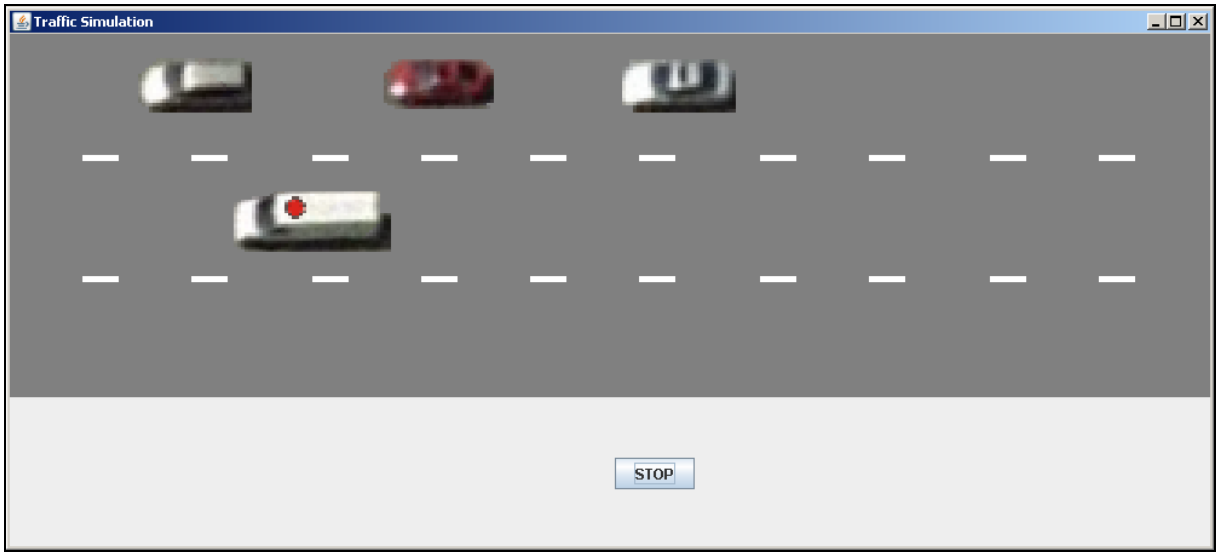

Figura 5.32 – Agente *car 03* com as novas regras de inferência, permitindo a passagem da ambulância em uma segunda situação.

Finalizando a simulação, o botão STOP pode ser utilizado para terminar a animação da interface gráfica. Para fechar a plataforma *SemantiCore*, seleciona-se o item *Shutdown Semanticore* no menu *File*.

5.4 Considerações sobre o Capítulo

Este capítulo apresentou o desenvolvimento do protótipo e a sua utilização em um cenário de aplicação, por meio de uma simulação. A descrição do protótipo, com a visualização detalhada de cada pacote, possibilitou a demonstração de um diagrama de implementação mais sucinto, facilitando a compreensão do sistema como um todo, além de estar aliado à explicação do fluxo geral de execução. É importante salientar que a implementação do *framework* foi simplificada, sabendo-se que no futuro suas funções podem ser ampliadas.

A apresentação de um cenário de aplicação, ainda não concretamente viável, ilustrou o uso de novos paradigmas e tecnologias, incentivando a continuação da pesquisa dos mesmos e do *framework*. Adicionalmente, prevendo-se que o sistema de transporte será caótico pela quantidade de veículos, é importante buscar alternativas para esse problema, principalmente relacionado à segurança dos usuários desse sistema.

Quanto à simulação do cenário de aplicação, o seu esclarecimento permitiu aprofundar a compreensão da plataforma *SemantiCore*, dos pontos de flexibilidade do *framework* proposto, e da adaptação em regras de inferência. A instanciação do *framework*, além de elucidar os ajustes nos arquivos de configuração, demonstrou que a tarefa de utilização dos *hotspots* não é extensa. Por fim, explicar a execução do *SemantiCore* foi benéfico para se notar como o seu uso é simples.

6 CONSIDERAÇÕES FINAIS E TRABALHOS FUTUROS

Sabendo-se da atual importância em se utilizar agentes adaptativos capazes de se ajustarem às mudanças dos ambientes nos atuais sistemas computacionais complexos, este trabalho apresentou o desenvolvimento de um *framework* que permite a adaptação estrutural e comportamental de agentes de software na Web Semântica, de acordo com o contexto composto por propriedades descritas, principalmente, por meio de ontologias. Esses agentes manipulam conhecimento e se adaptam conforme as informações do contexto de execução e conforme as políticas de adaptação definidas.

Quando se analisou os diversos significados da adaptação em agentes de software, ficou clara a dificuldade de se chegar a um consenso. Apesar disso, é possível que se definam vários tipos de adaptação sem a exclusão de conceituações tão importantes, presentes nos trabalhos pesquisados. Até pelo seu significado mais simples, de se adequar uma coisa a outra, qualquer mudança em qualquer nível de um agente poderia ser considerada uma adaptação, indo de uma visão reativa e individual a um processo pró-ativo e coletivo. Dessa forma, o conceito de adaptação definido para esta pesquisa é apenas uma parte específica do macro significado.

Buscando-se um caminho diferente dos encontrados nos trabalhos relacionados, no *framework* proposto não se defende o uso de uma arquitetura que siga um processo fixo de adaptação. Assim, uma comparação de desempenho entre esses trabalhos e o *framework* é uma tarefa difícil, inclusive por não terem sido encontradas métricas para a execução dessa atividade. Seria de grande valia encontrar uma maneira de medir, por meio de experimentos, os ganhos em se utilizar o *framework* como um facilitador para o desenvolvimento de agentes adaptativos. Apesar da falta de métricas, os prováveis ganhos seriam:

- Possibilidade da escolha das estratégias de adaptação.
- Abstração da arquitetura estrutural do agente.
- Auxílio às plataformas de desenvolvimento de SMAs.
- Guia para o desenvolvimento de agentes adaptativos.

Independentemente dos prováveis ganhos, a construção do *framework* permitiu a observação de que é possível criar agentes adaptativos seguindo um ponto de vista diferenciado. Por isso mesmo, os desafios para se obter uma arquitetura mais genérica e desvinculada de plataformas específicas de desenvolvimento de SMAs foram muito maiores.

O único ponto de forte dependência é a utilização de ontologias na descrição das propriedades de contexto.

A implementação do *framework* e a sua execução, com base em um cenário de aplicação, mostraram que a utilização de ontologias e de regras de inferência ainda não é trivial. Atualmente, é impraticável usufruir essa tecnologia sem um bom conhecimento. Questões como a unificação ou atualização de ontologias, bem como a adaptação de regras, geram diversas inconsistências. Isso tudo justifica terem sido usadas ontologias simples na simulação, pretendendo-se desenvolver outras mais complexas no futuro, além de processos mais "inteligentes" para a adaptação de regras.

Quando se fala em processos mais "inteligentes", quer-se dizer processos menos reativos, e a maneira como se definiu a adaptação das regras de inferência é reativa. Os trechos de mudança foram mantidos armazenados nos passos da política até a sua aplicação por meio do ponto de adaptação, e esse mecanismo, para um exemplo mais extenso, pode ser melhorado.

Uma questão importante para se analisar é a opção de como se implementar e utilizar os critérios de escolha das políticas de adaptação, pois eles são um ponto crucial para o funcionamento do *framework*. Trabalhar esse aspecto sem responsabilidade significa incapacitar a execução dos pontos de adaptação. Aumentar os tipos de critérios para a escolha das políticas é um trabalho futuro imprescindível.

Observando-se a adaptação presente na simulação, que é relativamente simples, notou-se que permitir adaptações autônomas mais complexas a agentes pode representar riscos à manutenção de código dos sistemas. Por isso, quando se utiliza um processo adaptativo autônomo, é crítico desenvolver técnicas que mantenham o código de um agente inteligível.

Lembrando-se das partes do *framework* não implementadas, as restrições e as avaliações de adaptação são indispensáveis para o futuro funcionamento – correto e completo – da arquitetura visionada. Atualmente, as partes faltantes ocasionam os seguintes prejuízos:

- Quanto à falta de restrições, é possível ocorrerem adaptações indesejadas, por não existirem limitações que as evitem, podendo incapacitar a execução contínua dos agentes.
- Quanto à falta de avaliações, não é possível distinguir as adaptações vantajosas das irrelevantes, incapacitando o processo de otimização da execução dos agentes.

A extensão dos mecanismos de inferência, além do *Pellet* aliado à API *Jena*, e o aumento das possibilidades de configuração serão bem-vindas. Adicionalmente, poderia ser

estudado o acoplamento do *framework* às plataformas de desenvolvimento de SMAs, como o *SemantiCore*, permitindo avaliar se o uso nativo seria compensatório.

Por fim, acreditando que na posteridade o *framework* estaria funcionando plenamente, seria inovador descobrir uma maneira de se compilar as informações das constantes adaptações, dentro do contexto de execução, e inferir a utilização destas em objetivos diferentes. O significado disso é poder manter um agente seguindo os seus objetivos, mas replicando as suas adaptações aos objetivos de outros agentes, algo semelhante como o que existe nos processos de gestão de conhecimento.

REFERÊNCIAS BIBLIOGRÁFICAS

[AMA05] AMARA-HACHMI, Nejla. **A Framework for Building Adaptive Mobile Agents**. In: Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems, 2005, pp. 1369-1369.

[AMA06] AMARA-HACHMI, Nejla. **An Ontology-based Model for Mobile Agents Adaptation in Pervasive Environments**. IEEE International Conference on Computers Systems and Applications, 2006. pp. 1106-1109.

[ANT04] ANTONIOU, Grigoris; HARMELEN, Frank van. **A Semantic Web Primer**. Massachusetts: The MIT Press, 2004. 238 p.

[API07] OWL API, The. Disponível em: <http://owlapi.sourceforge.net/>. Acesso em: 17 set. 2007.

[BOR97] BORST, W. N. **Construction of Engineering Ontologies for Knowledge Sharing and Reuse**, 1997, 227 f. Tese (PhD) – University of Twente, Enschede, 1997.

[CIC99] CICALESE, Ferdinando; NOLA, Antonio di; LOIA, Vincenzo. **A fuzzy evolutionary framework for adaptive agents**. In: Proceedings of the 1999 ACM symposium on Applied computing, San Antonio, Texas. New York: ACM Press, 1999, pp. 233-237.

[DAC03] DACONTA, M.; OBRST, L.; SMITH, K. **The Semantic Web: A Guide to the Future of XML, Web Services, and Knowledge Management**. Indiana: John Wiley & Sons, 2003. 328 p.

[DEI04] DEITEL, H. M. **Java How to Program, Sixth Edition**. New Jersey: Prentice Hall, 2004. 1568 p.

[ERD03] ERDUR, Riza Cenk; DIKENELLI, Oguz. **A Standards-Based Agent Framework for Instantiating Adaptive Agents**. In: Proceedings of the second international joint conference on Autonomous agents and multiagent systems, Melbourne, Australia. New York: ACM Press, 2003, pp. 984-985.

[ESC06] ESCOBAR, Mauricio; LEMKE, Ana Paula; RIBEIRO, Marcelo Blois. **SemantiCore 2006: Permitindo o Desenvolvimento de Aplicações baseadas em Agentes na Web Semântica**. In: Workshop on Software Engineering for Agent-Oriented Systems (SEAS), Florianópolis, 2006, pp. 72-82.

[FEN05] FENSEL, Dieter Ed. et al. **Spinning the Semantic Web: Bringing the World Wide Web to Its Full Potencial**. Massachusetts: The MIT Press, 2005. 479 p.

[FER99] FERBER, Jacques, **Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence**. Oxford: Addison-Wesley, 1999, 528 p.

[GAM95] GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. **Design Patterns: Elements of Reusable Object Oriented Software.** USA: Addison-Wesley, 1995, 395 p.

[GAR05] GARCIA, Alessandro; CHOREN, Ricardo; LUCENA, Carlos; ROMANOVSKY, Alexander; GIESE, Holger; WEYNS, Danny; HOLVOET, Tom; GIORGINI, Paolo. **Software Engineering for Large-Scale Multi-Agent Systems – SELMAS'05: workshop report**. In: ACM SIGSOFT Software Engineering Notes, vol. 30 (4), ACM Press, 2005, pp. 1-8.

[GRU93] GRUBER, Thomas R. **A translation approach to portable ontology specifications**. Knowledge Acquisition, vol. 5 (2), Jun. 1993, pp. 199-220.

[GRU96] KSL – Thomas R. Gruber. Desenvolvido por Knowledge Systems, AI Laboratory. Disponível em: <http://www.ksl. stanford.edu/kst/what-is-an-ontology.html>. Acesso em: 11 mai. 2006.

[HEW73] HEWITT, Carl; BISHOP, Peter; STEIGER, Richard. **A Universal Modular ACTOR Formalism for Artificial Intelligence**. In: Proceedings of Third International Joint Conferences on Artificial Intelligence, 1973, pp. 235-245.

[HEW77] HEWITT, Carl. **Viewing Control Structures as Patterns of Passing Messages**. In: Journal of Artificial Intelligence, vol. 8 (3), 1977, pp. 323-364.

[HUH98] HUHNS, Michael N, editor. **Readings in Agents**. São Francisco, CA: Morgan Kaufmann, 1998. 523 p.

[IBA98] IBA, Takashi; TAKEFUJI, Yoshiyasu. **Adaptation of Neural Agent in Dynamic Environment: Hybrid System of Genetic Algorithm and Neural Network**. In: Proceedings of the Second International Conference on Knowledge-Based Intelligent Electronic Systems, vol. 3, 1998, pp. 575-584.

[IMA04] IMAM, Ibrahim F. **Adaptive Applications of Intelligent Agents**. Proceedings of the Ninth International Symposium on Computers and Communications 2004, vol. 2, 2004, pp. 7-12.

[JAD07] JADE, Java Agent DEvelopment Framework. Disponível em: <http://jade.tilab .com>. Acesso em: 17 set. 2007.

[JAR05] JARS, Isabelle; KABACHI, Nadia; LAMURE, Michel. **Adaptive Agent's Integration in a New Environment: Interactions as a Source of Learning**. In: Proceedings of the fourth international joint conference on Autonomous

agents and multiagent systems, Netherlands. New York: ACM Press, 2005, pp. 1103-1104.

[JAV07] SUN Developer Network. Disponível em: <http://java.sun.com>. Acesso em: 10 fev. 2007.

[JEN07] JENA. Disponível em: <http://jena.sourceforge.net>. Acesso em: 10 fev. 2007.

[JEN96] JENNING, N. R.; O'HARE, G. M. P. O. **Foundations of distributed artificial intelligence**. New York: J. Wiley, 1996. 576 p.

[JUA03] JUAN, Thomas; STERLING, Leon. **A Meta-model for Intelligent Adaptive Multi-Agent Systems in Open Environments**. In: Proceedings of 2nd International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS), Melbourne, 2003, pp. 1024-1025.

[KIC97] KICZALES, Gregor; LAMPING, John; MENDHEKAR, Anurag; MAEDA, Chris; LOPES, Cristina Videira; LOINGTIER, Jean-Marc; IRWIN, John. **Aspect-Oriented Programming**. In: Proceedings of the European Conference on Object-Oriented Programming, vol. 1241, 1997, pp. 220-242.

[KOI01] KOIVUNEN, Marja-Riitta; MILLER, Eric. Disponível em: <http://www.w3.org/2001/12/semweb-fin/w3csw>. Acesso em: 28 abr. 2006.

[LAI97] LAIRD, John E.; PEARSON, Douglas J.; HUFFMAN, Scott B. **Knowledge-directed Adaptation in Multi-level Agents**. Journal of Intelligent Information Systems, vol. 9 (3), 1997, pp. 261-275.

[LEE95] W3C – People – Tim Berners-Lee. Disponível em: <http://www.w3.org/People/Berners-Lee/FAQ.html>. Acesso em: 20 abr. 2006.

[LEM07] LEMKE, Ana Paula. **Um Framework para a Organização do Conhecimento de Agentes de Software**, 2007, 137 f., Dissertação (Mestrado em Ciência da Computação) – Faculdade de Informática, PUCRS, Porto Alegre, 2007.

[LER03] LERMAN, Kristina; GALSTYAN, Aram. **Agent Memory and Adaptation in Multi-Agent Systems**. In: Proceedings of 2nd International Joint Conference on Autonomous Agents and Multiagent Systems, Melbourne, Australia. New York: ACM Press, 2003, pp. 797-803.

[LER07] LERICHE, Sebastien; ARCANGELI, Jean-Paul. **Flexible Architectures and Agents for Adaptive Autonomic Systems**. In: Proceedings of the Fourth IEEE International Workshop on Engineering of Autonomic and Autonomous Systems, 2007, pp. 99-106.

[MAD07] MADKIT, The Project. Disponível em: <http://www.madkit.org>. Acesso em: 17 set. 2007.

[MAE94] MAES, Pattie. **Modeling Adaptive Autonomous Agents**. Artificial Life, vol. 1 (1-2), 1994, pp. 135-162.

[MES07] UCLA CSD. Desenvolvido por Departamento de Ciência da Computação da Universidade da Califórnia. Disponível em: <http://www.cs.ucla.edu/ST/>. Acesso em: 18 jun. 2007.

[MIL04] MILES, Russel. **AspectJ Cookbook**. Sebastopol: O'Reilly, 2004, 354 p.

[NOY01] NOY, N.; MCGUINNESS, D. **Ontology Development 101: A Guide to Creating Your First Ontology**. Technical Report KSL-01-05 and SMI-2001-880, Stanford Knowledge Systems Laboratory and Stanford Medical Informatics, March 2001.

[OPC04] OPENCYBELE Agent Infrastructure. Disponível em: <http://www.opencybele .org>. Acesso em: 17 set. 2007.

[OWL04] W3C – OWL. Desenvolvido por World Wide Web Consortium. Disponível em: <http://www.w3.org/2004/OWL/>. Acesso em: 12 mai. 2006.

[PEL07] PELLET. Disponível em: <http://pellet.owldl.com/>. Acesso em: 10 fev. 2007.

[PIL05] PILONE, Dan; PITMAN, Neil. **UML 2.0 in a Nutshell**. Sebastopol: O'Reilly, 2005. 234 p.

[RDG04] W3C. Desenvolvido por World Wide Web Consortium. Disponível em: <http://www.w3.org/TR/rdf-syntax-grammar/>. Acesso em: 10 mai. 2006.

[RIB02] RIBEIRO, Marcelo Blois, **Web Life: Uma Arquitetura para a Implementação de Sistemas Multi-Agentes para a Web**, 2002, 204 f. Tese (Doutorado em Informática) – Departamento de Informática, PUCRJ, Rio de Janeiro, 2002.

[RIB04] RIBEIRO, Marcelo Blois; LUCENA, Carlos. **Multi-Agent Systems and the Semantic Web – The SemantiCore Agent-based Abstraction Layer**. In: Proceedings of Sixth International Conference on Enterprise Information Systems ICEIS 2004, Porto: INSTICC, vol. 4, 2004, pp. 263-270.

[RIB07] RIBEIRO, Marcelo Blois; ESCOBAR, Mauricio; CHOREN, Ricardo. **Using Agents and Ontologies for Application Development on the Semantic Web**. Journal of the Brazilian Computer Society, v. 1, 2007, pp. 1-15.

[RUS03] RUSSEL, Stuart J.; NORVIG, Peter. **Artificial Intelligence: A Modern Approach**. 2. ed. Upper Saddle River, NJ: Prentice Hall, 2003. 1080 p.

[RUS96] RUS, Daniela; GRAY, Robert; KOTZ, David. **Autonomous and Adaptive Agents that Gather Information**. In: Proceedings of AAAI '96 International Workshop on Intelligent Adaptive Agents, 1996, pp. 107-116.

[SCH05] SCHOREELS, Cyril; GARIBALDI, Jonathan M. **A Comparison of Adaptive and Static Agents in Equity Market Trading**. Proceedings of the 2005 IEEE/WIC/ACM International Conference on Intelligent Agent Technology, 2005, pp. 393-399.

[SIG07] OWLSight. Disponível em: <http://pellet.owldl.com/ontology-browser/>. Acesso em: 10 fev. 2007.

[SPL03] SPLUNTER, Sander van; WIJNGAARDS, Niek J. E.; BRAZIER, Frances M. T. Structuring Agents for Adaptation. In: KUDENKO, Daniel; KAZAKOV, Dimitar; ALONSO, Eduardo, editors, **Adaptive Agents and Multi-Agent Systems: Adaptation and Multi-Agent Learning**. Berlin: Springer, 2003, pp. 174-186.

[SWO07] SWOOP. Disponível em: <http://code.google.com/p/swoop/>. Acesso em: 17 set. 2007.

[URI94] W3C. Desenvolvido por World Wide Web Consortium. Disponível em: <http://www.w3.org/Addressing/>. Acesso em: 9 mai. 2006.

[VAY05] VAYSSE, Gaëtan; ANDRÉ, Françoise; BUISSON, Jérémy. **Using Aspects for Integrating a Middleware for Dynamic Adaptation**. In: Proceedings of the 1st Workshop on Aspect Oriented Middleware Development, vol. 118, 2005, pp. 1-6.

[VER78] VERMEIJ, G. J. **Biogeography and Adaptation: Patterns of Marine Life**. Massachusetts: Harvard University Press, 1978. 352 p.

[W3C06] W3C. Desenvolvido por World Wide Web Consortium. Disponível em: <http://www.w3.org>. Acesso em: 20 abr. 2006.

[WEI01] WEISS, Gerhard. **Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence**. Massachusetts: The MIT Press, 2001. 619 p.

[WEI95] WEISS, Gerhard. **Adaptation and Learning in Multi-Agent Systems: Some Remarks and a Bibliography**. In: Proceedings of IJCAI'95 Workshop, Montreal, Canada. Alemanha: Springer-Verlag, 1995, pp. 1-21.

[WIK04] WICKRAMASINGHE, L. K.; ALAHAKOON, L. D. **Adaptive Agent Architecture Inspired by Human Behavior**. Proceedings of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology, 2004. pp. 450-453.

[WOO02] WOOLDRIDGE, Michael. **An Introduction to MultiAgent Systems**. Chichester: John Wiley & Sons Ltd., 2002. 348 p.

[XML06] W3C – XML *Schema*. Desenvolvido por World Wide Web Consortium. Disponível em: <http://www.w3.org/XML/Schema>. Acesso em: 11 mai. 2006.

[YAM01] YAMAGUCHI, Tomohiro; MARUKAWA, Ryo. **Interactive Multiagent Reinforcement Learning with Motivation Rules**. Proceedings of the Fourth International Conference on Computational Intelligence and Multimedia Applications, Yokusika City, 2001. pp. 128-132.

APÊNDICE A: DIAGRAMAS DA IMPLEMENTAÇÃO

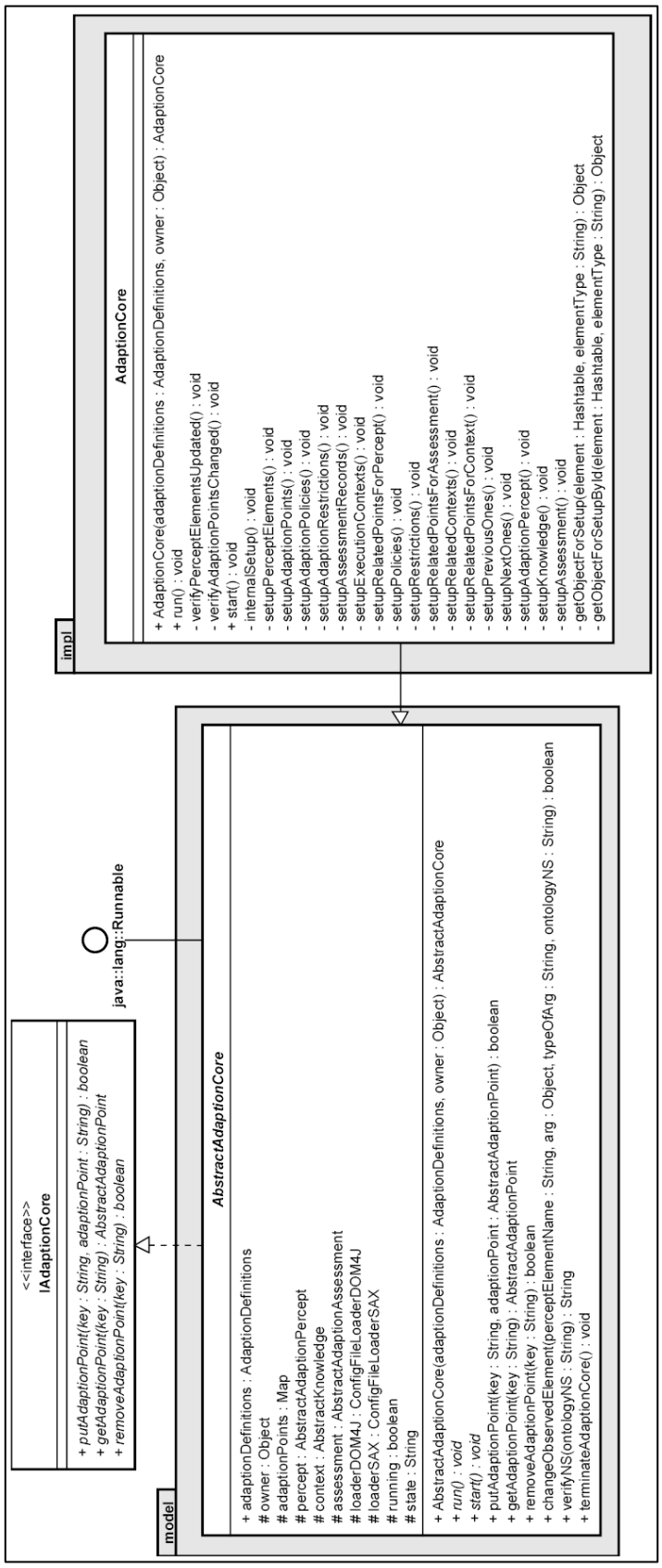

Diagrama do pacote *core*.

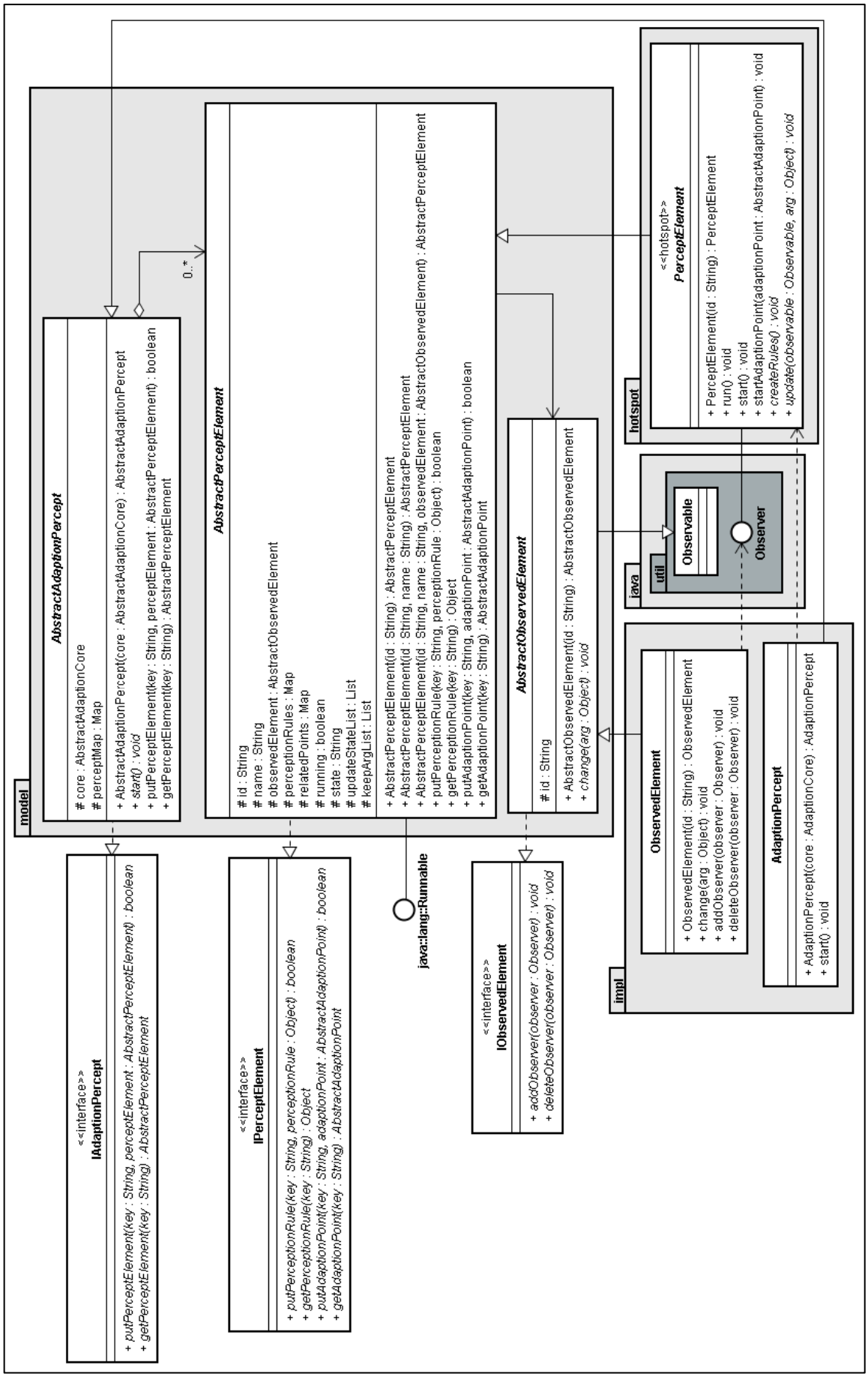

Diagrama do pacote *percept*.

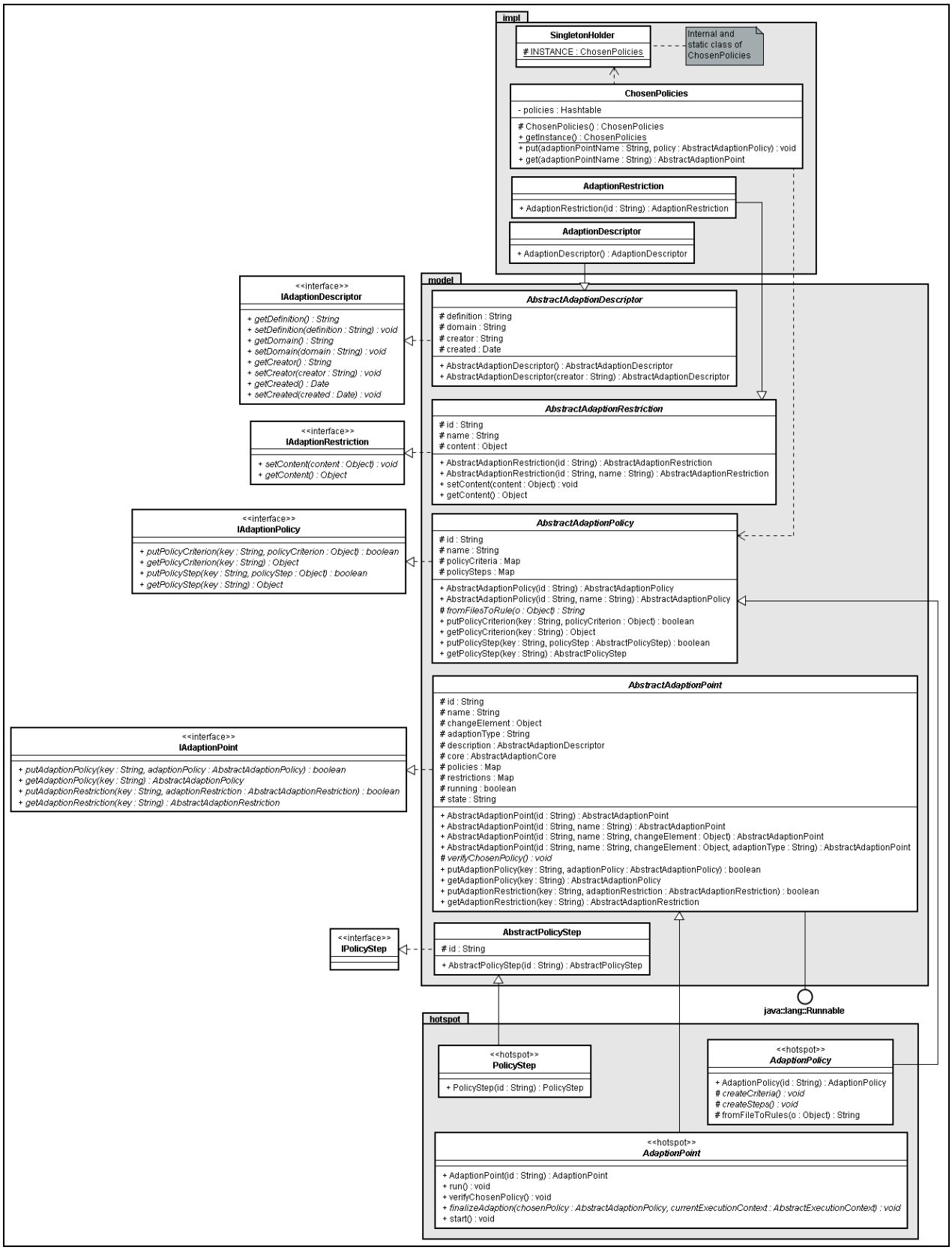

Diagrama do pacote *point*.

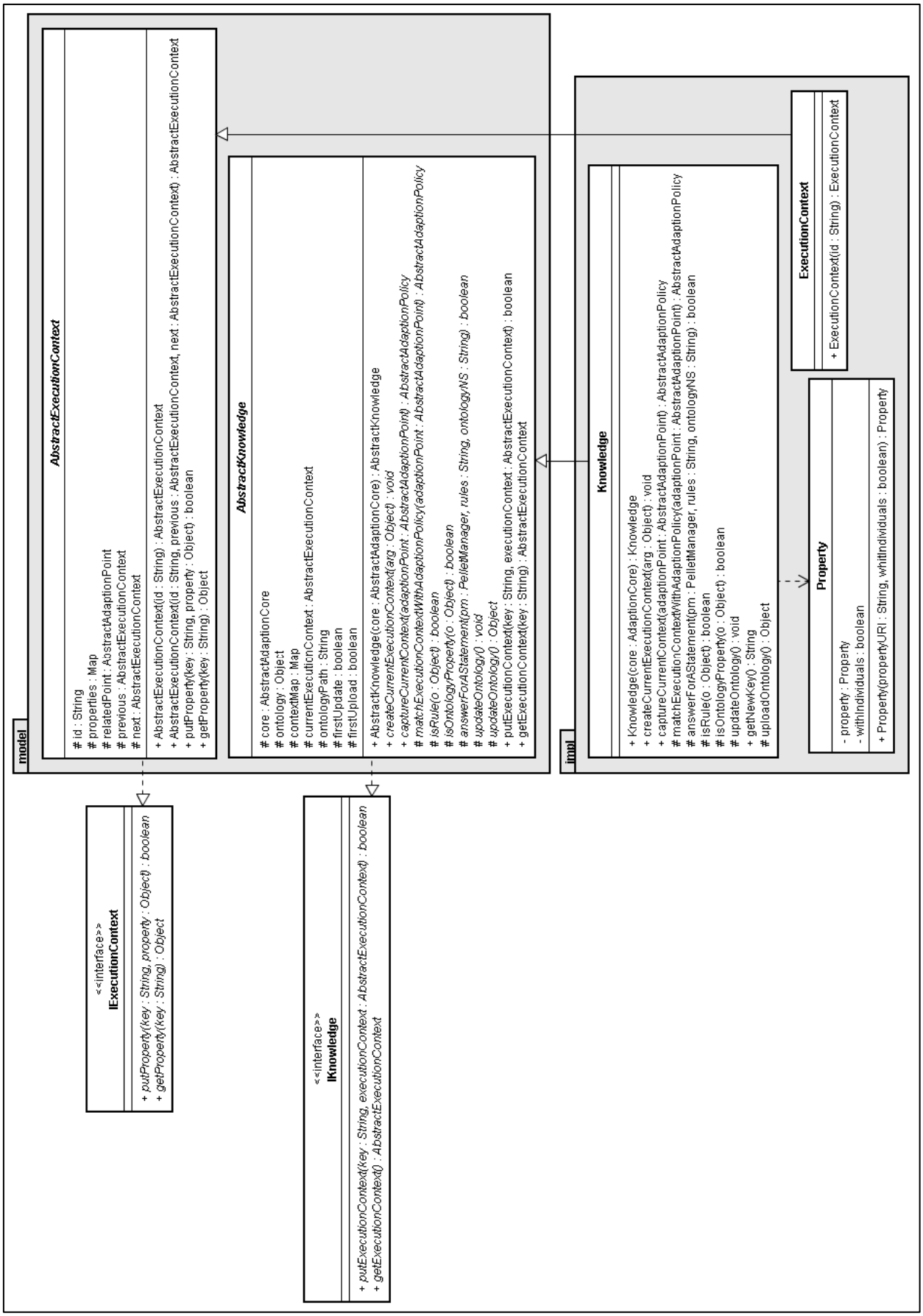

Diagrama do pacote *knowledge*.

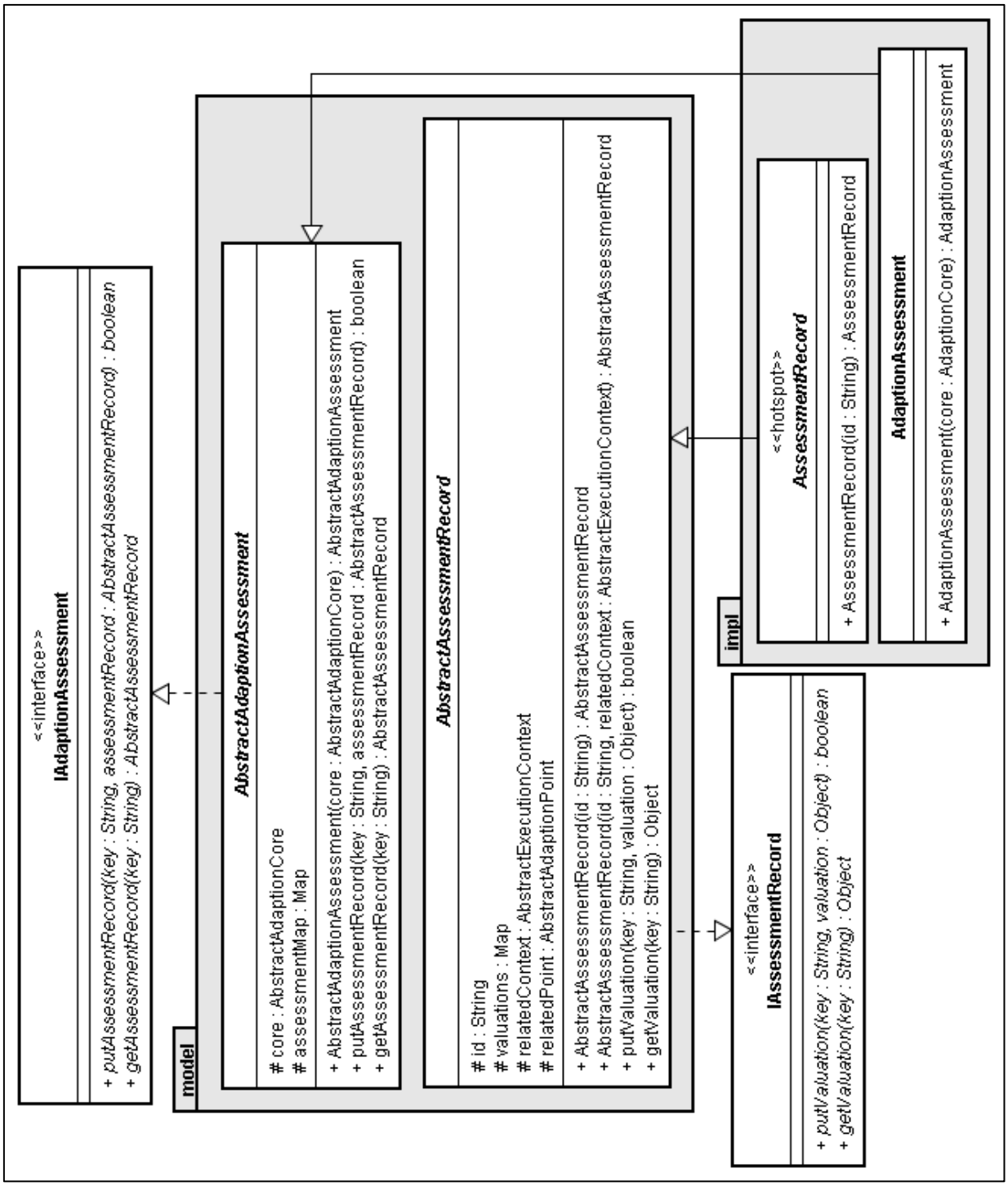

Diagrama do pacote *assessment*.

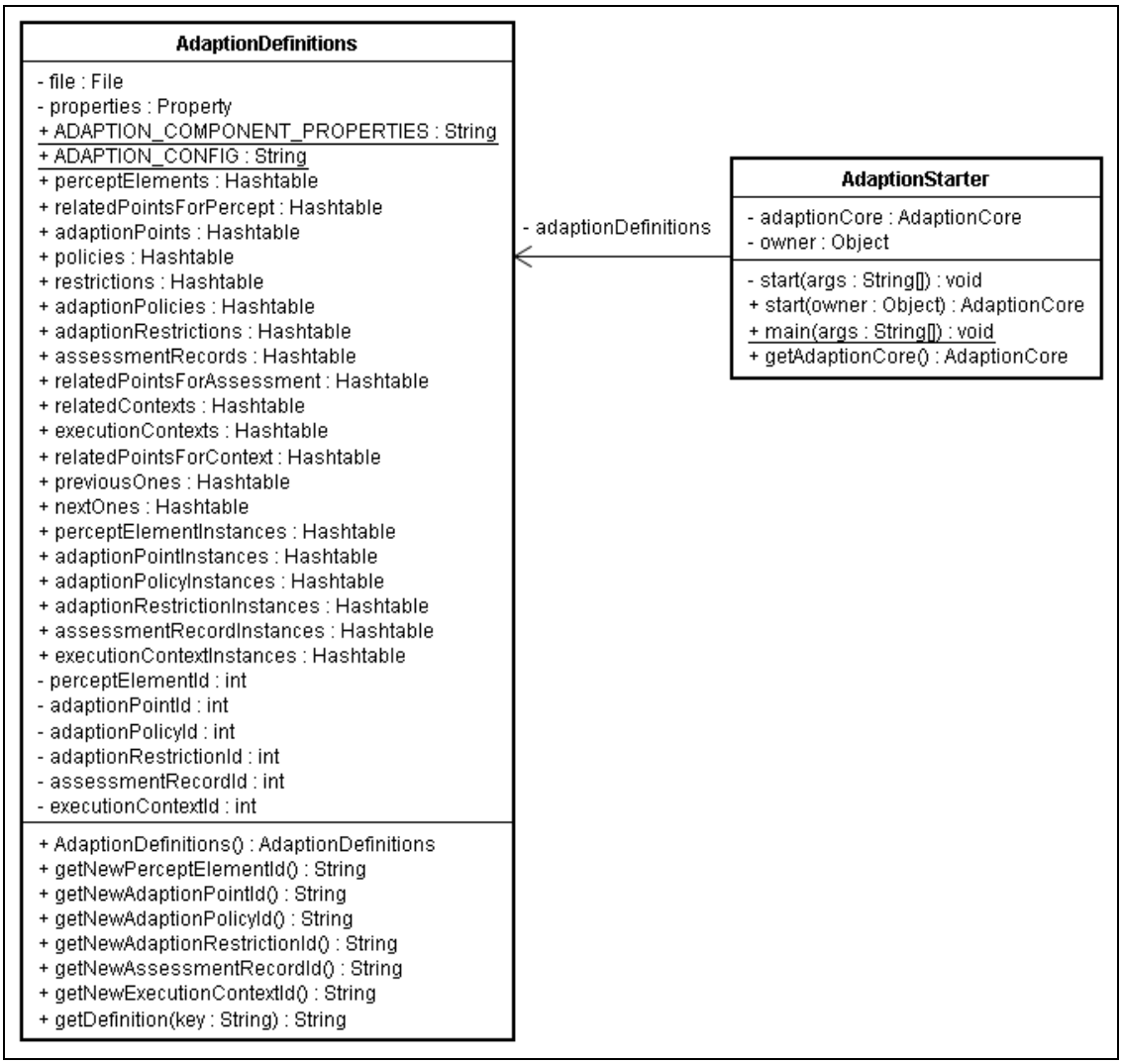

Diagrama do pacote *starter*.

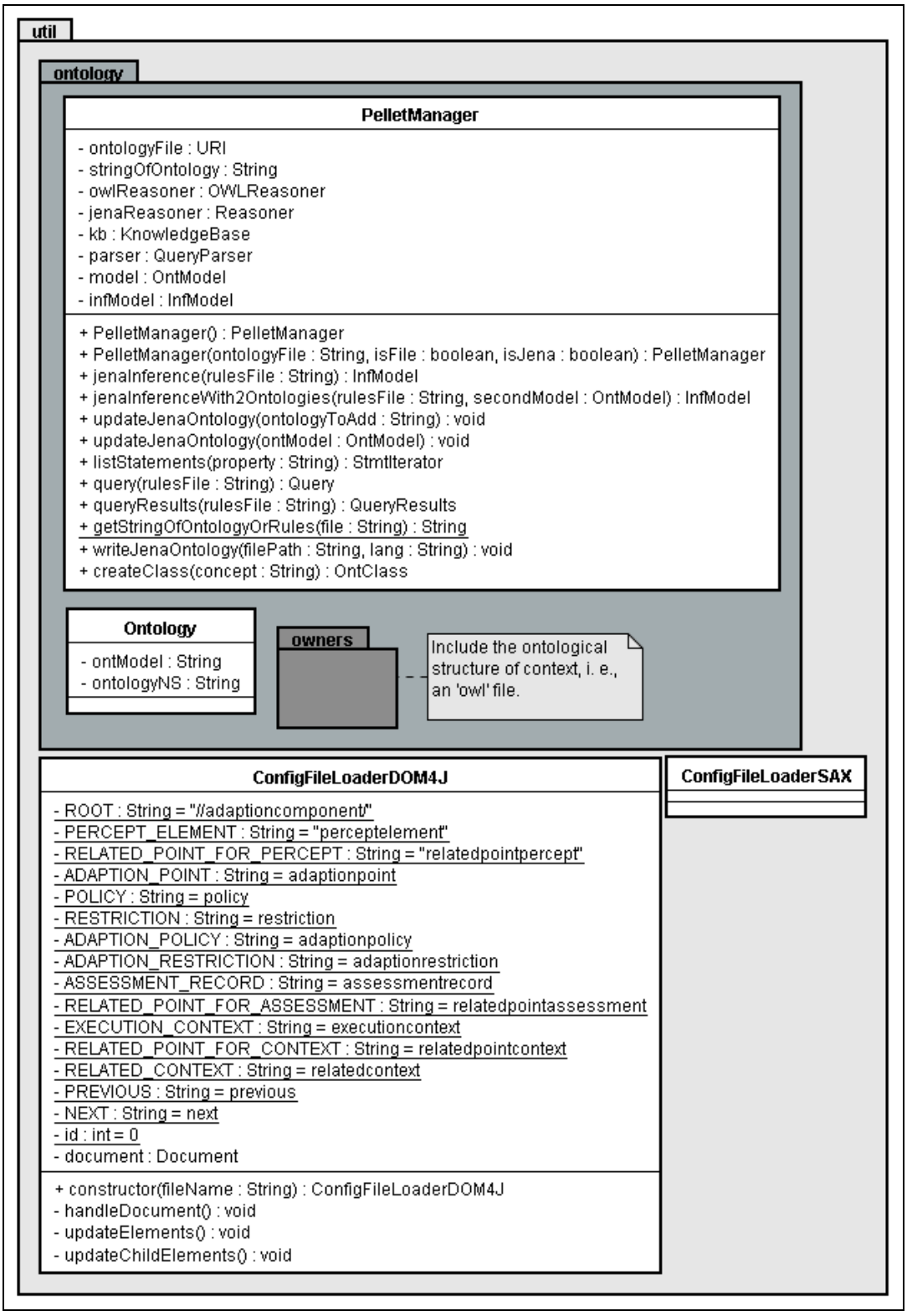

Diagrama do pacote *general*.

Diagrama geral de implementação do *framework* de adaptação.

APÊNDICE B: ESTRUTURAS ONTOLÓGICAS DA IMPLEMENTAÇÃO NA LINGUAGEM OWL

Estrutura Ontológica de Agrupamento dos Contextos de Execução Correntes

```
<?xml version="1.0"?>
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns="http://semanticore.adaption.context.com.br#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xml:base="http://semanticore.adaption.context.com.br">
  <owl:Ontology rdf:about=""/>
  <owl:Class rdf:ID="Property">
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:maxCardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:maxCardinality>
        <owl:onProperty>
          <owl:DatatypeProperty rdf:ID="string"/>
        </owl:onProperty>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:DatatypeProperty rdf:ID="ontologyNS"/>
        </owl:onProperty>
        <owl:maxCardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:maxCardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:DatatypeProperty rdf:ID="ontModel"/>
        </owl:onProperty>
        <owl:maxCardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:maxCardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf rdf:resource="http://www.w3.org/2002/07/owl#Thing"/>
  </owl:Class>
  <owl:Class rdf:ID="ExecutionContext">
    <rdfs:subClassOf rdf:resource="http://www.w3.org/2002/07/owl#Thing"/>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:DatatypeProperty rdf:ID="id"/>
        </owl:onProperty>
        <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:cardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:ObjectProperty rdf:ID="hasProperties"/>
        </owl:onProperty>
        <owl:minCardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
```

```
        >1</owl:minCardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:maxCardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:maxCardinality>
        <owl:onProperty>
          <owl:ObjectProperty rdf:ID="hasNext"/>
        </owl:onProperty>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:ObjectProperty rdf:ID="hasPrevious"/>
        </owl:onProperty>
        <owl:maxCardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:maxCardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:ObjectProperty rdf:about="#hasProperties">
    <rdfs:range rdf:resource="#Property"/>
    <rdfs:domain rdf:resource="#ExecutionContext"/>
  </owl:ObjectProperty>
  <owl:ObjectProperty rdf:about="#hasNext">
    <rdfs:domain rdf:resource="#ExecutionContext"/>
    <rdfs:range rdf:resource="#ExecutionContext"/>
  </owl:ObjectProperty>
  <owl:ObjectProperty rdf:about="#hasPrevious">
    <rdfs:domain rdf:resource="#ExecutionContext"/>
    <rdfs:range rdf:resource="#ExecutionContext"/>
  </owl:ObjectProperty>
  <owl:DatatypeProperty rdf:about="#id">
    <rdfs:domain rdf:resource="#ExecutionContext"/>
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  </owl:DatatypeProperty>
  <owl:DatatypeProperty rdf:about="#ontModel">
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <rdfs:domain rdf:resource="#Property"/>
  </owl:DatatypeProperty>
  <owl:DatatypeProperty rdf:about="#ontologyNS">
    <rdfs:domain rdf:resource="#Property"/>
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  </owl:DatatypeProperty>
  <owl:DatatypeProperty rdf:about="#string">
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <rdfs:domain rdf:resource="#Property"/>
  </owl:DatatypeProperty>
</rdf:RDF>
```

APÊNDICE C: ARQUIVOS DE CONFIGURAÇÃO DO FRAMEWORK

Exemplo do arquivo adaptioncomponent.properties

```
##------## Don't change these properties ##------##

adaption.assessment = adaptioncomponent.assessment.impl.AdaptionAssessment
adaption.core = adaptioncomponent.core.impl.AdaptionCore
adaption.descriptor = adaptioncomponent.point.impl.AdaptionDescriptor
adaption.percept = adaptioncomponent.percept.impl.AdaptionPercept
adaption.point = adaptionpoint
adaption.policy = adaptionpolicy
adaption.restriction = adaptionrestriction
assessment.record = assessmentrecord
execution.context = executioncontext
knowledge = adaptioncomponent.knowledge.impl.Knowledge
percept.element = perceptelement

related.point.for.percept = relatedpoint
policy = policy
restriction = restriction
##------####------####------####------####------####------##

#These properties can be changed

reasoner.type = pellet
no.internal = false
no.owner = false
```

Exemplo do arquivo adaptionconfig.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<adaptioncomponent>

      <perceptelement name="messageerrorpercept"
class="simulation.agent.adaption.percept.MessageErrorPercept" arg="">
            <relatedpointpercept name="comprehendMessage"/>
      </perceptelement>

      <adaptionpoint name="comprehendMessage"
class="simulation.agent.adaption.point.ComprehendMessagePoint" arg="">
            <policy name="messageRecognitionPolicy"/>
      </adaptionpoint>

      <adaptionpolicy name="messageRecognitionPolicy"
class="simulation.agent.adaption.point.MessageRecognitionPolicy" arg=""/>

</adaptioncomponent>
```

APÊNDICE D: ESTRUTURAS ONTOLÓGICAS DA SIMULAÇÃO NA LINGUAGEM OWL

Estrutura Ontológica das Mensagens Trocadas entre os Veículos – Exemplo de Requisição para a Mudança de Pista à Direita

```
<?xml version="1.0"?>
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns="http://semanticore.adaption.traffic.message.com.br#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xml:base="http://semanticore.adaption.traffic.message.com.br">
  <owl:Ontology rdf:about=""/>
  <owl:Class rdf:ID="VehicleCategory">
    <rdfs:subClassOf rdf:resource="http://www.w3.org/2002/07/owl#Thing"/>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:ObjectProperty rdf:ID="isVehicleCategoryOf"/>
        </owl:onProperty>
        <owl:minCardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:minCardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:ID="Vehicle">
    <rdfs:subClassOf rdf:resource="http://www.w3.org/2002/07/owl#Thing"/>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:cardinality>
        <owl:onProperty>
          <owl:ObjectProperty rdf:ID="hasVehicleCategory"/>
        </owl:onProperty>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:ID="Message">
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:cardinality>
        <owl:onProperty>
          <owl:ObjectProperty rdf:ID="hasRequisition"/>
        </owl:onProperty>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf rdf:resource="http://www.w3.org/2002/07/owl#Thing"/>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:cardinality>
        <owl:onProperty>
          <owl:ObjectProperty rdf:ID="messageWasSentBy"/>
        </owl:onProperty>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:ID="MessageRequisition">
```

```
  <rdfs:subClassOf rdf:resource="http://www.w3.org/2002/07/owl#Thing"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:minCardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
      >1</owl:minCardinality>
      <owl:onProperty>
        <owl:ObjectProperty rdf:ID="isRequisitionOf"/>
      </owl:onProperty>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
<owl:ObjectProperty rdf:about="#hasRequisition">
  <rdfs:domain rdf:resource="#Message"/>
  <rdfs:range rdf:resource="#MessageRequisition"/>
  <owl:inverseOf>
    <owl:ObjectProperty rdf:about="#isRequisitionOf"/>
  </owl:inverseOf>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#isVehicleCategoryOf">
  <rdfs:range rdf:resource="#Vehicle"/>
  <rdfs:domain rdf:resource="#VehicleCategory"/>
  <owl:inverseOf>
    <owl:ObjectProperty rdf:about="#hasVehicleCategory"/>
  </owl:inverseOf>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#messageWasSentBy">
  <rdfs:domain rdf:resource="#Message"/>
  <rdfs:range rdf:resource="#Vehicle"/>
  <owl:inverseOf>
    <owl:ObjectProperty rdf:ID="messageSent"/>
  </owl:inverseOf>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#hasVehicleCategory">
  <owl:inverseOf rdf:resource="#isVehicleCategoryOf"/>
  <rdfs:range rdf:resource="#VehicleCategory"/>
  <rdfs:domain rdf:resource="#Vehicle"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#isRequisitionOf">
  <rdfs:range rdf:resource="#Message"/>
  <owl:inverseOf rdf:resource="#hasRequisition"/>
  <rdfs:domain rdf:resource="#MessageRequisition"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#messageSent">
  <rdfs:domain rdf:resource="#Vehicle"/>
  <rdfs:range rdf:resource="#Message"/>
  <owl:inverseOf rdf:resource="#messageWasSentBy"/>
</owl:ObjectProperty>
<owl:DatatypeProperty rdf:ID="type">
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  <rdfs:domain>
    <owl:Class>
      <owl:unionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#MessageRequisition"/>
        <owl:Class rdf:about="#VehicleCategory"/>
      </owl:unionOf>
    </owl:Class>
  </rdfs:domain>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="model">
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  <rdfs:domain rdf:resource="#Vehicle"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="act">
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  <rdfs:domain rdf:resource="#MessageRequisition"/>
```

```
  </owl:DatatypeProperty>
  <owl:DatatypeProperty rdf:ID="chassis">
    <rdfs:domain rdf:resource="#Vehicle"/>
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  </owl:DatatypeProperty>
  <owl:DatatypeProperty rdf:ID="priority">
    <rdfs:domain rdf:resource="#Message"/>
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#int"/>
  </owl:DatatypeProperty>
  <owl:DatatypeProperty rdf:ID="usage">
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <rdfs:domain rdf:resource="#VehicleCategory"/>
  </owl:DatatypeProperty>
  <owl:DatatypeProperty rdf:ID="category">
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <rdfs:domain rdf:resource="#Message"/>
  </owl:DatatypeProperty>
  <owl:DatatypeProperty rdf:ID="tag">
    <rdfs:domain rdf:resource="#Vehicle"/>
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  </owl:DatatypeProperty>
  <Vehicle rdf:ID="ambulance">
    <tag rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >IAT0404</tag>
    <model rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >M4</model>
    <chassis rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >04</chassis>
    <messageSent>
      <Message rdf:ID="message">
        <priority rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</priority>
        <category rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
        >emergency</category>
        <messageWasSentBy rdf:resource="#ambulance"/>
        <hasRequisition>
          <MessageRequisition rdf:ID="messageRequisition">
            <type rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
            >crossing</type>
            <act rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
            >go right</act>
            <isRequisitionOf rdf:resource="#message"/>
          </MessageRequisition>
        </hasRequisition>
      </Message>
    </messageSent>
    <hasVehicleCategory>
      <VehicleCategory rdf:ID="eCategory">
        <isVehicleCategoryOf rdf:resource="#ambulance"/>
        <type rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
        >E</type>
        <usage rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
        >ambulance</usage>
      </VehicleCategory>
    </hasVehicleCategory>
  </Vehicle>
</rdf:RDF>
```

Estrutura Ontológica dos Veículos – Exemplo de Ontologia da Ambulância

```
<?xml version="1.0"?>
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns="http://semanticore.adaption.traffic.ambulance.com.br#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xml:base="http://semanticore.adaption.traffic.ambulance.com.br">
  <owl:Ontology rdf:about=""/>
  <owl:Class rdf:ID="ReceivedMessages"/>
  <owl:Class rdf:ID="Message">
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:cardinality>
        <owl:onProperty>
          <owl:ObjectProperty rdf:ID="hasRequisition"/>
        </owl:onProperty>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf rdf:resource="http://www.w3.org/2002/07/owl#Thing"/>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:ObjectProperty rdf:ID="messageWasSentBy"/>
        </owl:onProperty>
        <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:cardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:ID="MessageRequisition">
    <rdfs:subClassOf rdf:resource="http://www.w3.org/2002/07/owl#Thing"/>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:ObjectProperty rdf:ID="isRequisitionOf"/>
        </owl:onProperty>
        <owl:minCardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:minCardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:ID="Vehicle">
    <rdfs:subClassOf rdf:resource="http://www.w3.org/2002/07/owl#Thing"/>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:cardinality>
        <owl:onProperty>
          <owl:ObjectProperty rdf:ID="hasVehicleCategory"/>
        </owl:onProperty>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:ID="VehicleCategory">
    <rdfs:subClassOf rdf:resource="http://www.w3.org/2002/07/owl#Thing"/>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:ObjectProperty rdf:ID="isVehicleCategoryOf"/>
```

```
      </owl:onProperty>
      <owl:minCardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
      >1</owl:minCardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
<owl:ObjectProperty rdf:ID="isMessageFailedOf">
  <rdfs:range rdf:resource="#ReceivedMessages"/>
  <rdfs:domain rdf:resource="#Message"/>
  <owl:inverseOf>
    <owl:ObjectProperty rdf:ID="hasMessageFailed"/>
  </owl:inverseOf>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="messageSent">
  <rdfs:domain rdf:resource="#Vehicle"/>
  <rdfs:range rdf:resource="#Message"/>
  <owl:inverseOf>
    <owl:ObjectProperty rdf:about="#messageWasSentBy"/>
  </owl:inverseOf>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#messageWasSentBy">
  <rdfs:domain rdf:resource="#Message"/>
  <rdfs:range rdf:resource="#Vehicle"/>
  <owl:inverseOf rdf:resource="#messageSent"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#hasMessageFailed">
  <owl:inverseOf rdf:resource="#isMessageFailedOf"/>
  <rdfs:range rdf:resource="#Message"/>
  <rdfs:domain rdf:resource="#ReceivedMessages"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#hasRequisition">
  <owl:inverseOf>
    <owl:ObjectProperty rdf:about="#isRequisitionOf"/>
  </owl:inverseOf>
  <rdfs:domain rdf:resource="#Message"/>
  <rdfs:range rdf:resource="#MessageRequisition"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="hasMessageAchieved">
  <rdfs:range rdf:resource="#Message"/>
  <rdfs:domain rdf:resource="#ReceivedMessages"/>
  <owl:inverseOf>
    <owl:ObjectProperty rdf:ID="isMessageAchievedOf"/>
  </owl:inverseOf>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="hasMessageUnderstood">
  <rdfs:range rdf:resource="#Message"/>
  <rdfs:domain rdf:resource="#ReceivedMessages"/>
  <owl:inverseOf>
    <owl:ObjectProperty rdf:ID="isMessageUnderstoodOf"/>
  </owl:inverseOf>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#isMessageAchievedOf">
  <owl:inverseOf rdf:resource="#hasMessageAchieved"/>
  <rdfs:range rdf:resource="#ReceivedMessages"/>
  <rdfs:domain rdf:resource="#Message"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#hasVehicleCategory">
  <owl:inverseOf>
    <owl:ObjectProperty rdf:about="#isVehicleCategoryOf"/>
  </owl:inverseOf>
  <rdfs:domain rdf:resource="#Vehicle"/>
  <rdfs:range rdf:resource="#VehicleCategory"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#isVehicleCategoryOf">
  <rdfs:domain rdf:resource="#VehicleCategory"/>
```

```
  <rdfs:range rdf:resource="#Vehicle"/>
  <owl:inverseOf rdf:resource="#hasVehicleCategory"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="isMessageMisunderstoodOf">
  <owl:inverseOf>
    <owl:ObjectProperty rdf:ID="hasMessageMisunderstood"/>
  </owl:inverseOf>
  <rdfs:domain rdf:resource="#Message"/>
  <rdfs:range rdf:resource="#ReceivedMessages"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#hasMessageMisunderstood">
  <rdfs:domain rdf:resource="#ReceivedMessages"/>
  <rdfs:range rdf:resource="#Message"/>
  <owl:inverseOf rdf:resource="#isMessageMisunderstoodOf"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#isMessageUnderstoodOf">
  <rdfs:range rdf:resource="#ReceivedMessages"/>
  <rdfs:domain rdf:resource="#Message"/>
  <owl:inverseOf rdf:resource="#hasMessageUnderstood"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#isRequisitionOf">
  <owl:inverseOf rdf:resource="#hasRequisition"/>
  <rdfs:domain rdf:resource="#MessageRequisition"/>
  <rdfs:range rdf:resource="#Message"/>
</owl:ObjectProperty>
<owl:DatatypeProperty rdf:ID="category">
  <rdfs:domain rdf:resource="#Message"/>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="chassis">
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  <rdfs:domain rdf:resource="#Vehicle"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="priority">
  <rdfs:domain rdf:resource="#Message"/>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#int"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="tag">
  <rdfs:domain rdf:resource="#Vehicle"/>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="usage">
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  <rdfs:domain rdf:resource="#VehicleCategory"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="act">
  <rdfs:domain rdf:resource="#MessageRequisition"/>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="type">
  <rdfs:domain>
    <owl:Class>
      <owl:unionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#MessageRequisition"/>
        <owl:Class rdf:about="#VehicleCategory"/>
      </owl:unionOf>
    </owl:Class>
  </rdfs:domain>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="model">
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  <rdfs:domain rdf:resource="#Vehicle"/>
</owl:DatatypeProperty>
<Vehicle rdf:ID="ambulance">
```

```
    <model rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >M4</model>
    <hasVehicleCategory>
      <VehicleCategory rdf:ID="eCategory">
        <isVehicleCategoryOf rdf:resource="#ambulance"/>
        <usage rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
        >ambulance</usage>
        <type rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
        >E</type>
      </VehicleCategory>
    </hasVehicleCategory>
    <tag rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >IAT0404</tag>
    <chassis rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >04</chassis>
  </Vehicle>
  <ReceivedMessages rdf:ID="receivedMessages"/>
</rdf:RDF>
```

Estrutura Ontológica das Mensagens Advindas do Sensor Físico – Exemplo de Detecção de Movimento à Direita

```
<?xml version="1.0"?>
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns="http://semanticore.adaption.traffic.physicalsensor.com.br#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xml:base="http://semanticore.adaption.traffic.physicalsensor.com.br">
  <owl:Ontology rdf:about=""/>
  <owl:Class rdf:ID="Motion"/>
  <owl:DatatypeProperty rdf:ID="direction">
    <rdfs:domain rdf:resource="#Motion"/>
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  </owl:DatatypeProperty>
  <Motion rdf:ID="motion">
    <direction rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >right</direction>
  </Motion>
</rdf:RDF>
```

APÊNDICE E: REGRAS DE INFERÊNCIA DA SIMULAÇÃO

Regras de Inferência para as Mensagens Trocadas entre os Veículos

```
@prefix owl: <http://www.w3.org/2002/07/owl#>
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
@prefix tf: <http://semanticore.adaption.traffic.message.com.br#>
@prefix ps: <http://semanticore.adaption.traffic.physicalsensor.com.br#>

@include <OWL>.

[rule1:
       (?message rdf:type tf:Message)
       (?message tf:category ?category)
       (?message tf:hasRequisition ?requisition)
       (?requisition tf:type ?type)
       (?requisition tf:act ?act)
       equal (?category, "emergency")
       equal (?type, "crossing")
       equal (?act, "go right")

       -> (?message tf:decision "right")]

[rule2:
       (?message rdf:type tf:Message)
       (?message tf:category ?category)
       (?message tf:hasRequisition ?requisition)
       (?requisition tf:type ?type)
       (?requisition tf:act ?act)
       equal (?category, "emergency")
       equal (?type, "crossing")
       equal (?act, "go left")

       -> (?message tf:decision "left")]
```

A primeira regra (*rule1*) pode ser lida da seguinte forma: quando uma mensagem (*?message*) tem uma categoria (*?category*) e uma requisição (*?requisition*); e essa requisição é de um tipo (*?type*) e possui um ato (*?act*); e a categoria (*?category*) for uma emergência (igual a *emergency*), o tipo (*?type*) for uma ambulância pedindo para ultrapassar (igual a *crossing*), e o ato (*?act*) for uma ação para os carros se manterem na pista da direita (igual a *go right*); então, consegue-se inferir que da mensagem (*?message*) é formada uma decisão (*tf:decision*) para se manter na pista da direita (valor *right*).

A segunda regra (*rule 2*) pode ser lida da seguinte forma: quando uma mensagem (*?message*) tem uma categoria (*?category*) e uma requisição (*?requisition*); e essa requisição é de um tipo (*?type*) e possui um ato (*?act*); e a categoria (*?category*) for uma emergência (igual a *emergency*), o tipo (*?type*) for uma ambulância pedindo para ultrapassar (igual a *crossing*), e o ato (*?act*) for uma ação para os carros se manterem na pista da esquerda (igual a

go left); então, consegue-se inferir que da mensagem (*?message*) é formada uma decisão (*tf:decision*) para se manter na pista da esquerda (valor *left*).

Regra de Inferência para o Segundo Critério de Escolha da Política Presente na Simulação

```
@prefix owl: <http://www.w3.org/2002/07/owl#>
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
@prefix tf: <http://semanticore.adaption.traffic.message.com.br#>

@include <OWL>.

[rule1:
       (?message rdf:type tf:Message)
       (?message tf:category ?category)
       equal (?category, "emergency")
       -> (?message tf:match_result "true")]
```

A única regra (*rule1*) pode ser lida da seguinte forma: quando uma mensagem (*?message*) tem uma categoria (*?category*), e essa categoria for uma emergência (igual a *emergency*); então, consegue-se inferir que da mensagem (*?message*) se chega a um resultado (*tf:match_result*) verdadeiro (valor *true*).

É importantíssimo saber que todas as regras de inferência construídas para serem usadas na escolha de políticas de adaptação, como um tipo de critério "regra de inferência", devem resultar em um recurso chamado *match_result*, retornando uma string com valor *true* ou *false*.

Regras de Inferência Adaptadas pelo Framework

```
@prefix owl: <http://www.w3.org/2002/07/owl#>
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
@prefix tf: <http://semanticore.adaption.traffic.message.com.br#>
@prefix ps: <http://semanticore.adaption.traffic.physicalsensor.com.br#>

@include <OWL>.

[rule1:
       (?message rdf:type tf:Message)
       (?message tf:category ?category)
       (?message tf:hasRequisition ?requisition)
       (?requisition tf:type ?type)
       (?requisition tf:act ?act)
       equal (?category, "emergency")
       equal (?type, "crossing")
       equal (?act, "go right")

       -> (?message tf:decision "right")]
```

```
[rule2:
      (?message rdf:type tf:Message)
      (?message tf:category ?category)
      (?message tf:hasRequisition ?requisition)
      (?requisition tf:type ?type)
      (?requisition tf:act ?act)
      equal (?category, "emergency")
      equal (?type, "crossing")
      equal (?act, "go left")

      -> (?message tf:decision "left")]

[rule3:
      (?message rdf:type tf:Message)
      (?message tf:category ?category)
      (?motion rdf:type ps:Motion)
      (?motion ps:direction ?direction)
      equal (?category, "emergency")
      equal (?direction, "left")

      -> (?message tf:decision "left")]

 [rule4:
      (?message rdf:type tf:Message)
      (?message tf:category ?category)
      (?motion rdf:type ps:Motion)
      (?motion ps:direction ?direction)
      equal (?category, "emergency")
      equal (?direction, "right")

      -> (?message tf:decision "right")]
```

As duas primeiras regras (*rule1* e *rule2*) podem ser lidas exatamente como nas regras de inferência para as mensagens trocadas entre os veículos. A terceira regra (*rule3*) pode ser lida da seguinte forma: quando uma mensagem (*?message*) tem uma categoria (*?category*), e uma informação do sensor físico (*?motion*) indica uma direção (*?direction*); e a categoria (*?category*) for uma emergência (igual a *emergency*), e a direção (*?direction*) indicar que os carros se movem para a pista da esquerda (igual a *left*); então, consegue-se inferir que da mensagem (*?message*) é formada uma decisão (*tf:decision*) para se manter na pista da esquerda (valor *left*).

A quarta regra (*rule4*) pode ser lida da seguinte forma: quando uma mensagem (*?message*) tem uma categoria (*?category*), e uma informação do sensor físico (*?motion*) indica uma direção (*?direction*); e a categoria (*?category*) for uma emergência (igual a *emergency*), e a direção (*?direction*) indicar que os carros se movem para a pista da direita (igual a *right*); então, consegue-se inferir que da mensagem (*?message*) é formada uma decisão (*tf:decision*) para se manter na pista da direita (valor *right*).

APÊNDICE F: ARQUIVOS DE CONFIGURAÇÃO DO SEMANTICORE

Exemplo do arquivo semanticoreconfig.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<semanticore showgui="true">

       <!--agent name="VehicleAgent" class="simulation.agent.VehicleAgent" arg="" /-->
       <agent name="traffic" class="simulation.agent.GuiAgent" arg="" />
       <agent name="ambulance" class="simulation.agent.AmbulanceAgent" arg="" />
       <agent name="car 01" class="simulation.agent.CarAgent" arg="" />
       <agent name="car 02" class="simulation.agent.CarAgent" arg="" />
       <agent name="car 03" class="simulation.agent.CarAgent" arg="" />

</semanticore>
```

O atributo *showgui*, do elemento <*semanticore*>, permite definir se a interface gráfica da plataforma *SemantiCore* fica visível (valor *true*) ou não (valor *false*). O element <*agent*> permite definir os agentes a serem instanciados na plataforma, por meio dos seguintes atributos: *name*, define o nome do agente; *class*, define a classe que implementa o agente; e *arg*, algum argumento para ser passado ao agente.

Exemplo do arquivo semanticoreinstantiation.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<semanticoreinstantiation>
     <decisionengine class="semanticore.agent.decision.hotspots.GenericDecisionEngine"/>
     <executionengine class="semanticore.agent.execution.hotspots.SCWorkflowEngine"/>
</semanticoreinstantiation>
```

O elemento <*decisionengine*> permite definir o mecanismo para o componente decisório. O elemento <*executionengine*> permite definir o mecanismo para o componente executor. O elemento <*effectorengine*>, não presente no exemplo, permite definir o mecanismo para o componente efetuador. Nos três elementos, a definição dos mecanismos se dá por meio do atributo *class*, indicando a classe de implementação.

Exemplo do arquivo semanticore.properties

```
#GUI
semanticore.gui.treeNodeRoot.title = SemantiCore
semanticore.gui.menuFile = File
semanticore.gui.menuItemShutdownSemantiCore = Shutdown SemantiCore
```