Processo de Desenvolvimento de uma Ferramenta Gráfica de Apoio a Metodologia Prometheus AEOlus

Rafhael R. Cunha [1], Diana F. Adamatti[1], Cléo Z. Billa[1]

[1]Centro de Ciências Computacionais – Universidade Federal do Rio Grande (FURG)
Av. Itália km 8 – Bairro Carreiros – Rio Grande - RS - Brasil

{rcrafhaelrc,dianaada,cleo.billa}@gmail.com

Resumo. *A Engenharia de Software (ES) é uma área de engenharia que busca a construção de softwares com qualidade, utilizando métodos e respeitando prazos. Contudo, suas técnicas tradicionais não suportam completamente a demanda no desenvolvimento de Sistemas Multiagente, originando uma subárea, denominada Agent Oriented Software Engineering (AOSE). Ainda não existe uma padronização para esta subárea, resultando em diversas metodologias desenvolvidas por motivos distintos. Outro fator predominante para a instabilidade dessa subárea, consiste em suas ferramentas de apoio serem limitadas no processo de geração automática de código para plataformas específicas de desenvolvimento multiagente. O intuito principal deste trabalho é desenvolver uma ferramenta para apoiar a metodologia Prometheus AEOlus, permitindo que o usuário desenvolva os diagramas presentes na especificação da metodologia. Adicionalmente, como objetivo secundário, foi elaborado um mecanismo capaz de percorrer todas as informações definidas pelo usuário e realizar a geração automática de código para a linguagem agentspeak, que é aderente a plataforma de desenvolvimento Jason. A ferramenta proposta apresentou resultados satisfatórios, o que a valida como uma nova alternativa para o desenvolvimento de sistemas multiagente.*

1. Introdução

Na área de inteligência artificial, o paradigma orientado a agentes tem sido pesquisado e utilizado para minimizar a complexidade e aumentar a eficiência de softwares distribuídos. Esta prática tem-se mostrado eficiente para a construção de softwares com essas características, viabilizando um aumento no desenvolvimento de Sistemas Multiagente (SMA).

Uma das formas de se programar sistemas multiagente é utilizando a arquitetura *Beliefs, Desires e Intentions* (BDI). Essa arquitetura é uma forma de modelar o agente pensando em seu estado mental. A fundamentação filosófica para esta concepção de agentes vem dos trabalhos de [Dennett 1989] sobre sistemas intencionais e [Bratman 1987] sobre raciocínio prático. A metodologia Prometheus AEOlus surgiu como forma para modelar agentes BDI, suportando as dimensões de organização e ambiente, que também compõem o desenvolvimento de um SMA. Como propósito final, a metodologia Prometheus AEOlus promete ser aderente ao *framework* JaCaMo. Entretanto, até o presente momento, esta metodologia não possuía nenhuma ferramenta para apoiar o seu utilizador.

Desta forma, o objetivo deste trabalho é desenvolver uma ferramenta gráfica que suporte a metodologia Prometheus AEOlus, facilitando sua utilização. Além disso,

desenvolveu-se um mecanismo de geração de código automático para a linguagem *agents-peak*, parte da plataforma Jason, suprindo uma das dimensões presentes no *framework* JaCaMo.

2. Fundamentação Teórica-Tecnológica

Esta seção apresenta conceitos teóricos e tecnológicos para a compreensão dos tópicos abordados ao longo deste trabalho. Na subseção 2.1 é descrita a origem da engenharia de software orientada a agentes. A subseção 2.2 descreve em linhas gerais a metodologia Prometheus AEOlus, a qual é a base deste trabalho. Na subseção 2.3 é retratado o processo de desenvolvimento orientado a modelos. Por último, na subseção 2.4 é explicado o processo de desenvolvimento de *plug-ins* para a IDE Eclipse.

2.1. Engenharia de Software Orientada a Agentes

A (*Agent Oriented Software Engineering*) (AOSE) foi desenvolvida para suprir as necessidades encontradas no desenvolvimento de sistemas complexos. Esta subárea mescla as áreas de Inteligência Artificial e Engenharia de Software para oferecer suporte ao desenvolvimento de sistemas orientados a agentes [Guedes 2012].

Para [Gleizes and Gomez-Sanz 2009], a AOSE está ganhando relevância por duas principais razões: primeiro, as estruturas conceituais atingiram um nível de maturidade que torna razoável dedicar esforços para encontrar um consenso entre linguagens de modelagem já propostas e também suporte a ferramentas; segundo, a influência da engenharia orientada a modelos enfatiza o valor potencial de ter modelos no centro do processo de desenvolvimento.

2.2. Prometheus AEOlus

Segundo [Uez 2013], a metodologia Prometheus AEOlus foi desenvolvida baseando-se em duas tecnologias: a metodologia Prometheus e o *framework* JaCaMo. A autora complementa afirmando que o propósito da metodologia é fazer uma extensão da metodologia Prometheus de modo a incluir a especificação de Ambiente e Organização.

A metodologia possui notações gráficas definidas para cada conceito, as quais são utilizadas para representar agentes, ações, cenários, mensagens, percepções, crenças, entre outros. Esses conceitos são utilizados nos doze diagramas que a metodologia oferece para suporte ao desenvolvimento de SMA [Uez 2013].

O processo de desenvolvimento definido na metodologia é dividido em quatro fases: especificação do sistema, projeto arquitetural, projeto detalhado e implementação [Uez 2013]. A primeira fase tem como objetivo principal especificar os cenários e os objetivos do sistema. Quanto ao projeto arquitetural, nessa fase são definidos os elementos que formam o sistema. Soma-se a essas fases, o projeto detalhado, o qual tem por objetivo definir a estrutura interna dos agentes, através de suas crenças, planos e capacidades [Uez 2013]. A última fase é a implementação, na qual o objetivo é gerar código para o *framework* JaCaMo.

2.3. Model-Driven Engineering

Segundo [Rube 2013], a *Model-Driven-Engineering* (MDE) ou Engenharia Dirigida a Modelos é um novo enfoque na área de engenharia de software. MDE utiliza modelos

como artefatos de software com o objetivo de facilitar o trabalho e reduzir o tempo de desenvolvimento e o número de erros no software gerado. Modelos são um conjunto de elementos que descrevem um sistema [Mellor 2004] com grau de abstração maior que o próprio sistema [Kleppe et al. 2003]. Para [Miller et al. 2001], um modelo pode ser definido como uma especificação formal de uma função, estrutura e/ou comportamento de um sistema.

Segundo [Rube 2013], a MDE tem algumas aplicações, sendo: *Model-Driven Development* (MDD), Engenharia Reversa, *Software Process Engineering* (SPE), *Domain Specific Language* (DSL) e *Model-Driven Integration* (MDI). Para este trabalho, foi utilizada a DSL, que conforme [Van Deursen and Klint 2002], é uma linguagem específica de domínio que fornece uma notação adaptada para um domínio de aplicação e baseia-se seus conceitos e características em um domínio relevante.

2.4. Plataforma Eclipse para Desenvolvimento de *Plug-ins*

A plataforma Eclipse é baseada em *plug-ins* que são utilizados para ampliar as funcionalidades da IDE [Foundation 2014a]. Esses *plug-ins* são codificados na linguagem de programação Java e podem oferecer diversas modalidades de serviço como biblioteca de códigos, guias de documentação ou uma extensão da própria plataforma [Rivieres and Wiegand 2004].

As bibliotecas de código auxiliam os programadores com funcionalidades já implementadas e podem ser oferecidas pelo *plug-in* em formato de *Application Programming Interface* (API). A documentação auxilia os desenvolvedores nos aspectos técnicos da linguagem de programação, bibliotecas de código, demonstrando os recursos disponíveis e sua aplicações. Na extensão da plataforma, os desenvolvedores utilizam os componentes e recursos do Eclipse como interface gráfica, mecanismos de interpretação textual e de compilação para desenvolverem soluções tecnológicas para outras finalidades como por exemplo uma ferramenta de apoio a uma nova linguagem de programação.

2.5. Processo de Desenvolvimento de um Editor Gráfico utilizando o *plug-in Graphical Modelling Framework*

Segundo [Foundation 2014b], o *Graphical Modelling Framework* (GMF) é um arcabouço para desenvolvimento de editores gráficos para modelos de domínio dentro da plataforma Eclipse. Ele foi baseado em outros dois arcabouços denominados *Graphical Editing Framework* (GEF), utilizado para a criação de editores gráficos genéricos e, *Eclipse Modelling Framework* (EMF), que permite ao desenvolvedor construir metamodelos e gerar código Java referidos ao mesmo.

O processo de desenvolvimento do *plug-in* deste trabalho é mostrado na Figura 1. Esta figura demonstra o fluxo de trabalho necessário para gerar um editor gráfico utilizando o *plug-in* GMF do eclipse.

O processo de desenvolvimento consiste na concepção de seis arquivos. O primeiro chama-se Ecore e representa o *Domain Model*. É neste arquivo que é especificado o metamodelo do editor. O metamodelo serve para regulamentar todas as ações posteriores que o editor gráfico possa ter, por exemplo, quais entidades terão relacionamento e quais não terão.

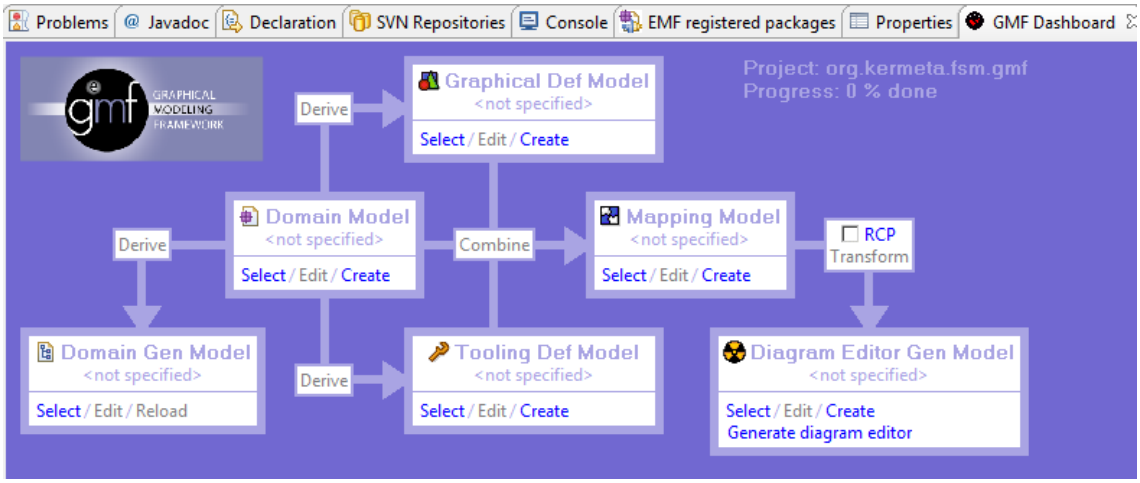

Figura 1. Dashboard do GMF Eclipse.

O próximo passo na elaboração do editor gráfico é derivar o *Domain Gen Model* do editor. Por esse motivo que existe o botão *Derive* ao lado esquerdo da caixa *Domain Model* no *dashboard*. O processo de derivação consiste em realizar uma transformação do metamodelo gerado no Ecore para um metamodelo específico com extensão **.genmodel**, para que a partir deste, sejam gerados códigos na linguagem de programação Java. Além disso, o arquivo com extensão **.genmodel**, é a base para a geração do restante de todos os artefatos do projeto de geração de um editor gráfico utilizando o GMF eclipse. Os arquivos **.ecore** e **.genmodel** correspondem ao *plug-in* EMF Eclipse, que correspondem a primeira parte da junção que resultou na concepção do GMF Eclipse.

Sobre o campo de desenho utilizado para modelar os diagramas, este é derivado do *Domain Model* e corresponde a caixa chamada *Graphical Del Model*. O arquivo gerado nesta transformação tem a extensão **.gmfgraph**. Além disso, através da plataforma eclipse pode-se editar este arquivo, adicionando pontos provenientes de figuras geométricas, ou outras customizações sobre cada entidade derivada do arquivo **.ecore**. É neste arquivo que se configura toda a caixa de desenho que será utilizada para modelar o diagrama que está sendo desenvolvimento através do editor gráfico.

Em relação a paleta de componentes que irá do lado do campo de desenho do editor gráfico desenvolvido, esta é concebida também por meio do *Domain Model*. Após a solicitação de derivação do *Domain Model* em relação a caixa chamada *Tooling Del Model* é gerado um arquivo com a extensão **.gmftool** [Gronback 2009]. Este arquivo contém a respectiva configuração de como irá ser apresentado os componentes gráficos para serem utilizados no campo de desenho do editor gráfico [Gronback 2009].

O próximo passo da elaboração da ferramenta é fazer a combinação entre todos os arquivos gerados anteriormente. Por intermédio do botão *Combine* é feito essa combinação, gerando um arquivo na caixa chamada *Mapping Model*, ilustrada na Figura 1. O arquivo gerado nessa caixa tem a extensão **.gmfmap**. Segundo [Gronback 2009], talvez o mais importante de todos os modelos em GMF é o *Mapping Model*. Nele, elementos de definição do diagrama (nós e *links*) são mapeados para o modelo de domínio e elementos de ferramentas atribuídas. O *Mapping Model* representa o diagrama de definição real e é usado para criar um modelo de gerador.

O último passo no desenvolvimento do editor gráfico é fazer a geração do arquivo que ficará responsável em fazer a junção de todos os outros componentes abordados no decorrer do desenvolvimento. O nome da caixa equivalente a esse processo é *Diagram Editor Gen Model*, ilustrada na Figura 1. A partir do arquivo **.gmfmap** e arquivo **.ecore** é gerado o arquivo **.gmfgen**. Este arquivo é responsável por armazenar as configurações de onde será gerado os códigos fontes para a execução do editor gráfico modelado no decorrer de todo esse processo. Posterior a configuração desse arquivo, inicia-se o processo de transformação, gerando o código fonte correspondente as etapas anteriores realizadas. No momento da finalização dessa transformação, tem-se os códigos fontes que poderão ser executados para a utilização do editor gráfico desenvolvido, finalizando assim o processo.

3. Ferramenta de Suporte a Metodologia Prometheus AEOlus

Esta seção apresenta a arquitetura geral do *plug-in* desenvolvido, bem como os passos realizados até a sua finalização. A subseção 3.1 descreve a arquitetura geral do *plug-in*. Na subseção 3.2 são descritos os procedimentos realizados para desenvolvimento do diagrama de Visão Geral do Agente. Por fim, na subseção 3.3, são transmitidos os passos incrementados para a geração automática de código.

3.1. Arquitetura Geral da Ferramenta

A arquitetura geral da ferramenta Prometheus AEOlus é ilustrada na Figura 2. Cada pacote representa um *plug-in* gerado para incorporar funções à ferramenta desenvolvida. O pacote chamado *gerador*, representa o *plug-in* responsável pela geração de código da ferramenta. Dentro desse *plug-in*, são agrupadas as classes responsáveis por realizarem a função designada ao *plug-in*. Esse pacote contém uma ligação de dependência com o pacote *diagram* porque ele necessita de informações fornecidas pelo mesmo.

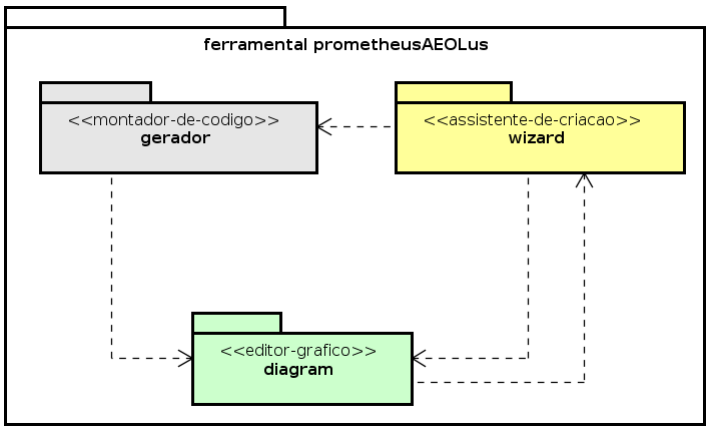

Figura 2. Arquitetura Geral do *plug-in* Prometheus AEOlus.

O pacote *diagram* comporta o *plug-in* editor gráfico. Ele contém todos os arquivos de configuração e os códigos gerados por todo o desenvolvimento proveniente do *plug-in* GMF Eclipse. Diferente dos demais *plug-ins* que compõem esta ferramenta, o editor gráfico não possui um inicializador, resultando na dependência ao *plug-in wizard* para ser inicializado.

O pacote *wizard* contém o *plug-in* responsável por fazer a junção entre os outros dois *plug-ins* que constituem a ferramenta desenvolvida. Ele reúne as funcionalidades

dos outros *plug-ins* descritos, por esse motivo possui dependência aos demais. A peculiaridade desse *plug-in* em relação aos demais consiste na inexistência de códigos-fontes, apenas pontos de extensão adicionados no arquivo *eXtensible Markup Language* (XML) de configuração do *plug-in*, onde são informados com quais *plug-ins* ele tem relação de dependência. Detalhes da implementação de um dos diagramas presentes na metodologia são descritos a seguir.

3.2. Desenvolvimento do Diagrama Visão Geral do Agente

Segundo [Uez 2014], o diagrama de visão geral do agente descreve o agente internamente, ou seja, detalhando seus planos, suas habilidades e crenças, bem como as capacidades que o agente pode ter. Os planos tem como finalidade indicar quais ações o agente deve executar para atingir um objetivo. Cada plano deve ter um evento *trigger* que iniciará a execução do plano. Eventos *triggers* podem ser desde mensagens recebidas ou até mesmo uma percepção do ambiente. Um evento é ligado ao plano através de uma seta pontilhada. Conforme [Uez 2014], os planos podem conter envio de mensagens, ações, outras percepções recebidas ou crenças. Quaisquer desses elementos são ligados ao plano através de setas simples e contínuas e são ligados entre si por setas pontilhadas que indicam a ordem em que essas ações devem ocorrer.

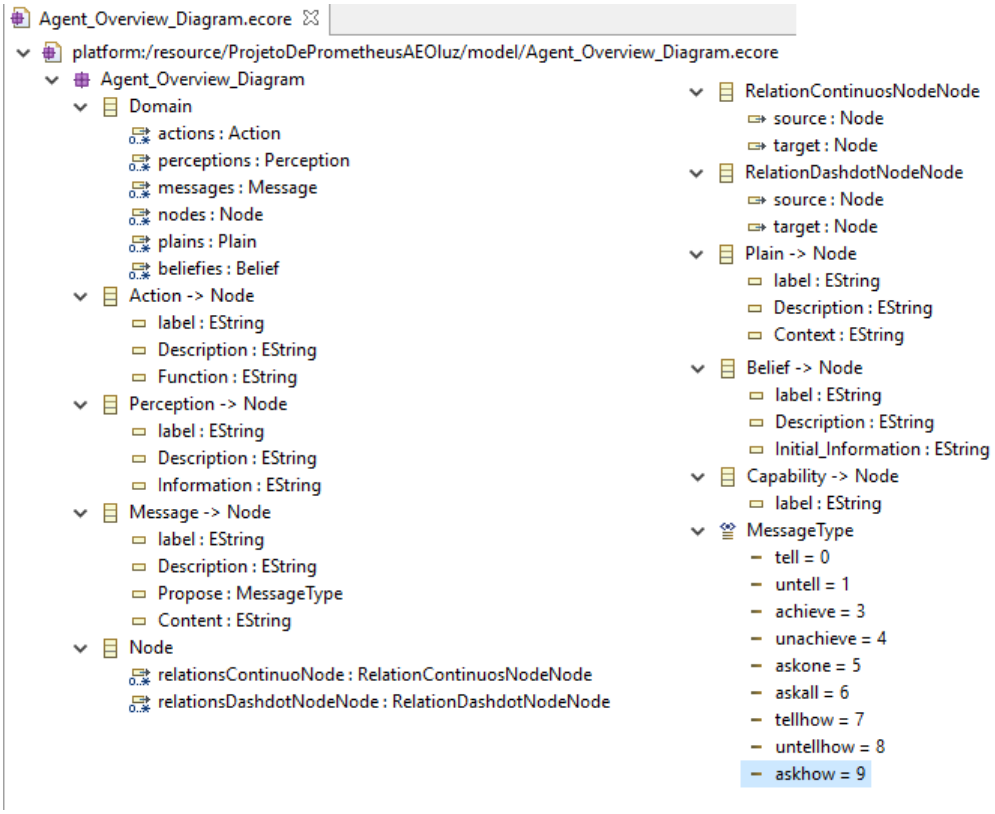

Figura 3. Metamodelo do Diagrama de Visão Geral do Agente na perspectiva do Ecore.

O metamodelo desenvolvido correspondente ao diagrama de visão geral do agente é ilustrado na Figura 3. A entidade *Node* tem como objetivo ser uma entidade genérica que isola atributos e relacionamentos comuns às demais entidades do metamodelo. Sua

diferença em relação às demais consiste no número de relacionamentos. A entidade *Node* neste metamodelo possui dois relacionamentos. O primeiro intitulado *relationsContinuoNode*, corresponde as ligações que utilizam uma seta simples, contínua. Já o outro relacionamento denominado *relationsDashdotNodeNode*, representa os relacionamentos que utilizam uma seta pontilhada entre duas entidades que herdam da entidade *Node*.

A entidade *RelationContinuosNodeNode* e *RelationDashdotNodeNode* possuem ambas os atributos *source* e *target* que representam a origem e o destino das relações. Ambos os atributos são do tipo *Node* justamente para serem reaproveitados nas entidades que herdam os relacionamentos da entidade *Node*.

A entidade *Action* representa uma ação e herda o relacionamento da entidade *Node*. Esta entidade contém os atributos *label*, *description* e *function*, ambos do tipo *EString*. A entidade *Perception* também herda o relacionamento da entidade *Node*. Ela possui os atributos *label*, *description* e *information*, ambos também do tipo *EString*. Já a entidade *Message* representa uma mensagem no metamodelo. Essa entidade também herda o relacionamento da entidade *Node*. Além disso, ela possui os atributos *label*, *description*, *propose* e *content*. Com exceção do atributo *propose*, ambos os outros são do tipo *EString*, portanto, armazenam textos. O atributo *propose* é do tipo *MessageType*, que é uma entidade do tipo *Enumeration*, ao qual representa os tipos de mensagens que podem ser enviados pelos agentes.

A entidade *Plain* representa os planos que os agentes terão. Essa entidade possui os atributos *label*, *description* e *context*, ambos do tipo *EString*. Além disso essa entidade herda os relacionamentos presentes na entidade *Node*. A entidade *Belief* representa as crenças no campo de desenho. Essa entidade possui os atributos *label*, *description* e *inicial_information*, ambos do tipo *EString*.

A entidade *Capability* não aparece na descrição deste diagrama, conforme [Uez 2013]. Entretanto, no momento de desenvolvimento desse metamodelo percebeu-se que sua única distinção em relação ao metamodelo do Diagrama de Capacidade era a entidade *Capability*, possibilitando assim que houvesse uma junção entre os metamodelos. Em seguida, modelou-se a entidade *Capability* no metamodelo deste diagrama. A entidade *Capability* possui somente o atributo *label* cujo o tipo é *EString*.

3.3. Desenvolvimento do *plug-in* gerador de código

O GMF Eclipse utiliza um arquivo derivado do XML, intitulado *XML Metadata Interchange* (XMI), para fazer sua persistência de dados, conforme Figura 4. Este arquivo contém diversas *tags* que tornam o processo de extração de informações mais acessível. No processo de geração de código para a plataforma Jason foi utilizado os arquivos XMI gerados pelo GMF. Estes arquivos continham informações referentes à modelagem de diagramas do gênero Modelo Geral da metodologia Prometheus AEOlus, ou seja, o diagrama que reúne todas as entidades e relacionamentos presentes na metodologia. Por questões de implementação, optou-se por desenvolver um diagrama geral que possibilitasse a modelagem de todas as entidades da metodologia, de modo que o processo de geração de código fosse mais acessível para este primeiro protótipo.

O *plug-in* gerador de código é dividido em 4 camadas, sendo: *modelo*, *gerador*, *visao* e *servico*. A primeira camada é responsável por representar através de objetos as entidades utilizadas pelos diagramas da metodologia Prometheus AEOlus. Além disso,

```
<?xml version="1.0" encoding="UTF-8"?>
<PrometheusAEOLus_generalModel:Domain xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:PrometheusAEOLus_generalModel="PrometheusAEOLus_generalModel">
  <nodes xsi:type="PrometheusAEOLus_generalModel:Goal"/>
  <nodes xsi:type="PrometheusAEOLus_generalModel:Role"/>
  <nodes xsi:type="PrometheusAEOLus_generalModel:Action"/>
  <nodes xsi:type="PrometheusAEOLus_generalModel:Artifact"/>
  <nodes xsi:type="PrometheusAEOLus_generalModel:Perception"/>
  <nodes xsi:type="PrometheusAEOLus_generalModel:Protocol"/>
  <nodes xsi:type="PrometheusAEOLus_generalModel:Message"/>
  <nodes xsi:type="PrometheusAEOLus_generalModel:Belief"/>
  <nodes xsi:type="PrometheusAEOLus_generalModel:Plain"/>
  <nodes xsi:type="PrometheusAEOLus_generalModel:Scenario"/>
  <nodes xsi:type="PrometheusAEOLus_generalModel:Capability"/>
  <nodes xsi:type="PrometheusAEOLus_generalModel:Agent"/>
  <missions/>
  <groups/>
  <workspaces/>
  <edgesSimple source="//@nodes.11" target="//@nodes.1"/>
</PrometheusAEOLus_generalModel:Domain>
```

Figura 4. Exemplo do Diagrama de Modelo Geral na perspectiva do *plug-in* desenvolvido.

essa camada também possui classes que ajudam o *plug-in* a encontrar informações presentes no XMI. A camada *gerador* engloba as classes que centralizam as tarefas do *plug-in* com o ambiente Eclipse e as classes responsáveis por escrever a *string* que conterá os códigos que devem ser persistidos no arquivo que o *plug-in* gerará posteriormente. A camada *servico* é responsável por ler o arquivo XMI gerado pelo *plug-in* editor gráfico, transformando-o em objetos, de modo que no final do processo seja retornado uma estrutura de dados do tipo *hashmap* com os devidos objetos transformados. A camada *visao* tem como propósito gerenciar a utilização dos serviços e a geração de código pelo *plug-in* desenvolvido. Por intermédio de métodos provenientes das classes dessa camada, é possível ativar as funcionalidades do *plug-in*. A ativação ocorre através de eventos que são disparados pelos usuários que o utilizam.

4. Estudo de Caso - *Build a House*

Em [Boissier et al. 2013], foi apresentado o estudo de caso chamado *Build a House*. O estudo de caso em questão trata-se de um personagem chamado *Giacomo* que deseja construir sua casa. Para isso, ele precisa contratar empresas que possam executar as tarefas relacionadas com a construção e coordenar o trabalho dessas empresas. A contratação será feita por meio de uma licitação, na qual a empresa que apresentar o menor preço para a tarefa será contratada.

Para comprovar a aplicabilidade do *plug-in*, optou-se nesta seção por demonstrar a modelagem de um agente deste estudo de caso, utilizando a ferramenta gráfica desenvolvida. Complementarmente, mostra-se os respectivos códigos gerados para o agente modelado, evidenciando o processo de geração de código automatizado.

A Figura 5 representa o diagrama do modelo Geral do Agente *Giacomo* modelado para o estudo de caso em questão. Além deste diagrama, o estudo de caso conta com outros diagramas, que foram suprimidos deste artigo por questão de espaço. Este dia-

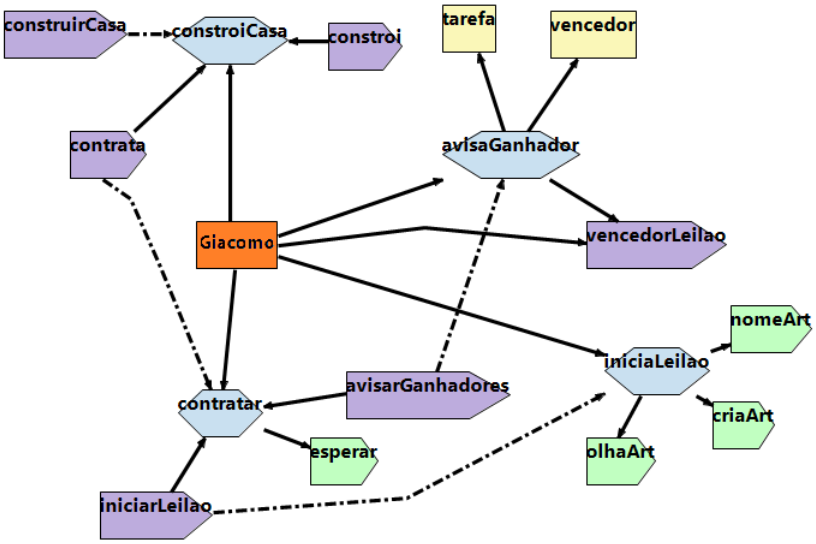

Figura 5. Diagrama de Modelo Geral do Agente Giacomo - Estudo de Caso *Build a House*

grama é oriundo do Diagrama de Visão Geral do sistema, onde os agentes são modelados de forma simplificada, sendo-os estendidos e enriquecidos através dos diagramas de visão geral do Agente. Para facilitar a geração de códigos, optou-se por criar um diagrama extra para a metodologia, intitulado Diagrama de Modelo Geral, onde possibilita-se que sejam modeladas todas as informações necessárias para realizar a geração de códigos para a ferramenta Jason. Este diagrama reúne os conceitos presentes no Diagrama de Visão Geral do Sistema, Diagrama de Visão Geral do Agente e Diagrama de Capacidade.

Na Figura 5, a entidade intitulada *Giacomo* representa o agente modelado. As entidades *constroiCasa*, *avisaGanhador*, *contratar* e *iniciaLeilao* representam os planos que compõem o agente. As entidades *construirCasa*, *constroi*, *contrata*, *vencedorLeilao*, *avisaGanhadores* e *iniciarLeilao* representam as mensagens enviadas/recebidas por esses planos ou as percepções obtidas por eles. As entidades *esperar*, *olhaArt*, *criaArt* e *nomeArt* representam as ações a serem desempenhadas na execução do plano. E por fim, as entidades *tarefa* e *vencedor* representam as crenças ligadas aos respectivos planos. Ressalta-se que este diagrama é um exemplo de uso do Diagrama de Modelo Geral, apresentando diferenças em relação aos demonstrados em [Uez 2013], sendo acrescentado a entidade que representa o agente, bem como as suas ligações com planos e mensagens. Essa alteração fez-se necessária em virtude do gerador de códigos precisar de informações precisas encontradas na concatenação de diversos diagramas, tornando o processo de geração de códigos complexo.

A Figura 6 exemplifica a geração de código ocorrida para o agente *Giacomo*. Esta geração é proveniente da modelagem do diagrama ilustrado na Figura 5. A sintaxe do Jason recomenda que primeiramente sejam informados os objetivos do agente, posteriormente as crenças que esse agente possui em relação ao ambiente e por fim, os planos que o compõem. Além disso, toda a parametrização das propriedades de cada um dos elementos foi baseado no estudo de caso de [Uez 2013].

Na Figura 6, a linha 2 refere-se a uma mensagem disponibilizada ao utilizador

```
1
2  //insert here the goals to boot your agent
3
4  @constroiCasa
5  +!construirCasa :true
6  <-!contrata;
7  !constroi.
8
9  @contratar
10 +!contrata :true
11 <-!iniciarLeilao(x);
12 !avisarGanhadores;
13 .wait(500).
14
15 @avisaGanhador
16 +!avisarGanhadores :true
17 <-.broadcast(tell, vendedor(Tarefa, Vencedor));
18 ?tarefa;
19 ?vencedor.
20
21 @iniciaLeilao
22 +!iniciarLeilao(x) :true
23 <-.concat("leilao_",Tarefa,NomeArtefato);
24 focus(id);
25 makeArtifact(NomeArtefato, "leilao", [Tarefa,ValorMaximo],Id).
26
```

Figura 6. Código gerado para o Agente Giacomo - Estudo de Caso *Build a House*

do *plug-in*, para que o mesmo possa informar os objetivos individuais de cada um dos agentes gerados. Posteriormente, são informados os planos dos agentes, conforme já falado anteriormente. Cabe lembrar que na linguagem *AgentSpeak*, a sintaxe de um plano ocorre da seguinte forma:

Trigger : context <- body

Portanto, as linhas 4, 9, 15 e 21 da Figura 6 representam a notação para representar a inicialização de um plano. A linha 5 representa a *Trigger* de um plano. A *Trigger* é um evento que dispara a execução de um plano. Conforme [Uez 2013], somente duas coisas podem representar uma *trigger* de um plano: entidades do tipo *Mensagem* ou *Percepção* ligadas ao plano através de setas pontilhadas. O *True* posterior aos ":"indica o contexto do plano, neste caso estipulado por parâmetros pelo diagramador. As linhas 6 e 7 representam respectivamente a inclusão de novos objetivos.

Ainda na Figura 6, as linhas 10, 11, 12 e 13 representam o plano *contratar*. Conforme é possível visualizar, após o <- são informadas todas as entidades que compõem a *body* de um plano. As linhas 15, 16, 17 e 18 representam o plano *avisaGanhador*. O comando *broadcast* refere-se a uma mensagem enviada em massa para todos os agentes que compõem o sistema. Além disso, através da Figura 5, nota-se que as ligações entre as entidades mensagens e planos representam as mensagens que devem ser trocadas pelo sistema, entretanto, é necessário que a mensagem também tenha a ligação com o agente remetente e os destinatários, quando houver o segundo caso. As linhas 18 e 19 representam consultas as crenças do plano e são originadas das ligações entre planos e crenças. Por último, as linhas 22, 23, 24 e 25 representam o plano *iniciaLeilao*. A linha 23 em diante ilustra uma concatenação de ações que compõem o plano em questão, findando assim a geração de código para o agente *Giacomo*.

5. Conclusões

Este trabalho propõe a criação de um *plug-in* de modelagem gráfica que tem como objetivo principal apoiar a metodologia Prometheus AEOlus. Esta metodologia foi desenvolvida por [Uez 2013] e tem como finalidade permitir a modelagem integrada das três dimensões que envolvem um SMA. Esta integração visa a interligação dos conceitos modelados nos *work products* com o *framework* de desenvolvimento de SMA chamado JaCaMo. Para colaborar com o processo de integração com o *framework* em questão, desenvolveu-se agregado ao *plug-in* gráfico, um mecanismo de varredura de informações que propicia a geração automática de código fonte para a linguagem *agentspeak*, associando o *plug-in* ao ambiente de desenvolvimento de agentes chamado Jason, o qual faz parte do *framework* JaCaMo. As dimensões do ambiente (CartAgO) e da organização (Moise+), as quais compõem o restante do *framework* JaCaMo, não foram tratadas neste trabalho.

Para avaliar o *plug-in* desenvolvido, modelou-se o estudo de caso *Build a House*. Demonstrou-se o resultado da modelagem do diagrama na ferramenta e posteriormente a sua geração de código para a plataforma de desenvolvimento Jason. Ao desenvolver o *plug-in* foi possível constatar que a metodologia possui uma série de diagramas, tornando seu uso bastante custoso. Essa diversidade é consequência de diversos deles possuírem entidades bastante semelhantes, ocorrendo uma interdependência e repetição de informações entre os diagramas desenvolvidos no decorrer do uso da metodologia.

Em suma, o desenvolvimento deste trabalho contribuiu para a potencialização da metodologia desenvolvida por [Uez 2013]. A utilização da plataforma GMF Eclipse foi uma escolha positiva para o desenvolvimento do *plug-in* gráfico, visto que apresenta uma boa estrutura de uso e possibilita que outras pessoas façam a manutenção da parte gráfica do *plug-in* posteriormente.

Outro ponto positivo no desenvolvimento deste trabalho foi a escolha de sua arquitetura. A mesma separa as responsabilidades da solução em diferentes *plug-ins*, facilitando sua manutenção e evitando a incumbência exagerada sobre as camadas do *plug-in*. Um fator a aprimorar é *plug-in* gerador de códigos, visto que o mesmo foi codificado na linguagem Java e apresenta uma solução engessada para o domínio específico da metodologia na qual se deseja apoiar. Em um trabalho futuro, pode-se desenvolver uma nova forma de geração de códigos, onde seja possível o uso de MDE, tornando o *plug-in* desenvolvido totalmente automatizado e orientado a modelos, contribuindo inclusive para a sua extensão para outras plataformas de desenvolvimento.

6. Agradecimentos

Os autores agradecem à Universidade Federal do Rio Grande - FURG e a Fundação de Amparo à Pesquisa do Estado do Rio Grande do Sul - FAPERGS pelo suporte financeiro na realização do presente trabalho.

Referências

Boissier, O., Bordini, R. H., Hübner, J. F., Ricci, A., and Santi, A. (2013). Multi-agent oriented programming with jacamo. *Science of Computer Programming*, 78(6):747–761.

Bratman, M. (1987). Intention, plans, and practical reason. Harvard University Press, Cambridge, MA.

Dennett, D. C. (1989). *The intentional stance*. MIT press.

Foundation, E. (2014a). Eclipse documentation - current release. Disponível em: http://help.eclipse.org/luna/index.jsp. Acesso em 19 de janeiro de 2016.

Foundation, E. (2014b). Graphical modeling framework. Disponível em: http:/wiki.eclipse.org/Graphical_Modeling_Framework. Acesso em 19 de janeiro de 2016.

Gleizes, M.-P. and Gomez-Sanz, J. J. (2009). *Agent-Oriented Software Engineering X: 10th International Workshop, AOSE 2009, Budapest, Hungary, May 11-12, 2009, Revised Selected Papers*, volume 6038. Springer.

Gronback, R. C. (2009). *Eclipse modeling project: a domain-specific language (DSL) toolkit*. Pearson Education.

Guedes, G. T. A. (2012). *Um metamodelo UML para a modelagem de requisitos em projetos de sistemas multiagentes*. PhD thesis, Universidade Federal do Rio Grande do Sul.

Kleppe, A. G., Warmer, J. B., and Bast, W. (2003). *MDA explained: the model driven architecture: practice and promise*. Addison-Wesley Professional.

Mellor, S. J. (2004). *MDA distilled: principles of model-driven architecture*. Addison-Wesley Professional.

Miller, J., Mukerji, J., et al. (2001). Model driven architecture (mda). *Object Management Group, Draft Specification ormsc/2001-07-01*.

Rivieres, J. and Wiegand, J. (2004). Eclipse: A platform for integrating development tools. *IBM Systems Journal*, 43(2):371–383.

Rube, R. I. (2013). Introducción a la ingeneriería del software dirigida por modelos. Disponível em: https://ocw.uca.es/pluginfile.php/2487/mod_resource/content/0/T1%20-%20MDE.pdf. Acesso em 20 de janeiro de 2016.

Uez, D. M. (2013). Método para o desenvolvimento de software orientado a agentes considerando o ambiente e a organização. Master's thesis, Universidade Federal de Santa Catarina.

Uez, D. M. (2014). Descrição do método prometheus aeolus. Disponível em: http://www.uez.com.br/aeolus/docs/aeolus_11112014.pdf. Acesso em 23 de setembro de 2015.

Van Deursen, A. and Klint, P. (2002). Domain-specific language design requires feature descriptions. *CIT. Journal of computing and information technology*, 10(1):1–17.