JAMDER: JADE para multiagente Recurso de Desenvolvimento de Sistemas

Abstract A diferença semântica é diferenciada pela diferença entre duas descrições geradas usando diferentes representações. Essa diferença tem um impacto negativo na produtividade do desenvolvedor e provavelmente na qualidade do código escrito. No contexto de desenvolvimento de software, a fase de codificação visa codificar o sistema de acordo com o projeto detalhado desenvolvido com um grupo de modelos projetados. Este artigo apresenta um esforço para consolidar diferentes definições de tipos de agentes e conceitos de implementação para Sistemas Multiagente (MAS), envolvendo a adaptação da estrutura JADE em relação aos conceitos teóricos no MAS. Além disso, ele contém uma padronização da geração de código. O principal benefício da extensão proposta é incluir as arquiteturas, entidades e relacionamentos internos do agente em uma estrutura de implementação e aumentar a produtividade por geração de código, garantindo a consistência entre design e código. A aplicabilidade da extensão é ilustrada pelo desenvolvimento de um sistema multiagente para o Moodle.

1. Introdução

Um Sistema Multiagente (MAS) envolve uma rica variedade de entidades, como organizações, ambientes, funções de agente e objeto, nas quais cada uma possui relacionamentos e comportamentos associados (Freire et al., 2013). Em particular, um único MAS pode ser composto por agentes com diferentes arquiteturas, em que cada arquitetura envolve alguns comportamentos e atributos específicos associados ao agente. Nesse contexto, a alta complexidade para desenvolver o MAS requer um conjunto de métodos e ferramentas para auxiliar na construção desse sistema, considerando a particularidade de cada entidade. Russell e Norvig (2002), definiram inicialmente quatro arquiteturas para agentes: agentes reflexos simples, agentes reflexivos baseados em modelo, agentes baseados em objetivos e agentes baseados em utilidade. O BDI (intenção de crença-desejo) é outra arquitetura conhecida que envolve o conceito de um agente para armazenar as etapas (crenças), de acordo com seus objetivos (desejos) e fazer planos (intenções).

Por outro lado, o paradigma de desenvolvimento orientado a agentes requer técnicas adequadas para explorar seus benefícios e recursos para apoiar a construção e manutenção desse tipo de software. Como é o caso de qualquer novo paradigma de engenharia de software, a implantação bem-sucedida e generalizada de MASs requer linguagens de modelagem que exploram o uso de abstrações relacionadas a agentes e promovem a rastreabilidade dos modelos de design ao código. Para reduzir o risco ao adotar uma nova tecnologia, é conveniente

apresentá-la como uma extensão incremental de métodos conhecidos e confiáveis e fornecer ferramentas explícitas de engenharia que suportam métodos aceitos pelo setor de implantação de tecnologia (Castro et al., 2006). A transição entre os estágios de modelagem e implementação geralmente é realizada de maneira não sistemática e, freqüentemente, os aspectos considerados na modelagem não têm contrapartida na implementação. Nesse contexto, a existência de um metamodelo conceitual é crucial, pois representa a ontologia das entidades em um MAS. A estrutura TAO + (domar agentes e objetos) fornece uma ontologia que abrange os fundamentos da engenharia de software com base em agentes e objetos e suporta o desenvolvimento do MAS em larga escala (Freire, 2013). O TAO + apresenta a definição de cada abstração, como um conceito de sua ontologia, e estabelece as relações entre elas.

Por outro lado, o uso de métodos e ferramentas para apoiar as atividades de desenvolvimento permite um aumento na produtividade e, em geral, garante a correção dos artefatos gerados (código produzido). Mais especificamente, estruturas e plataformas fornecem um ambiente para implementação. Mal, uma estrutura cumpre todas as características de diferentes tipos de agentes e outras entidades para aumentar a complexidade dos agentes em um ambiente.

A estrutura Java Agent Development 1 (JADE), amplamente utilizada no desenvolvimento do MAS, possui as seguintes características: (i) possui uma plataforma na linguagem Java, (ii) suporta sistemas distribuídos e (iii) é gratuita. No entanto, a implementação do agente no JADE é limitada principalmente a agentes, comportamentos e mensagens. O JADE também possui uma plataforma, que contém o ambiente necessário para o ciclo de vida do agente e contêineres em que os agentes residem.

Este artigo apresenta uma extensão da estrutura JADE para fornecer a infraestrutura adequada focada na implementação de agentes de acordo com as configurações abordadas em (Freire, 2013). Além disso, é apresentada uma abordagem baseada na arquitetura orientada a modelos (MDA 2) (Mellor, 2004) para dar suporte à geração de código, a fim de promover um desenvolvimento rápido e consistente para diferentes tipos de arquitetura de agentes. O artigo está organizado da seguinte forma: na Seção 2, são apresentados trabalhos relacionados. A Seção 3 apresenta o referencial teórico sobre as entidades no metamodelo TAO +, na estrutura JADE e na abordagem MDA. A seção 4 apresenta a extensão do JADE e o mecanismo de geração de código. Seção 5, um estudo de caso é ilustrado. Finalmente, a Seção 6 apresenta as conclusões e sugestões para trabalhos futuros.

2. Trabalhos Relacionados

Um conjunto de estruturas e linguagens de modelagem foi desenvolvido para apoiar a implementação do MAS. Em geral, essas ferramentas são associadas a uma linguagem de programação orientada a objetos, para compor entidades e fornecer um ambiente para sua execução.

2.1. Linguagens e ferramentas de modelagem do MAS

Jadex (JADE XML) (Braubach, 2003) (Pokahr et al., 2005) é um mecanismo de raciocínio orientado a agentes que os agentes são escritos na linguagem de programação XML e Java. Um dos principais aspectos do Jadex é não apresentar uma nova linguagem de programação. Ele usa o ambiente orientado a objetos já existente. Além disso, a utilização da linguagem XML na definição dos recursos do agente aumenta a complexidade do desenvolvimento. Por outro lado, uma atualização de plataforma implica que o código criado usando a versão anterior pode ser incompatível com a nova.

JaCaMo (Boissier et al., 2011) é uma estrutura para a Programação Multi-Agente construída em três plataformas existentes, Jason para programar agentes autônomos, Moise para organizações de programação e CArtAgO para programar ambientes compartilhados. Um sistema de software programado no JaCaMo é definido pela organização de agentes autônomos de BDI com base em conceitos como papéis, grupos, missão e esquemas; agentes autônomos são implementados em Jason; trabalhando em ambientes compartilhados baseados em artefatos distribuídos. O metamodelo JaCaMo define dependências, conexões e mapeamentos conceituais e sinergias entre todas as diferentes abstrações disponíveis nos metamodelos associados a cada nível de abstração.

Opal (Purvis et al., 2002) é uma plataforma para o desenvolvimento da arquitetura abstrata baseada em MAS, desenvolvida pelo FIPA 3. Ele fornece uma estrutura formada por oficiais especialistas que fornecem serviços essenciais para o desenvolvimento do MAS, com, por exemplo, registro e busca de agentes instanciados no sistema. Um agente é feito de uma combinação de vários microagentes. Esses microagentes são níveis mais baixos em um MAS.

O JADE (Castro et al., 2006) é amplamente utilizado em vários setores do mercado, como telecomunicações, o JADE fornece uma infraestrutura robusta e madura e oferece muitos recursos necessários para a implementação de sistemas multiagentes, que incluem páginas amarelas, troca de mensagens e suporte a ontologias. O JADE implementa um modelo orientado a tarefas, no qual os agentes têm um conjunto de comportamentos. Nenhuma habilidade cognitiva, como um ciclo de raciocínio, é fornecida para os agentes.

Santos (2006) fornece um metamodelo que representa os conceitos que definem um sistema de agentes, bem como os relacionamentos entre eles. A geração de código usa uma ferramenta de protótipo desenvolvida no Velocity 4 e as informações de

modelagem são representadas pelo arquivo XML (Extensible Markup Language) que contém as estruturas do agente. O autor utiliza um protótipo de uma ferramenta de modelagem, denominada MAS Modeler (Santos, 2006), onde as informações textuais são preenchidas através de telas passo a passo. Depois disso, essa estrutura é armazenada em um arquivo XML e pode ser usada pelo modelo Velocity para geração de código na estrutura Semanticore (Blois e Lucena, 2004), mas apenas para agentes.

O TAOM4E (Morandini et at., 2011) é um ambiente de modelagem orientado a agentes e suporta um desenvolvimento de software orientado a modelos e orientado a agentes. Foi desenvolvido levando em consideração as recomendações do MDA. A arquitetura TAOM4E permite uma integração flexível de diferentes ferramentas. A ferramenta é um plug-in para ECLIPSE Platform, mas permite a geração de código apenas para agentes BDI.

A Prometheus Design Tool (PDT) (Padgham e Winikoff, 2004) é uma ferramenta gráfica usada para projetar um MAS seguindo a Metodologia Prometheus. O PDT é integrado à plataforma Eclipse, permitindo que os usuários realizem o ciclo de vida completo do desenvolvimento de um aplicativo orientado a agentes em um IDE. De maneira semelhante ao TAOM4E, a geração de código é direcionada apenas aos agentes BDI.

Entre todas as estruturas destacadas nesta seção, apenas JADEX e JADE têm um IDE (Integrated Development Environment) com suporte gratuito para depuração de código. Considerando que qualquer um deles precisaria da extensão para melhor se adequar à linguagem de modelagem, o JADE mostrou o mais adequado, pois não é necessária a utilização de outro idioma para redefinir a estrutura do agente como JADEX (Nunes, 2008).

2.2. Abordagens MDD para o desenvolvimento do MAS

Várias abordagens e estruturas metodológicas para o projeto orientado a modelos de sistemas multiagentes já foram propostas (Gómez-Sanz et al., 2010) (Ficher et al., 2012). Nesse contexto, as técnicas de transformação de modelo são um dos aspectos principais da abordagem de desenvolvimento orientada a modelo. Em (Gascuena et al., 2014), são apresentadas transformações de modelo em modelo e modelo em texto para automatizar o processo de desenvolvimento para gerar código ICARO a partir do modelo INGENIAS.

Estudos sobre linguagens específicas de domínio (DSLs) e linguagens de modelagem específicas de domínio (DSMLs) para agentes surgiram no contexto do desenvolvimento do MAS

(Challenger, 2014). Um DSML para MAS é apresentado em (Gascuena et al., 2012), onde o resumo e a sintaxe concreta foram apresentados usando a Meta-object Facility (MOF) e GMF, respectivamente. Por fim, a geração de código para a plataforma do agente JACK 5. No entanto, a linguagem de modelagem desenvolvida é baseada no metamodelo (Challenger et al., 2014) de uma das metodologias específicas do MAS, denominadas Prometheus (Padgham e Winikoff, 2004). Estudo semelhante foi realizado em Fuentes-Fernandez et al. (2010), que se baseia na metodologia INGENIAS (Pavon et al., 2005) para o desenvolvimento do MAS.

Mais especificamente, o DSM e o DSML podem fornecer a abstração necessária e oferecer suporte a uma metodologia mais frutífera para o desenvolvimento do desenvolvimento de MASs habilitados para a Web Semântica. Nesse domínio, agentes autônomos podem avaliar dados semânticos e colaborar com entidades semanticamente definidas da Web Semântica, como os Serviços Web Semânticos. Challenger et al. (2016) apresenta uma nova metodologia DSL-base para o desenvolvimento do MAS com web semântica. Este estudo apresentou uma DSL chamada Semantic Web-Enabled Agent Language (SEA_L) (Demirkol, 2013) (Getir et al., 2014) e inclui geração de código e restrições para verificar os programas em relação à implementação de agentes SEA_L.

3. Referencial Teórico

Esta seção descreve os conceitos relacionados ao metamodelo TAO +, incluindo as entidades necessárias para as definições de MAS e Jade.

3.1. Entidades MAS em agentes e objetos domesticados (TAO +)

A estrutura TAO + fornece uma ontologia que abrange os fundamentos da Engenharia de Software com base em agentes e objetos e suporta o desenvolvimento do MAS em larga escala (Freire, 2013). O TAO + apresenta a definição de cada abstração, como um conceito de sua ontologia, e estabelece as relações entre elas (Figura 1). Gonçalves et al. (Gonçalves et al., 2010) (Gonçalves et al., 2015) descrevem as entidades necessárias para um MAS, como as arquiteturas internas dos agentes, como a seguir:

- Objeto: É um elemento passivo que possui estado e comportamento e pode estar relacionado a outros elementos.
- Agente: É um elemento autônomo, adaptável e interativo. Sua característica comportamental básica é a ação e os aspectos estruturais / mentais e comportamentais dependem de sua arquitetura interna:

- O Agente reflexo simples não possui características estruturais / mentais e tem como característica comportamental a Percepção e Ação (orientadas pelas Regras de Condição-Ação);
    - O Agente reflexo baseado no Modelo tem a característica estrutural / mental Crença e tem a Percepção, a Próxima Função e a Ação (orientadas pelas Regras de Condição-Ação) como características comportamentais;
    - o O agente do MAS-ML tem objetivo e crença como características estruturais / mentais e possui Plano e Ação (orientado pelo Plano escolhido de acordo com o objetivo);
    - o O Agente Baseado em Objetivos tem objetivo e crença como características estruturais / mentais e tem Percepção, Próxima Função, Função de Formulação de Objetivos, Função de Formulação de Problemas, Planejamento e Ação como características comportamentais;
    - o O Agente de Utilidade Pública possui objetivos e crenças como características estruturais / mentais e possui Percepção, Próxima Função, Função de Formulação de Objetivos, Função de Formulação de Problemas, Planejamento e Ação de Funções Utilitárias como características comportamentais.
- Organização: É um elemento que agrupa agentes, que desempenham papéis e têm objetivos comuns. Pode restringir o comportamento de seus agentes e suborganizações através do conceito de axioma, que define as ações que devem ser executadas.
- Função do objeto: é um elemento que guia e restringe o comportamento de um objeto na organização. Uma função de objeto pode adicionar informações, comportamento e relacionamentos ao objeto que desempenha a função.
- Função do agente: é um elemento que guia e restringe o comportamento de um agente na organização. Uma função de agente define (i) tarefas que definem uma ação que deve ser executada por um agente, (ii) direitos que definem uma ação que pode ser executada por um agente e (iii) protocolo que define uma interação com os outros elementos.

- Ambiente: é um elemento que representa o habitat de agentes, objetos e organizações. Um ambiente pode ser heterogêneo, dinâmico, aberto e distribuído.

Silva et al. (2007) definem os seguintes relacionamentos no TAO +: Habitação, Propriedade, Brincadeira, Especialização / Herança, Controle, Dependência, Associação e Agregação / Composição. Os relacionamentos não serão descritos aqui, pois a estrutura e a geração de código propostas neste trabalho se aproximam implicitamente do diagrama projetado e todas as validações desses relacionamentos também agora são cobertas pela ferramenta de modelagem.

3.2. Definições de Agente JADE

A estrutura JADE inclui classes e serviços para facilitar a fase de codificação do MAS. Dentre os recursos disponíveis, alguns estão listados abaixo: serviços de publicação em páginas amarelas, serviço de localização em páginas brancas, suporte a ontologias e comunicação de protocolos compatíveis com o padrão FIPA 6 (Foundations of Intelligent Physical Agents). De fato, o JADE é uma estrutura orientada a objetos escrita em Java.

A arquitetura no JADE contém contêineres onde os agentes residem e o sistema pode ser distribuído em diferentes plataformas. Cada agente é registrado no serviço AMS (Agent Management System) fornecido pela JADE que garante a unicidade dos agentes. Para encontrar outros agentes, outro serviço é fornecido, o DF (Directory Facilitator) e funciona como um serviço de Páginas Amarelas.

As classes de agente JADE herdam (in) diretamente de jade.core.Agent (Bellifemine et al., 2007), que representa uma classe básica comum para a definição de agente pelo usuário. Portanto, do ponto de vista do programador, um agente JADE é simplesmente uma instância da classe Java que herda da classe Agent (Castro et al. 2006). Isso implica, no caso de herança de recursos, para suportar as interações básicas com a plataforma do agente (registro, configuração e gerenciamento remotos). Inicialmente, um conjunto básico de métodos deve ser codificado para executar os comportamentos do agente básico, como métodos para enviar ou receber mensagens, relacionados à utilização de protocolos de comunicação padrão, registro de vários domínios, entre outros.

O ciclo de vida do JADE para um agente é definido de acordo com o FIPA. A primeira etapa é a execução do construtor do agente, seguida pela atribuição de um identificador para o agente a ser inserido no sistema. O método setup () é executado a partir do

momento em que o agente inicia suas atividades. Um comportamento do agente é projetado pela substituição da configuração ().

Os tipos de comportamento no JADE representam as ações dos agentes (Castro et al. 2006). A principal classe relacionada ao comportamento é o jade.core.Behavior, cujas subclasses implementam finalidades específicas, como composição do comportamento ou duração da tarefa. Esta classe define dois métodos básicos: action () e done (). O método action () contém o código do comportamento a ser executado pelo agente. Após sua execução, o método done () é executado automaticamente para verificar se o comportamento foi finalizado ou não. A classe deve manter o estado da execução, para que o método done () retorne um valor lógico false (equivalente ao literal literal em Java) pelo tempo necessário para executar o método action (). Caso contrário, ele deve retornar um valor lógico true (equivalente ao literal literal em Java).

3.3. Arquitetura orientada a modelo

A arquitetura orientada a modelo (MDA) é apresentada como uma abordagem apropriada para auxiliar na geração de código, porque o código pode ser gerado várias vezes sem comprometer o modelo. O OMG 7 define alguma padronização nesse processo que não está necessariamente associada a uma plataforma específica. Assim, os conceitos podem ser aplicados a diferentes linguagens de modelagem e implementação.

O processo de transformação ocorre através das etapas propostas pela OMG. Os artefatos gerados em cada uma dessas etapas são independentes de uma ferramenta específica a ser usada, por exemplo, quando o mesmo aplicativo pode ser implementado em idiomas diferentes. Os conceitos de cada uma dessas etapas são descritos a seguir (Beydeda et al., 2005):

- CIM (Modelo Independente de Computação): regras para requisitos funcionais;
- PIM (Platform-Independent Model): relações entre propriedades e seus relacionamentos com entidades.
- PSM (Modelo Específico da Plataforma): definição de como o sistema funcionará em uma plataforma específica.
- PDM (Platform Description Model): definição de como o PIM to PSM funcionará.

Transformação de modelo é o processo de conversão de um modelo para outro modelo do mesmo sistema e representa o ponto central no desenvolvimento orientado a modelo. Modelos de alto nível são transformados em modelos de baixo nível. Nesse sentido, a idéia

de gerar um modelo a partir de outro de maneira automática / semiautomática pretende fornecer uma maneira simples e rápida de desenvolvimento de software.

4. Extensão e mapeamento de JADE para MAS

Conforme explicado na seção anterior, além do conceito de agente, a estrutura conceitual TAO + oferece outros aspectos que podem estar envolvidos em um MAS, como objeto, organização, função de objeto e função de agente. Cada entidade especifica diferentes propriedades e comportamentos estruturais.

Considerando os recursos e recursos oferecidos no nível conceitual, são necessários ajustes no JADE para transformar as propriedades e os comportamentos nos componentes de implementação correspondentes. A extensão JADE resultante é chamada JAMDER 8 (JADE para MAS Development Resource).

4.1. Agentes

Os agentes têm sua definição de arquitetura e interagem com um ambiente. Além da definição do BDI, existem os agentes reativos e proativos definidos por Russell e Norvig. Com esse conhecimento, cinco arquiteturas ou tipos diferentes de agentes podem ser identificados: (i) agentes reflexos simples, (ii) agentes reflexos baseados em modelo, (iii) agentes baseados em objetivos com pesquisa (planejamento), (iv) baseados em utilidade agentes; e (v) agentes orientados a objetivos com planos. Nesta suposição, a classe JADE Agent deve ser estendida para incorporar esses cinco tipos de agentes (Figura 2).

Exceto para agentes baseados em objetivos com planos, as outras arquiteturas têm características semelhantes de acordo com a percepção e as funções usadas, mas os agentes baseados em objetivos com agentes de planejamento e baseados em utilidade não compartilham as regras de ação e condição. Portanto, três hierarquias de agentes no nível de modelagem foram estabelecidas: (i) agente baseado em objetivos com plano, (ii) agentes reflexos (agente reflexo simples e agente reflexo baseado em modelo) e (iii) agentes cognitivos (agente baseado em objetivos com agente de planejamento e utilidade). Essa classificação é baseada nas características estruturais e comportamentais dos agentes.

A classe jamder.agents.GenericAgent, que estende jade.core.Agent, foi definida para representar essas propriedades e outros atributos comuns entre as três ramificações. Os atributos comuns são: (i) a lista de funções de agente, (ii) a lista de organizações das quais o agente pode participar, (iii) o ambiente em que está e (iv) a lista de ações que ele pode executar. Essas listas são instâncias da classe java.util.Hashtable. Exceto pela lista de funções de agente, as outras listas são protegidas para garantir o acesso apenas por suas subclasses. Todas as listas com seus cinco métodos de acesso são definidas como seguintes cabeçalhos:

- getXXX (nome) - retorna a instância do nome do elemento da lista;
- addXXX (nome, XXX) - adiciona o elemento XXX na lista;

- removeXXX (nome) - remove o nome do elemento da lista e retorna a lista;
- removeAllXXX () - remove todos os elementos da lista;
- getAllXXX () - retorna a lista de elementos XXX;

Os agentes precisam de sensores para capturar as percepções; esse conceito é representado pelo novo jamder de classe. behavioral.Sensor que herda de jade.core.behaviors. TickerBehavior. É responsável por capturar percepções do ambiente, de tempos em tempos. O período deve ser definido pelo desenvolvedor no Sensor. A percepção é tratada pelo método abstrato chamado percepção (percepção) do GenericAgent.

No JADE, o agente desempenha pelo menos um papel de agente que contém as crenças, objetivos e ações. Ao adquirir o papel, o agente adquire as crenças e objetivos do papel. O agente só pode executar suas ações se as mesmas ações forem definidas pela função do agente.

4.1.1. Características estruturais do agente

As classes JADE que podem ser usadas para representar as propriedades dos agentes são essenciais. Para tanto, é necessário verificar os conceitos de cada um, quais sejam: crenças (crença), objetivos (meta), planos (plano), mensagens (mensagem ACL) (FIPA), ações (ação) e condição (condição).

No JADE não foram encontrados correlativos para crenças e objetivos. Devido à estrutura semelhante entre crença e objetivo (cada um é composto por campos: nome, tipo e valor) e, como esses campos, são as propriedades de cada agente, exceto o agente reflexo simples e o agente reflexo baseado em modelo, o jamder.structural A classe .Property foi criada e contém esses campos, que são herdados por jamder.structural.Belief e jamder.structural.Goal representando as crenças e os objetivos, respectivamente.

As metas podem ser simples ou compostas; portanto, o jamder.structural.LeafGoal e jamder das respectivas classes. estrutural.CompositeGoal representam esses recursos e herdam da meta. Além disso, a classe Goal contém uma lista de planos associados a um objetivo e o atributo booleano, alcançado, indica se o objetivo foi alcançado ou não.

4.1.2. Características comportamentais do agente

Ações são as características comportamentais dos agentes (Weiss, 1999) e, portanto, devem ser implementadas em associação com a classe JADE, jade.core.behaviors.Behavior. Uma classe chamada jamder.behavioral.Action é criada para esse fim como uma

subclasse de Behavior. Uma lista de ações indica as ações que o agente pode executar. Uma ação na lista é realmente executada chamando o método addBehavior (action).

A execução das ações ocorre apenas se todas as suas condições prévias forem atendidas. Por outro lado, as pós-condições definem as condições que serão verdadeiras quando a execução da ação for concluída com êxito. Nesse sentido, a classe Action define dois atributos: as listas de pré-condições e pós-condições, ambas representadas por um jamder. behavioral.Condition, definido como uma subclasse de Property. A ação possui um método chamado execute (), responsável por implementar as ações concretas da ação. As pré-condições e pós-condições também têm cinco métodos de acesso, de maneira análoga à lista.

Cada plano define um conjunto de ações e a ordem de execução. A classe correspondente no JADE é SequentialBehavior. Assim, o jamder.behavioral.Plan foi criado e herda de SequentialBehavior. Essa classe define o objetivo associado e uma lista de ações, cuja classificação básica no plano é determinada pelo método abstrato execute (). As ações do agente no plano devem corresponder às ações na função de agente, ou seja, as ações devem ser definidas tanto na função de agente quanto na de agente.

Os agentes se comunicam por meio de mensagens armazenadas em uma fila. Assíncrona, o agente decide o que fazer ao ler cada mensagem. Segundo Weiss (1999), uma mensagem é definida por um rótulo que especifica o tipo de mensagem, o conteúdo, o remetente (agente responsável pelo envio da mensagem) e o destinatário. No JADE, há uma classe correspondente denominada jade.lang.acl.ACLMessage (JADE).

A classe básica jade.core.Agent define uma lista de mensagens recebidas do tipo ACLMessage. Considerando que os tipos de agentes propostos herdam dessa classe, esse recurso também é herdado pelos agentes no JAMDER. As mensagens recebidas da fila são armazenadas até serem lidas e podem ser recuperadas através do método receive () do Agent.

O MAS-ML define que o agente possui uma fila de mensagens recebidas e uma fila de mensagens enviadas, mas no JADE a faixa de mensagens enviadas não é armazenada. Para atender a esse requisito, a classe GenericAgent é definida

para armazenar as mensagens enviadas por um agente por meio do método sendMessage (ACLMessage).

O JADE fornece um identificador para cada agente, consistindo em um corpo interno do jade.core.AID. Este atributo é criado automaticamente e contém apenas os elementos necessários para localização, acesso a seus endereços e nome local. Ele pode ser recuperado através do método getAID () do Agent. Esse recurso é importante para tratar e lidar com o envio de mensagens. Tipos de agentes no JAMDER Nas subseções a seguir, cada arquitetura interna é descrita como é estruturada no JAMDER de acordo com o tipo definido pelo TAO +.

Agente Reflex simples. No JAMDER, esse agente é definido pela classe jamder.agents.ReflexAgent, que herda de GenericAgent. Esta classe contém uma lista de regras de ação e condição com o tipo Hashtable <String, String>, em que o primeiro nome representa a percepção e o segundo corresponde ao identificador da ação.

Crenças e objetivos não são componentes da estrutura de funções do agente, em consistência com a estrutura do agente correspondente.

O Agente Reflex baseado em modelo é caracterizado por armazenar o histórico de ações, que pode ser usado na tomada de decisões, por exemplo, ele não executa a mesma etapa (Russell e Norvig, 2002). Esse agente é representado pela classe jamder.agents.ModelAgent que estende a classe ReflexAgent e inclui um atributo para representar o conhecimento ou crenças históricas como uma lista de jamder.structural.Belief. Além disso, a classe ReflexAgent incorpora o método abstrato nextFunction (crença, percepção), responsável por mapear as percepções e o estado interno atual (representação do modelo ou ambiente) para um novo estado interno atualizado, mostrando a próxima ação a ser selecionada (Russell e Norvig, 2002).

Outros agentes não podem acessar crenças, mas a classe concreta que herda desse agente pode mudar suas crenças. Portanto, seus métodos são protegidos. Esse tipo de agente, adquirindo a função, incorporará todas as crenças provenientes da função, ou seja, se houver crenças semelhantes, as crenças do agente serão substituídas pelas da função. O método addAgentRole (nome, função) está sobrecarregado para incorporar esse princípio. Como os objetivos não fazem parte desse tipo de agente, seu respectivo

agentRole (ModelAgentRole) também não possui essa característica.

O agente com planejamento baseado em metas pode gerar um plano de ação em tempo de execução. Esse tipo de agente também possui um conjunto de objetivos a serem alcançados pelo agente. Os planos são compostos por ações, sequencialmente. A classe para esse agente, que herda de GenericAgent, é jamder.agents.GoalAgent. Essa classe incorpora três métodos abstratos adicionais, a função de meta de formulação que pega o estado atual (crença ou modelo) e retorna uma meta formulada, a função de problema de formulação que identifica as ações necessárias para tentar o estado e a meta e o método de planejamento que retorna resultado das ações. Além disso, esse agente possui percepções e a próxima função que está presente nessa hierarquia através da lista de percepções e do método nextFunction (crença, percepção).

Ao adquirir a função, o agente baseado em metas também incorporará as metas da função. Se houver os mesmos objetivos, os objetivos do agente serão substituídos. O método addAgentRole (nome, função) nesta classe está sobrecarregado para incorporar esse princípio.

O método abstrato para implementar a função de meta formulada, representada por formulateGoalFunction (crença), recebe uma crença e retorna a meta formulada. O desenvolvedor deve fornecer o algoritmo que implementa esse método. Da mesma forma, o método abstrato responsável pela formulação da função de problema representada por formateProblemFunction (crença, objetivo) retorna um problema a ser usado no planejamento. O problema é um subconjunto de ações de ações de agentes (Russell e Norvig, 2002). Finalmente, o método de planejamento, planejamento (ações), recebe o problema (lista de ações) e retorna uma lista seqüencial dessas ações para atingir a meta. Segundo Gonçalves et al. (2011), o planejamento é baseado nas ações disponíveis na lista de ações, para criar uma sequência de ações (plano). Como os agentes reativos, essa hierarquia também precisa de um sensor para cuidar das percepções do ambiente. Existe o método de percepção (percepção) nesse agente que usa a classe Sensor para atingir esse objetivo.

O agente baseado em utilitários possui a mesma estrutura do agente baseado em objetivos com o planejamento.

Além disso, a função de utilitário é incorporada para orientar o comportamento do agente. Quando o design pode ser vinculado a mais de um objetivo a ser alcançado, esses objetivos podem ser conflitantes e, portanto, a função de utilitário é inserida para determinar o grau de valor dos objetivos associados. Portanto, a classe que representa esse tipo de agente é jamder.agents.UtilityAgent. Esta classe herda de GoalAgent. O utilitário fornece uma maneira de considerar a probabilidade de sucesso sobre a importância do objetivo (Russell e Norvig, 2002).

O método abstrato utilityFunction (action) é usado para identificar a prioridade das ações, com base na ação recebida, e verifica a prioridade dos objetivos sobre o agente. Esse método é chamado várias vezes quando, na formulação do problema, identifica as ações que serão executadas. O retorno desse método consiste em um valor indicando a ação com a maior prioridade.

Agente baseado em metas com plano. O agente com base em objetivos com plano também é chamado de agente BDI no contexto da engenharia de software orientada a agentes (Nunes et al., 2011). Ele é herdado do GenericAgent e contém atributos de crenças e objetivos, além de planos e ações. A função de objetivo de formulação, função de problema de formulação, função seguinte, função de utilidade e percepção não estão considerando neste caso. Uma diferença importante é que seus planos são definidos no tempo de modelagem, o que os torna estáticos. A classe que representa esse agente é o jamder. agentes.MASMLAgent.

4.2. As outras entidades no MAS

Além do conceito de agente, JAMDER considera a representação das outras entidades que frequentemente aparecem no MAS. Assim, representações específicas da implementação da função do agente, organização, ambiente, objeto e função do objeto são incluídas na estrutura.

4.2.1. Funções de agente

Uma função é um elemento que guia e restringe o comportamento social de um agente ou suborganização na organização. Cada instância da função de agente é membro de uma organização e determina o que o agente pode e deve fazer dentro de uma organização (Silva et al., 2003). Para representar o conceito de função do agente no MAS, JAMDER segue a hierarquia do agente definida na Figura 4, que designa uma

função específica do agente relacionada ao tipo de agente específico. Essa definição de funções de agente foi necessária porque uma função de agente adiciona novas crenças e objetivos às crenças e objetivos do agente, mas, dependendo do agente, alguns deles não contêm essas características. Com base na definição, foi criada a classe jamder.role.AgentRole e outras classes associadas para adequar as propriedades a ela. Essa classe está relacionada ao ReflexAgent e qualquer atributo estrutural é necessário nesse caso, porque o processo de seleção é baseado em regras de ação e condição. As propriedades básicas para funções de agente são:

- owner - Instância da organização em que a função está definida. Esta propriedade pode ser recuperada através do método getOwner (). Como o proprietário não muda, sua definição é feita dentro do construtor AgentRole;
- nome - identificador da função do agente (String), definido e recuperado pelos métodos de acesso, respectivamente, getName () e setName (nome da String);
- jogador - indica quem está exercendo o papel. É representado pela classe GenericAgent. Pode ser uma instância de agente ou organização, esta última no caso de uma suborganização. Como o player também não muda, sua definição é feita dentro do construtor AgentRole;

O ciclo de vida de uma função de agente começa quando é associado a uma entidade (agente ou suborganização) (Silva et al., 2003) (Silva et al. 2007) (Gonçalves et al., 2010). Isso significa que a instância da função de agente é criada após a criação da entidade associada. O link entre a entidade e a função de agente pode ser cancelado; nesse caso, é necessário que a entidade tenha outras funções de agente na mesma organização. Caso contrário, a instância da entidade também será cancelada, porque a entidade deve ter um link com pelo menos uma organização.

As funções de agente funcionam como ações que os agentes precisam executar e, quando o agente tem alguma função, incorpora as crenças ou objetivos da função, se houver. No JAMDER, o status definido para uma função de agente inclui: ATIVAR, DESATIVAR e MUDAR. Essas informações são obtidas usando o método getAgentRoleStatus () na classe

AgentRole. O status CHANGE informa que o agente está migrando de um ambiente para outro ou de uma organização para outra. O activeRole reinicia as ações do agente que desempenha a função e, por outro lado, o método changeDeactivateRole (AgentRoleStatus) retorna o status e desativa a função.

Os deveres e direitos das funções definem as ações atribuídas ao agente que desempenha a função relacionada às responsabilidades e permissões, respectivamente (Weiss, 1999). Esses atributos no AgentRole são definidos como uma instância da classe Hashtable <String, XXX>, em que XXX representa uma instância de jamder.behavioral.Duty ou jamder. comportamental. As classes Dever e Direito foram criadas no JAMDER para identificar um atributo que define a ação associada. A classe Action usada pelos agentes também representa uma ação associada a deveres e direitos, em que cada dever ou direito possui apenas uma ação. A criação de uma instância de função de agente envolve a análise de direitos e deveres que chamam o método initialize no construtor da classe.

Um protocolo define um grupo de mensagens que um agente pode enviar para outros agentes (Bellifemine, 2007). No JADE, existem vários protocolos compatíveis com o FIPA para padronizar a comunicação entre os agentes. Um protocolo de comunicação é definido pelo tipo e pelo comportamento dos participantes da comunicação. Assim, os protocolos na classe AgentRole são definidos usando uma instância de Hashtable <String, Behavior> em que Behavior representa uma extensão da classe iniciante ou participante. As funções desempenhadas por agentes baseados em modelo (ModelAgent) requerem a definição de crenças que são incorporadas pelo agente ao se comprometer com a função. A crença no papel do agente também é definida pela classe Belief. No caso de o agente ter crenças com o mesmo nome definido em sua função de agente, a crença na função substitui a crença no agente. A classe para representar esse tipo de função de agente no JAMDER é jamder.role.ModelAgentRole que estende a classe AgentRole e incorpora o atributo de crença. Essa função de agente está relacionada apenas aos agentes de modelo, o que significa que apenas instâncias do ModelAgent podem exercer esse tipo de função de agente.

Por fim, a classe jamder.role.ProactiveAgentRole herda da classe ModelAgentRole e incorpora objetivos. A característica da meta também é representada pela classe de meta. Os agentes proativos podem desempenhar esse papel, ou seja, GoalAgent, UtilityAgent e MASMLAgent. À medida que a suborganização se comporta como um agente, também pode exercer esse tipo de função de agente.

4.2.2. Objeto e função de objeto

Por outro lado, o objeto e a função do objeto são representados pelas classes Object de java e jamder.roles.ObjectRole no jamder, respeitosamente. O objeto responde apenas às solicitações que foram solicitadas. A classe ObjectRole define três atributos: nome da função, o objeto que desempenhará a função e a organização à qual pertence. Essa função auxilia a execução de um objeto semelhante ao que acontece com o agente de rolagem e o agente. Também é definido como uma organização e seus atributos são explicados da seguinte forma: identificador da função do objeto (nome); um objeto que exercerá essa função (objeto); e a organização da qual essa função é membro (proprietário).

4.2.3. Organização

A organização é um local onde os agentes agem, está em um ambiente e pode permanecer em apenas um, ou seja, não pode mudar ou mudar para outro ambiente. Ele é definido no JAMDER, pois a classe Organization é definida como uma extensão do MASMLAgent (Figura 3) e descreve um contêiner JADE em que as funções de agente e de objeto são definidas. Uma organização pode conter suborganizações, recursivamente. O atributo Hashtable <String, Organization> é usado para representar suborganizações, incluindo o nome e a instância de suborganização, respectivamente. Analogamente, uma instância de suborganização é uma organização e pode conter uma nova suborganização, portanto, esse relacionamento é estabelecido com o atributo superOrganization.

O JADE possui uma classe denominada ConteinerID que representa o identificador da organização. Cada instância contém o atributo containerID com o respectivo container (JADE) associado à organização no JAMDER. Como a organização herda de MAS_MLAgent, a herança de ContainerID não é possível.

Devido à classe Organization, estende a classe MASMLAgent. Assim, são incorporadas as características estruturais e os métodos de acesso dessa entidade, como Crenças, Metas, Planos e Ações. Além disso, uma organização inclui axiomas usados para controlar as ações do agente. O conceito de axioma é representado por jamder.Structural.Axiom, é uma subclasse de Propriedade, pois compartilha a mesma estrutura e restringe as ações de agentes e sub-organizações que residem na organização. O axioma é uma particularidade da organização e é tratado na classe Organização.

Os agentes da organização são obtidos por meio das funções que a organização possui quando a função do agente sabe que a organização está envolvida e o agente que a executa. Como a organização herda a lista de funções do MAS_MLAgent, essa lista pode ser executada de duas maneiras:

- No caso de uma organização, a lista consiste em todas as funções de agentes aceitas pela organização ou nas funções a serem armazenadas ou desempenhadas por um agente ou por uma suborganização;
- Se a instância tiver a função de suborganização, a lista de funções do agente consistirá nas funções de agente exercidas pela suborganização. O atributo superOrganization deve conter a instância da Organização que pertence a ser executada como suborganização.

Essa classe também define as funções de agente e de objeto. O atributo é especificado como uma instância do Hashtable <String, XXX>, onde XXX pode ser executado por AgentRole ou ObjectRole. Este atributo armazena todas as funções de agente ou objeto que o agente ou objeto pode executar. Seus métodos de acesso seguem o mesmo caminho dos outros atributos. Na organização, as propriedades de envio e recebimento de mensagens são tratadas da mesma maneira que os agentes.

4.2.4. Meio Ambiente

Finalmente, o ambiente representa a plataforma JADE, onde residem os contêineres (representados pela entidade da organização) e agentes, e define métodos para gerenciar (adicionar e remover) outras entidades. Essa entidade é representada pela classe Environment em JAMDER. Quando

uma instância do ambiente é criada, o construtor Environment (nome, host, porta) recebe o nome da plataforma, o nome da máquina (host) e a porta da máquina que serão armazenadas na plataforma. Ao criar um ambiente, a plataforma cria o contêiner principal que manterá os serviços da plataforma, como o Directory Facilitator (páginas amarelas) e o Agent Management System, onde este último gerencia as próximas organizações (contêineres) e agentes que serão criados nessa plataforma.

Quando a plataforma JADE é iniciada, ela pode fornecer os serviços necessários para o ciclo de vida das outras entidades, como por exemplo, o agente que procura o serviço. Nesse caso, como a classe Environment no JAMDER não possui uma superclasse do JADE, ela atua como um adaptador de um ambiente usando a plataforma JADE. Cada plataforma adquire naturalmente um identificador que é uma instância do jade.core.PlatformID. Da mesma forma, a classe Enviroment possui o atributo ID, onde é uma instância do PlatformID. Além disso, outros atributos compõem o ambiente como: nome, endereço e assim por diante. Além dessas propriedades, a classe Environment também possui outros métodos que gerenciarão o ambiente, por exemplo, adicionando uma organização no ambiente, o método addOrganization (String, Organization) aumenta uma instância de Organization na lista de organizações, e também cria um contêiner na plataforma JADE.

O processo inverso, isto é, a organização e a remoção do agente, precisa ser analisado com cautela, porque várias dependências precisam ser observadas. Primeiramente, as relações com agentes, suborganizações e objetos que habitam a organização devem ser descartadas. Se uma organização tiver agentes e objetos, mas se algum agente também participar de outra organização, somente isso cancelará as funções de agente vinculadas à organização que está sendo excluída. Caso contrário, o próprio agente também será excluído. O método removeOrganization (chave String) implementa esse comportamento se a organização contiver suborganizações, será executada recursivamente.

Supondo que o ambiente conheça todos os agentes que suportam, o procedimento de criação de agentes é necessário nesta classe por meio de addAgent (chave String, agente GenericAgent), em que a chave é o nome e a instância do

agente. A remoção de um agente é mais simples do que a remoção de uma organização porque o agente possui menos dependências. O processo de remoção do agente verifica inicialmente as funções em que organizações o agente está compactado. Ao remover o contrato com a função de agente, a instância do agente acaba não existindo. O removeAgent (nome da string) executa esse comportamento. As mudanças que o agente percebe acontecem no próprio ambiente. Essas mudanças vêm das ações que os próprios agentes executam, ou seja, quando um agente executa alguma ação, ele pode alterar qualquer estado do ambiente.

Mesmo que o agente possa se mover entre ambientes, não é possível estar nos dois ambientes ao mesmo tempo. Consequentemente, o agente deve cancelar todas as funções e relacionamentos que exerce antes de chegar a outro ambiente, bem como nas organizações nas quais exerce uma função. Ao entrar no novo ambiente, o agente é instanciado em uma organização e começa a exercer uma função de agente nessa organização. Se a migração ocorrer entre organizações, ela altera o status das funções do agente para CHANGE e suas ações são concluídas ou finalizadas.

No JADE, todo agente tem três métodos que ajudam quando estão se movendo. Esses métodos estão sobrecarregados no GenericAgent para adaptar a migração da equipe para atualizar as organizações e o ambiente. O método beforeMove () contém o mecanismo responsável por executar uma ação antes que o agente circule. Em seu código, o status das funções do agente é modificado para CHANGE. O método afterMove () é executado após a migração do agente e, em seu conteúdo, o desenvolvedor pode fornecer informações sobre o que pode ser feito depois que o agente se deslocar. Se a mobilidade for entre organizações, as funções de agente (da organização) que estão com o status CHANGE mudam para status ATIVAR e as ações atribuídas a essa função são novamente exercidas. Se a mobilidade for entre ambientes, ela exercitará as ações da função de agente que são obtidas na organização do ambiente de destino. Por fim, o método doMove (Location) é operável para verificar o desempenho adequado da migração. O parâmetro location é uma superclasse de ContainerID e PlatformID e é responsável por informar a organização ou o ID do ambiente, respectivamente.

5. Geração de código

No contexto da abordagem MDA, é proposto o suporte automatizado à geração de código para o MAS. No processo proposto, são aplicados os seguintes componentes para gerar código: Ferramenta MAS-ML (PIM); Framework Java / JAMDER (PSM); e modelos Acceleo (PDM). A ferramenta MAS-ML (Silva et al., 2007) (Gonçalves et al., 2015) é um plug-in do Eclipse 9 que segue uma abordagem direcionada por modelos para suportar a modelagem da linguagem MAS-ML 2.0 (Gonçalves et al., 2011) pelos conceitos e abstrações definidos no MAS.

A ferramenta MAS-ML suporta a modelagem dos seguintes diagramas: diagrama de organização, diagrama de funções e diagrama de classes, com base no metamodelo MAS-ML. Essa ferramenta gera o arquivo masml (XMI - XML Metadata Interchange) que armazena a estrutura das entidades de dados e os aspectos estruturais e comportamentais definidos no MAS-ML 2.0. Esses arquivos são a entrada para o processo de transformação em geração de código usando o plug-in Acceleo para suportar o conceito de MDA, pois permite a geração de código em diferentes linguagens de código e o desenvolvimento incremental.

Para formalizar a geração de código no Acceleo, é necessário estabelecer um modelo para cada entidade por meio de uma linguagem definida pelo OMG, o MTL (Model Transformation Language). Quando o modelo é executado no Eclipse, ele precisará da representação do modelo MAS-ML (arquivos .masml) e da pasta de saída para armazenar as classes JAMDER).

5.1. Meio Ambiente

O ambiente é uma instância de uma classe que herda da classe Environment no JAMDER e sua representação ocorre através do EnvironmentClass na ferramenta MAS-ML. Como essa instância herda os métodos definidos em Ambiente, é necessário apenas, na geração do código, o construtor dessa classe chamar a superclasse e criar organizações, agentes e funções de agente nessa ordem. A necessidade de criar instâncias de função de agente é apenas para vincular a organização e as instâncias do agente, nas quais a construção da função de agente faz esse link. O mapeamento de agentes de criação e organização de corpos ocorre por meio da relação entre o ambiente e essas entidades. Os objetos e funções de objeto também são criados no ambiente em que as funções são obtidas pela organização por meio do relacionamento de propriedade. Por sua vez, os objetos são alcançados pelo objeto através do relacionamento lúdico.

Se necessário, métodos ou atributos adicionais, a Ferramenta MASML oferece os componentes Operação e Propriedade para essa necessidade, respectivamente. A Figura 4 mostra o modelo para gerar o ambiente.

Um recurso importante a ser observado é que a função do agente precisa saber qual é a função do agente inicial. agente ou sua

sub-organização que exerce essa função de recurso, enquanto o agente ou sub-organização é necessário para que a função conhecida da inicial seja agente. Para resolver este problema, o parâmetro de função do agente começa nulo. função de agente, ou agente ou sub-organização, sua função que as necessidades iniciais começam a ser nulas. Ao criar a função de agente, seu construtor utiliza como parâmetro a instância do agente ou suborganização em que esse construtor define o agente ou suborganização, a função exercida inicialmente por meio do método addAgentRole (AgentRole).

5.2. Objeto e função do objeto

A função do objeto no JAMDER é representada pela ferramenta MASML para a entidade ObjectRoleClass. A estrutura desta classe não possui muitos atributos, por sua vez, gerar novo objeto dessa classe herda apenas funções. Enquanto a entidade objeto é qualquer objeto Java e que o acceleo já cria por meio de sua classe de entidade predefinida. A Figura 5 mostra os detalhes do modelo.

5.3. AgentRole

A ferramenta MAS-ML da entidade AgentRoleClass suporta três tipos de agentes definidos pelo MAS-ML 2.0. Se a função não tiver crenças e objetivos, a função do agente será AgentRole. Se a função não for apenas crenças, a função do agente será ModelAgentRole. E se a função contiver toda a estrutura disponível, será o tipo de ProactiveAgentRole. O modelo im .mtl que trata da função de agente cria a herança correspondente aos recursos encontrados (figura 6).

No final deste modelo, é realizada a chamada do método initialized () na classe AgentRole, estrutura JAMDER, para verificar o direito e os deveres devem ser exercidos por função. Em relação ao protocolo, como no JADE, existem tipos diferentes e eles de alguma forma herdam do comportamento, e o desenvolvedor precisa informar que tipo de protocolo no JADE será usado.

5.4. Agente

O modelo para a geração do agente é análogo ao modelo de função do agente, pois cada agente herda da classe correspondente ao agente JAMDER e o tipo de agente depende dos componentes que o agente contém. A Figura 7 mostra o modelo para a criação de todos os tipos de agentes definidos no MAS-ML 2.0.

Observe que a modelagem do agente precisa ser consistente com a sua marca, ou seja, sua estrutura deve estar em conformidade com a especificação MAS-ML 2.0, portanto, em sua geração, as propriedades do agente também correspondem ao seu tipo de agente atualmente.

Para facilitar a definição do tipo ou herança de agente de sua estrutura, é feita a verificação do diagrama dos componentes na seguinte ordem:

- Se o agente herda de outro agente, o relacionamento de herança é mantido;
- Se o agente não definir crenças e objetivos, ele herdará do ReflexAgent;
- Se o agente definir crenças, mas não tiver metas, ele herdará do ModelAgent;
- Se o agente definir um plano predefinido, ele herdará do MASMLAgent;
- Se o agente definir um plano para definir (planejamento) e não possuir utilityFunction (), esse agente herdará do GoalAgent;
- • Se o agente definir um plano para definir (planejar) e possuir uma utilityFunction (), esse agente herdará do UtilityAgent;

O componente Action of AgentClass definido no MAS-ML Tool funciona de duas maneiras na representação de agentes. Cada atributo desse componente possui um campo chamado ActionSemantics que pode ou não ser concluído.

Se o campo ActionSemantics não for preenchido na Ferramenta MAS-ML, isso significa que esse atributo é uma das ações que o agente pode executar, ou seja, é uma instância de Action of JAMDER. Um ponto importante sobre as ações é que, no MAS-ML, elas podem ter pré-condições e pós-condições, mas na versão atual do MAS-ML Tool, esse recurso ainda não está contemplado. Por outro lado, se o campo ActionSemantics estiver preenchido, ele funcionará como um dos métodos que executam o agente, dependendo da estrutura do agente. A alocação ou definição de métodos dependerá do tipo de agente que o designer deseja definir. A conclusão do campo ActionSemantics pode conter e executar uma das seguintes situações (Figura 8):

- <<PróximaFunção>>: indica que o agente tem a função do tipo nextFunction () e é útil para agentes reativos com conhecimento, ModelAgent;
- <<Formulate-Problem-Function>>: indica que o agente tem a função do tipo formateProblemFunction () e é usado para o planejamento com agentes, GoalAgent;
- <<Formulate-Goal-Function>>: indica que o agente tem a função do tipo formateGoalFunction () e é usado para o planejamento com agentes, GoalAgent;

- <<Utility-Function>>: indica que o agente tem a função do tipo utilityFunction () e é usado para o planejamento com agentes, UtilityAgent.

Embora esses métodos contenham algum estereótipo que identifique sua função, o método também contém um nome, mas em JAMDER, os nomes dos métodos são abstratos e já definidos. Em relação a resolver esse problema, o código gerado compreende dois métodos para cada propriedade ActionSemantics preenchida, caso o método JAMDER e o método com o nome usado na modelagem, em que o primeiro se refira ao segundo método. Por exemplo, se o agente contiver a propriedade <<Next-Function>> como nextGroups, o gerador gerará o método nextFunction () que chama o método nextGroups ().

Se o agente contiver algum atributo no compartimento Planejamento, isso indica que este agente (GoalAgent) contém a função planning (), usada para montar o plano em tempo de execução. A criação do plano ocorre de duas maneiras, plano ou planejamento. A primeira forma consiste em um plano definido no tempo de modelagem, que compreende em sua estrutura o atributo ownedAction que mantém suas ações e faz parte do agente. O segundo formulário é um plano sem ação, mas que pode ser construído em tempo de execução pelo agente. Ambos os tipos de plano da Ferramenta MAS-ML são instâncias da classe Plan no JAMDER.

As listas de mensagens enviadas e recebidas do agente não são definidas quando a criação do agente. Os métodos de acesso a essas listas são herdados da classe GenericAgent de JAMDER.

5.5. Organização

O nó organizationClass na ferramenta MAS-ML e a classe de organização em JAMDER representam o conceito de organização. Sua estrutura é semelhante aos agentes em termos de crenças, objetivos, ações e planos. Além desses componentes, a estrutura da organização compreende os axiomas.

O processo de criação das suborganizações é o mesmo que uma organização, no entanto, quando o parâmetro do construtor, org, não é nulo; isso significa que a organização atual possui uma superorganização. Esta regra está detalhada em JAMDER. a figura 9 mostra o modelo para oraganização.

Cada arquivo .mtl criado contém o modelo de geração de cada entidade no Acceleo, que permite a geração de código para JAMDER a partir de modelos projetados na ferramenta MAS-ML. Um processo semelhante pode ser usado para gerar código para outros frameworks

ou linguagem de programação. O processo em que o processo garante a correção do código para construção.

6. Estudo de Caso

Este capítulo ilustra a geração de código para desenvolver um MAS para o ambiente de aprendizado colaborativo Moodle10. O Moodle é uma Fonte de Gerenciamento de Curso em Sistema Aberto - Sistema de Gerenciamento de Curso (CMS), também conhecido como Sistema de Gerenciamento de Aprendizagem (LMS) ou um Ambiente Virtual de Aprendizagem (AVA), que é uma representação da interação entre alunos e professores em um ambiente virtual. O objetivo do estudo de caso é ilustrar a aplicação da abordagem proposta, criando um sistema multiagente para o Moodle no MAS-ML Tool. Depois disso, o modelo é usado para inserir o processo de geração de código. Inicialmente, ele deve criar um projeto Acceleo na ferramenta Eclipse que contém as classes e os modelos Acceleo (arquivo .mtl) para gerar código no JAMDER.

Para executar a geração de código nesse design de protótipo do MAS, foi utilizado o diagrama de organização da Ferramenta MAS-ML (Figura 12), incluindo: agentes, organização, funções e ambiente do agente e os relacionamentos entre os membros, propriedade e habitação. Cada modelo será executado individualmente usando o diagrama que contém a entidade necessária ao modelo para gerá-los. No entanto, todas as propriedades das entidades como crenças, objetivos, entre outras, foram omitidas neste diagrama; devido ao limite de espaço de exibição no texto do trabalho, no entanto, a estrutura das entidades possui as propriedades que serão detalhadas nas seções a seguir.

A representação do ambiente para este projeto possui apenas uma instância do Environment no JAMDER, onde essa instância contém outras entidades, aqui representadas como MoodleEnv. Tendo o conhecimento de que o objetivo do sistema moodle é trazer alunos e professores virtualmente, a representação da organização é apenas uma instância da organização no JAMDER, neste caso, representada pelo MoodleOrg. Por causa dessas características. todos os atores e papéis deste MAS estão na mesma organização e, portanto, no mesmo ambiente.

Um ponto importante a ser observado neste protótipo do MAS é que as crenças de agentes e funções de agente foram representadas na modelagem como arquivos .pl, onde esses arquivos contêm informações de crenças a serem lidas pela classe gerada. A função deste arquivo contém apenas as crenças e suas informações serão obtidas em sua palestra, adaptando o código após ser gerado pela ferramenta. A seguir, são apresentados detalhes das entidades de agente e funções de agente envolvidas neste protótipo do MAS.

6.1. Agentes

Com base na idéia do sistema Moodle, os seguintes agentes foram identificados para este ambiente no MAS-ML 2.0.

6.1.1. Agente de aprendizagem complementar

Esse tipo de agente deve poder escolher de forma independente entre um intervalo predeterminado de estratégias de interação afetiva, como mensagens de suporte. Apresenta mensagens encorajadoras (reforço positivo) quando o usuário, por meio das interações manifestas, fornece evidências simples para acompanhar discussões e / ou tarefas e / ou conteúdos propostos, e mesmo quando o aluno apresenta muitos ritmos acima da média na sua classe ou grupo de trabalho. Devido à necessidade de manter anotações de aula para comparação e enviar mensagens rapidamente, esse agente é caracterizado como um agente reativo baseado no conhecimento. Sua estrutura é apresentada abaixo e ilustrada na Figura 11.

Esse agente deve conhecer as estratégias de interação afetiva e as mensagens de apoio (mensagens de impressão emocional) que serão dadas ao usuário por suas crenças. Quando o usuário, através das percepções das interações, mostra sinais de dificuldade para acompanhar a discussão e / ou as tarefas e / ou conteúdos propostos, o atributo perceptivo é identificado pelas dificuldades da discussão e pode ser obtido através das faces dos emoticons. A percepção studentContentAccess foi projetada para determinar se o aluno está acessando o conteúdo.

A ação compareClass se compara à classe e verifica se o aluno é muito mais alto ou muito mais baixo; no entanto, antes dessa comparação, é necessário que a pré-condição studentAverage seja diferente de zero. A ação showSupportMessage exibe uma mensagem de suporte, mas a pré-condição noDificuldade deve ser atendida e o aluno deve estar obtendo conteúdo. A ação showReinforcementMessage tem um efeito oposto, onde a pré-condição studentWithDificuldade deve ser verdadeira. A ação requestCoordinatorAction solicita uma ação de outro agente para entrar em contato com o coordenador para atender às pré-condições de classTipMsgs e da dificuldadeFeatures, entre outras. A Figura 14 mostra o código gerado para o CompanionAgent no JAMDER.

De acordo com a Figura 12, o código gerado para esse agente tem em seu construtor uma chamada para sua superclasse, neste caso, ModelAgent. Logo depois, há a criação de instâncias de ações relacionadas a esse agente, o método addAction (String, Action), que informa que essas instâncias estão sendo incluídas na lista de ações do agente e,

finalmente, as percepções são definidas. Os métodos projetados para esse agente foram nextFunction (Crença, Percepção) e learningNextFuction (Crença, Percepção).

6.1.2. Assistente de Aprendizagem (Pedagógica)

Esse agente deve poder acompanhar o aluno nas diferentes disciplinas que participam para contribuir com o usuário por meio de dicas, sugestões e mensagens sobre o tópico em andamento e não apenas a natureza afetiva das mensagens (suporte). É um agente baseado em objetivos com planejamento, porque ele precisa criar um plano de estudo, sugerindo disciplinas para o aluno com base nas disciplinas que ele está realizando (figura 13).

O agente pedagógico percebe a (s) disciplina (s) existente (s) em que o aluno está matriculado. A partir das percepções, as ações pertencentes ao agente incluem ou relacionam as disciplinas nas quais o aluno está matriculado e sugerem outras disciplinas que o aluno frequenta.

Como um agente com base em objetivos e com planejamento, os métodos pertencentes a esse tipo de agente precisam ser implementados pelo desenvolvedor, por exemplo, para configurar seu plano de tempo de execução para atingir seu objetivo, onde o plano é acompanhar o aluno. A Figura 14 mostra o código gerado para PedagogicalAgent no JAMDER.

6.1.3. Agente do mecanismo de pesquisa de informações

Esse tipo de agente encontra pessoas no ambiente moodle envolvidas em disciplinas relacionadas a um tópico específico de interesse. O objetivo deste agente é fazer pesquisas contínuas e autonomamente, mostrar documentos e contatos pessoais sempre que localizar possíveis interesses comuns. O Figura 15 mostra a estrutura do agente.

As crenças dos agentes armazenam as pessoas, grupos e disciplinas. O objetivo relacionarPessoas verifica as pessoas no ambiente Moodle que estão envolvidas em projetos ou assuntos em comum. O objetivo relacionar páginas de pesquisa de documentos como documentos, projetos ou outros arquivos digitais que você tem em comum, por exemplo, a palavra-chave.

As ações pertencentes a esse agente podem ser de quatro maneiras. A ação searchPeople, como o nome indica, localizou as pessoas relacionadas ao mesmo tópico do usuário. A ação showRelatedPeople é usada para exibir o resultado, onde a pré-condição dessa ação é o resultado da localização de outras pessoas que é diferente de zero. Da mesma maneira, as

ações searchDocuments e showRelatedDocuments são usadas para documentos.

Como esse agente já sabe o que fazer e como fazer, ele é classificado como um plano com agente baseado em objetivos. Nesse caso, ele possui dois planos predefinidos. O plano searchPeopleInformationPlan possui duas ações, searchPeople e showRelatedPeople, nessa ordem, para atingir a meta relatePeople. Além disso, da mesma maneira, o plano searchDocumentInformationPlan usa as ações seekDocuments e showRelatedDocuments, nessa ordem, para atingir o objetivo relatedDocuments. A figura 16 mostra o código para o searcherAgent gerado no JAMDER.

As ações searchPeople e showRelatedPeople para o plano searchPeopleInformation e as ações searchDocuments e showRelatedDocuments para o plano searchDocumentInformation foram incluídas manualmente em seus respectivos planos após a geração desta classe para a Ferramenta MAS-ML; a versão atual ainda não tem associação entre o plano e as ações semanticamente.

6.1.4. Agente fornece ajuda no Moodle

Esse agente é do tipo reativo simples e possui uma lista de vários insights sobre as dificuldades que o usuário tem e, antes disso, escolhe a ação apropriada. Esse agente percebe em que momento o usuário está ao mesmo tempo, oferecendo dicas independentes sobre como fazer o melhor uso de uma funcionalidade particular. A Figura 17 mostra que esse agente é composto apenas de percepções e ações; consequentemente, é um simples agente reativo. Para corresponder à percepção, o agente executa uma das ações para abordar a percepção. Eles não têm pré ou pós-condições. A Figura 18 mostra o código gerado para o HelperAgent no JAMDER.

6.1.5. Agente Coordenador

Esse tipo de agente deve poder centralizar as solicitações dos agentes e as informações que eles enviam uns aos outros, tornando esse agente um intermediário entre outros agentes. O objetivo do agente coordenador é ordenar as ações dos agentes (requestAgentsActions). O plano requestActionPlan usa duas ações para atingir esse objetivo: checkAgentAction e requestAction, em que o primeiro verifica se o agente pode executar a tarefa e o segundo solicita que o agente execute a ação após verificar se o agente pode executar (checkAgentAction) (Figura 19).

Da mesma forma, para o SearcherAgent, o CoordinatorAgent é classificado como um plano com um agente baseado em metas. Portanto, a geração de código é omitida neste caso.

6.1.6. GroupAgent

Esse agente deve ser capaz de ajudar autonomamente usuários, estudantes e educadores, na composição de grupos de trabalho, levando em consideração temas de afinidade ou perfis de aprendizado. Para isso, deve considerar certos critérios estabelecidos por um treinador de uma ou mais classes ou pelo usuário interessado em integrar os grupos de trabalho. A Figura 20 mostra a estrutura do GroupAgent.

As crenças desse agente armazenam cursos por assunto e pelo professor ou pelo perfil do usuário que está aprendendo. O planejamento createGroupHelp é montado no tempo de execução do agente e possui dois objetivos, createGroupByLearningProfile e createGroupByAffinity. Eles formam grupos, mas a diferença é a suposição usada para planejar a melhor maneira e, portanto, sugerir maneiras de criar ou ingressar em grupos, seguindo um tema ou perfil de usuários que se enquadram no grupo a ser definido. Essa opção mostra dois objetivos escolhidos para serem alcançados durante o tempo de execução.

A Figura 21 mostra o código gerado para o GroupAgent no JAMDER. Como agente baseado em utilidade, ele possui os métodos definidos no MAS-ML 2.0 e é determinado pelos métodos:

- <<next-function>> nextFunctionGroup: fornece sugestões de criação ou grupos de integração;
- <<formulate-problem-function>> probFuncPerceiveGroups: reconhece a potencial formação de grupos de acordo com um tema ou perfil sugerido;
- <<formulate-goal-function>> probFuncIntegrateGroups: auxilia o usuário a formar o assunto de grupos temáticos, dando assim maior valor ao relacionamento de colaboração entre os membros do grupo, proporcionando melhor aprendizado;
- utilFuncIntegrateGroups: cria um equilíbrio entre problemas e perfis de treinamento;

Neste exemplo de código gerado, as ações foram inseridas manualmente porque a versão atual da ferramenta MAS-ML ainda não fornece a inclusão de informações de ações em sua estrutura. A saber: checkAgentAction e requestAction.

6.2. Agent Roles

Em um sistema MAS, os agentes podem exercer mais de uma função de agente na mesma organização, mas neste protótipo do MAS, cada agente tem uma função específica. Por causa dessa granularidade, as definições dos papéis dos agentes são as mesmas dos agentes em relação às crenças, objetivos e ações, mas os papéis têm outros componentes: direitos e deveres. A Figura 22 mostra mais detalhes das funções propostas para o agente. A estrutura de cada função de agente proposta para cada agente com base nos direitos e deveres é definida da seguinte maneira:

A função de agente de aprendizagem (CompanionAgentRole) é o tipo de função baseada no conhecimento, pois ela precisa armazenar as notas dos alunos por meio de crenças. Ele contém o displaySupportMsg correto, pode ou não exibi-lo e o dever de compareClasses, pois para executar alguma ação, é necessário comparar a classe primeiro.

A função de agente pedagógico (PedagogicAgentRole) é um tipo de função para um agente proativo, porque, dependendo de como é a evolução do aluno, as ações que o agente usará essa função farão o plano do agente.

A função de agente de informações do pesquisador (SearcherAgentRole) é um tipo de função proativa de agente, porque suas ações serão usadas em um plano de agente predefinido. Seus direitos são displayRelatedPeople e displayRelatedDocuments para exibir pessoas ou documentos relacionados aos interesses atuais do usuário. Suas funções searchPeople e searchDocuments identificam a obrigação de localizar pessoas ou documentos relacionados ao usuário atual.

A função de agente auxiliar do Moodle (HelperAgentRole) é um tipo de função de agente reativo simples, portanto, não possui crenças ou objetivos. Ele fornece ajuda na operação do Moodle.

A função de agente de formação de grupos (GroupAgentRole) é um tipo de função proativa de agente devido às próprias crenças e objetivos necessários para ajudar os usuários na composição de grupos de trabalho. Este papel não tem dever, mas tem vários direitos.
A função de agente coordenador (CoordinatorAgentRole) é um tipo de função de agente proativo. Centraliza os pedidos dos agentes e as informações que eles enviam uns aos outros. Assim, seu

comportamento é um intermediário entre outros agentes. Essa função não possui tarefas, no entanto, contém o direito checkAgentAction, que pode verificar qual ação o agente está executando e requestAction certo, que pode solicitar que outro usuário execute uma ação. A Figura 23 mostra todos os códigos de funções de agente gerados no JAMDER propostos neste estudo de caso.

6.3. Ambiente e Organização

Finalmente, as classes Organization e Environment para este MAS são mostradas na Figura 24. A estrutura dessas classes é mais simples porque contém apenas entidades que as habitam e as instâncias que as habitam.

As transformações do código em modelos com gráficos em Acceleo propostos apenas geram o esqueleto de classes e métodos das entidades e suas características. Algumas entidades exigiram incluir, como nomenclatura padrão, um sufixo devido à possível existência de inúmeras propriedades na entidade e, assim, organizar melhor sua estrutura, a saber: Plano (Plano), Ação (Ac), Objetivo (G), Crença (B) Esses nomes padronizados excluem apenas as classes Duty, Right e protocolo, conforme são geradas, suas instâncias são criadas e não são diretamente referenciadas no mesmo método sem a necessidade de uma variável. Essa nomenclatura adotada proporciona uma melhor compreensão das características envolvidas, possibilitando a utilização mais fácil no construtor da entidade. Outra vantagem é que ajuda a identificar rapidamente que tipo de classe ou variável foi gerada e evita possíveis erros de codificação, melhorando a qualidade do código.

Neste caso de estudo, JAMDER foi aplicado ao protótipo de um problema real, ilustrando a adequação da extensão JADE para a implementação de entidades no contexto do MAS-ML 2.0. O código fonte do JAMDER e o caso de estudo completo, bem como os modelos utilizados, podem ser obtidos no site https://bitbucket.org/yrley/jamder/src/.

7. Conclusão

A redução do espaço semântico entre as representações nas fases de design e implementação contribui para o aumento da produtividade do desenvolvedor e melhora a consistência entre as representações de design e código das entidades envolvidas. Além disso, a existência de um mapeamento entre entidades garante consistência e rastreabilidade ao código.

Neste artigo, apresentamos um mapeamento entre entidades e os mecanismos de suporte fornecidos ao JADE. O foco do trabalho são as diferentes arquiteturas internas dos agentes, a saber: agente reflexo simples, agente reflexo baseado em modelo, um agente baseado em objetivos com planejamento, agente

baseado em utilidade e agente baseado em objetivos com um plano. Como resultado, uma extensão JADE é apresentada através de um conjunto de adaptações para suportar a implementação das arquiteturas agendadas. Dessa forma, algumas classes JADE foram estendidas porque compartilham uma estrutura e um comportamento comum, como: Agente e Comportamento. Em outro caso, novas classes foram criadas porque não havia um conceito correspondente no JADE, a saber: Crença, Objetivo (CompositeGoal e LeafGoal), Plano, Ação, AgentRole, ObjectRole, Axioma, Dever, Direito, Ambiente, Organização e Condição.
A aplicabilidade da extensão proposta é ilustrada em um estudo de caso em que o mapeamento no nível da entidade entre a modelagem e sua implementação através do código JAMDER pode ser observado. Este estudo de caso ofereceu uma explicação de um sistema conhecido no campo MASs, o Moodle, que facilita o entendimento de diferentes tipos de agentes e sua implementação. As propriedades estruturais dos agentes propostos foram mostradas; no entanto, suas propriedades comportamentais devem ser desenvolvidas pelo desenvolvedor através dos métodos para cada agente. Cada modelo desenvolvido neste trabalho é realizado individualmente, um para cada tipo de entidade. Como trabalho futuro, consideramos estabelecer uma cadeia de chamadas para esses modelos sequencialmente que permita uma redução de tempo para gerar as classes também.

Este trabalho constitui o passo em direção à promoção do desenvolvimento de MASs usando ao mesmo tempo muitos tipos de agentes e JADE. Com base no mapeamento apresentado, a estratégia a ser seguida no processo de codificação pode ser rastreada. Além disso, nesse processo de codificação, outras estruturas para implementação também podem ser adaptadas à variedade de agentes.