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

UM META-MODELO PARA A REPRESENTAÇÃO INTERNA DE AGENTES DE SOFTWARE

DANILO ROSA DOS SANTOS

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)

S237m Santos, Danilo Rosa dos

Um meta-modelo para a representação interna de agentes de software / Danilo Rosa dos Santos. – Porto Alegre, 2008.

166 f.

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

1. Informática. 2. Sistemas Multiagentes. 3. Engenharia de Software. I. Título.

CDD 006.39

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 Meta-modelo para a Representação Interna de Agentes de Software**", apresentada por Danilo Rosa dos Santos, como parte dos requisitos para obtenção do grau de Mestre em Ciência da Computação, Sistemas de Informação, aprovada em 30/01/08 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 15/10/2008, conforme Ata No. 021/08 pela Comissão Coordenadora.

p/ 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, Noeli, e à
memória de meu pai, Flávio.

Danilo Rosa dos Santos

AGRADECIMENTOS

Primeiramente, agradeço ao professor Dr. Marcelo Blois Ribeiro, por me orientar neste trabalho e por auxiliar no desenvolvimento do meu aprendizado.

Aos membros da banca, professores Dr. Ricardo Melo Bastos e prof. Dr. Ricardo Choren Noya, por terem aceitado avaliar este trabalho.

Ao Convênio DELL/PUCRS, por financiar o meu curso de Mestrado.

Aos colegas do CDPe, pela amizade e pelo companheirismo nesses dois últimos anos.

Aos membros do ISEG, por estarem sempre disponíveis para a discussão de assuntos de interesse do grupo.

Aos meus colegas de curso, por terem contribuído para minha formação.

Aos professores, por terem possibilitado um aprendizado de qualidade.

Aos demais funcionários da universidade, pelo suporte prestado nestes anos.

Aos amigos, por terem me apoiado nesta trajetória e pela compreensão dos muitos momentos em que não pude compartilhar meu tempo.

Ao meu irmão Cristiano, pela amizade.

À minha mãe Noeli, pela educação, carinho e amor.

Ao meu pai Flávio, pelo exemplo de vida, de caráter, de amor e de carinho.

RESUMO

Sistemas Multiagentes (SMAs) estão ganhando atenção na área de desenvolvimento de software. [WOO02] afirma que o rápido crescimento desse campo decorre, em grande parte, da crença em que o paradigma de software baseado em agentes é apropriado para a exploração das possibilidades surgidas nos sistemas distribuídos totalmente abertos, como, por exemplo, a Internet. Devido ao crescente interesse na tecnologia de agentes no contexto da engenharia de software, diversas metodologias foram criadas para suportar o desenvolvimento de sistemas orientados a agentes.

A modelagem interna de agentes de software não é comum nas metodologias atualmente disponíveis. Este tipo de modelagem é importante devido à necessidade da representação da estrutura interna de um agente para a sua posterior implementação. Neste trabalho, será proposto um meta-modelo descritivo para a representação interna de agentes de software criado a partir do estudo de metodologias atuais, que possua aplicação de restrições de integridade e capaz de ser traduzido para código fonte por um mapeamento direto com os elementos de linguagem oferecidos por algumas plataformas de implementação orientadas a agentes atualmente disponíveis.

Palavras-chave — Sistemas Multiagentes, modelagem interna de agentes, metodologias orientadas a sistemas multiagentes, meta-modelo, plataformas de implementação orientadas a agentes.

ABSTRACT

Multi-Agent Systems are gaining attention in the software development area. The quick growth of multi-agent systems development relies on the belief that the agent paradigm is appropriate to explore the possibilities offered by open distributed systems such as the Internet [WOO02]. Due to the growing interest in agent technology in the context of the software engineering, many methodologies were created to support agent-oriented systems development.

The agent internal modeling is not common in the multi-agent oriented methodologies currently available. This modeling type is important due to the requirement of agent internal structure representation for the aftermost implementation. In this work, will be proposed a descriptive metamodel for the software agent internal representation created by the study of current methodologies, with application of integrity constraints and able to be translated to source code by a direct mapping with the use of language elements offered by some agent-oriented implementation platforms currently available.

Keywords — Multi-Agent Systems, internal modeling, multi-agent oriented methodologies, metamodel, agent-oriented implementation platforms.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS

AOS – Agent Oriented Software

AUML – Agent-based Unified Modeling Language

BDI – Belief-Desire-Intention

CRC – Class Responsibility Collaboration

DVSL – Declarative Velocity Style Language

EJB – Enterprise JavaBeans

JNI – Java Native Interface

LGPL – GNU Lesser General Public License

LOC – Lines of Code

MaSE – Multiagent Systems Engineering

MAS-ML – Multi-Agent System Modeling Language

MDA – Model Driven Architecture

MSC – Message Sequence Charts

OCL – Object Constraint Language

OCLCUD – OCL Compiler

Octopus – OCL Tool for Precise UML Specifications

OMT – Object Modeling Technique

OOSE – Object-oriented Software Engineering

OWL – Web Ontology Language

POJO – Plan Old Java Object

PRS – Procedural Reasoning System

RDD – Responsibility Driven Design

RUP – Rational Unified Process

SDL – Specification and Description Language

SMA – Sistema Multiagente

TAO – Taming Agents and Objects

USE – UML-based Specification Environment

VTL – Velocity Template Language

SUMÁRIO

1 INTRODUÇÃO

Um agente é um sistema computacional inserido em um ambiente, capaz de atingir os objetivos planejados por meio de ações autônomas nesse ambiente [WOO02]. Segundo [WEI01], sistemas multiagentes permitem manusear agentes coletivamente, com maior facilidade, como uma sociedade. [MÜL96] afirma que um sistema multiagentes consiste em um grupo de agentes que podem ter papéis específicos dentro de uma organização.

Sistemas multiagentes (SMAs) são cada vez mais usados em desenvolvimento de software. [WOO02] afirma que o rápido crescimento desse campo decorre, em grande parte, da crença em que o paradigma de software baseado em agentes é apropriado para a exploração das possibilidades surgidas nos sistemas distribuídos totalmente abertos, como, por exemplo, a Internet. Uma grande vantagem desse tipo de sistema é a possibilidade de descentralização e distribuição na tomada de decisão. [WEI01] afirma que um sistema multiagentes pode ser aplicado a um sistema em que a informação envolvida é necessariamente distribuída e, além disso, destaca que sistemas multiagentes devem prover uma infra-estrutura especificando protocolos de comunicação e interação.

Atualmente o desenvolvimento de agentes de software é mais voltado para a área acadêmica, com pouca aplicação comercial. Isto ocorre porque faltam metodologias que tornem o seu desenvolvimento produtivo. Todavia, existem várias metodologias que visam auxiliar a construção desses sistemas. Algumas abordagens conhecidas atualmente são: *Multi-agent Systems Unified Process (MASUP)* [BAS05], *Tropos* [CAS01], *Multiagent Systems Engineering* (*MaSE*) [DEL99], *Prometheus* [PAD02], *MAS-CommonKADS* [IGL98] e *MAS-ML* [SIL07]. Dentre essas abordagens, apenas a última não se propõe a ser utilizada como uma metodologia, mas sim como uma linguagem de modelagem de SMAs.

De maneira geral, as abordagens orientadas a sistemas multiagentes têm uma grande deficiência na modelagem interna dos agentes de software [SAN06]. Este tipo de modelagem é de grande importância devido à necessidade da representação da estrutura interna e do comportamento de um agente para a sua posterior implementação. Com isso, recomenda-se que as abordagens tenham um suporte claro para essa representação. Assim, neste trabalho é proposto um meta-modelo para a representação interna de agentes de software. Este meta-modelo poderá ser utilizado como ponte para a tradução entre diferentes abordagens de desenvolvimento e plataformas de implementação de SMAs. Nas subseções a seguir, serão

apresentados a questão de pesquisa, o objetivos geral, os objetivos específicos e a metodologia e organização da dissertação.

1.1 Questão de Pesquisa

Devido à lacuna existente nas abordagens atuais para a representação interna de agentes de software, surge a questão de pesquisa que guia este estudo: **"Como representar a estrutura interna de um agente de software em um sistema multiagentes de forma que essa contemple os conceitos tratados pelas metodologias orientadas a agentes e possa ser implementada nas plataformas orientadas a agentes existentes atualmente?"**.

1.2 Objetivo Geral

O objetivo geral deste trabalho é definir um meta-modelo descritivo para a representação interna de agentes de software criado a partir do estudo de metodologias atuais, que possua aplicação de restrições de integridade e capaz de ser traduzido para código fonte por um mapeamento direto com os elementos de linguagem oferecidos por algumas plataformas de implementação orientadas a agentes atualmente disponíveis. Desta maneira, uma grande vantagem no uso do meta-modelo é a independência com metodologias de desenvolvimento e plataformas de implementação, permitindo assim seu uso na tradução de modelos das metodologias para código fonte em diferentes plataformas.

1.3 Objetivos Específicos

Os objetivos específicos deste trabalho são os seguintes:

- Aprofundar o estudo teórico sobre a representação interna de agentes de software.
- Estabelecer um meta-modelo que possibilite a descrição dos conceitos e relacionamentos que compõem um agente de software.
- Propor um processo de consistência de modelos e geração de código.
- Propor um mapeamento entre o meta-modelo e a plataforma de implementação de *SemantiCore* [BLO04].
- Especificar e desenvolver um protótipo para a geração de código na plataforma de implementação *SemantiCore*.
- Avaliar o código gerado.

1.4 Metodologia e Organização da Dissertação

Este trabalho está dividido em quatro partes: embasamento teórico, proposta do meta-modelo, implementação do protótipo e exemplo de uso.

No Capítulo 2 será apresentado o embasamento teórico referente aos agentes de software, contemplando sistemas multiagentes, arquiteturas de agentes e representação interna de agentes. Além disso, será mostrado um estudo de diferentes abordagens de desenvolvimento de SMAs. Ainda no Capítulo 2, será descrito o estudo realizado sobre restrições de integridade e a *Object Constraint Language* (*OCL*) [OCL07]. Concluindo o capítulo, serão apresentadas diferentes plataformas de implementação de SMAs.

No Capítulo 3, será descrito o processo de criação do meta-modelo feito com base no estudo apresentado no capítulo anterior. Este processo contém a proposta inicial do meta-modelo, o refinamento e o detalhamento dos conceitos e dos relacionamentos que o compõem. Após, será descrita a aplicação das restrições de integridade ao meta-modelo. No final do capítulo, será apresentado um estudo para verificar se as conceitos propostos no meta-modelo contemplam os conceitos tratados em diferentes plataformas de implementação de SMAs.

O Capítulo 4 destaca os aspectos relacionados à implementação do meta-modelo. Inicialmente, serão apresentadas as ferramentas utilizadas para o desenvolvimento do protótipo criado para exemplificar o uso do meta-modelo. Em seguida, será descrito o processo de consistência de modelos e geração de código, além do mapeamento realizado entre o meta-modelo e a plataforma de implementação *SemantiCore*. Além disso, será detalhado o desenvolvimento do protótipo, como o mesmo pode ser estendido e a cobertura desse em relação às restrições de integridade aplicadas ao meta-modelo. No final do capítulo, será explicado o padrão de representação dos modelos em *XML* utilizado pelo protótipo e serão apresentadas as funcionalidades do protótipo.

Um exemplo de uso do meta-modelo inspirado no *Tac SCM* [TAC06] será descrito no Capítulo 5. Esse será modelado com o uso do protótipo. Por fim, será apresentada uma análise do código gerado para a plataforma *SemantiCore* em relação ao exemplo modelado.

No Capítulo 6 serão apresentadas as conclusões, assim como os possíveis trabalhos futuros. Por fim, serão descritas as referências bibliográficas e os apêndices utilizados nesse trabalho.

2 BASE TEÓRICA

Nesse capítulo, inicialmente será apresentado o embasamento teórico da área de sistemas multiagentes. Após serão descritas algumas arquiteturas que usam a abstração de agentes inteligentes e serão mostradas diferentes características encontradas na literatura que buscam mapear as funcionalidades internas de agentes de software. Além disso, será apresentado um estudo realizado sobre diferentes abordagens de SMAs. Por fim, será detalhado o estudo feito sobre restrições de integridade e plataformas de implementação de sistemas multiagentes.

2.1 Agentes de Software

[ODE00] afirma que um agente pode ser uma pessoa, uma máquina, uma parte de um software ou uma variedade de outras coisas. Todavia, para Sistemas de Informação essa definição é muito genérica. [SHO97] define um agente como uma entidade de software que funciona de maneira contínua e autônoma em um ambiente específico, também habitado por outros agentes e processos. Na Figura 2.1 é apresentada uma visão abstrata de um agente, ou ainda, o comportamento básico do mesmo. Percebe-se a ação de saída gerada pelo agente, visando à interação com o ambiente. Normalmente, o agente não possui o controle total do ambiente em que participa, mas, uma influência. Sendo assim, ações aparentemente idênticas podem apresentar efeitos completamente diferentes.

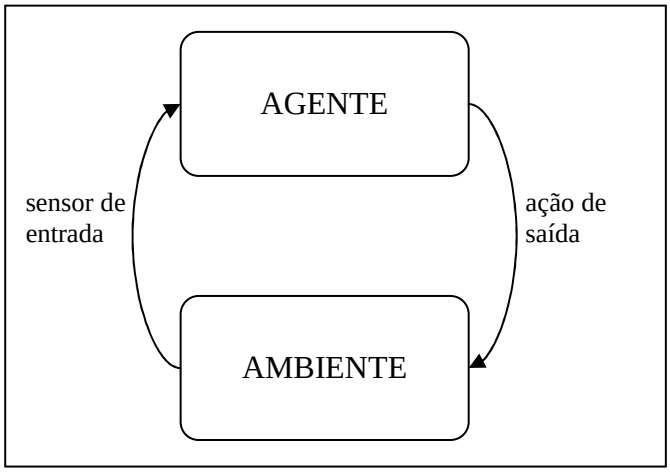


Figura 2.1 – Um agente em seu ambiente [WOO02a].

[ETZ95] e [FRA96] destacam que os agentes de software consistentes com os requisitos de um determinado problema devem possuir os seguintes atributos: reatividade, autonomia, comportamento colaborativo, capacidade de comunicação com outras pessoas e outros agentes com uma linguagem semelhante aos atos de fala, capacidade de inferência, continuidade temporal, personalidade, adaptabilidade e mobilidade. [NWA96] propõe uma tipologia de agentes que identifica outras dimensões de classificação. Nessa tipologia, agentes são classificados por sua mobilidade, pela presença de um modelo de raciocínio simbólico, pela exibição de um ideal e de atributos primários, como autonomia, cooperação e aprendizado, pelos papéis que exercem, por serem híbridos, pelos seus atributos secundários, como versatilidade, benevolência, veracidade, credibilidade, continuidade temporal, possibilidade de falha, capacidades mentais e emocionais.

Dentre as diversas classificações para agentes na literatura, destacam-se cinco. Primeiramente, os agentes podem ser classificados pela sua mobilidade, ou seja, por sua habilidade em se locomover por algum tipo de rede. Assim, são chamados estáticos ou móveis. A segunda classificação qualifica-os como deliberativos ou reativos. No primeiro, os agentes são capazes de se engajar em uma negociação por um pensamento. No segundo, o comportamento do agente depende dos estímulos gerados pelo ambiente onde está inserido. A terceira classificação divide os agentes por atributos que deveriam ser demonstrados pelos mesmos. Três destes atributos são verificados facilmente: autonomia, aprendizado e cooperação. A quarta classificação identifica cada agente pelo seu papel. Por último, o agente pode ser chamado híbrido, quando combina duas ou mais dessas classificações.

No estudo de agentes de software é necessário destacar o conceito de agente inteligente. [GIL95] descreve agentes inteligentes em termos de um espaço definido em três dimensões: agência, inteligência e mobilidade. Onde, agência é o grau de autonomia e autoridade adquirido por um agente, podendo ser medida qualitativamente pela natureza da interação entre agentes e outras entidades do sistema. Inteligência é definida como um grau de raciocínio e comportamento aprendido, ou seja, a capacidade de agentes para aceitar as declarações de objetivos dos usuários e realizar as tarefas delegadas para esses. Por fim, mobilidade é uma característica que permite aos agentes mudarem de local em uma rede. Um agente inteligente deve ser capaz de realizar ações autônomas flexíveis. A flexibilidade significa três fatores:

- Reatividade: capacidade de perceber seu ambiente e reagir em um tempo satisfatório às mudanças, satisfazendo seus objetivos.

- Pró-atividade: aptidão em se comportar com uma tomada de iniciativa própria.
- Habilidade social: capacidade de interação com outros agentes, alcançando seus objetivos.

2.1.1 Sistemas Multiagentes

Sistemas multiagentes podem ser utilizados para a representação de uma sociedade de agentes. Para a criação de sistemas multiagentes na Ciência do Computação, [WOO02] considera cinco tendências importantes: ubiqüidade, interconexão, inteligência, delegação e orientação humana. A ubiqüidade trata da possibilidade de se introduzir poder de processamento em incontáveis locais e dispositivos que, em outros tempos, eram inviáveis economicamente. A interconexão aparece como uma norma na computação comercial e industrial, afirmando a obsolescência de sistemas computacionais isolados. A terceira tendência, a inteligência, traduz a atual capacidade dos sistemas em desenvolver tarefas extremamente complexas. A delegação implica na confiança do homem em repassar o controle de julgamento e execução total dos processos aos sistemas criados. A orientação humana, como última tendência, tenta inserir nas máquinas um entendimento do mundo mais próximo ao modo como um ser humano o percebe.

[JEN96] justifica que um domínio em que é aplicada a tecnologia de sistemas multiagentes 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.

Ainda segundo [JEN96], os ambientes multiagentes devem prover uma infra-estrutura especificando protocolos de interação e comunicação, além disso, possuem as seguintes características: são tipicamente abertos, tem um controle distribuído, possui dados descentralizados, a computação é assíncrona, cada agente tem apenas uma informação incompleta e é restringido pelas suas capacidades, os agentes são autônomos e distribuídos, podendo apresentar interesse próprio ou comportamento cooperativo. [NOR03] afirma que os

ambientes multiagentes são na sua maioria: inacessíveis, não determinísticos, casuais, dinâmicos e contínuos. Inacessível significa não conseguir retirar toda a informação do estado de um ambiente. Não determinístico concretiza a idéia de diferentes efeitos gerados por ações idênticas. Casual define um desempenho do agente independente do número de cenários existentes. Por dinâmico, entende-se um ambiente que sofre mudanças não só por agentes, mas por outros processos. Por último, o ambiente contínuo possui um número indefinido e não fixo de ações e percepções.

Sistemas multiagentes também devem prover protocolos para a interação e comunicação dos agentes. Segundo [WEI01], um protocolo de comunicação pode especificar os seguintes tipos de mensagens para troca entre agentes: propor um curso de ação, aceitar um curso de ação, rejeitar um curso de ação, cancelar um curso de ação, discordar de um curso de ação proposto e contrapropor um curso de ação.

O agente tem, geralmente, um repertório de ações disponíveis capazes de modificar o seu ambiente. Estas ações não são executadas em todas as situações. Além disso, por ter em si pré-condições associadas, apenas as situações possíveis destas associações ocorrem. O problema surge da decisão de quais ações precisam ser executadas para satisfazer, da melhor forma, os objetivos buscados pelo agente. Disso, são introduzidas as arquiteturas de agentes, confirmando o seu uso como sistemas de tomada de decisão embutidos em um ambiente.

2.1.2 Arquiteturas de Agentes

Agentes inteligentes podem ser classificados de diversas formas. Uma das melhores delas é caracterizada por [WOO02], na qual quatro classes de agentes são consideradas conforme a arquitetura interna dos mesmos: agentes baseados em lógica, agentes reativos, agentes de *Belief-Desire-Intention* (*BDI*) e agentes organizados em camadas.

A arquitetura de agentes baseada em lógica diz que a tomada de decisão é realizada pela dedução lógica. Uma representação simbólica do ambiente e do comportamento almejado é usada, havendo, sintaticamente, manipulação dessa representação para a geração de um comportamento inteligente.

A arquitetura reativa possui um mapeamento direto da tomada de decisão. Assim, o agente age conforme o que recebe do ambiente em que está inserido. Um conjunto de comportamentos para a conclusão de tarefas é necessário e vários destes comportamentos podem trabalhar simultaneamente.

A arquitetura de agentes *BDI* se baseia na manipulação de estruturas de dados que representam as crenças, os desejos e as intenções dos agentes, para a tomada de decisão. Tem suas bases na tradição filosófica de entender o raciocínio prático, ou seja, o processo de decidir, momento a momento, qual ação deve ser realizada no amparo das metas.

Finalmente, a arquitetura em camadas apresenta a realização da tomada de decisão por meio de várias camadas de software, estas demonstrando raciocínio, mais ou menos explícito, sobre o ambiente, em diferentes níveis de abstração. Diversos subsistemas de comportamento são organizados em uma hierarquia de camadas que interagem entre si.

2.1.3 Representação Interna de Agentes

[JEN98] descreve um modelo que pode ser utilizado para o projeto de agentes. Este modelo é dividido em três blocos tratados de forma particular, porém interconectados. São eles: capacidades, conhecimento e interface do usuário. Capacidades descrevem o que o agente pode fazer e o que o usuário espera dos agentes, conhecimento é freqüentemente representado por regras utilizadas para um agente tomar uma ação e interface contém as interfaces do usuário e da aplicação.

[RAO95] propõe uma arquitetura *BDI* denominada *INTERRAP*, onde o estado informacional, motivacional e deliberativo de um agente é descrito por desejos, objetivos, planos e intenções. As entradas do agente ou percepções são ligadas às ações por um conjunto de funções que expressam o inter-relacionamento entre as categorias mentais do agente.

Na Figura 2.2 é apresentado o modelo conceitual de um agente *INTERRAP*. Nesse, os modelos mentais de um agente são formados pelos seguintes componentes:

- Percepção corrente do agente.
- Crenças, definindo o estado informacional.
- Situações, descrevendo estruturas relevantes dos desejos do agente.
- Objetivos.
- Opções, representando o estado motivacional.
- Intenções, definindo o estado deliberativo do agente.
- Primitivas operacionais, ligando o estado motivacional do agente ao estado deliberativo.

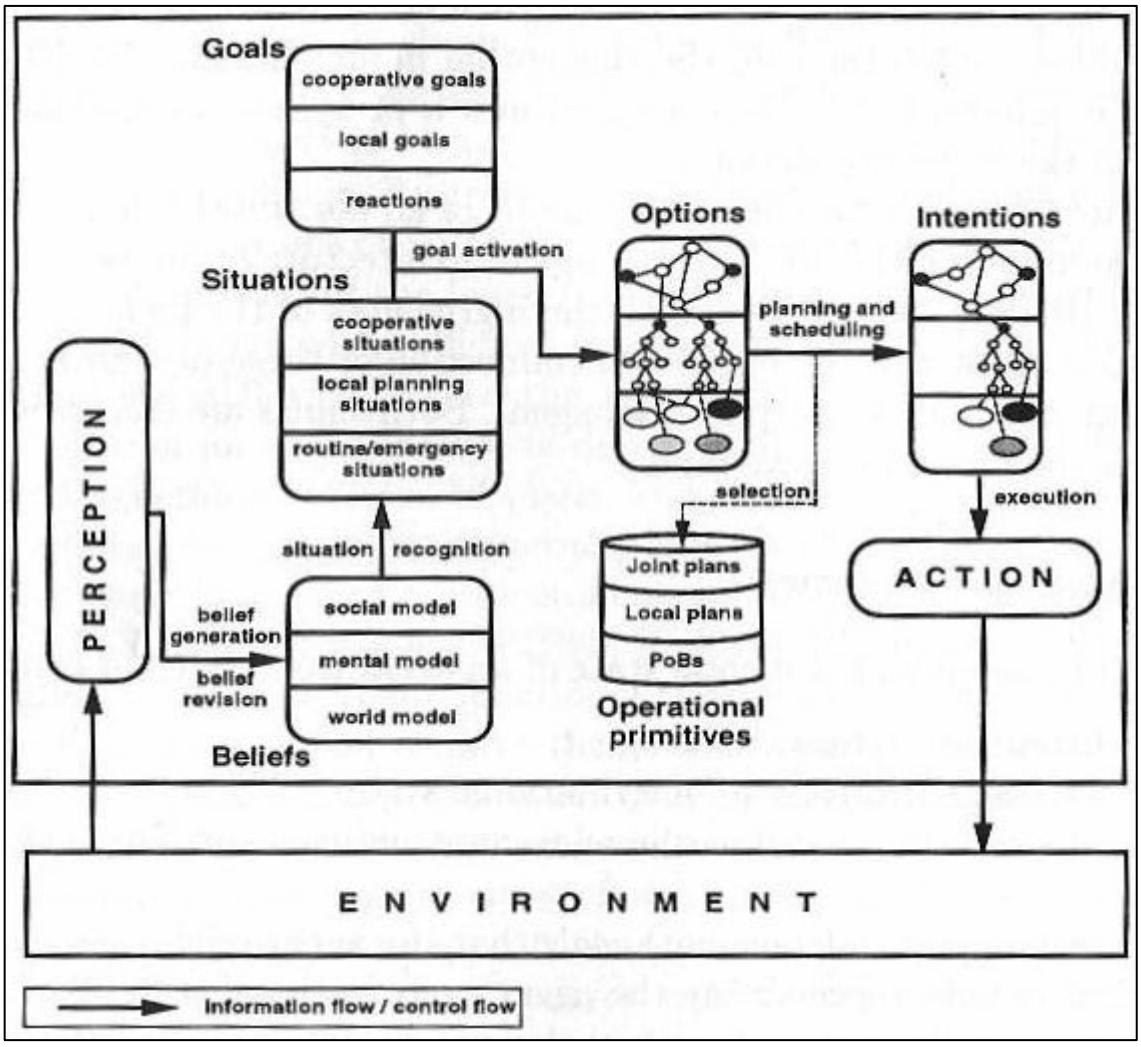


Figura 2.2 – Modelo Conceitual de um Agente *INTERRAP* [MÜL96].

[MÜL96] descreve as setas da Figura 2.2 como relacionamentos funcionais entre os componentes do modelo mental de um agente, definindo o fluxo deste pelo mapeamento de percepções em ações. Estes relacionamentos funcionais podem ser:

- Geração de desejos e revisão de desejos: explica o relacionamento entre os desejos de um agente e a corrente percepção.
- Reconhecimento da situação: extrai as situações (estruturadas) dos desejos (não estruturados) de um agente.
- Ativação de objetivos: descreve quais dos possíveis objetivos de um agente são opções correntes conforme um conjunto de situações.
- Planejamento: mapeia os objetivos correntes de um agente para as primitivas operacionais necessárias para o alcance dos mesmos.

- Agendamento: é o processo de integrar planos parciais referentes a diferentes objetivos em uma agenda de execução.
- Execução: é a implementação dos compromissos determinados nas fases de planejamento e agendamento de forma correta e em um tempo adequado.

[MÜL96] afirma que um agente *BDI* além de ser composto pelos conceitos de desejo, crença e intenção, pode conter objetivos e planos. Os objetivos são classificados como um subconjunto dos desejos e representam as opções que o agente pode seguir. Planos definem um conjunto de ações que o agente pode executar para alcançar seus objetivos. [BRA99] também destaca a importância da especificação e modelagem do comportamento dinâmico dos agentes por meio dos seguintes aspectos: composição de tarefas, troca de informações entre tarefas, seqüência de tarefas, delegação de tarefas e estruturas de conhecimento.

Em [SHO93] foi proposta a idéia de programar sistemas computacionais em termos de estados mentais. A primeira implementação de um paradigma de programação orientado a agentes foi a linguagem de programação *AGENT0*. Nessa linguagem, um agente é especificado em termos de um conjunto de capacidades, um conjunto inicial de desejos, um conjunto inicial de compromissos e um conjunto de regras para esses compromissos. O componente-chave que determina como o agente age é esse último conjunto. Cada regra associada a um compromisso contém uma condição para a mensagem, uma condição para o estado mental e uma ação. Para determinar quando uma regra é disparada, a condição associada à mensagem é atingida por meio das mensagens recebidas pelo agente e a condição associada ao estado mental é alcançada de acordo com as crenças do agente. Caso a regra dispare, o agente se compromete com a ação.

[WEI01] sumariza o processo de raciocínio prático de um agente *BDI* conforme a Figura 2.3. Nessa, existem sete componentes principais que formam este tipo de agente, são eles:

- Um conjunto de crenças, representando a informação que o agente tem do ambiente atual.
- Uma função de revisão da crença, a qual recebe uma entrada perceptiva e as crenças correntes do agente, e com base nisso, determina o novo conjunto de crenças.

- Uma função de geração de opções, que determina as opções disponíveis para o agente (seus desejos), com base nas suas crenças correntes sobre o ambiente e nas suas intenções correntes.
- Um conjunto de opções correntes, representando possíveis cursos de ação disponíveis para o agente.
- Uma função filtro, que representa o processo de deliberação do agente, e que determina as intenções do agente com base nas crenças, desejos e intenções correntes.
- Um conjunto de intenções correntes, representando o foco atual do agente.
- Uma função de seleção de ações (*execute*), que determina a ação que será executada com base nas intenções correntes.

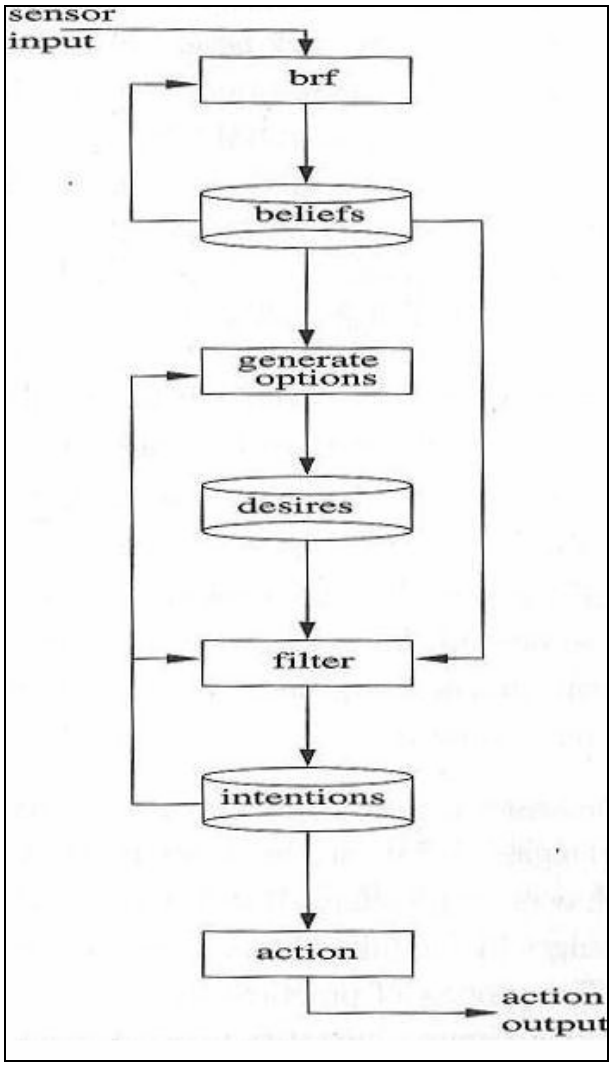


Figura 2.3 – Diagrama esquemático de uma arquitetura *BDI* genérica [WEI01].

[FER99] ressalta que um agente deve possuir um conjunto de ações e percepções, objetivos, recursos, habilidades, serviços, interface de comunicação e uma representação parcial do ambiente.

Nesta seção foram destacadas algumas das diferentes características tratadas na literatura para a formação de um agente. Assim, podem-se destacar algumas dessas, são elas: desejos, objetivos, planos, intenções, percepções, ações, crenças, regras, recursos e interfaces de interação. No próximo item, serão apresentadas algumas abordagens para o desenvolvimento de sistemas multiagentes.

2.2 Abordagens de Desenvolvimento de Sistemas Multiagentes

Nesta seção será apresentada uma visão geral de cada uma das abordagens estudadas. Além disso, para cada abordagem, serão mostrados os diferentes passos e artefatos gerados que almejam a modelagem das características internas de agentes.

2.2.1 MASUP

O *MASUP* nada mais é do que uma extensão do *Rational Unified Process (RUP*) focada em sistemas multiagentes. [BAS04] salienta que o principal objetivo de tal metodologia é identificar sistematicamente a aplicabilidade de uma solução de agentes durante a modelagem.

As fases de Levantamento de Requisitos, Análise e Projeto, são usadas na metodologia *MASUP*, pois a especificação do sistema tem seu início no Levantamento de Requisitos e, além disso, as fases de Implementação, Teste e Implantação tratariam da especificação interna dos agentes, fugindo do escopo inicial.

O processo de Levantamento de Requisitos do *MASUP* é idêntico ao do *RUP*. As diferenças entre as duas metodologias ficam mais claras nas fases de Análise e Projeto. Nessas, são derivados diferentes artefatos para modelar as características específicas dos agentes. Para a modelagem de artefatos, o *MASUP* utiliza diagramas da *Agent-based Unified Modeling Language (AUML)*. A Figura 2.4 apresenta os artefatos gerados em cada fase do *MASUP*.

Um agente no *MASUP* é identificado na fase de Análise e é definido como uma agregação de papéis. A Especificação de uma Classe de Agente é apresentada na Figura 2.5 e possibilita a modelagem de um agente na metodologia. No exemplo, é modelado um agente

Comprador com os atributos Componente, Quantidade, Data e Preço Reservado. Este agente possui duas interfaces de interação: uma contendo um performativo *call for proposal* e outra contendo um performativo *accept-proposal*. Além disso, o agente Comprador exerce um papel Comprador e tem como atribuição a negociação e a aceitação de propostas de compra de componentes.

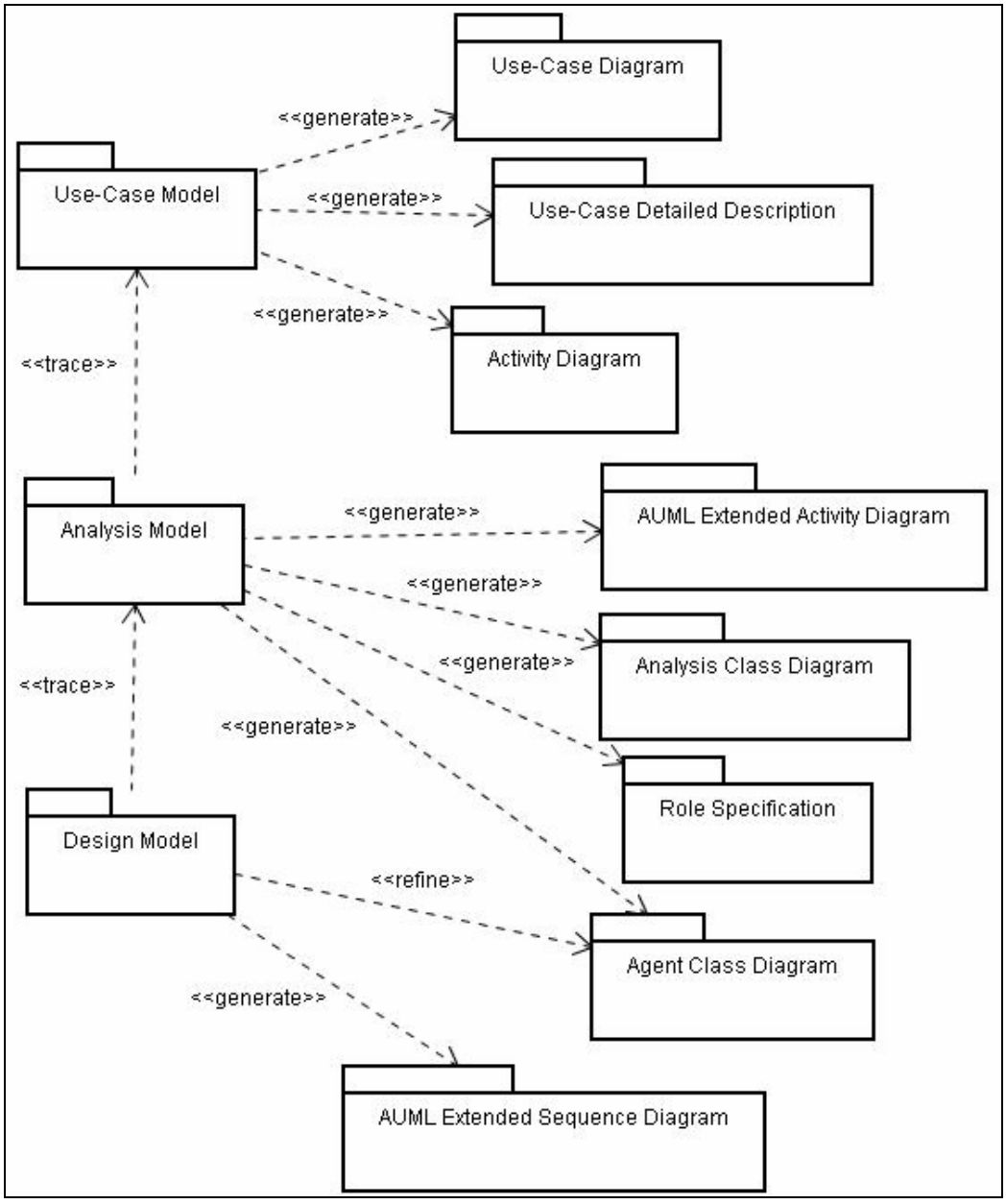


Figura 2.4 – Modelos do *MASUP* e artefatos [BAS05].

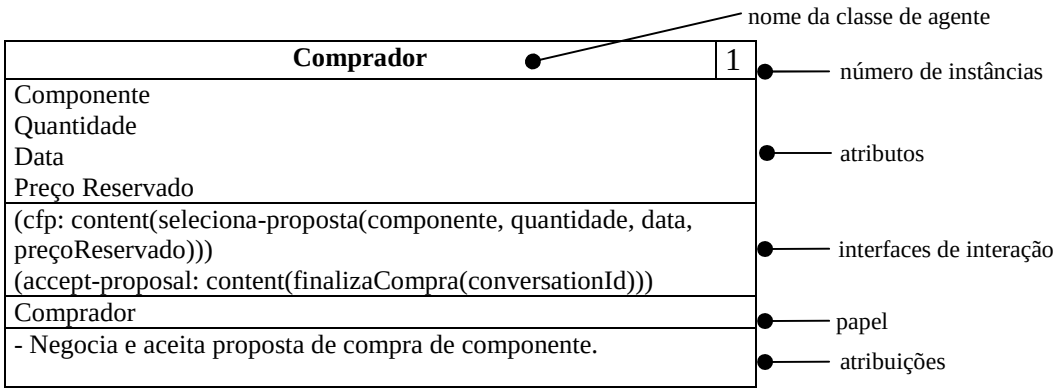


Figura 2.5 – Especificação de uma Classe de Agente.

2.2.2 Tropos

[TRO06] propõe uma metodologia de desenvolvimento de software baseada em conceitos utilizados para a modelagem de requisitos iniciais. O *Tropos* utiliza o *framework i**, proposto por Eric Yu [YU95]. Este *framework* provê noções como atores, objetivos e dependências entre atores que são utilizados durante todo o ciclo de desenvolvimento. A metodologia enfatiza aspectos relacionados às fases iniciais da análise de requisitos, permitindo um melhor entendimento do ambiente onde o software irá operar. [BRE04] afirma que a metodologia *Tropos* é dividida nas fases de Requisitos Iniciais, Requisitos Finais, Projeto Arquitetural, Projeto Detalhado e Implementação. O meta-modelo do *Tropos* [GIO05] apresenta as relações e restrições de cada uma das entidades associadas à metodologia e é apresentado na Figura 2.6.

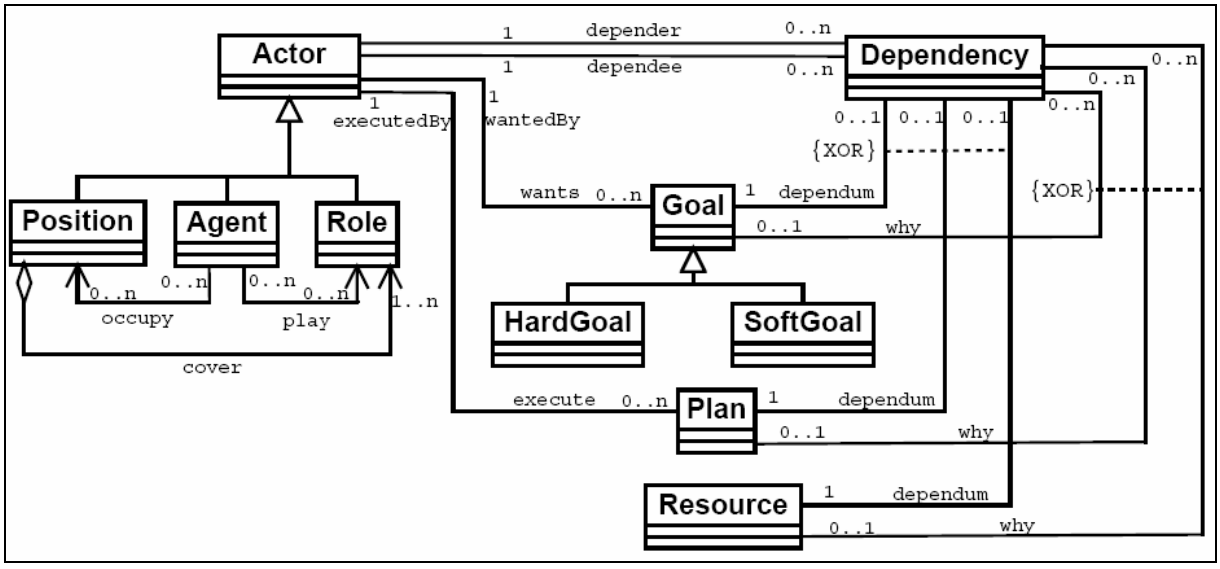


Figura 2.6 – Meta-modelo *Tropos* [GIO05].

No meta-modelo apresentado, nota-se que posição, agente e papel são especializações de ator. Sendo que zero ou mais agentes podem ocupar zero ou mais posições na organização, assim como exercer zero ou mais papéis. Além disso, uma posição cobre um ou mais papéis. No meta-modelo do *Tropos* também existe o conceito de dependência de ator. Esse consiste em um relacionamento contendo *depender*, *dependee* e *dependum*. Sempre que um ator depender de outro no relacionamento esse será denominado *depender***,** o ator responsável pela dependência é conhecido como *dependee* e o relacionamento de dependência entre esses atores se chama *dependum*. Este último pode ser uma meta, tarefa, recurso ou meta-soft. Onde as dependências de metas podem ser expressas como um desejo, as tarefas são vistas como atividades a serem realizadas na organização, um recurso é definido como uma entidade física ou informação e uma meta-soft está relacionada a requisitos não-funcionais.

2.2.3 MaSE

O *Multiagent Systems Engineering (MaSE)* surgiu de esforços realizados por pesquisas do Instituto de Tecnologia da Força Aérea Americana. [DEL01] destaca que o foco principal desta metodologia é auxiliar o projetista a ter um conjunto inicial de requisitos, analisar, projetar e implementar sistemas multiagentes.

O *MaSE* é semelhante às metodologias tradicionais de engenharia de software, porém é orientado para a construção de sistemas multiagentes. A metodologia se divide em duas fases: Análise e Projeto. A primeira é dividida nos seguintes passos: Capturar Objetivos, Aplicar Casos de Uso e Refinar Papéis. Os passos de Criar Classes de Agente, Construir Conversas, Montar Classes de Agente e Projeto do Sistema são parte integrante da fase de Projeto. A Figura 2.7 apresenta uma visão geral das fases da metodologia.

A construção das classes de agente no *MaSE* ocorre no primeiro passo da fase de Projeto. Nesse passo, deve ser criado um Diagrama de Classes de Agente. Além da identificação dos agentes do sistema, esse diagrama deve conter as conversas entre os mesmos. Agentes podem exercer papéis, normalmente em uma relação um para um, assim como na relação entre objetivos e papéis. Porém, fica a critério do projetista a combinação de múltiplos papéis em uma única classe de agente ou vice-versa, onde cada papel deve ter o seu comportamento definido por um conjunto de tarefas. A comunicação entre os papéis do sistema é herdada pelas classes de agente geradas, ou seja, conversas entre papéis se tornam conversas entre classes de agente. Atribuídos todos os papéis, a organização geral do sistema está definida. Para uma organização mais eficiente, [DEL01a] afirma que é desejável

combinar dois papéis que compartilham um alto volume de tráfego de mensagens. Quando determinamos quais papéis devem ser combinados, conceitos como coesão e o volume do tráfego de mensagens são importantes considerações. A Figura 2.8 apresenta um exemplo de um Diagrama de Classes de Agente. Nesse, existem dois agentes: o Agente de Compras e o Agente Fornecedor, exercendo respectivamente os papéis de Gerente de Compras e Fornecedor. O primeiro inicia duas conversas, que são o Envio de Pedidos e a Confirmação de Compra, enquanto que o último inicia apenas a conversa de Envio da Proposta.

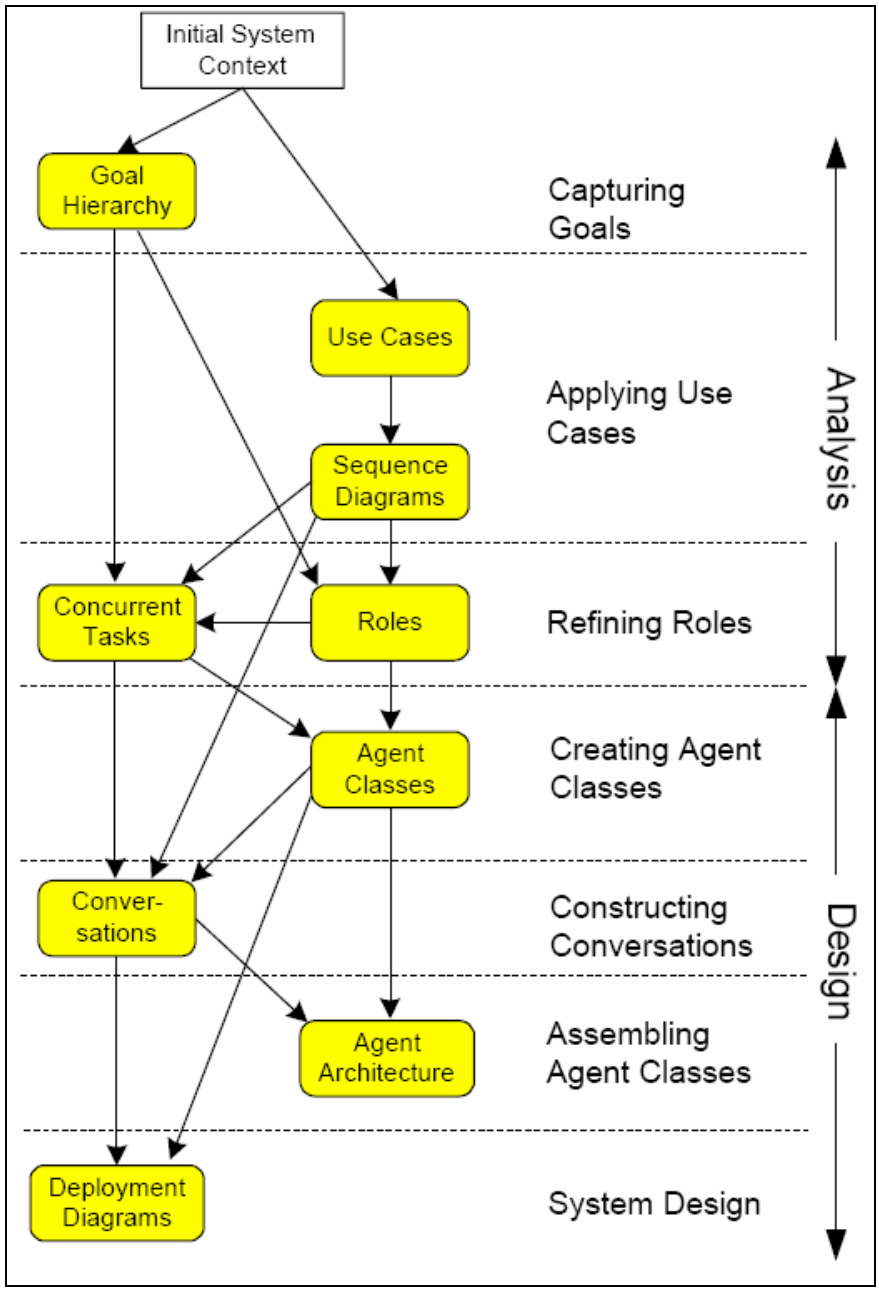


Figura 2.7 – Visão geral das fases da metodologia *MaSE* [DEL01].

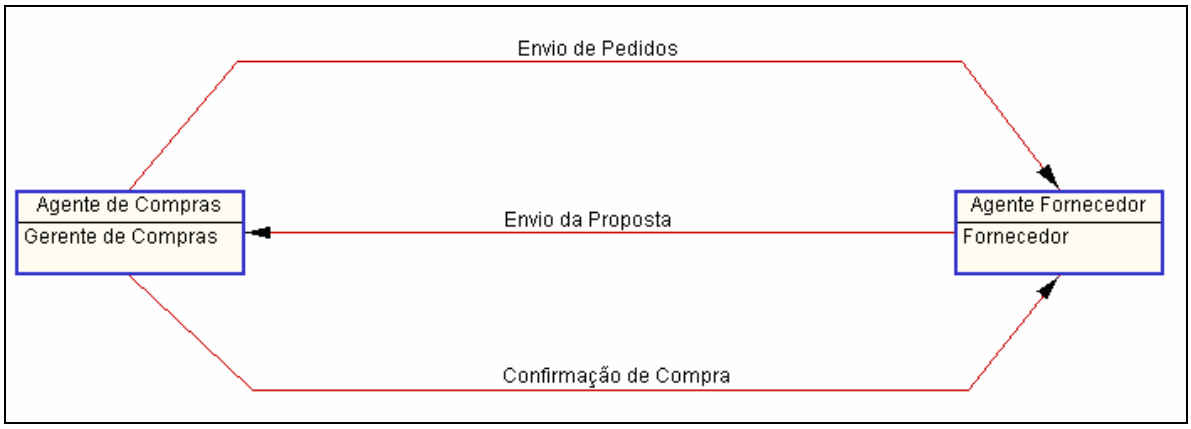


Figura 2.8 – Diagrama de Classes de Agente.

Para a modelagem interna dos agentes, também deve ser considerado o passo Montar Agentes. Nesse, deve ser definida a arquitetura dos agentes do sistema e os componentes internos dos agentes que formarão essa arquitetura. Uma possível solução nesse passo é a derivação da arquitetura de agentes diretamente dos papéis e tarefas definidas no projeto. A Figura 2.9 apresenta um exemplo de arquitetura de agente. Nesse exemplo, um Agente de Compras exerce o papel Gerente de Compras e cada tarefa executada por esse papel se torna um componente do agente detalhado. Com isso, as tarefas Enviar Pedido de Componente, Avaliar Proposta de Compra e Enviar Confirmação de Compra se tornam componente internos do Agente de Compras.

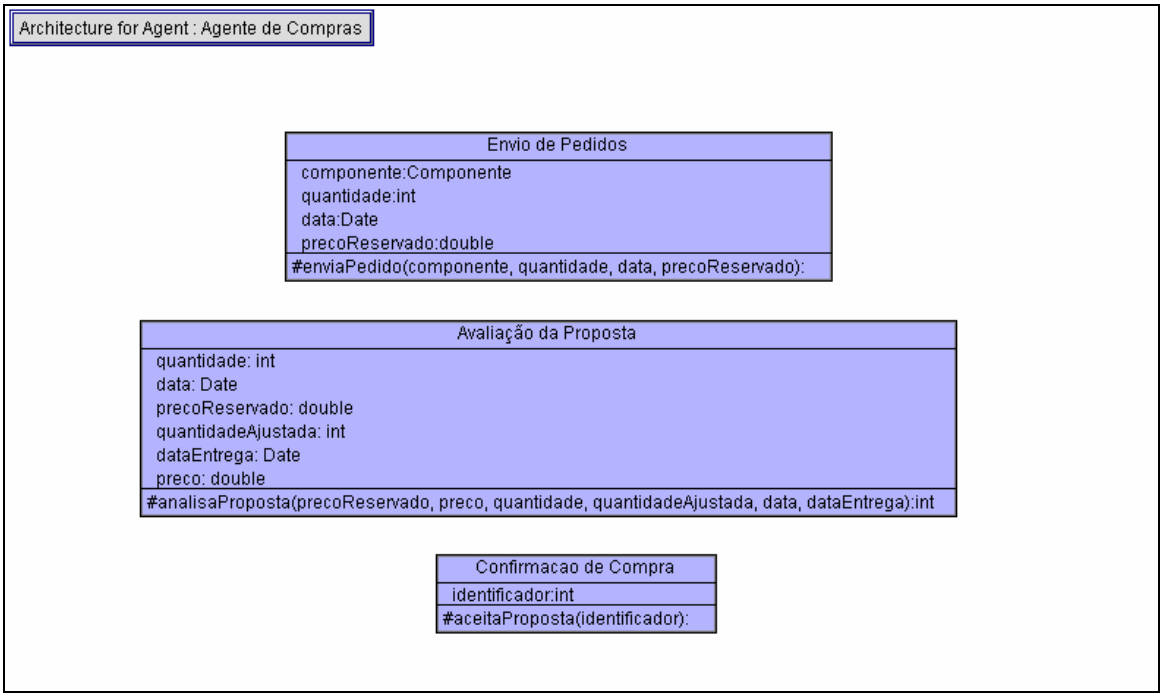


Figura 2.9 – Arquitetura de agente.

2.2.4 Prometheus

[PDT06] afirma que a metodologia *Prometheus* é um processo detalhado para especificar, projetar e implementar sistemas de agentes inteligentes. Essa metodologia foi desenvolvida em colaboração com o *Agent Oriented Software (AOS)* e tem como objetivo poder ser usada tanto por especialistas, quanto por usuários comuns, auxiliados por um processo bem definido. O *Prometheus* foca na construção de sistemas que utilizam agentes *BDI*.

[PAD02] destaca que as principais vantagens do uso dessa metodologia são:

- Auxilia no desenvolvimento de agentes inteligentes que usam objetivos, crenças, planos e eventos.
- Fornece suporte da especificação até o projeto detalhado e a implementação, e disponibiliza um processo detalhado, utilizando artefatos de projeto construídos e passos para derivação de artefatos.
- É orientada para não especialistas em sistemas multiagentes.
- Disponibiliza mecanismos de estruturação hierárquica que permitem a construção do projeto em vários níveis de abstração. Esses mecanismos são fundamentais para a praticidade da metodologia em projetos extensos.
- Utiliza um processo iterativo ao longo das fases de desenvolvimento. Embora a primeira iteração tenha quase todas as suas atividades associadas à fase de especificação do sistema, as iterações subseqüentes envolverão de maneira crescente uma mistura de atividades de diferentes fases.
- Fornece uma checagem automática dos artefatos do projeto.

A metodologia *Prometheus* é dividida em três fases, são elas: Especificação do Sistema, Projeto Arquitetural e Projeto Detalhado. A primeira foca na identificação dos papéis básicos do sistema, com o uso de percepções (entradas), ações (saídas) e compartilhamento de dados. O Projeto Arquitetural recebe como entrada as saídas da fase anterior, além disso, devem ser definidos os agentes participantes do sistema e suas interações. A última fase apóia a modelagem interna dos agentes e define como esses realizarão as tarefas do sistema. A Figura 2.10 mostra uma visão geral das fases do *Prometheus*.

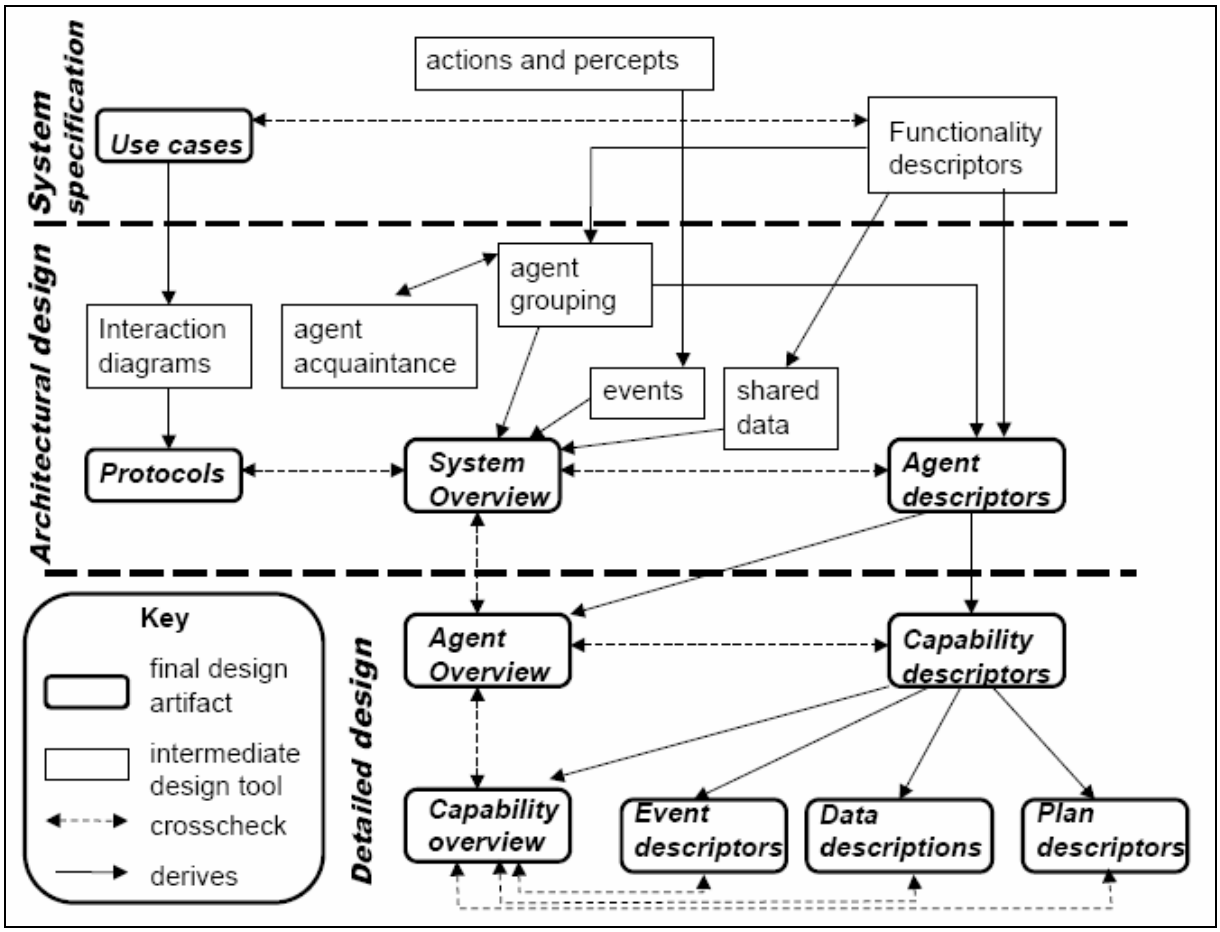


Figura 2.10 – Visão geral das fases do *Prometheus* [PAD02].

O Projeto Detalhado foca no desenvolvimento da estrutura interna de agentes *BDI* e de como esses deverão cumprir suas tarefas dentro do sistema. Nesta etapa, também fica clara a plataforma de implementação a ser utilizada.

A estrutura interna dos agentes é descrita em termos de capacidades. Essas são definidas como eventos internos, planos e estruturas de dados detalhadas. A partir da definição das capacidades, outras vão sendo usadas ou introduzidas, seguindo um processo de refinamento progressivo.

Os papéis da fase de especificação fornecem um bom início do conjunto de capacidades, que podem ser refinadas depois se desejado. As capacidades do sistema são detalhadas em Descritores de Capacidade. Uma visão de alto nível das capacidades internas de um agente é demonstrada no Diagrama de Visão Geral do Agente. Esse também explicita os fluxos de eventos ou tarefas entre essas capacidades, assim como os dados internos do agente. O uso desse diagrama em conjunto com os Descritores de Capacidade possibilita uma visão clara de como os módulos internos do agente interagirão para atingir as tarefas globais do mesmo. Um exemplo de Diagrama de Visão Geral do Agente é apresentado na Figura

2.11. Nesse, o Agente de Compras possui duas capacidades, o Envio de Pedidos e a Confirmação de Compra. A primeira capacidade possui uma ação Enviar Pedido, utiliza os dados de Fornecedores e Pedidos e pode enviar uma mensagem enviaPedido. A capacidade Confirmação de Compra possui uma percepção denominada Proposta Enviada e uma ação Encerrar Compra. Além disso, pode receber uma mensagem enviaProposta e enviar uma mensagem encerraCompra, essa capacidade utiliza os dados de Pedidos e Propostas e gera os dados de Compras.

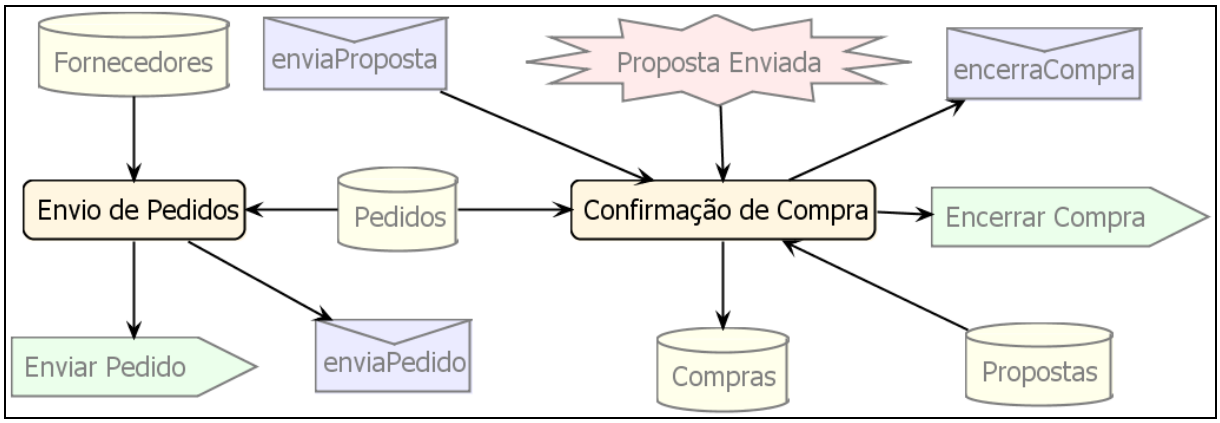


Figura 2.11 – Diagrama de Visão Geral do Agente - Agente de Compras.

Ainda na fase de Projeto Detalhado, deve ser definida uma representação contendo os planos e as conexões entre esses (por meio de eventos), tal representação é denominada Diagrama de Capacidade. Esse tipo de diagrama também permite a representação de capacidades aninhadas e deve estar consistente com o agente e com as demais capacidades definidas.

O Diagrama de Capacidade exemplo é apresentado na Figura 2.12 e mostra que a capacidade Confirmação de Compra possui um plano denominado Selecionar Melhor Proposta. Esse plano possui a percepção Proposta Enviada e a ação Encerrar Compra. O mesmo plano também pode receber a mensagem enviaProposta e enviar a mensagem encerraCompra. Além disso, utiliza os dados de Pedidos e Propostas e gera os dados de Compras.

Ainda no Projeto Detalhado, devem ser definidos o plano individual e os Descritores de Eventos e de Dados. Todos esses dependem diretamente da plataforma de implementação que será escolhida. Quando um evento é aplicável a pelo menos um plano, esse é classificado como coberto, já no caso de ser aplicável a no máximo um plano é denominado como não-

ambíguo. Por fim, a fase de projeto detalhado também pode conter um dicionário de dados, evitando inconsistências no armazenamento.

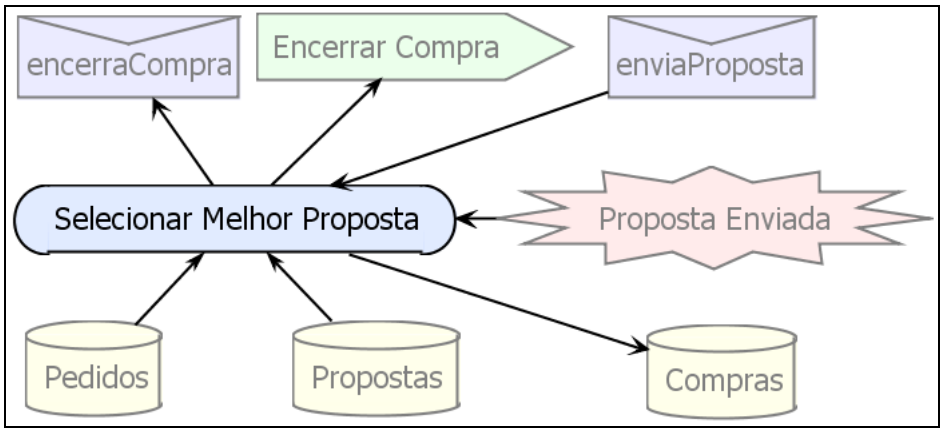


Figura 2.12 – Diagrama de Capacidade - Confirmação de Compra do Agente de Compras.

2.2.5 MAS-ML

[SIL04] afirma que o *Multi-Agent System Modeling Language (MAS-ML)* estende o meta-modelo da *UML* com base no *framework* conceitual *Taming Agents and Objects (TAO)*. O *TAO* define aspectos estáticos e dinâmicos de sistemas multiagentes. O aspecto sintático do *TAO* captura os elementos do sistema e suas propriedades e relacionamentos. Os elementos definidos no *TAO* são agentes, objetos, organizações, ambientes, papéis de agentes e papéis de objetos. Enquanto que os relacionamentos que ligam esses elementos são habitar, exercer, controlar, relação de propriedade, de dependência, de associação, de agregação e de especialização. Os aspectos dinâmicos do *TAO* estão diretamente ligados aos relacionamentos entre os elementos do SMA e definem os comportamentos independentes de domínio associados com a interação entre esses elementos. Por exemplo, a criação e destruição de elementos do SMA e a migração de um agente entre ambientes são descritos como aspectos dinâmicos independentes de domínio de um SMA.

Para prover uma extensão *UML* onde agentes, organizações, ambientes, papéis de agentes e papéis de objetos possam ser representados, novas meta-classes necessitam ser criadas. Na Figura 2.13 é apresentado um dos meta-modelos do *MAS-ML* com a representação dessas novas meta-classes. Estereótipos estendem as capacidades de modelagem pela semântica e não pela estrutura de meta-classes existente.

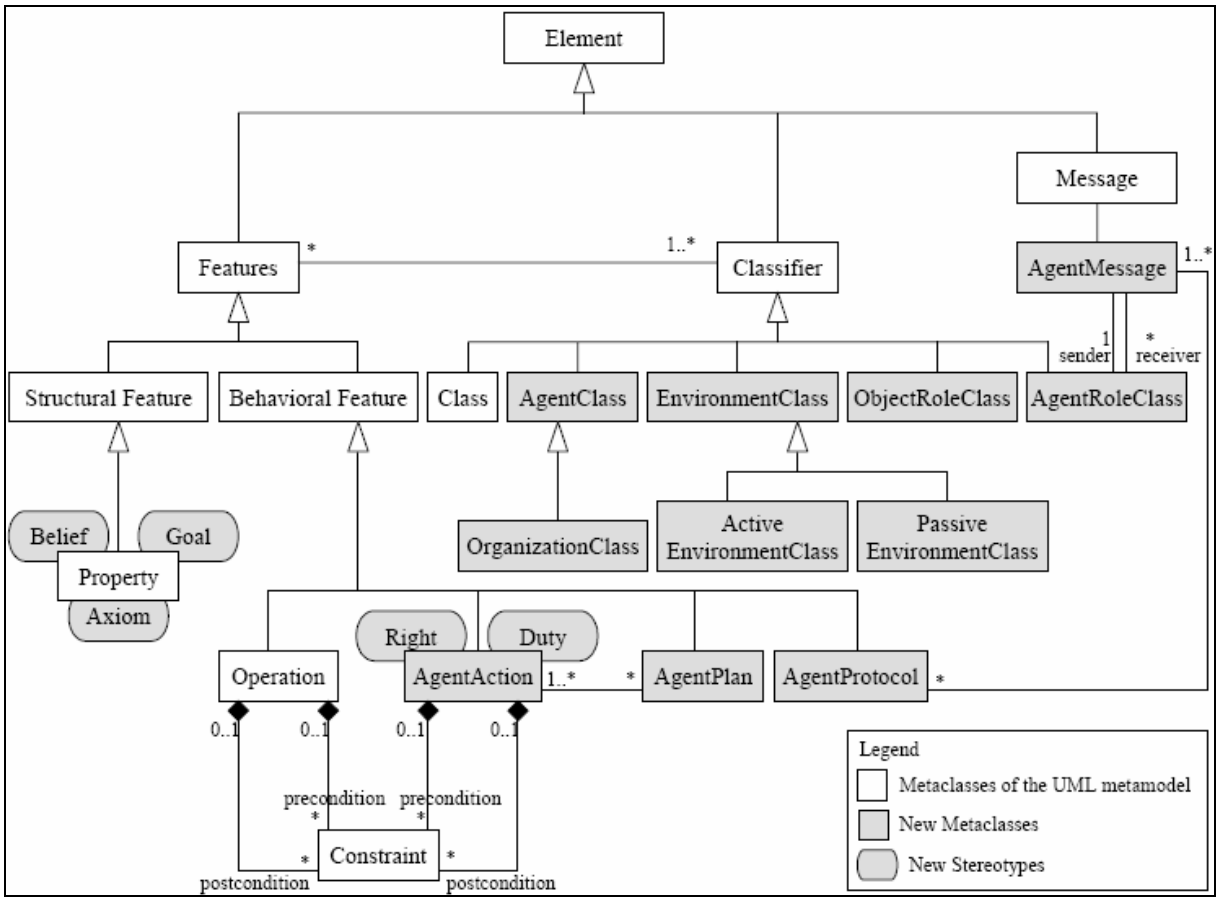


Figura 2.13 – Meta-modelo *TAO/MAS-ML* [SIL04a].

[SIL04a] destaca a criação da meta-classe *AgentClass* para representar agentes. Isso é necessário porque agentes devem ser expressos por meio de componentes mentais como objetivos, crenças, ações e planos. A meta-classe *AgentClass* estende a meta-classe *Classifier*, a qual é associada com as meta-classes *StructuralFeature* e *BehavioralFeature*. Objetivos e crenças são definidos pelos estereótipos *Goal* e *Belief* na meta-classe *Property* e podem ser expressos como atributos de agentes caracterizados por um tipo, um nome e um valor padrão que pode ser alterado durante a execução do agente. Por exemplo, objetivos podem ser modelados usando uma lógica linear em tempo real. Além disso, a lógica modal é utilizada para dar significado a conceitos como crença e conhecimento. A Figura 2.14 apresenta um exemplo de definição de *AgentClass*. Nesse exemplo, o Agente de Compras possui dois objetivos, o envioPedido e o confirmaCompra. O primeiro objetivo é associado ao plano enviando-pedido, enquanto que o último é associado ao plano confirmando-compra. O Agente de Compras têm como crenças um pedido, uma data e um preço. O plano enviando-pedido possui a ação enviarPedido, que por sua vez tem um pedido montado como pré-condição e um pedido enviado como pós-condição. O plano confirmando-compra é composto pela ação

confirmarCompra que possui como pré-condição uma proposta aceita e como pós-condição uma compra confirmada.

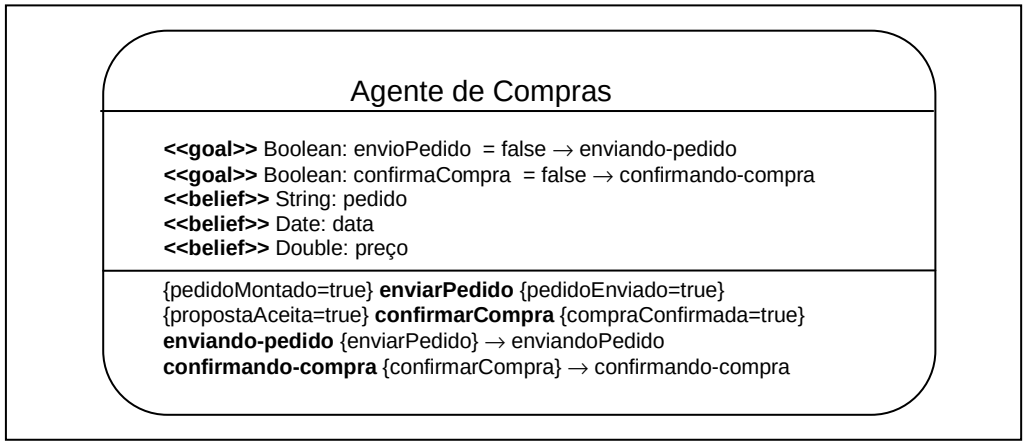


Figura 2.14 – *AgentClass* – Agente de Compras.

Ações são características comportamentais de agentes. Todavia, ações não podem ser definidas como estereótipos baseados na meta-classe *Operation*, pois as definições de ações e operações são diferentes. Uma operação pode ser implementada por um método que pode ser requisitado por outro objeto. Por outro lado, ações nunca são chamadas por outro objeto ou agente, ou seja, são apenas executadas sob o controle do próprio agente. Agentes interagem pelo envio e recebimento de mensagens e não pela chamada da execução de ações. Uma importante consideração, é que a meta-classe *Action* definida em *UML* não é usada para representar as ações dos agentes, pois esta não estende a meta-classe *BehavioralFeature* e não pode ser descrita como uma característica de *Classifier*. Para tal representação deve ser utilizada a meta-classe *AgentAction* que estende a meta-classe *BehavioralFeature*.

Na definição do meta-modelo também foi necessária a criação de uma meta-classe para a especificação dos planos dos agentes em *UML*, esta estende a meta-classe *BehavioralFeature* e é denominada *AgentPlan*. Um plano é associado a um objetivo e é representado por uma seqüência de ações que são executadas por um agente para atingir um objetivo almejado.

O *TAO* também provê o conceito de papéis de agentes. Papéis de agentes são baseados em objetivos, crenças, deveres, direitos e protocolos. Para representar um papel de agente, a meta-classe *AgentRoleClass* foi criada. Deveres e direitos são representados pelos estereótipos *Duty* e *Right* na meta-classe *AgentAction*. Por fim, protocolos são representados pela meta-classe *AgentProtocol*, que é uma extensão da meta-classe *BehavioralFeature*.

2.2.6 MAS-CommonKADS

[HEN05] afirma que as origens do *MAS-CommonKADS* vêm de uma conhecida metodologia de engenharia de conhecimento, o *CommonKADS*, e de metodologias orientadas a objetos como a *Object Modeling Technique (OMT)*, *Object-oriented Software Engineering (OOSE)* e *Responsibility Driven Design (RDD)*. Adicionalmente ela inclui técnicas da engenharia de protocolos como *Specification and Description Language (SDL)* e *Message Sequence Charts (MSC)*. Todas essas técnicas são combinadas com o objetivo de prover suporte aos desenvolvedores de agentes.

MAS-CommonKADS é baseado nos modelos do *CommonKADS* estendido e adaptado para a modelagem de agentes, incluindo a definição de um novo modelo, o modelo de coordenação, para descrever as interações entre agentes.

Segundo [HEN05], o ciclo de vida de desenvolvimento de software do *MAS-CommonKADS* tem as seguintes fases:

- Conceitualização: nesta fase, é obtida uma primeira descrição do problema por meio da definição de um conjunto de casos de uso que ajudam no entendimento do sistema e na tarefa de testar o mesmo.
- Análise: a fase de análise determina os requisitos funcionais do sistema. Assim, o sistema é descrito pelo desenvolvimento de um conjunto de modelos.
- Projeto: a fase de projeto combina abordagens *top-down* e *bottom-up*, reusando componentes desenvolvidos e desenvolvendo novos, dependendo da plataforma de agentes utilizada. A fase de projeto tem como entrada os modelos de análise que são transformados em especificações (modelo de projeto) prontas para serem implementadas. Assim, a arquitetura interna de cada agente e a arquitetura de rede do sistema são determinadas.
- Desenvolvimento e teste: nesta fase, os agentes pré-definidos são codificados e testados.
- Operação: nesta fase ocorre a manutenção e operação do sistema.

A metodologia *MAS-CommonKADS* é composta por diversos modelos, a Figura 2.15 apresenta os modelos *MAS-CommonKADS*, assim como os relacionamentos entre esses.

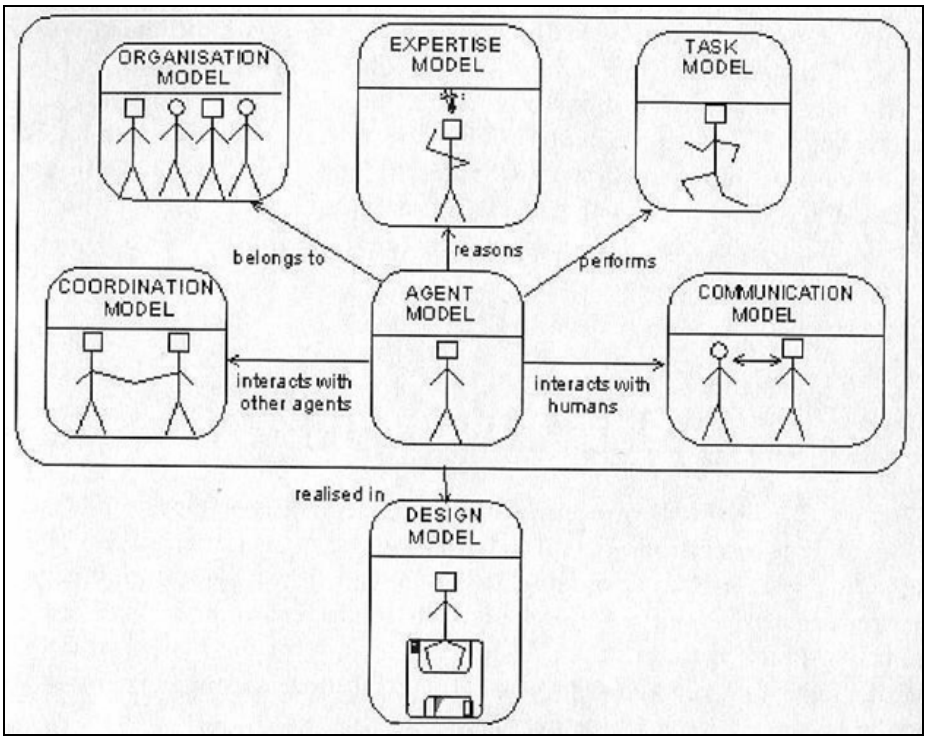


Figura 2.15 – Modelos do *MAS-CommonKADS* [HEN05].

Apresentados os modelos que compõem a metodologia, se faz necessária uma descrição de cada um destes. Sendo assim, estes modelos podem ser caracterizados da seguinte forma:

- Modelo de agentes: especifica as características dos agentes. Essas podem ser capacidades de raciocínio, habilidades (sensores e efetuadores), serviços, grupos de agentes e hierarquias (ambos modelados no modelo da organização).
- Modelo de tarefas: descreve as tarefas que os agentes podem carregar. Essas podem ser objetivos, decomposições, ingredientes, métodos de solução de problemas e assim por diante.
- Modelo de conhecimento especializado: descreve o conhecimento necessário para os agentes atingirem seus objetivos.
- Modelo da organização: descreve a organização em que o SMA será introduzido e a organização da sociedade de agentes.
- Modelo de coordenação: descreve as conversas entre agentes, suas interações, protocolos e capacidades requeridas.
- Modelo de comunicação: detalha as interações entre os agentes humanos e os agentes de software e os fatores humanos para desenvolver essas interfaces de usuário.

- Modelo de projeto: coleta os modelos prévios e é composto por três submodelos. O primeiro submodelo é o projeto de rede, utilizado para projetar os aspectos relevantes da infra-estrutura de rede do agente (rede requerida, conhecimento e facilidades telemáticas); o segundo é o projeto de agente, utilizado para a divisão e a composição dos agentes da análise, de acordo com um critério pragmático e pela seleção da arquitetura mais adequada para cada agente; e o último é o projeto de plataforma, utilizado para selecionar a plataforma de desenvolvimento de agentes para cada arquitetura de agentes.

O modelo de agentes funciona como uma associação entre os demais modelos do *MAS-CommonKADS*, pois coleta as capacidades e restrições dos agentes. *MAS-CommonKADS* propõe diferentes estratégias que podem ser combinadas com o objetivo de identificar os agentes do problema. Algumas destas técnicas são:

- Análise dos atores de casos de uso definidos na fase de conceitualização. Os atores de casos de uso delimitam os agentes externos do sistema. Diversos papéis (atores) similares podem ser mapeados para um agente com o intuito de simplificar a comunicação.
- Análise da declaração do problema. A análise sintática da declaração do problema pode ajudar a identificar alguns agentes. Os agentes candidatos são os sujeitos das sentenças, conhecidos como objetos ativos. As ações carregadas por estes sujeitos devem ser desenvolvidas pelos agentes como objetivos (com iniciativa) ou como serviços (sob demanda).
- Uso de heurísticas. Os agentes podem ser identificados pela determinação de quando existe alguma distância conceitual na distribuição de conhecimento, distribuição geográfica, distribuição lógica ou distribuição organizacional.
- Tarefa inicial ou modelos de conhecimento especializado. Podem ajudar na identificação das funções necessárias e das capacidades de conhecimento requeridas, resultando em uma definição preliminar dos agentes. Os objetivos das tarefas serão atribuídos aos agentes.
- Aplicação da técnica de casos de uso interna.
- Aplicação da técnica refinada de cartões *Class Responsibility Collaboration (CRC)*.

Uma vez identificados os agentes, esses devem ser descritos pelo uso de *templates* textuais que coletam suas características principais. Esses *templates* são compostos por características como: nome, tipo, papel, posição, descrição, serviços oferecidos, objetivos, habilidades (sensores e efetuadores), capacidades de raciocínio, capacidades gerais, normas, preferências e permissões. O processo de aplicar esses *templates* ajuda o engenheiro a rever seu entendimento do problema e serve como um meio de comunicação com o resto do time.

2.3 Restrições de Integridade

Os agentes definidos em cada uma das abordagens apresentadas devem ser descritos pelo uso de modelos. Todavia, esses modelos devem possuir restrições adicionais que garantam a consistência dos mesmos. Um diagrama *UML*, por exemplo, normalmente não é refinado o suficiente para prover todos os aspectos relevantes de uma aplicação [OCL07]. As restrições aplicadas em modelos também são descritas em linguagem natural e a prática tem mostrado que isto quase sempre resulta em ambigüidades. Com o objetivo de escrever restrições não ambíguas, foram desenvolvidas as linguagens formais. A desvantagem do uso de linguagens formais tradicionais é que essas são usadas por pessoas com sólidos conhecimentos matemáticos, dificultando o seu uso por um modelador de sistema. Da necessidade de formalizar restrições em modelos surgiu a *Object Constraint Language (OCL)* [OCL07]. [PEN03] afirma que a *OCL* é uma linguagem formal utilizada para especificar restrições em atributos e associações e, de fato, em qualquer categoria de elementos *UML*.

2.3.1 Object Constraint Language (OCL)

[OCL07] destaca que a *OCL* é uma linguagem formal usada para descrever expressões em modelos *UML*. Essas expressões tipicamente especificam condições que devem ser mantidas para o sistema modelado ou definem consultas aplicadas aos objetos descritos no modelo. Uma expressão *OCL* quando avaliada não traz efeitos colaterais.

Uma expressão ou restrição *OCL* pode ser especificada como uma invariante, uma pré-condição ou uma pós-condição [PEN03]. Uma invariante define um estado que deve ser mantido como verdadeiro por todo o ciclo de vida do objeto. Uma pré-condição define o estado que o sistema deve assumir antes que a ação especificada seja realizada. Por fim, uma pós-condição define o estado do sistema que o objeto deve executar assim que a ação esteja completa.

Cada restrição definida em *OCL* deve estar ligada a um contexto de um modelo. Este contexto pode ser uma classe de objetos ou mesmo uma operação. Para a representação de um contexto em *OCL* é utilizada a palavra reservada *context* e a instância de um contexto pode ser referenciada com a palavra reservada *self*. Também é importante salientar que toda a informação utilizada nas expressões construídas em *OCL* assim como o resultado dessas expressões deve ser de um tipo de dado *OCL* definido na Biblioteca Padrão OCL.

2.4 Plataformas de Implementação

Nesta seção, serão apresentadas as diferentes plataformas de implementação de SMAs estudadas. Esse estudo foi realizado com o objetivo de verificar se os conceitos internos de agentes tratadas nas plataformas são cobertos pelo meta-modelo proposto.

2.4.1 SemantiCore

[BLO04] define o *SemantiCore* como um *framework* que fornece uma camada de abstração sobre serviços de distribuição e uma definição interna de agente capaz de oferecer aos desenvolvedores uma abstração de alto nível para a construção de SMAs. Este *framework* é dividido em dois modelos: o modelo do agente (*SemanticAgent*) e o modelo do domínio semântico. Os dois modelos dispõem de pontos de flexibilidade (*hotspots*) permitindo aos desenvolvedores associar diferentes padrões, protocolos e tecnologias.

O modelo do agente possui uma estrutura orientada a componentes, onde cada componente contribui para uma parte essencial do funcionamento do agente, agregando todos os aspectos necessários a sua implementação. Com a retirada de um ou mais componentes não relacionados ao desempenho das tarefas do agente é possível simplificar a sua arquitetura. A Figura 2.16 apresenta o modelo do agente do *SemantiCore* com seus quatro componentes básicos: o sensorial, o decisório, o executor e o efetuador.

Para perceber e capturar os recursos que trafegam pelo ambiente, o agente possui o componente sensorial. Esse componente contém uma série de sensores definidos pelo desenvolvedor (cada sensor captura um tipo diferente de objeto do ambiente) que são verificados toda vez que é percebido um objeto semântico nesse ambiente. Se um ou mais sensores forem ativados, os objetos são encaminhados para o processamento em outros componentes.

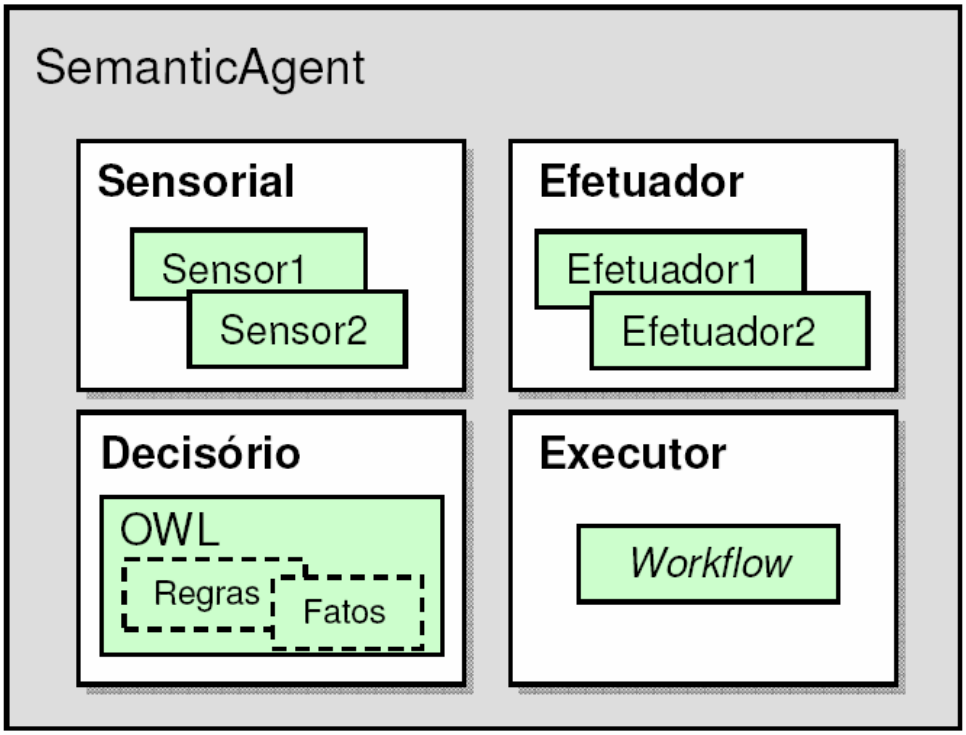


Figura 2.16 – Modelo do Agente [ESC06].

O componente decisório encapsula o mecanismo de tomada de decisão do agente. O mecanismo decisório presente no componente é um dos pontos de flexibilidade do *framework*. Como o *SemantiCore* almeja o desenvolvimento de aplicações voltadas a Web Semântica, o componente decisório opera sobre ontologias (em termos de fatos e regras), o que torna necessário o uso de uma linguagem apropriada para a definição semântica dos dados, como a *Web Ontology Language (OWL)*. A saída gerada pelo componente decisório deve ser uma instância de uma ação (*Action*). As ações mapeiam todos os possíveis comandos que um agente deve entender para trabalhar de forma apropriada e podem ser aplicadas tanto aos elementos do agente quanto aos elementos do domínio semântico. O desenvolvedor pode definir suas próprias ações por meio de uma extensão da classe *Action* (*hotspot*) presente no *framework*. No *SemantiCore*, as ações são vistas como processos de computação, cujo modelo é apresentado por [FER99].

O componente executor contém os planos de ação que serão executados pelo agente e trabalha com o mecanismo de *workflow*. Esse mecanismo é necessário para o controle das transições de atividades dentro de um processo do *workflow*.

O encapsulamento de dados em mensagens para transmissão no ambiente é feito pelo componente efetuador. De maneira semelhante ao componente sensorial, o componente

efetuador armazena uma série de efetuadores, onde cada efetuador é responsável por publicar um tipo diferente de objeto no ambiente. Uma das características do *SemantiCore* é a abstração da plataforma de software e do protocolo de comunicação, possibilitando ao desenvolvedor da aplicação enviar e receber mensagens usando diferentes padrões, como *Web Service SOAP* [GUD02] e *FIPA ACL* [FIP07].

Como pode ser visto na Figura 2.16, não há nenhuma ligação (ou caminho de dados) definida entre os componentes do agente. A comunicação entre os componentes no *SemantiCore* é também um ponto de flexibilidade e deve ser definida pelo desenvolvedor.

2.4.2 Jadex

O *Jadex* é um *framework* para a criação de agentes orientados a objetivos que segue o modelo *BDI* e basicamente o modelo computacional *Procedural Reasoning System (PRS)* [GEO87] [ING92]. Segundo [POK05], o projeto *Jadex* objetiva o desenvolvimento de sistemas baseados em agentes de maneira simples sem sacrificar o poder do paradigma de agentes.

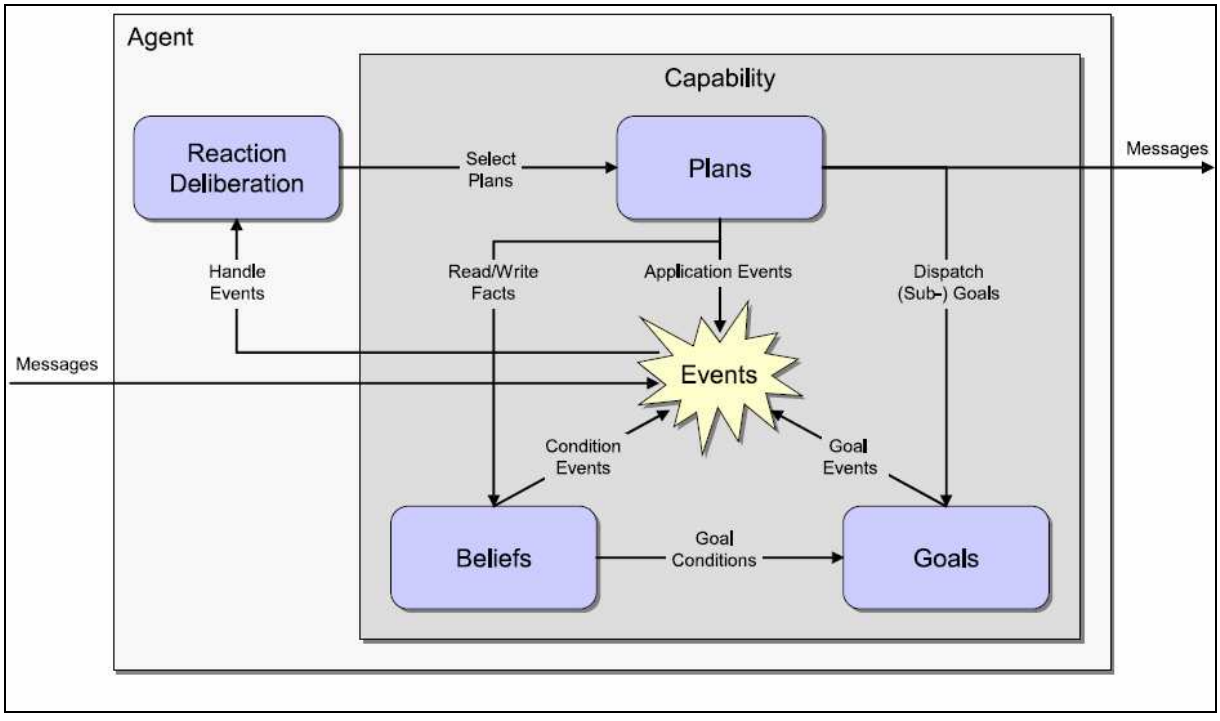


Figura 2.17 – Visão Geral da Arquitetura Abstrata *Jadex* [POK05].

Na Figura 2.17 é apresentada uma visão geral da arquitetura abstrata *Jadex*. Em uma visão externa, um agente é uma caixa preta, que recebe e envia mensagens. Todos os tipos de

eventos, como mensagens de entrada e eventos de objetivo servem como entrada para a reação interna e para o mecanismo de deliberação, que despacha os eventos para os planos selecionados da biblioteca de planos. No *Jadex,* o mecanismo de reação e deliberação é o único componente global de um agente. Todos os outros componentes são agrupados em módulos reusáveis chamados capacidades.

Um dos objetivos do projeto *Jadex* é a adoção de uma perspectiva de engenharia de software para descrever agentes. Em outros sistemas *BDI*, crenças são representadas em algum tipo de lógica de predicados de primeira ordem ou pelo uso de modelos relacionais. No *Jadex*, uma representação orientada a objetos das crenças é empregada, onde objetos arbitrários podem ser gravados como fatos nomeados (chamados crenças) ou conjuntos de fatos nomeados (chamados conjuntos de fatos). Operações em uma base de crenças podem ser emitidas em uma linguagem de consulta orientada a conjuntos descritivos. Além disso, uma base de crenças não é apenas um armazenamento de dados passivo, mas recebe uma parte ativa da execução do agente, pela monitoração das condições dos estados das crenças. Mudanças nas crenças podem então levar diretamente a ações, como eventos sendo gerados ou objetivos sendo criados ou abandonados.

Objetivo é um conceito central no *Jadex*, seguindo a idéia geral de que objetivos são concretos desejos momentâneos de um agente. Para qualquer objetivo que possui, um agente irá se engajar em ações convenientes, até que considere o objetivo como alcançado, não-alcançado, ou como um objetivo não mais procurado. Em outros sistemas *PRS*, objetivos são representados por tipos especiais de eventos. Conseqüentemente, nesses sistemas os objetivos correntes de um agente são apenas implicitamente disponíveis como as causas de planos executando correntemente. No *Jadex,* objetivos são representados como objetos explícitos contidos em uma base de objetivos, que são acessíveis ao componente de raciocínio assim como aos planos, caso eles necessitem saber ou mudar os objetivos correntes do agente. Devido aos objetivos serem representados separadamente em relação aos planos, o sistema pode reter objetivos que não estão correntemente associados a planos. Como resultado, diferentemente de outros sistemas *BDI,* o *Jadex* não requer que todos os objetivos adotados sejam consistentes entre si, contanto que somente subconjuntos consistentes desses objetivos sejam procurados em um dado momento. Para distinguir entre os objetivos que são apenas adotados e os objetivos ativamente procurados, um ciclo de vida de objetivo é introduzido e consiste nos estados opcional, ativo e suspenso. Quando um objetivo é adotado, esse se torna uma opção que é adicionada à base de objetivos do agente como um objetivo de alto-nível, ou quando criados de um plano, como um subobjetivo de um objetivo raiz do plano. A

deliberação de objetivos específicos de uma aplicação atribui dependências específicas entre os objetivos, essas dependências são usadas para gerenciar as transições entre estados de todos os objetivos adotados (isto é, decidindo quais objetivos são ativos e quais são apenas opções). Adicionalmente, alguns objetivos podem apenas ser válidos em contextos específicos determinados pelas crenças dos agentes. Quando o contexto de um objetivo é inválido, esse será suspenso até que o contexto seja novamente válido.

Jadex suporta quatro tipos de objetivos: execução, alcance, consulta e manutenção. Esses tipos estendem o ciclo de vida geral e exibem comportamentos diferentes considerando seu processamento como explicado anteriormente. Inicialmente, um objetivo de execução é diretamente relacionado à execução de ações. Conseqüentemente, um objetivo é considerado um objetivo que deve ser alcançado, quando algumas ações foram executadas, independente do resultado dessas ações. Um objetivo de alcance é um objetivo no senso tradicional, definindo um estado de mundo desejado sem especificar como alcançá-lo. Agentes podem tentar a execução de vários planos alternativos para alcançar um objetivo desse tipo. Um objetivo de consulta é similar a um objetivo de alcance, mas o estado desejado não é o estado do mundo (externo), mas um estado interno do agente, relativo à disponibilidade de algumas informações que o agente busca conhecer. Para objetivos do tipo manutenção, um agente sustenta o caminho de um estado desejado, e executará continuamente planos apropriados para restabelecer esse estado mantido sempre que necessário.

No *Jadex*, planos representam os elementos comportamentais de um agente e são compostos por um cabeçalho e um corpo. A especificação do cabeçalho de um plano no *Jadex* é similar a de outros sistemas *BDI* e especifica principalmente as circunstâncias em que um plano pode ser selecionado, como por exemplo, a declaração de eventos ou objetivos manuseados pelo plano, e as pré-condições para a execução do plano. Adicionalmente, no cabeçalho de um plano, uma condição de contexto pode ser declarada como verdadeira para o plano poder continuar sua execução. O corpo de um plano dispõe de um curso pré-definido de ações, dado em uma linguagem procedural. Esse curso de ações deve ser executado pelo agente quando o plano for selecionado para a execução e pode conter ações fornecidas pela *API* do sistema, como mensagens de envio, crenças em manipulação, ou criação de subobjetivos.

Por fim, as capacidades representam um mecanismo de agrupamento para os elementos de um agente *BDI*. Esses elementos podem ser crenças, objetivos, planos e eventos. Sendo assim, elementos altamente relacionados podem ser colocados juntamente em um módulo reutilizável, que encapsula alguma funcionalidade (por exemplo, a interação com o

facilitador de diretórios *FIPA*). A capacidade de um elemento representa seu escopo e um elemento tem acesso apenas a elementos do mesmo escopo (por exemplo, um plano pode acessar apenas crenças, objetivos ou eventos da mesma capacidade). Para conectar diferentes capacidades, mecanismos de importação e exportação flexíveis podem ser utilizados definindo a interface externa da capacidade.

2.4.3 Jason

[BOR06] afirma que o *Jason* utiliza a linguagem de programação *AgentSpeak(L)* como base, provendo várias extensões que são necessárias para o desenvolvimento prático de SMAs. A *AgentSpeak(L)* é uma extensão natural de programação lógica para a arquitetura de agentes *BDI*, e disponibiliza um *framework* abstrato para a programação de agentes.

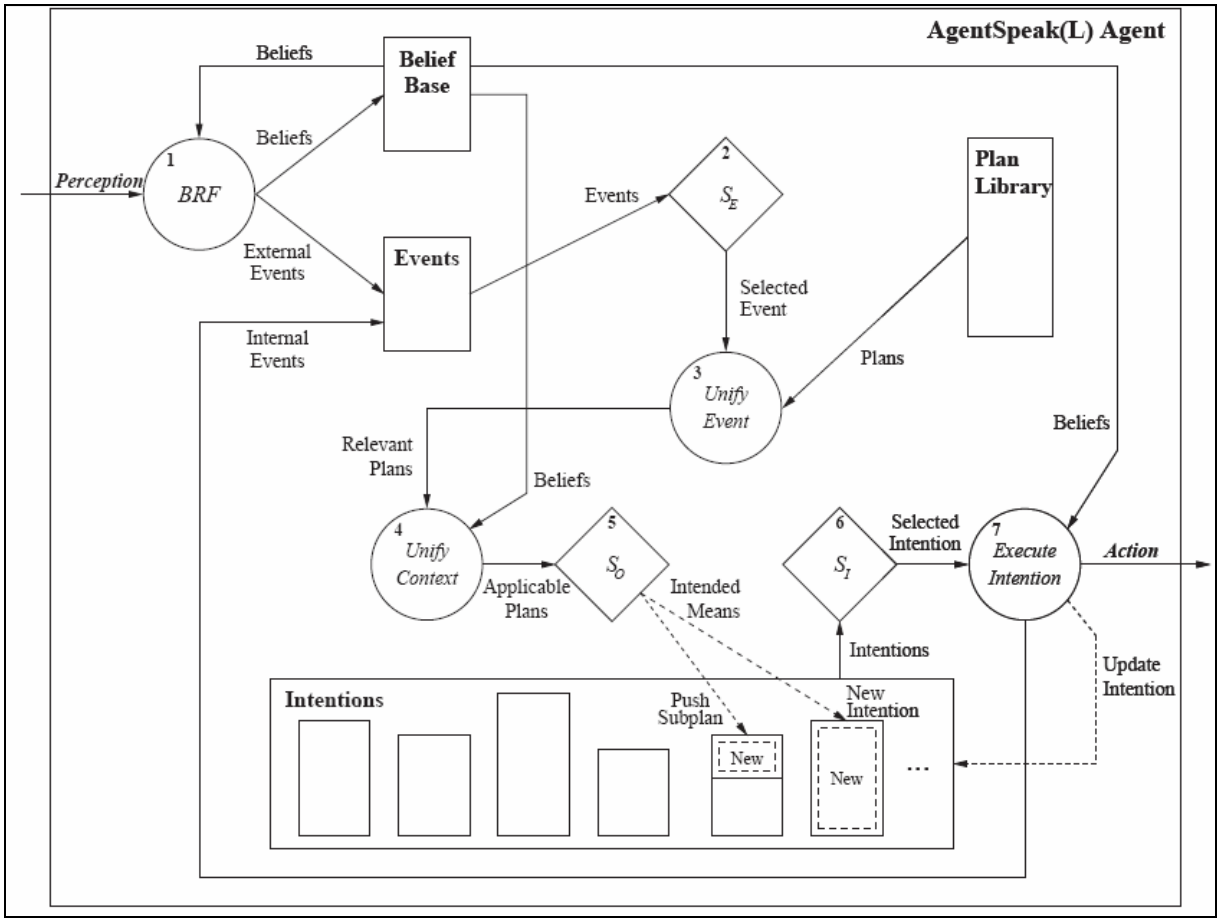


Figura 2.18 – Ciclo de Interpretação de um Programa *AgentSpeak* [MAC02].

Um agente *AgentSpeak* é definido por um conjunto de crenças que estabelecem o estado inicial da base de crenças (conjunto de fórmulas atômicas de primeira ordem) e por um

conjunto de planos que formam a biblioteca de planos. Antes de explicar como um plano é descrito, é necessário introduzir as noções de objetivos e eventos de disparo. *AgentSpeak* distingue dois tipos de objetivos: objetivos de alcance e objetivos de teste. Objetivos de alcance são formados por uma formula atômica pré-fixada com o operador "!" , enquanto que objetivos de teste são pré-fixados com o operador "?". Um objetivo de alcance declara que um agente busca atingir um estado do mundo quando a fórmula atômica associada é verdadeira. Um objetivo de teste declara que o agente busca testar se a fórmula atômica associada é (ou pode ser unificada com) uma de suas crenças. Conforme [MAC02], a Figura 2.18 apresenta um ciclo de interpretação de um programa *AgentSpeak*.

[BOR06] destaca que um agente *AgentSpeak* é um sistema de planejamento reativo. Os eventos aos quais o agente reage são relacionados a mudanças nas crenças devido à percepção do ambiente, ou a mudanças nos objetivos do agente que se originam da execução de planos disparados por eventos prévios. Um evento de disparo define quais eventos podem iniciar a execução de um plano em particular. Planos são disparados pela adição ("+") ou exclusão ("-") de crenças ou objetivos (as atitudes mentais de agentes *AgentSpeak*).

Um plano *AgentSpeak* tem um cabeçalho, que é formado por um evento de disparo (especificando os eventos para os quais o plano é relevante), e uma conjunção de literais de crenças representando um contexto. A conjunção de literais deve ser uma conseqüência lógica das crenças correntes do agente caso o plano seja considerado aplicável naquele momento (apenas planos aplicáveis podem ser escolhidos para execução). Um plano também tem um corpo, que é uma seqüência básica de ações ou objetivos (subobjetivos) que o agente deve atingir (ou testar) quando o plano é disparado. Corpos de planos incluem ações básicas, ou seja, ações representando operações atômicas que o agente pode executar de forma que altere o ambiente. Tais ações também são escritas como fórmulas atômicas, porém usando preferencialmente um conjunto de símbolos de ações ao invés de símbolos de predicado.

As principais diferenças entre a linguagem interpretada pelo *Jason* e a *AgentSpeak(L)* original são descritas a seguir. Inicialmente, sempre que uma fórmula atômica for permitida na linguagem original, é utilizado um literal em seu lugar. Isto é, uma fórmula atômica ou uma formula atômica negada (~ (negação forte)). A negação padrão é usada no contexto de planos, e é denotada por um *not* precedendo um literal. Um contexto é então uma conjunção de literais padrão. Termos no *Jason* podem ser variáveis, listas (com sintaxe *Prolog*), assim como números inteiros ou ponto-flutuantes, e *strings*; fora isso, qualquer fórmula atômica pode ser tratada como um termo, e variáveis amarradas podem ser tratadas como literais

(particularmente importante para introduzir comunicação). Operadores relacionais in-fixados, como no *Prolog*, são permitidos em contextos de planos.

Além disso, uma mudança maior é que uma fórmula atômica no *Jason* pode ter "anotações". Isto é, uma lista de termos entre colchetes seguindo imediatamente a fórmula. Dentro da base de crenças, anotações são usadas, por exemplo, para registrar as origens da informação. O termo origem é usado nas anotações para essa proposta; a origem pode ser um nome de agente (para denotar o agente que comunicou aquela informação), ou dois átomos especiais, percepção e *self*, que são usados respectivamente para denotar que uma crença surge de uma percepção do ambiente ou do agente explicitamente adicionando uma crença à sua base de crenças na execução de um corpo de plano. As crenças iniciais que fazem parte do código fonte de um agente *AgentSpeak* são assumidas como crenças internas, a menos que uma crença tenha qualquer anotação explícita dada pelo usuário.

Planos também têm *labels*. Todavia, um *label* de um plano no *Jason* pode ser qualquer fórmula atômica, incluindo anotações. Anotações em *labels* de planos podem ser usadas para a implementação de funções de seleção de planos. Embora isso ainda não esteja disponível na distribuição corrente do *Jason*, fica claro para o usuário definir, por exemplo, funções de seleção decisão-teoria que usam algo como funcionalidades esperadas anotadas nos *labels* de planos para escolher entre planos alternativos. A customização de funções de seleção é feita em Java. Além disso, como o *label* é parte de uma instância de um plano no conjunto de intenções e as anotações podem ser alteradas dinamicamente, são disponibilizados todos os meios necessários para a implementação de funções de seleção de intenções eficientes. Porém, essa funcionalidade também não está disponível como parte da distribuição *Jason*, mas pode ser configurada por usuários com alguma customização.

Eventos para manusear falhas de planos já estão disponíveis no *Jason*, embora ainda não estejam formalizados na semântica. Se uma ação falha ou não existe plano aplicável para o subobjetivo do plano que está sendo executado, então o plano total com falha é removido do topo da intenção e um evento interno associado com aquela mesma intenção é gerado. Se o programador disponibiliza um plano que tem um evento de disparo combinado e é aplicável, tal plano será colocado no topo da intenção, e então o programador pode especificar no corpo de tal plano como que uma falha particular deve ser manuseada. Se não existir um plano disponível, a intenção como um todo é descartada e um *warning* é impresso no console. Efetivamente, isso provê um meio para os programadores fazer uma limpeza após um plano falho.

Finalmente, ações internas podem ser usadas no contexto e no corpo de planos. Qualquer símbolo de ação iniciado com “.” ou tendo um “.” em qualquer posição, denota uma ação interna. Uma ação é chamada de interna para fazer a clara distinção com as ações que aparecem no corpo de um plano e aquelas que denotam ações que um agente pode executar com o objetivo de mudar o ambiente compartilhado (por meio de seus “efetuadores”). No *Jason*, ações internas são codificadas em Java, ou em outras linguagens de programação pelo uso de *Java Native Interface (JNI)*, e podem ser organizadas em bibliotecas de ações para propósitos específicos.

2.4.4 JACK

[HOW01] afirma que o *framework JACK Intelligent Agents* desenvolvido pelo *AOS* traz o conceito de agentes inteligentes como uma importante tendência na engenharia de software comercial e em Java. *JACK* é um *framework* de agentes de terceira geração, projetado como um conjunto de componentes leves com alto desempenho e fortemente tipados.

Para um programador de aplicações, *JACK* atualmente consiste de três principais extensões para Java. A primeira é um conjunto de adições sintáticas para sua linguagem hospedeira. Esse pode ser dividido como:

- Um pequeno número de palavras-chave para a identificação dos componentes principais de um agente (como agente, plano e evento).
- Um conjunto de expressões para a declaração de atributos e outras características de componentes (por exemplo, a informação contida em crenças e carregada por eventos).
- Um conjunto de expressões para a definição de relacionamentos estáticos (por exemplo, quais planos podem ser adotados para reagir a um certo evento).
- Um conjunto de expressões para a manipulação de um estado de agente (por exemplo, adições de novos objetivos ou subobjetivos a serem alcançados, mudanças nas crenças, interação com outros agentes).

A segunda extensão para Java é um compilador que converte as adições sintáticas descritas acima para classes Java puras e para expressões que podem ser carregadas e

chamadas por outro código Java. O compilador também transforma parcialmente o código dos planos para obter a correta semântica da arquitetura *BDI*.

Finalmente, um conjunto de classes (chamadas de *kernel*) fornece o suporte requerido em tempo de execução para o código gerado. Isso inclui:

- Gerenciamento automático de concorrência entre tarefas sendo buscadas em paralelo (intenções na terminologia *BDI*).
- Comportamento padrão de um agente em reação a eventos, falhas de ações e tarefas e assim por diante.
- Leveza nativa, infra-estrutura de comunicação de alto desempenho para SMAs.

O *kernel JACK* suporta múltiplos agentes dentro de um processo único, múltiplos agentes em múltiplos processos, e uma combinação desses. Isso é particularmente conveniente para salvar recursos do sistema.

O *JACK* utiliza agentes *BDI*. Com isso, um agente racional tem recursos, entendimento limitado e conhecimento incompleto do que ocorre no ambiente em que vive. Um agente tem crenças sobre o mundo e desejos para satisfazer, dirigindo isso em intenções para agir. Uma intenção é um compromisso para executar um plano. Em geral, um plano é parcialmente especificado no momento de sua formulação visto que os passos exatos a serem executados podem depender do estado do ambiente. A atividade de um agente racional consiste em executar ações que são pretendidas sem qualquer raciocínio adicional, até que isso force a revisão das intenções do agente por mudanças nas crenças ou desejos. Crenças, desejos e intenções são chamados de atitudes mentais (ou estado mentais) de um agente.

Com base em pesquisas prévias e na aplicação prática, [RAO92] descreveu um modelo computacional para um sistema genérico de software implementando um agente *BDI*. Esse sistema é um exemplo de programas guiados por eventos. Em reação a um evento, por exemplo, uma mudança no ambiente ou em suas próprias crenças, um agente *BDI* adota um plano como uma de suas intenções. Planos são procedimentos pré-compilados que dependem de um conjunto de condições para serem aplicáveis. O processo de adotar um plano como uma das intenções do agente pode requerer a seleção entre múltiplos candidatos.

O agente executa os passos dos planos que adotou como intenções até que uma deliberação a mais seja requerida, isso pode ocorrer por causa de novos eventos ou por falha ou sucesso na conclusão das intenções existentes.

Um passo de um plano pode consistir em adicionar um objetivo para o próprio agente, alterar suas crenças, interagir com outros agentes ou qualquer outra ação atômica no estado do agente ou no mundo externo.

2.5 Considerações

Nesse capítulo, foi apresentado todo o estudo teórico realizado para o desenvolvimento do meta-modelo. O estudo de agentes de software possibilitou uma visão geral da área, além de permitir a identificação das características internas de agentes tratadas na literatura. Por outro lado, com o estudo das abordagens, foi possível a definição de um meta-modelo inicial. Ainda nesse capítulo, foram introduzidos os conceitos de restrições de integridade e linguagem *OCL*, possibilitando assim a aplicação das restrições de integridade e a posterior verificação da consistência de modelos instanciados a partir do meta-modelo. Por fim, o estudo de algumas plataformas de implementação de SMAs foi realizado com o objetivo de verificar a possibilidade de geração de código nas mesmas.

No capítulo seguinte será apresentado o processo de desenvolvimento do meta-modelo baseado no estudo teórico realizado ao longo desse capítulo.

3 DESENVOLVIMENTO DO META-MODELO

Neste capítulo, é apresentado o meta-modelo inicial desenvolvido em [SAN06a]. Após isso, o refinamento do meta-modelo inicial, assim como a aplicação das restrições de integridade *OCL* ao meta-modelo refinado, são apresentados. Por fim, será descrito o estudo realizado com o objetivo de verificar se os conceitos internos de um agente tratados em plataformas de implementação de SMAs são cobertos pelo meta-modelo refinado.

3.1 Meta-modelo Inicial

Com base no estudo realizado nas diferentes abordagens orientadas a agentes, foram identificados alguns conceitos e relacionamentos para a modelagem interna de agentes, apresentados por meio da definição de um meta-modelo inicial.

É importante ressaltar que alguns dos conceitos e denominações definidos nas abordagens apresentadas não serão aplicados diretamente ao meta-modelo inicialmente proposto neste trabalho. Por exemplo, na atividade de Especificação de uma Classe de Agente da metodologia *MASUP*, a denominação “atribuições” não foi utilizada, pois a mesma foi mapeada no meta-modelo inicial como um conjunto de ações exercidas por um determinado papel. Por outro lado, o conceito de instâncias em uma sociedade não foi utilizado por não tratar diretamente da modelagem interna de agentes e sim da modelagem de seu ambiente.

Para a metodologia *Tropos*, não foram considerados os conceitos de ator e posição, pois estes se referem à organização e não à estrutura interna dos agentes.

O conceito tarefas da metodologia *MaSE* foi mapeado para o conceito de ações, pois segundo [HEN05a] a palavra ação pode ser empregada para descrever o trabalho feito para atingir um objetivo ou subobjetivo.

As capacidades definidas na metodologia *Prometheus* são representadas no meta-modelo inicial por eventos, recursos e planos.

Na abordagem *MAS-ML*, todos os conceitos definidos na Classe de Agente foram aplicados na construção do meta-modelo, são eles: objetivos, crenças e ações.

Baseando-se nos *templates* textuais da metodologia *MAS-CommonKADS*, os únicos conceitos que não foram utilizados no meta-modelo são a posição, pelo mesmo motivo explicado na metodologia *Tropos*, e as capacidades de raciocínio, por não serem abordadas diretamente nas demais abordagens e por se tratarem apenas de uma descrição textual no Modelo de Agentes da fase de Análise.

A Tabela 3.1 apresenta a relação dos conceitos identificados para o meta-modelo inicial com as abordagens estudadas em que estes conceitos são utilizados. Nessa tabela, cada relação marcada com "X" indica que o conceito representado na linha é tratado na abordagem representada na coluna. Por exemplo, o conceito plano é tratado nas abordagens *Tropos*, *Prometheus* e *MAS-ML*. Por outro lado, pode-se dizer que a metodologia *MAS-CommonKADS* trata dos conceitos objetivo, ação, crença, papel, restrição e percepção.

TABELA 3.1 – CONCEITOS DO META-MODELO INICIAL X CARACTERÍSTICAS DAS ABORDAGENS.

	MASUP	*Tropos*	*MaSE*	*Prometheus*	*MAS-ML*	*MAS-CommonKADS*
Objetivo	X	X	X	X	X	X
Subobjetivo			X	X		
Plano		X		X	X	
Ação	X	X	X	X	X	X
Crença	X	X	X	X	X	X
Interface de interação	X					
Papel	X	X	X	X	X	X
Restrição	X	X	X		X	X
Recurso	X	X	X	X		
Percepção	X	X	X	X	X	X
Evento				X		

Os conceitos identificados da análise das abordagens e da literatura da área são detalhados a seguir:

- **Objetivo**: antes da definição em si, é importante a descrição de dois conceitos relacionados aos objetivos no contexto de agentes *BDI*. O primeiro deles são os desejos e o segundo são as intenções. [MÜL96] define desejos como uma noção abstrata que especifica preferências por estados futuros do mundo ou cursos de ação. Uma importante característica de um desejo é que agentes podem ter desejos inconsistentes e estes não precisam necessariamente acreditar que seus desejos são alcançáveis. Além disso, um agente é ligado a recursos, e, devido a isso, na maioria das vezes ele não pode ter todos os seus objetivos satisfeitos imediatamente. Mesmo que um conjunto de objetivos seja consistente, também é necessário fazer a seleção de um determinado objetivo (ou conjunto de objetivos) para confirmação. Este processo é chamado de formação de intenções. Com isso, pode-se destacar que as intenções correntes de um agente ou sua estrutura intencional são descritas por um conjunto de objetivos selecionados juntamente com seus estados de processamento. Detalhados os conceitos de desejo e intenção, [ODE00] define os objetivos como desejos ou estados futuros que o agente deve

atingir, ajudando-o a determinar que ações adotar em circunstâncias particulares. [HEN05a] salienta que para atingir um objetivo deve existir uma ação ou uma série de ações que permitam o alcance deste em um tempo finito.

- **Subobjetivo:** [HEN05a] destaca que a noção de subobjetivo é ambígua. Em certos casos, subobjetivo é conceituado como um objetivo contido no caminho do alcance do objetivo principal, outras vezes, é considerado como uma parte do objetivo principal. Para o segundo caso, conforme [HEN05a], podemos denominá-lo como **objetivo parcial**.
- **Plano**: [WOO02a] define um plano como uma tupla formada por: um conjunto de pré-condições, definindo em quais circunstâncias um plano é aplicável; um corpo, definindo uma seqüência de ações possíveis; e um conjunto de pós-condições, definindo os estados que um plano pode atingir. [MÜL96] destaca que embora os planos não sejam ingredientes conceituais da teoria *BDI*, esses são muito importantes para a implementação pragmática das intenções. [BRA87] coloca ênfase no fato que as intenções são planos parciais de ações que o agente deve executar para atingir seus objetivos. Logo, é possível estruturar intenções em grandes planos, e definir intenções de agentes como planos correntemente adaptados.
- **Ação**: como descrito anteriormente, planos são formados por conjuntos de ações. Essas também são conhecidas como tarefas. [ZAM04] conceitua uma tarefa como uma parte de trabalho que pode ser atribuída a um agente ou ser executada por este.
- **Crença**: [MÜL96] afirma que crenças de um agente expressam as expectativas desse sobre o estado atual do mundo e sobre a probabilidade de um curso de ação atingir certos efeitos. Crenças são modeladas com o uso de semânticas de possíveis palavras, onde um conjunto de possíveis palavras é associado com cada situação, denotando os mundos que o agente acredita serem possíveis. [FER99] salienta que crenças podem ser formalizadas como proposições que representam o mundo e essas proposições podem ser interpretadas quando mapeadas para modelos. O poder de um agente cognitivo está na sua faculdade de representação, isto é, na sua capacidade de desenhar modelos do mundo, tornando possível o entendimento, a explicação e a predição de eventos e a evolução dos fenômenos.

- **Interface de interação**: interfaces de interação definem o protocolo de mensagens aceitas por determinado agente. [ZAM04] conceitua protocolo como um conjunto de mensagens ordenado que define os padrões aceitáveis de um tipo particular de interações entre entidades. Mensagens são meios de trocar fatos ou objetos entre entidades.
- **Papel**: [ZAM04] conceitua papel como uma representação abstrata de uma função de agente, serviço ou identificação dentro de um grupo. Cada papel pode ter associado a si um conjunto de atribuições e restrições.
- **Restrição**: em [BAS05] é utilizado o conceito de restrições associadas a um determinado papel. Essas limitam a execução de ações de um agente que exerce determinado papel.
- **Recurso**: segundo [BRE04], um recurso é definido como uma entidade física ou uma informação necessária para o correto funcionamento de um agente.
- **Percepção**: [WOO02] destaca que uma função que representa uma percepção deve capturar a habilidade do agente de observar o ambiente em que está inserido.
- **Evento**: [HEN05] destaca que um agente reativo apenas responde ao ambiente. Alterações neste ambiente são comunicadas por meio de eventos. Esses também podem ocorrer como um resultado direto de mensagens enviadas por outros agentes ou mesmo mensagens enviadas internamente. Para cada evento se espera que o agente dispare uma ação ou um plano.

O meta-modelo inicial para a representação interna de agentes de software é apresentado na Figura 3.1 pelo uso de um Diagrama de Classes *UML* mostrando os conceitos identificados e os relacionamentos entre os mesmos. Nesse, cada papel (*Role*) exercido por um agente pode executar uma série de ações (*Actions*) e possuir diversas restrições (*Restrictions*) associadas, ambos representando quais os comportamentos possíveis para um dado papel. Cada papel também almeja atingir diversos objetivos (*Goals*). Esses permitem duas formas de decomposição: subobjetivo (*SubGoal*) e objetivo parcial (*PartialGoal*), ambas já explicadas anteriormente. Cada objetivo pode ser cumprido por um ou mais planos (*Plans*), e por zero ou mais ações. Um plano pode ser composto por diversas ações e se baseia em uma ou mais crenças (*Beliefs*). Sempre que um agente tem uma percepção (*Perception*), zero ou mais eventos (*Events*) podem ser disparados respeitando uma ou mais crenças do agente e, com isso, uma ou mais ações podem ser iniciadas. Outro conceito representado no meta-

modelo inicial são as interfaces de interação (*InteractionInterfaces*), que especificam os padrões dos agentes para o envio e recebimento de mensagens do ambiente, e podem estar associadas a uma ou mais ações e a uma ou mais percepções. Por fim, uma interface de interação usa um ou mais recursos (*Resources*).

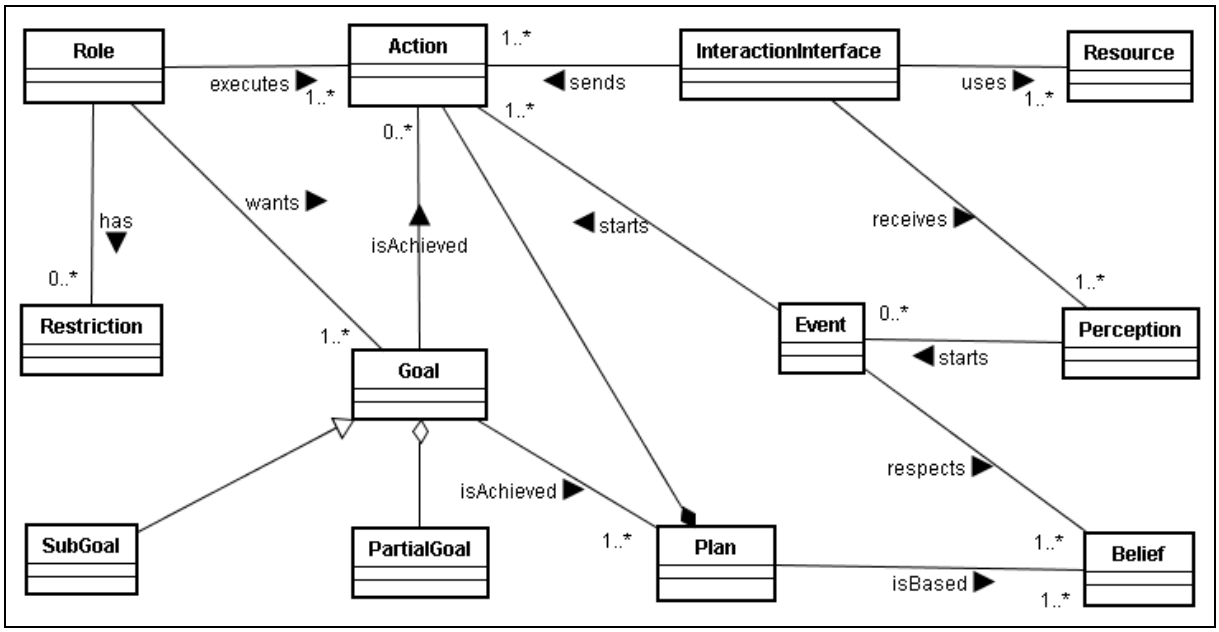


Figura 3.1 – Meta-modelo inicial proposto.

3.2 Refinamento do Meta-modelo

Após a definição do meta-modelo inicial, foi aprofundado o estudo da representação interna de agentes na literatura da área e nas abordagens. Desse estudo, foram aplicadas algumas alterações no meta-modelo inicial assim como a inclusão de novos conceitos e novos relacionamentos. O meta-modelo resultante deste refinamento é apresentado na Figura 3.2.

3.2.1 Refinamento dos Conceitos do Meta-modelo

Nos itens a seguir serão descritas as alterações realizadas nos diferentes conceitos do meta-modelo. São elas:

- **InteractionInterface**: esse conceito agora é representado pelos conceitos **Protocol**, representando o protocolo de comunicação de mensagens utilizado pelo agente; **Message**, representando as mensagens de entrada e saída do agente; e

Field, representando os diferentes parâmetros que compõem um determinado tipo de mensagem.

- **Restriction:** esse conceito é agora representado pelos relacionamentos de direitos e deveres do conceito *Role* com o conceito *Action*.

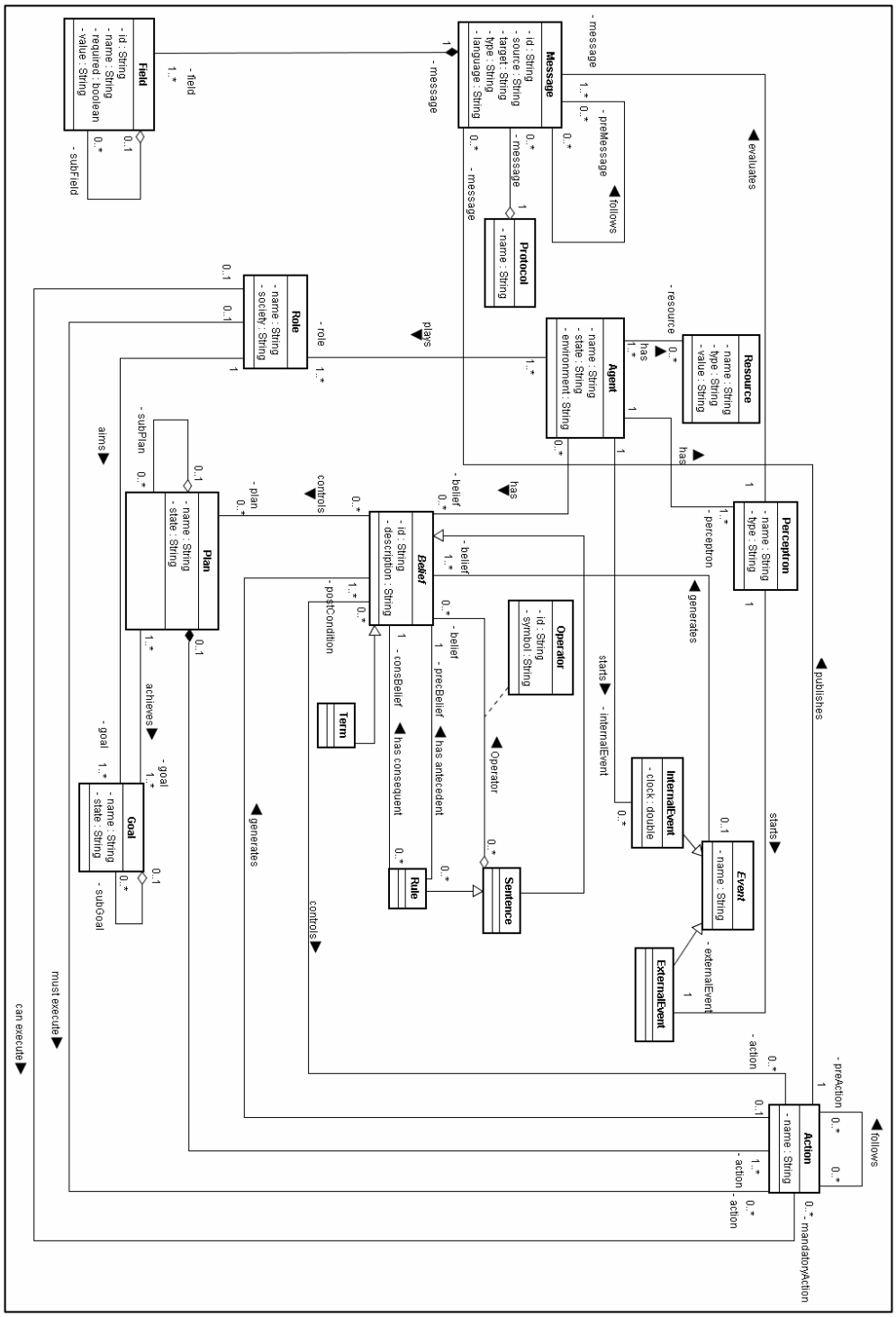


Figura 3.2 – Meta-modelo refinado.

- **Perception:** a denominação *Perception* foi retirada do meta-modelo, sendo substituída pelo conceito **Perceptron.** O *Perceptron* aceita mensagens que vem do ambiente para o agente de acordo com um padrão pré-definido. As mensagens aceitas disparam eventos externos (**ExternalEvents),** outro conceito incluído no meta-modelo, esses podem ser definidos como alterações que o agente recebe do ambiente.
- **PartialGoal e SubGoal:** esses conceitos apresentam um significado semelhante entre si. Devido a isso, eles passaram a ser considerados simplesmente como partes de um objetivo principal. Disso, ambos os conceitos foram retirados do meta-modelo e agora são representados na auto-relação de agregação existente no conceito *Goal*.

Além das alterações descritas, foram incluídos os seguintes conceitos no meta-modelo:

- **Agent:** esse conceito foi incluído com o objetivo de melhorar a representação da relação conceitual entre o agente e os conceitos que o compõem. Um agente é um sistema computacional inserido em um ambiente, capaz de atingir os objetivos planejados por meio de ações autônomas nesse ambiente [WOO02].
- **InternalEvent:** apesar da existência de eventos no meta-modelo inicial, foram criadas duas especializações para esse conceito, o evento externo (já citado anteriormente) e o evento interno (*InternalEvent*), que pode ser disparado com ou sem o uso de um *clock* interno do agente. Um evento interno pode ser conceituado como uma alteração interna no comportamento do agente.
- **Term:** uma crença pode ser representada por um termo, este, segundo [NOR03] é uma expressão lógica que se refere a um objeto.
- **Sentence:** segundo [NOR03], uma sentença enuncia fatos, sendo representada por um símbolo de predicado seguido por uma lista de termos, podendo utilizar conectivos lógicos.
- **Operator:** no meta-modelo refinado, representa os conectivos lógicos utilizados na relação entre uma sentença e uma crença.
- **Rule**: são tipos de sentença que devem necessariamente possuir crenças como antecedente e conseqüente, em que a primeira implica na segunda.

3.2.2 Aplicação das Restrições de Integridade

Expressões escritas em uma linguagem como *OCL* oferecem diversos benefícios sobre diagramas para especificar um sistema [WAR03]. Essas expressões não podem ser interpretadas diferentemente por diferentes pessoas, como por exemplo, um analista e um programador. Elas são não-ambíguas e tornam um modelo mais preciso e mais detalhado. Essas expressões podem ser verificadas por ferramentas de automação para garantir que estão corretas e consistentes com outros elementos do modelo. Com restrições de integridade, a geração de código se torna muito mais poderosa.

Da necessidade de especificar um meta-modelo consistente e sem ambigüidades foram aplicadas restrições de integridade *OCL* nos conceitos identificados assim como nos relacionamentos entre esses. Inicialmente, serão explicadas as restrições de integridade aplicadas aos atributos dos conceitos. Após isso, serão descritas as restrições aplicadas aos relacionamentos. Uma importante consideração é que todas as restrições aplicadas ao meta-modelo consideram os atributos como do tipo *String*, pois esse é o único tipo de dado que será usado na classificação dos atributos de cada conceito no protótipo, com o objetivo de evitar constantes conversões de tipos dados no mesmo.

3.2.3 Detalhamento do Meta-modelo

O meta-modelo desenvolvido também pode ser apresentado por meio de uma visão de pacotes, facilitando sua compreensão. Sendo assim, na Figura 3.3 é apresentada a visão geral dos pacotes do meta-modelo.

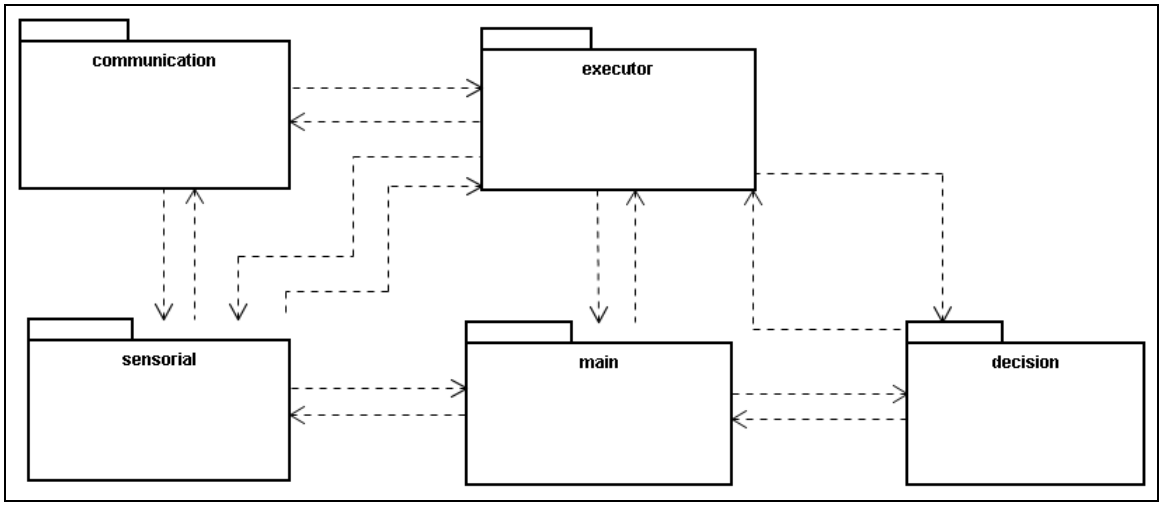


Figura 3.3 – Visão Geral dos Pacotes do Meta-modelo.

Nas Figuras 3.4, 3.5, 3.6, 3.7 e 3.8 são apresentados os diferentes pacotes que compõem o meta-modelo, são eles: Pacote *Main*, Pacote *Sensorial*, Pacote *Executor*, Pacote *Decision* e Pacote *Communication*. Após a apresentação visual, são detalhados os atributos de cada pacote, os relacionamentos entre conceitos e as restrições de integridade aplicadas.

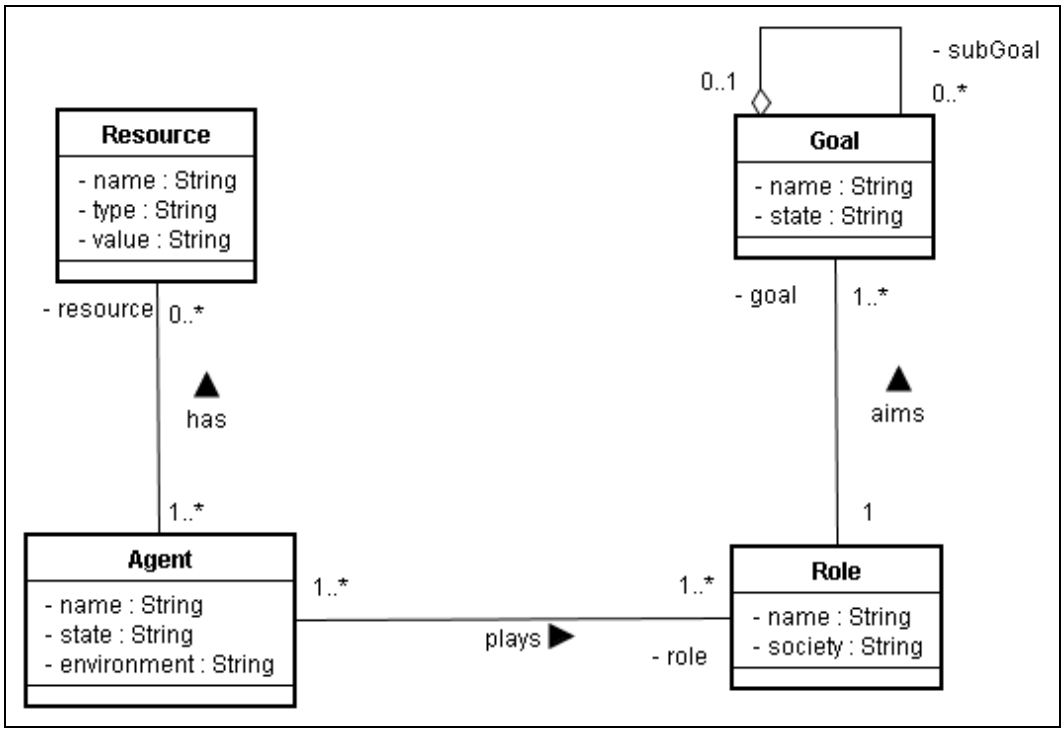


Figura 3.4 – Pacote *Main*.

Detalhamento dos atributos do pacote *Main*:

- **Agent:** *name*, atributo alfanumérico que identifica um agente no ambiente; *state*, atributo alfanumérico que descreve o estado atual de um agente, podendo assumir os valores *created* (agente criado no ambiente), *execution* (agente realizando uma tarefa), *ready* (agente pronto para executar tarefa), *blocked* (agente em espera) e *finished* (agente finalizado); *environment*, atributo alfanumérico que descreve o ambiente em que um agente está localizado. Ao atributo *name* é aplicada uma restrição de obrigatoriedade, indicando que o atributo *name* deve ser informado para o conceito *Agent*, e uma restrição de unicidade, indicando que o atributo *name* não pode assumir o mesmo valor para diferentes instâncias do conceito *Agent*. Ao atributo *state* é aplicada uma restrição de obrigatoriedade e uma restrição que indica que o atributo pode assumir os seguintes valores: *created*, *execution*, *ready*,

blocked e *finished*. A seguir, são apresentadas as restrições aplicadas ao atributo *name* do conceito *Agent* e a restrição que verifica os valores do atributo *state*:

context Agent inv MandatoryAgentName: self.name.size()>0

context Agent inv UniqueAgentName: Agent.allInstances -> forAll(other|self.name = other.name implies self = other)

context Agent invAgentState: Agent.allInstances->forAll(self.state = 'created' xor self.state= 'execution' xor self.state = 'ready' xor self.state = 'blocked' xor self.state = 'finished')

As demais restrições de obrigatoriedade e unicidade de atributos seguem o mesmo padrão (uma lista completa das restrições aplicadas ao meta-modelo é apresentada no Apêndice I).

O conceito *Agent* possui os seguintes relacionamentos:

- **Agent has Resource**: um agente usa zero ou mais recursos de determinado tipo para auxiliar no alcance de seus objetivos. Um recurso é usado por um ou mais agentes.
- **Agent starts InternalEvent**: um agente dispara zero ou mais eventos internos. Estes podem ser disparados no instante em que os *clocks* dos mesmos coincidirem com o tempo atual do sistema ou mesmo sem nenhuma condição associada. Um evento interno é disparado por um agente.
- **Agent has Belief**: um agente contém zero ou mais crenças que armazenam seu conhecimento. Uma crença está contida em zero ou mais agentes.
- **Agent has Perceptron**: um agente contém um ou mais *perceptrons* que avaliam as mensagens recebidas do ambiente. Um *perceptron* está contido em um agente.
- **Agent plays Role**: um agente exerce um ou mais papéis relacionados a sociedades. Um papel é exercido por um ou mais agentes.

- **Resource:** *name*, atributo alfanumérico que identifica um recurso; *type*, atributo alfanumérico que descreve o tipo de um recurso; *value*, atributo alfanumérico que define o valor de um recurso. Por exemplo, um recurso pode ser representado pela seguinte linha de código: “String pedido = pedido1”, assim “pedido” define o nome do recurso, “String” define o tipo do recurso e “pedido1” define o valor do recurso. Ao atributo *name* é aplicada uma restrição de obrigatoriedade e uma

restrição de unicidade. Ao atributo *type* é aplicada apenas uma restrição de obrigatoriedade. O conceito *Resource* possui os seguintes relacionamentos:

- **Agent has Resource**.

- **Role:** *name*, atributo alfanumérico que identifica um papel; *society*, atributo alfanumérico que identifica a sociedade em que um papel é exercido. Ao atributo *name* é aplicada uma restrição de integridade e uma restrição de unicidade. A restrição de unicidade aplicada a esse atributo se difere das demais, visto que o atributo *name* de *Role* deve ser único apenas em uma mesma sociedade. Dessa forma, essa restrição pode ser expressa da seguinte maneira:

  context Role inv UniqueSocietyName:Role.allInstances->forAll(other | self.society = other.society implies self=other xor self.name <> other.name)

Ao atributo *society* é aplicada apenas uma restrição de obrigatoriedade. O conceito *Role* possui os seguintes relacionamentos:

- **Agent plays Role**.
- **Role aims Goal**: um papel almeja o alcance de um ou mais objetivos. Um objetivo é almejado por um papel.
- **Role must execute Action**: um papel deve executar zero ou mais ações. Uma ação deve ser executada por zero ou um papel.
- **Role can execute Action**: um papel pode executar zero ou mais ações. Uma ação pode ser executada por zero ou um papel.

Nos dois últimos relacionamentos é aplicada uma restrição de integridade que indica que um papel pode ou deve executar pelo menos uma ação, conforme a seguir:

context Role inv Actions:
self.action->notEmpty() or
self.mandatoryAction->notEmpty()

Nessses relacionamentos também é aplicada uma restrição de integridade que indica que as ações de um plano que alcança um objetivo almejado por um papel devem estar dentre as ações que o papel pode ou deve executar:

context Role inv ActionsPlan:
(self.action->union(self.mandatoryAction))->includesAll(self.goal.plan.action)

- **Goal:** *name***,** atributo alfanumérico que identifica um objetivo; *state***,** atributo alfanumérico que define o estado necessário para que um plano alcance esse objetivo. Esse estado é representado pelas crenças que o agente possui. Ao atributo *name* é aplicada uma restrição de obrigatoriedade e uma restrição de unicidade. Ao atributo *state* é aplicada apenas uma restrição de obrigatoriedade. O conceito *Goal* possui os seguintes relacionamentos:
  - **Role aims Goal**.
  - **Goal aggregates Goal**: um objetivo agrega zero ou mais subobjetivos, que também são objetivos. Um objetivo é agregado por zero ou um objetivo.
  - **Plan achieves Goal**: um plano alcança um ou mais objetivos. Um objetivo é alcançado por um ou mais planos.

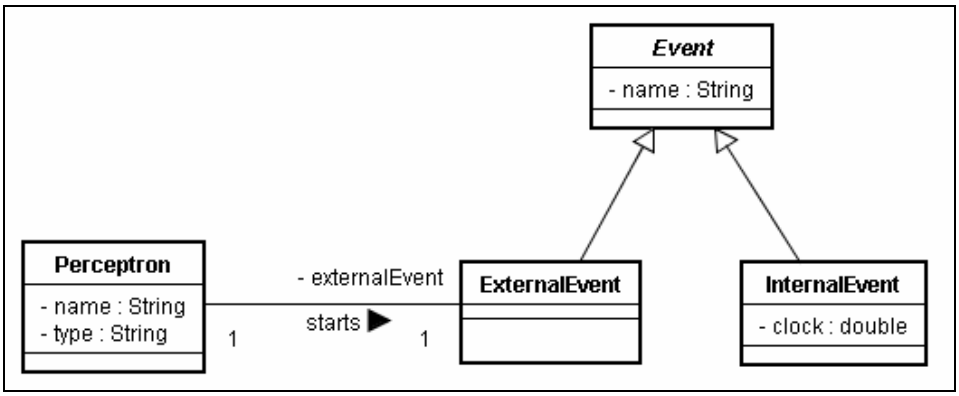


Figura 3.5 – Pacote *Sensorial*.

Detalhamento dos atributos do pacote *Sensorial*:

- **Perceptron:** *name*, atributo alfanumérico que identifica um *perceptron*; *type*, atributo alfanumérico que define o padrão de mensagens aceita por um *perceptron*. Ao atributo *name* é aplicada uma restrição de obrigatoriedade e uma restrição de unicidade. Ao atributo *type* é aplicada apenas uma restrição de obrigatoriedade. O conceito *Perceptron* possui os seguintes relacionamentos:
  - **Agent has Perceptron**.
  - **Perceptron starts ExternalEvent**: um *perceptron* dispara um evento externo. Um evento externo é disparado por um *perceptron*.
  - **Perceptron evaluates Message**: um *perceptron* avalia uma ou mais mensagens. Uma mensagem é avaliada por um *perceptron*.

- **Event:** *name*, atributo alfanumérico que identifica um evento. Ao atributo *name* é aplicada uma restrição de obrigatoriedade e uma restrição de unicidade. O conceito *Event* possui os seguintes relacionamentos:
  - **InternalEvent extends Event**: um evento interno especializa um evento.
  - **ExternalEvent extends Event**: um evento externo especializa um evento.
  - **Event generates Belief**: um evento gera uma ou mais crenças. Uma crença é gerada por zero ou um evento.

- **InternalEvent:** *clock*, atributo numérico que define o instante de tempo que um evento interno será disparado. O conceito *InternalEvent* possui os seguintes relacionamentos:
  - **Agent starts InternalEvent**.
  - **InternalEvent extends Event**.

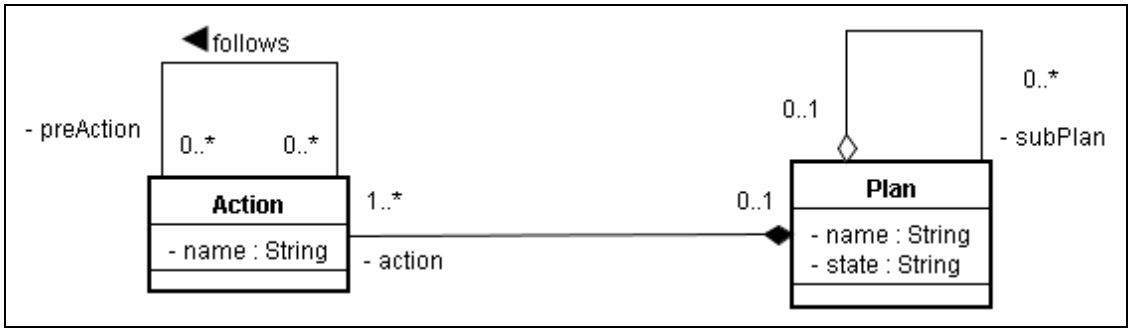


Figura 3.6 – Pacote *Executor*.

Detalhamento dos atributos do pacote *Executor*:

- **Action:** *name*, atributo alfanumérico que identifica uma ação. Ao atributo *name* é aplicada uma restrição de obrigatoriedade e uma restrição de unicidade. O conceito *Action* possui os seguintes relacionamentos:
  - **Role must execute Action**.
  - **Role can execute Action**.
  - **Plan is composed by Action**: um plano é composto por uma ou mais ações. Uma ação compõe zero ou um plano.

- **Action generates Belief**: uma ação gera uma ou mais crenças, sendo tratadas como pós-condições dessa. Uma crença é gerada por zero ou uma ação.
- **Belief controls Action**: uma crença regula zero ou mais ações, sendo tratada como pré-condição destas. Uma ação é regulada por zero ou mais crenças.
- **Action publishes Message**: uma ação publica zero ou mais mensagens no ambiente. Uma mensagem é publicada por uma ação.
- **Action follows Action**: uma ação posterior sucede zero ou mais ações. Uma ação anterior precede zero ou mais ações.

- **Plan:** *name*, atributo alfanumérico que identifica um plano; *state*, atributo alfanumérico que descreve o estado atual da execução de um plano. Esse estado pode ser representado por crenças do agente ou ainda por ações que estão sendo executadas em um dado instante. Ao atributo *name* é aplicada uma restrição de obrigatoriedade e uma restrição de unicidade. Ao atributo *state* é aplicada apenas uma restrição de obrigatoriedade. O conceito *Plan* possui os seguintes relacionamentos:
  - **Plan achieves Goal**.
  - **Plan is composed by Action**.
  - **Plan aggregates Plan**: um plano agrega zero ou mais subplanos, que também são planos. Um plano é agregado por zero ou um plano.
  - **Belief controls Plan**: uma crença regula zero ou mais planos, sendo tratada como pré-condição destes. Um plano é regulado por zero ou mais crenças.

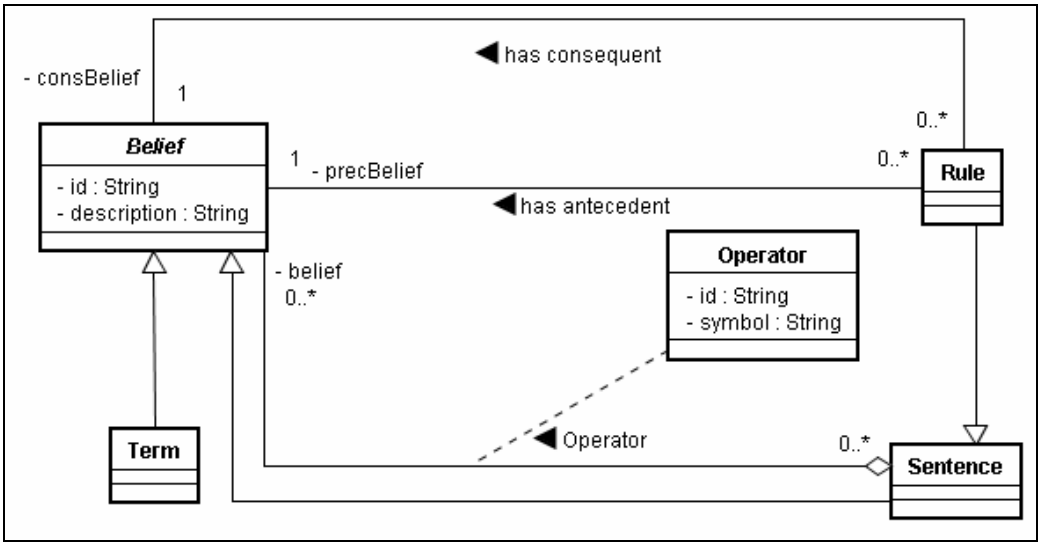


Figura 3.7 – Pacote *Decision*.

Detalhamento dos atributos do pacote *Decision*:

- **Belief:** *id*, atributo alfanumérico que identifica uma crença; *description*, atributo alfanumérico que descreve uma crença. Ao atributo *id* é aplicada uma restrição de obrigatoriedade e uma restrição de unicidade. Ao atributo *description* é aplicada apenas uma restrição de obrigatoriedade. O conceito *Belief* possui os seguintes relacionamentos:
  - **Agent has Belief**.
  - **Event generates Belief**.
  - **Action generates Belief**.
  - **Belief controls Action**.
  - **Belief controls Plan**.
  - **Rule has antecedent Belief**: uma regra tem uma crença como antecedente. Uma crença é antecedente de zero ou mais regras.
  - **Rule has consequent Belief**: uma regra tem uma crença como conseqüente. Uma crença é conseqüente de zero ou mais regras.
  - **Sentence extends Belief**: uma sentença especializa uma crença.
  - **Term extends Belief**: um termo especializa uma crença.
  - **Sentence Operator Belief:** uma sentença agrega zero ou mais crenças com o uso de uma classe associativa *Operator*. Uma crença é agregada por zero ou mais sentenças com o uso de uma classe associativa *Operator*.

- **Operator:** *id***,** atributo alfanumérico que identifica um operador; *symbol*, atributo alfanumérico que representa o conetivo lógico associado ao conceito *Operator***,** podendo assumir os valores $\neg$, $\wedge$, $\vee$ e $\Rightarrow$. . Ao atributo *id* é aplicada uma restrição de obrigatoriedade e uma restrição de unicidade. Enquanto que ao atributo *symbol* é aplicada apenas uma restrição de obrigatoriedade. O conceito *Operator* possui os seguintes relacionamentos:
  - **Sentence Operator Belief**.

- O pacote *Decision* possui o relacionamento detalhado a seguir:
  - **Rule extends Sentence**: uma regra especializa uma sentença.

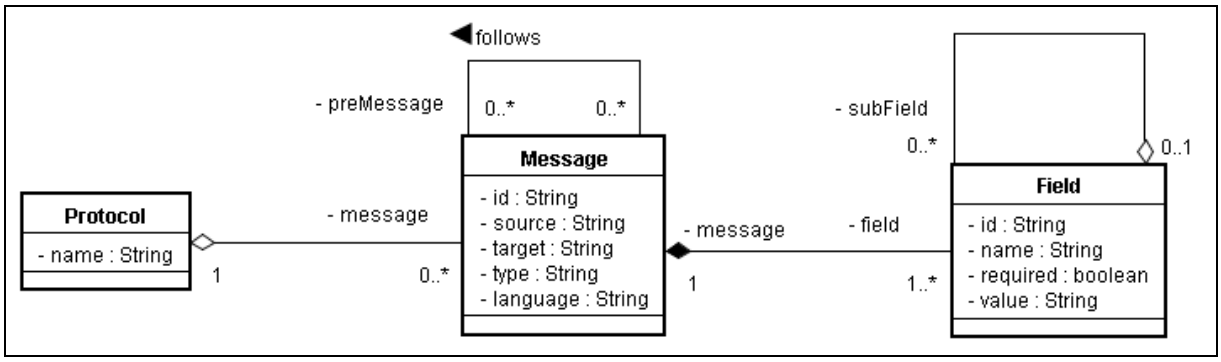


Figura 3.8 – Pacote *Communication*.

Detalhamento dos atributos do pacote *Communication*:

- **Protocol:** *name*, atributo alfanumérico que identifica o nome de um protocolo de comunicação usado pelo agente. Ao atributo *name* é aplicada uma restrição de obrigatoriedade e uma restrição de unicidade. O conceito *Protocol* possui os seguintes relacionamentos:
  - **Protocol aggregates Message**: um protocolo agrega zero ou mais mensagens. Uma mensagem é agregada por um protocolo.

- **Message:** *id***,** atributo alfanumérico usado como identificador de uma mensagem; *source*, atributo alfanumérico usado para identificar o agente emissor de uma mensagem; *target***,** atributo alfanumérico usado para identificar o agente receptor de uma mensagem; *type*, atributo alfanumérico usado para identificar o tipo de mensagem ou performativo de determinado protocolo correspondente a uma mensagem; *language*, atributo alfanumérico usado para identificar a linguagem que está sendo utilizada para a representação de uma mensagem. Ao atributo *id* é aplicada uma restrição de obrigatoriedade e uma restrição de unicidade. Aos atributos *source*, *target*, *type* e *language* é aplicada apenas uma restrição de obrigatoriedade. O conceito *Message* possui os seguintes relacionamentos:
  - **Perceptron evaluates Message**.
  - **Action publishes Message**.
  - **Protocol aggregates Message**.
  - **Message is composed by Field**: uma mensagem é composta por um ou mais campos. Um campo compõe uma mensagem.

- **Message follows Message**: uma mensagem posterior sucede zero ou mais mensagens. Uma mensagem anterior precede zero ou mais mensagens.

- **Field:** *id,* atributo alfanumérico que identifica um campo; *name***,** atributo alfanumérico que descreve o nome de um campo; *required***,** atributo booleano que define se um campo é obrigatório ou não para determinado tipo de mensagem; *value***,** atributo alfanumérico que define o valor de um campo. Ao atributo *id* é aplicada uma restrição de obrigatoriedade e uma restrição de unicidade. Ao atributos *name* é aplicada apenas uma restrição de obrigatoriedade. Ao atributo *required* é aplicada uma restrição de obrigatoriedade e uma restrição que indica que esse atributo pode assumir o valor ‘*True’* ou *‘False’*, conforme a seguir:

  context Field inv FieldRequired: self.required = 'True' xor self.required = 'False’

  O conceito *Field* possui os seguintes relacionamentos:

  - **Message is composed by Field**.
  - **Field aggregates Field**: um campo agrega zero ou mais subcampos, que também são campos. Um campo é agregado por zero ou um campo.

3.3 Comparação entre os Conceitos do Meta-modelo e os Conceitos das Plataformas de Implementação

Após o refinamento do meta-modelo inicial e a aplicação das restrições de integridade ao meta-modelo refinado, se fez necessária uma avaliação para verificar se esse atingia os requisitos para os quais foi proposto. Para isso, foi realizado um estudo contemplando diferentes plataformas de implementação de sistemas multiagentes. Conforme apresentado no capítulo anterior, foram estudadas as plataformas *SemantiCore*, *Jadex*, *Jason* e *JACK*. Na Tabela 3.2, são apresentadas as relações entre os conceitos do meta-modelo e os conceitos das diferentes plataformas de implementação. A seguir são descritas essas relações:

- *Agent*: o conceito *Agent* é abordado em todas as plataformas de implementação estudadas.
- *Role*: o conceito *Role* é abordado apenas na plataforma *JACK*.
- *Goal*: o conceito *Goal* é abordado em todas as plataformas de implementação estudadas.

TABELA 3.2 – TABELA COMPARATIVA ENTRE META-MODELO E PLATAFORMAS DE IMPLEMENTAÇÃO.

Meta-modelo	*SemantiCore*	*Jason*	*JACK*	*Jadex*
Agent	X	X	X	X
Role			X	
Goal	X	X	X	X
Resource	X	X	X	X
Event		X	X	X
InternalEvent		X	X	X
ExternalEvent		X	X	X
Perceptron	X			X
Plan	X	X	X	X
Action	X	X	X	X
Decision (Belief, Term, Sentence, Rule, Operator)	X	X	X	X
Protocol				
Message	X	X	X	X
Field	X	X	X	X

- *Resource*: o conceito *Resource* é abordado em todas as plataformas de implementação estudadas.
- *Event*: o conceito *Event* é abordado na estrutura geral da maioria das plataformas estudadas. No *JACK* e no *Jadex*, esse conceito é denominado simplesmente como evento, por outro lado, no *Jason*, existe a separação explícita entre os conceitos de evento externo e evento interno. No *SemantiCore*, o conceito não é abordado explicitamente.
- *Perceptron*: o conceito *Perceptron* é abordado no *SemantiCore* com a denominação *sensor*; no *Jadex* existe uma classe *MessageEventFilter* que checa se determinado objeto é compatível com determinado evento de mensagem. Nas demais plataformas esse conceito não é abordado explicitamente.
- *Plan*: o conceito *Plan* é abordado em todas as plataformas de implementação estudadas.
- *Action*: o conceito *Action* é abordado em todas as plataformas de implementação estudadas.
- *Decision* (*Belief*, *Term*, *Sentence*, *Rule*, *Operator*): os conceitos relacionados ao pacote *Decision* do meta-modelo podem ser mapeados para crenças nas plataformas *Jason* e *JACK*, para crenças e fatos no *Jadex* e para fatos e regras no *SemantiCore*.
- *Protocol*: o conceito *Protocol* não é explicitamente abordado nas plataformas de implementação estudadas.

- *Message*: o conceito *Message* é abordado em todas as plataformas de implementação estudadas.
- *Field*: o conceito *Field* é abordado em todas as plataformas de implementação estudadas.

De acordo com o estudo realizado, também se verificou que todos os conceitos tratados nas plataformas de implementação estudadas são tratados no meta-modelo proposto. Assim, é possível o mapeamento de modelos criados com base no meta-modelo para código nas diferentes plataformas.

3.4 Considerações

Neste capítulo, primeiramente foi apresentado o meta-modelo inicial desenvolvido. Após isso, a continuação do estudo sobre a representação interna de agentes na literatura da área e nas abordagens possibilitou o refinamento aplicado aos conceitos e aos relacionamentos do meta-modelo. A aplicação de restrições de integridade *OCL* permitiu a garantia de consistência entre o meta-modelo e os modelos instanciados do mesmo. Por fim, com o estudo entre os conceitos que compõem o meta-modelo e os conceitos que compõem algumas plataformas de implementação de SMAs, foi possível identificar que todos os conceitos abordados nas plataformas de implementação estudadas estão contemplados no meta-modelo.

Uma outra importante consideração desse capítulo é a idéia de que o meta-modelo proposto pode ser utilizado tanto para a modelagem de agentes reativos quanto deliberativos, visto que na construção do mesmo foram consideradas algumas abordagens que permitem as duas formas.

No próximo capítulo será apresentado todo o processo de construção do protótipo para o uso do meta-modelo aqui definido. Além disso, serão apresentadas algumas ferramentas estudadas e uma visão das funcionalidades do protótipo.

4 IMPLEMENTAÇÃO

Neste capítulo, inicialmente serão apresentadas as ferramentas utilizadas para a construção do protótipo, permitindo a criação e validação de modelos com base no meta-modelo. Após, o processo de consistência de modelos e geração de código é detalhado, seguido do mapeamento dos conceitos e dos relacionamentos do meta-modelo para a plataforma de implementação *SemantiCore*. Neste capítulo também é detalhado o desenvolvimento do protótipo, como o mesmo pode ser estendido e são descritas quais restrições de integridade já estão cobertas no mesmo. Por fim, é explicado o padrão de representação dos modelos em *XML* utilizado pelo protótipo, assim como são apresentadas as funcionalidades do protótipo por meio da visualização de suas interfaces.

4.1 Ferramentas Utilizadas

Para o desenvolvimento do protótipo foram utilizadas basicamente três ferramentas, são elas: Java 6, como linguagem de programação, *USE*, como ferramenta para verificação das restrições de integridade e *Velocity*, como ferramenta para auxílio na geração de código. Nesta seção, será explicada a escolha da ferramenta *USE* e é apresentada uma visão geral da ferramenta *Velocity*.

4.1.1 Ferramentas para a Aplicação de Restrições de Integridade

Inicialmente, foram estudadas algumas ferramentas para a aplicação de restrições de integridade em modelos. Deste estudo, destacam-se as ferramentas: *OCL Compiler*, *Octopus* e *USE*. As mesmas são detalhadas nas seções a seguir.

4.1.1.1 OCL Compiler

O *OCL Compiler* (*OCLCUD*) é um compilador *OCL* escrito na linguagem Java pela Universidade de Dresden. Segundo [TOV07], existem duas maneiras de trabalhar com a ferramenta. *OCLCUD* pode ser usado independentemente como um compilador *OCL* ou como uma parte da ferramenta *Argo/UML*. As principais características dessa ferramenta são a checagem sintática e semântica de expressões *OCL* e a possibilidade de gerar código Java e SQL dessas expressões. A versão suportada pela ferramenta é a *OCL 2.0*, embora pequenas

mudanças tenham sido introduzidas pelos criadores da mesma com o objetivo de resolver inconsistências encontradas na especificação *OCL*.

Um modelo *UML* deve ser carregado para a análise de restrições e pode ser obtido de um arquivo *XMI* gerado com o uso da ferramenta *Argo/UML*. Além dos módulos de *Parser* e Geração de Código, o *OCLCUD* é composto pelos módulos de Análise Semântica e Normalização.

O módulo de Análise Semântica é responsável por checar a consistência de expressões e por executar checagem de tipos. A checagem de consistência nessa ferramenta é a verificação da compatibilidade de restrições. Isso confirma que expressões têm um contexto e que essas são corretas de acordo com o modelo carregado.

A Normalização é a atividade executada antes da Geração de Código, sendo aplicada a uma árvore de sintaxe abstrata obtida na análise. A função principal desse módulo é reduzir a complexidade do gerador de código. A Geração de Código é uma das mais interessantes características dessa ferramenta.

Existem três interfaces entre o *OCLCUD* e seu ambiente (por exemplo, uma ferramenta *CASE*), são elas:

- *OCL Constraint*: o compilador é invocado.
- *Model Information*: informação de um modelo *UML* por meio de um arquivo *XMI*.
- *Target code interface*: o compilador gera informação sobre a restrição (por exemplo, quando uma restrição sob análise se refere a uma invariante ou a uma operação).

A ferramenta é publicada sob a licença *GNU Lesser General Public License* (*LGPL*) e está disponível em [DRE07] juntamente com o código fonte.

4.1.1.2 *Octopus*

[WAR07] afirma que a ferramenta *OCL Tool for Precise UML Specifications* (*Octopus*) é capaz de checar de forma estática a sintaxe de expressões *OCL*, assim como tipos de expressões e o uso correto de elementos do modelo como papéis de associação e atributos. Essa ferramenta suporta o padrão *OCL 2.0* em sua totalidade e foi construída por *Klasse Objecten*.

Todas as novas construções de *OCL 2.0*, como regras de derivação e especificações de valores iniciais, são completamente suportadas. Além disso, *Octopus* oferece a possibilidade de visualizar expressões em uma sintaxe SQL. A semântica das expressões originais, escritas na sintaxe padrão, ficam totalmente intactas, enquanto a aparência se torna mais familiar para aqueles que trabalham com banco de dados.

O *Octopus* é capaz de gerar um protótipo de três camadas completo a partir de um modelo *UML*/*OCL*. A camada intermediária consiste em *Plain Old Java Objects* (*POJOs*). Os *POJOs* são objetos Java que seguem uma estrutura simplificada em contraposição aos *Enterprise JavaBeans* (*EJBs*). Esses *POJOs* incluem código para a checagem de invariantes e multiplicidades do modelo. Expressões *OCL* que definem o corpo de uma operação são transformadas no corpo do método Java correspondente. Regras de derivação e especificações de valores iniciais também são transformadas. Opcionalmente, podem ser criados métodos para cada classe.

A camada de armazenamento consiste em um *reader* (leitor) e um *writer* (escritor) *XML* dedicados ao modelo *UML*/*OCL*. Essa camada armazena e recupera qualquer dado da aplicação protótipo em um arquivo *XML*. O *reader* lerá os conteúdos do arquivo *XML* e irá gerar objetos para quaisquer classes, atributos e *association ends* (definem as regras sobre como os objetos de classes participam em uma associação) que estão no modelo.

A camada de interface do usuário apresenta todas as instâncias do sistema, assim, as instâncias de um modelo *UML*/*OCL* podem ser criadas e examinadas. A ferramenta também possibilita a checagem das invariantes ou multiplicidades de uma instância.

A geração de código do *Octopus* considera a visão *Model Driven Architecture* (*MDA*). Ou seja, uma aplicação pode ser construída pela criação de um modelo independente de plataforma e transformado em código dependente de plataforma. Todavia, a aplicação gerada é chamada de protótipo, pois existe a necessidade de melhorar as ferramentas de transformação de código. Maiores informações sobre o *Octopus* podem ser vistas em [WAR07].

4.1.1.3 *USE*

[TOV07] afirma que a aplicação *UML-based Specification Environment* (*USE*) tem sido desenvolvida em Java por Mark Richters da Universidade de Bremen.

Segundo [USE07], o *USE* é um sistema para a especificação de sistemas de informação. Uma especificação do *USE* contêm uma descrição textual (classes, associações,

atributos, operações e restrições) de Diagramas de Classes *UML* previamente carregados. A descrição textual do modelo é própria da ferramenta e não adere a nenhum padrão.

O *USE* pode ser utilizado para validar um modelo de acordo com os requisitos do sistema. Estes são representados por meio de restrições *OCL*, que podem ser avaliadas durante instanciação do modelo. Novas restrições também podem ser introduzidas em *OCL*, e aplicadas ao modelo carregado.

A informação do sistema é apresentada por diversas visões gráficas. Podem ser visualizadas as características dos modelos, o diagrama de objetos, as invariantes de classes, a evolução dos estados, as propriedades de objetos e até uma chamada da pilha.

Uma vez carregado o modelo, necessita-se criar objetos e iniciar os atributos. Após isso, podem ser estabelecidas as associações entre os objetos. Em cada instante, ocorre uma mudança no estado do sistema, assim uma checagem automática das invariantes de classes é executada. Por exemplo, quando um novo objeto é introduzido, esse é checado para verificar se o mesmo respeita o modelo e suas restrições.

O analisador de expressões *OCL* incluído no *USE*, incorpora checagem de tipos, assim como a consistência de restrições. O analisador detecta expressões *OCL* que usam classes, atributos, operações ou associações que não são definidas no modelo. Por outro lado, restrições contraditórias não são detectadas.

Um novo aspecto importante dessa ferramenta é a validação de pré-condições e pós-condições de *OCL*. Por fim, deve-se notar que essa ferramenta é capaz de visualizar e avaliar operações de diagramas de seqüência. O *USE 2.3.1* está disponível em [USE07] e pode ser livremente distribuído por *LGPL*.

4.1.1.4 *Considerações sobre a Escolha da Ferramenta para a Elaboração do Protótipo*

Analisadas algumas ferramentas existentes atualmente para a aplicação de restrições de integridade em modelos, decidiu-se utilizar a ferramenta *USE*. Além de preencher os requisitos básicos necessários para o protótipo deste trabalho (no caso, a compilação de restrições *OCL* e a possibilidade de verificação de consistência entre o meta-modelo e seus modelos), a ferramenta tem boa usabilidade e permite a integração com diversas aplicações por ter código aberto escrito em Java.

4.1.2 Geração de Código

Para auxiliar na geração de código por meio do protótipo desenvolvido foi utilizada a ferramenta *Velocity* [VEL07]. O *Velocity* é um projeto da *Apache Software Foundation*, carregado com a criação e a manutenção de software *open-source* relativos à *Apache Velocity Engine*. Todos os softwares criados no projeto *Velocity* estão disponíveis sob a Licença de Software Apache e são livres para o público. O *Velocity* é escrito 100% em Java e pode ser facilmente embutido em aplicações.

O *Apache Velocity* oferece os seguintes projetos:

- *Velocity Engine*: é um mecanismo de *templates* de código aberto, permitindo o uso de uma linguagem de *templates* para referenciar objetos definidos em código Java. O *Velocity Engine* é o núcleo do Projeto *Apache Velocity*.
- *Velocity Tools*: esse projeto contém ferramentas e outras infra-estruturas úteis para construir aplicações usando o *Velocity Engine*.
- *Velocity DVSL*: *Declarative Velocity Style Language (DVSL)* é uma ferramenta destinada para transformações *XML* usando a *Velocity Template Language (VTL)* como linguagem de transformação.

Para a integração, o *Velocity Engine* inclui um número de tarefas pré-definidas disponíveis em [ANT07]. A integração em aplicações Web e *frameworks* está disponível em [VEL07a].

4.2 Processo de Consistência de Modelos e Geração de Código

Nesta seção, serão descritos os passos utilizados para a definição do processo de consistência de modelos e geração de código entre o meta-modelo e as diferentes plataformas de implementação de SMAs. Inicialmente, o protótipo desenvolvido deve receber como entrada três arquivos, são eles: um arquivo *XMI* representando o meta-modelo (no protótipo foi utilizado um arquivo *XMI* gerado pela ferramenta *Argo/UML*); um arquivo *OCL*, descrevendo todas as restrições aplicadas ao meta-modelo; e um arquivo *XML* representando um modelo de aplicação criado, sendo que este é construído em um padrão de representação de modelos proprietário (detalhado na seção 4.7) e pode ser gerado com o uso do protótipo. A Figura 4.1 apresenta o processo de consistência de modelos e geração de código.

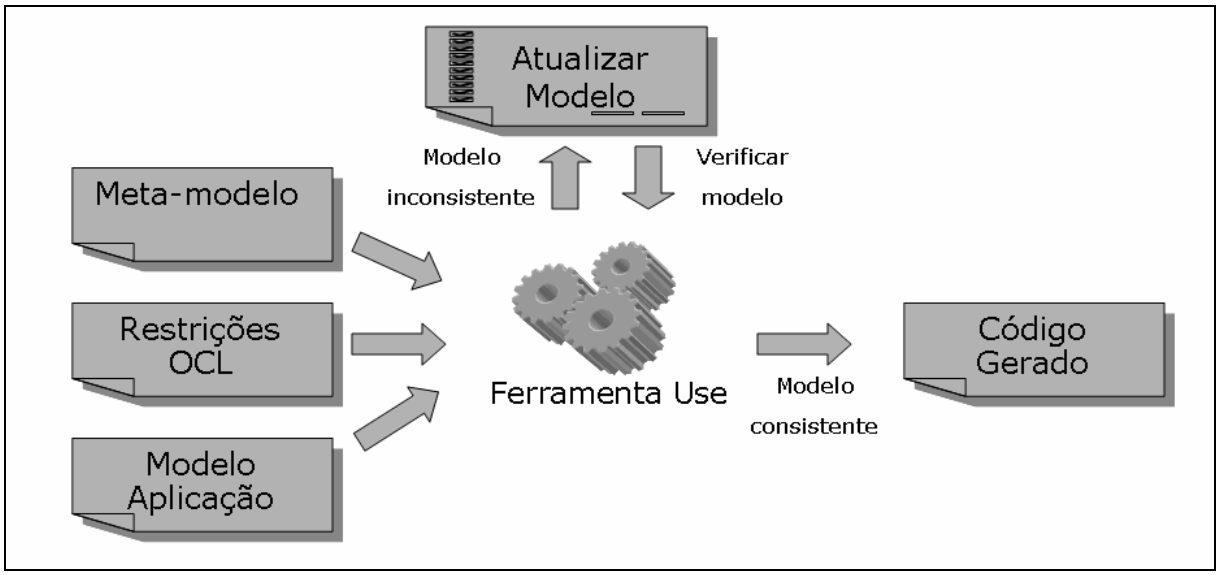


Figura 4.1 – Processo de Consistência de Modelos e Geração de Código.

O processo se divide basicamente em três passos incrementais, são eles: verificação das restrições *OCL*, verificação da consistência entre um modelo de aplicação e as restrições aplicadas ao meta-modelo e, por fim, geração de código a partir de um modelo de aplicação.

No primeiro passo, o protótipo recebe como entradas o arquivo *XMI* do meta-modelo e o arquivo *OCL* das restrições de integridade. Esses dois arquivos são transformados por um *parser* do protótipo em um arquivo *USE*. Com isso, o arquivo *USE* gerado poderá ser usado como entrada da ferramenta *USE* que verificará se as restrições aplicadas ao meta-modelo estão escritas de maneira correta.

Uma vez realizado o primeiro passo, pode-se verificar se o modelo de uma aplicação está consistente com o meta-modelo. Para isso, o protótipo usa um modelo de aplicação representado por um arquivo *XML* em um formato proprietário. Esse arquivo será transformado em um arquivo do *CMD* que será utilizado como entrada da ferramenta *USE*, instanciando o meta-modelo com o modelo de uma aplicação. Com isso, a ferramenta *USE* terá a representação do meta-modelo e suas restrições (por meio do arquivo *USE* gerado no primeiro passo) e a representação de um modelo de aplicação (por meio do arquivo *CMD*). Desta forma, a ferramenta *USE* pode verificar se um modelo de aplicação está consistente com o meta-modelo definido. Caso esteja consistente, o protótipo permitirá a geração de código do modelo em uma dada plataforma de implementação. Em caso negativo, o modelo deve ser atualizado até que esteja de acordo com o meta-modelo e suas restrições.

O terceiro passo consiste na geração de código em uma plataforma de implementação de SMA. Basicamente, este passo recebe como entrada o arquivo *XML* de um modelo e o

transforma em código fonte proprietário de uma plataforma de implementação por meio de um *parser* (ponto de flexibilidade do protótipo) definido pelo usuário do meta-modelo.

4.3 Mapeamento do Meta-modelo para o SemantiCore

Nesta seção, será apresentado o mapeamento criado entre os conceitos e relacionamentos do meta-modelo e os elementos da plataforma *SemantiCore*. Esse mapeamento foi necessário para a correta geração de código a partir do uso de um modelo de aplicação definido pelo usuário do meta-modelo. No decorrer da seção serão apresentados pequenos trechos de código com o objetivo de facilitar a visualização de parte do mapeamento. O código completo gerado será apresentado na seção 5.1.2 e no Apêndice III. Na descrição do mapeamento dos relacionamentos, não será apresentado o código caso o mesmo já tenha sido apresentado na descrição do mapeamento de algum conceito. Sendo assim, o mapeamento dos conceitos é descrito a seguir:

- *Agent*: um conceito *Agent* foi mapeado para uma extensão da classe *SemanticAgent*. O atributo *name* foi mapeado para o nome da extensão; o atributo *state* não foi mapeado diretamente, pois o *SemantiCore* trata o estado do agente internamente; o atributo *environment* foi mapeado para o atributo *environment* passado como argumento de um *SemanticAgent* em um arquivo *semanticoreconfig.xml* que instancia os agentes na plataforma. O trecho a seguir exemplifica parte do mapeamento:

  public class Cliente extends SemanticAgent

- *Role*: um conceito *Role* não foi mapeado diretamente para o *SemantiCore*, pois a plataforma não trata esse conceito.
- *Goal*: um conceito *Goal* foi mapeado para uma classe *Goal*. O atributo *name* foi mapeado para o nome de uma instância da classe; o atributo *state* não foi mapeado diretamente, pois a plataforma trata o estado dos objetivos internamente. O trecho a seguir exemplifica parte do mapeamento, onde o primeiro parâmetro indica o agente e o segundo indica o modelo de ontologia associado ao objetivo:

  Goal comprarComponentes = new Goal(this.getOwner(),null);

- *Resource*: um conceito *Resource* foi mapeado para um atributo de uma extensão da classe *SemanticAgent*. Os atributos *name*, *type* e *value* foram mapeados respectivamente para o nome, o tipo e o valor do atributo que representa o conceito *Resource*. O trecho a seguir exemplifica parte do mapeamento:

  private String nomeComponente = "Processador";

- *InternalEvent*: esse conceito não foi mapeado diretamente para o *SemantiCore*, pois a plataforma não trata o mesmo.
- *ExternalEvent*: esse conceito não foi mapeado diretamente para o *SemantiCore*, pois a plataforma não trata o mesmo.
- *Perceptron*: um conceito *Perceptron* foi mapeado para uma extensão da classe *Sensor*. O atributo *name* foi concatenado com a palavra *Sensor* e mapeado para o nome da extensão; o atributo *type* foi mapeado para o argumento do tipo *Object* passado no método *evaluate* da extensão. O trecho a seguir exemplifica parte do mapeamento:

  public class AvaliaPedidosSensor extends Sensor

- *Plan*: um conceito *Plan* foi mapeado para uma extensão da classe *ActionPlan*. O atributo *name* foi mapeado para o nome da extensão; o atributo *state* não foi mapeado diretamente para o *SemantiCore*, pois o estado de um plano é tratado internamente na plataforma. O trecho a seguir exemplifica parte do mapeamento:

  public class EfetuarCompra extends ActionPlan

- *Action*: um conceito *Action* foi mapeado para uma extensão da classe *Action*. O atributo *name* foi mapeado para o nome da extensão. O trecho a seguir exemplifica parte do mapeamento:

  public class EnviarPedido extends Action

- *Term*: um conceito *Term* foi mapeado para uma classe *SimpleFact*. O atributo *id* foi mapeado para o nome de uma instância da classe e o atributo *description* foi mapeado para os atributos sujeito, predicado e objeto passados como argumentos da instância. O trecho a seguir exemplifica parte do mapeamento:

  SimpleFact a = new SimpleFact("Pedido","Nome","Placa-Mae");

- *Sentence*: um conceito *Sentence* pode ser mapeado para uma classe *ComposedFact* ou para duas instâncias, cada uma pode ser das classes *SimpleFact*, *ComposedFact* ou *Rule*. Explicações adicionais sobre o mapeamento de um conceito *Sentence* são apresentadas no detalhamento do relacionamento *Sentence Operator Belief*. Quando o conceito *Sentence* é mapeado para uma classe *ComposedFact*, o atributo *id* é mapeado para o nome de uma instância da classe e o atributo *description* não é mapeado diretamente para o *SemantiCore*, pois não existe um atributo que permita a descrição de uma sentença. O trecho a seguir exemplifica parte de um possível mapeamento, onde *ped01* representa o nome da instância, e *a* e *b* representam *SimpleFacts* instanciados:

  ComposedFact ped01 = new ComposedFact(a,b);

- *Rule*: um conceito *Rule* foi mapeado para uma classe *Rule*. O atributo *id* foi mapeado para o nome de uma instância da classe e o atributo *description* foi mapeado para o atributo *name* da instância. O trecho a seguir exemplifica parte do mapeamento, onde *regra* representa o nome da instância, *DecisaoCompra* representa o atributo *name*, enquanto *u* e *v* representam respectivamente as crenças antecedente e conseqüente da regra:

  Rule regra = new Rule("DecisaoCompra",u,v);

- *Operator*: um conceito *Operator* não foi mapeado diretamente para o *SemantiCore*. Porém, foi considerado no mapeamento do relacionamento *Sentence Operator Belief*.
- *Message*: um conceito *Message* foi mapeado para uma extensão da classe *SemanticMessage*. O atributo *id* foi mapeado para o nome de uma instância da extensão; os atributos *source* e *target* foram mapeados respectivamente para os atributos *from* e *to*, já existentes na *SemanticMessage*; o atributo *type* foi concatenado com a palavra *Message* e mapeado para o nome da extensão; o atributo *language* foi mapeado para o atributo *language* que foi incluído na extensão. O trecho a seguir exemplifica parte do mapeamento:

  public class CfpMessage extends SemanticMessage

- *Protocol*: um conceito *Protocol* foi mapeado para um atributo que foi incluído nas extensões da classe *SemanticMessage* (conceitos *Message* associados a um conceito *Protocol*). O atributo *name* foi mapeado para o valor desse atributo. O trecho a seguir exemplifica parte do mapeamento:

  private String protocol = "ContractNet";

- *Field*: um conceito *Field* foi mapeado para um atributo que foi incluído nas extensões da classe *SemanticMessage* (conceito *Message* associado a um conceito *Field*). O atributo *id* não foi mapeado diretamente para o *SemantiCore*, pois o mesmo não necessita ser gerado em código visto que sua função é apenas permitir a associação com um conceito *Message*; o atributo *name* foi mapeado para o nome do atributo incluído; o atributo *value* foi mapeado para o valor do atributo incluído; o atributo *required* não foi mapeado diretamente, pois esse apenas indica se o valor de um campo deve ou não ser informado. O trecho a seguir exemplifica parte do mapeamento:

  private String ontology = "componentes";

Detalhado o mapeamento dos conceitos do meta-modelo para a plataforma *SemantiCore*, o mapeamento dos relacionamentos entre esses conceitos é descrito a seguir:

- *Agent has Resource*: como explicado no detalhamento do conceito *Resource*, esse conceito foi mapeado para um atributo de uma extensão da classe *SemanticAgent*. O trecho a seguir exemplifica parte do mapeamento:

```
public class Cliente extends SemanticAgent {
    private String nomeComponente;
}
```

- *Agent starts InternalEvent*: esse relacionamento não foi mapeado diretamente para o *SemantiCore*, pois o conceito *InternalEvent* não foi mapeado.
- *Agent has Belief*: um conceito *Belief* foi mapeado para uma classe *SimpleFact*, *ComposedFact* ou *Rule* criada e adicionada no método *setup* de uma extensão da classe *SemanticAgent*. O trecho a seguir exemplifica parte do mapeamento:

```
public class Cliente extends SemanticAgent {
```

```
    protected void setup ( ) {
            SimpleFact a = new SimpleFact("Pedido","Nome","Placa-Mae");
            addFact(a);   }
}
```

- *Agent has Perceptron*: esse relacionamento foi mapeado para uma chamada do método *addSensor* dentro do método *setup* de uma extensão da classe *SemanticAgent*. O trecho a seguir exemplifica parte do mapeamento:

```
public class Cliente extends SemanticAgent {
        protected void setup ( ) {
            addSensor ( new AvaliaPropostasSensor ("AvaliaPropostas") );
        }
}
```

- *Agent plays Role*: esse relacionamento não foi mapeado diretamente para o *SemantiCore*, pois o conceito *Role* não foi mapeado.
- *Role aims Goal*: esse relacionamento não foi mapeado diretamente para o *SemantiCore*, pois o conceito *Role* não foi mapeado.
- *Role must execute Action*: esse relacionamento não foi mapeado diretamente para o *SemantiCore*, pois o conceito *Role* não foi mapeado.
- *Role can execute Action*: esse relacionamento não foi mapeado diretamente para o *SemantiCore*, pois o conceito *Role* não foi mapeado.
- *Goal aggregates Goal*: esse relacionamento não foi mapeado diretamente para o *SemantiCore*, pois a plataforma não trabalha com o conceito de subobjetivos.
- *Plan achieves Goal*: um conceito *Plan* foi mapeado para o atributo *plan* da classe *Goal*. O plano é representado no terceiro argumento apresentado no trecho a seguir:

```
Goal comprarComponentes = new Goal(this.getOwner(),null,plan,null);
```

- *Plan aggregates Plan*: um conceito *Plan* foi mapeado para uma extensão da classe *ActionPlan*. Uma instância do tipo *ActionPlan* pode ser incluída em uma outra instância do mesmo tipo pelo uso do método *addAction*. O trecho a seguir exemplifica parte do mapeamento:

```
Plano plano = new Plano ("plano");
```

plano.addAction((ActionPlan)new SubPlano(“subPlano”));

- *Plan is composed by Action*: um conceito *Action* foi mapeado para uma extensão da classe *Action*. Uma instância do tipo *Action* pode ser incluída em uma instância do tipo *ActionPlan* pelo uso do método *addAction*. O trecho a seguir exemplifica parte do mapeamento:

EfetuarVenda efetuarVenda = new EfetuarVenda ("EfetuarVenda");
efetuarVenda.addAction((ActionPlan) new EnviarProposta());

- *Belief controls Plan*: um conceito *Belief* foi mapeado para o atributo *preCondition* de uma extensão da classe *Action* que inicia um *ActionPlan*. O trecho a seguir exemplifica parte do mapeamento, onde *preCondition* representa a crença e *AcaoInicial* representa a ação que inicia um plano:

SimpleFact preCondition = new SimpleFact("Pedido","Nome","Placa-Mae");
SimpleFact postCondition = new SimpleFact("Proposta","Preco","500");
new AcaoInicial (“AcaoInicial”, preCondition,postCondition);

- *Action generates Belief*: um conceito *Belief* foi mapeado para o atributo *postCondition* de uma extensão da classe *Action*. O trecho a seguir exemplifica parte do mapeamento, onde *EnviarProposta* representa a ação e *postCondition* representa a crença:

SimpleFact preCondition = new SimpleFact("Pedido","Nome","Placa-Mae");
SimpleFact postCondition = new SimpleFact("Proposta","Preco","500");
new EnviarProposta (“EnviarProposta”, preCondition,postCondition);

- *Belief controls Action*: um conceito *Belief* foi mapeado para o atributo *preCondition* de uma extensão da classe *Action*. O trecho a seguir exemplifica parte do mapeamento, onde *EnviarProposta* representa a ação e *preCondition* representa a crença:

SimpleFact preCondition = new SimpleFact("Pedido","Nome","Placa-Mae");
SimpleFact postCondition = new SimpleFact("Proposta","Preco","500");
new EnviarProposta (“EnviarProposta”, preCondition,postCondition);

- *Action publishes Message*: um conceito *Message* foi mapeado para uma extensão da classe *SemanticMessage*. A instância dessa extensão é passada como argumento do método *transmit* da extensão da classe *Action*. O trecho a seguir exemplifica parte do mapeamento:

```
CfpMessage mensagem = new CfpMessage(from, to, content);
transmit ( mensagem);
```

- *Rule has antecedent Belief*: a instância de uma classe *SimpleFact*, *ComposedFact* ou *Rule* é mapeada para o atributo *fact* da instância da classe *Rule*. O trecho de código para esse relacionamento pode ser visto no mapeamento do conceito *Rule*.
- *Rule has consequent Belief*: a instância de uma classe *SimpleFact*, *ComposedFact* ou *Rule* é mapeada para o atributo *consequence* da instância da classe *Rule*. O trecho de código para esse relacionamento pode ser visto no mapeamento do conceito *Rule*.
- *Sentence Operator Belief*: caso o atributo *symbol* de *Operator* seja igual à $\wedge$, a *Sentence* é mapeada para uma instância da classe *ComposedFact*. Caso o atributo *symbol* tenha o valor $\vee$, a *Sentence* é mapeada para duas instâncias, cada uma pode ser das classes *SimpleFact*, *ComposedFact* ou *Rule*. Por fim, caso o valor de *symbol* seja $\neg$, a crença que a *Sentence* agrega é negada e é mapeada para uma instância da classe *SimpleFact*, *ComposedFact* ou *Rule*. O trecho de código para esse relacionamento pode ser visto no mapeamento do conceito *Sentence*.
- *Perceptron starts ExternalEvent*: esse relacionamento não foi mapeado diretamente para o *SemantiCore*, pois o conceito *ExternalEvent* não foi mapeado.
- *Perceptron evaluates Message*: esse relacionamento foi mapeado para o método *evaluate* de uma extensão da classe *Sensor*. O trecho a seguir exemplifica parte do mapeamento:

```
public Object evaluate(Object arg0){
    if (arg0 instanceof CfpMessage) {          }
}
```

- *Protocol aggregates Message*: esse relacionamento foi mapeado para o atributo *protocol* que foi incluído nas extensões da classe *SemanticMessage*. O trecho a seguir exemplifica parte do mapeamento:

```
public class CfpMessage extends SemanticMessage {
```

```
    private String protocol;
}
```

- *Message is composed by Field*: esse relacionamento foi mapeado para atributos que foram incluídos nas extensões da classe *SemanticMessage*. O trecho a seguir exemplifica parte do mapeamento:

```
public class CfpMessage extends SemanticMessage {
    private String ontology;
}
```

- *Message follows Message*: esse relacionamento não foi mapeado diretamente para o *SemantiCore*, pois não é possível criar relação de ordem entre as mensagens no *SemantiCore*.
- *Action follows Action*: o valor do atributo *postCondition* de uma extensão da classe *Action* definida previamente deve ter o mesmo valor do atributo *preCondition* de uma extensão da classe *Action* definida na seqüência. O trecho a seguir exemplifica parte do mapeamento:

```
super ( "AcaoPrevia", beliefAnt, beliefCons );
super ( "AcaoPosterior", beliefCons, belief );
```

- *Field aggregates Field*: o atributo representando o subcampo no relacionamento deve ser mapeado para o atributo de um campo representado por uma classe que é um atributo de uma extensão da classe *SemanticMessage*. O trecho a seguir exemplifica parte do mapeamento:

```
public class Campo {
    private String subcampo;
}
public class Message extends SemanticMessage {
    private Campo campo;
}
```

- *Event generates Belief*: esse relacionamento não foi mapeado diretamente para o *SemantiCore*, pois o conceito *Event* não foi mapeado.

4.4 Desenvolvimento do Protótipo

Nesta seção será apresentada a estrutura geral do protótipo criado para o uso do meta-modelo definido. Os principais objetivos desse protótipo são facilitar a entrada de dados de modelos de aplicação, verificar a consistência desses modelos com o meta-modelo e possibilitar a geração de código em uma plataforma de implementação, neste caso, no *SemantiCore*. Para o protótipo, não foi escolhida nenhuma linguagem diagramática para ser suportada, pois o foco do mesmo foi no processo de consistência de modelos e geração de código. O protótipo é dividido nos seguintes pacotes principais: *application*, *constraints*, *concepts*, *gui*, *metamodel*, *parser*, *relationships*, *support*, *use* e *velocity*. Na apresentação de cada pacote, serão suprimidos atributos e métodos secundários para facilitar a leitura. A Figura 4.2 apresenta o Diagrama de Pacotes do protótipo desenvolvido e as próximas seções detalham cada pacote assim como as classes e arquivos que compõem cada um desses.

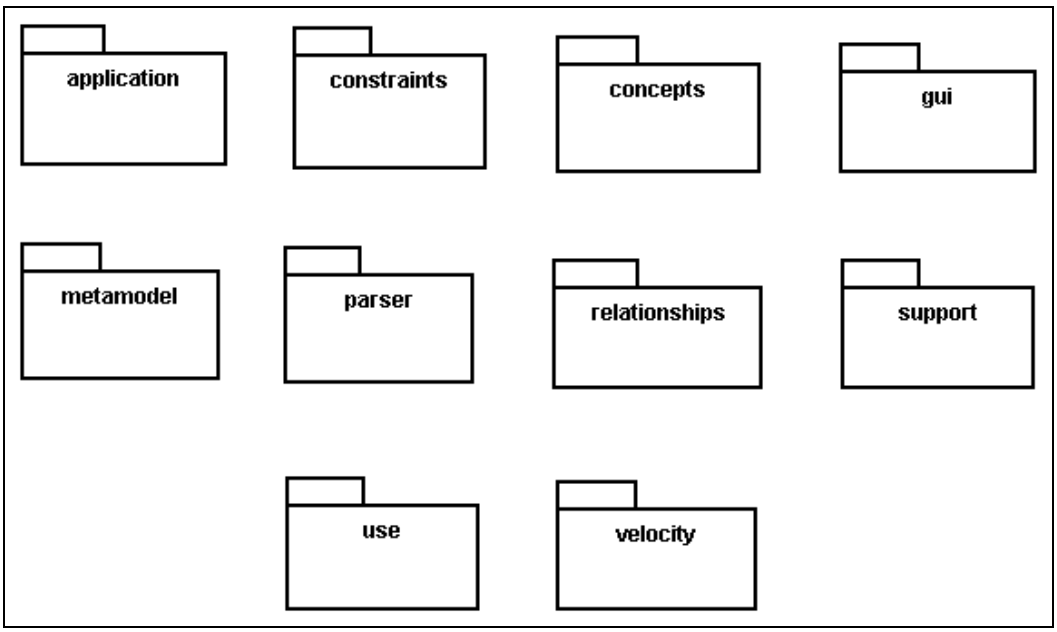


Figura 4.2 – Diagrama de Pacotes de Protótipo.

4.4.1 Pacote application

Neste pacote são armazenadas as classes geradas que representam os agentes de um modelo de aplicação na plataforma *SemantiCore*.

4.4.2 Pacote constraints

Neste pacote é armazenado o arquivo *Constraints.ocl*, contendo todas as restrições aplicáveis ao meta-modelo. Esse arquivo é transformado em uma entrada da ferramenta *USE* com o objetivo de verificar a integridade de um modelo de aplicação.

4.4.3 Pacote concepts

Neste pacote são armazenadas as classes utilizadas para guardar temporariamente os valores dos conceitos do meta-modelo durante o processo de geração de código com a plataforma de implementação. Dentre as classes que compõem esse pacote, apenas *Sentence* e *Rule* possuem uma estrutura que se difere das demais. A primeira armazena os atributos do conceito *Sentence*, além dos identificadores das crenças que essa agrega e o valor do atributo *symbol* do conceito *Operator* participante da relação de agregação. Por outro lado, a segunda armazena os atributos do conceito *Rule*, juntamente com os identificadores das crenças antecedente e conseqüente que se relacionam com a mesma. A Figura 4.3 apresenta o Diagrama de Classes *UML* desse pacote.

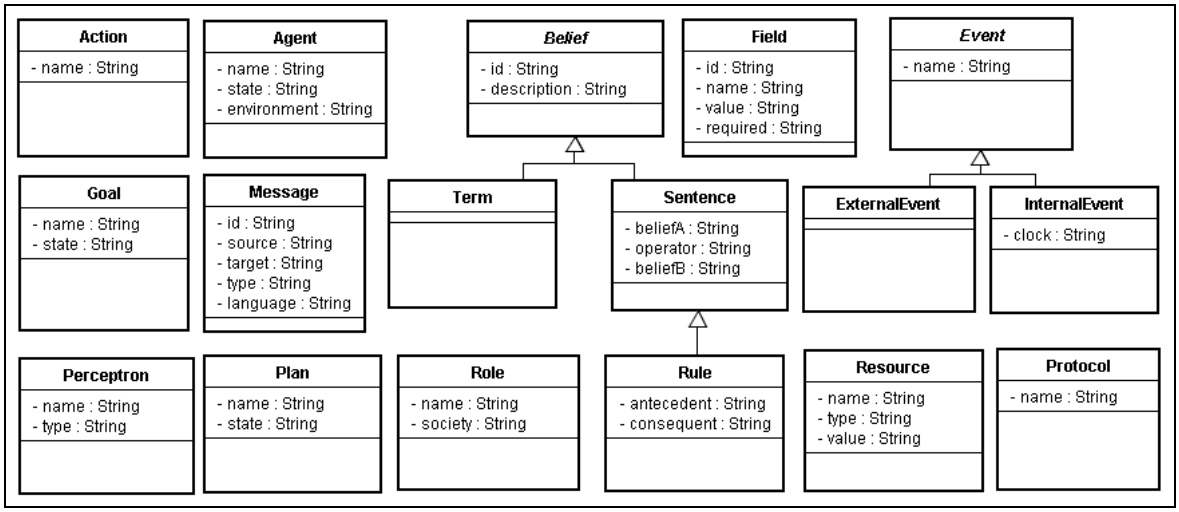


Figura 4.3 – Diagrama de Classes *UML* do pacote *concepts*.

Cada classe criada nesse pacote representa um conceito do meta-modelo. Todos os atributos de conceitos do meta-modelo, independente do tipo, foram mapeados para atributos do tipo *String* com o objetivo de facilitar a implementação.

4.4.4 Pacote gui

Neste pacote são armazenadas todas as interfaces gráficas do protótipo. Além dos subpacotes *gui.consult* e *gui.register*, o pacote é composto pelas classes *CreateModel*, *LoadModel*, *MainGui* e *UseLog*. A Figura 4.4 apresenta a estrutura geral do mesmo.

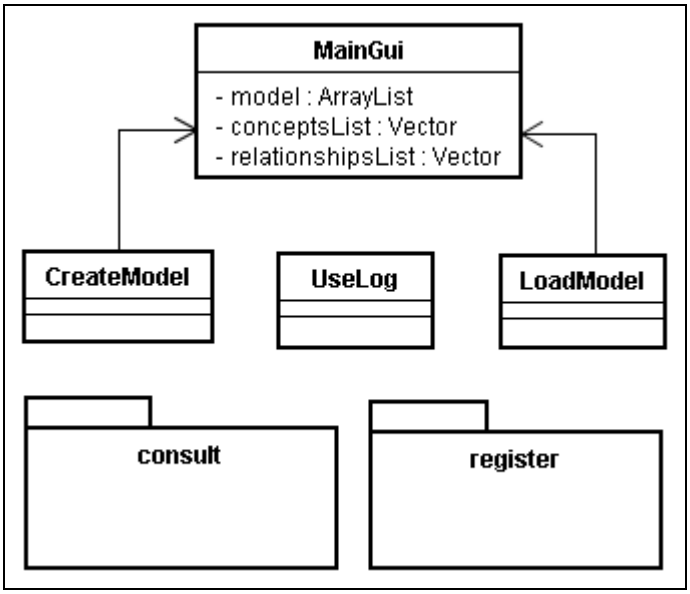


Figura 4.4 – Estrutura geral do pacote *gui*.

A classe *MainGui* representa a interface principal da aplicação. Os principais atributos dessa classe são: *ArrayList model*, *Vector conceptsList* e *Vector relationshipsList*. O primeiro atributo representa todos os dados de um modelo de aplicação corrente, o segundo representa uma lista dos conceitos criados para esse modelo e o último representa uma lista dos relacionamentos entre esses conceitos. As classes *CreateModel* e *LoadModel* possuem uma referência para *MainGui*. A primeira é responsável pelo armazenamento de um modelo em um arquivo *XML* no padrão de representação de modelos do protótipo, enquanto que a última permite o carregamento de um modelo representado por um arquivo desse mesmo tipo. Por fim, a classe *UseLog* representa a interface gráfica onde são apresentados os resultados da checagem de modelos de aplicação.

O pacote *gui* ainda é constituído pelos subpacotes *gui.register* e *gui.consult* detalhados nas próximas seções.

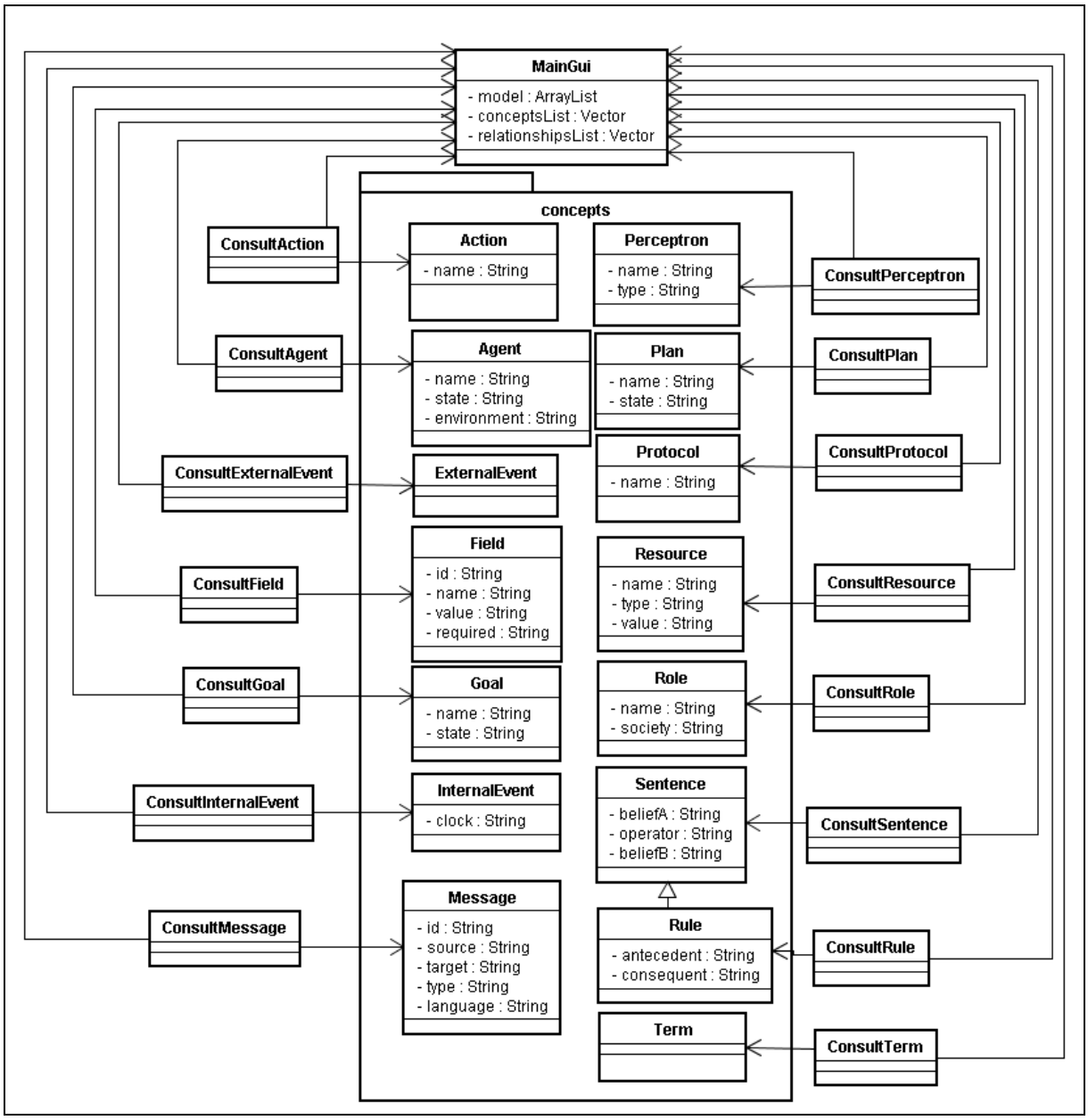


Figura 4.5 – Diagrama de Classes *UML* do pacote *gui.consult*.

4.4.4.1 *Pacote gui.consult*

Neste pacote são armazenadas todas as interfaces gráficas das consultas de conceitos do protótipo. Estas interfaces apresentam o conteúdo de cada conceito que é consultado. Cada interface possui uma referência para *MainGui* e outra para a classe que armazena o conceito (por exemplo, *ConsultAction* possui uma referência para *MainGui* e para *Action*). A Figura 4.5 apresenta o Diagrama de Classes *UML* desse pacote.

4.4.4.2 Pacote gui.register

Neste pacote são armazenadas todas as interfaces gráficas dos cadastros da aplicação. Além dos subpacotes *gui.register.concepts* e *gui.register.relationships*, o pacote é composto pelas classes *CreateConcept* e *CreateRelationship*. A primeira consiste em uma interface gráfica onde é feita a escolha do tipo de conceito que será criado e a segunda consiste em uma interface gráfica onde é feita a escolha do tipo de relacionamento entre conceitos que será criado. Ambas possuem uma referência para a classe *MainGui*. A estrutura geral do pacote *gui.register* é apresentada na Figura 4.6.

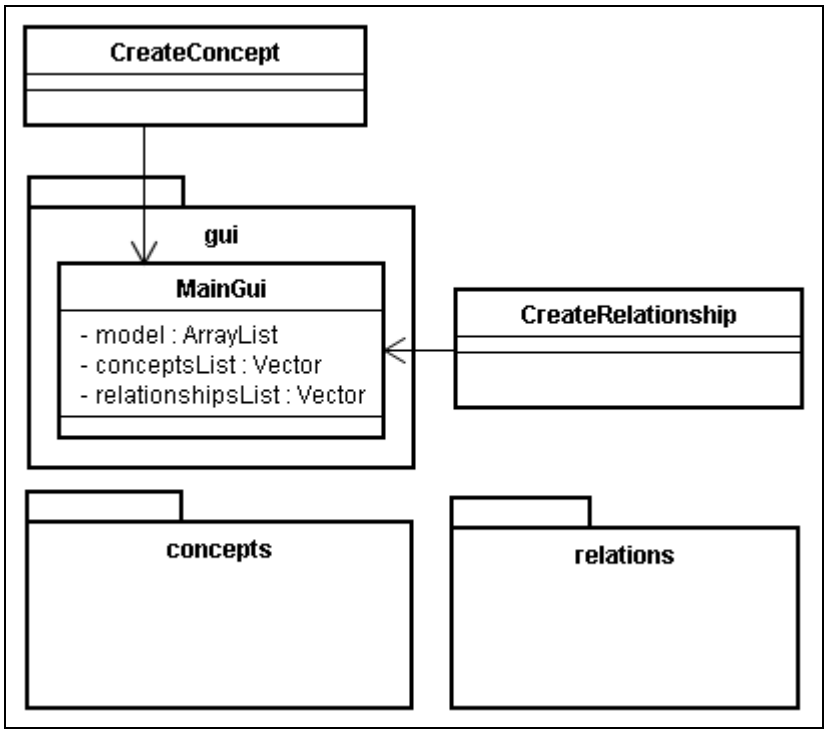


Figura 4.6 – Estrutura geral do pacote *gui.register*.

4.4.4.2.1 Pacote gui.register.concepts

Neste pacote são armazenadas todas as interfaces gráficas dos cadastros de conceitos da aplicação. Assim, em cada interface devem ser informados os dados do conceito a ser cadastrado. Cada uma das classes do pacote possui uma referência para a classe *MainGui*. A classe *CreateMessage* possui o atributo *Vector agentsList* representando a lista de conceitos *Agent* criados no modelo que poderão ser utilizados nos atributos *from* e *to* da mensagem. Por outro lado, a classe *CreateSentence* possui o atributo *Vector beliefsList* representando a lista de conceitos *Belief* criados no modelo que poderão ser agregados em um conceito *Sentence*. A

classe *CreateRule* também possui o atributo *Vector beliefsList* representando a lista de conceitos *Belief* criados no modelo que poderão ser usados como uma crença antecedente ou conseqüente de um conceito *Rule*. A Figura 4.7 apresenta o Diagrama de Classes *UML* deste pacote.

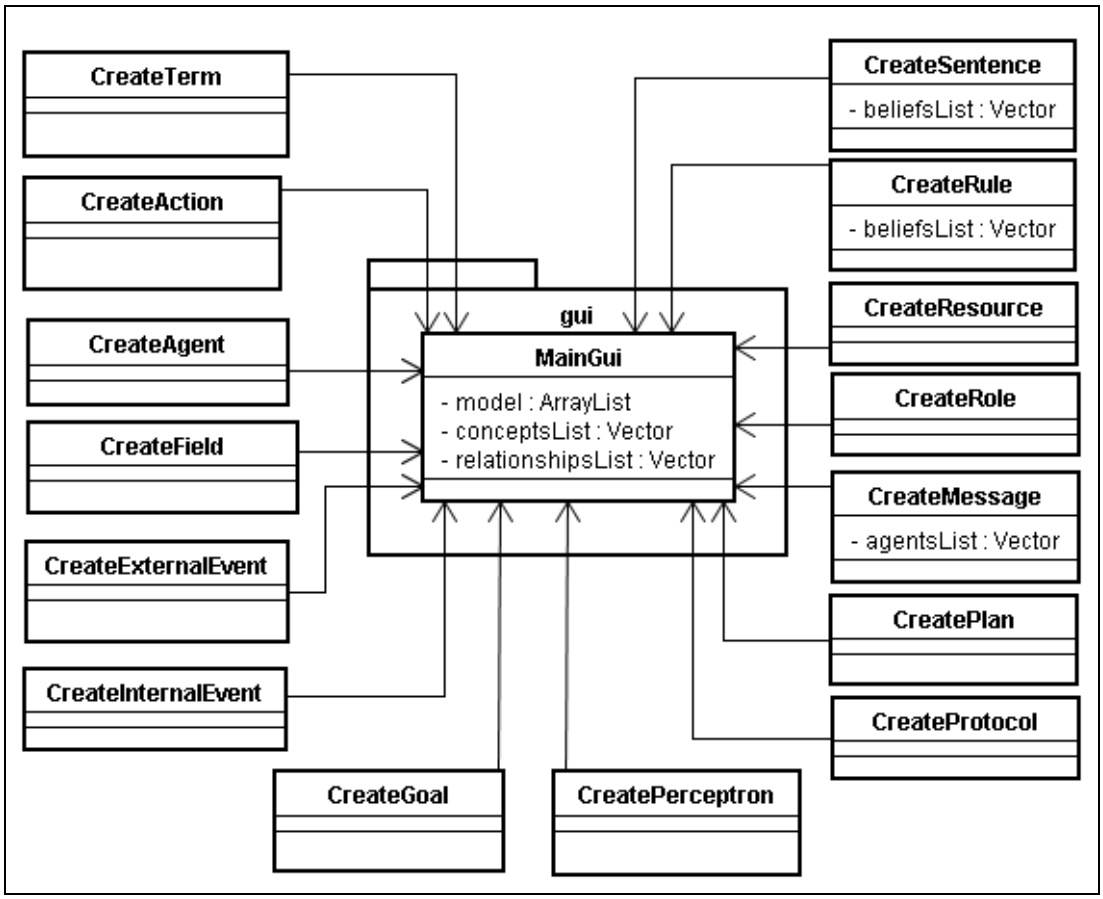


Figura 4.7 – Diagrama de Classes *UML* do pacote *gui.register.concepts*.

4.4.4.2.2 Pacote gui.register.relationships

Neste pacote são armazenadas todas as interfaces gráficas dos cadastros de relacionamentos da aplicação. Assim, em cada interface devem ser informados os dados do relacionamento a ser cadastrado. Cada uma das classes desse pacote possui uma referência para *MainGui* e uma lista para cada conceito que participa de um tipo de relacionamento. Por exemplo, a classe *ActionBelief* possui duas listas: uma de ações (*Vector actionsList*) e outra de crenças (*Vector beliefsList*). A Figura 4.8 apresenta o Diagrama de Classes *UML* desse pacote.

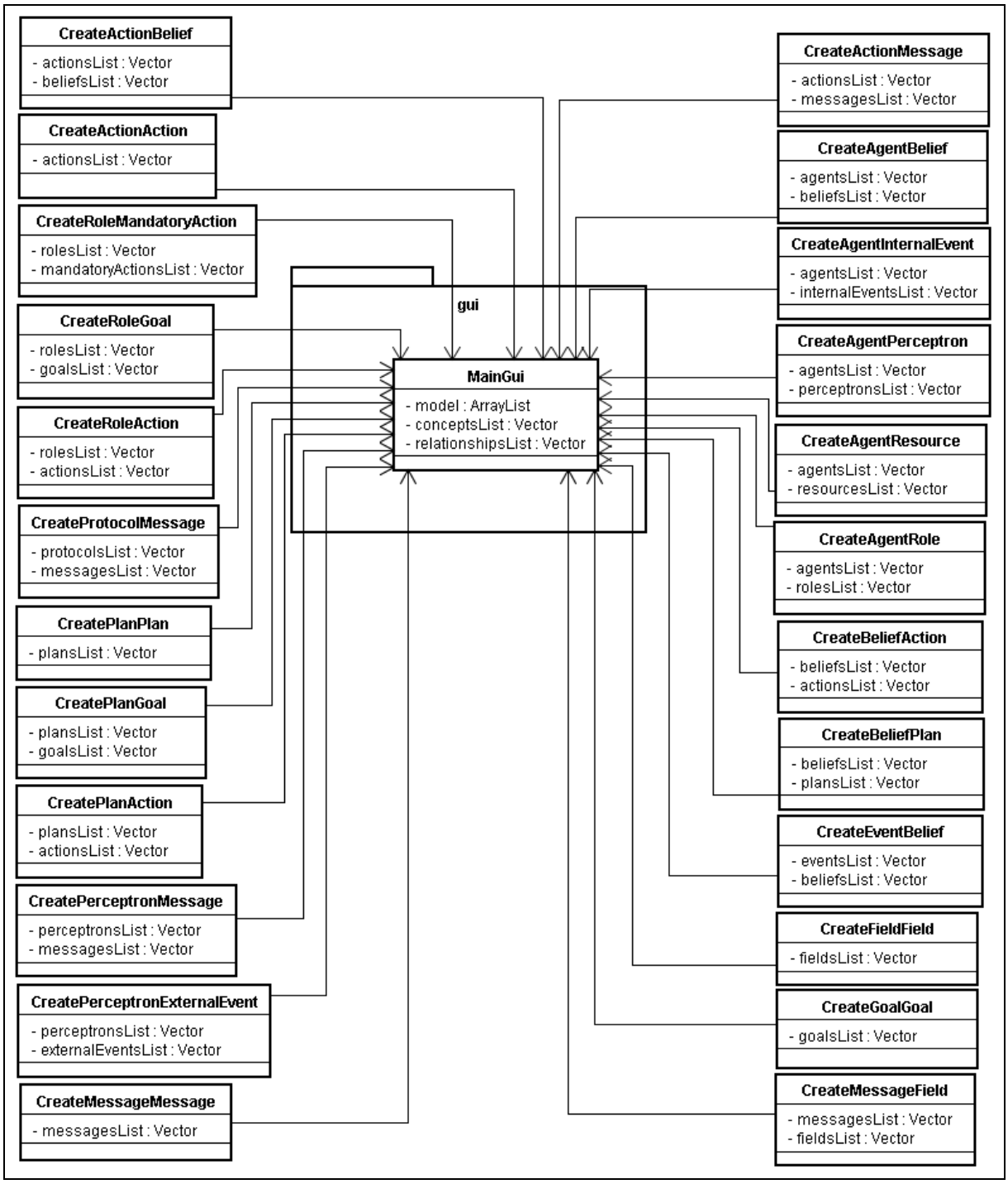


Figura 4.8 – Diagrama de Classes *UML* do pacote *gui.register.relationships*.

4.4.5 Pacote metamodel

Neste pacote é armazenado o arquivo *Metamodel.xmi*. Esse arquivo é uma representação do Diagrama de Classes *UML* do meta-modelo e é gerado como saída da ferramenta *Argo/UML*.

4.4.6 Pacote use

Neste pacote são armazenados os subpacotes *use.source* e *use.output*, além dos arquivos *Metamodel.use* e *Model.cmd*. Ambos os arquivos são utilizados como entrada da ferramenta *USE*. Sendo que o primeiro contém as definições dos conceitos, dos relacionamentos e das restrições aplicadas ao meta-modelo, enquanto o segundo representa o modelo de uma aplicação instanciada do meta-modelo.

4.4.6.1 Pacote use.source

Neste pacote são armazenadas as classes que foram criadas com o objetivo de integrar a ferramenta *USE* com o protótipo desenvolvido. Sendo assim, foram criadas as seguintes classes: *MyMain*, *MyModelToGraph*, *MyOptions*, *MySession* e *MyShell*. Essas têm como base algumas classes do código fonte da ferramenta *USE* com pequenas modificações, possibilitando assim seu uso com o protótipo. A criação de cada uma das classes é detalhada a seguir:

- *MyMain*: tem como origem a classe *org.tzi.use.main.Main*. Difere da classe original na referência para as novas classes *MyOptions*, *MySession* e *MyShell*, ao invés de ter referência para as classes *Options*, *Session* e *Shell*. Além disso, nesta classe, as saídas da ferramenta *USE* foram mapeadas para os arquivos *logUse.txt* e *logErr.txt*.
- *MyModelToGraph*: tem como origem a classe *org.tzi.use.main.shell.ModelToGraph.* Difere da classe original na alteração do escopo *default* da classe para público, permitindo assim o acesso pela classe *MyShell*.
- *MyOptions*: tem como origem a classe *org.tzi.use.config.Options*. Difere da classe original nos valores dos atributos *specFileName* e *cmdFileName*, representando respectivamente o arquivo *USE* do meta-modelo e o arquivo *CMD* do modelo de aplicação. Dessa maneira, tornaram-se possíveis várias checagens de consistência do modelo para uma mesma instância da aplicação *USE*.
- *MySession*: tem como origem a classe *org.tzi.use.main.Session*. Difere da classe original na inclusão dos métodos *getFSystem()* e *setFSystem(MSystem system)*, possibilitando o retorno e a atribuição do estado atual do sistema.

- *MyShell*: tem como origem a classe *org.tzi.use.main.shell.Shell*. Difere da classe original na atribuição do valor *false* à variável *fFinished*, possibilitando assim novas checagens de consistência para uma mesma instância da aplicação *USE*, e no método *cmdExit()*, onde ao invés do encerramento da aplicação com o comando *System.exit()* é utilizada a linha de comando *fSession.getFSystem().reset()*, reiniciando o estado atual do sistema.

4.4.6.2 Pacote use.output

Neste pacote são armazenados os arquivos que gravam as saídas que seriam impressas no console da ferramenta *USE*, são eles: *logUse.txt* e *logErr.txt*. Assim, o primeiro arquivo armazena erros na estrutura do modelo criado e erros de consistência do modelo com o meta-modelo e suas restrições de integridade, enquanto que o segundo armazena apenas os erros de construção dos arquivos de entrada da ferramenta *USE* (caso a modelagem seja feita pelo protótipo, não ocorrem erros de construção).

4.4.7 Pacote parser

Neste pacote são armazenadas as classes responsáveis pelas traduções do protótipo. Dessa forma, o pacote é composto pelas classes *MetamodelToUseParser*, *ModelToObjectParser*, *ObjectToUseParser*, *CodeParser* e *ObjectToSemantiCoreParser* e é apresentado na Figura 4.9. Essas classes são detalhadas a seguir:

- *MetamodelToUseParser*: essa classe é responsável pela tradução do arquivo *XMI* que representa o meta-modelo juntamente com o arquivo de restrições de integridade *OCL* em um arquivo *USE* usado como entrada da ferramenta *USE*. Possui os atributos *xmiPathname*, *oclPathname* e *usePathname*. Esses representam respectivamente os caminhos dos arquivos *Metamodel.xmi*, *Constraints.ocl* e o caminho onde será gerado o arquivo *Metamodel.use*. Além do método construtor, essa classe possui o método *convertMetamodelToUse*, que tem como função a tradução dos arquivos *Metamodel.xmi* e *Constraints.ocl* em um arquivo *Metamodel.use*.

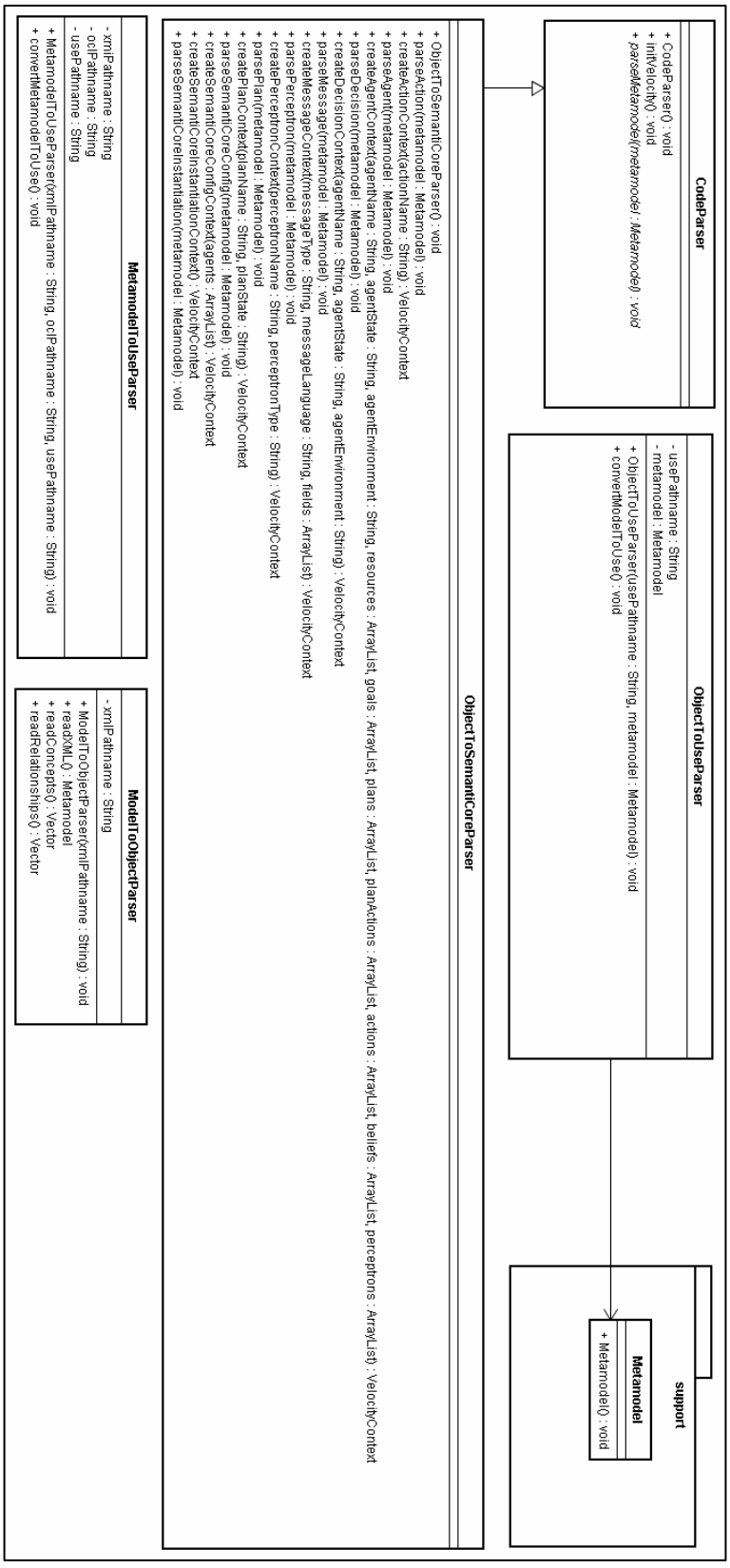


Figura 4.9 – Diagrama de Classes *UML* do pacote *parser*.

- *ModelToObjectParser*: essa classe é responsável pela tradução do arquivo que representa o modelo da aplicação em um objeto da classe *Metamodel*. Possui o

atributo *xmlPathname* que indica o caminho do arquivo *XML* referente ao modelo da aplicação. Além do método construtor, possui três métodos: *readXML*, *readConcepts* e *readRelationships*. O primeiro retorna um objeto da classe *Metamodel* contendo todos os conceitos e relacionamentos do modelo, o segundo retorna uma lista com todos os conceitos do modelo enquanto o último retorna uma lista com todos os relacionamentos do modelo.

- *ObjectToUseParser*: essa classe é responsável pela tradução de um objeto da classe *Metamodel* em um arquivo *CMD* usado como entrada da ferramenta *USE*. Possui os atributos *usePathname* e *metamodel*. O primeiro indica o caminho onde será gerado o arquivo *Model.cmd* e o segundo representa um objeto da classe *Metamodel* que contém o modelo da aplicação. Possui o método *convertModelToUse* que traduz o modelo da aplicação em um arquivo *Model.cmd*.
- *CodeParser*: essa é uma classe abstrata que deve ser estendida para a construção de *parsers* entre modelos de aplicação (representados por objetos da classe *Metamodel*) e código fonte de plataformas de implementação de SMAs. Além do método construtor, possui os métodos *initVelocity()*, responsável pela inicialização da ferramenta *Velocity*, e *parseMetamodel(Metamodel metamodel)*, responsável pela tradução do modelo em código. Este último consiste em um método abstrato que deverá ser implementado na classe filha, conforme a plataforma de implementação escolhida.
- *ObjectToSemantiCoreParser*: essa classe é responsável pela tradução de um objeto da classe *Metamodel* para código fonte da plataforma *SemantiCore*. Para isso, deve estender a classe *CodeParser* e implementar o método *parseMetamodel(Metamodel metamodel)*. Assim, nesse método são feitas as chamadas para os demais métodos do tipo *parser*, são eles: *parseAction*, *parseAgent*, *parseDecision*, *parseMessage*, *parseMetamodel*, *parsePerceptron*, *parsePlan*, *parseSemantiCoreConfig* e *parseSemantiCoreInstantiation*. Cada um desses métodos recebe um objeto da classe *Metamodel* como argumento e efetua a criação de um tipo de arquivo no *SemantiCore*. Dentro de cada método ainda existe uma chamada para um método do tipo *createContext* que recebe como argumentos apenas os atributos do objeto da classe *Metamodel* relevantes para o contexto do arquivo que está sendo criado. Por exemplo, o método *parseAction* efetua a chamada do método *createActionContext (String actionName)* e o valor de *actionName* é atribuído a uma variável de contexto *Velocity*. Uma vez atribuídos os valores às variáveis de contexto *Velocity*, as mesmas podem ser

referenciadas por arquivos do tipo *VTL* que servirão como *templates* para os arquivos gerados.

4.4.8 Pacote relationships

Neste pacote são armazenadas as classes utilizadas para guardar temporariamente os valores dos relacionamentos do meta-modelo durante o processo de geração de código com a plataforma de implementação. Todas as classes desse pacote estendem a classe *Relationship*, que é composta por três atributos, são eles: *name*, definindo o nome do relacionamento, *idA*, definindo o nome do conceito que inicia o relacionamento, e *idB*, definindo o nome do conceito que finaliza o relacionamento. A Figura 4.10 apresenta o Diagrama de Classes *UML* desse pacote.

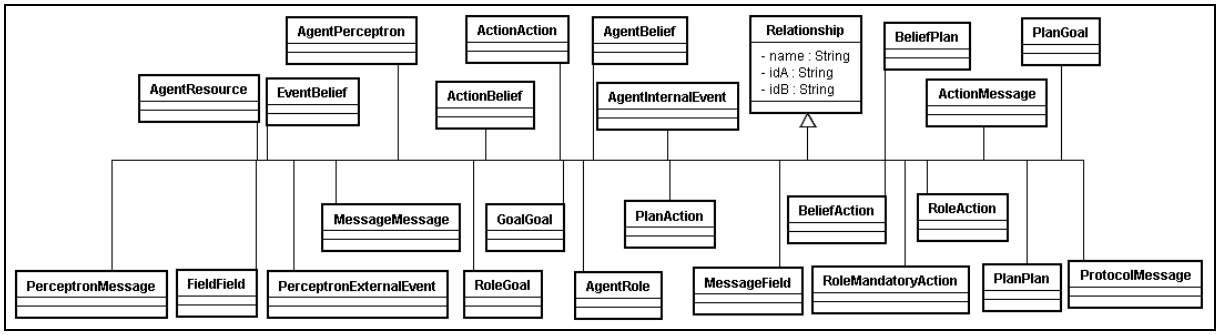


Figura 4.10 – Diagrama de Classes *UML* do pacote *relationships*.

4.4.9 Pacote support

Neste pacote são armazenadas algumas classes auxiliares do protótipo, são elas:

- *Metamodel*: responsável por armazenar todos os conceitos e relacionamentos de um modelo de aplicação, para isso, possui uma lista (*ArrayList*) de cada conceito e de cada relacionamento do meta-modelo.
- *ClassAux*: é utilizada na classe *MetamodelToUseParser* para armazenar temporariamente os dados de conceitos do meta-modelo. Possui um atributo *id* do tipo *String* e um atributo *description* do tipo *String*.
- *FormatPlan*: responsável pela formatação em caixa baixa ou caixa alta do atributo *name* do plano a ser gerado no código fonte da plataforma de implementação. Possui os atributos *name*, *nameUpper*, *nameLower* e *state*, todos do tipo *String*.

- *FormatResource*: responsável pela formatação em caixa baixa ou caixa alta do atributo *name* do recurso a ser gerado no código fonte da plataforma de implementação. Possui os atributos *name*, *nameUpper*, *nameLower*, *type* e *value*, todos do tipo *String*.

4.4.9.1 Pacote support.semanticore

Neste pacote são armazenadas algumas classes auxiliares usadas para a geração de código para a plataforma *SemantiCore*, são elas:

- *MessageAux*: esta classe é responsável por armazenar os dados de um conceito *Message* e dos conceitos *Protocol* e *Field* que se relacionam com o mesmo.
- *TermAux*: esta classe é responsável por armazenar o atributo *id* de um termo e armazenar partes do atributo *description* desse termo nos atributos *subject, predicate e object* que serão usados no *SemantiCore* para a representação de um *SimpleFact*.

4.4.10 Pacote velocity

Neste pacote são armazenados todos os arquivos do tipo *VTL* usados na geração de código para determinada plataforma de implementação. Para o *SemantiCore*, foram criados os seguintes arquivos:

- *action.vm*: *template* para a geração de código de uma classe *Action*.
- *agent.vm*: *template* para a geração de código de uma classe *SemanticAgent*.
- *decision.vm*: *template* para a geração de código de uma classe *DecisonEngine* (mecanismo decisório).
- *message.vm*: *template* para a geração de código de uma classe *SemanticMessage*.
- *perceptron.vm*: *template* para a geração de código de uma classe *Sensor*.
- *plan.vm*: *template* para a geração de código de uma classe *ActionPlan*.
- *semanticoreconfig.vm*: *template* para a geração do arquivo *semanticoreconfig.xml* (nesse arquivo são configurados os agentes que serão instanciados na plataforma *SemantiCore*).

- *semanticoreinstantiation.vm*: *template* para a geração do arquivo *semanticoreinstantiation.xml* (nesse arquivo estão as informações dos *hotspots* da plataforma).

4.4.10.1 *Pacote velocity.conf*

Neste pacote são armazenados os arquivos de configuração e de manutenção de *logs* da ferramenta *Velocity*, são eles: *velocity.properties e velocity.log*.

4.5 Especialização do Protótipo

Uma importante característica do protótipo desenvolvido é a possibilidade de extensão do mesmo de forma que possa ser usado para gerar código em outras plataformas de implementação existentes. Para isso, os seguintes passos são necessários:

- Mapeamento dos conceitos e relacionamentos do meta-modelo para a plataforma de implementação escolhida.
- Criação de uma classe que estende a classe *CodeParser* e implementa o método *parseMetamodel(Metamodel metamodel)*. Nesse método, os valores dos conceitos e dos relacionamentos de um modelo devem ser atribuídos a diferentes variáveis inclusas em contextos da ferramenta *Velocity*. Com isso, os valores podem ser recuperados por arquivos *VTL* que servirão como *templates* para os arquivos gerados.
- Criação dos arquivos *VTL* usados na geração de código para determinada plataforma de implementação.
- Conforme a plataforma de implementação usada, pode ser necessária a criação de classes que auxiliem na geração de código.

4.6 Inclusão das Restrições de Integridade no Protótipo Desenvolvido

O protótipo desenvolvido não tem o objetivo de garantir a consistência de um modelo de aplicação pela sua construção. Apesar disso, algumas das restrições de integridade aplicadas ao meta-modelo já estão contempladas no protótipo. Com isso, algumas das

restrições nunca serão disparadas a menos que o modelo da aplicação seja construído manualmente sem o uso do protótipo.

Dentre as restrições apresentadas no Apêndice I, já são tratadas no protótipo as seguintes:

- Restrições de obrigatoriedade e de unicidade dos atributos identificadores dos conceitos, são eles: *name* de *Agent*; *name* de *Resource*; *name* de *Role* (único em uma mesma sociedade); *name* de *Plan*; *name* de *Goal*; *name* de *Action*; *id* de *Message*; *name* de *Protocol*; *id* de *Field*; *name* de *Perceptron*; *name* de *Event* e *id* de *Belief*.

Assim, das quarenta e três restrições aplicadas ao meta-modelo, vinte e quatro delas já são contempladas no desenvolvimento do protótipo.

4.7 Padrão de Representação dos Modelos

O padrão de representação dos modelos em *XML* utilizado pelo protótipo foi criado com o objetivo de facilitar a integração do mesmo com diferentes ferramentas. Dessa forma, o arquivo *XML* gerado pode ser traduzido para uma entrada da ferramenta *USE* (representada pelo arquivo *Model.cmd*), assim como pode ser traduzido para código fonte de uma plataforma de implementação (no exemplo de uso, o *SemantiCore*). Na Figura 4.11 é apresentado um exemplo de um possível trecho de um modelo de aplicação representado no padrão utilizado pelo protótipo.

```
<?xml version = '1.0' encoding = 'UTF-8' ?>
<metamodel>
<concept def='Agent' name='Cliente' state='created' environment='ComponentesEnvironment'/>
<concept def='Role' name='Comprador' society='TacSCM'/>
<relationship def='Agent_Role' idA='Cliente' idB='Comprador'/>
</metamodel>
```

Figura 4.11 – Padrão de Representação dos Modelos.

Conforme a Figura 4.11, a primeira linha representa apenas a inicialização do arquivo *XML*. As *tags* <*metamodel*> e </*metamodel*> indicam respectivamente o início e o fim do

modelo de aplicação. Existem dois tipos de elementos no padrão: *concept* e *relationship*. O elemento *concept* é composto pelo atributo *def*, descrevendo o conceito do meta-modelo, e pelos atributos relacionados ao conceito. Porém, existem duas exceções, essas ocorrem quando o atributo *def* for igual à *Sentence* ou à *Rule*. Na primeira, além de *def* e dos atributos relacionados ao conceito, o elemento *concept* possui os atributos *beliefA*, *beliefB* e *operator*, descrevendo o relacionamento de agregação entre o conceito *Sentence* e o conceito *Belief* com o uso do atributo *symbol* da classe associativa *Operator*. Na segunda, o elemento *concept* além de *def* e dos atributos relacionados ao conceito, também é composto pelos atributos *antecedent* e *consequent*, descrevendo os relacionamentos entre um conceito *Rule* e dois conceitos *Belief*. No exemplo, é apresentado um *concept* com os atributos *def* igual à *Agent*, *name* igual à *Cliente*, *state* igual à *created* e *environment* igual à *ComponentesEnvironment*. Além disso, é apresentado um *concept* com o atributo *def* igual à *Role*, atributo *name* igual à *Comprador* e atributo *society* igual à *TacSCM*. De maneira semelhante, o elemento *relationship* sempre possui um atributo *def*, descrevendo o relacionamento do meta-modelo, e dois atributos, *idA* e *idB*, identificando os conceitos participantes do relacionamento. Assim, no exemplo é apresentado um *relationship* com os atributos *def* igual à *Agent_Role*, *idA* igual à *Cliente* e *idB* igual à *Comprador*.

4.8 Uso do Protótipo

Nesta seção será apresentado o detalhamento da estrutura geral do protótipo e serão mostradas as suas funcionalidades específicas por meio da visualização das interfaces que o compõem.

4.8.1 Estrutura Geral

A tela principal do protótipo é composta por quatro menus principais, são eles: *File*, *Model*, *Code* e *Help*. A Figura 4.12 apresenta a tela principal do protótipo.

O menu *File* é composto por cinco itens, são eles: *New*, *Load*, *Close*, *Save* e *Exit*. O primeiro possibilita a criação de um novo modelo de aplicação, o segundo permite o carregamento de um modelo contido em um arquivo *XML*, o terceiro viabiliza o fechamento de um modelo carregado no protótipo, o quarto permite o armazenamento do modelo em uso no protótipo em um arquivo *XML*, e o último possibilita o fechamento do protótipo.

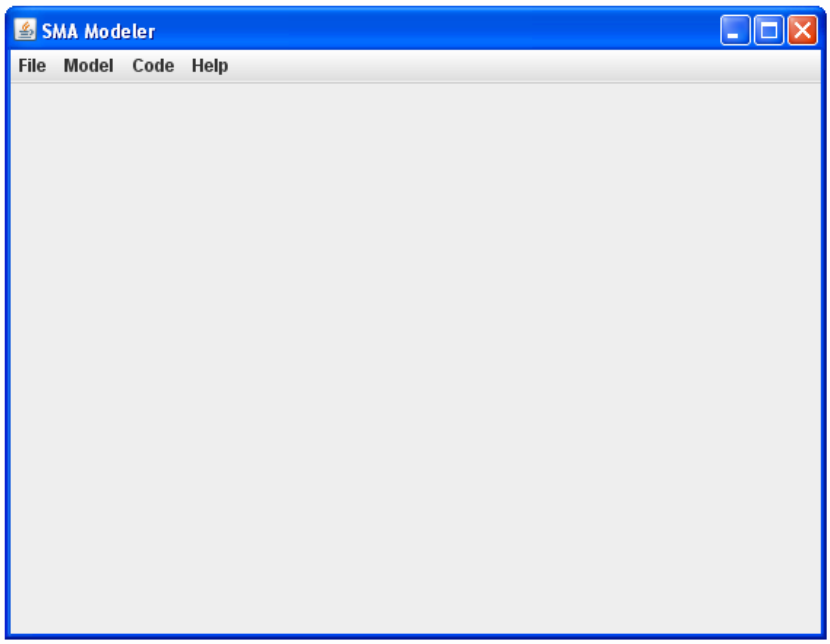


Figura 4.12 – Tela Principal do Protótipo.

O menu *Model* possibilita a checagem de modelos com base no meta-modelo e suas restrições de integridade. Para isso, existe o item *Check Model* que quando acionado dispara o processo de checagem retornando para o usuário possíveis erros de consistência entre o modelo de aplicação e o meta-modelo proposto e suas restrições.

O menu *Code* permite a geração de código caso o modelo esteja consistente com o meta-modelo e suas restrições. No escopo deste trabalho, foi feita a geração de código na plataforma *SemantiCore*. Sendo assim, quando o item *Generate SemantiCore code* é acionado será gerado o código fonte da plataforma respeitando a modelagem realizada.

O menu *Help* apresenta apenas informações sobre a versão do protótipo disponibilizada no item *About*.

4.8.2 Funcionalidades Específicas

Inicialmente, quando um modelo é criado ou carregado no protótipo, é apresentada uma tela contendo duas listas, uma de conceitos (*Concepts List*) e outra de relacionamentos (*Relationships List*), a mesma é mostrada na Figura 4.13. Nessa tela, o protótipo disponibiliza algumas funcionalidades específicas para o tratamento de modelos de uma aplicação orientada a agentes. Essas funcionalidades são a criação, consulta, alteração e exclusão de conceitos, e a criação e exclusão de relacionamentos. As próximas seções detalham cada uma dessas funcionalidades.

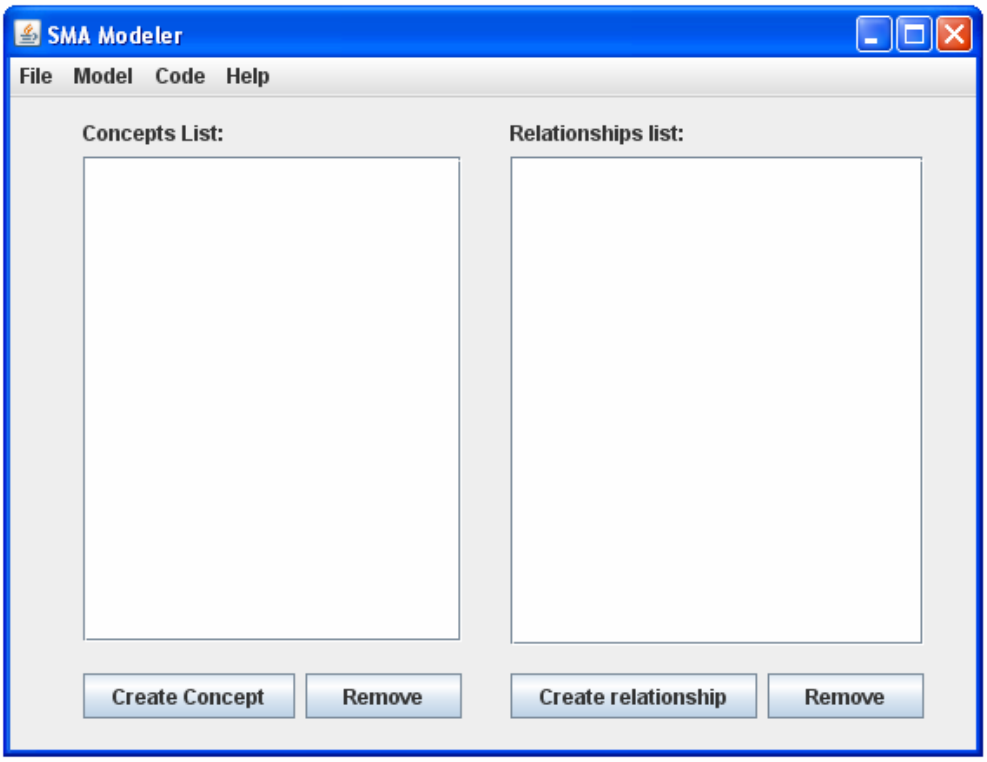


Figura 4.13 – Tela Inicial do Protótipo.

4.8.2.1 Criação de Conceitos

A tela inicial do protótipo permite a criação de conceitos pelo acionamento do botão *Create Concept*. Caso esse seja acionado, será apresentada a tela para seleção do conceito a ser criado, conforme a Figura 4.14. Após a seleção do conceito e o acionamento do botão *Next* é apresentada a tela de criação do conceito escolhido.

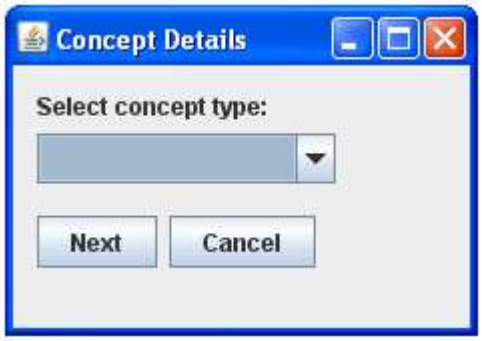


Figura 4.14 – Tela de Seleção de Conceito.

Na Figura 4.15 é exibida a tela de criação de um conceito *Agent*. O símbolo *, apresentado ao lado do campo *Name*, indica que este campo é o identificador do conceito e

deve obrigatoriamente ser informado. As demais telas de criação de conceitos seguem este modelo.

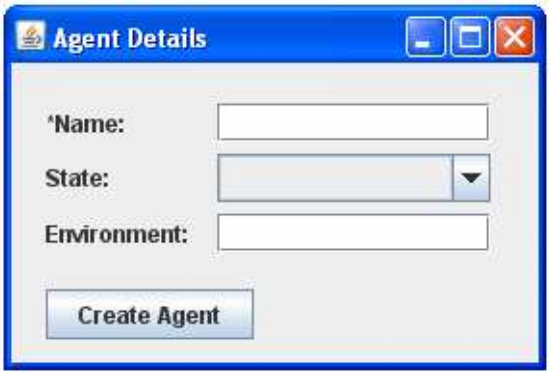


Figura 4.15 – Tela de Criação do Conceito *Agent*.

Uma importante consideração é que caso seja necessária a associação de uma sentença a mais de duas crenças devem ser criados conceitos *Sentence* em cascata. Por exemplo, para uma sentença que agrega as crenças *a*, *b* e *c*, deve ser criada uma subsentença que agrega *a* e *b* e depois uma sentença que agrega a subsentença criada e a crença *c*.

4.8.2.2 Consulta de Conceitos

O duplo clique em um dos conceitos constantes na lista de conceitos da tela inicial permite a consulta desse. Como exemplo, na Figura 4.16 é apresentada a tela de consulta do conceito *Agent* preenchida com valores já cadastrados. As demais telas de consulta seguem este modelo.

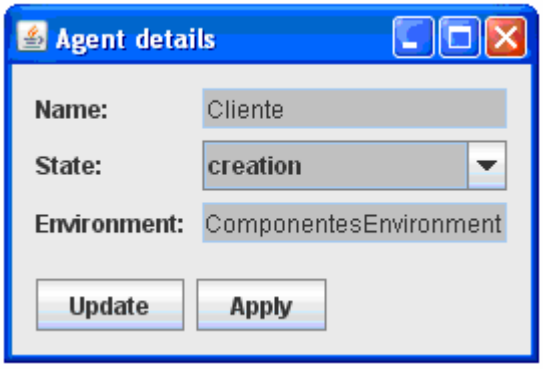


Figura 4.16 – Tela de Consulta do Conceito *Agent*.

4.8.2.3 Alteração de Conceitos

Na tela de consulta de um conceito, é possível a atualização de seus atributos, com exceção do seu atributo identificador. Assim, para a modificação dos atributos de um conceito deve ser acionado o botão *Update* e as alterações realizadas só serão aplicadas com o acionamento do botão *Apply*. A Figura 4.17 apresenta a tela de consulta do conceito *Agent* no estado de alteração.

Figura 4.17 – Tela de Consulta do Conceito *Agent* (estado de alteração).

4.8.2.4 Exclusão de Conceitos

Para a exclusão de um conceito, é necessária a seleção do conceito a ser excluído na lista de conceitos e o acionamento do botão *Remove*. A exclusão de um conceito é replicada a todos os relacionamentos do mesmo.

4.8.2.5 Criação de Relacionamentos

A tela inicial do protótipo permite a criação de relacionamentos pelo acionamento do botão *Create Relationship*. Caso esse seja acionado, será apresentada a tela para seleção do relacionamento a ser criado, conforme a Figura 4.18. Após a seleção do relacionamento e o acionamento do botão *Next* é apresentada a tela de criação do relacionamento escolhido. Na Figura 4.19 é apresentada a tela de criação do relacionamento entre um conceito *Agent* identificado como Cliente e um conceito *Role* identificado como Comprador. As demais telas de criação de relacionamentos seguem este modelo.

Figura 4.18 – Tela de Seleção de Relacionamento.

Figura 4.19 – Tela de Criação de Relacionamento entre *Agent* e *Role*.

4.8.2.6 Exclusão de Relacionamentos

Para a exclusão de um relacionamento, é necessária a seleção do relacionamento a ser excluído na lista de relacionamentos e o acionamento do botão *Remove*.

4.9 Considerações

Este capítulo possibilita um entendimento das ferramentas aplicadas na construção do protótipo. Com esse estudo, foi possível a definição do processo de consistência de modelos e geração de código entre o meta-modelo e as plataformas de implementação. A grande vantagem desse processo é a possibilidade de extensão do protótipo, permitindo a geração de código nas plataformas de implementação mapeadas. O padrão de representação dos modelos auxilia na integração do protótipo com outras ferramentas pelo uso da linguagem *XML*.

No capítulo seguinte é apresentado um exemplo prático de aplicação do meta-modelo. Além disso, é feita uma análise da cobertura do código fonte gerado para esse exemplo na plataforma de implementação *SemantiCore*.

5 EXEMPLO DE USO DO META-MODELO

Neste capítulo será apresentado um exemplo de uso do protótipo desenvolvido para o meta-modelo. Apenas aspectos específicos do problema serão explorados com o objetivo de demonstrar a aplicação de grande parte dos conceitos e dos relacionamentos que compõem o meta-modelo, e não de modelar um sistema por completo. Por fim, será apresentada uma análise da cobertura do código gerado para a plataforma de implementação *SemantiCore*.

O exemplo elaborado foi inspirado no *Supply Chain Management Game* (*Tac SCM*). Esse é um jogo onde os agentes devem competir entre si por pedidos de computadores feitos por clientes e aquisição de componentes de fornecedores, gerenciando inventários e produzindo computadores. O *Tac SCM* foi desenvolvido por uma equipe de pesquisadores do *e-Supply Chain Management Lab* da Universidade de Carnegie Mellon juntamente com pesquisadores da Universidade de Minnesota e do Instituto de Ciência da Computação Sueco. A especificação do *Tac SCM* pode ser consultada em [TAC06].

O exemplo busca o gerenciamento de uma cadeia de fornecimento de computadores baseado no planejamento e coordenação das atividades de uma organização, desde a aquisição de componentes até a entrega bem sucedida. Atualmente, o gerenciamento de cadeias de fornecimento é vital para a competitividade de empresas de manufatura assim como impacta diretamente na capacidade de cumprir demandas de mercado mutáveis em um tempo e custo efetivos. Um sistema multiagentes pode ser aplicado nesse modelo com o objetivo de suportar os aspectos dinâmicos da cadeia, tais como as diferentes necessidades de clientes e o estabelecimento de contratos com fornecedores. Na seção seguinte, é explicado o cenário baseado no gerenciamento de pedidos de componentes.

5.1 Gerenciamento de Pedidos de Componentes

O exemplo simula uma negociação entre dois agentes. Para isso, foi utilizado o protocolo *Contract Net* [FIP07a], simulando a troca de mensagens entre os agentes modelados. A Figura 5.1 apresenta a estrutura do protocolo.

Para o exemplo, foram simuladas apenas as trocas de mensagens *cfp*, *propose* e *accept-proposal* entre os agentes Cliente e Fornecedor. O agente Cliente possui nomes e marcas de componentes como recursos, além de exercer o papel Comprador. Esse papel almeja atingir o objetivo ComprarComponentes e deve executar uma ação EnviarPedido,

podendo ainda executar uma ação ConfirmarCompra. Além disso, o objetivo ComprarComponentes é cumprido pelo plano EfetuarCompra que é composto pela ação ConfirmarCompra.

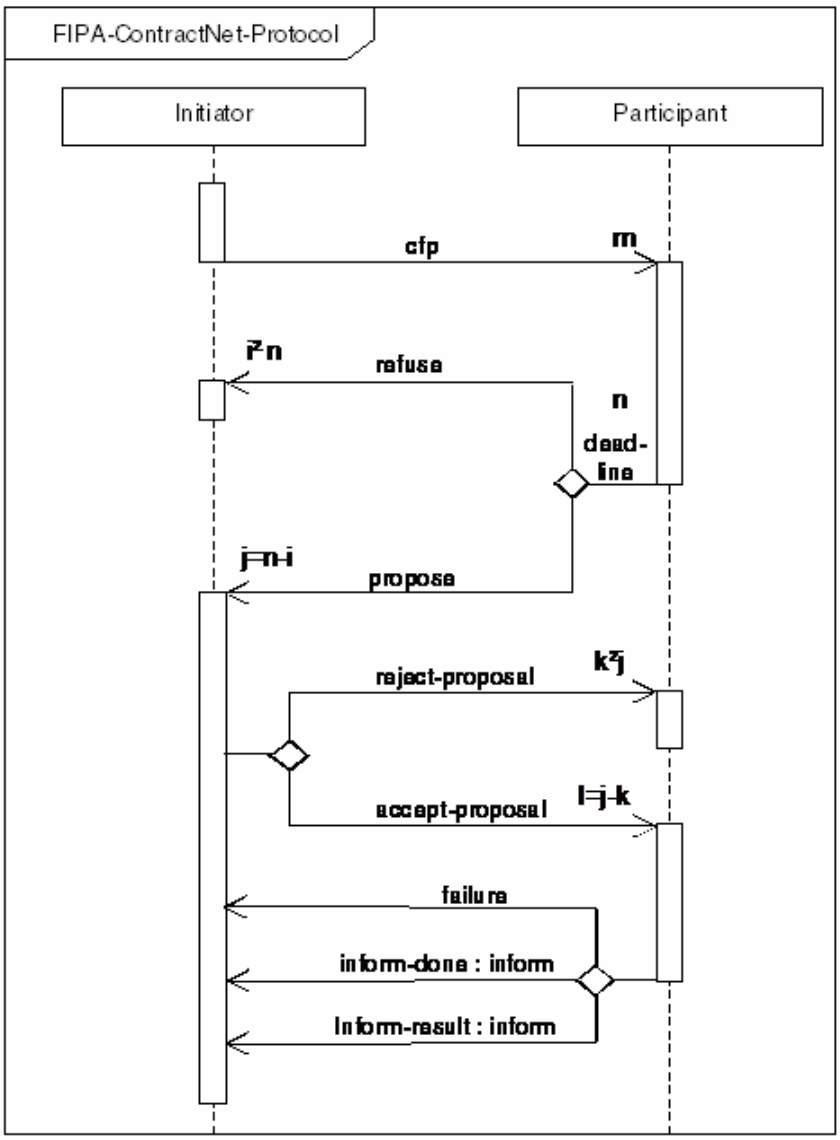


Figura 5.1 – Estrutura do protocolo *Contract Net* [FIP07a].

O agente Fornecedor possui nomes, marcas e preços de componentes como recursos, além de exercer o papel Vendedor. Esse papel almeja atingir o objetivo VenderComponentes e pode executar uma ação EnviarProposta. O objetivo VenderComponentes é cumprido pelo plano EfetuarVenda que é composto pela ação EnviarProposta.

A Figura 5.2 apresenta uma visão geral da comunicação entre os agentes participantes da negociação para a compra de componentes. No exemplo aplicado ao protótipo, não foram

consideradas todas as mensagens pertencentes à estrutura do protocolo *Contract Net*, mas apenas aquelas suficientes para uma breve demonstração do uso do meta-modelo.

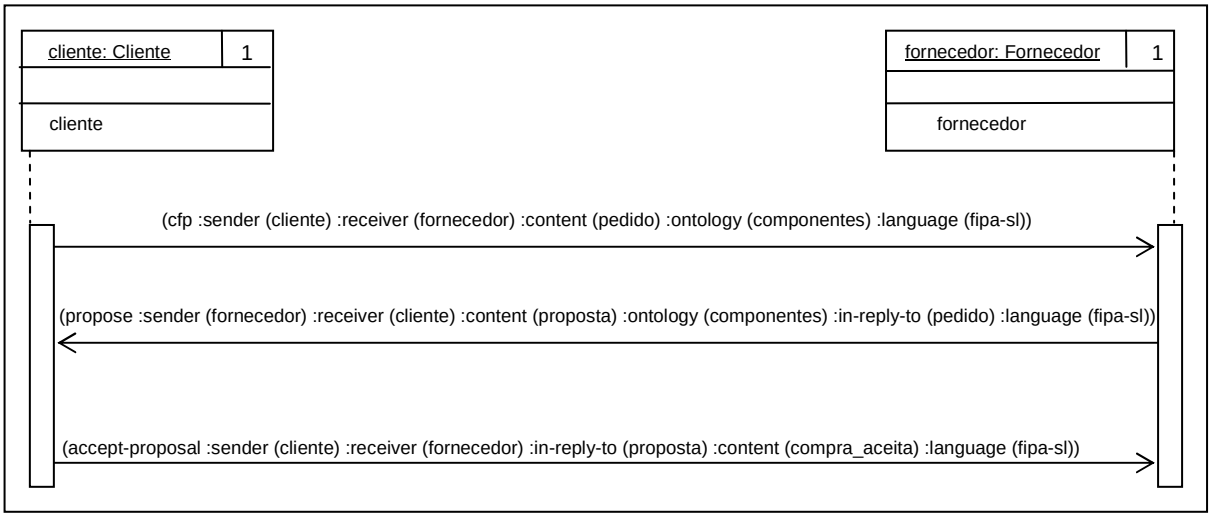


Figura 5.2 – Comunicação entre os agentes do exemplo modelado.

No início de uma negociação, o agente Cliente possui como crenças nomes e marcas de componentes. Além disso, o Cliente possui um evento interno denominado EventoEnviarPedido que quando disparado gera uma crença sinalizando que o pedido pode ser enviado. Assim, essa crença funciona como pré-condição para o início da ação EnviarPedido, que por sua vez, publica no ambiente uma mensagem do tipo *cfp* composta pelos campos *content* e *ontology*. O campo *content* é formado pelo pedido do Cliente e é representado por uma sentença composta por dois termos, onde o primeiro representa o nome de um componente solicitado e o segundo representa a marca do mesmo. Logo que o pedido é enviado, outro termo é gerado, indicando o envio desse pedido.

O outro agente participante da negociação, o Fornecedor, possui um *perceptron* denominado AvaliaPedidos, que aceita mensagens do tipo *cfp do* ambiente. Com isso, assim que o *perceptron* AvaliaPedidos aceita a mensagem *cfp* do ambiente, o mesmo dispara o evento externo EventoRecebePedido, que gera as crenças que representam o pedido recebido. Com o pedido recebido, o plano EfetuarVenda pode ser iniciado, disparando assim a ação EnviarProposta, que por sua vez publica uma mensagem do tipo *propose* composta pelos campos *content* e *ontology*. O campo *content* dessa mensagem é formado pela proposta com o preço sugerido para o pedido recebido.

Na continuação da negociação, a mensagem do tipo *propose* é aceita pelo *perceptron* AvaliaPropostas do agente Cliente. Esse *perceptron* dispara um evento externo EventoRecebeProposta, gerando as crenças que indicam a proposta recebida. Caso a proposta

recebida tenha um preço aceitável é gerado um termo que dispara a geração de outro termo no agente, indicando que a compra foi aceita. Essa geração é feita pelo uso de uma regra, composta pelos dois termos citados, sendo o primeiro como antecedente e o segundo como conseqüente da mesma. Gerado o termo conseqüente, o plano EfetuarCompra é iniciado, disparando a ação ConfirmarCompra. Essa ação então publica uma mensagem do tipo a*ccept-proposal* composta pelo campo *content* que carrega a aceitação da compra.

O agente Fornecedor possui o *perceptron* AvaliaCompras que aceita a mensagem do tipo *accept-proposal* e dispara o evento externo EventoRecebeCompra, gerando o termo que indica que a compra foi efetuada com sucesso e encerrando a negociação. É importante notar que as três mensagens publicadas pelos agentes nesse exemplo estão associadas ao protocolo *Contract Net* e possuem uma relação de dependência entre si. Sendo assim, a mensagem *accept-proposal* sucede a mensagem *propose*, que por sua vez sucede a mensagem *cfp*. O Apêndice II apresenta a representação em *XML* do exemplo.

5.1.1 Aplicação do Exemplo

O modelo instanciado do meta-modelo é apresentado no Apêndice II, esse é representado por meio de um arquivo *XML* gerado como saída do protótipo. Além disso, o arquivo gerado pode ser carregado no protótipo. Após o carregamento, a tela representada na Figura 5.3 é apresentada ao usuário.

Carregado o modelo da aplicação, pôde ser realizada a checagem do mesmo pelo uso do menu *Model – Check Model*. Nessa checagem, são gerados dois arquivos dentro do subpacote *use.output*, são eles: *logErr.txt* e *logUse.txt*. O primeiro foi utilizado com maior freqüência durante o desenvolvimento do protótipo, pois apresenta erros na construção dos arquivos de entrada da ferramenta *USE*. Todavia, na demonstração do exemplo, esse arquivo será gerado vazio, visto que o protótipo gera os arquivos de entrada da ferramenta *USE* de forma correta. O segundo arquivo apresenta erros na estrutura do modelo criado e erros de consistência do modelo com o meta-modelo e suas restrições de integridade. Nesse caso, podem ocorrer erros na construção do modelo e o conteúdo do arquivo é apresentado em uma interface do protótipo. A tela apresentada na Figura 5.4, indica que o modelo está estruturalmente consistente, pois após a checagem da estrutura (*checking structure*) não é apresentado nenhum tipo de erro. Na mesma tela, pode-se verificar se as restrições de integridade aplicadas ao meta-modelo estão sendo respeitadas. Isso é indicado na última linha

da Figura 5.5, que informa que foram checadas quarenta e três restrições invariantes em um tempo de 0.062s e com um número zero de falhas.

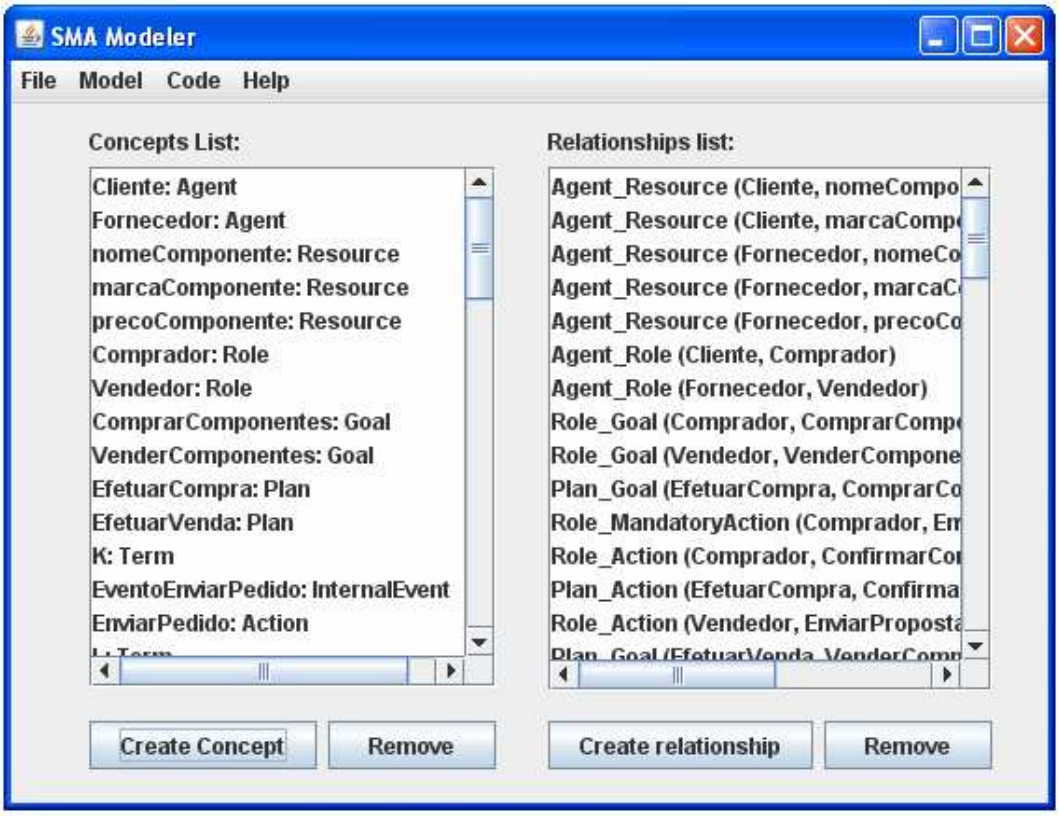


Figura 5.3 – Tela com Modelo Carregado.

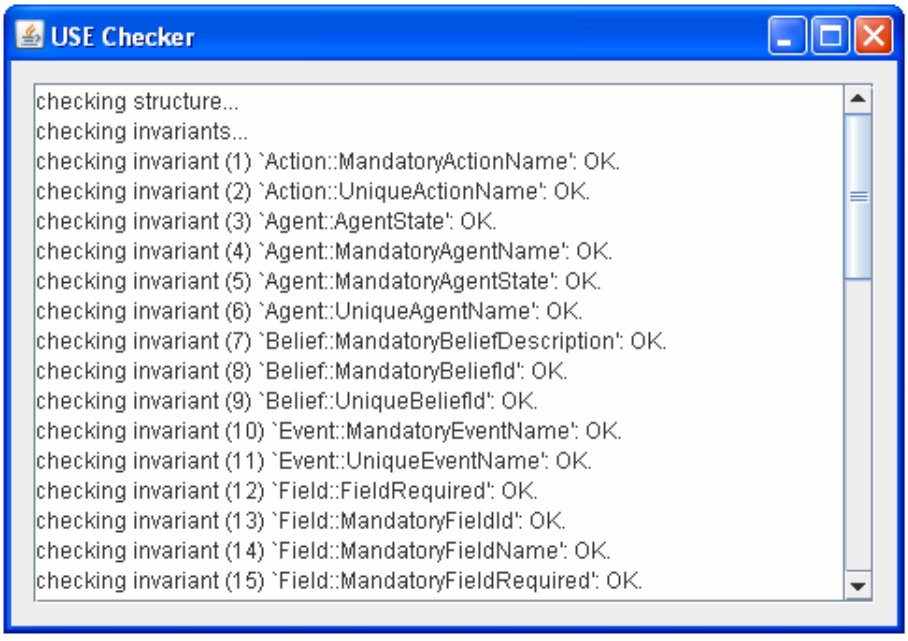


Figura 5.4 – Modelo Estruturalmente Consistente.

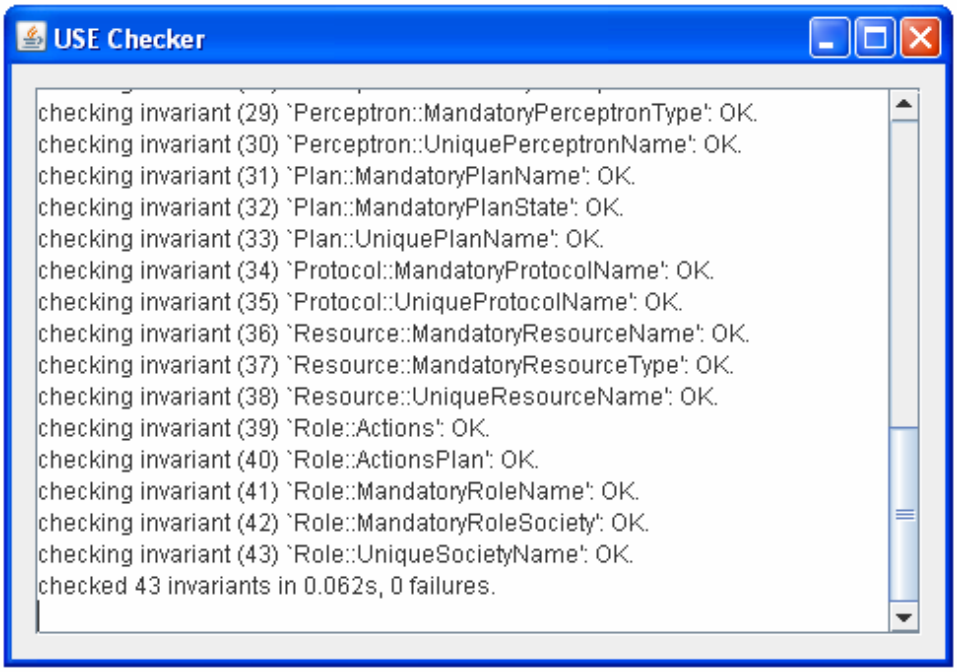


Figura 5.5 – Modelo Consistente com as Restrições Aplicadas.

Todavia, pode ocorrer do modelo não estar consistente com a estrutura do meta-modelo, assim como com as restrições de integridade aplicadas a esse. Dessa forma, o protótipo pode apresentar os dois tipos de erros de consistência citados anteriormente. O primeiro deles diz respeito à estrutura do modelo não estar de acordo com a estrutura dos conceitos e dos relacionamentos definidos no meta-modelo. Como um simples exemplo disso, pode-se dizer que um conceito *Plan* modelado não está associado a nenhum conceito *Goal*, nesse caso, o erro é apresentado na Figura 5.6, indicando que uma instância do conceito *Plan* deve estar associada a uma ou mais instâncias do conceito *Goal*. O segundo tipo de erro se refere à violação de uma das restrições de integridade definidas no arquivo *Constraints.ocl*. Como exemplo disso, pode-se dizer que as ações de um plano que alcança um objetivo almejado por um papel devem estar dentre as ações que o papel pode ou deve executar. Caso essa restrição seja violada, é apresentado o erro representado na Figura 5.7. Esse erro indica que a restrição denominada *ActionsPlan* falhou. Além disso, no final da tela é apresentado que foram checadas quarenta e três restrições de integridade em 0.062s, com uma falha no modelo.

Tomando-se como base o modelo consistente com o meta-modelo criado, para a continuidade da apresentação, prossegue-se para o processo de geração de código para a plataforma *SemantiCore* por meio do menu *Code – Generate SemantiCore code*. O resultado dessa geração é apresentado na próxima seção.

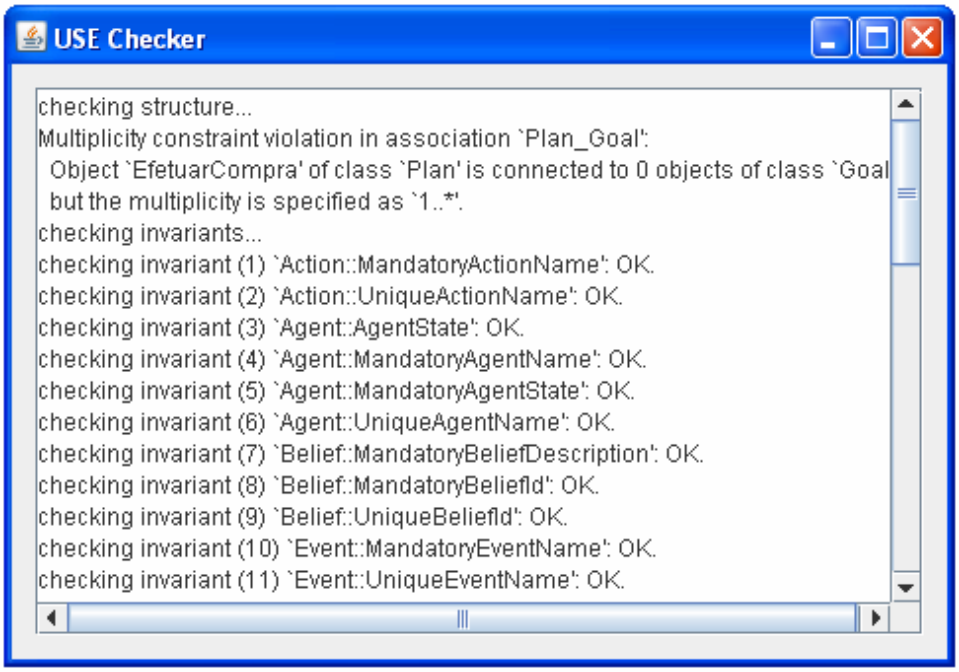


Figura 5.6 – Erro de Consistência Estrutural.

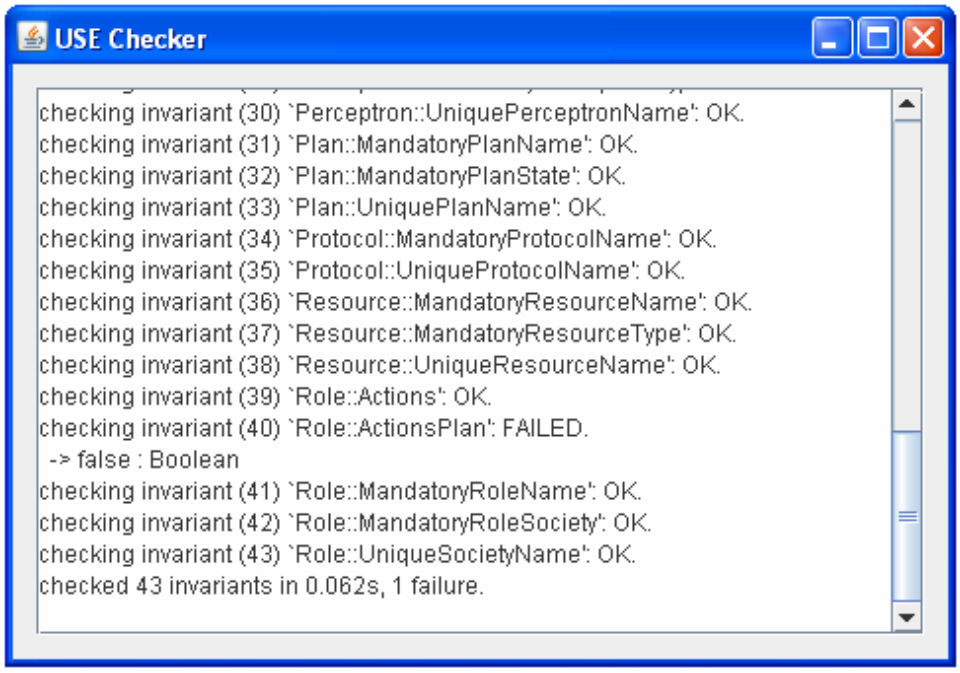


Figura 5.7 – Erro de Consistência nas Restrições de Integridade.

5.1.2 Cobertura do Código Gerado

O código gerado para o exemplo modelado é composto por dezessete arquivos, distribuídos da seguinte maneira: um arquivo *semanticoreconfig.xml* gerado na raiz do projeto, contendo os dados dos agentes que serão instanciados na plataforma *SemantiCore*;

um arquivo *semanticoreinstantiation.xml* gerado na raiz do projeto, contendo as informações dos *hotspots* da plataforma; as classes AvaliaPedidosSensor, AvaliaPropostasSensor e AvaliaComprasSensor do tipo *Sensor* e geradas no pacote *semanticore.agent.sensorial.hotspots*; as classes CfpMessage, ProposeMessage e AcceptProposalMessage do tipo *SemanticMessage* e geradas no pacote *semanticore.domain.model.hotspots*; as classes Cliente e Fornecedor do tipo *SemanticAgent* e geradas no pacote *application*; as classes ClienteDecisorio e FornecedorDecisorio do tipo *DecisionEngine* e geradas no pacote *semanticore.agent.decision.hotspots*; as classes EfetuarCompra e EfetuarVenda do tipo *ActionPlan* e geradas no pacote *semanticore.domain.actions.lib*; e as classes EnviarPedido, EnviarProposta e ConfirmarCompra do tipo *Action* e geradas no pacote *semanticore.domain.actions.lib*.

Para uma análise da cobertura do código gerado, foi desenvolvido o mesmo exemplo sem o auxílio do protótipo. Na análise, foi aplicada uma métrica de linhas de código útil em ambos os exemplos. Esta métrica foi criada com base em algumas métricas existentes para medir o tamanho do código fonte de aplicações (*lines of code metrics - LOC* [LOC07]). Assim, na contagem da métrica não foram consideradas expressões de início e fim de blocos Java (“{”, “}”), assim como não foi realizada a contagem das linhas que representam a declaração e a importação de pacotes (*package* e *import*) nas diferentes classes geradas. Para o arquivo *semanticoreconfig.xml* foram consideradas apenas as linhas que representavam os agentes instanciados. Por outro lado, para o arquivo *semanticoreinstantiation.xml* foram consideradas apenas as linhas que indicavam os *hotspots* da plataforma. Assim, da aplicação da métrica, foram identificados os aspectos do exemplo modelado que puderam ser traduzidos diretamente para código fonte da plataforma *SemantiCore*. A descrição da análise do código coberto é apresentada nos itens a seguir:

- *semanticoreinstantiation.xml*: este arquivo representa os *hotspots* da plataforma. O arquivo pode ser gerado em sua totalidade.
- *semanticoreconfig.xml*: este arquivo representa os agentes instanciados na plataforma. O arquivo pode ser gerado em sua totalidade.
- *Sensor*: foram criadas três classes do tipo *Sensor*, a AvaliaPedidosSensor, que avalia mensagens do tipo *cfp*, a AvaliaPropostasSensor, que avalia mensagens do tipo *propose* e a AvaliaComprasSensor, que avalia mensagens do tipo *accept-proposal*. O código das três classes foi gerado em sua totalidade.

- *SemanticMessage*: foram criadas três classes do tipo *SemanticMessage*, a CfpMessage, representando uma mensagem do tipo *cfp*, a ProposeMessage, representando uma mensagem do tipo *propose*, e a AcceptProposalMessage representando uma mensagem do tipo *accept-proposal*. Para essas classes foram gerados todos os atributos das mensagens modeladas assim como o protocolo e todos os campos associados a essas mensagens. O código das três classes foi gerado em sua totalidade.

```
public class Cliente extends SemanticAgent
{
	private String nomeComponente;
	private String marcaComponente;

	public Cliente ( Environment env, String agentName, String arg )
	{
		super ( env, agentName, arg );
	}

	protected void setup ( )
	{
		SimpleFact a = new SimpleFact("Pedido","Nome","Placa-Mae");
		SimpleFact b = new SimpleFact("Pedido","Marca","ASUS");
		SimpleFact c = new SimpleFact("Pedido","Nome","Disco-
		Rigido");
		SimpleFact d = new SimpleFact("Pedido","Marca","Samsung");
		SimpleFact e = new SimpleFact("Pedido","Nome","Processador");
		SimpleFact f = new SimpleFact("Pedido","Marca","Intel");
		SimpleFact g = new SimpleFact("Pedido","Nome","DVD");
		SimpleFact h = new SimpleFact("Pedido","Marca","LG");
		SimpleFact i = new SimpleFact("Pedido","Nome","Placa-Video");
		SimpleFact j = new SimpleFact("Pedido","Marca","Creative");
		ComposedFact ped01 = new ComposedFact(a,b);
		ComposedFact ped02 = new ComposedFact(c,d);
		ComposedFact ped03 = new ComposedFact(e,f);
		ComposedFact ped04 = new ComposedFact(g,h);
		ComposedFact ped05 = new ComposedFact(i,j);
		addFact(a);
		addFact(b);
		addFact(c);
		addFact(d);
		addFact(e);
		addFact(f);
		addFact(g);
		addFact(h);
		addFact(i);
		addFact(j);
		addFact(ped01);
		addFact(ped02);
		addFact(ped03);
		addFact(ped04);
		addFact(ped05);
```

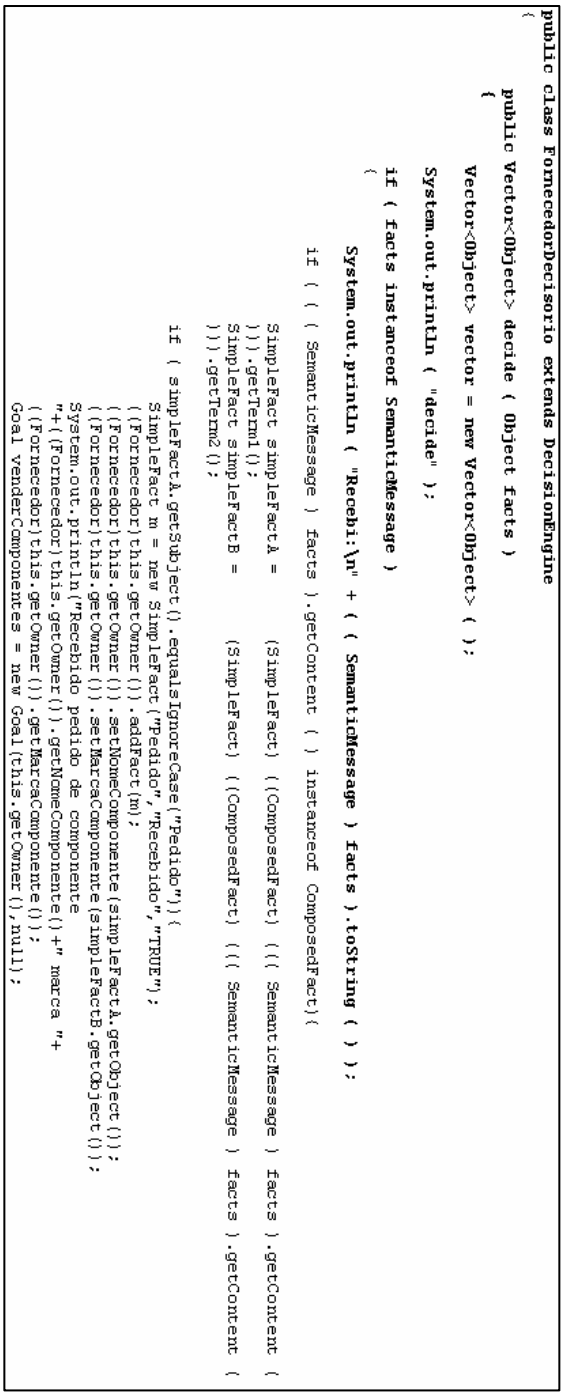
Figura 5.8 – Código gerado para a classe Cliente (1).

```
      int random = (int) (Math.random() * 5) + 1;
            if(random==1){
                  this.setNomeComponente(a.getObject());
                  this.setMarcaComponente(b.getObject());
            } else if (random==2){
                  this.setNomeComponente(c.getObject());
                  this.setMarcaComponente(d.getObject());
            } else if (random==3){
                  this.setNomeComponente(e.getObject());
                  this.setMarcaComponente(f.getObject());
            } else if (random ==4){
                  this.setNomeComponente(g.getObject());
                  this.setMarcaComponente(h.getObject());
            } else if (random==5){
                  this.setNomeComponente(i.getObject());
                  this.setMarcaComponente(j.getObject());
            }
            addSensor ( new UnicastSensor () );
            addSensor ( new AvaliaPropostasSensor ("AvaliaPropostas") );
            setDecisionEngine ( new ClienteDecisorio( ) );
            ActionPlan enviarPedido = new ActionPlan ( "EnviarPedido" );
            enviarPedido.addAction (new EnviarPedido());
            addActionPlan ( enviarPedido );
            EfetuarCompra efetuarCompra = new EfetuarCompra (
            "EfetuarCompra" );
            efetuarCompra.addAction ( new ConfirmarCompra ( ) );
            addActionPlan ( efetuarCompra );
            SimpleFact u = new SimpleFact("Pedido","Preco","Acessivel");
            SimpleFact v = new SimpleFact("Compra","Aceita","TRUE");
            addFact(u);
            addFact(v);
            Rule rule1 = new Rule("DecisaoCompra",u,v);
            addRule(rule1);
      }

      public String getNomeComponente() {
            return nomeComponente;
      }
      public void setNomeComponente(String nomeComponente) {
            this.nomeComponente = nomeComponente;
      }

      public String getMarcaComponente() {
            return marcaComponente;
      }

      public void setMarcaComponente(String marcaComponente) {
            this.marcaComponente = marcaComponente;
      }
}
```

Figura 5.9 – Código gerado para a classe Cliente (2).

- *SemanticAgent*: foram criadas duas classes do tipo *SemanticAgent*, o Cliente e o Fornecedor. Para essas classes foram gerados automaticamente os recursos, os termos, as sentenças e as regras associadas ao agente. Assim como foram criadas as associações com as classes *Sensor*, *ActionPlan* e a associação entre as classes

ActionPlan e *Action.* Não foi gerado automaticamente o trecho de código que representa a seleção de um pedido a ser enviado. Além disso, no *SemantiCore*, cada ação deve estar associada à pelo menos um plano, e no meta-modelo, uma ação pode ser independente de planos. Caso isso ocorra, o trecho de código com a criação e o uso de uma classe *ActionPlan* não é gerado automaticamente. A Figura 5.8 e a Figura 5.9 apresentam em negrito o código gerado para a classe Cliente.



```
public class FornecedorDecisorio extends DecisionEngine
{
    public Vector<Object> decide ( Object facts )
    {
        Vector<Object> vector = new Vector<Object> ( );

        System.out.println ( "decide" );
        if ( facts instanceof SemanticMessage )
        {
            System.out.println ( "Recebi:\n" + ( ( SemanticMessage ) facts ).toString ( ) );
            if ( ( ( SemanticMessage ) facts ).getContent ( ) instanceof ComposedFact){
                SimpleFact simpleFactA =     (SimpleFact) ((ComposedFact) ((( SemanticMessage ) facts ).getContent (
                )).getTerm1();
                SimpleFact simpleFactB =     (SimpleFact) ((ComposedFact) ((( SemanticMessage ) facts ).getContent (
                )).getTerm2();
                if ( simpleFactA.getSubject().equalsIgnoreCase("Pedido")){
                    SimpleFact m = new SimpleFact("Pedido","Recebido","TRUE");
                    ((Fornecedor)this.getOwner()).addFact(m);
                    ((Fornecedor)this.getOwner()).setNomeComponente(simpleFactA.getObject());
                    ((Fornecedor)this.getOwner()).setMarcaComponente(simpleFactB.getObject());
                    System.out.println("Recebido pedido de componente
                    "+((Fornecedor)this.getOwner()).getNomeComponente()+" marca "+
                    ((Fornecedor)this.getOwner()).getMarcaComponente());
                    Goal venderComponentes = new Goal(this.getOwner(),null);
```

Figura 5.10 – Código gerado para a classe FornecedorDecisorio (1).

```
                for(int i=0;i<this.getOwner().getExecutionComponent().getActionPlans().size();i++){
if(((ActionPlan)this.getOwner().getExecutionComponent().getActionPlans().get(i)).getName().equalsIgnoreCase("EfetuarVenda")){
venderComponentes.setPlan((ActionPlan)this.getOwner().getExecutionComponent().getActionPlans().get(i));
                    }
                    venderComponentes.start();
                }
            }
            if ( ( ( SemanticMessage ) facts ).getContent ( ) instanceof SimpleFact){
                SimpleFact v = (SimpleFact) ((( SemanticMessage ) facts ).getContent ( ));
                if ( v.getSubject().equalsIgnoreCase("Compra")){
                    if ( v.getObject().equalsIgnoreCase("TRUE")){
                        System.out.println ( "Venda efetuada");
                    }
                }
            }
        }
        return vector;
    }
}
```

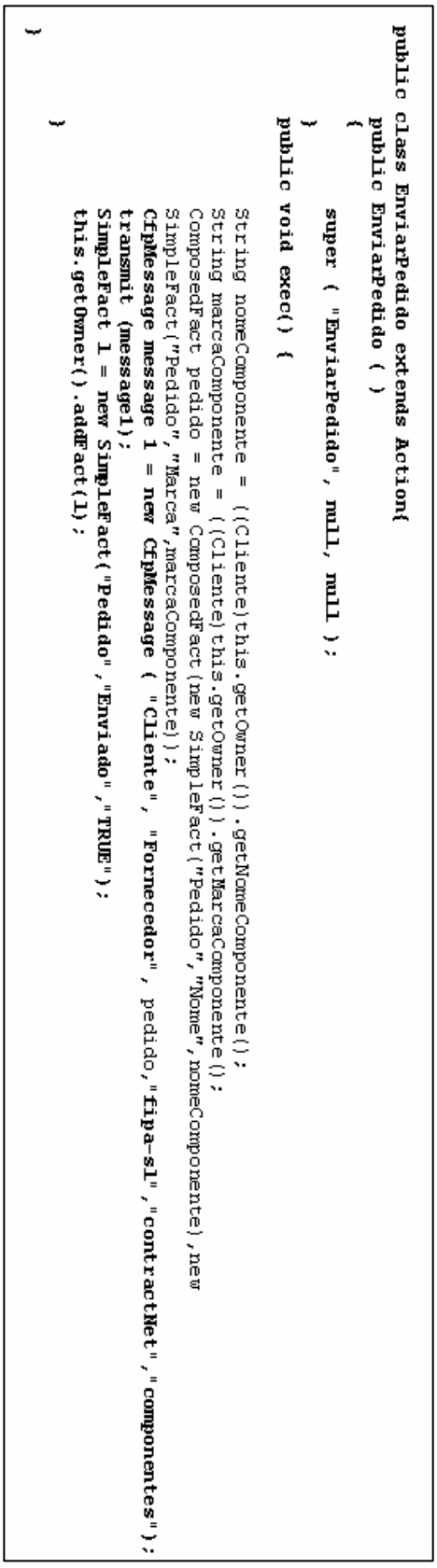
Figura 5.11 – Código gerado para a classe FornecedorDecisorio (2).

- *DecisionEngine*: foram criadas duas classes do tipo *DecisionEngine*, o ClienteDecisorio e o FornecedorDecisorio. Para essas, foi gerada apenas a estrutura geral da classe, pois as demais estruturas do mecanismo decisório serão dependentes de cada aplicação modelada. Dentre o código gerado, está inclusa a assinatura do método *decide*, responsável por avaliar os fatos que a classe *SemanticAgent* recebe do ambiente. A Figura 5.10 e a Figura 5.11 apresentam em negrito o código gerado para a classe FornecedorDecisorio.
- *ActionPlan*: foram criadas duas classes do tipo *ActionPlan*, a EfetuarVenda e a EfetuarCompra. O código de ambas as classes foi gerado em sua totalidade.
- *Action*: foram criadas três classes do tipo *Action*, a EnviarPedido, a EnviarProposta e a ConfirmarCompra. Para essas classes foram geradas automaticamente as mensagens associadas à ação assim como as crenças geradas pela mesma. O campo *content* (conteúdo da mensagem) associado a uma mensagem (extensão da classe *SemanticMessage*) não foi gerado, pois esse terá um valor diferente para cada execução da aplicação. A Figura 5.12 apresenta em negrito o código gerado para a classe EnviarPedido.

A Tabela 5.1 permite a visualização do número total de linhas de código fonte geradas totalmente e parcialmente, do total de linhas desenvolvidas sem o uso do protótipo e do percentual de código coberto para cada tipo de classe do *SemantiCore*, assim como para os arquivos *semanticoreconfig.xml* e *semanticoreinstantiation.xml* para o exemplo modelado.

Da análise da Tabela 5.1, constatou-se que para os arquivos *semanticoreconfig.xml* e *semanticoreinstantiation.xml* foi gerado 100% do código, assim como para as três classes do tipo *Sensor*, para as três classes do tipo *SemanticMessage* e para as duas classes do tipo *ActionPlan*. Para as classes *SemanticAgent* a média de código gerado foi de 75,41%, as classes do tipo *DecisionEngine* tiveram uma média de 21,54% de código gerado, enquanto que as classes do tipo *Action* tiveram 67,74% de código gerado e 9,68% de código parcialmente gerado. Por fim, das trezentas e sessenta e cinco linhas necessárias para a construção do exemplo sem o uso do protótipo desenvolvido, duzentas e setenta e seis foram geradas automaticamente, e três foram parcialmente geradas representando respectivamente uma média de 75,62% de código gerado e 0,82% de código parcialmente gerado. Esses números são válidos para o mapeamento para a plataforma *SemantiCore* com o uso do exemplo modelado, todavia para outras plataformas poderia ser feita uma nova análise do

código gerado. O Apêndice III apresenta em negrito os demais arquivos e classes gerados para o exemplo modelado.



```
public class EnviarPedido extends Action{
    public EnviarPedido ( )
    {
        super ( "EnviarPedido", null, null );
    }
    public void exec() {

        String nomeComponente = ((Cliente)this.getOwner()).getNomeComponente();
        String marcaComponente = ((Cliente)this.getOwner()).getMarcaComponente();
        ComposedFact pedido = new ComposedFact(new SimpleFact("Pedido","Nome",nomeComponente),new
        SimpleFact("Pedido","Marca",marcaComponente));
        CfpMessage message 1 = new CfpMessage ( "Cliente", "Fornecedor", pedido,"fipa-sl","contractNet","componentes");
        transmit (message1);
        SimpleFact 1 = new SimpleFact("Pedido","Enviado","TRUE");
        this.getOwner().addFact(1);
    }
}
```

Figura 5.12 – Código gerado para a classe EnviarPedido.

TABELA 5.1 – COBERTURA DO CÓDIGO.

	semanticoreconfig	*semanticoreinstantiation*	*Sensor*	*SemanticMessage*	*SemanticAgent*	*DecisionEngine*	*ActionPlan*	*Action*	Total
Total de linhas geradas	2	2	51	88	92	14	6	21	276
Total de linhas geradas parcialmente	0	0	0	0	0	0	0	3	3
Total de linhas desenvolvidas	2	2	51	88	122	65	6	31	365
Percentual médio coberto parcialmente	0%	0%	0%	0%	0%	0%	0%	9,68%	0,82%
Percentual médio coberto	100%	100%	100%	100%	75,41%	21,54%	100%	67,74%	75,62%

5.2 Considerações

Nesse capítulo, inicialmente foi detalhado o Gerenciamento de Pedidos de Componentes baseado no *Tac SCM*. Esse exemplo foi aplicado no protótipo possibilitando assim demonstrar o funcionamento do mesmo. No final do capítulo, ainda foi apresentada uma análise comparativa do código fonte gerado automaticamente em relação ao código fonte criado para o exemplo sem o auxílio do protótipo. Dessa análise, foi constatado que para o exemplo modelado 75,62% do código pôde ser gerado automaticamente e 0,82% do código pôde ser gerado parcialmente.

No capítulo a seguir são apresentadas as conclusões do trabalho assim como os possíveis trabalhos futuros.

6 CONCLUSÕES E TRABALHOS FUTUROS

O meta-modelo proposto busca possibilitar a representação dos conceitos que compõem um agente de software, assim como os relacionamentos entre os mesmos. Atualmente, existem diversas metodologias que podem ser utilizadas no desenvolvimento de sistemas multiagentes. Neste trabalho, foram estudadas as metodologias *MASUP*, *Tropos*, *MaSE*, *Prometheus* e *MAS-CommonKADS*. Deste estudo foram publicados dois artigos. O primeiro [SAN06b] trata de um estudo comparativo das metodologias *MASUP*, *Tropos*, *MaSE* e *Prometheus*, enquanto que o último [SAN07] trata de um estudo de caso utilizando a metodologia *MASUP*.

Além disso, também foi estudada a linguagem de modelagem *MAS-ML*, apresentada no Capítulo II. O *MAS-ML* é fortemente relacionado com o trabalho aqui proposto, pois define meta-modelos para o desenvolvimento de sistemas multiagentes. Dessa forma, o *MAS-ML* possibilita a modelagem de todos os aspectos estruturais e dinâmicos definidos no *framework* conceitual *TAO*. O *MAS-ML* também propõe um processo para a geração automática de código orientado a objetos a partir dos modelos descritos no nível de abstração do agente [SIL07].

No meta-modelo proposto, são consideradas as características internas tratadas na literatura e nas metodologias estudadas, assim como as do *MAS-ML* para a representação interna de agentes. Assim, torna-se possível o mapeamento da representação interna de um agente de cada uma das abordagens estudadas para o meta-modelo proposto. Além disso, se comparado com as abordagens estudadas, o meta-modelo proposto foca na parte interna dos agentes, enquanto as metodologias trabalham com todo o processo para o desenvolvimento de um SMA. Definido o meta-modelo, busca-se um mapeamento direto para a geração de código fonte em diversas plataformas de sistemas multiagentes.

Outro trabalho fortemente relacionado é proposto em [HEN05a]. Nesse, é descrito um meta-modelo para a avaliação dos termos “objetivos” e “tarefas” em metodologias orientadas a agentes. Porém, o mesmo não possui a aplicação de nenhuma linguagem de restrições e de nenhum mecanismo para a geração de código.

Do estudo realizado, as maiores contribuições são: a independência do meta-modelo com diferentes abordagens de desenvolvimento e plataformas de implementação, possibilitando criar modelos de SMA a partir de qualquer das abordagens estudadas, mapear estes modelos para os conceitos e relacionamentos do meta-modelo e traduzir esses de

maneira automática para código fonte de alguma das plataformas de implementação estudadas; a síntese dos conceitos tratados em diversas abordagens orientadas a agentes, permitindo a definição dos conceitos mais relevantes que compõe um agente de software; a aplicação da linguagem de restrições de integridade, garantindo a consistência dos modelos com o meta-modelo proposto; e o protótipo extensível, possibilitando assim a geração de código em diferentes plataformas de implementação mediante os passos apresentados na seção 4.5.

Contudo, surgem diversos trabalhos futuros para o trabalho desenvolvido. Um deles é a aplicação de restrições de integridade adicionais, melhorando assim o processo de consistência de modelos. A extensão do meta-modelo com os conceitos externos ao agente, possibilitaria a construção de um SMA por completo, levando em conta o ambiente e a organização dos agentes. A construção de uma notação diagramática para o meta-modelo, facilitaria a construção e a visualização de diferentes modelos pelo uso de diagramas. Além disso, o estudo de outras abordagens existentes, como o *ANote* [CHO05], o *INGENIAS* [PAV05], o *AGR* [FER04] e o *ISLANDER* [EST02] auxiliaria no refinamento do meta-modelo de maneira mais ampla. A descrição de casos apresentados nas metodologias com o uso do meta-modelo, permitiria que o meta-modelo fosse validado por diversos exemplos de uso. A integração do protótipo com a ferramenta visual da metodologia *MASUP*, possibilitaria a modelagem de sistemas multiagentes desde o levantamento de requisitos até a geração de código pelo uso de uma ferramenta integrada a uma metodologia. A extensão do protótipo para a geração de código em outras plataformas de implementação, permitiria a geração nas diferentes plataformas mapeadas. Por fim, o refinamento do protótipo para o tratamento de diferentes tipos de dados, possibilitaria que esse trabalhasse sempre com os mesmos tipos de dados definidos no meta-modelo.

REFERÊNCIAS BIBLIOGRÁFICAS

[ANT07] THE APACHE ANT PROJECT. Disponível em: <http://ant.apache.org/>. Acesso em: 11 dez. 2007.

[BAS04] BASTOS, R. M.; RIBEIRO, M. B. **Modeling Agent-Oriented Information Systems for Business Processes**. In: THIRD INTERNATIONAL WORKSHOP ON SOFTWARE ENGINEERING FOR LARGE-SCALE MULTI-AGENT SYSTEMS, 2004, Edimburgo, Escócia. 26th International Conference on Software Engineering – Workshop. Reino Unido: The IEE, 2004. p. 90-97.

[BAS05] BASTOS, R. M. ; RIBEIRO, M. B. **MASUP: An Agent-Oriented Modeling Process for Information Systems**. In: Ricardo Choren, Alessandro Garcia, Carlos Lucena, Alexander Romanovsky. (Org.). Software Engineering for Multi-agent Systems III. 1 ed. Berlin: Springer-Verlag, 2005, v. 3390, p. 19-35.

[BLO04] BLOIS, M., LUCENA, C. **MULTI-AGENT SYSTEMS AND THE SEMANTIC WEB - The SemanticCore Agent-Based Abstraction Layer**. In: ICEIS - International Conference on Enterprise Information Systems, 2004, Porto. Proceedings of Sixth International Conference on Enterprise Information Systems ICEIS 2004. Porto: INSTICC, 2004. p. 263-270.

[BOR06] BORDINI, R. and HÜBNER, J. BDI **Agent Programming in AgentSpeak Using *Jason* (Tutorial Paper)**. In: Computational Logic in Multi-Agent Systems. Berlin; Heidelberg: Springer, 2006. p. 143-164.

[BRA87] BRATMAN, M. E.; ISRAEL, D. J.; POLACK, M. E. **Toward an architecture for resource-bounded agents**. Technical Report CSLI-87-104, Center for the Study of Language and Information, SRI and Stanford University, CA, August 1987. 19 p.

[BRA99] BRAZIER, F.; DUNIN-KEPLICZ, B.; TREUR, J.; VERBRUGGE, R. **Modelling Internal Dynamic Behaviour of BDI Agents**. In: Formal Models of Agents: ESPRIT Project ModelAge Final Workshop. Berlin; Heidelberg: Springer, 1999. p.36-56.

[BRE04] BRESCIANI, P.; PERINI, A.; GIORGINI, P.; GIUNCHIGLIA, F.; MYLOPOULOS, J. **Tropos: An Agent-Oriented Software Development Methodology**. Journal of Autonomous Agents and Multi-Agent Systems, Netherlands: Springer, 8(3):203-236, 2004.

[CAS01] CASTRO, J.; KOLP, M.; MYLOPOULOS, J. **A Requirements-Driven Development Methodology**. In: 13th International Conference on Advanced Information Systems Engineering CAiSE 01, Interlaken, Switzerland, June 4-8, 2001. p. 108-123.

[CHO05] CHOREN, R.; LUCENA, C. **Modeling multi-agent systems with ANote**. Software And System Modeling, Berlim, v. 4, n. 2, p. 199-208, 2005.

[DEL99] DELOACH, S. A. **Multiagent Systems Engineering: A Methodology and Language for Designing Agent Systems**. In: Agent-Oriented Information Systems '99 (AOIS'99), Seattle WA, 1 May 1999. 10 p.

[DEL01] DELOACH, S. A.; WOOD, M. F.; SPARKMAN, C. H. **Multiagent Systems Engineering**. The International Journal of Software Engineering and Knowledge Engineering, Volume 11, no. 3, 231-258, June 2001.

[DEL01a] DELOACH, S. A. **Analysis and Design using MaSE and agentTool**. In: Proceedings of the 12th Midwest Artificial Intelligence and Cognitive Science Conference (MAICS 2001). Miami University, Oxford, Ohio, March 31 - April 1, 2001. 7 p.

[DRE07] DRESDEN OCL2 TOOLKIT. Desenvolvido por Software Technology Group at Technische Universität Dresden. Disponível em: http://dresden-ocl.sourceforge.net. Acesso em: 11 dez. 2007.

[ESC06] ESCOBAR, M.; LEMKE, A.; BLOIS, M. **SemantiCore 2006 – Permitindo o Desenvolvimento de Aplicações baseadas em Agentes na Web Semântica**. In: Second Workshop on Software Engineering for Agent-oriented Systems (SEAS 2006). XX Simpósio Brasileiro de Engenharia de Software (SBES 2006). Florianópolis, 2006. 11 p.

[ETZ95] ETZIONI, O.; WELD, D. S. **Inteligent Agents on the Internet: Fact, Fiction, and Forecast**. IEEE Expert, Seattle WA, 10 (4): 44-49, 1995.

[EST02] ESTEVA, M.; PADGET, J.; SIERRA, C. **Formalizing a language for institutions and norms**. In: Intelligent Agents VIII: 8th International Workshop, Meyer, J.-J. C. and Tambe, M., editors, ATAL 2001, Seattle, WA, USA, August 1-3. Revised Papers, volume 2333 of LNAI, pages 348.366, Berlin; Heidelberg: Springer, 2002.

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

[FER04] FERBER, J.; GUTKNECHT, O.; MICHEL, F. **From agents to organizations: an organizational view of multi-agent systems**. In: Agent-Oriented Software Engineering IV: 4th International Workshop, AOSE 2003, Giorgini, P., Müller, J. P., and Odell, J., editors, Melbourne, Australia, July 15, 2003, Revised Papers, volume 2935 of LNCS, pages 214.230, Berlin; Heidelberg: Springer, 2004.

[FIP07] FIPA ACL. Desenvolvido por Foundation for Intelligent Physical Agents. Disponível em: <http://www.fipa.org>. Acesso em: 02 nov. 2007.

[FIP07a] FIPA CONTRACT NET INTERACTION PROTOCOL SPECIFICATION. Desenvolvido por Foundation for Intelligent Physical Agents. Disponível em: <http://www.fipa.org/specs/fipa00029/SC00029H.html>. Acesso em: 08 jul. 2007.

[FRA96] FRANKLIN, S.; GRAESSER, A. **Is It an Agent or Just a Program? A Taxonomy for Autonomous Agents**. In: Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages. New York: Springer-Verlag, 1996. p. 21-35.

[GIL95] GILBERT, D.; APARICIO, M.; ATKINSON, B.; BRADY, S.; CICCARINO, J.; GROSOF, B.; O'CONNOR, P.; OSISEK, D.; PRITKO, S.; SPAGNA, R.; WILSON, L. **IBM Intelligent Agent Strategy**. IBM Corporation, 1995. 71p.

[GIO05] GIORGINI, P.; MYLOPOULOS, J.; PERINI, A.; SUSI, A. **The Tropos Metamodel and its Use**. Informatica, Italy, 29, 401-408, 2005.

[GEO87] GEORGEFF, M.; LANSKY, A. **Reactive Reasoning and Planning: An Experiment With a Mobile Robot**. In: Proceedings of the 1987 National Conference on Artificial Intelligence (AAAI 87), pages 677–682, Seattle, Washington, July 1987.

[GUD02] GUDGIN, M.; HADLEY, M.; MANDELSOHN, N.; MOREAU, J. e NIELSEN, H. (2002). **SOAP Version 1.2**, 2002-2006. Disponível em: <http://www.w3c.org/2000/xp/Group/2/06/LC/soap12-part1.html>. Acesso em: 20 Abril 2006.

[HEN05] HENDERSON-SELLERS, B.; GIORGINI, P. **Agent-Oriented Methodologies.** Hershey; London; Melbourne; Singapore: Idea Group, 2005. 413 p.

[HEN05a] HENDERSON-SELLERS, B.; TRAN, Q. N..; DEBENHAM, J. **An Etymological and Metamodel-Based Evaluation of the Terms "Goals and Tasks" in Agent-Oriented Methodologies**. Journal of Object Technology, 4(2):131-150, 2005.

[HOW01] HOWDEN, N.; RÖNNQUIST, R.; HODGSON, A.; LUCAS, A. **JACK: Summary of an Agent Infrastructure**. In: 5th International Conference on Autonomous Agents, Montreal, 2001. p. 251-257.

[IGL98] IGLESIAS, C. A.; GARIJO, M.; GONZALEZ, J.C.; VELASCO, J.R. **Analysis and Design of Multi-Agent Systems using MAS-CommonKADS**. In: Intelligent Agents IV (LNAI Volume 1365) Singh, M.P., Rao, A., Wooldridge, M.J., (eds.): Springer-Verlag: Berlin, Germany, P. 313-326, 1998.

[ING92] INGRAND, F.; GEORGEFF, M. and RAO, A. **An Architecture for Real-Time Reasoning and System Control**. IEEE Expert, New Jersey, USA, 7(6):34–44, 1992.

[JEN96] JENNINGS, N. R.; FARATIN, P.; JOHNSON, M. J.; O'BRIEN, P.; WIEGAND, M. E. **Using Intelligent Agents to Manage Business Processes**. In: Proceedings of First International Conference on The Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM96), London, UK, 345-360. Disponível na Internet em: <http://www.ecs.soton.ac.uk/~nrj/pubs.html>. Acesso em: 10 set. 2006.

[JEN98] JENNINGS, R.; WOOLDRIDGE, M. **Agent technology: foundations, applications, and markets**. Berlin; Heidelberg; New York; Barcelona; Budapest; Hong Kong; London; Milan; Paris; Santa Clara; Singapore; Tokio: Springer, 1998. 325 p.

[LOC07] PROJECT METRICS HELP – LINES OF CODE METRICS. Disponível em: <http://www.aivosto.com/project/help/pm-loc.html>. Acesso em 10 jan. 2008.

[MAC02] MACHADO, R.; BORDINI, R.. **Running AgentSpeak(L) agents on SIM AGENT**. In: Intelligent Agents VIII – Proceedings of the Eighth International Workshop on Agent Theories, Architectures, and Languages (ATAL-2001), J.-J. Meyer and M. Tambe, editors, August 1–3, 2001, Seattle, WA, number 2333 in Lecture Notes in Artificial Intelligence, pages 158–174, Berlin, 2002. Springer-Verlag.

[MÜL96] MÜLLER, J. P. **The design of Intelligent Agents: A Layered Approach**. Alemanha: Springer-Verlag, 1996. 227 p.

[NOR03] NORVIG, P.; RUSSEL, S. **Artificial Intelligence: A Modern Approach**. New Jersey: Prentice-Hall, 2003. 1132 p.

[NWA96] NWANA, H. S. **Software Agents: An Overview**. United Kingdom: Cambridge University Press, Knowledge Engineering Review, 11(3): 205-244, 1996.

[OCL07] UML 2.0 OCL SPECIFICATION. Desenvolvido por Object Management Group. Disponível em: < www.omg.org/docs/ptc/05-06-06.pdf >. Acesso em: 28 dez. 2007.

[ODE00] ODELL, J. **Agent Technology Green Paper**. Version 1.0, Agent Working Group OMG Document, Needham, MA, 2000. 70 p.

[PAD02] PADGHAM, L.; WINIKOFF, M. **Prometheus: A Methodology for Developing Intelligent Agents**. In: Proceedings of the the Third International Workshop on Agent-Oriented Software Engineering at AAMAS'02, Bologna, 2002. p. 174-185.

[PAV05] PAVÓN, J.; GOMEZ-SANZ, J.J.; FUENTES, R. 2005. **The INGENIAS Methodology and Tools**. In: Agent-Oriented Methodologies. Henderson-Sellers, B. and Giorgini, P., editors. Hershey; London; Melbourne, Singapore: Idea Group, chapter IX, pp. 236-276.

[PEN03] PENDER, T. **UML Bible**. Indianapolis: Wiley Publishing, 2003. 984 p.

[PDT06] PDT – PROMETHEUS DESIGN TOOL. Desenvolvido por RMIT Intelligent Agents Group. Disponível em: <http://www.cs.rmit.edu.au/agents/pdt/>. Acesso em: 06 set. 2006.

[POK05] POKAHR, A.; BRAUBACH, L.; LAMERSDORF, W. **Jadex: A BDI Reasoning Engine**. In: Multi-Agent Programming, Bordini, R., Dastani, M., Dix, J. and Seghrouchni A. (eds.). USA: Springer Science + Business Media Inc., P. 149-174, 2005.

[RAO92] RAO, A.; GEORGEFF, M. **An Abstract Architecture for Rational Agents**. In: Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning (KR'92), C. Rich, W. Swartout and B. Nebel (editors), Morgan Kaufmann Publishers, Toulose, France, 1992. p. 439-449.

[RAO95] RAO, A. S.; GEORGEFF, M. P. **BDI-agents: from theory to practice**. In: Proceedings of the First Intl. Conference on Multiagent Systems, San Francisco, 1995. p. 312-319.

[SAN06] SANTOS, D. R. **Estudo Comparativo de Metodologias de Desenvolvimento de Sistemas Multiagentes**. Trabalho Individual I (Mestrado em Ciência da Computação). Programa de Pós-Graduação em Ciência da Computação - PPGCC, PUCRS, Porto Alegre, 2006. 89 p.

[SAN06a] SANTOS, D. R. **Proposta de um Meta-modelo para a Modelagem Interna de Agentes de Software**. Trabalho Individual II (Mestrado em Ciência da Computação). Programa de Pós-Graduação em Ciência da Computação - PPGCC, PUCRS, Porto Alegre, 2006. 40 p.

[SAN06b] SANTOS, D. R.; RIBEIRO, M. B.; BASTOS, R. M. **A Comparative Study of Multi-Agent Systems Development Methodologies**. In: Second Workshop on Software Engineering for Agent-oriented Systems (SEAS 2006). XX Simpósio Brasileiro de Engenharia de Software (SBES 2006). Florianópolis, 2006. 12 p.

[SAN07] SANTOS, D. R.; BLOIS, M.; BASTOS, R. **Developing a Conference Management System with the Multi-agent Systems Unified Process – A Case Study**. In: 8th International Workshop on AGENT ORIENTED SOFTWARE ENGINEERING. AAMAS 2007. Honolulu, Hawai'i, 2007. p. 212-224.

[SHO93] SHOHAM, Y. **Agent-Oriented Programming**. Journal of Artificial Intelligence, 60(1): 51-92, 1993.

[SHO97] SHOHAM, Y. **An Overview of Agent-Oriented Programming**. In: Software Agents. Bradshaw, J. M., ed., Cambridge, MA, USA: AAAI Press/The MIT Press, p. 271 – 290, 1997.

[SIL04] SILVA, V.T. **Uma Linguagem de Modelagem para Sistemas Multiagentes baseada em um Framework Conceitual para Agentes e Objetos**. Tese de Doutorado. Programa de Pós-graduação em Informática da PUC-Rio, Rio de Janeiro, 2004. 252 p.

[SIL04a] SILVA, V.T.; CHOREN, R.C.; LUCENA, C.J.P. **A UML Based Approach for Modeling and Implementing Multi-Agent Systems**. In: AAMAS 2004: 914-921, New York, 2004.

[SIL07] SILVA, V.T.; LUCENA, C.J.P. **Modeling Multi-Agent Systems**. Communications of the ACM, New York, Vol. 50, Nº.5, 2007. p. 103-108.

[TAC06] TRADING AGENT COMPETITION. Disponível em: <http://www.sics.se/tac>. Acesso em 12 jun. 2006.

[TOV07] TOVAL, A.; REQUENA, V.; FERNÁNDEZ, J. **OCL Tools**. Software Engineering Research Group. Department of Informatics, and Systems. Faculty of Informatics. University of Murcia. Disponível em: <http://www.um.es/giisw/ocltools/about.htm>. Acesso em: 11 dez. 2007.

[TRO06] TROPOS PROJECT. Disponível em: <http://www.troposproject.org/>. Acesso em: 06 set. 2006.

[USE07] USE. Desenvolvido por Universidade de Bremen. Disponível em: <http://www.db.informatik.uni-bremen.de/projects/USE/>. Acesso em: 11 dez. 2007.

[VEL07] THE APACHE VELOCITY PROJECT. Disponível em: <http://velocity.apache.org/engine/index.html>. Acesso em: 11 dez. 2007.

[VEL07a] THE APACHE VELOCITY TOOLS PROJECT. Disponível em: <http://velocity.apache.org/tools/releases/1.4/>. Acesso em: 11 dez. 2007.

[WAR03] WARMER, J.; KLEPPE, A. **The Object Constraint Language Second Edition Getting your Models Ready for MDA**. Boston: Addison-Wesley, 2003. 240 p.

[WAR07] WARMER, J.; KLEPPE, A. **Octopus**. Disponível em: <http://www.klasse.nl/octopus/index.html>. Acesso em: 11 dez. 2007.

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

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

[WOO02a] WOOLDRIDGE, M. **Reasoning about rational agents**. Cambridge: MIT Press, 2000. 240 p.

[YU95] YU, E. **Modelling Strategic Relationships for Process Reengineering**. Ph.D. thesis. Dept. of Computer Science, Univ. of Toronto, Canada, 1995. 181 p.

[ZAM04] ZAMBONELLI, F.; BERGENTI, F.; GLEIZES, M. **Methodologies and Software Engineering for Agent Systems**. USA: Kluwer Academic Publishers, 2004. 505 p.

APÊNDICE I: RESTRIÇÕES APLICADAS AO META-MODELO

```
constraints

context Agent inv MandatoryAgentName: self.name.size()>0

context Agent inv MandatoryAgentState: self.state.size()>0

context Resource inv MandatoryResourceName: self.name.size()>0

context Resource inv MandatoryResourceType: self.type.size()>0

context Role inv MandatoryRoleName: self.name.size()>0

context Role inv MandatoryRoleSociety: self.society.size()>0

context Plan inv MandatoryPlanName: self.name.size()>0

context Plan inv MandatoryPlanState: self.state.size()>0

context Goal inv MandatoryGoalName: self.name.size()>0

context Goal inv MandatoryGoalState: self.state.size()>0

context Action inv MandatoryActionName: self.name.size()>0

context Message inv MandatoryMessageId: self.id.size()>0

context Message inv MandatoryMessageSource: self.source.size()>0

context Message inv MandatoryMessageTarget: self.target.size()>0

context Message inv MandatoryMessageType: self.type.size()>0

context Message inv MandatoryMessageLanguage: self.language.size()>0

context Protocol inv MandatoryProtocolName: self.name.size()>0

context Field inv MandatoryFieldId: self.id.size()>0

context Field inv MandatoryFieldName: self.name.size()>0

context Field inv MandatoryFieldRequired: self.required.size()>0

context Perceptron inv MandatoryPerceptronName: self.name.size()>0

context Perceptron inv MandatoryPerceptronType: self.type.size()>0

context Event inv MandatoryEventName: self.name.size()>0

context Belief inv MandatoryBeliefId: self.id.size()>0

context Belief inv MandatoryBeliefDescription: self.description.size()>0

context Operator inv MandatoryOperatorId: self.id.size()>0

context Operator inv MandatoryOperatorSymbol: self.symbol.size()>0
```

```
context Agent inv UniqueAgentName: Agent.allInstances->forAll(other|self.name = other.name implies self = other)

context Agent inv AgentState: Agent.allInstances->forAll(self.state = 'created' xor self.state= 'execution' xor self.state = 'ready' xor self.state = 'blocked' xor self.state = 'finished')

context Field inv FieldRequired: self.required = 'True' xor self.required = 'False'

context Resource inv UniqueResourceName: Resource.allInstances->forAll(other|self.name = other.name implies self = other)

context Role inv UniqueSocietyName:Role.allInstances->forAll(other | self.society = other.society implies self=other xor self.name <> other.name)

context Role  inv Actions: self.action->notEmpty() or self.mandatoryAction->notEmpty()

context Role inv ActionsPlan: (self.action->union(self.mandatoryAction))->includesAll(self.goal.plan.action)

context Plan inv UniquePlanName: Plan.allInstances->forAll(other|self.name = other.name implies self = other)

context Goal inv UniqueGoalName: Goal.allInstances->forAll(other|self.name = other.name implies self = other)

context Action inv UniqueActionName: Action.allInstances->forAll(other|self.name = other.name implies self = other)

context Message inv UniqueMessageId: Message.allInstances->forAll(other|self.id = other.id implies self = other)

context Protocol inv UniqueProtocolName: Protocol.allInstances->forAll(other|self.name = other.name implies self = other)

context Field inv UniqueFieldId: Field.allInstances->forAll(other|self.id = other.id implies self = other)

context Perceptron inv UniquePerceptronName: Perceptron.allInstances->forAll(other|self.name = other.name implies self = other)

context Event inv UniqueEventName: Event.allInstances->forAll(other|self.name = other.name implies self = other)

context Belief inv UniqueBeliefId: Belief.allInstances->forAll(other|self.id = other.id implies self = other)
```

APÊNDICE II: REPRESENTAÇÃO EM XML DO EXEMPLO

```
<?xml version = '1.0' encoding = 'UTF-8' ?>
<metamodel>
<concept def='Agent' name='Cliente' state='created' environment='ComponentesEnvironment'/>
<concept def='Agent' name='Fornecedor' state='created' environment='ComponentesEnvironment'/>
<concept def='Resource' name='nomeComponente' type='String' value=''/>
<concept def='Resource' name='marcaComponente' type='String' value=''/>
<concept def='Resource' name='precoComponente' type='String' value=''/>
<concept def='Role' name='Comprador' society='TacSCM'/>
<concept def='Role' name='Vendedor' society='TacSCM'/>
<concept def='Goal' name='ComprarComponentes' state='CompraEfetuada'/>
<concept def='Goal' name='VenderComponentes' state='PropostaEnviada'/>
<concept def='Plan' name='EfetuarCompra' state='PedidoPronto'/>
<concept def='Plan' name='EfetuarVenda' state='PedidoRecebido'/>
<concept def='Term' id='K' description='Pedido Enviar TRUE'/>
<concept def='InternalEvent' name='EventoEnviarPedido' clock='5'/>
<concept def='Action' name='EnviarPedido'/>
<concept def='Term' id='L' description='Pedido Enviado TRUE'/>
<concept def='Action' name='EnviarProposta'/>
<concept def='Term' id='S' description='Proposta Enviada TRUE'/>
<concept def='Action' name='ConfirmarCompra'/>
<concept def='Term' id='A' description='Pedido Nome Placa-Mae'/>
<concept def='Term' id='B' description='Pedido Marca ASUS'/>
<concept def='Term' id='C' description='Pedido Nome Disco-Rigido'/>
<concept def='Term' id='D' description='Pedido Marca Samsung'/>
<concept def='Term' id='E' description='Pedido Nome Processador'/>
<concept def='Term' id='F' description='Pedido Marca Intel'/>
<concept def='Term' id='G' description='Pedido Nome DVD'/>
<concept def='Term' id='H' description='Pedido Marca LG'/>
<concept def='Term' id='I' description='Pedido Nome Placa-Video'/>
<concept def='Term' id='J' description='Pedido Marca Creative'/>
<concept def='Sentence' id='PED01' description='Pedido' beliefA='A' operator='AND' beliefB='B'/>
<concept def='Sentence' id='PED02' description='Pedido' beliefA='C' operator='AND' beliefB='D'/>
<concept def='Sentence' id='PED03' description='Pedido' beliefA='E' operator='AND' beliefB='F'/>
<concept def='Sentence' id='PED04' description='Pedido' beliefA='G' operator='AND' beliefB='H'/>
<concept def='Sentence' id='PED05' description='Pedido' beliefA='I' operator='AND' beliefB='J'/>
<concept def='Message' id='Message1' source='Cliente' target='Fornecedor' type='cfp' language='fipa-sl'/>
<concept def='Field' id='Message1_Content' name='content' value='' required='True'/>
```

```
<concept def='Field' id='Message1_Ontology' name='ontology' value='componentes' required='False'/>
<concept def='Perceptron' name='AvaliaPedidos' type='cfp'/>
<concept def='ExternalEvent' name='EventoRecebePedido'/>
<concept def='Term' id='M' description='Pedido Recebido TRUE'/>
<concept def='Term' id='N' description='Proposta Preco 100'/>
<concept def='Term' id='O' description='Proposta Preco 200'/>
<concept def='Term' id='P' description='Proposta Preco 300'/>
<concept def='Term' id='Q' description='Proposta Preco 400'/>
<concept def='Term' id='R' description='Proposta Preco 500'/>
<concept def='Message' id='Message2' source='Fornecedor' target='Cliente' type='propose' language='fipa-sl'/>
<concept def='Field' id='Message2_Content' name='content' value='' required='True'/>
<concept def='Field' id='Message2_Ontology' name='ontology' value='componentes' required='False'/>
<concept def='Perceptron' name='AvaliaPropostas' type='propose'/>
<concept def='ExternalEvent' name='EventoRecebeProposta'/>
<concept def='Term' id='T' description='Proposta Recebida TRUE'/>
<concept def='Term' id='U' description='Pedido Preco Acessivel'/>
<concept def='Term' id='V' description='Compra Aceita TRUE'/>
<concept def='Rule' id='rule1' description='DecisaoCompra' antecedent='U' consequent='V'/>
<concept def='Message' id='Message3' source='Cliente' target='Fornecedor' type='acceptProposal'
language='fipa-sl'/>
<concept def='Field' id='Message3_Content' name='content' value='' required='True'/>
<concept def='Perceptron' name='AvaliaCompras' type='acceptProposal'/>
<concept def='ExternalEvent' name='EventoRecebeCompra'/>
<concept def='Term' id='X' description='Compra Confirmacao TRUE'/>
<concept def='Term' id='Y' description='Venda Efetuada TRUE'/>
<concept def='Protocol' name='ContractNet'/>
<relationship def='Agent_Resource' idA='Cliente' idB='nomeComponente'/>
<relationship def='Agent_Resource' idA='Cliente' idB='marcaComponente'/>
<relationship def='Agent_Resource' idA='Fornecedor' idB='nomeComponente'/>
<relationship def='Agent_Resource' idA='Fornecedor' idB='marcaComponente'/>
<relationship def='Agent_Resource' idA='Fornecedor' idB='precoComponente'/>
<relationship def='Agent_Role' idA='Cliente' idB='Comprador'/>
<relationship def='Agent_Role' idA='Fornecedor' idB='Vendedor'/>
<relationship def='Role_Goal' idA='Comprador' idB='ComprarComponentes'/>
<relationship def='Role_Goal' idA='Vendedor' idB='VenderComponentes'/>
<relationship def='Plan_Goal' idA='EfetuarCompra' idB='ComprarComponentes'/>
<relationship def='Role_MandatoryAction' idA='Comprador' idB='EnviarPedido'/>
<relationship def='Role_Action' idA='Comprador' idB='ConfirmarCompra'/>
<relationship def='Plan_Action' idA='EfetuarCompra' idB='ConfirmarCompra'/>
<relationship def='Role_Action' idA='Vendedor' idB='EnviarProposta'/>
```

```
<relationship def='Plan_Goal' idA='EfetuarVenda' idB='VenderComponentes'/>
<relationship def='Plan_Action' idA='EfetuarVenda' idB='EnviarProposta'/>
<relationship def='Agent_Belief' idA='Cliente' idB='A'/>
<relationship def='Agent_Belief' idA='Cliente' idB='B'/>
<relationship def='Agent_Belief' idA='Cliente' idB='C'/>
<relationship def='Agent_Belief' idA='Cliente' idB='D'/>
<relationship def='Agent_Belief' idA='Cliente' idB='E'/>
<relationship def='Agent_Belief' idA='Cliente' idB='F'/>
<relationship def='Agent_Belief' idA='Cliente' idB='G'/>
<relationship def='Agent_Belief' idA='Cliente' idB='H'/>
<relationship def='Agent_Belief' idA='Cliente' idB='I'/>
<relationship def='Agent_Belief' idA='Cliente' idB='J'/>
<relationship def='Agent_Belief' idA='Cliente' idB='PED01'/>
<relationship def='Agent_Belief' idA='Cliente' idB='PED02'/>
<relationship def='Agent_Belief' idA='Cliente' idB='PED03'/>
<relationship def='Agent_Belief' idA='Cliente' idB='PED04'/>
<relationship def='Agent_Belief' idA='Cliente' idB='PED05'/>
<relationship def='Agent_InternalEvent' idA='Cliente' idB='EventoEnviarPedido'/>
<relationship def='Event_Belief' idA='EventoEnviarPedido' idB='K'/>
<relationship def='Belief_Action' idA='K' idB='EnviarPedido'/>
<relationship def='Action_Belief' idA='EnviarPedido' idB='L'/>
<relationship def='Action_Message' idA='EnviarPedido' idB='Message1'/>
<relationship def='Message_Field' idA='Message1' idB='Message1_Content'/>
<relationship def='Message_Field' idA='Message1' idB='Message1_Ontology'/>
<relationship def='Agent_Perceptron' idA='Fornecedor' idB='AvaliaPedidos'/>
<relationship def='Perceptron_Message' idA='AvaliaPedidos' idB='Message1'/>
<relationship def='Perceptron_ExternalEvent' idA='AvaliaPedidos' idB='EventoRecebePedido'/>
<relationship def='Event_Belief' idA='EventoRecebePedido' idB='A'/>
<relationship def='Event_Belief' idA='EventoRecebePedido' idB='B'/>
<relationship def='Event_Belief' idA='EventoRecebePedido' idB='PED01'/>
<relationship def='Event_Belief' idA='EventoRecebePedido' idB='E'/>
<relationship def='Belief_Plan' idA='M' idB='EfetuarVenda'/>
<relationship def='Agent_Belief' idA='Fornecedor' idB='N'/>
<relationship def='Agent_Belief' idA='Fornecedor' idB='O'/>
<relationship def='Agent_Belief' idA='Fornecedor' idB='P'/>
<relationship def='Agent_Belief' idA='Fornecedor' idB='Q'/>
<relationship def='Agent_Belief' idA='Fornecedor' idB='R'/>
<relationship def='Action_Belief' idA='EnviarProposta' idB='Q'/>
<relationship def='Action_Message' idA='EnviarProposta' idB='Message2'/>
<relationship def='Message_Field' idA='Message2' idB='Message2_Content'/>
```

```
<relationship def='Message_Field' idA='Message2' idB='Message2_Ontology'/>
<relationship def='Agent_Perceptron' idA='Cliente' idB='AvaliaPropostas'/>
<relationship def='Perceptron_Message' idA='AvaliaPropostas' idB='Message2'/>
<relationship def='Perceptron_ExternalEvent' idA='AvaliaPropostas' idB='EventoRecebeProposta'/>
<relationship def='Event_Belief' idA='EventoRecebeProposta' idB='Q'/>
<relationship def='Event_Belief' idA='EventoRecebeProposta' idB='T'/>
<relationship def='Agent_Belief' idA='Cliente' idB='U'/>
<relationship def='Agent_Belief' idA='Cliente' idB='V'/>
<relationship def='Agent_Belief' idA='Cliente' idB='rule1'/>
<relationship def='Belief_Plan' idA='V' idB='EfetuarCompra'/>
<relationship def='Action_Message' idA='ConfirmarCompra' idB='Message3'/>
<relationship def='Message_Field' idA='Message3' idB='Message3_Content'/>
<relationship def='Action_Belief' idA='ConfirmarCompra' idB='X'/>
<relationship def='Agent_Perceptron' idA='Fornecedor' idB='AvaliaCompras'/>
<relationship def='Perceptron_Message' idA='AvaliaCompras' idB='Message3'/>
<relationship def='Perceptron_ExternalEvent' idA='AvaliaCompras' idB='EventoRecebeCompra'/>
<relationship def='Event_Belief' idA='EventoRecebeCompra' idB='Y'/>
<relationship def='Protocol_Message' idA='ContractNet' idB='Message1'/>
<relationship def='Protocol_Message' idA='ContractNet' idB='Message2'/>
<relationship def='Protocol_Message' idA='ContractNet' idB='Message3'/>
<relationship def='Message_Message' idA='Message3' idB='Message2'/>
<relationship def='Message_Message' idA='Message2' idB='Message1'/>
</metamodel>
```

APÊNDICE III: ARQUIVOS E CÓDIGOS GERADOS

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>
```

Arquivo semanticoreconfig.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<semanticore>
<agent env="ComponentesEnvironment" name="Cliente" class="application.Cliente" arg=""/>
<agent env="ComponentesEnvironment" name="Fornecedor" class="application.Fornecedor" arg=""/>
</semanticore>
```

Classe AvaliaPedidosSensor

```
public class AvaliaPedidosSensor extends Sensor {
	public AvaliaPedidosSensor(String sName) {
		super(sName);
	}
	public Object evaluate(Object arg0) {
		String nameLocal = this.getSensorialComponent().getOwner().getName();
		if (arg0 instanceof CfpMessage) {
			CfpMessage m = (CfpMessage) arg0;
			boolean a = false;
			for (int i = 0; i < m.getTo().length; i++) {
				System.out.println("--->" + m.getTo()[i].toString());
				if (m.getTo()[i].equals(this.getSensorialComponent().getOwner()
				.getName()))
					a = true;
			}
			if (!a) {
				System.out.println("> " + nameLocal + ": Recebi algo estranho 1 ");
				return null;
			}
			return m;
		}
		return null;
	}
}
```

Classe AvaliaPropostasSensor

```
public class AvaliaPropostasSensor extends Sensor {
	public AvaliaPropostasSensor(String sName) {
		super(sName);
	}
	public Object evaluate(Object arg0) {
		String nameLocal = this.getSensorialComponent().getOwner().getName();
		if (arg0 instanceof ProposeMessage) {
			ProposeMessage m = (ProposeMessage) arg0;
			boolean a = false;
			for (int i = 0; i < m.getTo().length; i++) {
				System.out.println("--->" + m.getTo()[i].toString());
				if (m.getTo()[i].equals(this.getSensorialComponent().getOwner()
				.getName()))
					a = true;
			}
			if (!a) {
				System.out.println("> " + nameLocal + ": Recebi algo estranho 1 ");
				return null;
			}
			return m;
		}
		return null;
	}
}
```

Classe AvaliaComprasSensor

```
public class AvaliaComprasSensor extends Sensor {
	public AvaliaComprasSensor(String sName) {
		super(sName);
	}
	public Object evaluate(Object arg0) {
		String nameLocal = this.getSensorialComponent().getOwner().getName();
		if (arg0 instanceof AcceptProposalMessage) {
			AcceptProposalMessage m = (AcceptProposalMessage) arg0;

			boolean a = false;
			for (int i = 0; i < m.getTo().length; i++) {
				System.out.println("--->" + m.getTo()[i].toString());
				if (m.getTo()[i].equals(this.getSensorialComponent()
				.getOwner().getName()))
					a = true;
			}
			if (!a) {
				System.out.println("> " + nameLocal + ": Recebi algo estranho 1 ");
					return null;
				}
				return m;
		}
		return null;
	}
}
```

Classe CfpMessage

```
public class CfpMessage extends SemanticMessage
{
	private String language;
	private String protocol;
	private String ontology;
	public CfpMessage ( String from, String to, Object content)
	{
		super(from,to,content);
	}
	public CfpMessage ( String from, String to, Object content, String language, String protocol,
	String ontology)
	{
		super(from,to,content);
		init ( "", from, new String [ ] { to }, null, content, getTime ( ), null, language, protocol,
		ontology );
	}
	public CfpMessage ( String from, String [ ] to, Object content, String language, String protocol,
	String ontology )
	{
		super(from,to,content);
		init ( "", from, to, null, content, getTime ( ), null, language, protocol , ontology);
	}
	private void init ( Object subject, Object from, Object to, Object domain, Object content, Object
	timestamp, Object machine, String language, String protocol, String ontology )
	{
		this.domainTo = ( String ) domain;
		this.subject = ( String ) subject;
		this.from = ( String ) from;
		this.to = ( String [ ] ) to;
		this.content = content;
		this.machine = ( String ) machine;
		try
		{
			this.timestamp = Long.parseLong ( ( String ) timestamp );
		}
		catch ( Exception e )
		{
			this.timestamp = getTime ( );
		}
```

```
                this.language = language;
                this.protocol = protocol;
                this.ontology = ontology;
        }
        public String toString ( )
        {
        return "CfpMessage\nFrom:" + from + "\nTo:" + to [ 0 ] + "\nContent:" + content +
        "\nLanguage:" + language + "\nProtocol:" + protocol+ "\nOntology:" + ontology;
        }
        public long getTimestamp ( )
        {
                return timestamp;
        }
}
```

Classe ProposeMessage

```
public class ProposeMessage extends SemanticMessage {
        private String language;
        private String protocol;
        private String ontology;
        public ProposeMessage ( String from, String to, Object content)
        {
                super(from,to,content);
        }
        public ProposeMessage ( String from, String to, Object content, String language, String
        protocol, String ontology)
        {
                super(from,to,content);
                init ( "", from, new String [ ] { to }, null, content, getTime ( ), null, language,
                protocol, ontology );
        }
        public ProposeMessage ( String from, String [ ] to, Object content, String language,
        String protocol, String ontology )
        {
                super(from,to,content);
                init ( "", from, to, null, content, getTime ( ), null, language, protocol, ontology );
        }
        private void init ( Object subject, Object from, Object to, Object domain, Object content,
        Object timestamp, Object machine, String language, String protocol, String ontology )
        {
                this.domainTo = ( String ) domain;
                this.subject = ( String ) subject;
                this.from = ( String ) from;
                this.to = ( String [ ] ) to;
                this.content = content;
                this.machine = ( String ) machine;
                try
                {
                        this.timestamp = Long.parseLong ( ( String ) timestamp );
                }
                catch ( Exception e ){
                this.timestamp = getTime ( );
                }
```

```
                this.language = language;
                this.protocol = protocol;
                this.ontology = ontology;
        }
        public String toString ( )
        {
                return "ProposeMessage\nFrom:" + from + "\nTo:" + to [ 0 ] + "\nContent:" +
                content +"\nLanguage:" + language + "\nProtocol:" + protocol+ "\nOntology:"
                + ontology;
        }
        public long getTimestamp ( )
        {
                return timestamp;
        }
}
```

Classe AcceptProposalMessage

```
public class AcceptProposalMessage extends SemanticMessage
{
	private String language;
	private String protocol;
	public AcceptProposalMessage ( String from, String to, Object content)
	{
		super(from,to,content);
	}
	public AcceptProposalMessage ( String from, String to, Object content, String language, String
	protocol)
	{
		super(from,to,content);
		init ( "", from, new String [ ] { to }, null, content, getTime ( ), null, language, protocol );
	}
	public AcceptProposalMessage ( String from, String [ ] to, Object content, String language, String
	protocol )
	{
		super(from,to,content);
		init ( "", from, to, null, content, getTime ( ), null, language, protocol );
	}
	private void init ( Object subject, Object from, Object to, Object domain, Object content, Object
	timestamp, Object machine, String language, String protocol )
	{
		this.domainTo = ( String ) domain;
		this.subject = ( String ) subject;
		this.from = ( String ) from;
		this.to = ( String [ ] ) to;
		this.content = content;
		this.machine = ( String ) machine;
		try
		{
			this.timestamp = Long.parseLong ( ( String ) timestamp );
		}
		catch ( Exception e )
		{
			this.timestamp = getTime ( );
		}
		this.language = language;
		this.protocol = protocol;
```

```
        }
        public String toString ( )
        {
                return "AcceptProposalMessage\nFrom:" + from + "\nTo:" + to [ 0 ] + "\nContent:" +
                content +"\nLanguage:" + language + "\nProtocol:" + protocol;
        }
        public long getTimestamp ( )
        {
                return timestamp;
        }
}
```

Classe Fornecedor

```
public class Fornecedor extends SemanticAgent
{
	private String nomeComponente;
	private String marcaComponente;
	private String precoComponente;
	public Fornecedor ( Environment env, String agentName, String arg )
	{
		super ( env, agentName, arg );
	}
	protected void setup ( )
	{
		SimpleFact n = new SimpleFact("Proposta","Preco","100");
		SimpleFact o = new SimpleFact("Proposta","Preco","200");
		SimpleFact p = new SimpleFact("Proposta","Preco","300");
		SimpleFact q = new SimpleFact("Proposta","Preco","400");
		SimpleFact r = new SimpleFact("Proposta","Preco","500");
		addFact(n);
		addFact(o);
		addFact(p);
		addFact(q);
		addFact(r);
		int random = (int) (Math.random() * 5) + 1;
		if(random==1){
			this.setPrecoComponente(n.getObject());
		} else if (random==2){
			this.setPrecoComponente(o.getObject());
		} else if (random==3){
			this.setPrecoComponente(p.getObject());
		} else if (random ==4){
			this.setPrecoComponente(q.getObject());
		} else if (random==5){
			this.setPrecoComponente(r.getObject());
		}
		addSensor (new UnicastSensor());
		addSensor ( new AvaliaPedidosSensor("AvaliaPedidos") );
		addSensor ( new AvaliaComprasSensor("AvaliaCompras") );
		setDecisionEngine ( new FornecedorDecisorio ( ) );
		EfetuarVenda efetuarVenda = new EfetuarVenda ( "EfetuarVenda" );
		efetuarVenda.addAction ( new EnviarProposta ( ) );
```

```
                addActionPlan ( efetuarVenda );
        }
        public String getNomeComponente() {
                return nomeComponente;
        }
        public void setNomeComponente(String nomeComponente) {
                this.nomeComponente = nomeComponente;
        }
        public String getMarcaComponente() {
                return marcaComponente;
        }
        public void setMarcaComponente(String marcaComponente) {
                this.marcaComponente = marcaComponente;
        }
        public String getPrecoComponente() {
                return precoComponente;
        }
        public void setPrecoComponente(String precoComponente) {
                this.precoComponente = precoComponente;
        }
}
```

Classe ClienteDecisorio

```
public class ClienteDecisorio extends DecisionEngine
{
	public Vector<Object> decide ( Object facts )
	{
		Vector<Object> vector = new Vector<Object> ( );
		SimpleFact u = null;
		System.out.println ( "decide" );
		if ( facts instanceof SemanticMessage )
		{
			System.out.println ( "Recebi:\n" + ( ( SemanticMessage ) facts ).toString ( ) );
			if ( ( ( SemanticMessage ) facts ).getContent ( ).toString ( ).equals ( "start" ) ){
				try {
					Thread.sleep(5000);
				} catch (InterruptedException e) {
					e.printStackTrace();
				}
				vector.add ( new ExecuteProcessAction ( "EnviarPedido" ) );
			}
			if ( ( ( SemanticMessage ) facts ).getContent ( ) instanceof SimpleFact){
				SimpleFact m =  (SimpleFact) ((( SemanticMessage ) facts ).getContent ( ));
				if ( m.getSubject().equalsIgnoreCase("Proposta")){
					SimpleFact g = new SimpleFact("Proposta","Recebida","TRUE");
					((Cliente)this.getOwner()).addFact(g);
					System.out.println("Recebida proposta com preco "+m.getObject());
					if(Integer.parseInt(m.getObject())<500){
						u = new SimpleFact("Pedido","Preco","Acessivel");
					} else{
						System.out.println("Proposta com preco "+m.getObject()+"
						recusada.");
						return null;
					}
					Hashtable<String, Rule> rules = getRules();
					if(u.getSubject().equalsIgnoreCase(((SimpleFact)((Rule)rules
					.get("DecisaoCompra")).getFact()).getSubject())
					&&u.getPredicate().equalsIgnoreCase(((SimpleFact)((Rule)rules
					.get("DecisaoCompra")).getFact()).getPredicate())
					&& u.getObject().equalsIgnoreCase(((SimpleFact)((Rule)rules
					.get("DecisaoCompra")).getFact()).getObject()))
					{
```

```
                                        this.getOwner().addFact(((Rule)rules
                                        .get("DecisaoCompra")).getConsequence());
                                }
                                Hashtable<String, Fact> agentFacts = this.getFacts();
                                Collection<Fact> values = agentFacts.values();
                                 for (Iterator<Fact> iter = values.iterator(); iter.hasNext();) {
                                        Fact v = (Fact) iter.next();
                                        if(v instanceof SimpleFact){
                                                if(((SimpleFact)v).getSubject()
                                                .equalsIgnoreCase("Compra")&&
                                                 ((SimpleFact)v).getPredicate()
                                                .equalsIgnoreCase("Aceita")&&
                                                 ((SimpleFact)v).getObject()
                                                .equalsIgnoreCase("TRUE")){
                                                 Goal comprarComponentes
                                                 = new Goal(this.getOwner(),null);
                                                 for(intj=0;j<this.getOwner().
                                                 getExecutionComponent()
                                                .getActionPlans().size();j++){
                                                        if(((ActionPlan)this.getOwner()
                                                        .getExecutionComponent()
                                                        .getActionPlans()
                                                        .get(j)).getName()
                                                        .equalsIgnoreCase("EfetuarCompra")){
                                                                comprarComponentes.setPlan
                                                                ((ActionPlan)this.getOwner()
                                                                .getExecutionComponent().
                                                                getActionPlans().get(j));
                                                         }
                                                    }
                                                    comprarComponentes.start();
                                            }
                                        }
                                }
                        }
                }
                return vector;
        }
}
```

Classe EfetuarVenda

```
public class EfetuarVenda extends ActionPlan {
	public EfetuarVenda(String arg0) {
		super(arg0);
	}
}
```

Classe EfetuarCompra

```
public class EfetuarCompra extends ActionPlan {
	public EfetuarCompra(String arg0) {
		super(arg0);
	}
}
```

Classe EnviarProposta

```
public class EnviarProposta extends Action{
        public EnviarProposta ( )
        {
                super ( "EnviarProposta", null, null );
        }
        public void exec() {
                String precoComponente = ((Fornecedor)this.getOwner()).getPrecoComponente();
                SimpleFact proposta = new SimpleFact("Proposta","Preco",precoComponente);
                ProposeMessage message2 = new ProposeMessage ( "Fornecedor", "Cliente", proposta,
                "fipa-sl", "contractNet","componentes" );
                transmit (message2);
                SimpleFact s = new SimpleFact("Proposta","Enviada","TRUE");
                this.getOwner().addFact(s);
        }
}
```

Classe ConfirmarCompra

```
public class ConfirmarCompra extends Action{
        public ConfirmarCompra ( )
        {
                super ( "ConfirmarCompra", null, null );
        }
        public void exec() {
                SimpleFact v = new SimpleFact("Compra","Aceita","TRUE");
                this.getOwner().addFact(v);
                AcceptProposalMessage message3 = new AcceptProposalMessage ( "Cliente",
                "Fornecedor", v, "fipa-sl","contractNet" );
                transmit (message3);
                SimpleFact x = new SimpleFact("Compra","Confirmacao","TRUE");
                this.getOwner().addFact(x);
        }
}
```