Separação e Validação de Regras de Negócio MDA através de Ontologias e Orientação à Aspectos

Jaguaraci Batista Silva, Luciano Porto Barreto

Departamento de Ciência da Computação / Laboratório de Sistemas Distribuídos

Universidade Federal da Bahia

Campus de Ondina, CEP: 40170-110, Salvador-BA, Brasil

{jaguarac,lportoba}@ufba.br

Abstract. *Two important challenges for MDA designers are to make business rules independent of the platform specific model and to ensure that the resulting implementation does not violate the properties within these business rules. In this paper we present an approach that allows developers to both describe and verify domain properties at run time. In our approach, an ontology specifies domain properties that are translated to aspect-oriented code. This code is automatically merged in the implementation. As a result, we are able to execute validation tests at runtime and to generate business rules that can be reused in other MDA specific models. We evaluated our approach and toolkit by successfully reengineering an industrial furnace management application.*

Resumo. *Dois desafios importantes para projetistas MDA consistem em separar as regras de negócio do modelo específico de plataforma e, por fim, garantir que tais regras sejam atendidas pela implementação resultante. Este artigo apresenta uma abordagem que permite a descrição de propriedades de domínio e a verificação de tais propriedades na implementação. Para isso, é feita uma modelagem das propriedades conceituais através de uma ontologia, a qual é posteriormente transformada em aspectos. Estes aspectos são finalmente combinados e automaticamente inseridos no código da aplicação. Assim, as regras de negócio podem ser geradas automaticamente e reutilizadas em outros modelos MDA específicos da plataforma Java. A ferramenta e abordagem foram validadas através da reestruturação de uma aplicação crítica de gerenciamento de fornos industriais.*

1. Introdução

A Arquitetura Orientada a Modelos ou MDA (*Model-Driven Architecture*) (OMG 2006) é uma abordagem de desenvolvimento de software dirigido por modelos (MDD ou *Model-Driven Development*) que apresenta diferentes níveis de abstração com o intuito de separar a arquitetura conceitual do sistema de sua implementação específica. Dessa forma, o foco é centrado no desenvolvimento da aplicação e não nos detalhes de tecnologia. A definição de mecanismos de transformação permite os modelos descritos em linguagem de alto nível sejam traduzidos, em um ou mais passos, em código executável, o que tende a facilitar o reuso e a manutenção das aplicações. Tais

vantagens têm alavancado uma crescente adoção da MDA como metodologia de desenvolvimento em diversos domínios de aplicação nos últimos anos.

No desenvolvimento de software orientado a modelos, uma questão relevante consiste em garantir que as regras de negócio, ou propriedades específicas da aplicação, sejam atendidas pela implementação gerada. De modo a assegurar essa funcionalidade, é comum aos desenvolvedores inserir código diretamente no modelo específico de plataforma (PSM), o que pode ser tedioso e sujeito a erros. Assim, a falta de mecanismos facilitadores que permitam a garantia de atendimento às regras de negócio do domínio de aplicação frente à implementação correspondente torna-se um empecilho importante para a construção de aplicações confiáveis e de fácil reuso e manutenção.

Confiabilidade e segurança no funcionamento (*dependability*) são características essenciais para aplicações críticas existentes na indústria. Estas possuem requisitos estritos de funcionamento e, portanto, requerem ferramentas que garantam a validação do software a ser utilizado antes de sua utilização em ambientes reais. Dessa forma, tais requisitos ainda são entraves a serem vencidos para adoção mais efetiva de técnicas modernas de engenharia de software, a exemplo da MDA, em ambientes industriais.

Este trabalho apresenta uma abordagem que separa as regras de negócio do modelo específico de plataforma (PSM) e permite sua inserção diretamente no código da implementação. Nessa abordagem, as regras de negócio são escritas na linguagem OWL (*Ontology Web Language*) [W3C 2008] e transformadas em aspectos na linguagem Java. Estes aspectos são, por sua vez, combinados e compilados para gerar o código final da aplicação. Para efetuar a validação da abordagem e da ferramenta correspondente (*OWLtoAspectJ*), realizamos a reestruturação da modelagem e programação de uma aplicação industrial de controle de fornos que possui requisitos críticos de funcionamento.

Este artigo está estruturado em seis seções. A segunda seção fornece uma breve introdução sobre a abordagem MDA, a linguagem OWL e a representação das regras de negócio nessa linguagem, bem como, uma introdução sobre a abordagem de orientação a aspectos. A seção 3 fornece uma visão geral da nossa abordagem, ao passo que, a seção seguinte descreve a modelagem e a implementação da ferramenta e sua utilização através de um estudo de caso. A seção 5 destaca os trabalhos correlatos e, por fim, a seção 6 conclui o trabalho e apresenta possibilidades futuras.

2. Conceitos básicos

Nessa seção apresentamos os fundamentos da abordagem MDA, linguagem OWL e orientação a aspectos, que fundamentam a concepção desse trabalho.

2.1 Modelos e níveis de abstrações em MDA

A MDA propõe a criação de modelos em diferentes níveis de abstração, separando os interesses de implementação de uma arquitetura específica do modelo conceitual de uma aplicação. A MDA é uma arquitetura em camadas que utiliza quatro níveis de modelos: M3, M2, M1, e M0. Os três primeiros níveis especificam modelos e metamodelos. Estes últimos podem ser vistos como padrões para representar modelos. O último nível (M0) corresponde à camada de gerenciamento e instanciação dos dados/objetos.

Os modelos podem relacionar-se através de mapeamentos realizados de forma manual ou automática. As regras de mapeamento definem a tradução de um ou mais modelos de origem para um modelo de destino. Essas regras são escritas em metamodelos e aplicáveis a todos os modelos de origem que obedeçam às especificações desses metamodelos. Para que seja possível a utilização de mapeamentos em modelos UML é necessário importar o modelo em uma ferramenta MDA através de um arquivo no formato XMI (*XML Metadata Interchange*). O XMI é um artefato gerado a partir das linguagens XML (*Extensible Markup Language*), UML e MOF (*Meta Object Facility*). O XMI é o mecanismo fundamental de interoperabilidade da arquitetura e combina os benefícios da XML para definição, validação e compartilhamento de formatos de documentos com os benefícios da linguagem de modelagem visual UML para especificação, visualização, construção e documentação.

2.2. Ontologias e a linguagem OWL

As características fundamentais do domínio de uma aplicação podem ser adequamente representados por ontologias. A linguagem OWL (*Ontology Web Language*) [W3C 2008] permite expressar a estrutura, relacionamentos e características dos conceitos de um domínio através de axiomas lógicos. A OWL é baseada na lógica de descrições (*Description Logic*), possui estrutura similar a XML e é subdividida em três sub-linguagens diferentes: OWL Lite, OWL-DL e OWL Full. A Tabela 1 apresenta um trecho do arquivo OWL gerado pelo editor de ontologia Protégé [Protégé, 2008]. Este exemplo, relacionado ao contexto da aplicação desenvolvida como estudo de caso, descrito na seção 4, especifica que apenas o operador pode realizar o abastecimento de insumos, conforme os requisitos da aplicação de abastecimento de fornos. As regras de negócio são representadas pelas *tag* owl:Restriction no formato XML. A restrição do domínio é composta por uma propriedade associada a uma classe indicada pela *tag* rdfs:domain rdf:resource. A *tag* rfds:range rdf:resource, por sua vez, determina qual valor deve ser aceito. Neste caso, são apenas permitidas instâncias da classe Operador.

Tabela 1. Regra de negócio para operação de forno industrial em OWL.

```
<owl:Restriction>
  <owl:someValuesFrom>
     <owl:Class rdf:ID="Forno"/>
      </owl:someValuesFrom>
      <owl:onProperty>
  <owl:ObjectProperty rdf:ID="#E_carregado_por_Operador "/>
<owl:Restriction>
<owl:ObjectProperty rdf:about="# E_carregado_por_Operador ">
 <rdfs:domain rdf:resource="#Forno"/>
 <rdfs:range rdf:resource="#Operador"/>
 <owl:inverseOf rdf:resource="#Carrega_Forno"/>
</owl:ObjectProperty>
```

2.3 Programação Orientada a Aspectos

A Programação Orientada a Aspectos ou AOP (*Aspect-Oriented Programming*) [Kiczales, 1996] é uma das tecnologias recentes utilizadas visando a "separação de interesses" e modularização de código. Esta abordagem complementa a programação tradicional, especialmente os modelos orientados a objetos, ao oferecer um conjunto de

técnicas que disassociam interesses transversais em novas abstrações e um mecanismo de composição de aspectos e componentes.

Uma das principais linguagens orientadas a aspectos é *AspectJ* [AspectJ, 2008], que provê extensões de aspectos para Java. As abstrações fundamentais de AspectJ são os aspectos (*aspects*), pontos de junção (*join points*), *pointcuts* e *advices*. Os aspectos são os elementos principais que podem alterar a estrutura estática ou dinâmica de um programa, a exemplo de atributos ou métodos de uma classe. Tais alterações ocorrem em tempo de execução em regiões de código chamadas de pontos de junção (*join points*), a exemplo de chamada de métodos. Os pontos de junção escolhidos são descritos por *pointcuts*, que permitem variadas formas de seleção (eg, métodos com parâmetros específicos ou operadores lógicos). As ações associadas quando da interceptação dos pontos de combinação são definidas em blocos chamados de *advices*. Nestes, é possível definir se o comportamento é executado antes, durante ou após o método (ou *pointcut*), por exemplo. O código da aplicação Java é combinado com o código com aspectos por um processo de compilação/combinação conhecido por *weaving*. De forma geral, a AOP tem apresentado bons resultados na modularização e especialização de código em diversos contextos. O interesse na utilização de aspectos neste trabalho reside na facilidade e modularidade obtida na introdução das regras de domínio juntamente ao código final da aplicação gerada.

3. Visão Geral do Processo de Desenvolvimento

A Figura 2 apresenta o processo de desenvolvimento proposto neste trabalho. Em primeiro lugar, é criado o domínio conceitual da aplicação com base em uma ontologia de domínio e os requisitos da aplicação. Em seguida, essa ontologia é transformada em um modelo independente de plataforma (PIM) usando a ferramenta de edição de ontologias Protégé para a linguagem UML. Na sua concepção original, a MDA prima por definir as regras de negócio nos modelos específicos de plataforma, ocasionando alguns problemas relatados na seção 1. Neste trabalho, as regras de negócio são definidas e verificadas no modelo conceitual, antes da geração do modelo PIM e são transformadas em um modelo PSM com o auxílio de uma ferramenta construída para este propósito, a *OWLtoAspectJ*. Podem ser gerados diversos modelos PSMs, sendo: um representando as regras de negócio e os outros para plataformas específicas (*e.g.* Java SE, Java EE, RMI, Corba). Por fim, a aplicação final é gerada a partir da combinação do modelo de regras de negócio com o PSM da aplicação utilizando uma linguagem orientada à aspectos, a exemplo da AspectJ [Kiczales,1996].

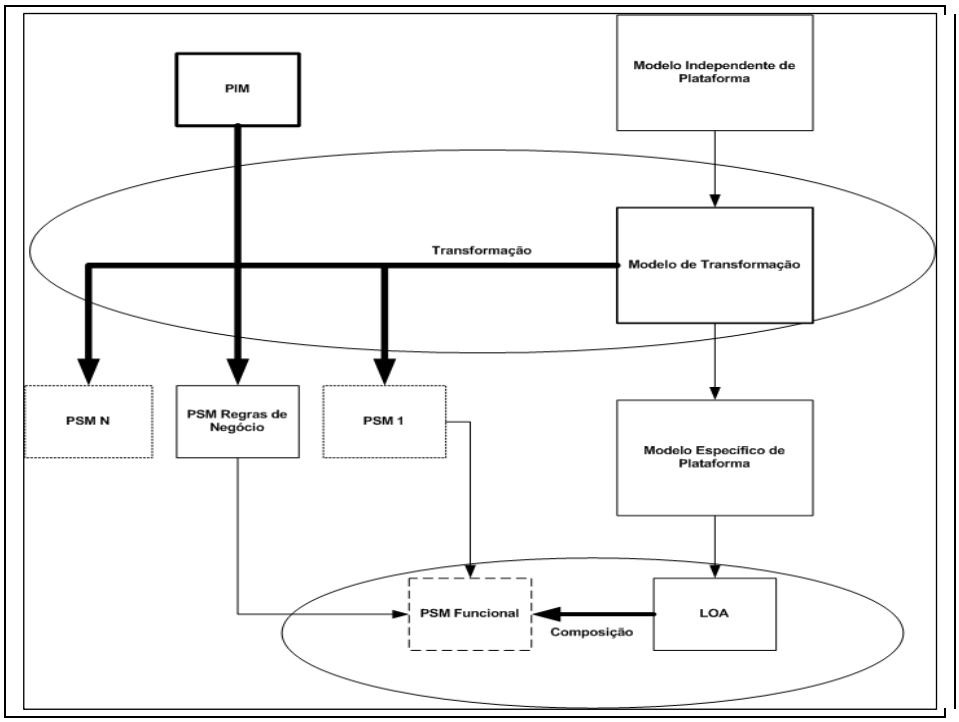

Figura 2. Visão geral do Processo de Desenvolvimento.

A Figura 3 apresenta com maior nível de detalhe o procedimento e as ferramentas utilizadas na construção de modelos MDA através de ontologias. Em primeiro lugar, o modelo conceitual da ontologia é definido na ferramenta Protégé que produz uma descrição OWL da ontologia. Em seguida, a ontologia é verificada na ferramenta Racer (Santos *et al*, 2005). Uma vez atendidas as propriedades desejadas da ontologia, a ferramenta Poseidon (POSEIDON, 2006) importa o arquivo XMI produzido pelo Protégé e cria um modelo da aplicação no nível M1. Este modelo PIM é finalmente traduzido para o modelo PSM com o uso da ferramenta OptimalJ (OPTIMALJ, 2006). O trabalho descrito neste artigo utiliza este arcabouço de ferramentas com um adicional: a utilização de orientação a aspectos para a inserção de código referente às regras de negócio diretamente na aplicação.

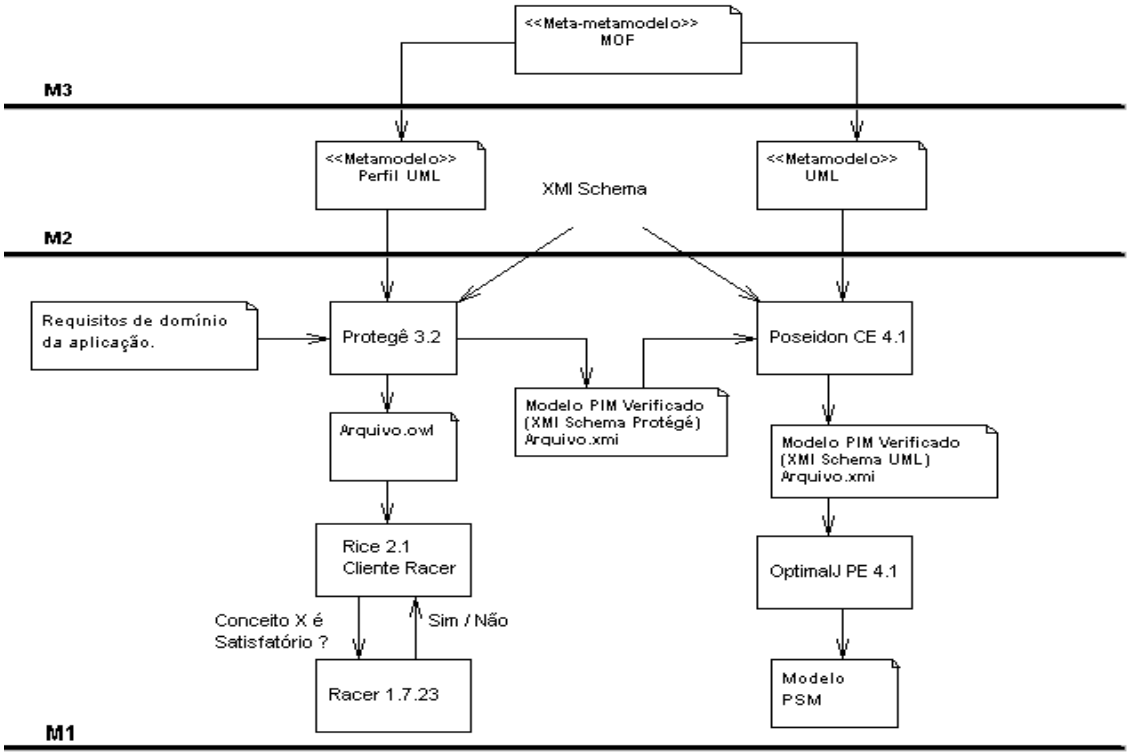

Figura 3. Visão geral da construção de modelos a partir de uma ontologia.

4. Estudo de Caso : Gerenciamento de Fornos Industriais

Para validação da abordagem, foi utilizada uma aplicação para supervisão e controle de abastecimento de fornos industriais para fabricação de ligas de ferro. A seguir apresentamos uma breve descrição dos requisitos dessa aplicação e sua modelagem com o uso da MDA.

O processo de fabricação de uma liga de ferro é definido pelo analista de planejamento de produção. Este é constituído por uma receita composta de insumos. Esta receita é destinada ao operador do forno que efetua o abastecimento deste com a quantidade de insumos especificada. Cada forno dispõe de um controle de temperatura e sensores que indicam a posição e quantidade dos insumos. Os sensores enviam os dados para um Controlador Lógico Programável (CLP), o qual é gerenciado pelo sistema de produção aqui descrito.

Por questões de brevidade, apresentaremos apenas os itens mais importantes referentes à modelagem do domínio. Após a análise de requisitos, definimos o diagrama de casos de uso apresentado na Figura 4. Neste diagrama observamos que os tipos de usuários são operadores e administrador. O administrador é responsável pelas operações de cadastro de insumos ao passo que o operador é o único usuário apto a realizar o abastecimento dos fornos.

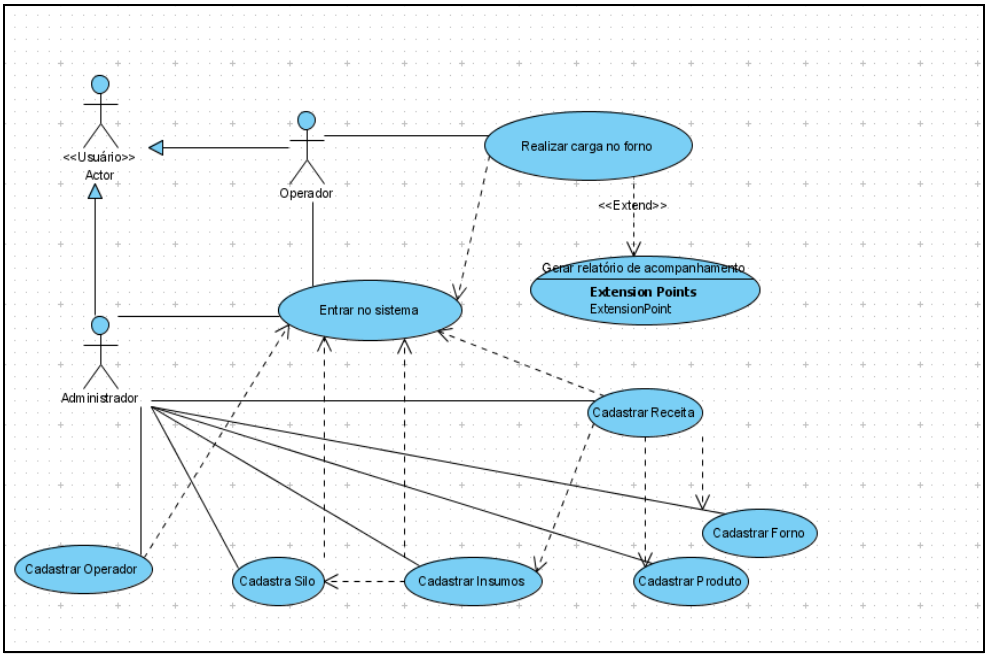

Figura 4. Diagrama de Caso de Uso do Sistema de Abastecimento de Fornos.

A partir das informações contidas nas descrições dos casos de uso e documento de requisitos, foi construído o domínio conceitual da aplicação, representado através da linguagem UML do diagrama da Figura 5. Neste diagrama, observa-se os usuários e componentes (conceitos) principais: forno, produto, receita, insumo e silo (local de armazenamento de insumos). Vale ressaltar que o diagrama da Figura 5 é apenas ilustrativo. De fato, para construção da aplicação, o domínio da aplicação foi modelado através de uma ontologia que, em seguida, foi transformada automaticamente em UML.

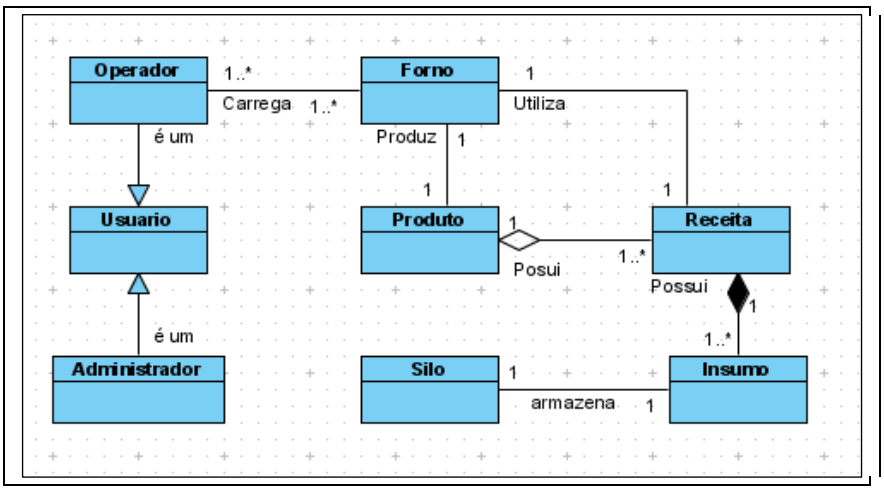

Figura 5. Domínio Conceitual da Aplicação de Fornos Industriais.

4.1 Concepção da Ontologia

A ontologia concebida é formada por conceitos, propriedades e axiomas. Os conceitos são representados por classes referentes ao modelo de domínio. As propriedades são

atributos associados aos conceitos. Por fim, os axiomas são as regras de negócio encontradas nos casos de uso. A construção da ontologia fornece a capacidade de criação e verificação das regras de negócio no nível do modelo independente de plataforma. Todos os três elementos definidos na ontologia da aplicação foram especificados em OWL. As regras de negócio da ontologia foram descritas na linguagem axiomática PAL (*Protégé Axiomatic Language*), que permite a inserção de restrições e axiomas sobre as classes e instâncias de uma ontologia. Durante a construção das restrições são feitas associações das propriedades e a faixa de valores permitida. A Tabela 2 apresenta algumas das regras de negócio da aplicação escritas em PAL.

Tabela 2. Regras de negócio transcritas para a Protégé Axiomatic Language.

Regras de Negócio	PAL
1- Um forno produz um tipo de produto.	Possui_um_Forno **exactly** 1
2- Durante a carga do forno, este deve somente aceitar um insumo que seja parte da receita especificada.	Produz_um_Produto **exactly** 1 Possui_uma_Receita **exactly** 1 Possui_uma_Receita **only** Receita
3 – O forno é carregado apenas pelo operador.	E_Carregado_por_Operador **only** Operador Not (Carrega_Forno some Forno)
4 – Uma receita possui ao menos 1 insumo.	Possui_Insumos **some** Insumo

A verificação da corretude dos conceitos é realizada pelo mecanismo de raciocínio (*reasoner*) da ferramenta Racer (Haarslev *et al*, 2004). Este mecanismo é baseado no cálculo SHIQ (Horrocks *et al*, 1999), um método de tableaux para Lógica de Descrição. A lógica SHIQ é composta por diversos elementos: conceito atômico, conceito universal, conceito base, negação atômica, interseção de conceitos, quantificação universal, quantificação existencial limitada, negação, restrição de números, regras hierárquicas, regras de inversão e regras de transitividade. Estes elementos facilitam a especificação de propriedades associadas aos conceitos.

4.2 Transformação de Modelos

Após a verificação do domínio conceitual, a transformação dos modelos ocorre em dois passos: (1) a geração de um modelo independente de plataforma para a UML com base na ontologia de domínio e (2) a criação de modelos PSMs para a aplicação, a partir do modelo UML, e para as regras de negócio através do domínio conceitual especificado em OWL.

O primeiro passo é realizado com o uso do *plugin* UML Backend (Protégé, 2008), a qual produz uma instância do metamodelo MOF. Nesse sentido, diversas ferramentas de tradução podem ser utilizadas. Para a geração do modelo PSM referente às regras de negócio, foi construída uma ferramenta implementada na linguagem Java (Sun, 2008) e a API Jena (Jena, 2008). Os modelos PIM e PSM da aplicação foram gerados a partir do modelo conceitual automaticamente.

A Figura 6 mostra um trecho da interface da ferramenta *OWLtoAspectJ* que efetua a transformação de regras do modelo conceitual em aspectos. O desenvolvedor

especifica o arquivo OWL contendo a ontologia do domínio (entrada) e define a pasta onde será gerado o arquivo do modelo PSM com os aspectos inclusos (saída).

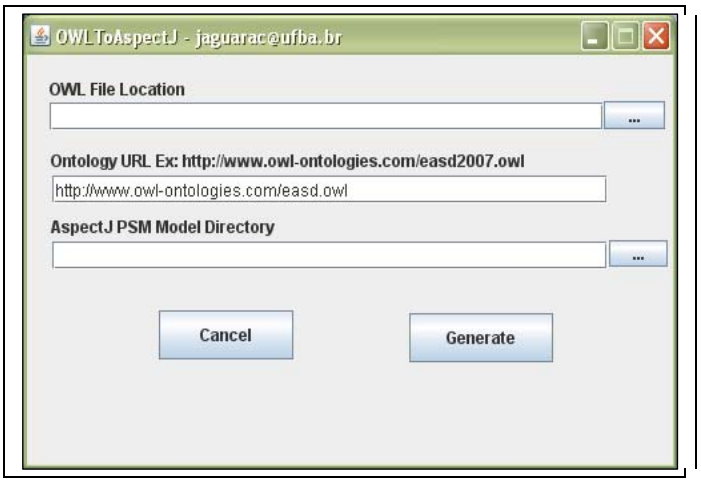

Figura 6. Trecho da Interface de Transformação da ontologia em OWL (PIM) para AspectJ (PSM).

4.3 Transformação de OWL para AspectJ

Para a transformação do modelo PIM no formato OWL para o PSM em *AspectJ*, primeiro é realizada uma leitura de cada restrição e de acordo com os tipos suportados pela linguagem axiomática PAL é feita uma separação. Em seguida, são associados os elementos de cada linguagem usando a API Jena para capturar os dados dos elementos em OWL utilizados para a criação dos aspectos.

Para a criação de um aspecto com a capacidade de implementar as regras de negócio da aplicação foram estabelecidos *advices* e *pointcuts* específicos. No nome do *pointcut* foi indicado o conceito ou classe cujo o aspecto deve agir, o que é representado em OWL pela *tag* rdfs:domain rdf:resource, seguido da informação do conceito relacionado obtido com a informação da *tag* rdfs:range rdf:resource.

Como o modelo PSM para os aspectos deve ser adaptado automaticamente, o método a ser interceptado deve ter o nome da classe relacionada ao conceito em sua assinatura (*e.g.* Operador.carregaForno(), Receita.insereInsumo()). Assim, utilizamos um designador (*e.g.* execution, call, Get, Set) para o *pointcut* com o nome da classe associada de modo a evitar a necessidade de escrita da assinatura completa do método. Na ontologia, a *tag* owl:ObjectProperty rdf:ID representa o nome da regra de negócio, a qual serviu para nomear o aspecto, facilitando a identificação de artefatos. As regras que fazem referência às restrições por tipos de objetos (*e.g.* owl:allValuesFrom, owl:hasValuesFrom e owl:someValuesfrom) utilizam o operador instanceof na composição da estrutura condicional, enquanto que as restrições de cardinalidade são implementadas com o uso dos operadores lógicos e aritméticos convencionais da linguagem Java (*e.g.* >=, <= e ==) .

De acordo com uma das regras destacadas na Tabela 2, um administrador não pode realizar o abastecimento em fornos. A Listagem 1, mostra a implementação da regra de negócio E_carregado_por_Operador codificada em *AspectJ* pelo aspecto

FornoE_carregado_por_Operador. Esse aspecto insere um ponto de verificação da classe do usuário antes da execução dos métodos associados ao forno.

Listagem 1. Regra de Negócio Transformada em código AspectJ.

```
package businessRules;
import abastecimento.Operador;
 public aspect FornoE_carregado_por_Operador {
            public pointcut Operador() :
                        call(* *.*Forno(..)) &&
                        (!(call(* *.getForno(*)) || call(* *.setForno(*))));
            before():Operador() {
                        Object obj = thisJoinPoint.getTarget();
                        If (!(obj instanceof Operador)){
                        System.out.print("\n Exception Rule:
                          FornoE_carregado_por_Operador");
                        }
            }
}
```

4.4 Verificação da Implementação

A fim de possibilitar a verificação das regras de negócio, foram implementadas as classes Java, utilizando dois modelos PSM. O primeiro para a criação dos conceitos, atributos e suas relações e o segundo gerado automaticamente através ferramenta *OWLtoAspectJ* representando as regras de negócio.

Para validar a verificação das regras de negócio frente à implementação existente, efetuamos uma implementação simplificada da aplicação e analisamos sua execução. O código de teste utilizado, escrito manualmente, está descrito na Listagem 2. Neste código foram criadas as instâncias para realizar as ações descritas nos caso de uso do sistema: o cadastramento de insumos, silos, receitas de produtos contendo os insumos necessários para sua fabricação, os produtos a serem fabricados nos fornos, as fábricas e seus respectivos fornos e os usuários do sistema.

Listagem 2. Exemplo de um Conceito Implementado em Java.

```
package abastecimento;
public class Administrador extends Usuario{
  public void carregarForno(Forno forno, Insumo insumo) {
            forno.carregadoPorOperador(insumo);
            System.out.print("\n insumo adicionado no forno da Fábrica");
  }
}
```

O trecho de código de teste está descrito na Listagem 3. Neste código são criadas instâncias dos conceitos fábrica, forno, administrador e, ao final, é feita uma tentativa de abastecimento de um forno por um administrador do sistema, o que caracterizaria um exemplo de violação às regras de negócio estabelecidas.

Listagem 3. Trecho do Código Utilizado para Teste das Regras de Negócio.

```
Fabrica fabrica = new Fabrica();
fabrica.cadastrarForno(forno1);
Operador op = new Operador();
op.setMatricula(0);op.setNome("N");op.setSenha("0");
operador.carregarForno(forno1, insumo);
Administrador adm = new Administrador();
adm.setMatricula(0); adm.setNome("N"); adm.setSenha("0");
adm.carregarForno(forno1, insumo);
```

A Figura 7 apresenta a execução do código da Listagem 3. Assim como previsto, a tentativa de execução do abastecimento do forno por um administrador provoca uma exceção no código da aplicação que informa qual regra de negócio foi violada (Exception Rule). Apesar das atuais limitações quanto à verificação estática das regras de negócio, tal ferramenta mostra-se extremamente útil na fase de validação e testes do sistema, pois permite a identificação de inconsistências da implementação frente às propriedades do domínio antes da entrada do sistema em operação.

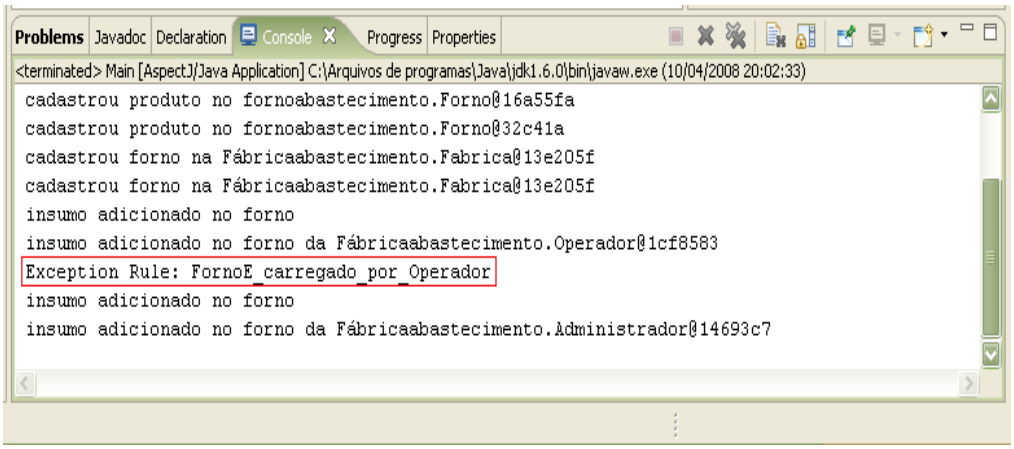

Figura 7. Validação da Regras de Negócio em Tempo de Execução.

5. Trabalhos relacionados

Alguns trabalhos vistos na literatura utilizam as aproximações entre MDA e ontologias [Djuric 2004] ou MDA e AOP (*Aspect-Oriented Programming*) [Dhondt *et al*, 2008]. Tais projetos preocupam-se em utilizar as definições de metamodelos para a criação de um perfil UML para formar a ontologia. Assim, uma ontologia pode ser descrita por classes, atributos e propriedades, com o objetivo de facilitar a sua transformação ou utilização junto aos outros perfis e\ou modelos UML. Dhondt [Dhondt *et al*, 2008] são pioneiros quando se trata da utilização de aspectos e representação de conhecimento. Em um dos seus trabalhos, Dhondt [Dhondt *et al*, 2006] utiliza a programação orientada a aspectos para implementar as regras de negócio em forma de aspectos a partir de modelos conceituais de alto nível.

Trabalhos recentes destacam o uso da técnica de programação orientada a aspectos e MDA. Braga *et al* [Braga *et al*, 2007] propuseram um framework baseado em aspectos no contexto de linhas de produto de software com o intuito de capturar

requisitos funcionais. Este trabalho, contudo, não lida especificamente com o uso de aspectos em procedimentos de verificação/validação. Alves *et al* [Alves *et al*, 2007], por sua vez, utilizam aspectos para nortear as atividades de transformação de modelos.

6. Conclusão

Apesar da sua crescente adoção como metodologia de desenvolvimento e da proliferação de ferramentas existentes, os sistemas baseados em MDA ainda demandam mecanismos adicionais de validação e verificação.

Este trabalho descreveu a implementação e avaliação de uma proposta que define propriedades do domínio através de regras de negócio descritas por meio de uma ontologia e, posteriormente, efetua a fusão das rotinas de verificação dessas propriedades com o uso da tecnologia de aspectos. Esta abordagem visa primordialmente separar as regras de negócio dos modelos específicos de plataforma. Nos experimentos realizados com uma aplicação real de controle de fornos industriais, verificamos que essa abordagem simplifica o desenvolvimento, pois dispensa a escrita de código para as regras de negócio nos métodos das classes envolvidas já que tal funcionalidade é gerenciada pelos aspectos. Além disso, as regras geradas usando a linguagem AspectJ podem ser reutilizadas em outras plataformas, a exemplo de CORBA e RMI.

Espera-se, em trabalhos futuros, avaliar a concorrência entre aspectos, sob o ponto de vista de sobreposição das regras de negócio, e aprimorar a ferramenta *OWLtoAspectJ* visando a composição dinâmica de outros modelos para plataforma Java e no contexto de Web Services. Além disso, seria importante avaliar abordagens complementares para verificação estática do modelo conceitual antes da instalação ou execução da implementação resultante.

Referências

Alves, M. P., Pires, P. F., Delicato, F. C., Campos, M. L. M. (2007). "CrossMDA: Arcabouço para integração de interesses transversais no desenvolvimento orientado a modelos". Simpósio Brasileiro de Componentes, Arquiteturas e Reutilização de Software (SBCARS), 2007.

AspectJ (2008). "AspectJ Java Aspect-Oriented Programming". http://www.eclipse.org/ajdt/. Acesso: Abril/2008.

Booch, G., Jacobson, I., Rumbaugh, J. (1999). "Unified Modeling Language – User's Guide". Addison-Wesley, 1999.

Braga, R. T. V, Germano, F. S. R., Pacios, S. F., Masiero, P. C. (2007). "AIPLE-IS An Approach to Develop Product Lines for Information Systems Using Aspects". Simpósio Brasileiro de Componentes, Arquiteturas e Reutilização de Software (SBCARS), 2007.

Dhondt, M., Streten, R. V. D. "Programme Inria Equipes Associées". http://jacquard.lifl.fr/CALA/CALA2007.html. Acesso: abril/2008.

Dhondt, M. Cibrán, M. A. (2006) "A Slice of MDE with AOP: Transforming High-Level Business Rules to Aspects". ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems (Models), 2006, Genova, Itália.

Djuric, D. 2004 “MDA-Based Ontology Infrastructure”. International Journal on Computer Science and Information Systems, Vol. 1, No. 1, 2004, pp. 91-116.

Freitas, F.. 2003 “Ontologias e Web Semântica”. IV ENIA – Encontro Nacional de Inteligência Artificial, Campinas, Minicurso. Anais do XXIII Congresso da Sociedade Brasileira de Computação, 2003, 52 p.

Guizzard, Giancarlo. (2000) “Uma abordagem metodológica de desenvolvimento para e com reuso, baseada em ontologias formais de domínio”. Dissertação de Mestrado. Universidade Federal do Espírito Santo, Vitória, Julho, 2000.

Haarslev, V., Muller, R.. (2004) “Racer’s User Guide and Reference Manual”. Versão 1.7.19.

Han, Y., Kniesel, G., Cremers, A.B. (2005). “Towards Visual AspectJ by a Meta Model and Modeling Notation”. 6th Aspect-Oriented Software Development Conference (AOSD), 2005.

Horrocks, I., Sattler, U., Tobies, S. (2000). “Reasoning with Individuals for the Description Logics SHIQ”. Proceedings of the 17th Int. Conf. on Automated Deduction (CADE), volume 1831 of Lecture Notes in Computer Science, pages 482-496. Springer, 2000.

Jena. “A Semantic Web Framework for Java”. http://jena.sourceforge.net/. Acesso em abril/2008.

Kiczales, G. (1996). “Aspect-oriented programming”. ACM Computing Surveys, 28A(4), 1996.

Laddad, R. (2003). “AspectJ in Action: Practical Aspect-Oriented Programming”. Manning, 2003. 512 p.

Mellor, J. Stephen et al (2003). “MDA Destilada: Princípios da Arquitetura Orientada por Modelos”. Ed. Ciência Moderna Ltda., p. 15-169, 2003.

OMG (2008). MDA Guide version 1.0.1. Formal Document: 03-06-01. Disponível em: http://www.omg.org/cgi-bin/apps/doc?omg/03-06-01.pdf. Acesso em: maio/2008.

OptimalJ.“Model-Driven Development for Java”. http://www.compuware.com/products/optimalj/, Julho, 2006.

Noy, N., Fergerson, R., Musen., M. 2000. “The knowledge model of Protégé-2000: Combining interoperability and flexibility”. 2nd International Conference on Knowledge Engineering and Knowledge Management (EKAW), Juan-les-Pins, France, 2000.

Poseidon. “Poseidon for UML 4.2”. http://gentleware.com/index.php, Julho, 2006.

Protégé. “Ontology Editor and knowledge-base framework”. http://protege.stanford.edu/. Acesso: Abril/2008.

Santos, D. A., Vieira, Renata, Santana, M. R., Silva, D. M.. (2005) “Web Semântica: Ontologias, Lógica de Descrições e Inferências”. WebMídia 2005. Porto Alegre: SBC, v. 1, p. 127-165, 2005.

Silva, J.B., Pezzin J. (2007). “The formal verification of an application conceptual model using MDA and OWL”. World Congress on Engineering and Computer Science (WCECS), San Francisco, 2007.

Sun (2008). Sun Microsystems (2008). “Java Technology”. http://www.java.sun.com. Abril, 2008.

W3C. World Wide Web Consortium (2008). “OWL Web Ontology Language Guide”. http://www.w3.org/TR/owl-guide/. Acesso :Abril/2008.