Alexandre dos Santos Mignon

Aplicação da Técnica de Tecelagem de Modelos na Transformação de Modelos na MDA

Dissertação apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do Título de Mestre em Engenharia.

São Paulo
2007

Alexandre dos Santos Mignon

Aplicação da Técnica de Tecelagem de Modelos na Transformação de Modelos na MDA

Dissertação apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do Título de Mestre em Engenharia.

Área de concentração:
Sistemas Digitais

Orientador:
Prof. Dr. Paulo Sérgio Muniz Silva

São Paulo
2007

Agradecimentos

Ao Prof. Dr. Paulo Sérgio Muniz Silva, pelo excelente trabalho de orientação, pelo apoio, dedicação e estímulo prestados ao longo deste período e pela oportunidade única de aprendizagem proporcionada.

Ao Prof. Dr. Ricardo Luis de Azevedo da Rocha e ao Prof. Dr. Jorge Luis Risco Becerra, pela grande contribuição no exame de qualificação.

Ao meu pai Dorlei, à minha mãe Emilia, à minha avó Tereza e aos meus irmãos Camila e Diogo, pelo amor, carinho e apoio ao longo deste período.

A minha madrinha Rosangela, pelo incentivo e pelo grande exemplo de dedicação à carreira acadêmica.

E a todas as outras pessoas que, de forma direta ou indireta, me ajudaram na elaboração deste trabalho.

Resumo

Uma das principais atividades dos enfoques de desenvolvimento de software centrados em modelos, como por exemplo a Arquitetura Dirigida por Modelo (*Model Driven Architecture* - MDA), é o processo de transformação de modelos. Geralmente, um passo preliminar para a transformação dos modelos é o mapeamento dos elementos do meta-modelo fonte nos elementos do meta-modelo alvo. Este trabalho apresenta uma aplicação de uma técnica de mapeamento de modelos denominada tecelagem de modelos. Esta técnica permite ao usuário definir a semântica das ligações estabelecidas entre os elementos do meta-modelo fonte e os elementos do meta-modelo alvo. A semântica é definida através de tipos fortes associados às ligações. O presente trabalho analisa, através de dois experimentos, alguns aspectos da geração de modelos de transformação de modelos no arcabouço MDA, utilizando a técnica de tecelagem de modelos. A análise utiliza duas alternativas de especificação de transformação de modelos a título de comparação: a que usa somente uma linguagem de especificação de modelos de transformação e a que usa a técnica de tecelagem de modelos. Os aspectos analisados são: a reutilização de trechos de código escritos na linguagem de geração de especificações de transformação e a reutilização de decisões de projeto (*design*) no mapeamento entre dois meta-modelos distintos.

Palavras-chave: MDA. Transformação de modelos. Mapeamento de modelos. Tecelagem de modelos.

Abstract

One of the main activities of the model-centric approaches of software development, as for example the Model Driven Architecture (MDA), is the process of model transformation. Usually, a preliminary step for model transformation is the mapping of source metamodel elements into target metamodel elements. This work presents an application of a technique for model mapping called model weaving. This technique allows the user to define the semantics of the links binding source metamodel elements and target metamodel elements. The semantics is defined through types associated to links. The work analyzes, through two experiments, some aspects of the generation of models transformation in the MDA framework, using a technique known as model weaving. The analysis, for comparison purposes, uses two techniques of model transformation specification: one using only a specification language for model transformation specification and another using model weaving. The analyzed aspects are: the reuse of pieces of code written in the language that generates the transformation of specifications and the reuse of design decisions in the mapping between two distinct metamodels.

Keywords: MDA. Model transformation. Model mapping. Model weaving.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

API Application Program Interface

AMMA ATLAS Model Management Architecture

AMW ATLAS Model Weaving

ATL ATLAS Transformation Language

ATL VM ATL Virtual Machine

CIM Computation Independent Model

DSL Domain Specific Language

HOT High Order Transformation

ISM Implementation Specific Model

KM3 Kernel MetaMetaModel

MDA Model Driven Architecture

MDD Model Driven Development

MOF Meta Object Facility

OMG Object Management Group

OCL Object Constraint Language

PIM Plataform Independent Model

PSM Plataform Specific Model

QVT Query Views Transformations

UML Unified Modeling Language

XML eXtensible Markup Language

XMI XML Metadata Interchange

XSLT eXtensible Stylesheet Language for Transformation

1 Introdução

1.1 Motivação

Em 2001, o *Object Management Group (OMG)* apresentou a MDA (*Model Driven Architecture* - Arquitetura Dirigida a Modelo), um enfoque centrado em modelos para o desenvolvimento de sistemas. Uma das principais características da solução apresentada pela MDA é a total separação entre os aspectos de negócio e os aspectos tecnológicos do sistema que está sendo construído (BROWN, 2004). São criados modelos somente com os elementos relevantes a um determinado ponto de vista, como por exemplo, um modelo contendo apenas elementos relacionados ao domínio do problema.

A MDA apresenta uma nomenclatura uniforme para classificar os diversos tipos de modelos construídos ao longo do ciclo de desenvolvimento do sistema (MILLER; MUKERJI, 2003). O modelo independente de plataforma (*Plataform Independent Model* - PIM) descreve o sistema que está sendo construído sem se preocupar com detalhes específicos de uma determinada plataforma. O modelo específico de plataforma (*Plataform Specifc Model* - PSM) contém detalhes específicos e relevantes para o tipo de plataforma utilizada pelo sistema, como por exemplo, J2EE, CORBA, .NET, etc.

Linguagens específicas de domínio (*Domain-Specific Language* - DSL) podem ser amplamente utilizadas no enfoque de desenvolvimento de software centrado em modelos. As DSLs podem ser utilizadas, por exemplo, para descrever aspectos relacionados a um determinado domínio de negócio ou a uma tecnologia específica (GREENFIELD; SHORT, 2004). A utilização destas linguagens permite a criação de modelos relacionados a um domínio de problema em particular.

Um dos pontos-chave da MDA é a transformação de modelos. Segundo (MILLER; MUKERJI, 2003, p. 2-7) uma transformação de modelos é "um processo de conversão de um modelo em outro modelo do mesmo sistema". Este processo é utilizado, por exemplo, para a geração de um PSM a partir de um PIM. Com

isso, a partir de um único PIM é possível à geração de diversos PSMs, um para cada plataforma específica.

JOUAULT e KURTEV (2005) afirmam que uma direção para fornecer um suporte à transformação de modelos é o desenvolvimento de linguagens específicas de domínio para auxiliar em tarefas comuns de transformação de modelos. Baseadas neste enfoque, diversas linguagens de transformação de modelos têm sido propostas (CZARNECKI; HELSEN, 2003). Estas linguagens permitem que sejam criados programas de transformação de modelos. Cada programa de transformação pode ser também considerado um modelo, que deve estar em conformidade com o meta-modelo da linguagem de transformação (JOUAULT; KURTEV, 2005).

A tecelagem de modelos (*model weaving*) é um outro tipo de operação que pode ser executada sobre modelos para auxiliar o processo de transformação de modelos (FABRO et al., 2005). A realização final da transformação é apoiada por uma ferramenta de transformação baseada em uma linguagem de transformação, por exemplo a ATL (*ATLAS Transformation Language*) (JOUAULT; KURTEV, 2005). Os modelos são tecidos (*weaved*) através de ligações entre os elementos dos modelos fonte e alvo. A principal diferença entre a tecelagem de modelos e as linguagens de transformação de modelos, é o fato de que os meta-modelos das linguagens de transformação de modelos têm uma semântica fixa para permitir que estas linguagens sejam implementadas em motores de transformação adequados às linguagens, enquanto os meta-modelos de tecelagem têm uma semântica definida pelo usuário. Neste caso, reconhece-se a questão de que cada domínio de aplicação deva ter um meta-modelo específico, o qual, potencialmente, capta mais acuradamente propriedades particulares do domínio em questão. Assim, a definição da semântica deste meta-modelo específico pelo usuário facilita a tarefa da modelagem em virtude do aumento de expressividade proporcionado pela especificidade do meta-modelo. Um outro aspecto importante é que as transformações baseadas em semântica fixa tendem a ser dependentes das ferramentas de transformação (FABRO et al., 2005). Há, portanto, uma motivação suplementar para a utilização de técnicas que tenham uma maior independência em relação a tais ferramentas. A tecelagem de modelos tem como um dos objetivos a independência em relação a ferramentas de transformação.

Na tecelagem de modelos, o usuário pode definir a semântica do meta-modelo de tecelagem através da criação de tipos que serão associados às ligações entre os elementos dos modelos tecidos. A definição destes tipos permite uma melhor adequação deles a um domínio de problema em particular, ou a uma certa

decisão de projeto (*design*). Além disso, as ligações não pressupõem qualquer direção, permitindo uma transformação bi-direcional e economizando o custo da construção de uma transformação para cada direção. As ligações entre dois elementos de modelos diferentes podem ser armazenadas, juntamente com a decisão de projeto (*design*) que o tipo da ligação representa. Esta decisão pode ser então rastreada ao longo do projeto ou mesmo reutilizada, de forma manual, em outros mapeamentos de modelos relacionados ao domínio do problema utilizado.

1.2 Objetivos

O presente trabalho tem como objetivo principal estudar, através de dois experimentos, alguns aspectos da geração de modelos de transformação de modelos no arcabouço MDA utilizando a técnica de tecelagem de modelos. O estudo utiliza duas técnicas de especificação de transformação de modelos a título de comparação: a que usa somente uma linguagem de especificação de modelos de transformação de modelos e a que usa a técnica de tecelagem de modelos. Os aspectos investigados são:

- A reutilização de trechos de código manualmente escritos na linguagem de geração de especificações de transformação. Analisa-se o aspecto da redução da duplicação de código escrito manualmente, fenômeno presente quando se gera especificações de transformação de modelos utilizando apenas linguagens de transformação de modelos. Compara-se aqui, do ponto de vista quantitativo, as duas técnicas de especificação de modelos de transformação mencionadas;

- A reutilização de decisões de projeto (*design*) no mapeamento entre dois meta-modelos distintos. A literatura relata dificuldades neste quesito, sobretudo pela dependência da semântica dos meta-modelos fonte e alvo com a linguagem de transformação e, conseqüentemente, com a ferramenta de transformação. Analisam-se aqui as possibilidades de reutilização de decisões de projeto (*design*) dos tipos das ligações do mapeamento entre os meta-modelos proporcionadas pelas duas técnicas mencionadas.

No presente trabalho, as análises mencionadas estarão restritas somente a tipos de ligação com semântica de igualdade, significando que os elementos vinculados representam a mesma informação. Os experimentos consideram apenas a semântica de igualdade de classes. Além disso, consideram-se apenas transformações unidirecionais.

Adicionalmente, um guia de utilização da técnica de tecelagem de modelos é apresentado ao longo do trabalho, uma vez que a literatura disponível até o momento não apresenta um roteiro didático para a aplicação da técnica em um problema mais elaborado. A comparação entre as duas técnicas de geração de especificação de modelos de transformação é feita através de um estudo de caso, sobre o qual realizam-se dois experimentos. O contexto utilizado nos experimentos é a transformação de um modelo PIM, associado a um domínio genérico de ordens de produção, para um modelo PSM. A especificação deste domínio exigiu a definição de uma DSL apropriada, a qual é descrita em detalhes no presente trabalho.

O primeiro experimento efetua a transformação do modelo PIM para um modelo PSM relativo a uma plataforma de banco de dados relacional. Este experimento tem por objetivo analisar as vantagens da técnica de tecelagem de modelos em relação à reutilização de trechos de código para a geração de especificações de transformação. Para isto, são construídas especificações de transformação do modelo PIM utilizando tanto a técnica de tecelagem de modelos, como somente uma linguagem de transformação de modelos. No caso, a linguagem de transformação utilizada é a ATL (*ATLAS Transformation Language*).

O segundo experimento efetua a transformação do modelo PIM para um modelo PSM relativo à plataforma JAVA. Este experimento tem por objetivo analisar as vantagens da técnica de tecelagem de modelos em relação à reutilização de decisões de projeto (*design*) no mapeamento entre dois meta-modelos diferentes. Para isto, são construídas especificações de transformação do modelo PIM utilizando tanto a técnica de tecelagem de modelos, como somente uma linguagem de transformação de modelos. Como anteriormente, a linguagem de transformação utilizada é a ATL.

A principal contribuição do presente trabalho é uma avaliação prática, embora limitada, das vantagens apresentadas pela técnica de tecelagem de modelos, nas questões de reutilização de trechos de código e de reutilização de decisões de projeto (*design*) no mapeamento entre dois meta-modelos distintos, em relação à técnica que utiliza somente uma linguagem de especificação de modelos de transformação. A avaliação é limitada pois não são definidos e desenvolvidos critérios mais formais para métricas que permitam efetuar medições comparativas nos aspectos de reutilização mencionados. As avaliações resumem-se à observação fatual dos resultados dos experimentos. A literatura disponível até o momento apenas sugere algumas vantagens da técnica de tecelagem de modelos, porém, não apresenta uma avaliação mais concreta destas vantagens.

Outra contribuição é a apresentação, juntamente com a técnica de tecelagem de modelos, de um guia de utilização desta técnica. Do mesmo modo, e como mencionado, a literatura disponível até o momento não apresenta um roteiro didático para a aplicação da técnica de tecelagem de modelos em um problema mais elaborado. Outra contribuição didática, é a criação de uma DSL a partir de um meta-modelo de domínio de problema proposto por HAY (1996). A DSL foi criada, com base no meta-modelo de domínio de ordens de produção proposto pelo autor, para permitir a geração de modelos PIM específicos deste determinado domínio de problema. O presente trabalho adaptou os elementos do modelo proposto pelo autor, definindo-os segundo o mecanismo de perfil UML, o que possibilitou a geração de uma DSL para este domínio de problema.

1.3 Metodologia Utilizada nos Experimentos

Cada experimento é composto de duas etapas: a primeira etapa tem por objetivo especificar a transformação de modelos usando a técnica que utiliza apenas uma linguagem de especificação de modelos de transformação; a segunda etapa tem por objetivo especificar a mesma transformação de modelos, porém, utilizando a técnica de tecelagem de modelos. Os passos utilizados para a realização dos experimentos são:

1. Criação de uma DSL, utilizando o mecanismo de perfil UML, relacionada ao domínio genérico de ordens de trabalho. Esta DSL contém os elementos utilizados para a criação do modelo PIM que será usado nos experimentos. Por questões de compatibilidade com as ferramentas utilizadas no experimento, um fragmento desta DSL foi adaptada para a linguagem KM3, conforme descrito na seção 6.3.

2. Geração de um modelo PIM utilizando a DSL criada. Este modelo é usado como modelo fonte nas transformações de modelos realizadas nos experimentos.

3. Nos dois experimentos propostos, uma especificação de transformação de modelos é gerada utilizando a linguagem ATL (primeira etapa). Cada experimento tem uma especificação de transformação específica. A geração destas especificações tem por objetivo avaliar a técnica que utiliza apenas uma linguagem de especificação de modelos de transformação. Após o término da implementação da especificação de transformação de modelos,

esta é executada, com objetivo de se obter o modelo alvo do processo de transformação.

4. O meta-modelo base de tecelagem de modelos é estendido com o objetivo de se adicionar um tipo de ligação com semântica de igualdade. Este tipo de ligação é utilizado no mapeamento entre os elementos dos meta-modelos fonte e alvo utilizados para a realização dos experimentos. O meta-modelo de tecelagem estendido é utilizado nos dois experimentos propostos. A extensão deste meta-modelo é realizada na linguagem KM3.

5. Nos dois experimentos propostos, na etapa que utiliza a técnica de tecelagem de modelos (segunda etapa), é gerado inicialmente um mapeamento entre os elementos dos meta-modelos fonte e alvo utilizados no processo de transformação. O primeiro experimento realiza o mapeamento entre os elementos da DSL criada no passo 1 e um meta-modelo que contém elementos associados a uma plataforma de banco de dados relacional. O segundo experimento realiza o mapeamento entre os elementos da DSL criada no passo 1 e um meta-modelo que contém elementos associados à linguagem de programação JAVA. Nos dois mapeamentos, o meta-modelo de tecelagem estendido com a semântica de igualdade (passo 4) é utilizado.

6. Geração de uma especificação de transformação de modelo, em linguagem ATL, com o objetivo de implementar a transformação relacionada com a semântica de igualdade do meta-modelo de tecelagem estendido. Este tipo de especificação de transformação é conhecido como transformação de alta ordem (*high order transformation* - HOT). A execução desta especificação de transformação gera como saída um arquivo que contém uma outra especificação de transformação de modelos, novamente em linguagem ATL. Esta última especificação, gerada automaticamente, deve ser idêntica à especificação de transformação gerada no passo 3. Após obter esta última especificação de transformação, ela é executada com objetivo de se obter o modelo alvo do processo de transformação. Novamente, o modelo alvo obtido deve ser idêntico ao obtido no passo 3.

7. Análise dos resultados dos experimentos.

1.4 Estrutura do Trabalho

O presente trabalho está estruturado da seguinte forma. O capítulo 2 apresenta uma visão geral da arquitetura dirigida a modelos (MDA), destacando os

seus componentes básicos. O capítulo 3 apresenta brevemente os padrões adotados pela OMG que dão suporte ao MDA.

O capítulo 4 apresenta de forma mais detalhada o tema de transformação de modelos, apresentando algumas técnicas e linguagens de transformação de modelos. A técnica de tecelagem de modelos é também apresentada neste capítulo.

O capitulo 5 apresenta uma visão geral sobre linguagens específicas de domínio. O objetivo deste capítulo é contextualizar tais linguagens no contexto da MDA e apresentar uma forma de geração deste tipo de linguagem utilizando o mecanismo de extensão da UML denominado perfil UML. Um exemplo simples de criação de um perfil UML é apresentado neste capítulo.

O capítulo 6 apresenta o estudo de caso e os dois experimentos realizados para atingir os objetivos propostos pelo trabalho. Este capítulo apresenta ainda a criação de uma linguagem específica de domínio, associada a um domínio genérico de ordens de produção, utilizando um perfil UML. Esta linguagem é utilizada para a geração dos modelos PIM utilizados nos experimentos.

Por fim, o capítulo 7 apresenta as conclusões e os trabalhos futuros relacionados ao presente trabalho.

2 Arquitetura Dirigida a Modelo - Visão Geral

Este capítulo tem por objetivo apresentar uma visão geral do arcabouço de desenvolvimento Model Driven Architecture (MDA). São apresentados os componentes básicos deste arcabouço, colocando-os no contexto de um desenvolvimento utilizando a MDA.

2.1 O Arcabouço MDA

A MDA é um arcabouço conceitual [1], proposto pela Object Management Group (OMG), com o objetivo de oferecer um conjunto de padrões para dar suporte ao desenvolvimento dirigido a modelo (*Model Driven Development* - MDD) (SELIC, 2003). O principal componente deste arcabouço são os modelos, utilizados como artefato principal para o desenvolvimento do sistema.

O foco do arcabouço MDA está na separação entre os aspectos de negócio e os aspectos técnicos do sistema. São definidos pontos de vista para a modelagem do sistema. Cada ponto de vista tem um interesse particular a ser modelado, seja ele aspectos relativos somente ao negócio ou aspectos técnicos do sistema.

Os modelos, como artefatos principais do desenvolvimento, são responsáveis por captar os conceitos de interesse de um determinado ponto de vista do sistema. Um ponto de vista de um sistema é uma técnica de abstração, através do uso de um conjunto de conceitos arquiteturais e regras estruturais, permitindo que se tenha o foco em interesses particulares de um determinado sistema(MILLER; MUKERJI, 2003). O sentido da abstração aqui utilizado significa a omissão de detalhes não relevantes para um determinado ponto de vista. Por exemplo, a captura de conceitos-chave de um negócio sem a preocupação com aspectos técnicos da implementação do sistema.

[1]Neste contexto, um arcabouço conceitual é um conjunto de conceitos-chave e estruturas que guiam o planejamento, entendimento e um desenvolvimento de um software. (BROWN, 2004)

O arcabouço conceitual do MDA especifica três pontos de vista: (MILLER; MUKERJI, 2003)

- Ponto de Vista Independente de Computação: cujo foco é o ambiente do sistema e os requisitos para o sistema. Detalhes de estrutura e processamento do sistema não são relevantes.
- Ponto de Vista Independente de Plataforma: cujo foco é a operação do sistema, porém detalhes específicos de uma plataforma são omitidos. As especificações deste ponto de vista não devem mudar de uma plataforma para outra.
- Ponto de Vista Específico de Plataforma: adiciona ao ponto de vista independente de plataforma detalhes específicos da plataforma utilizada pelo sistema.

Nos próximos itens são apresentados os conceitos básicos do arcabouço MDA.

2.1.1 Modelo

O artefato principal e o foco de todo o desenvolvimento utilizando o enfoque MDA está centrado em modelos. Um modelo de um sistema é a descrição ou especificação daquele sistema e de seu ambiente (MILLER; MUKERJI, 2003). Os modelos geralmente são formados por uma combinação de desenhos e texto. Geralmente os modelos são construídos utilizando uma linguagem de modelagem, como por exemplo, a Unified Modeling Language (UML) (OMG, 2005c).

Os modelos fornecem uma abstração do sistema que está sendo construído (BROWN, 2004). Os desenvolvedores, através dos modelos, podem raciocinar sobre o sistema, ressaltando aspectos relevantes e/ou críticos em um determinado ponto de vista do sistema, facilitando assim a compreensão de sistemas complexos. Os modelos ainda melhoram a comunicação entre os membros da equipe de desenvolvimento e permitem que sejam feitas previsões sobre a qualidade do sistema.

Na engenharia de software tradicional, os modelos representam um papel secundário (SELIC, 2003). O foco principal é o código do sistema. Os modelos são utilizados apenas para expressar algumas decisões arquiteturais ou de projeto (*design*) para posteriormente serem mapeados em construções de linguagens de programação. Quando isso ocorre, os modelos tornam-se rapidamente desatualizados e acabam perdendo o seu valor.

A figura 2.1 apresenta algumas estratégias de modelagem utilizadas atualmente pelos desenvolvedores de software. Para cada tipo de estratégia foi associado um nível para facilitar a identificação da utilização dos modelos para criar aplicações (código) em uma plataforma específica de execução (*runtime*). A figura 2.1 e a descrição das estratégias foi baseada em (BROWN, 2004).

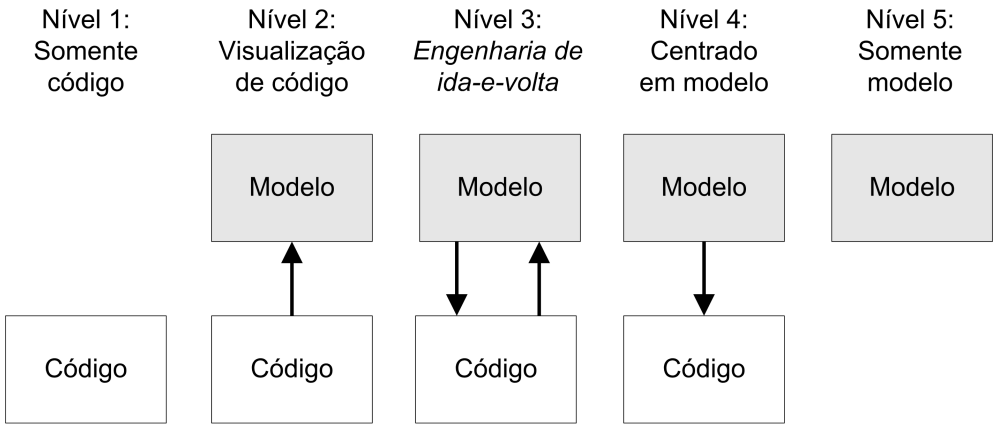

Figura 2.1: Abordagens de modelagem.

Os desenvolvedores, no processo tradicional de software, estão muito centrados na criação do código do sistema e, normalmente, durante o projeto, não geram nenhum tipo de modelo para os auxiliarem na construção do sistema. Na figura 2.1 esta estratégia foi caracterizada como nível 1, em que se tem apenas o código do sistema. O modelo do sistema acaba sendo o próprio código.

Na segunda estratégia apresentada, caracterizada como nível 2, os desenvolvedores geram e utilizam modelos como uma forma de criação e visualização do sistema. Estes modelos são gerados utilizando-se de alguma notação de modelagem. Geralmente estas notações são gráficas.

A terceira estratégia, caracterizada como nível 3, utiliza um conceito denominado engenharia de ida-e-volta (*roundtrip engineering*). Nesta estratégia, o desenvolvedor tipicamente elabora modelos de projeto (*design*) da solução do sistema com um certo nível de detalhe. Posteriormente, aplica uma transformação sobre este modelo, geralmente manual, e gera o código do sistema. Quando uma mudança é feita no código, esta deve ser refletida no modelo original de projeto. Esta estratégia permite que os modelos de projetos gerados não fiquem tão defasados em relação ao código à medida que o projeto avança.

A quarta estratégia é denominada centrada em modelos. Os modelos elaborados têm um nível suficiente de detalhes que permite a geração quase total do código do sistema a partir destes modelos. A implementação do sistema é feita através de sucessivas transformações de refinamento do modelo para o código.

Esta é a estratégia adotada para o arcabouço MDA. Estes modelos devem ter uma sintaxe e uma semântica bem definidas permitindo que sejam interpretados pelo computador.

A abordagem apresentada como nível 5 refere-se somente à utilização de modelos. Nesta, os modelos são utilizados apenas para que os desenvolvedores possam entender o domínio do problema ou analisar um possível arquitetura da solução. Os modelos são utilizados para comunicação, discussão e análise entre os membros da equipe e também entre os interessados no sistema. Este tipo de estratégia auxilia as empresas a manterem um certo controle sobre a arquitetura corporativas dos seus sistemas.

Os modelos gerados no MDA portanto, devem seguir a estratégia de nível 4. Devem ter um detalhamento suficiente que permita a sua interpretação pelo computador. Uma definição mais precisa de modelo é feita por KLEPPE (2003, p. 16) para o qual:

> Um modelo é uma definição de um sistema, ou parte de um sistema, escrito em uma linguagem bem definida.
> Uma linguagem bem definida é uma linguagem com uma forma bem definida (sintaxe), e com um significado (semântica) adequado para a interpretação automatizada pelo computador.

A figura 2.2 apresenta a relação entre o sistema, o modelo e uma linguagem de modelagem (KLEPPE, 2003). Um modelo, portanto, é a descrição de um sistema. Os modelos são escritos em uma linguagem de modelagem, a qual contém todos os elementos que podem ser utilizados pelos modelos. Então, a precisão de um modelo depende de os elementos fornecidos pela linguagem de modelagem terem uma semântica e uma sintaxe precisas e formalmente bem definidas.

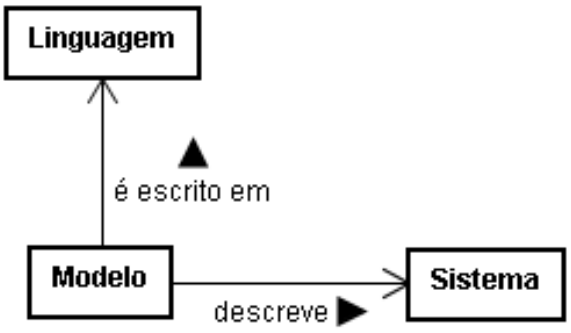

Figura 2.2: Modelos e linguagens.

Conforme apresentado anteriormente, o MDA possui diferentes pontos de vista com diferentes níveis de abstração. Em cada ponto de vista são gerados

modelos contendo elementos relevantes para aquele ponto de vista. Este modelos são então ligados, através do uso de transformações, para formar uma implementação do sistema. (MELLOR, 2004)

As definições que seguem foram resumidas de (MILLER; MUKERJI, 2003).

Uma plataforma é definida em Miller e Mukerji (2003, p. 2-3) como "um conjunto de subsistemas e tecnologias que provê um conjunto coerente de funcionalidades através de interfaces e especifica o uso de padrões, que qualquer aplicação suportada pela plataforma pode utilizar sem se preocupar com detalhes de como a funcionalidade fornecida pela plataforma é implementada". Alguns exemplos de plataforma são: CORBA, J2EE, Microsoft .NET, entre outras.

O modelo independente de computação (*Computation Independent Model* - CIM) é um modelo que tem como finalidade modelar os requisitos do sistema e descrever as situações em que o sistema será utilizado. Este modelo é geralmente, chamado de modelo de domínio ou modelo de negócio.

O CIM pode esconder muita ou toda a informação sobre processamento do sistema. Tipicamente este modelo é independente de como o sistema é executado e é utilizado para a facilitar comunicação de termos, conceitos e requisitos do domínio do problema, entre os especialistas do negócio e os desenvolvedores do sistema.

O modelo independente de plataforma (*Plataform Independent Model* - PIM) descreve o sistema que será construído sem se preocupar com detalhes específicos de uma determinada plataforma. Os elementos contidos em um PIM são adequados para serem utilizados com um número diferente de plataformas do mesmo tipo.

O modelo específico de plataforma (*Plataform Specific Model* - PSM) é o produto de um processo de transformação do PIM para uma determinada plataforma. O modelo PSM contém detalhes específicos e relevantes para o tipo de plataforma utilizada pelo sistema. O PSM representa o mesmo modelo de sistema que o PIM, porém adaptado a uma plataforma específica.

Um PSM pode conter uma grande ou pequena quantidade de detalhes sobre a construção do sistema, dependendo de sua finalidade. A partir de um único PIM é possível gerar um ou mais PSMs, para diferentes plataformas e diferentes finalidades.

O modelo específico de implementação (*Implementation Specific Model* - ISM) é um modelo que contém todos os detalhes necessários para a construção e

operação do sistema. Este modelo pode ser mapeado diretamente para o código, gerando assim a implementação do sistema. A geração do ISM é o passo final do processo de transformação de modelos do MDA. Através dos PSMs é obtido o ISM, que será realizado através da geração do código do sistema.

2.1.2 Transformação de Modelos

Em (MILLER; MUKERJI, 2003, p. 2-7) uma transformação de modelos é definida como "um processo de conversão de um modelo em outro modelo do mesmo sistema". No processo de transformação, os modelos do sistema são refinados e novas informações são adicionada a eles. Este processo é responsável, por exemplo, por gerar um PSM a partir de um PIM.

É através de sucessivos processos de transformação que as fases do processo de desenvolvimento do MDA são atingidas (ver seção 2.1.3). Ao final de uma sucessão de transformações é obtido o código do sistema. Essas transformações devem, idealmente, ser executadas de modo automático, ou pelo menos, com pouca intervenção manual.

A figura 2.3 apresenta um esboço básico do processo de transformação do MDA. Inicialmente tem-se um modelo CIM. Aplicando-se um processo de transformação utilizando esse modelo gera-se então o modelo PIM. O modelo PIM submetido a um processo de transformação pode gerar um ou mais PSMs. Esses PSMs submetidos a outro processo de transformação são utilizados para gerar o modelo ISM, que será realizado na forma do código do sistema.

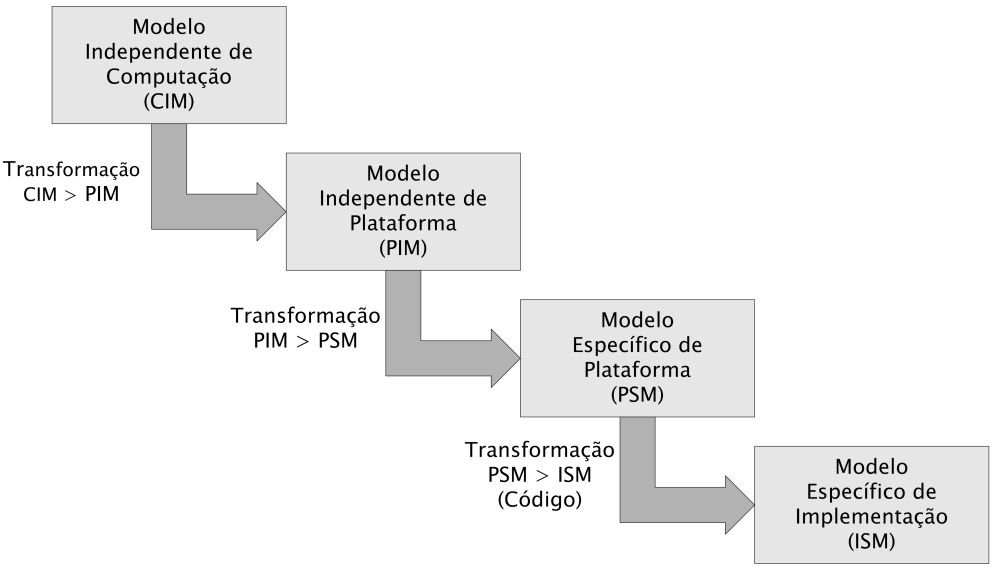

Figura 2.3: Processo de Transformação de Modelos na MDA.

O capítulo 4 aborda de forma abrangente a especificação e o processo de

transformação dos modelos na MDA.

2.1.3 Etapas do Desenvolvimento

As etapas básicas do ciclo de vida de desenvolvimento de software da MDA são realizadas basicamente em cinco passos: (MESERVY; FENSTERMACHER, 2005)

- Representação dos requisitos através do modelo CIM: Neste passo os elementos do domínio são eliciados sem que haja referência a um sistema particular de implementação ou tecnologia. Este modelo não se preocupa com os detalhes da solução do problema.
- Criação de um PIM: A criação de um PIM pode ser feita através de uma transformação do modelo CIM ou não. O modelo PIM deve captar conceitos chaves do domínio sem especificar detalhes de uma plataforma específica.
- O PIM é transformado em um ou mais PSMs, adicionando regras específicas de uma plataforma e códigos que a transformação não fornece. Nesta etapa pode-se ter vários passos de transformação para a geração dos PSMs.
- O PSM é transformado em código.
- Ocorre a implantação do sistema em um ambiente específico.

2.2 Meta-modelagem

A seção 2.1.1 apresentou a definição de modelo como a descrição de, ou parte de, um sistema escrito em uma linguagem bem definida. Um modelo de um sistema descreve quais elementos podem existir neste sistema (KLEPPE, 2003). Por exemplo, se um modelo orientado a objetos de um sistema define uma classe *Carro*, pode-se então, criar instâncias do tipo *Carro* naquele sistema. Uma linguagem de modelagem define quais elementos podem ser utilizados para a criação de modelos. Por exemplo, uma linguagem de modelagem utilizada para a construção de modelos orientados a objetos deve conter elementos como: Classe, Atributo, Operação, etc.

O conjunto de elementos que constituem uma linguagem de modelagem pode ser interpretado como um modelo (KLEPPE, 2003). Um modelo que define uma linguagem de modelagem é denominado meta-modelo. Um meta-modelo é, portanto, um modelo de uma linguagem de modelagem. Ele define a estrutura, a

semântica e as restrições para uma família de modelos, isto é, grupos de modelos que compartilham uma sintaxe e uma semântica comuns (MELLOR, 2004). A figura 2.4 apresenta a relação entre modelo, linguagem de modelagem e meta-modelo. Um modelo é escrito através da utilização de uma linguagem de modelagem, sendo que uma linguagem de modelagem é definida por um meta-modelo.

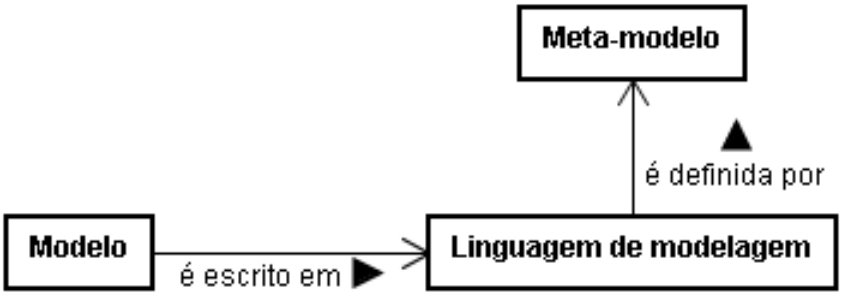

Figura 2.4: Relação entre modelo, linguagem de modelagem e meta-modelo.

Uma técnica comum na especificação de uma linguagem de modelagem é definir primeiro a sintaxe da linguagem e então a sua semântica estática e dinâmica (OMG, 2006d). Em linguagens que contêm sintaxe gráfica, é importante definir uma sintaxe que seja independente de uma notação. Este tipo de sintaxe é denominada de sintaxe abstrata. A sintaxe abstrata da linguagem é responsável por capturar a estrutura da linguagem (MELLOR, 2004). Ela define os elementos da linguagem e a regra de composição destes elementos (FRANKEL, 2003). A sintaxe abstrata separa a estrutura da linguagem dos símbolos notacionais da linguagem, que são representados na sintaxe concreta. A definição da sintaxe abstrata de uma linguagem pode ser expressa em um meta-modelo. OMG (2006d, p. 21) define que "a semântica estática de uma linguagem define como uma instância de uma construção deve ser conectada a outras instâncias para ser significante, e a semântica dinâmica define o significado de uma construção bem formada.". O capítulo 5 apresenta de forma detalhada os aspectos para a construção de uma linguagem de modelagem, no contexto da construção de linguagens de modelagem específicas a um domínio particular de problema.

Como um meta-modelo é também um modelo, este precisa ser especificado por uma linguagem de modelagem, ou um meta-modelo. O modelo que contém os elementos utilizados para a criação de um meta-modelo é denominado meta-meta-modelo. A meta-meta-modelagem forma uma base para uma hierarquia de meta-modelos (OMG, 2006d). Ela tem por responsabilidade definir uma linguagem para a especificação de meta-modelos. A figura 2.5 se expande do diagrama da figura 2.4, acrescentando uma relação entre um meta-modelo e um meta-meta-

modelo. Esta relação indica que um meta-modelo é definido por um meta-meta-modelo.

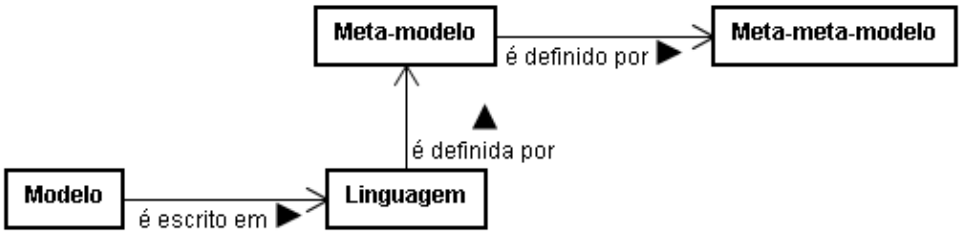

Figura 2.5: Relação entre modelo, linguagem de modelagem, meta-modelo e meta-meta-modelo.

A relação entre modelo, meta-modelo e meta-meta-modelo pode ser apresentada na forma de camadas. Esta apresentação em camadas, seguida pela OMG, é conhecida como as quatro camadas de modelagem da OMG. Estas camadas são denominadas da seguinte forma: M0, M1, M2, M3. A camada M0 contém os dados de uma aplicação, por exemplo, instâncias em tempo de execução do sistema. A camada M1 contém o modelo da aplicação, por exemplo, um modelo UML. A camada M2 contém o meta-modelo, por exemplo, o meta-modelo da UML. A camada M3 contém o meta-meta-modelo, por exemplo, o que define o meta-modelo da UML. A seção 3.1 adiante apresenta de forma mais detalhada este conceito.

Um meta-meta-modelo pode ser interpretado, também, como um modelo. Este então, pode ser definido por um outro meta-meta-modelo, isto é, pode existir uma camada M4 para definir os elementos utilizados pela camada M3. Em teoria, isto indica que pode existir um número infinito de camadas de meta-modelagem e meta-meta-modelagem (KLEPPE, 2003). Entretanto, a criação de uma camada M4 não é necessária na prática, uma vez que ela deveria conter os mesmos elementos da camada M3 (MELLOR, 2004). Portanto, a camada de meta-meta-modelagem é definida utilizando os elementos contidos nesta mesma camada, não havendo a necessidade de especificação de uma camada além da camada M3.

A OMG define um modelo, que reside na camada M3, e é utilizado para especificação de meta-modelos denominado *Meta Object Facility* (MOF) (OMG, 2006b). O MOF é um meta-meta-modelo responsável por definir elementos básicos para a criação de meta-modelos. O meta-modelo da UML é uma instância do MOF. Na verdade, cada meta-classe da UML é uma instância de um dos elementos da biblioteca de infra-estrutura definida na UML (*InfrastructureLibrary*) (OMG,

2006d). Com isso, pode-se dizer que a UML é especificada na própria UML. O MOF é também definido utilizando a biblioteca de infra-estrutura da UML.

Os modelos de um sistema são construídos utilizando elementos de um determinado meta-modelo. Isto é, os elementos utilizados por um projetista, que utiliza uma determinada linguagem de modelagem, devem estar contidos no meta-modelo da linguagem. Por exemplo, um modelo expresso através de diagramas UML deve conter somente elementos especificados no meta-modelo da UML. A figura 2.6 apresenta um sub-conjunto do meta-modelo da linguagem de modelagem UML (MELLOR, 2004) [2]. Nela pode-se observar alguns elementos que podem ser modelados utilizando a UML como, por exemplo, uma classe (*Class*), que é uma especialização de um classificador (*Classifier*) e que pode ter muitos propriedades (*Property*) e operações (*Operation*). O meta-modelo permite também que um classificador se relacione com outros classificadores através de associações (*Association*).

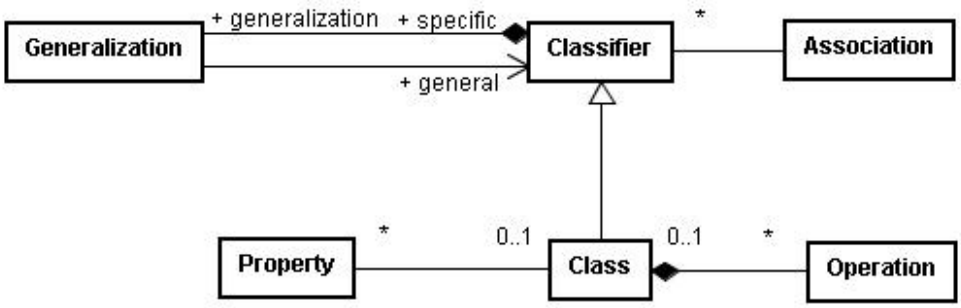

Figura 2.6: Sub-conjunto do meta-modelo da UML.

Pode-se então, utilizando os elementos do meta-modelo, criar modelos de uma determinada linguagem. Quando um determinado elemento do meta-modelo é utilizado para a geração de um modelo, define-se que o elemento do modelo é uma instância do (*instance of*) elemento do meta-modelo. Este conceito é similar ao conceito de instância na orientação a objetos, no qual um determinado objeto é uma instância de uma determinada classe.

A figura 2.7 apresenta um exemplo de um diagrama de classes descrito na linguagem UML. Este exemplo apresenta uma classe denominada "Pedido", sendo que este elemento de modelagem é uma instância de classe (*Class*) do meta-modelo da UML. Caso o elemento classe do meta-modelo da UML não existisse, não poderia ser criada uma classe "Pedido" no diagrama de classes UML. A figura apresenta ainda outros elementos de modelagem como associação (*Association*),

[2] Nesta figura foram mantidos os termos em inglês da especificação original da OMG.

atributo (*Attribute*) e multiplicidade (*Multiplicity*), também definidos no meta-modelo UML.

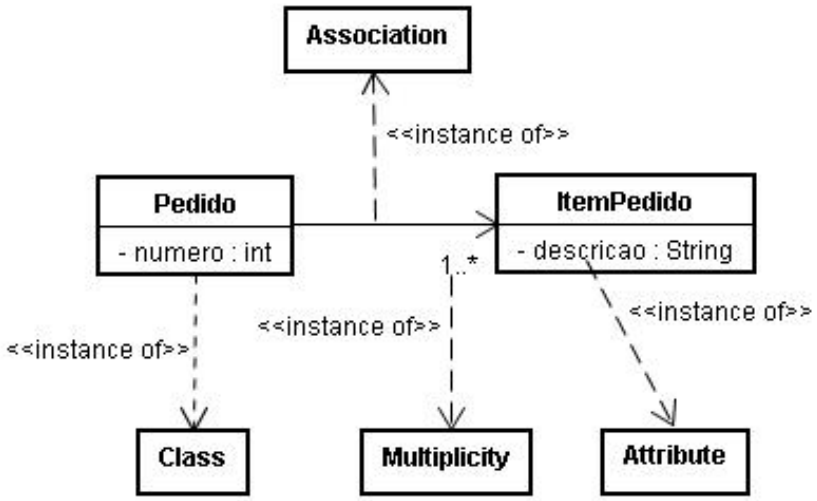

Figura 2.7: Exemplo de um diagrama de classe UML.

3 Padrões da OMG na MDA

Este capítulo apresenta brevemente os padrões criados pela OMG responsáveis por darem suporte ao arcabouço MDA. Inicialmente são apresentadas as quatro camadas de modelagem propostas pelo OMG. Posteriormente são abordados os padrões UML, *Meta Object Facility* (MOF), *XML Metadata Interchange* (XMI) e *Query Views Transformations* (QVT).

3.1 As Quatro Camadas de Modelagem da OMG

A relação de instanciação entre objetos, modelos e meta-modelos apresenta uma estrutura de camadas de modelagem (OMG, 2006d). Os elementos da camada de objetos são instâncias dos elementos da camada de modelos e estes são instâncias dos elementos da camada de meta-modelos. Para facilitar a comunicação sobre as camadas a OMG padronizou uma terminologia. As camadas então passaram a ser denominadas como M0, M1, M2 e M3.

- Camada M0: contém os dados da aplicação, como objetos em um sistema orientado a objetos ou linha de uma tabela de um banco de dados relacional. Esta camada representa o sistema em tempo de execução, que está rodando com as instâncias reais existentes.

- Camada M1: contém os modelos da aplicação, como diagrama de classes UML, modelos entidade-relacionamento, entre outros. Existe, uma relação entre as camadas M0 e M1, sendo que os conceitos da camada M1 são todos categorizações ou classificações das instâncias da camada M0. Cada elemento da camada M0 é sempre uma instância de um elemento da camada M1.

- Camada M2: contém o meta-modelo de uma determinada linguagem. Nesta camada estão contidos elementos como Classe, Atributo, Operação, os quais são representados no meta-modelo da UML. Este é o nível em que as ferramentas operam.

Além das camadas que compreendem os objetos, modelos e meta-modelos, a OMG criou uma quarta camada referente ao meta-meta-modelo. Esta camada contém os elementos que podem ser utilizados para a criação de meta-modelos (KLEPPE, 2003).

- Camada M3: contém o meta-meta-modelo que descreve os elementos que um meta-modelo pode exibir. Este é o nível no qual as linguagens de modelagem e os meta-modelos operam, provendo intercâmbio entre as ferramentas. Cada elemento da camada M3 categoriza elementos da camada M2, portanto os elementos da camada M2 são instâncias dos elementos da camada M3. É nesta camada que reside o *Meta Object Facility* (MOF).

Esta hierarquia de camadas poder-se-ia estender ao infinito. Por exemplo, poderia-se criar uma camada M4 para descrever os elementos da linguagem da camada M3, e assim por diante. Entretanto, a criação de uma camada M4 não é necessária na prática, uma vez que ela deveria conter os mesmos elementos da camada M3 (MELLOR, 2004). Sendo assim a OMG definiu que os elementos da camada M3 devem ser instâncias de conceitos da própria camada M3, isto é, a camada M3 se auto-descreve. Com isso, não é necessária a criação de novas camadas de modelagem.

A OMG padronizou uma linguagem que reside na camada M3 e é responsável pela criação de meta-modelos, denominada *Meta Object Facility* (MOF). Linguagens de modelagem como UML, entre outras, são instâncias de MOF. O MOF contém um conjunto de construções que são utilizados para a definição de meta-modelos. MELLOR (2004, p. 43) afirma que o MOF "capta a estrutura e a semântica de meta-modelos arbitrários, em particular o meta-modelo da UML, e também vários tipos de meta-dados".

A figura 3.1 apresenta uma visão geral das quatro camadas da OMG e a relação existente entre cada uma delas. A camada mais acima (M3) apresenta o elemento classe (*Class*) descrito no modelo MOF. Este elemento é utilizado pelo meta-modelo da UML (camada M2) para definir conceitos como classe (*Class*) e atributo (*Attribute*). Na camada M1 tem-se uma classe da representada em um diagrama de classes da UML, sendo que os elementos deste diagrama são instâncias dos conceitos apresentados na camada M2. A classe "Pedido" é uma instância do conceito classe do meta-modelo da UML e o atributo "numero" da classe "Pedido" é uma instância do conceito atributo do mesmo meta-modelo. Por fim, na camada M0, tem-se um objeto denominado "umPedido" que é uma

instância da classe "Pedido" do diagrama de classes do sistema. Este objeto representa uma instância em tempo de execução do sistema.

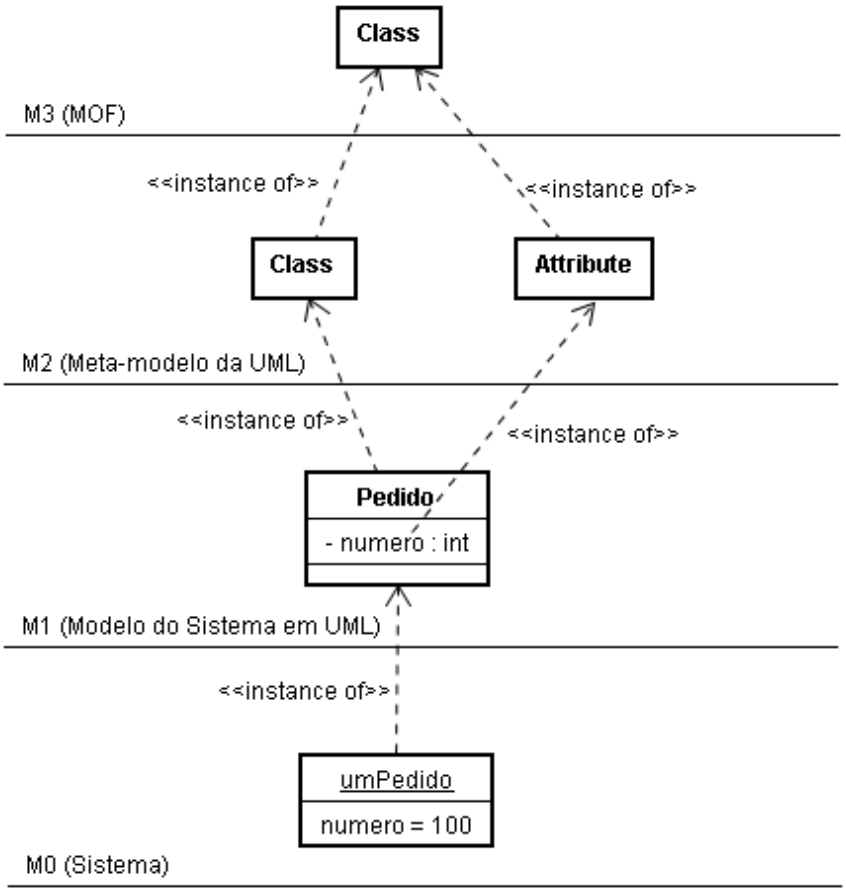

Figura 3.1: Visão geral das quatro camadas de modelagem da OMG.

3.2 *Unified Modeling Language - UML*

Não se pretende, nesta seção, descrever os elementos básicos da UML (OMG, 2005c) hoje bastante conhecidos, mas apresentar de modo sintético alguns conceitos básicos ainda não muito difundidos, os quais são essenciais para o presente trabalho.

Segundo (RUMBAUGH; JACOBSON; BOOCH, 2005), "a *Unified Modeling Language* (UML) é uma linguagem de modelagem visual de propósito geral que é utilizada para especificar, visualizar, construir e documentar os artefatos de um sistema de software". A UML é uma família de notações gráficas, apoiado por um único meta-modelo, que auxilia na descrição e no projeto (*design*) de sistemas de software, particularmente sistemas de software que são construídos utilizando o estilo orientado a objetos. (FOWLER, 2004)

A UML é recomendada e amplamente aceita como uma linguagem básica

para a especificação de modelos MDA (FOWLER, 2004). Um dos fatores que contribuem para essa ampla aceitação é o fato de a UML ser definida e mantida de forma padronizada pela OMG. Esta padronização permite a construção e comercialização de diferentes, porém complementares, ferramentas UML que podem ser utilizadas pelos desenvolvedores para criação dos seus ambientes de desenvolvimento (FRANKEL, 2003).

A definição da UML faz um clara separação entre a sintaxe abstrata da linguagem de sua sintaxe concreta (FRANKEL, 2003). A sintaxe abstrata define os conceitos da linguagem e como eles são relacionados entre si. A sintaxe concreta define a notação gráfica da linguagem utilizada para expressar a sintaxe abstrata. A clara separação entre a sintaxe abstrata e a sintaxe concreta da linguagem possibilita a utilização de outras sintaxes concretas para a linguagem. Um exemplo, é o padrão XMI que define uma sintaxe XML para expressar os modelos (ver seção 3.4).

A sintaxe abstrata da UML é definida através de um meta-modelo. O meta-modelo da UML é um diagrama de classes que define os conceitos da linguagem (FOWLER, 2004). Este meta-modelo descreve ainda a semântica das construções de modelagem (FRANKEL, 2003). A semântica dessas construções pode ser formalmente definida com o auxilio da *Object Constraint Language* (OCL) (OMG, 2006c).

A especificação da UML é composta por quatro partes (FRANCE et al., 2006):

- Infra-estrutura (*Infrastructure*) (OMG, 2006d): define os elementos básicos para as construções de modelagem da UML;
- Super-estrutura (*Superstructure*) (OMG, 2005c): define os conceitos utilizados pelos desenvolvedores para construir os modelos UML;
- Linguagem de restrição de objetos (*OCL*) (OMG, 2006c): define a linguagem para especificar consultas, restrições e operações nos modelos UML. Estas restrições são especificadas como invariantes, pré- e pós-condições;
- Intercâmbio de diagramas (*diagram interchange*) (OMG, 2006a): define uma extensão ao meta-modelo da UML para dar suporte ao armazenamento e a troca de informações relativos ao desenho (*layout*) dos diagramas UML.

3.2.1 *Object Constraint Language - OCL*

Um diagrama UML, como um diagrama de classe, para certos aspectos de um projeto (*design*), não possui todas as informações relevantes para uma especificação. Estes fatores podem ser alcançados adicionando anotações textuais aos modelos gerados (OMG, 2006c).

Muitos modelos gerados durante o desenvolvimento de *software* consistem de figuras com "bolhas e setas" e algum texto complementar. Segundo WARMER e KLEPPE (2003) "A informação expressa por tais modelos tem a tendência de ser incompleta, informal, imprecisa e às vezes até mesmo inconsistente". Estes diagramas não são capazes de expressar declarações que deveriam fazer parte de uma especificação mais completa. A figura 3.2 apresenta um exemplo de não expressividade de certos informações por parte do diagrama de classe da UML.

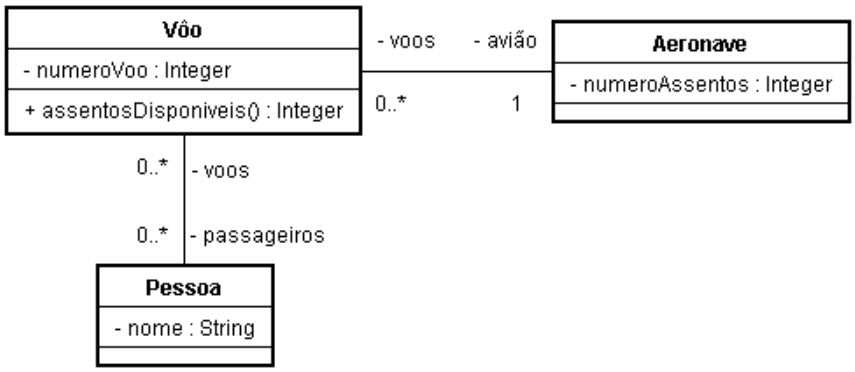

Figura 3.2: Um modelo expresso em um diagrama de classe UML.

A figura 3.2 apresenta uma associação entre a classe Pessoa e a classe Vôo. Esta associação indica que um certo grupo de pessoas são passageiros de um determinado vôo. A associação tem multiplicidade de muitos (0..*) no lado da classe de Pessoas. Isto significa que o número de passageiros para um determinado vôo pode ser ilimitado. Porém o número de passageiros é restrito ao número de assentos da aeronave que está associada ao vôo.

Apesar de a UML ter elementos que permitam resolver este problema (por exemplo, utilizando associações derivadas), o diagrama de classes pode ficar complexo. Uma alternativa para melhorar a legibilidade e, ao mesmo tempo, a precisão semântica, a OMG, responsável pela manutenção da UML, adicionou desde a versão 1.1 da UML a linguagem de restrição de objetos (*Object Constraint Language* - OCL) (OMG, 2006c). A OCL é um arcabouço para especificar restrições em um modelo de uma maneira formal. Este arcabouço é utilizado no processo de desenvolvimento de software para aumentar a expressividade de certos artefatos

gerados durante a fase de análise e projeto (*design*) do software. (RICHTER; GOGOLLA, 2002) (TOVAL; REQUENA; FERNÁNDEZ, 2003)

A OCL é uma linguagem formal utilizada para expressar condições que devem ser verdadeiras para o sistema que está sendo modelado. As expressões em OCL não são ambíguas. Estas expressões normalmente especificam condições invariantes que precisam ser asseguradas pelo sistema que está sendo modelado. A OCL permite ainda especificar consultas sobre objetos descritos em um modelo. (OMG, 2006c)

A OCL tem como característica ser uma linguagem textual baseada em expressões (WARMER; KLEPPE, 2003). Cada expressão escrita em OCL depende dos tipos (classes, interfaces, etc.) que são definidos em um diagrama UML. Portanto para utilizar a OCL é necessário que se utilize pelo menos algum aspecto da UML. As principais características da OCL são: (OMG, 2006c) (TOVAL; REQUENA; FERNÁNDEZ, 2003)

- A OCL é uma linguagem de especificação puramente declarativa. Isto garante que as expressões OCL não causam nenhum efeito colateral quando avaliadas. A expressão simplesmente retorna um valor;
- A OCL provê um meio preciso e fácil de expressar restrições na estrutura dos modelos;
- A OCL não é uma linguagem de programação. Portanto, não é possível escrever lógica de programação ou fluxo de controle utilizando OCL;
- A OCL é uma linguagem com tipos fortes, então cada expressão OCL tem um tipo. Para uma expressão estar correta, todos os tipos precisam estar em conformidade com as regras da linguagem. Por exemplo, não se pode comparar um inteiro (*Integer*) com uma cadeia de caracteres (*String*);
- Cada expressão OCL é anexada a um modelo e descreve características específicas do modelo.

A OCL pode ser utilizada para um número diferente de propósitos:

- Como uma linguagem de consulta;
- Para especificar invariantes em classes e tipos em um modelo de classes;
- Para especificar invariantes de tipos para estereótipos;

- Para descrever pré- e pós-condições em operações e métodos;
- Para descrever guardas (*guards*);
- Para especificar (conjunto de) alvos para mensagens e ações;
- Para especificar restrições em operações;
- Para especificar regras de derivação para atributos para qualquer expressão sobre um modelo UML.

O modelo da figura 3.2, conforme apresentado anteriormente, permite que um número ilimitado de passageiros possa viajar em um determinado vôo. Para adicionar maior expressividade ao modelo e solucionar o problema de número ilimitado de passageiros, pode-se adicionar ao modelo uma expressão OCL especificando uma condição invariante para a classe Vôo. Esta expressão indica que o número de passageiros de um vôo não deve ser maior que o número de assentos do avião associado ao vôo. A expressão OCL que descreve esta invariante é a seguinte:

```
-- A quantidade de passageiros de um vôo não pode ser
-- maior que o número de acentos de um avião.
context Vôo
   inv:  passageiros->size() <= avião.numeroAssentos
```

A OCL adiciona ao modelo maior precisão e formalidade. No processo de desenvolvimento MDA os modelos gerados devem ser bem definidos, para que possam ser entendidos pelo computador e para permitir que o processo de transformação entre os modelos seja feita de forma automatizada. Para atingir estes e outros objetivos, a OCL torna-se um ingrediente chave dentro do processo de desenvolvimento MDA.

Entretanto, o papel da OCL dentro do MDA não se limita apenas a aumentar a precisão dos modelos gerados. A OCL pode ser usada também para definir meta-modelos, definir novas linguagens através da utilização de perfis UML e definir transformações entre os modelos.

Uma vez que um meta-modelo é simplesmente um modelo que reside em um outro nível de abstração, a utilização da OCL é similar à descrita nas seções anteriores. Isto faz com que estes meta-modelos apresentem regras bem formadas (WARMER; KLEPPE, 2003). O próprio meta-modelo da OCL é definido utilizando expressões OCL, sendo essas utilizadas para adicionarem restrições aos elementos do meta-modelo. Um exemplo de expressão OCL utilizada no meta-modelo da UML é apresentada a seguir. O exemplo apresenta uma restrição ao

elemento da UML denominado *Interface*. Uma interface não pode ter nenhum atributo. A expressão especifica uma invariante ao elementos Interface, conforme descrito abaixo. WARMER e KLEPPE (2003, p. 41) afirma que "uma invariante é uma expressão booleana que declara uma condição que deve ser sempre satisfeita por todas as instâncias do tipo ao qual ela foi definida".

```
-- Não pode ter nenhum atributo em uma Interface.
context Interface
   inv:  features->select(f | f.oclIsKindOf(Attribute))->isEmpty()
```

Uma transformação descreve como um modelo escrito em uma linguagem pode ser transformado em um modelo escrito em uma outra linguagem (WARMER; KLEPPE, 2003). As definições de transformação utilizam conceitos das duas linguagens para criarem as transformações. Os conceitos destas linguagens estão descritos em seus respectivos meta-modelos. As definições de transformação portanto, relacionam elementos dos meta-modelos das linguagens fonte e alvo.

Para que as especificações de transformações possam ser automatizadas, elas precisam ser escritas de uma maneira precisa e não ambígua. A OCL pode auxiliar na obtenção deste objetivo. Na especificação de uma transformação a OCL pode, por exemplo, indicar precisamente quais elementos do meta-modelo fonte e do meta-modelo alvo são utilizados em uma determinada transformação e como os elementos do meta-modelo fonte podem ser mapeados nos elementos do meta-modelo alvo. Um outro exemplo de utilização é fornecido em (CARIOU et al., 2004) sugerindo que a OCL especifique contratos de transformações de modelos, como por exemplo, pré- e pós-condições que um determinado processo de transformação deve respeitar.

3.2.2 Perfil UML

A UML pode ser facilmente personalizada utilizando um conjunto de mecanismos de extensão que é fornecido pelo própria UML (FUENTES-FERNÁNDEZ; VALLECILLO-MORENO, 2004). Mais precisamente, o pacote *Profiles*, incluso na UML 2.0, define um conjunto de artefatos que permite a especificação de um modelo MOF para lidar com conceitos e notações específicos requeridos por um domínio de aplicação particular, ou por uma tecnologia de implementação. Os perfis UML permitem a personalização de qualquer meta-modelo definido pelo MOF (não somente a UML). Um perfil UML pode também especificar outro perfil UML.

Os perfis UML são definidos em termos de dois mecanismos básicos: es-

tereótipos (*stereotypes*), restrições (*constraints*).

1. Um estereótipo é definido por um nome e por um conjunto de elementos do meta-modelo.

2. Restrições podem ser associadas com estereótipos, impondo restrições nos elementos do meta-modelo correspondentes. As restrições podem ser expressas em qualquer linguagem, incluindo a linguagem natural ou a OCL.

Abaixo apresenta-se um exemplo de uma definição de perfil UML.

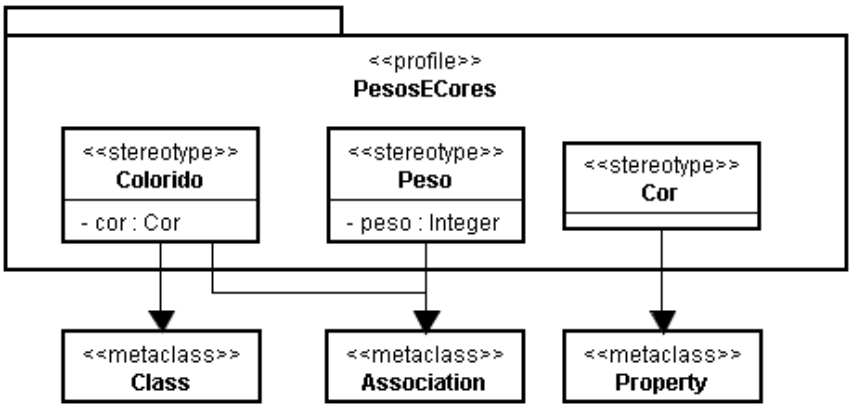

Figura 3.3: Exemplo de um perfil UML.

Um perfil UML é um conjunto de mecanismos de extensão agrupados em um pacote UML estereotipado por ≪profile≫. A figura 3.3 apresenta o perfil UML *PesosECores*. Este perfil define três estereótipos: *Colorido*, *Peso* e *Cor*. Cada estereótipo é associado ao elemento do meta-modelo da UML que ele estende. A notação de extensão do elemento do meta-modelo da UML é uma seta, apontando do estereótipo para a meta-classe que ele estende, onde a ponta da seta é um triângulo preenchido (OMG, 2005c). O estereótipo *Colorido* estende a meta-classe *Class* do meta-modelo da UML e contém o atributo *cor*. O estereótipo *Peso* estende a meta-classe *Association* e contém o atributo *peso*. O estereótipo *Cor* estende a meta-classe *Property*. O capítulo 5 apresenta como os perfis UML podem ser utilizados para a criação de linguagens específicas de um determinado domínio.

3.3 *Meta Object Facility - MOF*

O MOF (OMG, 2006b) é um padrão da OMG que, conforme apresentado na seção 3.1 reside na camada M3 da arquitetura de modelagem. Ele é um meta-

modelo designado para a criação de outros meta-modelos, sendo então referenciado como um meta-meta-modelo. Fornece um arcabouço para gerenciamento de meta-dados, e um conjunto de serviços de meta-dados que permitem o desenvolvimento e a interoperabilidade de sistemas dirigidos a modelos e meta-dados. Os meta-modelos utilizam o Meta Object Facility (MOF) para definir formalmente a sintaxe abstrata do seu conjunto de construção de modelagem (FRANKEL, 2003). A semântica dos meta-modelos é especificada utilizando linguagem natural e *Object Constraint Language* (OCL).

O MOF fornece também um mecanismo de extensão de meta-modelo. Este mecanismo é geralmente referenciado com um mecanismo de extensão mais detalhado do que os perfis UML. Este tipo de extensão permite criar classes adicionais no nível M2 (meta-modelo) das camadas de modelagem da OMG. Esta abordagem é significantemente mais flexível do que a abordagem utilizando perfil UML, porque novas classes M2 podem ser criadas tanto como sub-classes das classes do meta-modelo ou como classes separadas e associadas ao meta-modelo. Entretanto, a utilização deste mecanismo não garante que a semântica dos elementos do meta-modelo original seja preservada (COOK, 2000).

3.4 *XML Metadata Interchange - XMI*

O padrão MOF contém propriedades que permitem a serialização de modelos e meta-modelos, com o objetivo de fornecer um padrão externo de representação destes modelos. Esta serialização é feita através de um documento *eXtensible Markup Language* (XML) e permite o intercâmbio de modelos entre localizações geográficas, seres humanos, computadores e ferramentas (BÉZIVIN; GÉRARD, 2002). As propriedades de serializações e geração do documento eXtensible Markup Language (XML) são definidas pelo padrão XML Metadata Interchange (XMI) (OMG, 2005b).

O padrão XMI define regras para derivar um esquema XML de uma linguagem de modelagem em conformidade com o padrão MOF e também regras para interpretar um modelo em um documento XML (MELLOR, 2004). Pode-se, por exemplo, gerar um arquivo XMI de um certo modelo descrito pela linguagem UML. Os principais benefícios do XMI são (GROSE; DONEY; BRODSKY, 2002):

- Fornecer um padrão de representação de objetos no formato XML, permitindo o intercâmbio efetivo de objetos usando XML;

- Especificar a criação de esquemas XML a partir de modelos;
- Auxiliar a produção de documentos XML que podem ser facilmente intercambiados;
- Permitir trabalhar com dados e meta-dados.

O formato de representação dos modelos utilizando o XMI permite a interoperabilidade dos modelos entre diversos tipos de ferramentas e repositórios. Para isto, as ferramentas devem ter a capacidade de importar e/ou exportar documentos XMI. Por ser uma representação padronizada para a serialização de modelos, o XMI possibilita, por exemplo, que um determinado modelo UML gerado em uma ferramenta de modelagem, e exportado em formato XMI, possa ser utilizado por uma ferramenta de transformação de modelos, desde que esta permita a importação de documentos XMI.

3.5 *Query Views Transformations - QVT*

O *Query Views Transformations* (QVT) tem por objetivo definir uma forma padronizada para a especificação de transformações de modelos no contexto da MDA (OMG, 2005a). Esta padronização tem como alguns objetivos (DUDDY et al., 2003):

- Definir uma linguagem para pesquisar em modelos baseados no MOF;
- Definir uma linguagem para especificação de transformação de modelos;
- Permitir a criação de vistas de um modelo;
- Garantir que a linguagem de transformação seja declarativa e expresse transformações completas;
- Garantir que mudanças incrementais no modelo fonte possam ser imediatamente reproduzidas nos modelos alvo.

O capítulo 4 aborda de forma mais abrangente o tema de transformação de modelos e o padrão Query Views Transformations (QVT).

4 Transformação de Modelos

A transformação de modelos é um processo-chave dentro do ciclo de desenvolvimento MDA. Segundo Kleppe et al. (KLEPPE, 2003, p. 24)

> Uma transformação é a geração automática de um modelo alvo a partir de um modelo fonte, de acordo com uma definição de transformação. Uma definição de transformação é um conjunto de regras de transformação que, juntas, descrevem como um modelo em uma linguagem fonte pode ser transformado em um modelo em uma linguagem alvo. Uma regra de transformação é a descrição de como um ou mais construções na linguagem fonte podem ser transformadas em uma ou mais construções na linguagem alvo.

Em (MENS; GORP, 2005) propõe-se uma generalização da definição apresentada em que a transformação de modelos pode ser aplicada para múltiplos modelos fontes e/ou múltiplos modelos alvos. Exemplos desta definição são as combinações de modelos (*model merging*) e a geração de vários PSMs a partir de um único PIM. A figura 4.1 apresenta uma representação destes dois exemplos.

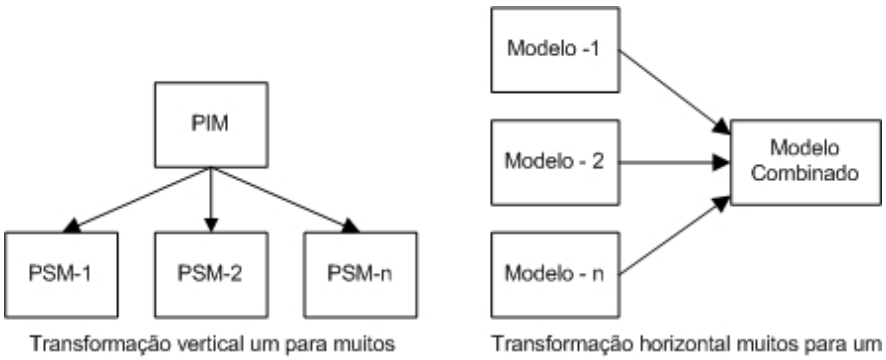

Figura 4.1: Exemplos de Transformação de Modelos.

Para que o processo de transformação de modelos seja automático, as ferramentas que o executarem precisam de um modelo que define a estrutura e regras de boa formação da linguagem no qual os modelos serão expressados. Conforme apresentado na seção 2.2, tais modelos são denominados meta-modelos. Segundo MENS e GORP (2005), deve-se ter meta-modelos precisos, isto é, especificados de uma maneira formal, como um pré-requisito para a execução automática de

transformação de modelos, uma vez que os meta-modelos devem ser interpretados pelas ferramentas de transformação de modelos.

4.1 Tipos de Transformação de Modelos

Os modelos fonte e alvo do processo de transformação precisam ser expressos através de uma linguagem de modelagem. Baseado na linguagem em que os modelos fonte e alvos são expressados pode-se definir a transformação como sendo endógena ou exógena. (MENS; GORP, 2005)

Transformações endógenas são transformações entre modelos expressos na mesma linguagem. Exemplos típicos deste tipo de transformação são (MENS; GORP, 2005):

- Otimização: transformação com o objetivo de aprimorar certas qualidades operacionais, preservando a semântica do modelo.
- Refatoração: altera a estrutura do modelo para aprimorar certas características de qualidade, sem alterar as características comportamentais do modelo.
- Simplificação e normalização: utilizada para diminuir a complexidade sintática de um modelo.

Transformações exógenas são transformações entre modelos expressos utilizando diferentes linguagens. Exemplos típicos deste tipo de transformação são (MENS; GORP, 2005):

- Sintese: transformação de um modelo em um alto nível de abstração em um modelo em um nível mais baixo de abstração, mais concreto.
- Engenharia Reversa: é o inverso da sintese, extraindo uma especificação de alto nível de uma específicação de baixo nível.
- Migração: transforma um modelo escrito em uma linguagem para um modelo escrito em outra linguagem, porém preservando o mesmo nível de abstração.

As transformações podem ser executadas de forma horizontal ou de forma vertical. Uma transformação horizontal é uma transformação na qual os modelos fonte e alvo residem no mesmo nível de abstração. Uma transformação vertical é

uma transformação na qual os modelos fonte e alvo residem em níveis de abstração diferentes. (MENS; GORP, 2005)

4.2 Mapeamento

No presente trabalho, o termo mapeamento é utilizado como sinônimo de correspondência entre elementos de dois meta-modelos, seguindo as definições de (LOPES et al., 2005). Os autores definem também que uma transformação é a atividade de transformar um modelo fonte em um modelo alvo em conformidade com uma definição de transformação. Baseado nestes dois conceitos, um mapeamento pode ser um passo preliminar antes da criação de uma definição de transformação. Ele pode estabelecer uma ligação entre os elementos do meta-modelo fonte e do meta-modelo alvo que serão utilizados em um processo de transformação de modelos. Um mapeamento MDA provê especificações para o processo de transformação de um PIM em um PSM para uma plataforma particular. O modelo da plataforma irá determinar a natureza do mapeamento. Em (MILLER; MUKERJI, 2003) são especificados os seguintes tipos de mapeamento:

- Mapeamento de Tipo de Modelo: Especifica um mapeamento de qualquer modelo utilizando tipos especificados em uma linguagem PIM para modelos expressos usando tipos de uma linguagem PSM. Um mapeamento de meta-modelo é um exemplo específico de um mapeamento de tipo de modelo, em que os tipos de elementos do PIM e do PSM são especificados como meta-modelos MOF.

- Mapeamento de Instâncias do Modelo: Outra abordagem de mapeamento é identificar elementos de um modelo PIM que deve ser transformado de um modo particular, dando uma escolha da plataforma específica para o PSM. Este tipo de mapeamento irá definir marcas. Uma marca representa um conceito do PSM e é aplicada em um elemento do PIM, indicando como aquele elemento deve ser transformado. As marcas não fazem parte do PIM.

Um mapeamento pode também incluir gabaritos (*templates*), que são modelos parametrizados responsáveis por especificar tipos particulares de transformações. Os gabaritos podem ser usados em regras para transformar um padrão de elementos de um modelo em um mapeamento de tipo de modelo dentro de outro padrão de elementos de modelo. Um conjunto de marcas podem ser associados a um gabarito para indicar instâncias de um modelo que devem ser transformados de acordo com o gabarito. (MILLER; MUKERJI, 2003)

Dado $M_1(s)/M_a$ e $M_2(s)/M_b$, em que M_1 é um modelo de um sistema s criado utilizando o meta-modelo M_a e M_2 é um modelo do mesmo sistema s criado utilizando o meta-modelo M_b. Um mapeamento pode ser definido como $C_{M_a \to M_b}/M_c$, em que $C_{M_a \to M_b}$ é o mapeamento entre os meta-modelos M_a e M_b e é criado utilizando o meta-modelo M_c (LOPES et al., 2005). Pode-se perceber a necessidade de um meta-modelo capaz de efetuar o mapeamento entre outros dois meta-modelos. Um mapeamento, então, é especificado utilizando uma linguagem para descrevê-lo. Para descrever um mapeamento pode-se utilizar uma linguagem natural, um algoritmo em uma linguagem de ação ou mesmo uma linguagem de mapeamento de modelos (MILLER; MUKERJI, 2003).

4.3 Atributos para Linguagens de Transformações

Esta seção está baseada em (CZARNECKI; HELSEN, 2003), apresentando uma série de atributos que as linguagens de transformação de modelos podem oferecer para a especificação de uma transformação de modelos. Os atributos que seguem estão baseados em diversas linguagens e enfoques de transformação de modelos.

Regras de Transformação

Uma especificação de transformação é formada por uma ou mais regras de transformação. Uma regra de transformação é composta por partes: o lado esquerdo (LHS) e o lado direito (RHS). O LHS acessa o modelo fonte, enquanto o RHS cria e/ou expande o modelo alvo.

Uma regra de transformação é representada utilizando variáveis, padrões e lógica. As variáveis guardam elementos do modelo fonte e/ou do modelo alvo. Padrões são fragmentos de modelo contendo zero ou mais variáveis. A lógica expressa computações e restrições nos elementos do modelo.

Escopo de Aplicação de Regra

O escopo de aplicação da regra permite que uma transformação limite partes de um modelo que participam de uma transformação. Isto, possibilita que apenas parte do modelo fonte seja utilizada para a execução da transformação.

Relacionamento entre o modelo fonte e o modelo alvo

O relacionamento entre o modelo fonte e o modelo indica se a transformação deve criar um novo modelo alvo, se pode utilizar um modelo já existente ou se

o modelo alvo e o modelo fonte são os mesmos. Os dois primeiros casos são denominados de transformação *out-place*. O último caso é denominado de transformação *in-place*. Transformação do tipo exógeno são sempre transformações *out-place* (MENS; GORP, 2005).

Estratégia de aplicação de uma regra

Em certos casos pode ocorrer mais de uma combinação para uma regra dentro do escopo de aplicação do modelo fonte. Quando isso ocorrer é necessário que haja uma estratégia de aplicação da regra. As estratégias de aplicação de uma regra de transformação podem ser determinísticas, não determinísticas e interativas. Por exemplo, uma estratégia determinística pode utilizar uma alguma estratégia de travessia padrão, como busca em profundidade (*depth-first*), sobre o conteúdo hierárquico do modelo fonte. Exemplos de estratégias não determinísticas incluem aplicação em um ponto (*one-point*), em que uma regra é aplicada em um ponto selecionado não deterministicamente, e aplicação concorrente, em que uma regra é aplicada concorrentemente em todos os locais do modelo fonte. Algumas regras são aplicadas interativamente, por exemplo, através da interação do usuário com a ferramenta de transformação.

Escalonador de Regra

Determina a ordem em que uma determinada regra será aplicada. O mecanismo de programação pode variar em quatro grandes áreas: forma, seleção de regra, iteração de regra e fases. A forma indica se a programação é implícita ou explicita, sendo que na implícita o usuário não tem nenhum controle sobre o algoritmo de escalonamento. A seleção de regra é utilizada para selecionar uma regra através de uma condição explícita. A iteração de regras permite a utilização de mecanismos de laço e iteração de ponto fixo. O processo de transformação pode ser organizado em fases, permitindo que cada fase tenha um propósito e que certas regras só podem ser acionadas em uma determinada fase.

Ligações de Rastreabilidade

As transformações podem armazenar registros de ligações entre os elementos do modelo fonte e do modelo alvo. Estes registros podem ser úteis para, por exemplo, análise de impacto, sincronização entre modelos, etc. O registro das ligações pode ser efetuado de forma automática ou manual. É recomendado que as informações de rastreabilidade sejam armazenadas de forma separada ao modelo fonte e ao modelo alvo.

Direcionalidade

Transformações podem ser unidirecionais ou bidirecionais. As transformações unidirecionais podem somente ser executadas em uma direção. As transformações bidirecionais podem ser executadas em ambas as direções. Nas linguagens de transformação que permitem apenas transformações unidirecionais, a transformação bidirecional é feita através da definição de duas regras unidirecionais complementares, sendo uma regra para cada direção. Regras declarativas freqüentemente podem ser aplicadas na direção inversa, dependendo da inversibilidade da regra e da lógica de escalonamento.

4.4 Mecanismos de Transformação de Modelos

Esta seção apresenta alguns mecanismos utilizados para a transformação de modelos. Estes mecanismos incluem técnicas, linguagens, métodos e idéias de paradigmas de programação que podem ser utilizadas para especificar e aplicar transformações de modelos (MENS; GORP, 2005). Técnicas dos paradigmas procedimental, orientado a objetos, funcional ou lógico podem ser utilizadas para tal propósito. É possível ainda, a utilização de um mecanismo híbrido, o qual combina vários paradigmas.

A maior distinção entre os mecanismos de transformação está entre os enfoques declarativos ou operacionais (ou imperativos) (MENS; GORP, 2005). O enfoque declarativo preocupa-se com o aspecto *do que* é necessário ser feito, por exemplo, através da definição de um mapeamento entre os elementos do modelo fonte e do modelo alvo. O enfoque operacional preocupa-se com o aspecto de *como* fazer, por exemplo, através da especificação dos passos que devem ser seguidos para que seja possível derivar o modelo alvo do modelo fonte.

CZARNECKI e HELSEN (2003) apresentam alguns enfoques que podem ser utilizados para a transformação de modelos. Esses enfoques estão separados em duas grandes categorias: transformação de modelo para código e transformação de modelo para modelo. A transformação de modelo para código pode ser interpretada como um caso especial da transformação modelo para modelo, sendo somente necessária a existência de um meta-modelo da linguagem de programação.

Os enfoques de transformação de modelos para código são divididos nas categorias: baseados em visita e baseados em gabaritos. O enfoque baseado em visita é um enfoque básico para a geração de código que consiste em fornecer algum mecanismo de visita que atravessa a representação interna de um modelo e escreve o código em um fluxo de texto. O enfoque baseado em gabaritos geral-

mente consiste de um texto alvo contendo junções de meta-códigos para acessar informações de um modelo fonte e executar a seleção e a expansão iterativa do código.

Os enfoques de transformação de modelos para modelo são divididos nas categorias: manipulação direta, relacional, baseados em transformação de grafos e dirigida a estruturas. O enfoque de manipulação direta oferece uma representação interna do modelo e uma interface para programação (*Application Program Interface* - API) para manipulá-lo. Esse enfoque geralmente é implementado como um arcabouço orientado a objetos, fornecendo uma infra-estrutura para a organização das transformações. Os usuários destes mecanismo geralmente têm que implementar regras de transformação utilizando uma linguagem de programação, como por exemplo, a linguagem JAVA.

A categoria relacional agrupa enfoques declarativos, em que os principais conceitos estão em relações matemáticas e regras de mapeamento. CZARNECKI e HELSEN (2003, p. 11) afirmam que a principal idéia desse enfoque é "expressar o tipo de elemento fonte e alvo de uma relação e especificar essa relação utilizando restrições". Essas especificações, em sua forma pura, são não executáveis. Entretanto, as restrições declarativas podem fornecer semânticas executáveis, como em programação em lógica.

O enfoque baseado em transformação de grafos é fundamentado na teoria de grafos (BALOGH; VARRÓ, 2006). Dada a complexidade da questão e como seus conceitos não serão necessários ao presente trabalho, não será apresentado um resumo dessa categoria de transformação.

O enfoque dirigido a estrutura primeiramente cria uma estrutura do modelo alvo e, posteriormente, coloca os atributos e referências no modelo alvo. O arcabouço desse tipo de enfoque determina a programação e a estratégia de aplicação das regras. Os usuários devem apenas fornecer as especificações das regras de transformação.

Um outro enfoque para a transformação de modelos é implementar transformações utilizando a linguagem de folhas de estilo extensível para transformação *eXtensible Stylesheet Language for Transformation* (XSLT), que é uma tecnologia padrão, utilizada para especificar transformações em documentos XML. Para realizar a transformação utilizando a XSLT, os modelos MOF/UML devem ser serializados como um XML utilizando a especificação XMI. Entretanto, esse enfoque tem sérias limitações de escalabilidade e é muito trabalhoso para ser criado e mantido manualmente, devido à grande verbosidade e dificuldade de leitura dos

documentos XMI e XSLT.

4.5 Linguagens de Transformação de Modelos

Não se pretende, nesta seção, apresentar todo o conjunto de linguagens de transformações criadas para dar suporte ao MDA. A lista seria grande e está fora do escopo do trabalho. O que se pretende nesta seção é apresentar algumas características utilizadas pelas linguagens para efetuar a transformação de modelos. Resumem-se, apenas, as principais características das linguagens de transformação MT (TRATT, 2006) e QVT (OMG, 2005a). A linguagem *ATLAS Transformation Language* (ATL), utilizada ao longo do presente trabalho, é descrita na seção 4.6. A escolha da ATL deve-se ao fato de haver um ambiente de desenvolvimento que permite a implementação, depuração e execução das especificações de transformação de modelos. Outro fator importante para a escolha da ATL é o fato de seu ambiente de desenvolvimento permitir a integração da ferramenta utilizada para a especificação dos modelos de tecelagem de modelos, denominada *ATLAS Model Weaving* (AMW).

Linguagem de Transformação de Modelos MT

A linguagem de transformação MT (TRATT, 2006) é implementada como uma linguagem específica de domínio (*Domain Specific Language* - DSL) sobre a linguagem de programação Converge (TRATT, 2005). Ela é uma linguagem de transformação unidirecional e utiliza padrões declarativos para combinar com os elementos do modelo fonte.

A linguagem de transformação MT é baseada em regras. Uma transformação contém um nome e consiste de uma ou mais regras. A ordem das regras é significativa na execução da transformação. As regras são basicamente funções que definem um número fixo de parâmetros. Para que a execução da transformação tenha sucesso é preciso que os parâmetros combinem com os padrões definidos na regra.

Quando uma regra MT é executada com sucesso, ela produz um ou mais elementos do modelo fonte. Uma característica importante é que o modelo fonte não sofre nenhum tipo de alteração. Um novo modelo, destinado a ser o modelo alvo é criado na execução da transformação. Durante a execução das regras são gravados automaticamente informações de rastreabilidade.

Query/Views/Transformation - QVT

A linguagem QVT, proposta pela OMG, é uma linguagem padronizada para a transformação de modelos no contexto da arquitetura de meta-modelagem MOF 2.0 (OMG, 2005a). A QVT é uma linguagem híbrida, pois aceita construções tanto declarativas quanto imperativas. Ela é formada por três sub-linguagens: Núcleo (*Core*), Relações (*Relations*) e Mapeamentos Operacionais (*Operational Mappings*).

A sub-linguagem Relações especifica declarativamente transformações como um conjunto de relações entre modelos MOF (OMG, 2005a). Ela contém um conjunto de padrões de objetos que podem ser combinados com elementos de modelos existentes, utilizados para criar elementos em novos modelos ou serem utilizados para aplicar mudanças em modelos existentes (JOUAULT; KURTEV, 2006). Durante a execução da transformação, ligações de rastreabilidade são geradas automaticamente.

A sub-linguagem Núcleo, segundo (OMG, 2005a, p. 9):

> apóia apenas um padrão de combinação sobre um conjunto de variáveis horizontais através da avaliação de condições sobre essas variáveis contra um conjunto de modelos. Todos os elementos dos modelos fonte, alvo e modelos de rastreamento são tratados simetricamente.

As ligações de rastreabilidade são tratadas como elementos de modelos, e é de responsabilidade do desenvolvedor criar e utilizar estas ligações. Segundo (JOUAULT; KURTEV, 2006, p. 2) "um dos propósitos da linguagem Núcleo é fornecer a base para a especificação semântica da linguagem Relações".

A sub-linguagem Mapeamentos Operacionais é uma extensão das linguagens Núcleo e Relações e tem por objetivo fornecer construções imperativas e restrições OCL com efeito colateral (JOUAULT; KURTEV, 2006). As restrições OCL com efeito colateral permitem um estilo procedimental e uma sintaxe concreta similar a uma linguagem de programação imperativa. A idéia básica desta linguagem é utilizar construções imperativas para instanciar padrões de objetos especificados utilizando a linguagem Relações. Este mecanismo é utilizado quando as linguagens declarativas não oferecem soluções completas para um determinado problema de transformação.

A figura 4.2 apresenta a arquitetura da linguagem QVT. A parte declarativa da linguagem é estruturada por duas camadas da arquitetura e é constituída pelas sub-linguagens Núcleo e Relações (OMG, 2005a). A parte imperativa é constituída por dois mecanismos: a sub-linguagem Mapeamentos Operacionais e uma implementação Caixa-Preta.

O mecanismo de Caixa-Preta permite conectar e executar um código externo durante a execução da transformação, fazendo com que partes da transformação tornem-se opacas, permitindo a execução de funcionalidade arbitrária que não são controladas pelo motor de transformação (JOUAULT; KURTEV, 2006). Este mecanismo é útil para a implementação de algoritmos complexos e reutilização de bibliotecas já existentes.

Os dois mecanismos estendem as linguagens Núcleo e Relações para fornecer implementações imperativas da linguagem QVT. Uma especificação na sub-linguagem Relações pode ser transformada para uma especificação na sub-linguagem Núcleo, através da transformação RelaçãoParaNúcleo.

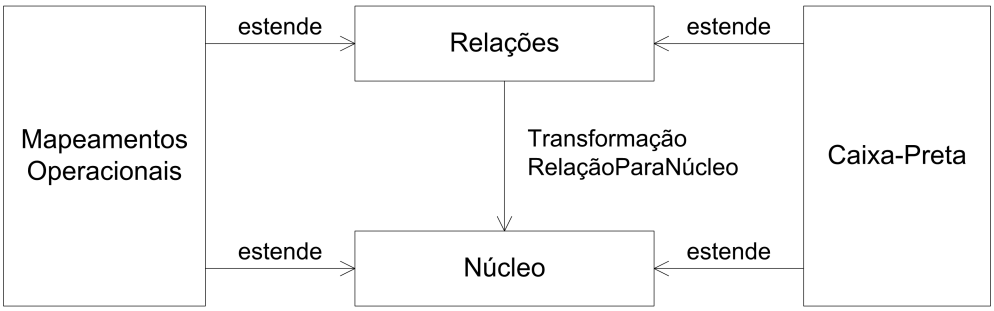

Figura 4.2: Arquitetura da linguagem QVT.

4.6 A Linguagem de Transformação ATL

A linguagem de transformação ATL (ATLAS Transformation Language - ATL) (JOUAULT; KURTEV, 2005) é uma linguagem híbrida, como a QVT (OMG, 2005a), pois aceita tanto construções declarativas quanto construções imperativas. A parte declarativa da linguagem é baseada em regras de combinação. Essas regras são compostas por padrões fonte e padrões alvo. Os padrões fonte são combinados com o modelo fonte e o padrão alvo é criado no modelo alvo, para cada uma das combinações do modelo fonte. A navegação nos modelos é realizada através de expressões OCL. A execução de uma regra em uma combinação cria automaticamente ligações de rastreabilidade. As transformações são sempre executadas de forma unidirecional.

A parte imperativa é formada por duas construções: regra invocada e bloco de ação. Uma regra invocada é chamada como uma função, através de um nome e um conjunto de argumentos. Essas regras podem conter construções declarativas referentes a padrões do modelo alvo. Um bloco de ação é uma seqüência de instruções imperativas que podem ser utilizadas pelas regras de combinação ou regras invocadas. Essas instruções imperativas da linguagem permitem es-

pecificar fluxos de controle como condições, laços, etc (JOUAULT; KURTEV, 2005). É recomendado que seja sempre utilizado o estilo declarativo, sendo o estilo imperativo utilizado somente em casos especiais nos quais as construções declarativas não oferecem os recursos necessários para atender a uma determinada necessidade.

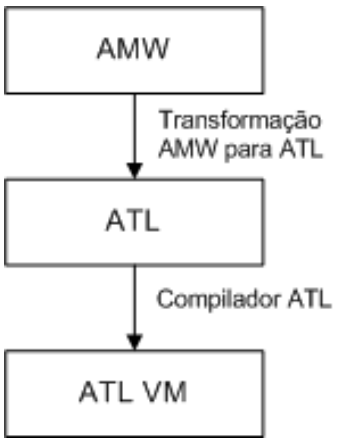

Figura 4.3: Arquitetura da linguagem ATL.

A arquitetura da linguagem é formada por três camadas: *Atlas Model Weaving* (AMW), ATL e *ATL Virtual Machine* (ATL VM), conforme apresentado na Figura 4.3 retirada de (JOUAULT; KURTEV, 2005). A ATL VM é um ambiente de execução para os programas compilados na linguagem ATL. Este ambiente de execução utiliza um conjunto de instruções orientadas a modelos. Segundo JOUAULT e KURTEV (2005, p. 3) a "AMW pode opcionalmente ser usada como uma linguagem de especificação de transformação de alto nível de abstração". A AMW, juntamente com a técnica de tecelagem de modelos, é apresentada com maiores detalhes na próxima seção.

A Figura 4.4 apresenta o contexto operacional da linguagem de transformação ATL. O objetivo é criar um programa de transformação que seja capaz de transformar um modelo fonte em um modelo alvo. O programa de transformação, representado pelo elemento *MTransf*, é baseado em dois meta-modelos, representados pelos elementos *MMFonte* e *MMAlvo*. O *MMFonte* representa o meta-modelo fonte, enquanto, o *MMAlvo* representa o meta-modelo alvo do processo de transformação de modelos. O programa de transformação deve estar em conformidade com o meta-modelo da linguagem de transformação (*MMTransf*), no caso o meta-modelo da linguagem ATL. Todos os meta-modelos devem estar em conformidade com um meta-meta-modelo específico (*M3*), como por exemplo, o MOF.

O processo de transformação de modelos ocorre quando o programa *MTransf*

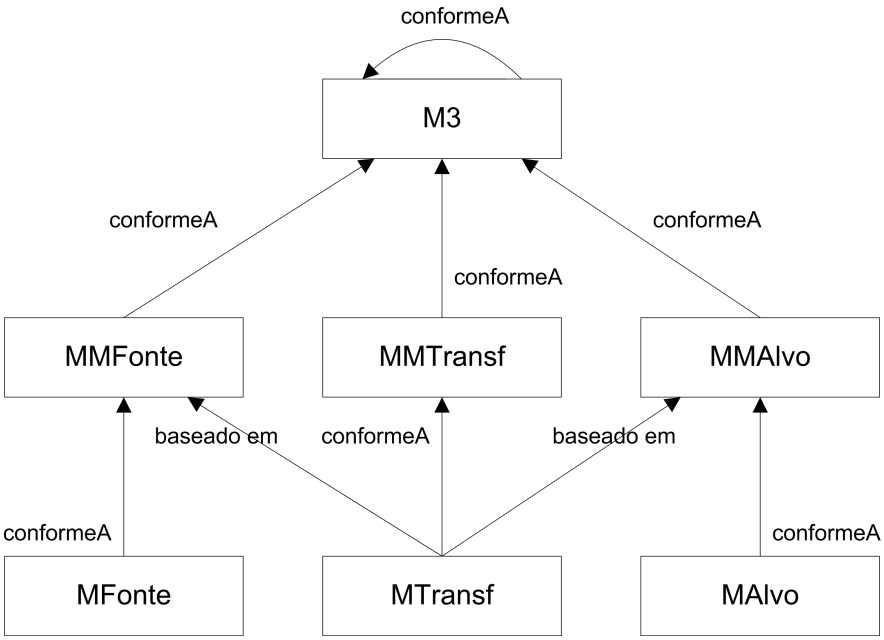

Figura 4.4: Contexto operacional da linguagem ATL.

é executado. Para isto, este programa deve ter conhecimento dos meta-modelos fonte (*MMFonte*) e alvo (*MMAlvo*) e também do modelo *MFonte* a ser transformado. O modelo *MFonte* deve estar em conformidade com o meta-modelo *MMFonte*. A execução do programa de transformação gera então o modelo *MAlvo*. O modelo *MAlvo*, deve estar em conformidade com o meta-modelo *MMAlvo*.

4.7 Tecelagem de Modelos

A técnica de tecelagem de modelos (*model weaving*) é um outro tipo de operação sobre modelos (FABRO et al., 2005). Os modelos são trançados através do estabelecimento de ligações entre os elementos dos modelos. As ligações, cujos tipos estão definidos no meta-modelo, formam um modelo que deve estar em conformidade com o meta-modelo de tecelagem.

A principal diferença entre as linguagens transformação e a tecelagem de modelos é que o meta-modelo das linguagens de transformação de modelos tem uma semântica fixa que pode ser implementada por um motor de transformação. Já os meta-modelos de tecelagem têm uma semântica definida pelo usuário. Como conseqüência, as ligações da tecelagem não podem ser geradas automaticamente, pois estão relacionadas a heurísticas e decisões de projeto (*design*) dos usuários. Em contrapartida, tais ligações podem ser salvas conjuntamente, permitindo que o modelo de tecelagem seja utilizado como entrada em ferramentas de transformação de modelos distintas, tornando-o significativamente independente da

ferramenta de transformação.

Segundo (FABRO et al., 2005, p. 2) as "transformações de modelos podem ser utilizadas para resolver alguns problemas através da especificação de traduções automáticas de uma representação para outra. A criação de um programa de transformação, entretanto, não é automática e tem um certo custo". Os autores identificaram ainda que existem três limitações da utilização somente de transformação de modelos. Essas limitações estão relacionadas à direcionalidade, à reutilização de padrões e à propagação de mudanças. Normalmente, uma transformação é executada em uma única direção (limitação de direcionalidade). Para se ter a transformação inversa é necessário escrever outro programa de transformação. Uma aplicação diferente, embora com arquitetura similar a uma outra, deve ser completamente reescrita, não havendo a reutilização de expressões ou padrões (limitação de reutilização de padrões). A maioria das modificações que ocorrem em um meta-modelo não é propagada para o outro meta-modelo (limitação da propagação de mudanças). A tecelagem de modelos pode ser utilizada para superar estas limitações, melhorando a eficiência na criação e manutenção de programas de transformação de modelos.

A Figura 4.5 apresenta o contexto operacional da tecelagem de modelos (FABRO et al., 2005). O objetivo da tecelagem é estabelecer ligações entre elementos de dois meta-modelos. Esses meta-modelos são representados pelos elementos MMEsquerda e MMDireita. O MMEsquerda representa o meta-modelo fonte, enquanto, o MMDireita representa o meta-modelo alvo do processo de transformação de modelos. As ligações estabelecidas entre estes dois meta-modelos geram o modelo da tecelagem (WM). O modelo WM deve estar em conformidade com um meta-modelo de tecelagem (WMM) específico. Todos os metamodelos devem estar em conformidade com um meta-meta-modelo específico, como por exemplo, o MOF.

Não existe um meta-modelo de tecelagem (WMM) padrão capaz de captar toda a semântica das ligações na tecelagem de modelos de vários domínios de aplicação. Cada domínio de aplicação tem necessidades diferentes, que devem ser consideradas na construção do meta-modelo de tecelagem. Entretanto, FABRO et al. (2005) afirmam que existem algumas similaridades entre os meta-modelos de tecelagem, permitindo que seja criado um núcleo genérico de meta-modelo de tecelagem. Os autores propõem, então, um meta-modelo base, contendo um conjunto mínimo de elementos necessários à tecelagem de modelos. Este meta-modelo base pode ser estendido, adicionando elementos relacionados a um domínio particular. Um meta-modelo de tecelagem pode ser expresso como extensão de outro

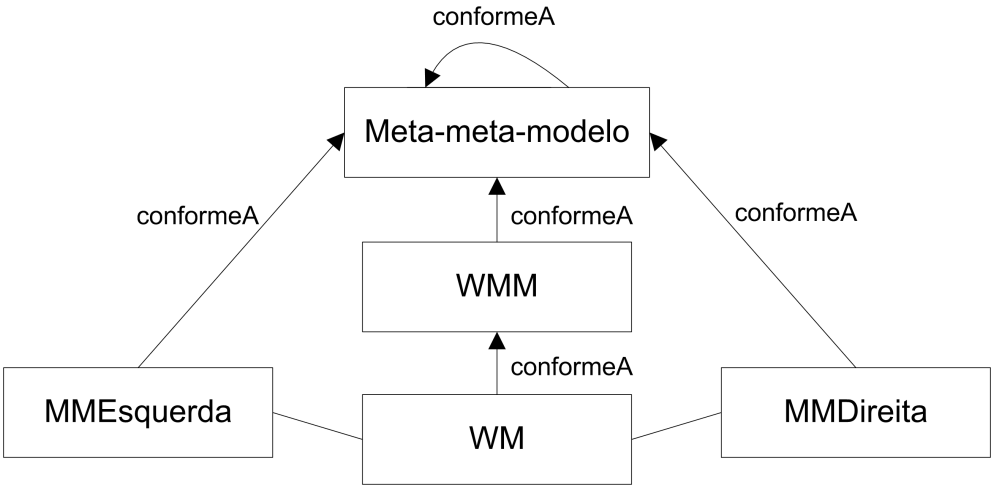

Figura 4.5: Contexto operacional da tecelagem de modelos.

meta-modelo de tecelagem. A Figura 4.6 apresenta o meta-modelo de tecelagem genérico proposto por FABRO et al. (2005) [1]. Segue uma breve descrição de cada um dos elementos do meta-modelo.

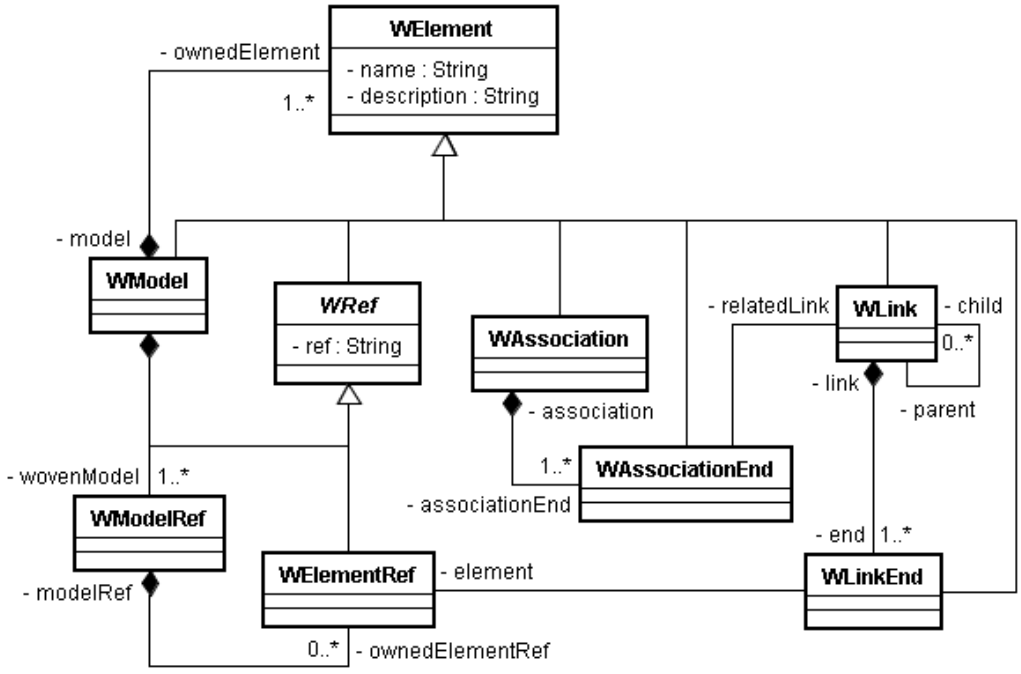

Figura 4.6: Meta-modelo de tecelagem genérico.

- *WElement*: é o elemento base de todos os elementos do meta-modelo. Todos os outros elementos o estendem. Este elemento tem dois atributos: nome (*name*) e descrição (*description*);

- *WModel*: o elemento raiz do meta-modelo de tecelagem. Este elemento

[1]Nesta figura foram mantidos os termos em inglês da especificação original.

é composto de elementos de tecelagem e referências para modelos tecidos (*woven models*);

- *WLink*: representa a ligação entre elementos de modelos. A referência *end* permite ligações entre um número arbritário de elementos. Este elemento deve ser estendido para adicionar diferentes ligações semânticas ao meta-modelo de tecelagem genérico;
- *WLinkEnd*: indica a extremidade de uma ligação, referenciando os elementos do modelo tecido através de um *WElementRef*;
- *WRef*: classe abstrata que representa as referências;
- *WElementRef*: todo elemento referenciado de um modelo de tecelagem. O atributo *ref* contém o identificador do elemento tecido. Este elemento deve ser estendido para adicionar diferentes mecanismos de identificação;
- *WModelRef*: referencia um (meta-)modelo que está sendo tecido. Este elemento permite manter um registro dos (meta-)modelos tecidos;
- *WAssociation*: utilizado para criar relacionamento de associação entre as ligações;
- *WAssociationEnd*: similar ao elemento *WLinkEnd*, especifica as extremidades de uma associação.

A Figura 4.7 apresenta o contexto operacional, utilizado no capítulo 6 do presente trabalho, servindo como guia para a utilização da técnica de tecelagem de modelos. O elemento *Meta-ModeloTecelagemBase*, representa o meta-modelo de tecelagem genérico, ou meta-modelo base, proposto por FABRO et al. (2005), acima descrito. Este elemento deve ser estendido para definir a semântica das ligações necessárias para o mapeamento de dois meta-modelos. A extensão do meta-modelo base é representada pelo elemento *Meta-ModeloTecelagemEstendido*. Um exemplo de extensão do meta-modelo genérico de tecelagem de modelo é apresentado na seção 5.4. O elemento *ModeloTecelagem* representa efetivamente as ligações de mapeamento entre dois meta-modelos, as quais geram o modelo de tecelagem. Este modelo de tecelagem deve estar em conformidade com o meta-modelo de tecelagem estendido e depende dos meta-modelos representados pelos elementos *MMEsquerda* e *MMDireita*. O elemento *MMEsquerda* representa o meta-modelo fonte da transformação, enquanto o elemento *MMDireita* representa o meta-modelo alvo.

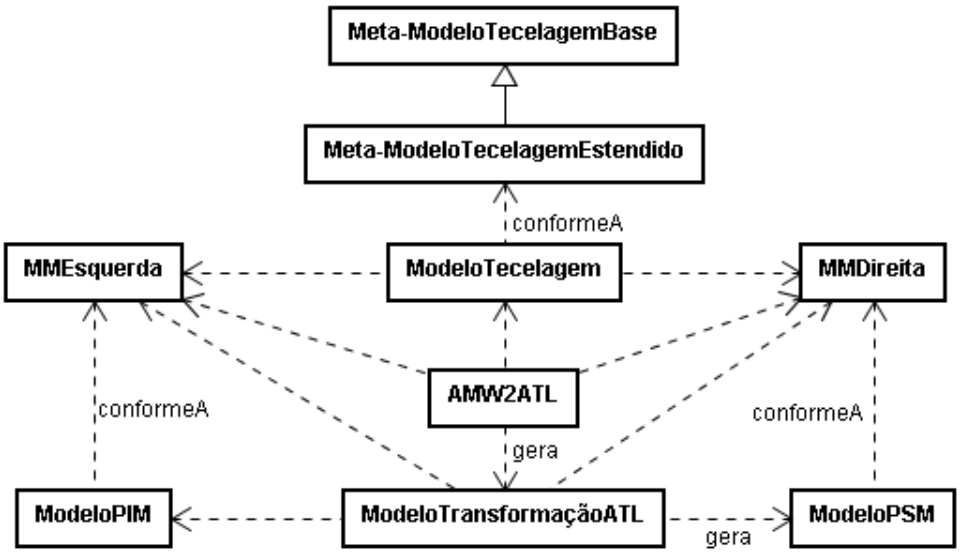

Figura 4.7: Guia da tecelagem de modelos.

O elemento *AMW2ATL* é um programa de transformação, escrito na linguagem de transformação ATL, tendo por objetivo a transformação do modelo de tecelagem representado pelo elemento *ModeloTecelagem* em um programa de transformação ATL. O programa é capaz de transformar modelos em conformidade com o meta-modelo fonte (*MMEsquerda*) em modelos em conformidade com o meta-modelo alvo (*MMDireita*). Esse tipo de programa de transformação é conhecido como transformação de ordem alta (*high order transformation - HOT*). Além do modelo de tecelagem, o programa *AMW2ATL* deve também referenciar os meta-modelos fonte e alvo da transformação. Esta transformação gera, automaticamente, um programa de transformação escrito na linguagem ATL, representado pelo elemento *ModeloTransformaçãoATL*.

O programa de transformação *ModeloTransformaçãoATL* é efetivamente responsável por transformar, automaticamente, um modelo PIM em um modelo PSM. Para isto, o programa também deve ter conhecimento dos meta-modelos fonte (*MMEsquerda*) e alvo (*MMDireita*), bem como do modelo PIM a ser transformado. O modelo PIM, representado pelo elemento *ModeloPIM* deve estar em conformidade com o meta-modelo *MMEsquerda*. A execução do programa de transformação gera então, automaticamente, o modelo PSM. O modelo PSM, representado pelo elemento *ModeloPSM*, deve estar em conformidade com o meta-modelo *MMDireita*.

5 Linguagens Específicas de Domínio

Este capítulo tem por objetivo apresentar os conceitos de linguagens específicas de domínio (*Domain Specific Language - DSL*) e a motivação para a criação deste tipo de linguagem no contexto da MDA. É apresentado também um mecanismo para a criação de DSLs a partir da UML ou de meta-modelos baseados no MOF. Este mecanismo é denominado perfil UML. Um exemplo simples é utilizado para apresentar a criação de um perfil UML para um determinado domínio de aplicação.

5.1 Introdução

A MDA propõe a criação de modelos para descrever diferentes aspectos do sistema em diferentes níveis de abstração (FRANKEL, 2003). Porém, as linguagens de modelagem, como por exemplo a UML, podem não conter todos os elementos apropriados para modelar mais precisamente determinados aspectos de um domínio específico. Nestes casos, pode ser útil a criação de linguagens específicas de domínio para descreverem, com maior precisão, os aspectos relevantes a um determinado domínio. Segundo CZARNECKI (2005) uma DSL "é uma linguagem que oferece um poder de expressividade focalizado em um domínio particular de problema, tal como uma classe específica de aplicações ou aspectos de aplicações". A criação destas DSLs, mais adequadas ao domínio do problema, facilita a comunicação entre os membros da equipe e permite também a criação de linguagens com um maior formalismo para a comunicação entre máquinas (MELLOR, 2004).

A definição do termo domínio particular de problema é um pouco vaga. Entretanto, DEURSEN, KINT e VISSER (2000) afirmam que em vez de tentar definir esse conceito, devido à sua volatilidade, é preferível enumerar e categorizar alguns domínios de problema, afim de exemplificar o contexto no qual as DSLs têm sido construídas. Algumas categorias de domínio enumeradas pelos autores são:

- Engenharia de Software: produtos financeiros, arquiteturas de software, bases de dados;
- Sistemas de Software: descrição e análise de árvores de sintaxe abstrata, estrutura de dados, protocolos;
- Multi-mídia: computação Web, manipulação de imagens, animação 3D;
- Telecomunicações: protocolos de comunicação, *switches* de telecomunicações, etc.;
- Diversos: simulação, agentes móveis, controle de robôs, etc.

As DSLs, portanto, trocam a generalidade de linguagens de propósito geral por uma expressividade em um domínio limitado, através de construções e notações destinadas a um domínio particular de aplicação (MERNIK; HEERING; SLOANA, 2005). Essas construções e notações oferecem um ganho substancial de expressividade e facilidade de uso, principalmente aos especialistas do domínio do problema, se comparadas às construções e notações oferecidas pelas linguagens de propósito geral. As DSLs permitem também a especificação do software através de um ponto de vista específico. Segundo GREENFIELD e SHORT (2004), este tipo de especificação define abstrações que codificam o vocabulário do domínio de um determinado ponto de vista. Uma DSL bem definida tende a fornecer também maior rigor em relação a linguagens de propósito geral, como a UML. Essas linguagens podem ter tanto uma notação textual como uma notação gráfica.

Em suma, os principais motivos para o uso de uma DSL são:

- As soluções são expressas em um idioma em um nível de abstração mais próximo ao domínio do problema. Conseqüentemente, os especialistas no domínio podem entender, validar e modificar os modelos gerados pelas DSLs;
- A incorporação do conhecimento relativo ao domínio, permitindo, assim, a conservação e o reuso deste conhecimento.

5.2 Desenvolvimento de uma DSL

O desenvolvimento de uma linguagem específica de domínio envolve tipicamente três etapas: análise, implementação e uso (DEURSEN; KINT; VISSER,

2000). Cada uma destas etapas é composta por um ou mais passos. Os passos descritos a seguir são baseado no trabalho de DEURSEN, KINT e VISSER (2000)

A análise tem por objetivo identificar o domínio do problema, reunir todo o conhecimento relevante para este domínio, agrupar este conhecimento em noções semânticas e operações, projetar uma DSL que concisamente descreva aplicações em um domínio.

A implementação tem por objetivo desenvolver efetivamente a nova linguagem. Após a implementação, a DSL é usada criando-se modelos que utilizem os elementos da linguagem. A figura 5.1, baseada no trabalho de (GREENFIELD; SHORT, 2004), apresenta a anatomia geral de uma linguagem. Neste diagrama, os círculos concêntricos representam a topologia arquitetural da linguagem. Eles demonstram quais partes da linguagem estão diretamente relacionadas com outras partes. A sintaxe abstrata - os conceitos essenciais e a estrutura das expressões das linguagens - está diretamente relacionada com a semântica, a qual fornece o significado dos conceitos abstratos. As sintaxes concreta e de serialização definem como os conceitos da sintaxe abstrata são realizados em uma notação concreta. A sintaxe concreta define uma notação legível ao ser humano e a sintaxe de serialização define expressões, na forma serializada, não necessariamente legíveis ao ser humano.

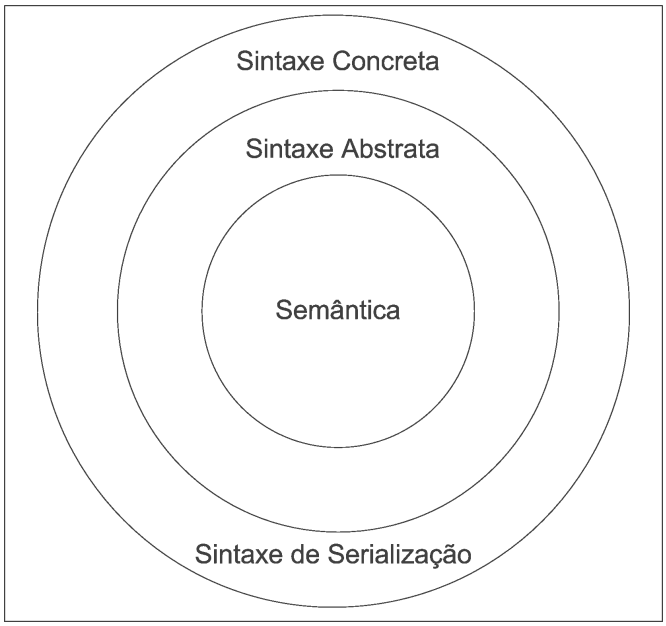

Figura 5.1: Anatomia geral de uma linguagem.

A sintaxe abstrata da linguagem, portanto, define os elementos da linguagem e a regra de composição destes elementos (FRANKEL, 2003). A sintaxe abstrata separa a estrutura da linguagem dos símbolos notacionais da linguagem, os quais são representados na sintaxe concreta.

A definição da sintaxe abstrata de uma linguagem normalmente é expressa como um meta-modelo. A seguir são apresentados dois mecanismos para a geração desses meta-modelos. O primeiro mecanismo utiliza o meta-meta-modelo MOF (OMG, 2006b). O segundo mecanismo utiliza o perfil UML (OMG, 2005c).

O meta-meta-modelo MOF, conforme apresentado na seção 3.3, é um modelo que contém um conjunto de conceitos exigidos para a criação de modelos e meta-modelos (MELLOR, 2004). As construções do MOF têm poder suficiente para representar a estrutura estática de um modelo, permitindo definir sintaxes abstratas de linguagens de modelagem. Portanto, o MOF é um mecanismo para a especificação de linguagens modelagem, através da criação de sintaxes abstratas, ou seja, de meta-modelos destas linguagens.

O MOF oferece cinco conceitos importantes que podem ser utilizados para a criação de uma linguagem (MELLOR, 2004): Tipos (classes, tipos primitivos e enumerações), Generalização, Atributos, Associações e Operações. O MOF não tem uma sintaxe concreta definida, isto é, uma forma gráfica ou textual de representar os modelos. Geralmente, estes modelos são expressos utilizando a notação da UML. Entretanto, é possível utilizar outras notações.

O perfil UML é um mecanismo de extensão da UML e de meta-modelos MOF. A seção 5.3 apresenta o perfil UML e um exemplo de construção de uma DSL utilizando este mecanismo.

Como mencionado, a semântica da linguagem define o significado da linguagem. A semântica dos elementos da linguagem construídas utilizando o mecanismo MOF ou o perfil UML pode ser definida de forma textual, por exemplo, em português, ou utilizando linguagens que expressem restrições, como a OCL (GREENFIELD; SHORT, 2004).

Para uma mesma sintaxe abstrata de uma linguagem é possível haver várias sintaxes concretas. Outro termo utilizado para denominar a sintaxe concreta é o termo notação. As notações de uma linguagem são, geralmente, textuais ou gráficas, ou uma combinação destas duas formas. Outra forma de sintaxe concreta é o que denomina-se sintaxe de serialização. Este tipo de sintaxe não têm uma forma simples de ser utilizada pelo ser humano. A sintaxe de serialização é utilizada para persistir ou intercambiar expressões da linguagem de uma forma serializada (GREENFIELD; SHORT, 2004).

5.3 Construindo uma DSL utilizando Perfil UML

Esta seção apresenta, através de um exemplo simples, a construção de uma linguagem específica de domínio utilizando o mecanismo de extensão da UML denominado perfil UML. A utilização do perfil UML permite a criação de novas linguagens de modelagem utilizando linguagens de modelagem baseadas no meta-meta-modelo MOF. Este mecanismo será utilizado no capítulo 6 para a construção de uma DSL adequada ao estudo de caso.

Um fator importante da criação de DSLs utilizando o mecanismo de perfil UML é que, segundo a definição da (OMG, 2005c), a nova linguagem preserva a semântica dos elementos do meta-modelo existente. O presente trabalho sustenta-se nesta definição, embora reconheça que se tenha aqui a necessidade de uma prova formal desta afirmação. Sendo assim, a extensão não afeta o meta-modelo e os modelos existentes antes da criação do perfil UML. Outro fator importante é que a nova linguagem criada através de um perfil UML pode ser utilizada em ferramentas que dêem suporte a este mecanismo, sem que haja a necessidade de adaptação destas ferramentas.

Segundo RUMBAUGH, JACOBSON e BOOCH (2005, p. 118), "um perfil é um pacote que identifica um subconjunto de um meta-modelo base existente, definindo estereótipos e restrições que podem ser aplicadas ao subconjunto do meta-modelo selecionado". Para a criação de um perfil são utilizados estereótipos e restrições.

Um estereótipo fornece um significado e uma utilização diferentes a um determinado elemento de modelagem (RUMBAUGH; JACOBSON; BOOCH, 2005). Ele é um elemento de modelagem baseado em elementos de modelagem existentes. A informação contida em um estereótipo é a mesma contida nos elementos base do modelo. Isto permite que ferramentas armazenem e manipulem um novo elemento do mesmo modo que faz com os elementos existentes.

Um estereótipo é definido por um nome e por um conjunto de elementos de um meta-modelo (FUENTES-FERNÁNDEZ; VALLECILLO-MORENO, 2004). Graficamente, é definido como uma classe em UML, dentro de uma caixa, com a adição do nome do estereótipo entre os símbolos $\ll \gg$. Os estereótipos estendem um determinado elemento do meta-modelo, estereotipado por ≪metaclass≫, utilizando uma seta apontando do estereótipo para a meta-classe. É possível também a criação de um ícone para um estereótipo particular, permitindo a substituição do símbolo do elemento base (RUMBAUGH; JACOBSON; BOOCH, 2005).

Antes da UML 2.0 os atributos das extensões de meta-classes, i.e. os meta-atributos, eram definidos com valores etiquetados (*tagged values*). Os valores etiquetados tinham um nome e um tipo, sendo associados a um estereótipo específico. Na UML 2.0 os estereótipos podem definir tudo o que uma classe define, incluindo atributos e associações (MELLOR, 2004).

Restrições podem ser associadas aos estereótipos restringindo os elementos correspondentes do meta-modelo. As restrições podem ser expressas em qualquer linguagem, incluindo a linguagem natural ou a *Object Constraint Language* (OCL) (OMG, 2006c). Entretanto, é recomendada a utilização de linguagens com regras de formação bem definidas.

A seguir é apresentado um exemplo simples de criação de um perfil UML. O exemplo foi adaptado do trabalho de (FUENTES-FERNÁNDEZ; VALLECILLO-MORENO, 2004). A adaptação consiste em modificar apropriadamente o perfil para atender às alterações introduzidas pela UML 2.0, uma vez que o perfil criado no trabalho mencionado é anterior à UML 2.0. O perfil UML tem por objetivo criar uma DSL para modelar as conexões entre os elementos de um sistema de informação com uma topologia em estrela. Segundo (FUENTES-FERNÁNDEZ; VALLECILLO-MORENO, 2004) a criação de um perfil UML consiste dos seguintes passos:

1. Definir um conjunto de elementos e a relação entre eles, com o objetivo de apresentar as entidades do domínio, a relação entre elas e as restrições relacionadas à estrutura e ao comportamento das entidades. Estes elementos podem ser expressos em termos de um meta-modelo;
2. Incluir um estereótipo para cada elemento relevante do meta-modelo que será incluído no perfil;
3. Associar cada estereótipo a uma meta-classe UML;
4. Definir os atributos que apareceram no meta-modelo;
5. Definir restrições de domínio ao perfil.

A figura 5.2 apresenta o meta-modelo do domínio de aplicação da topologia em estrela. Neste domínio são definidos nós (classe *Nó*), que podem estar conectados através de nós locais (associação *LigaçãoLocal*) ao nó central da estrela. É permitido também que nós centrais (classe *NóPrincipal*) conectem-se com outros nós centrais (associação *Ligação*). Cada nó é identificado por sua localização (atributo *localização* da classe *Nó*).

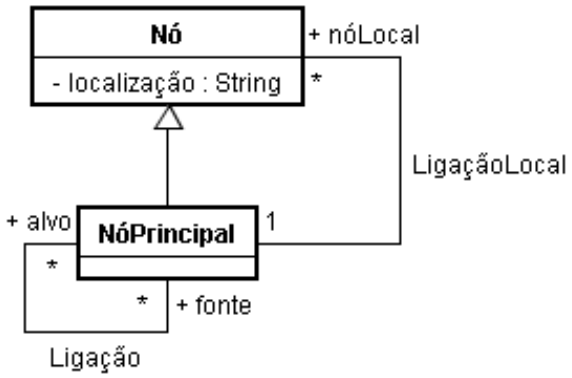

Figura 5.2: Meta-modelo de um domínio de aplicação.

A figura 5.3 apresenta o perfil UML denominado *PerfilTopologia*. O perfil é descrito como um pacote UML estereotipado com ≪profile≫. São definidos quatro estereótipos que correspondem aos elementos de modelagem do domínio do problema. Cada estereótipo criado foi associado ao elemento do meta-modelo UML que ele estende. O estereótipo *Nó* estende a meta-classe *Class* do meta-modelo da UML e contém um atributo denominado *localização*, sendo esse do tipo *String*. O estereótipo *NóPrincipal* é uma generalização do estereótipo *Nó*. Os estereótipos *Ligação* e *LigaçãoLocal* estendem a meta-classe *Association* do meta-modelo da UML. Pode-se ainda especificar as restrições ao perfil UML utilizando a linguagem OCL.

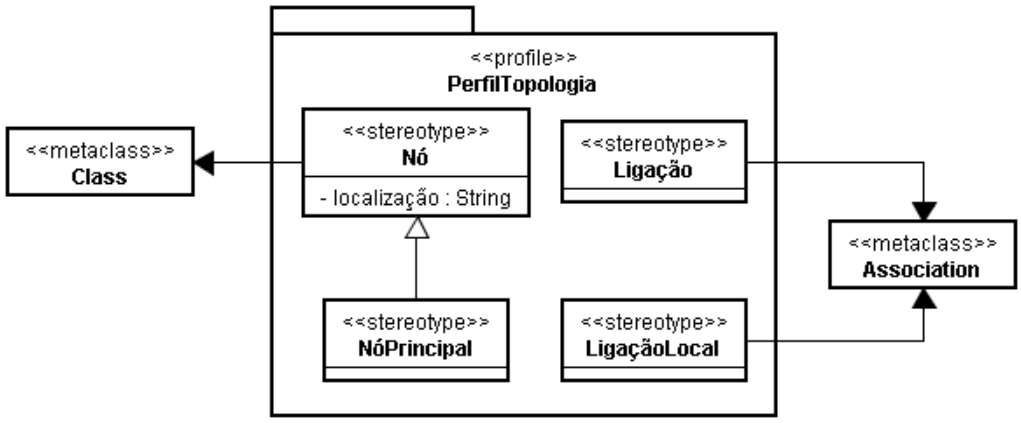

Figura 5.3: Perfil UML do domínio da aplicação.

Não existe uma forma padronizada para se especificar um perfil UML (ABOU-ZAHRA et al., 2005). Entretanto, no presente trabalho será adotada uma forma padronizada, através do auxílio de duas tabelas. O modelo das tabelas foi baseado no anexo D do documento (OMG, 2005c). A primeira tabela (tabela 5.1), responsável por descrever os elementos do perfil UML, contém os seguintes campos: Estereótipo, Classe Base, Pai, Restrições e Descrição. O campo Estereótipo apresenta o nome do estereótipo criado entre os símbolos ≪ ≫. O campo Classe Base

apresenta o nome da classe a qual o estereótipo estende. O campo Pai apresenta, caso exista, o estereótipo pai do estereótipo criado. O campo Restrições apresenta um identificador que faz referência ao campos de mesmo nome da segunda tabela. O campo Descrição apresenta uma descrição textual do estereótipo.

Tabela 5.1: Especificação do perfil UML PerfilTopologia.

Estereótipo	Classe Base	Pai	Restrições	Descrição
≪Nó≫	Class	-	Restrição 1	Representa um nó da topologia estrela. Estes nós podem estar conectados, através de ligações locais, ao central da estrela.
≪NóPrincipal≫	Class	≪Nó≫	-	Representa um nó principal da topologia estrela. Um nó principal pode estar conectado, através de ligações, a outros nós centrais.
≪Ligação≫	Association	-	-	Representa uma ligação entre dois nós centrais.
≪LigaçãoLocal≫	Association	-	-	Representa uma ligação entre um nó e um nó principal da topologia estrela.

A segunda tabela (tabela 5.2), responsável por descrever as restrições em OCL, para cada um dos elementos do perfil UML, foi criada por questões de espaço e para facilitar a leitura das expressões OCL. Esta tabela contém os seguintes campos: Restrição e OCL. O campo Restrição contém um identificador da restrição, que deve ser referenciado pela primeira tabela no campo com o mesmo nome. O campo OCL contém a expressão OCL associada ao elemento do perfil UML.

Tabela 5.2: Restrições do perfil UML PerfilTopologia.

Restrição	OCL

Restrição 1	`-- Os nós locais devem estar conectados a somente um nó principal`
	`context UML::InfrastructureLibrary::Core::Constructs::Class`
	`  inv:  self.isStereotyped(''Nó'') implies self.connection->select(`
	`  isStereotyped(''LigaçãoLocal''))-> size = 1 and`
	`  self.connection->select( isStereotyped(''Ligação''))->isEmpty()`
	
	`context UML::InfrastructureLibrary::Core::Constructs::Association`
	`  inv:  self.isStereotyped(''LigaçãoLocal'') implies`
	`  self.connection->select(participant.isStereotyped(''Nó'') or`
	`  participant.isStereotyped(''NóPrincipal''))->`
	`  forAll(n1, n2 | n1.localização = n2.localização)`
	`  inv:  self.isStereotyped(''LigaçãoLocal'') implies`
	`  self.connection->exists(`
	`  participant.isStereotyped(''NóPrincipal'') and`
	`  multiplicity.min=1 and multiplicity.max=1)`
	`  inv:  self.isStereotyped(''Ligação'') implies`
	`  self.connection->select(`
	`  participant.isStereotyped(''Nó''))->isEmpty() and`
	`  self.connection->select(`
	`  participant.isStereotyped(''NóPrincipal''))->`
	`  forAll(n1, n2 | n1.localização <> n2.localização)`

Uma vez criado o perfil UML, pode-se utilizá-lo para a criação de modelos específicos do domínio do problema. A figura 5.4 apresenta um modelo utilizando o perfil *PerfilTopologia* criado anteriormente. Este modelo define duas classes: a classe *Ramo*, estereotipada com ≪Nó≫ e a classe *ServiçoCentral*, estereotipada

com ≪NóPrincipal≫. Uma associação entre estas duas classes é criada e estereotipada com ≪LigaçãoLocal≫. Este modelo indica que um *ServiçoCentral* pode ter de um até dez ramos, sendo que todos eles têm uma única localização, estando assim de acordo com as restrições do perfil.

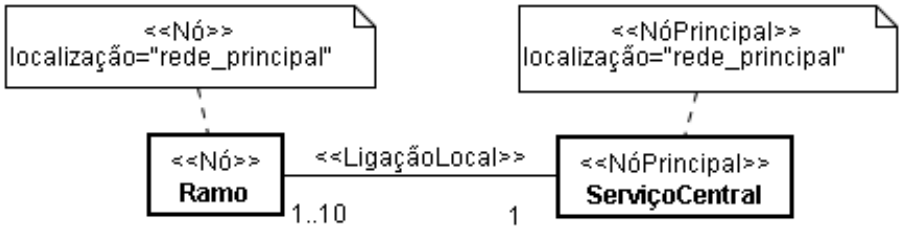

Figura 5.4: Exemplo de utilização do perfil UML criado.

Para mostrar que uma determinada aplicação é especificada utilizando um determinado perfil UML, é utilizado o relacionamento de dependência, estereotipado com ≪apply≫. A figura 5.5 apresenta uma aplicação que utiliza o perfil *PerfilTopologia* criado anteriormente.

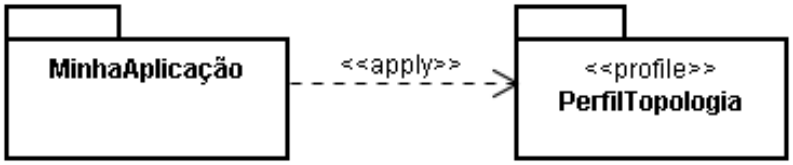

Figura 5.5: Exemplo de uma aplicação utilizando um perfil UML.

5.4 A Linguagem KM3

A linguagem KM3 (*Kernel MetaMetaModel*) é uma linguagem textual simples para a definição de meta-modelos e será utilizada no capítulo 6. Segundo (JOUAULT; BÉZIVIN, 2006), a linguagem KM3 foi definida como uma resposta às freqüentes requisições dos usuários que definem transformação de modelos utilizando a linguagem ATL. Inicialmente, os meta-modelos fonte e alvo utilizados no processo de transformação de modelos eram meta-modelos padronizados, como o meta-modelo da UML. Entretanto, a prática de transformação de modelos mostrou que durante o desenvolvimento dessas transformações muitas delas precisavam de meta-modelos específicos. Além disto, a definição desses meta-modelos é freqüentemente um processo interativo envolvendo uma elaboração progressiva. A definição de meta-modelos utilizando, por exemplo, a linguagem MOF (OMG, 2006b), apresenta o problema de, segundo (JOUAULT; BÉZIVIN, 2006), não haver um ambiente prático que dê suporte a esta linguagem.

Um dos objetivos da linguagem KM3 é facilitar a criação e modificação de meta-modelos. A linguagem tem propriedades que facilitam a leitura dos meta-modelos. O formalismo da linguagem é rico o suficiente para dar suporte a informações essenciais na criação de meta-modelos. Os meta-modelos podem ainda ser facilmente convertidos de/para outras notações, como por exemplo, a notação XMI (JOUAULT; BÉZIVIN, 2006). A linguagem KM3 contém:

- **Meta-modelo para a definição de domínios.** Este meta-modelo é, na realidade, um meta-meta-modelo e utiliza conceitos como *Classe*, *Atributo* e *Referência*. Ele está estruturalmente próximo ao MOF 2.0 (OMG, 2006b).
- **Sintaxe concreta.** A linguagem KM3 tem uma sintaxe concreta padrão na forma textual, o que permite a definição de meta-modelos utilizando qualquer editor de texto.
- **Semântica.** A semântica da linguagem KM3 permite a especificação de modelos e meta-modelos de acordo com definições apresentadas em (JOUAULT; BÉZIVIN, 2006).

Para uma idéia geral de uma descrição na linguagem KM3, apresenta-se, a seguir, o meta-modelo representado na figura 5.2 da seção anterior.

```
package Topologia {
    class Nó {
        attribute localizacao :  String;
        reference nóLocal[*] :  NóPrincipal oppositeOf nóCentral
    }
    class NóPrincipal extends Nó {
        reference fonte[*] :  NóPrincipal oppositeOf alvo;
        reference nóCentral[1] :  Nó oppositeOf nóLocal;
    }
}

package TiposPrimitivos {
    datatype String;
}
```

6 Experimentos Utilizando a Técnica de Tecelagem de Modelos

Este capítulo tem por objetivo apresentar um estudo de caso e dois experimentos nele realizados para atingir os objetivos propostos pelo presente trabalho. Observa-se, como destacado na seção 1.2, que os experimentos foram realizados sob duas restrições: os tipos de ligações do mapeamento entre meta-modelos possuem apenas a semântica de igualdade e as transformações são apenas unidirecionais.

6.1 Domínio do Problema

O domínio do problema do estudo de caso que permite a realização dos experimentos é baseado nas definições genéricas de procedimentos e atividades aplicadas a ordens de trabalho em empresas. Os elementos e a estrutura do domínio estão definidos em HAY (1996). Segundo o autor, uma organização deve fazer alguma coisa, como por exemplo, oferecer um determinado tipo de serviço. Entretanto, a maior tarefa de uma organização é planejar, escalonar e registrar as tarefas comprometidas, os procedimentos a serem seguidos e os serviços a serem feitos. No trabalho mencionado, há uma coleção de meta-modelos de vários domínios de problemas, centrados nas definições essenciais de procedimentos e atividades. Procedimento é um nome genérico referente à definição de uma série de passos que constituem uma tarefa a ser realizada. Um procedimento pode ter uma coleção de atributos, tais como descrição, duração esperada, etc. A realização de um procedimento é uma Atividade, a qual pode ter data de início, data de término, etc., como atributos. Um procedimento pode estar implementado em várias atividades. Os passos que constituem o procedimento são ocorrências de um passo de procedimento, enquanto os passos realmente realizados quando a atividade é efetuada são ocorrências de um ou mais passos de atividade. Pode haver vários

níveis de divisão de procedimentos e atividades, formando uma composição recursiva. Enquanto o trabalho a ser feito é denotado por atividades e procedimentos, o patrocínio do trabalho é denotado por uma Ordem de Trabalho.

Para esse domínio de problema será criada uma linguagem de modelagem específica do domínio (uma DSL) a ser utilizada nos experimentos, seguindo as orientações do capítulo 5. A criação desta linguagem está integralmente baseada em HAY (1996). As definições dos meta-modelos de procedimentos e atividades aplicadas às ordens de trabalho em empresas, são adaptadas para a criação da DSL. Para fins do estudo, não é necessário considerar a totalidade do domínio do problema, o qual tem a potencialidade para a criação de várias DSLs. Assim, procede-se a uma adaptação em um fragmento do domínio capaz de gerar uma única DSL que, embora mais simples, permite a realização apropriada dos experimentos. O mecanismo utilizado para a criação da linguagem é o perfil UML. Ao longo desta seção, os elementos do domínio do problema, utilizados para a criação da DSL, são apresentados juntamente com a especificação do perfil UML.

6.1.1 Arquitetura do Perfil UML

Esta seção apresenta a arquitetura do perfil UML para o domínio do problema em questão. A arquitetura deste perfil é apresentada em termos de pacotes (*packages*) UML e a relação entre cada um destes pacotes. Cada pacote UML contém termos e conceitos associados ao fragmento considerado do domínio do problema. A divisão dos pacotes baseou-se na descrição do domínio do problema proposta por HAY (1996).

A figura 6.1 apresenta os pacotes contidos no perfil UML denominado *OrdemServiço* e o relacionamento entre os pacotes. Os pacotes são os seguintes: *OrdemTrabalho*, *Participante*, *Ativos* e *Util*. O pacote *OrdemTrabalho* agrupa elementos relativos a procedimentos, atividades e ordem de trabalho. O pacote *Participante* agrupa elementos relativos aos participantes de uma ordem de trabalho. O pacote *Ativos* agrupa elementos relativos aos materiais utilizados em uma ordem de trabalho. O pacote *Util* tem por característica agrupar elementos comuns a todo perfil.

As próximas seções apresentam os conceitos associados a cada um dos pacotes e especificam os elementos de perfil UML contidos nestes pacotes.

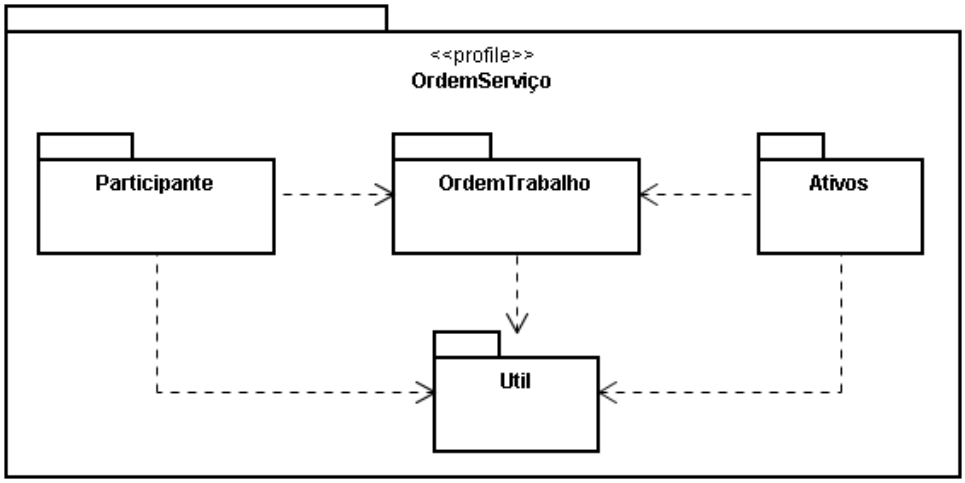

Figura 6.1: Arquitetura de pacotes do perfil UML *OrdemServiço*.

6.1.2 Pacote *OrdemTrabalho*

O pacote *OrdemTrabalho* pertence ao perfil UML *OrdemServiço* e agrupa elementos relativos a procedimentos, atividades e ordem de trabalho. A figura 6.2 apresenta um diagrama de classe que esboça os estereótipos contidos neste pacote. Todos os estereótipos contidos neste pacote estendem a meta-classe *Class* do meta-modelo da UML.

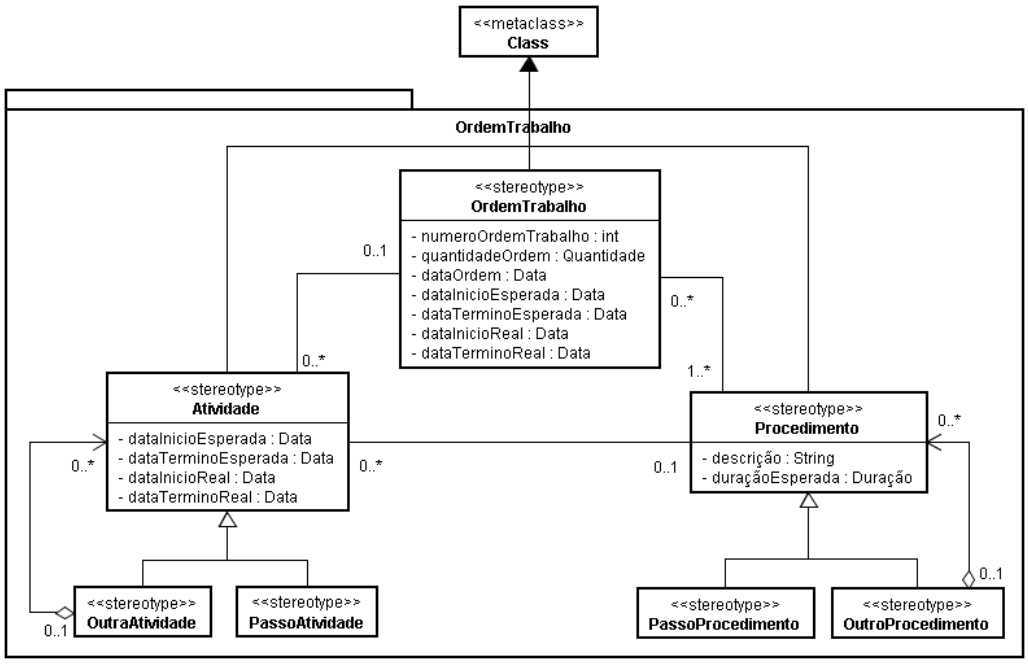

Figura 6.2: Pacote *OrdemTrabalho* do perfil UML *OrdemServiço*.

Os estereótipos *OrdemTrabalho*, *Procedimento* e *Atividade*, denotam os conceitos de ordem de trabalho, procedimentos e atividades mencionados na seção 6.1. Os estereótipos *Atividade* e *Procedimento* são, neste trabalho, tratados como

elementos primitivos da linguagem DSL criada, uma vez que a proposta de HAY (1996) para estes elementos é muito mais elaborada. A aplicação dos estereótipos *Atividade* e *Procedimento* permite a definição de tipos em um domínio de problema cujo universo de discurso fale sobre ordens de trabalho e, conseqüentemente, de atividades e procedimentos. Para modelar a composição recursiva de procedimentos e atividades, os estereótipos *Atividade* e *Procedimento* seguem o padrão de projeto Compósito (Composite) (GAMMA et al., 1995). Os estereótipos criados para modelar a composição recursiva de atividades são: *PassoAtividade* e *OutraAtividade*, sendo este último composto por zero ou mais atividades. Os estereótipos criados para modelar a composição recursiva de procedimentos são: *PassoProcedimento* e *OutroProcedimento*, sendo esse último composto por zero ou mais procedimentos. O estereótipo *OrdemTrabalho* contém os seguintes atributos: *numeroOrdemTrabalho*, o qual deve ser único no sistema, *quantidade* a ser produzida, *dataOrdem*, *dataInicioEsperada*, *dataTerminoEsperada*, *dataInicioReal* e *dataTerminoReal*. O estereótipo *Procedimento* tem como atributos uma *descrição* e a *duraçãoEsperada* do procedimento. O estereótipo *Atividade* tem como atributos *dataInicioEsperada*, *dataTerminoEsperada*, *dataInicioReal* e *dataTerminoReal*.

A tabela 6.1 apresenta a definição do pacote *OrdemTrabalho* do perfil UML *OrdemDeTrabalho* de acordo com a estrutura apresentada na seção 5.3.

Tabela 6.1: Especificação do perfil UML *OrdemServiço* - Pacote *OrdemTrabalho*.

Estereótipo	Classe Base	Pai	Restrições	Descrição
≪OrdemTrabalho≫	Class	-	-	Representa o conceito de ordem de trabalho.
≪Atividade≫	Class	-	-	Classe que representa o conceito de atividade no contexto de uma ordem de trabalho.
≪PassoAtividade≫	Class	≪Atividade≫	-	Representa um passo de atividade em uma ordem de trabalho.
≪OutraAtividade≫	Class	-	-	Representa um outro tipo de atividade que pode ser executada em uma ordem de trabalho.
≪Procedimento≫	Class	-	-	Classe que representa o conceito de procedimento no contexto de uma ordem de trabalho.
≪PassoProcedimento≫	Class	≪Procedimento≫	-	Representa um passo de procedimento a ser realizado em uma ordem de trabalho.
≪OutroProcedimento≫	Class	-	-	Representa um outro tipo de procedimento que pode ser realizado em uma ordem de trabalho.

6.1.3 Pacote *Participante*

O pacote *Participante* pertence ao perfil UML *OrdemServiço* e agrupa elementos relativos aos participantes de uma ordem de trabalho. Uma ordem de trabalho deve ser de responsabilidade de alguém, geralmente de uma pessoa, mas pode ser também de uma organização. A ordem de trabalho é, provavelmente, preparada por uma pessoa. A figura 6.3 apresenta o pacote *Participante* do perfil UML *OrdemServiço* contendo os elementos relativos aos conceitos mencionados. Todos os estereótipos deste pacote estendem a meta-classe *Class* do meta-modelo da UML.

O estereótipo *Grupo* reúne características comuns aos seus sub-tipos. Ele tem atributos como *nome*, *endereco*, *custo*, etc. Os sub-tipos deste estereótipo aqui modelados são *Pessoa* e *Organização*. Um *Grupo* pode ser responsável por uma ou mais ordens de trabalho, sendo que uma *OrdemTrabalho* deve ser de responsabilidade de um *Grupo*. Uma *Pessoa* pode ser responsável por uma ou mais ordens de trabalho. Uma *OrdemTrabalho* pode ter sido preparado por uma *Pessoa*.

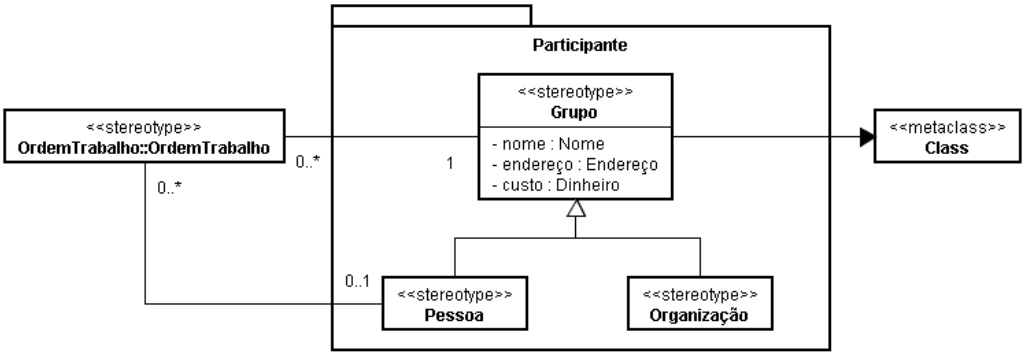

Figura 6.3: Pacote *Participante* do perfil UML *OrdemServiço*.

A tabela 6.2 apresenta a definição do pacote *Participante* do perfil UML *OrdemServiço* de acordo com a estrutura apresentada na seção 5.3.

Tabela 6.2: Especificação do perfil UML *OrdemServiço* - Pacote *Participante*.

Estereótipo	Classe Base	Pai	Restrições	Descrição
≪Grupo≫	Class	-	-	Agrupa as características comuns de pessoa e organização.
≪Pessoa≫	Class	≪Grupo≫	-	Representa uma pessoa no contexto do domínio de ordem de trabalho.
≪Organização≫	Class	≪Grupo≫	-	Representa uma organização no contexto do domínio de ordem de trabalho.

6.1.4 Pacote *Ativos*

O pacote *Ativos* pertence ao perfil UML *OrdemServiço* e agrupa elementos relativos aos materiais utilizados em uma ordem de trabalho. Os estereótipos contidos neste pacote são: *Ativo* e *TipoAtivo*. Uma ordem de trabalho deve fazer um *TipoAtivo*, sendo que um *TipoAtivo* pode ser feito por uma ou mais *OrdemTrabalho*. Um *TipoAtivo* tem como atributo o *custoPadrão* do tipo do ativo. Um *Ativo* é um exemplo de um tipo de ativo. Um *TipoAtivo* pode ser associado a um *Ativo*. Um *Ativo* tem como atributo o *custoUnitário* do ativo. A figura 6.4 apresenta um diagrama de classe que esboça os estereótipos contidos neste pacote. Todos os estereótipos contidos neste pacote estendem a meta-classe *Class* do meta-modelo da UML.

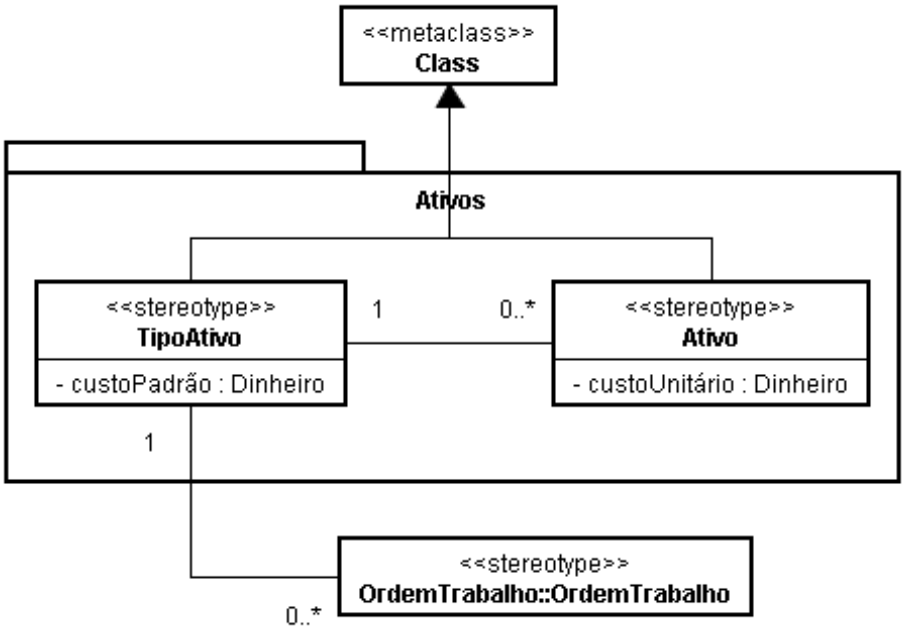

Figura 6.4: Pacote *Ativos* do perfil UML *OrdemServiço*.

A tabela 6.3 apresenta a definição do pacote *Ativos* do perfil UML *OrdemServiço* de acordo com a estrutura apresentada na seção 5.3.

Tabela 6.3: Especificação do perfil UML *OrdemServiço* - Pacote *Ativos*.

Estereótipo	Classe Base	Pai	Restrições	Descrição
≪TipoAtivo≫	Class	-	-	Representa o tipo de um ativo feito por uma ordem de trabalho.
≪Ativo≫	Class	-	-	Representa um exemplo de um tipo de ativo.

6.1.5 Pacote *Util*

O pacote *Util* pertence ao perfil UML *OrdemServiço* e agrupa elementos comuns a todo perfil. Estes elementos são utilizadas pelos outros pacotes do perfil para, por exemplo, especificar o tipo de atributos dos estereótipos. A figura 6.5 apresenta um diagrama de classe que esboça os estereótipos contidos neste pacote. Todos os estereótipos contidos neste pacote estendem a meta-classe *Property* do meta-modelo da UML. O elemento *Property* do meta-modelo da UML pode representar um atributo de um classificador. Sendo assim, ele relaciona uma instância da classe a um valor ou coleção de valores do tipo do atributo (OMG, 2005c).

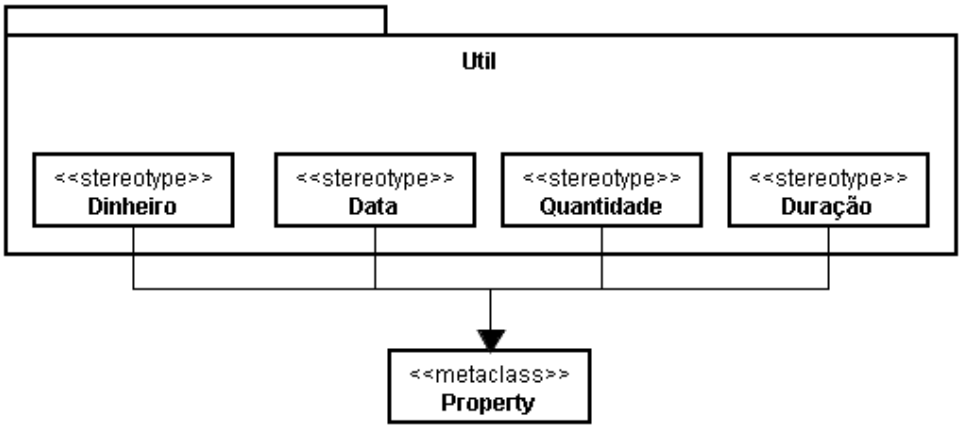

Figura 6.5: Pacote *Util*do perfil UML *OrdemServiço*.

A tabela 6.4 apresenta a definição do pacote *Util* do perfil UML *OrdemDe-Trabalho* de acordo com a estrutura apresentada na seção 5.3.

Tabela 6.4: Especificação do perfil UML *OrdemServiço* - Pacote *Util*.

Estereótipo	Classe Base	Pai	Restrições	Descrição
≪Dinheiro≫	Property	-	-	Representa uma unidade monetária.
≪Data≫	Property	-	-	Representa uma data.
≪Quantidade≫	Property	-	-	Representa a quantidade de um determinado item.
≪Duração≫	Property	-	-	Representa a duração de um determinado procedimento ou atividade.

6.2 Modelo PIM do Domínio do Problema

Esta seção apresenta o modelo do domínio do problema, caracterizado como um modelo PIM, o qual é utilizado nas transformações de modelos ao longo dos experimentos realizados no presente trabalho. O domínio do problema a ser modelado é relativo a ordens de trabalho no contexto de manufatura. Uma ordem

de trabalho neste contexto é denominada ordem de produção. A construção do modelo PIM é derivada do meta-modelo representado pelo perfil UML *OrdemServiço*. No modelo, serão desconsideradas as operações e especializações de classes por razões de simplicidade.

A figura 6.6 apresenta o modelo PIM de ordem de produção. Todas as classes deste modelo, exceto a classe *EntregaProdução*, são derivadas do meta-modelo representado pelo perfil UML *OrdemServiço*, aplicadas ao domínio específico de manufatura. Como a classe *EntregaProdução* é utilizada somente no modelo PIM, no contexto particular da manufatura, ela não é parte do meta-modelo do domínio genérico do problema. Portanto, cada classe do modelo tem a semântica associada a um estereótipo do perfil UML. As classes estereotipadas herdam os atributos dos estereótipos aos quais estão associadas. Porém, é possível adicionar novos atributos à classe estereotipada. Os atributos pertencentes à meta-classe foram aqui reproduzidos para facilitar a compreensão do modelo.

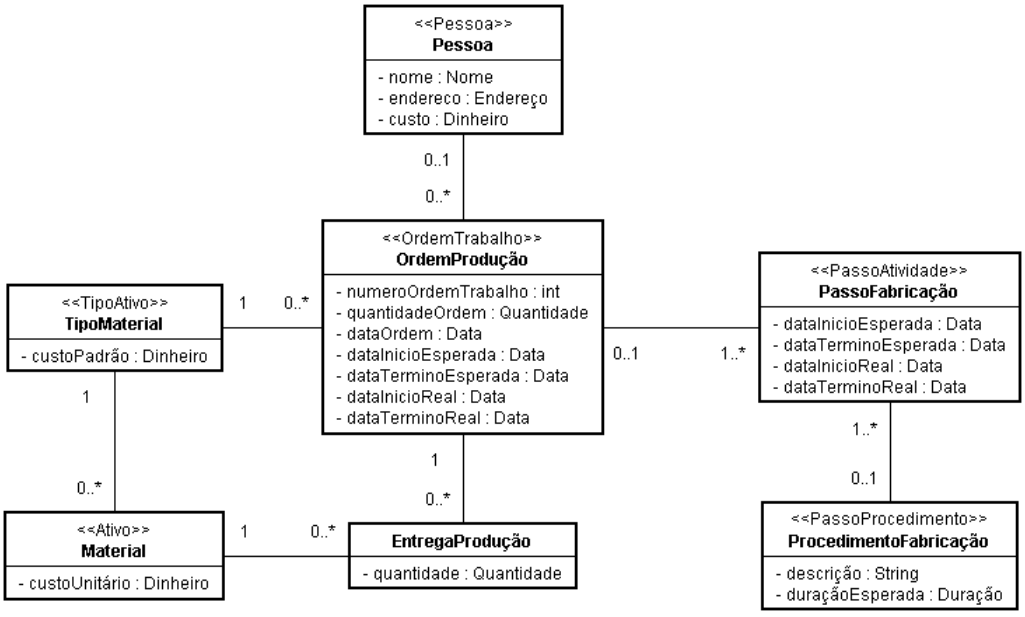

Figura 6.6: Modelo PIM de Ordem de Produção.

A classe *OrdemProdução* denota uma ordem de trabalho no contexto da manufatura, responsável por programar a produção de ativos, no caso, de materiais. A classe é estereotipada com $\ll OrdemTrabalho \gg$ e contém os seguintes atributos: *numero* que deve ser único no sistema, *quantidade* a ser produzida, *dataOrdem*, *dataInicioEsperada*, *dataTerminoEsperada*, *dataInicioReal* e *dataTerminoReal*. A classe *PassoFabricação* denota os passos da atividade de fabricação de um determinado material. Por razões de simplicidade do modelo do domínio em questão para o presente trabalho, neste fragmento denotam-se apenas os passos de atividades de fabricação. Esta classe é estereotipada por $\ll PassoAtividade \gg$. Do

mesmo modo, os passos de fabricação devem realizar passos de procedimentos pré-definidos. Assim, a classe *ProcedimentoFabricação*, estereotipada por ≪*PassoProcedimento*≫, representa um passo de procedimento. Esta classe tem como atributos uma *descrição* e a *duraçãoEsperada* do procedimento. A classe *EntregaProducao* denota o material que deve ser entregue pela produção de uma ordem de produção. Esta classe é utilizada somente no contexto da manufatura, por isto, não contém nenhum estereótipo associado, como dito anteriormente. A classe tem como atributo a *quantidade* do material a ser entregue. As classes, *TipoMaterial* e *Material* estão associadas ao estereótipos *TipoAtivo* e *Ativo* respectivamente, e a classe *Pessoa* está associada ao estereótipo de mesmo nome. Elas têm a mesma denotação do perfil UML *OrdemServiço*. Todos os atributos destas três classes foram obtidos das meta-classes associadas a elas pelos seus respectivos estereótipos.

6.3 Ferramentas Utilizadas nos Experimentos

A plataforma *ATLAS Model Management Architecture* (AMMA) é uma plataforma de gerenciamento de modelos, baseada na plataforma Eclipse [1], composta de uma série de componentes, dentre os quais: a linguagem ATL e a ferramenta *ATLAS Model Weaving* (AMW) (FABRO et al., 2005). A ATL, conforme mencionado anteriormente, é uma linguagem para a especificação de transformação de modelos. A plataforma AMMA fornece um ambiente para o desenvolvimento e execução de transformações ATL. A AMW é uma ferramenta utilizada para o desenvolvimento de modelos de tecelagem, fornecendo um mecanismo gráfico para a importação de meta-modelos e o estabelecimento de ligações entre dois meta-modelos. Um meta-modelo de tecelagem estendido pode também ser importado pela ferramenta. As ferramentas ATL e AMW são utilizadas como o ambiente de execução dos experimentos descritos adiante.

A versão atual das ferramentas ATL e AMW não têm suporte nativo ao mecanismo de perfil UML da versão 2.0 da linguagem. Entretanto, estas ferramentas dão suporte à linguagem KM3 (JOUAULT; BÉZIVIN, 2006). Devido ao problema de incompatibilidade por parte da ferramenta, a DSL *OrdemServiço*, especificada utilizando o mecanismo de perfil UML, precisa ser especificada na linguagem KM3, para que os experimentos propostos no trabalho possam ser realizados. Mesmo tendo conhecimento das limitações das ferramentas, foi mantida a decisão de especificar a DSL utilizando o mecanismo de perfil UML devido às

[1] Disponível em: <http://www.eclipse.org>. Acesso em: 08 fev. 2007.

vantagens oferecidas por ele, conforme apresentado na seção 5.3. Outro aspecto para manter a decisão de utilização do mecanismo de perfil UML é o fato dessas ferramentas, no futuro, poderem dar suporte ao perfil UML da versão 2.0 da linguagem.

O perfil UML *OrdemServiço* é utilizado para auxiliar na especificação da DSL em linguagem KM3. Os passos para especificar os elementos do perfil UML *OrdemServiço* em linguagem KM3 são os seguintes:

- Selecionar os estereótipos do perfil UML *OrdemServiço* que serão especificados em linguagem KM3;
- Criar uma classe abstrata em linguagem KM3 denominada *Nomeado*, a qual contém o atributo *nome*, sendo este do tipo *String*;
- Para cada estereótipo selecionado, gerar uma classe em linguagem KM3, com o mesmo nome do estereótipo. Na geração desta classe, considerar os atributos e relacionamentos associados ao estereótipo mapeado. Somente são considerados os relacionamentos para as classes que serão efetivamente mapeadas. Os demais relacionamentos são desconsiderados;
- Toda a classe gerada a partir de um estereótipo deve estender a classe abstrata *Nomeado*.

A figura 6.7 apresenta um exemplo da especificação (parcial), na linguagem KM3, do estereótipo *OrdemTrabalho* do perfil UML *OrdemServiço*. O estereótipo *OrdemTrabalho* é especificado em uma classe da linguagem KM3 com o mesmo nome do estereótipo. Esta classe estende a classe abstrata *Nomeado*. Todas as classes devem estar contidas em um pacote (*package*), no caso, denominado *OrdemServiço*. A classe *EntregaProducao*, mesmo não pertencendo ao perfil UML *OrdemServiço*, foi especificada na versão em linguagem KM3 da DSL, para permitir criar o elemento *EntregaProducao* no modelo PIM (ver Figura 6.6). Na especificação da DSL em linguagem KM3, todos os caracteres especiais, como acentos, foram retirados dos nomes das classes e atributos, pois a linguagem não oferece suporte a caracteres especiais. A especificação completa da DSL *OrdemServiço*, em linguagem KM3, é apresentada no Apêndice A.

O modelo PIM, especificado utilizando o perfil UML *OrdemServiço* (ver seção 6.2), também precisa ser especificado em um formato que as ferramentas ATL e AMW dêem suporte. O formato selecionado para a especificação do modelo é o XMI (ver seção 3.4), uma vez que as ferramentas dão suporte a este formato. Este

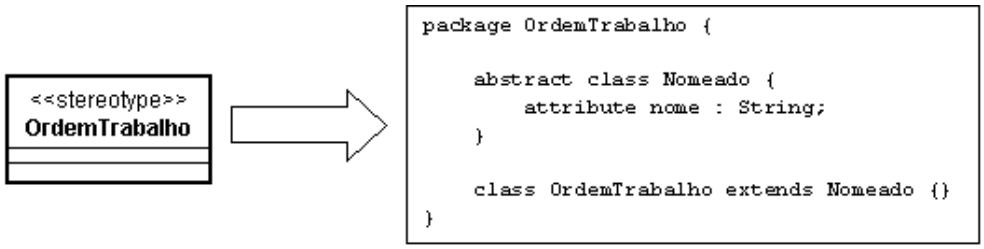

Figura 6.7: Exemplo da especificação da DSL *OrdemServiço* em linguagem KM3.

modelo deve estar em conformidade com a DSL *OrdemServiço* especificada utilizando a linguagem KM3, anteriormente criada. A figura 6.8 apresenta uma classe especificada em UML e estereotipada por um elemento do perfil UML *OrdemServiço*, e a sua especificação no formato XMI. Cada etiqueta da representação XMI, por exemplo a etiqueta *OrdemTrabalho*, representa uma classe do modelo PIM associada ao estereótipo de mesmo nome da etiqueta XMI. Portanto, o nome da etiqueta deve ser o mesmo do estereótipo utilizado no diagrama de classes. O atributo *nome* do elemento XMI, contém o nome da classe estereotipada, por exemplo, a classe *OrdemProdução*. Os atributos da classe do modelo PIM são representados como atributos dos elementos XMI. O nome de cada atributo do XMI deve ser o mesmo nome do atributo da classe respectivamente. Cada atributo XMI é iniciado com um valor apropriado. A especificação completa do modelo PIM, representado na figura 6.6, em formato XMI é apresentada no Apêndice B.

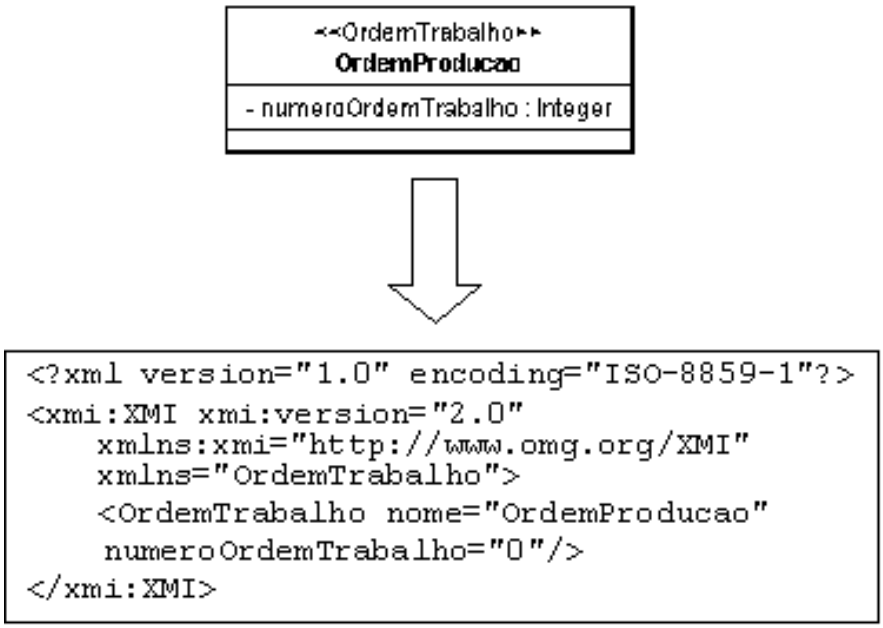

Figura 6.8: Exemplo da descrição do modelo PIM em XMI.

Como mencionado anteriormente, os experimentos consideram apenas a semântica de igualdade de classes, isto é, consideram apenas que os elementos vinculados

representam a mesma informação. No caso, apenas a igualdade que considera a classe como elemento fonte.

e não as semânticas de atributos, de relacionamentos e de tipos. Devido a isto, o modelo PIM especificado em formato XMI só descreve as classes do modelo, não considerando seus atributos e relacionamentos. O fragmento de código abaixo, apresenta a especificação em formato XMI da classe *OrdemProdução* considerando a restrição mencionada.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xmi:XMI xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI"
    xmlns="OrdemTrabalho">
  <OrdemTrabalho nome="OrdemProducao"/>
</xmi:XMI>
```

6.4 Experimento 1

Esta seção apresenta os objetivos, a descrição e execução, e os resultados do primeiro experimento proposto.

6.4.1 Objetivo

Este primeiro experimento tem por objetivo investigar as vantagens que a técnica de tecelagem de modelos oferece em relação à reutilização de trechos de código manualmente escritos para a geração de especificações de transformação. Compara-se a técnica de tecelagem de modelos com um enfoque de geração de especificação de transformação de modelos utilizando apenas uma linguagem de transformação de modelos. Analisa-se o aspecto da redução da duplicação de código escrito manualmente, comparando, do ponto de vista quantitativo, as duas técnicas de especificação de modelos de transformação mencionadas. A cobertura deste primeiro experimento está associada somente ao primeiro objetivo proposto no presente trabalho. Conforme apresentado na seção 6.3, as ferramentas utilizadas para este experimento são a AMW e a ATL.

6.4.2 Descrição do Experimento

O experimento consiste em transformar o modelo PIM, criado na seção 6.2, em um modelo PSM associado a uma plataforma de banco de dados relacional. O experimento está dividido em duas etapas. A primeira etapa é a geração

da especificação de transformação utilizando apenas a linguagem ATL. Isto é, nesta etapa a técnica de tecelagem de modelos não é utilizada. A segunda etapa é a geração da especificação de transformação de modelos utilizando a técnica de tecelagem de modelos. Nesta, a ferramenta AMW e a linguagem ATL são utilizadas conjuntamente.

A figura 6.6 apresentou o modelo PIM utilizado como entrada para o processo de transformação das duas etapas do experimento. Este modelo é um fragmento do modelo de domínio genérico de ordens de produção descrito na seção 6.2. Como mencionada, por questões de simplicidade, as especializações de classes foram excluídas do experimento. Conforme apresentado na seção 6.3, o modelo PIM utilizado no experimento, descrito na linguagem UML, deve ser especificado em formato XMI e deve estar em conformidade com a DSL *OrdemServiço* especificada na linguagem KM3. O Apêndice A apresenta o meta-modelo fonte, especificado em linguagem KM3 utilizado neste experimento. O Apêndice B apresenta o modelo de entrada do processo de transformação (Modelo PIM), especificado no formato XMI, utilizado neste experimento.

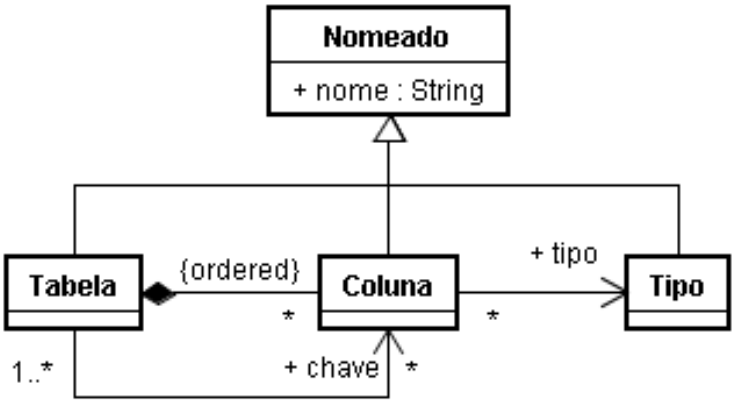

Figura 6.9: Meta-modelo de um banco de dados relacional.

A plataforma na qual o modelo PIM será transformado é uma plataforma de banco de dados relacionais. A Figura 6.9 apresenta o diagrama do meta-modelo que descreve os elementos relativos a um banco de dados relacional (JOUAULT; KURTEV, 2005). O meta-modelo é composto pelos seguintes elementos: *Nomeado*, *Tabela*, *Coluna* e *Tipo*. O elemento *Nomeado* é um elemento abstrato, contendo um atributo do tipo *String* denominado *nome*, e indica que todas as suas extensões devem possuir um nome. O elemento *Tabela* representa uma tabela de um banco de dados relacional. Uma tabela é composta por colunas, representada pelo elemento *Coluna*. As colunas de uma tabela devem ser ordenadas. Uma tabela utiliza colunas para compor chaves. Uma coluna contém um tipo, repre-

sentada pelo elemento *Tipo*. Os tipos de um banco de dados relacional não são apresentados nesta figura por questões de simplicidade. O modelo PSM gerado pela transformação deve estar em conformidade com este meta-modelo. Para o experimento do presente trabalho, este meta-modelo foi definido na linguagem KM3. A decisão de utilização desta linguagem é facilitar a utilização do meta-modelo na ferramenta de transformação, uma vez que as ferramentas AMW e ATL têm suporte nativo para a linguagem KM3. A definição do meta-modelo foi baseado no trabalho de (JOUAULT; KURTEV, 2005) e sua especificação é apresentada no Apêndice C.

6.4.2.1 Etapa 1 - Especificação de transformação de modelos utilizando a linguagem ATL

O objetivo desta etapa é escrever uma especificação de transformação de modelos, em linguagem ATL, para transformar o modelo PIM para uma plataforma de banco de dados relacional. Assume-se que cada classe do modelo PIM deve ser transformada em uma tabela do banco de dados. Os atributos da classe devem se tornar, respectivamente, colunas das tabelas criadas. Para cada tabela criada deve ser criada uma coluna, denominada *idObjeto*, representando o elemento chave da tabela.

Para realizar esta transformação, regras de transformação na linguagem ATL são criadas. Para cada tipo do modelo PIM é necessário criar uma regra especifica de transformação, seja ela para as classes ou atributos do modelo. Como cada classe do modelo representa um tipo diferente do domínio do problema utilizado, há a necessidade de especificar uma regra de transformação para cada uma delas. O mesmo acontece com os diferentes tipos de atributos presentes no modelo. O código a seguir, apresenta um fragmento da especificação de transformação escrita para esta etapa do experimento. A especificação da transformação completa está descrita no Apêndice D.

Uma especificação de transformação na linguagem ATL inicia-se com a criação do comando *module* que contém o nome da especificação de transformação (*OrdemTrabalho2BancoDados*), o meta-modelo fonte e o meta-modelo alvo da transformação. O meta-modelo fonte é referenciado por *from* e o meta-modelo alvo é referenciado por *create*. No exemplo, as variáveis que armazenam a referência para os meta-modelos fonte e alvo são *IN* e *OUT* respectivamente (linha 2).

```
1.  module OrdemTrabalho2BancoDados;
2.  create OUT : Relacional from IN : OrdemTrabalho;
```

```
3.  rule OrdemTrabalho2Tabela {
4.     from
5.        ot :  OrdemTrabalho!OrdemTrabalho
6.     to
7.        saidaTabela :  Relacional!Tabela (
8.           nome <- ot.nome,
9.           col <- Sequence{chave},
10.         chaveTabela <- Set{chave}
11.       ),
12.       chave :  Relacional!Coluna (
13.          nome <- 'idObjeto'
14.       )
15. }
```

A partir da linha 3, inicia-se a criação de uma regra ATL com o objetivo de transformar um tipo *OrdemTrabalho* do modelo PIM em uma tabela de banco de dados relacional. A especificação desta regra inicia-se com o comando *rule*, seguido pelo nome da regra. Uma regra deve obter os elementos fonte da transformação, especificados no comando *from*, e criar os elementos alvo, especificados no comando *to*. No comando *from* associa-se o elemento *OrdemTrabalho* do meta-modelo de entrada *OrdemServiço* com uma variável denominada *ot*. No comando *to*, inicialmente (linha 7) especifica-se que um elemento *Tabela* do meta-modelo *Relacional* deve ser criado e deve ser associado à variável *saidaTabela*. Posteriormente (linha 12), especifica-se que um elemento *Coluna* do meta-modelo *Relacional* deve ser criado e associado à variável *chave*.

Os passos para a criação de uma tabela são especificados nas linhas de 8 a 12. Primeiramente, indica-se que o nome da tabela deve ter o mesmo nome do elemento do tipo *OrdemTrabalho* do modelo fonte. Posteriormente, cria-se a coluna chave da tabela, adicionando o valor da variável *chaveTabela* a uma lista de colunas representada pela variável *col*. O valor da variável *chaveTabela* é um conjunto que, no caso, contém o valor da variável *chave*. A variável *chave* está associada a uma coluna de nome *idObjeto*. O comando *Set* (linha 10) cria um conjunto de elementos, especificados entre os símbolos { }. Para o elemento *chaveTabela* foi necessário criar um conjunto para respeitar o meta-modelo da banco de dados relacional, o qual indica que uma tabela pode composta por uma ou mais colunas-chave. No caso da variável *col*, o comando utilizado foi o *Sequence*, uma vez que o meta-modelo indica que uma tabela é composta por um conjunto de colunas ordenadas.

O resultado da execução da especificação de transformação de modelo gerada é um arquivo, em formato XMI, contendo a descrição do modelo alvo. O código a seguir apresenta um fragmento desta descrição. O fragmento apresenta a descrição de uma tabela do modelo PSM gerado, a qual foi gerada a partir da classe

OrdemProdução associada ao estereótipo *OrdemTrabalho*. O elemento *Tabela* do modelo PSM contém um atributo *nome*, iniciado com o valor *OrdemProducao*, representando o nome da tabela e um atributo *chave*, contendo uma referência para a coluna chave da tabela. Esta tabela contém ainda uma coluna, representada pelo elemento *col*, denominada *idObjeto*. A descrição completa do modelo PSM gerado é apresentada no Apêndice E.

```
<Table nome=''OrdemProducao'' chave=''/1/@col.0''>
   <col nome=''idObjeto'' chaveDe=''/1''/>
</Table>
```

6.4.2.2 Etapa 2 - Especificação de transformação de modelos utilizando a tecelagem de modelos

O objetivo desta etapa é especificar a transformação de modelos, utilizando a técnica de tecelagem de modelos, para transformar o modelo PIM para uma plataforma de banco de dados relacional. Cada classe do modelo PIM deve ser transformada em uma tabela do banco de dados. Os atributos da classe devem se tornar, respectivamente, colunas das tabelas criadas. Para cada tabela criada deve ser criada uma coluna, denominada *idObjeto*, representando o elemento chave da tabela. Ao final do processo de transformação de modelos o resultado obtido deve ser o mesmo da etapa anterior, esperando-se uma redução na duplicação de código produzido manualmente.

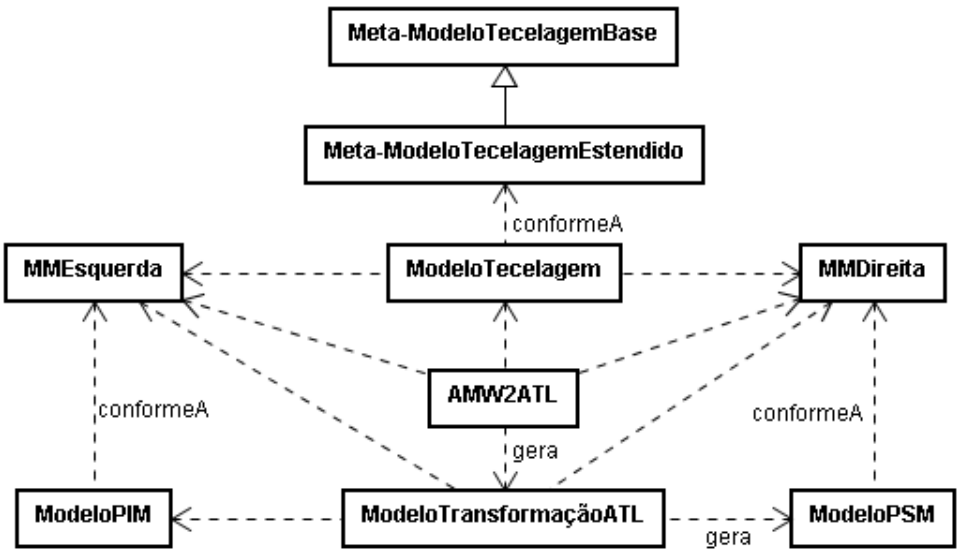

Figura 6.10: Guia da tecelagem de modelos.

A especificação segue o guia de tecelagem de modelos apresentado na seção 4.7. A figura 6.10, duplicação da figura 4.7, apresenta o guia geral de especificação

de transformação da técnica de tecelagem de modelos, a ser seguido nesta etapa.

Extensão do meta-modelo base de tecelagem de modelo

Inicialmente, é necessário criar uma extensão do meta-modelo base de tecelagem de modelos (*Meta-ModeloTecelagemEstendido*), proposto por FABRO et al. (2005), com os elementos que descrevem a semântica das ligações para o domínio do problema em questão. Tal extensão é gerada utilizando a linguagem KM3 devido às facilidades anteriormente mencionadas. Os elementos criados na extensão são: *IgualdadeClasse*, *IgualdadeAtributo* e *IgualdadeTipo*, os quais estabelecem a semântica para uma determinada ligação. Eles são extensões do elemento *WLink*, contido no meta-modelo base de tecelagem de modelos. Como estabelecido nos objetivos do trabalho, por questões de simplicidade, utilizar-se-á apenas a semântica de igualdade, isto é, a extensão IgualdadeClasse. O elemento *IgualdadeClasse* indica que o elemento do meta-modelo à esquerda, o qual está relacionado ao modelo fonte da transformação, tem uma relação de igualdade com o elemento do meta-modelo à direita, o qual está relacionado ao modelo alvo da transformação. Esta ligação deve ser utilizada apenas para estabelecer uma ligação de igualdade entre uma classe do meta-modelo fonte e o seu respectivo elemento do meta-modelo alvo. Por exemplo, estabelecer uma ligação entre o elemento *OrdemTrabalho* do meta-modelo *OrdemServiço* e o elemento *Tabela* do meta-modelo de banco de dados relacional.

O código a seguir, apresenta um fragmento da especificação, em linguagem KM3, da extensão do meta-modelo base de tecelagem de modelos. A linha 1 cria uma classe de nome *IgualdadeClasse* que estende o elemento *WLink* do meta-modelo base de tecelagem de modelos. A classe criada representa o tipo da ligação de igualdade de classe mencionado anteriormente. São criadas também mais duas classes, *Esquerda* e *Direita*, ambas estendendo o elemento *WLinkEnd* do meta-modelo base de tecelagem de modelos. As classes *Esquerda* e *Direita* representam o elemento do modelo fonte e o elemento do modelo alvo, respectivamente, ligados pelo tipo de ligação *IgualdadeClasse*. A especificação completa da extensão do meta-modelo base de tecelagem de modelos com os tipos de ligação mencionados é apresentada no Apêndice F.

```
1.    class IgualdadeClasse extends WLink { }
2.    class Esquerda extends WLinkEnd { }
3.    class Direita extends WLinkEnd { }
```

Geração do modelo de tecelagem

Após a extensão do meta-modelo base de tecelagem de modelos é necessário criar o modelo de tecelagem (*ModeloTecelagem*). Para a geração deste modelo são necessários: o meta-modelo fonte (*MMEsquerda*), o meta-modelo alvo (*MMDireita*) e o meta-modelo de tecelagem de modelos estendido (*Meta-ModeloTecelagemEstendido*). O modelo é gerado estabelecendo ligações entre os elementos do meta-modelo fonte e do meta-modelo alvo. Os tipos de ligações utilizados são as especificadas no meta-modelo de tecelagem estendida. Para a realização do mapeamento utiliza-se a ferramenta AMW.

Todas as classes do meta-modelo fonte têm uma ligação do tipo *IgualdadeClasse* com o elemento *Tabela* do meta-modelo alvo, em virtude da decisão mencionada de se ter cada classe do modelo PIM transformada em uma tabela correspondente no modelo PSM.

Geração do modelo de transformação de alta ordem (HOT)

A geração do modelo de transformação de modelos de alta ordem (HOT) é especificada em linguagem ATL (*AMW2ATL*), e seu objetivo é implementar a semântica dos tipos de ligações adicionados ao meta-modelo base de tecelagem de modelos, através da extensão deste. Para a geração do modelo de transformação são necessários: o meta-modelo fonte (*MMEsquerda*), o meta-modelo alvo (*MMDireita*) e o meta-modelo de tecelagem de modelos estendido (*Meta-ModeloTecelagemEstendido*). A saída da execução do modelo de transformação gera, automaticamente, uma especificação de transformação de modelos em linguagem ATL (*ModeloTransformaçãoATL*). O modelo gerado deve ser idêntico ao modelo de transformação implementado na etapa 1 deste experimento.

A implementação da semântica de igualdade específica, por exemplo, como realizar a transformação do tipo de ligação *IgualdadeClasse* estabelecida entre elementos do meta-modelo fonte e meta-modelo alvo, isto é, como transformar uma classe do modelo fonte e uma tabela do modelo alvo. Para cada tipo de ligação contida no meta-modelo estendido de tecelagem de modelos, deve ser criada uma regra de transformação na especificação de transformação de modelo de alta ordem. A especificação completa do modelo de transformação de alta ordem é apresentada no Apêndice G.

Execução do modelo de transformação de modelos

Para completar o processo de transformação de modelos utilizando a técnica de tecelagem de modelos, é necessário executar o *ModeloTransformaçãoATL* ge-

rado anteriormente. Para a execução deste modelo de transformação são necessários o meta-modelo fonte (*MMEsquerda*), o meta-modelo alvo (*MMDireita*) e o modelo fonte da transformação (*ModeloPIM*). O resultado desta transformação, é a geração automática do modelo alvo (*ModeloPSM*). Este modelo gerado deve ser idêntico ao modelo gerado na etapa 1 deste experimento. A descrição completa do modelo PSM é apresentada no Apêndice E

6.4.3 Resultados Obtidos

Esta seção apresenta os resultados obtidos no experimento de acordo com os objetivos propostos. Para auxiliar a comparação das duas técnicas de transformação de modelos utilizadas na realização destes experimento, a Tabela 6.5 apresenta a quantidade de linhas de código escritas manualmente e a quantidade de linhas de código, escritas manualmente, duplicadas como resultado de cada uma das técnicas utilizadas. A coluna *ATL* da tabela, apresenta os resultados obtidos na primeira etapa do experimento, em que foi utilizada a técnica que usa somente uma linguagem de especificação de transformação de modelos, no caso a linguagem ATL. A coluna *Tecelagem de Modelos - HOT* da tabela, apresenta os resultados obtidos na segunda etapa do experimento, onde foi utilizada a técnica de tecelagem de modelos. Na tabela foram considerados apenas os códigos gerados manualmente, em linguagem ATL, nas duas etapas do experimento.

Tabela 6.5: Avaliação Linhas de Código - Experimento 1.

Linhas de Código	**ATL**	**Tecelagem de Modelos - HOT**
Total	93	158
Duplicadas	48	-

Analisando somente os dados de quantidade de linhas de código geradas por cada uma das técnicas, tem-se a impressão de que a técnica utilizando somente a linguagem ATL tem vantagem nesse aspecto. Entretanto, conforme apresentado na primeira etapa do experimento, para cada entidade do meta-modelo fonte que deveria ser transformado em, por exemplo, uma tabela da plataforma de banco de dados relacional, foi necessário a criação de uma regra específica para esta transformação. Em decorrência disto, foi necessário gerar uma grande quantidade de código duplicado, ao todo 48 linhas, conforme apresentado na tabela acima. Porém, o número de linhas de código duplicado pode variar, como por exemplo, cada regra que especifica a transformação de um elemento do meta-modelo fonte em um elemento do meta-modelo alvo contém um total de 13 linhas de código (ver Apêndice D). Caso seja adicionado um novo elemento no meta-modelo fonte que deva ser mapeado em uma *Tabela* do meta-modelo alvo, uma nova regra de

transformação deve ser escrita. Esta nova regra irá gerar um total de 8 linhas a mais de código duplicado na especificação de transformação. Portanto, à medida que se adicionarem elementos ao meta-modelo fonte, aumenta também o número de código duplicado da especificação de transformação de modelos. A manutenção neste código fica também prejudicada, pois, será necessário corrigir, manualmente, o mesmo tipo de código diversas vezes, estando assim sujeito a erros na manutenção do código.

Observando apenas a quantidade de linhas de código manualmente requerido pelas duas técnicas, aparentemente a técnica utilizando somente uma linguagem de especificação de transformação de modelos mostra-se mais fácil do que a técnica de tecelagem de modelos. No entanto, quando analisada a quantidade de linhas de código duplicadas percebe-se um ganho do uso da técnica de tecelagem de modelos, já que esta não gerou nenhuma linha de código duplicada. Na realidade, a utilização da técnica de tecelagem de modelos facilita tanto a expansão do meta-modelo fonte quanto a manutenção da especificação de transformação de modelos. Quando um novo elemento for adicionado ao meta-modelo fonte e, por exemplo, precisar ser mapeado em um elemento *Tabela* do meta-modelo de banco de dados relacional, é necessário apenas adicionar uma ligação de *IgualdadeClasse* entre estes elementos no modelo de tecelagem (*ModeloTecelagem*) e executar novamente a especificação de transformação *AMW2ATL*, para que uma nova especificação de transformação de modelos (*ModeloTransformaçãoATL*) seja automaticamente gerada. Conseqüentemente, a medida que são adicionados novos elementos ao meta-modelo fonte há um aumento no código utilizando somente a linguagem ATL, enquanto a especificação de transformação de modelos de alta ordem (HOT) permanece fixa.

Caso seja necessário algum tipo de manutenção na especificação de transformação de modelos (*ModeloTransformaçãoATL*), é necessário apenas efetuar as alterações na especificação de transformação de modelos de alta ordem (HOT), para que posteriormente ela gere uma nova especificação de transformação de modelos já contemplando as alterações em todas as regras de transformação necessárias. Desse modo, a maior facilidade na escrita de código utilizando apenas a linguagem ATL é aparente.

Outro fator observado analisando-se as duas técnicas é o fato de o mapeamento propriamente dito entre os dois meta-modelos, fonte e alvo, é desvantajoso quando se utiliza somente a linguagem ATL. Isto ocorre, porque, a semântica do mapeamento entre os elementos dos meta-modelos fica oculta sob o código ATL da especificação de transformação de modelo. Aqui, é necessário ter um conhe-

cimento detalhado da linguagem ATL para captar a semântica do mapeamento contida em uma determinada regra. Para exemplificar este ponto, apresenta-se a seguir um fragmento do código da especificação de transformação de modelos gerado na primeira etapa do experimento. O código apresenta uma regra de transformação de uma elemento *OrdemTrabalho* do meta-modelo fonte em uma elemento *Tabela* do meta-modelo alvo. O desenvolvedor deve ler o conteúdo dos comandos *from* e *to* da regra de transformação para deduzir a semântica da transformação.

```
1.  rule OrdemTrabalho2Tabela {
2.     from
3.       ot :  OrdemTrabalho!OrdemTrabalho
4.     to
5.       saidaTabela :  Relacional!Tabela (
6.         nome <- ot.nome,
7.         col <- Sequence{chave},
8.         chaveTabela <- Set{chave}
9.       ),
10.      chave :  Relacional!Coluna (
11.        nome <- 'idObjeto'
12.      )
13.  }
```

Com a utilização da técnica de tecelagem de modelos, o mapeamento entre os elementos do meta-modelo fonte e alvo fica registrado em um modelo independente de uma linguagem específica de transformação de modelos. Outro fator vantajoso da tecelagem de modelos é o fato de as ligações do mapeamento estarem associadas a um tipo, fornecendo assim a verificação (formal) de tipos para a ligação estabelecida entre os elementos dos meta-modelos.

6.5 Experimento 2

Esta seção apresenta os objetivos, a descrição e execução, e os resultados do segundo experimento proposto.

6.5.1 Objetivo

Este segundo experimento tem por objetivo investigar as vantagens que a técnica de tecelagem de modelos oferece em relação à reutilização de decisões de projeto (*design*) no mapeamento entre dois meta-modelos distintos. Compara-se, do mesmo modo que no Experimento 1, a técnica de tecelagem de modelos com um enfoque de geração de especificação de transformação de modelos utilizando apenas uma linguagem de transformação de modelos. Analisa-se aqui as possibi-

lidades de reutilização de decisões de projeto dos tipos de ligação do mapeamento entre os meta-modelos proporcionadas pelas duas técnicas mencionadas. A cobertura deste segundo experimento está associada aos dois objetivos propostos no presente trabalho. Apesar deste experimento cobrir os dois objetivos propostos, houve a necessidade de realizar um experimento anterior para que este último pudesse gerar os dados preliminares para a investigação da reutilização de decisões de projeto proposto por este segundo experimento. Conforme apresentado na seção 6.3, as ferramentas utilizadas para este experimento são a AMW e a ATL.

6.5.2 Descrição do Experimento

O experimento consiste em transformar o modelo PIM, criado na seção 6.2, em um modelo PSM associado à plataforma da linguagem de programação JAVA. O experimento está dividido em duas etapas. A primeira etapa é a geração da especificação de transformação utilizando apenas a linguagem ATL. Isto é, nesta etapa a técnica de tecelagem de modelos não é utilizada. A segunda etapa é a geração de especificação de transformação de modelos utilizando a técnica de tecelagem de modelos. Nesta, a ferramenta AMW e a linguagem ATL são utilizadas conjuntamente.

A figura 6.6 apresentou o modelo PIM utilizado como entrada para o processo de transformação das duas etapas do experimento. O modelo é um fragmento do modelo de domínio genérico de ordens de produção apresentado na seção 6.2. Do mesmo modo que no Experimento 1, por questões de simplicidade, as especializações de foram excluídas do experimento e o modelo PIM utilizado no experimento, descrito na linguagem UML, deve ser especificado em formato XMI e estar em conformidade com a DSL *OrdemServiço* especificada em linguagem KM3. O Apêndice A apresenta o meta-modelo fonte, especificado em linguagem KM3 utilizado neste experimento. O Apêndice B apresenta o modelo de entrada do processo de transformação (Modelo PIM), especificado no formato XMI, utilizado neste experimento.

A Figura 6.11 apresenta o diagrama do meta-modelo simplificado que descreve alguns elementos da linguagem JAVA (INRIA, 2005). Por questões de simplicidade, só foram modelados os aspectos da linguagem de interesse a este experimento. O meta-modelo contém os seguintes elementos: *ElementoJava*, *Tipo*, *Campo*, *Pacote*, *TipoPrimitivo*, *Modificador*, *Metodo* e *ClasseJava*. O elemento *ElementoJava* é um elemento abstrato, contendo um atributo do tipo *String* de-

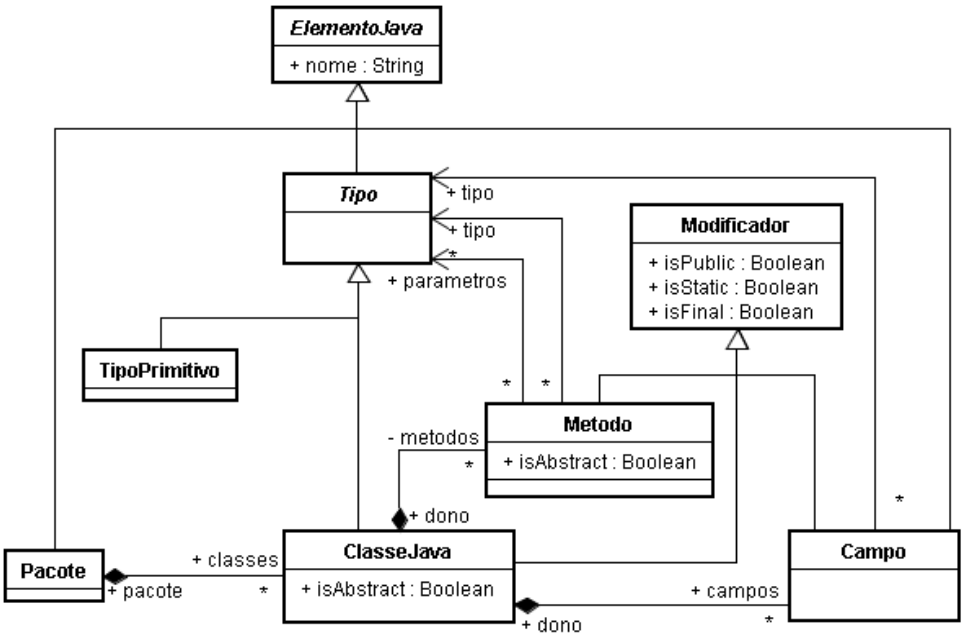

Figura 6.11: Meta-modelo simplificado da linguagem de programação JAVA.

nominado *nome*, indicando que todas as suas extensões devem possuir um nome. O elemento *Tipo* é um *ElementoJava* e representa os tipos da linguagem. O elemento *TipoPrimitivo* é um *Tipo* e representa os tipos primitivos contidos na linguagem. O elemento *Pacote* é um *ElementoJava* e representa um pacote (*package*) da linguagem JAVA. Um pacote é composto por classes, representada pelo elemento *ClasseJava*. O elemento *ClasseJava* é um *Tipo* e representa uma classe na linguagem. Este elemento é também uma especialização do elemento abstrato *Modificador* e é composto por métodos e campos, representados pelos elementos *Metodo* e *Campo* respectivamente. O elemento *Metodo* é composto por um tipo e por parâmetros, ambos representados pelo elemento *Tipo*. O elemento *Campo* está associado a um *Tipo*. O modelo PSM gerado pela transformação deve estar em conformidade com este meta-modelo. Para este experimento, o meta-modelo foi definido na linguagem KM3, pelas mesmas razões que no Experimento 1. A definição deste meta-modelo foi baseada no trabalho de (INRIA, 2005) e sua especificação é apresentada no Apêndice H.

6.5.2.1 Etapa 1 - Especificação de transformação de modelos utilizando a linguagem ATL

O objetivo desta etapa é escrever uma especificação de transformação de modelos, em linguagem ATL, para transformar o modelo PIM para a plataforma da linguagem de programação JAVA. Assume-se que cada classe do modelo PIM deve ser transformada em uma classe da linguagem JAVA. Os atributos da classe

devem se tornar, respectivamente, atributos da classe JAVA.

Do mesmo modo que no Experimento 1, para realizar a transformação, regras de transformação na linguagem ATL são criadas havendo a necessidade de especificar uma regra de transformação para cada tipo do modelo PIM. O código a seguir, apresenta um fragmento da especificação de transformação escrita para esta etapa do experimento. A especificação da transformação completa está descrita no Apêndice I. O resultado do processo de transformação é um arquivo, em formato XMI, contendo a especificação do modelo alvo. O conteúdo do arquivo é apresentado no Apêndice J.

A especificação da transformação na linguagem ATL segue os mesmos passos do Experimento 1. Aqui cria-se um *module* com o nome da especificação de transformação (*OrdemTrabalho2JAVA*), o meta-modelo fonte e o meta-modelo alvo da transformação.

```
1.  module OrdemTrabalho2JAVA;
2.  create OUT : JAVA from IN : OrdemTrabalho;
3.  rule OrdemTrabalho2Classe {
4.    from
5.      ot :  OrdemTrabalho!OrdemTrabalho
6.    to
7.      saida :  Java!ClasseJava (
8.        nome <- ot.nome
9.      )
10.  }
```

A partir da linha 3, inicia-se a criação de uma regra ATL com o objetivo de transformar um tipo *OrdemTrabalho* do modelo PIM em uma classe JAVA. A especificação desta regra inicia-se com o comando *rule*, seguido pelo nome da regra (*OrdemTrabalho2JAVA*). As linhas de 4 a 6 são idênticas à especificação correspondente do Experimento 1. No comando *to*, inicialmente (linha 7) especifica-se que um elemento *ClasseJava* do meta-modelo *JAVA* deve ser criado e este deve ser associado à variável *saida*. Os passos para a criação de uma classe são especificados na linha 8, indicando que o nome da classe deve ter o mesmo nome do elemento do tipo *OrdemTrabalho* do modelo fonte.

O resultado da execução da especificação de transformação de modelo gerada é um arquivo, em formato XMI, contendo a descrição do modelo alvo. A linha de código a seguir apresenta a descrição de uma classe Java do modelo PSM gerado, gerada a partir da classe *OrdemProdução* associada ao estereótipo *OrdemTrabalho*. O elemento *ClasseJava* do modelo PSM contém um atributo *nome*, iniciado com o valor *OrdemProducao*, representando o nome da classe. A descrição completa do modelo PSM gerado é apresentada no Apêndice J.

```
1.  <ClasseJava nome="OrdemProducao"/>
```

Nesta etapa do experimento, pouco se pôde aproveitar da especificação de transformação criada na primeira etapa do Experimento 1. Essa dificuldade, conforme mencionado na seção 6.4.3, deve-se ao fato de a semântica do mapeamento entre os elementos do meta-modelo fonte e alvo ficar oculta sob o código que efetivamente realiza a transformação dos modelos. Outro fator que dificulta o reaproveitamento da especificação de transformação de modelo é o fato de os elementos de cada um dos meta-modelos estarem amarrados a elementos específicos da linguagem de transformação.

6.5.2.2 Etapa 2 - Especificação de transformação de modelos utilizando a tecelagem de modelos

O objetivo desta etapa é especificar a transformação de modelos, utilizando a técnica de tecelagem de modelos, para transformar o modelo PIM para um modelo em conformidade com plataforma da linguagem de programação JAVA. Como na etapa 1, assume-se que cada classe do modelo PIM deve ser transformada em uma classe JAVA. Ao final do processo de transformação de modelos o resultado obtido (modelo PSM) deve ser o mesmo da etapa anterior.

A especificação segue o guia de tecelagem de modelos apresentado na seção 4.7. A figura 6.12, novamente duplicada na figura 4.7, apresenta o guia de especificação de transformação da técnica de tecelagem de modelos, a ser seguido nesta etapa.

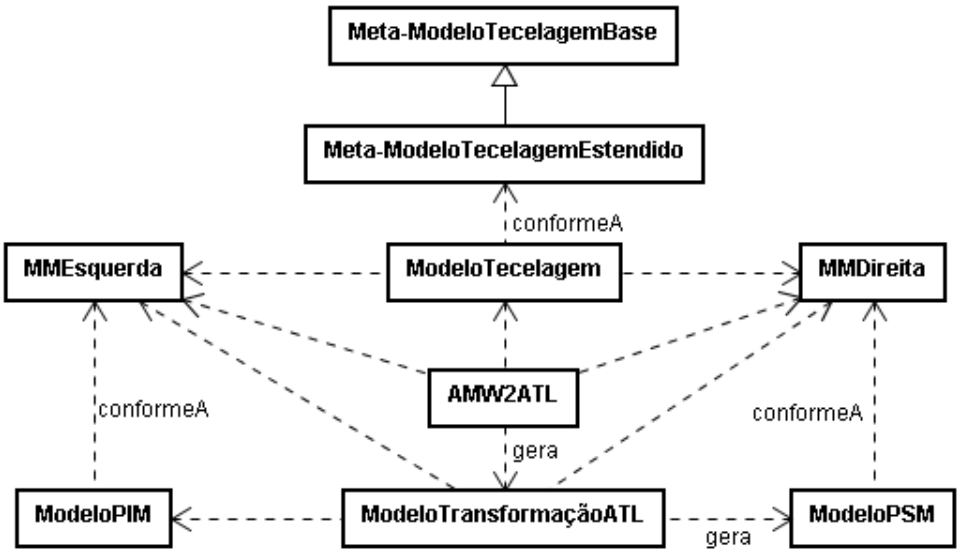

Figura 6.12: Guia da tecelagem de modelos.

Extensão do meta-modelo base de tecelagem de modelo

Na segunda etapa do primeiro experimento, inicialmente foi necessário a criação de uma extensão do meta-modelo base de tecelagem de modelo com os elementos que descrevem a semântica das ligações para o domínio do problema em questão. Entretanto, neste experimento, esta extensão do meta-modelo base de tecelagem de modelo será reutilizada. Isto permite que os tipos de ligações utilizados no mapeamento anterior sejam reutilizados neste novo mapeamento, uma vez que o domínio do problema é o mesmo (meta-modelo fonte), alterando apenas a plataforma na qual o modelo fonte será transformado (plataforma JAVA). As decisões de projeto (*design*) utilizadas para estabelecer o mapeamento entre os elementos do meta-modelo fonte e os elementos do meta-modelo alvo do experimento anterior, podem, neste experimento, ser utilizadas para auxiliar na geração do mapeamento. A especificação completa da extensão do meta-modelo base de tecelagem de modelos é apresentada no Apêndice F.

Geração do modelo de tecelagem

Uma vez obtida a extensão do meta-modelo base de tecelagem de modelos é necessário criar o modelo de tecelagem (*ModeloTecelagem*). A geração deste modelo realiza-se do mesmo modo como no Experimento 1. No entanto, a geração reutiliza tanto os tipos de ligação gerados no experimento anterior, como as decisões de projeto utilizadas para estabelecer o mapeamento entre os elementos do meta-modelo fonte e os elementos do meta-modelo alvo do experimento anterior. A título de recordação, o Experimento 1 efetuou o mapeamento, com o tipo de ligação *IgualdadeClasse*, entre as classes do meta-modelo fonte e o elemento *Tabela* do meta-modelo alvo. Esta decisão de projeto (*design*), é utilizada para efetuar o mapeamento, com o tipo de ligação *IgualdadeClasse*, entre as classes do meta-modelo fonte e o elemento *ClasseJava* do meta-modelo alvo, significando que para cada uma da classes utilizadas no modelo PIM, uma classe JAVA correspondente deverá ser criada no modelo PSM.

Geração do modelo de transformação de alta ordem (HOT)

A geração do modelo de transformação de modelos de alta ordem (HOT) é realizada como no Experimento 1. No entanto, a especificação HOT do experimento anterior não pode ser reaproveitada, pois o meta-modelo alvo foi alterado e a semântica da ligação *IgualdadeClasse* deve ser especificada utilizando tanto os elementos do modelo fonte quanto os elementos do modelo alvo da transformação. Portanto, é necessário escrever completamente uma nova especificação de transformação de alta ordem. A saída da execução do modelo de transformação gera,

automaticamente, uma especificação de transformação de modelos em linguagem ATL (*ModeloTransformaçãoATL*). O modelo gerado deve ser idêntico ao modelo de transformação implementado na primeira etapa deste experimento.

A implementação da semântica, mencionada anteriormente, especifica, por exemplo, como realizar a transformação do tipo de ligação *IgualdadeClasse* estabelecida entre elementos do meta-modelo fonte e meta-modelo alvo, isto é, como transformar uma classe do modelo fonte e uma classe JAVA do modelo alvo. Para cada tipo de ligação contida no meta-modelo estendido de tecelagem de modelos, um regra de transformação para este tipo dever ser criada na especificação de transformação de modelo de alta ordem. A especificação completa desta especificação é apresentada no Apêndice K.

Execução do modelo de transformação de modelos

Para completar o processo de transformação de modelos utilizando a técnica de tecelagem de modelos, é necessário executar o *ModeloTransformaçãoATL* gerado anteriormente. A execução é idêntica a do Experimento 1. A descrição completa do modelo PSM é apresentada no Apêndice J.

6.5.3 Resultados Obtidos

Esta seção apresenta os resultados obtidos no experimento de acordo com os objetivos propostos. O primeiro aspecto analisado é idêntico ao do Experimento 1, isto é, comparam-se as duas técnicas de transformação de modelos quanto à redução de linhas de código duplicadas manualmente produzidas. A Tabela 6.6, do mesmo modo que no Experimento 1, apresenta o total de linhas de código e a quantidade de linhas de código duplicadas. Todas as observações, descritas na seção 6.4.3, correspondentes a essa comparação, aplicam-se integralmente neste caso.

Tabela 6.6: Linhas de Código - Experimento 2.

Linhas de Código	**ATL**	**Tecelagem de Modelos - HOT**
Total	58	118
Duplicadas	18	-

Outro aspecto investigado foi a possibilidade de reutilização de decisões de projeto (*design*) tomadas anteriormente. No enfoque utilizando somente uma linguagem de especificação de modelos (Etapa 1), pouco se pôde reaproveitar da especificação de transformação de modelo gerada na primeira etapa do Experimento 1. Tal dificuldade, conforme mencionado e detalhado na seção 6.4.3, se deve ao fato de a semântica do mapeamento entre os elementos do meta-modelo

fonte e alvo ficar oculta sob o código que efetivamente realiza a transformação dos modelos [2]. Outro fator que dificulta o reaproveitamento da especificação de transformação do modelo, mesmo que de forma manual, é o fato de os elementos de cada um dos meta-modelos estarem diretamente relacionados e dependentes dos elementos e comandos específicos da linguagem de transformação. O código a seguir apresenta um exemplo desta afirmação. Para obter o nome da classe Java a ser gerada (linha 6), é necessário utilizar uma variável anteriormente criada (linha 3). Portanto, o desenvolvedor que pretenda utilizar, mesmo que de forma manual, esta especificação de transformação de modelo deve ter o conhecimento da linguagem de transformação, no caso a linguagem ATL. Além disso, deve saber separar os elementos do meta-modelo fonte e alvo dos elementos e comandos da linguagem de especificação de transformação de modelos.

```
1.  rule OrdemTrabalho2Classe {
2.    from
3.      ot :  OrdemTrabalho!OrdemTrabalho
4.    to
5.      saida :  Java!ClasseJava (
6.        nome <- ot.nome
7.      )
8.  }
```

A utilização da técnica de tecelagem de modelos permitiu, inicialmente, a reutilização do meta-modelo estendido de tecelagem de modelo (*Meta-ModeloTecelagemEstendido*) gerado no primeiro experimento, uma vez que este meta-modelo contém os tipos de ligações necessários para o mapeamento dos elementos do meta-modelo fonte e alvo neste segundo experimento. O fato de o domínio do problema (meta-modelo e modelo fonte) utilizado nos dois experimentos ser o mesmo, também contribuiu para o reaproveitamento do meta-modelo estendido de tecelagem de modelo.

A técnica de tecelagem de modelos permitiu ainda, a reutilização, de forma manual, de decisões de projeto (*design*) e heurísticas, do experimento anterior, para gerar o modelo de tecelagem (*ModeloTecelagem*) deste segundo experimento. Isto é, a análise do mapeamento entre os elementos do meta-modelo fonte e do meta-modelo alvo do experimento anterior auxiliou na tomada de decisão de como estabelecer o mapeamento entre os elementos do meta-modelo fonte e do meta-modelo alvo. Por exemplo, o experimento anterior efetuou o mapeamento, com o tipo de ligação *IgualdadeClasse*, entre as classes do meta-modelo fonte e o elemento *Tabela* do meta-modelo alvo. A análise desta decisão de projeto auxiliou

[2] A apresentação desse fenômeno - ocultação da semântica no código - encontra-se na seção 6.4.3.

a gerar o mapeamento, com o tipo de ligação *IgualdadeClasse*, entre as classes do meta-modelo fonte e o elemento *ClasseJava* do meta-modelo alvo.

7 Conclusão e Trabalhos Futuros

Este trabalho apresentou a técnica de tecelagem de modelos, motivado pela propriedade desta técnica apresentar uma importante independência em relação a linguagens de transformação de modelos. A técnica permite que sejam estabelecidas ligações, com tipo definido, entre elementos de dois meta-modelos. Desta estratégia decorre uma outra vantagem da técnica: a definição direta da semântica dos tipos das ligações pelo usuário permite a geração mais simples de elementos do domínio do problema relacionado ao sistema que está sendo desenvolvido. Tal geração é mais simples quando comparada a transformações realizadas em linguagens de transformação com semântica fixa, que requerem um significativo trabalho de composição sobre os recursos oferecidos por essas linguagens.

Um outro aspecto importante é que a definição manual desses tipos permite uma flexibilidade no emprego de heurísticas e decisões de projeto (*design*) dos usuários, nem sempre fácil de ser aplicado em linguagens de semântica fixa. Além disso, uma outra característica da técnica é que a semântica definida para uma ligação em um determinado mapeamento de modelos pode ser reutilizada em um outro mapeamento de modelos.

Para analisar as vantagens oferecidas pela técnica de tecelagem de modelos, em relação a um enfoque utilizando apenas uma linguagem de transformação, foram propostos dois experimentos. O objetivo do primeiro experimento foi apresentar as vantagens que a técnica de tecelagem de modelos oferece em relação a reutilização de trechos de código para a geração de especificações de transformação. O resultado do experimento mostrou que, para o estudo de caso em questão, a técnica de tecelagem de modelos, através da utilização de transformações de alta ordem (HOTs) permitiu a reutilização de trechos de código, evitando assim a duplicação de código, como ocorre no enfoque utilizando apenas a linguagem de transformação ATL.

O objetivo do segundo experimento foi apresentar as vantagens que a técnica

de tecelagem de modelos oferece em relação à reutilização de decisões de projeto (*design*) no mapeamento entre dois meta-modelos diferentes. O resultado do experimento mostrou que a técnica de tecelagem de modelos possibilita:

- O reaproveitamento e a extensão do meta-modelo de tecelagem estendido gerado no primeiro experimento;
- A reutilização, de forma manual, de decisões de projeto (*design*) e heurísticas utilizadas em mapeamentos anteriores.

Um aspecto observado na realização deste trabalho, principalmente dos experimentos, é a complexidade dos enfoques atuais de transformação de modelos no contexto da MDA. Os desenvolvedores de especificação de transformação de modelos precisam ter um conhecimento de aspectos da teoria da computação e de fundamentos formais da engenharia de software, como por exemplo, linguagens específicas de domínio, meta-modelagem, entre outros.

Em trabalho futuros pretende-se:

- Implementar outros tipos de semântica dos tipos de ligação relacionados ao modelo de domínio utilizado;
- Utilizar a técnica de tecelagem de modelos para avaliar as vantagens oferecidas nas questões de bidirecionalidade e propagação de mudanças;
- Avaliar as vantagens da técnica de tecelagem de modelos em outros contextos de transformação como, por exemplo, transformações do tipo horizontal ou transformações de um PSM para um ISM;
- Enfocar a atual definição da OMG de que a extensão de uma linguagem DSL realizada com a utilização do perfil UML preserva a semântica dos elementos do meta-modelo existente (OMG, 2005c), como um teorema e elaborar a sua prova.

Referências Bibliográficas

ABOUZAHRA, A.; BÉZIVIN, J.; FABRO, M. D. D.; JOUAULT, F. A Practical Approach to Bridging Domain Specific Languages with UML profiles. In: *Proceedings of the Best Practices for Model Driven Software Development at OOPSLA'05*. San Diego, California, USA: [s.n.], 2005. Disponível em: <http://www.softmetaware.com/oopsla2005/bezivin1.pdf>.

BALOGH, A.; VARRÓ, D. Advanced Model Transformation Language Constructs in the VIATRA2 Framework. In: *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*. [S.l.: s.n.], 2006. p. 1280–1287.

BROWN, A. W. Model driven architecture: Principles and practice. *Software and Systems Modeling*, v. 3, n. 4, p. 314–327, Dezembro 2004.

BÉZIVIN, J.; GÉRARD, S. A Preliminary Identification of MDA Components. In: *OOPSLA 2002 Workshop on Generative Techniques in the context of Model Driven Architecture*. [S.l.: s.n.], 2002.

CARIOU, E.; MARVIE, R.; SEINTURIER, L.; DUCHIEN, L. OCL for the Specification of Model Transformation Contracts. In: *Workshop OCL and Model Driven Engineering of the Seventh International Conference on UML Modeling Languages and Applications (UML 2004)*. Portugal: [s.n.], 2004.

COOK, S. The UML family: Profiles, prefaces and packages. In: EVANS, A.; KENT, S.; SELIC, B. (Ed.). *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*. [S.l.]: Springer, 2000. (LNCS, v. 1939), p. 255–264.

CZARNECKI, K. Overview of Generative Software Development. *LNCS*, v. 3566, p. 326–341, 2005.

CZARNECKI, K.; HELSEN, S. Classification of Model Transformation Approaches. In: *Proceedings of the 2nd OOPSLA'03 Workshop on Generative Techniquesin the Context of MDA*. [S.l.: s.n.], 2003.

DEURSEN, A. V.; KINT, P.; VISSER, J. Domain-Specific Languages: An Annotated Bibliography. In: *ACM SIGPLAN*. [S.l.: s.n.], 2000. v. 35, n. 6.

DUDDY, K.; GERBER, A.; LAWLEY, M.; RAYMOND, J. S. K. Model Transformation: A declarative, reusable patterns approach. In: Proceedings of the Seventh IEEE International Enterprise Distributed Object Computing Conference (EDOC'03). *Proceedings of the Enterprise Distributed Object Computing Conference, Seventh IEEE International (EDOC'03)*. [S.l.], 2003. p. 174–185.

FABRO, M. D. D.; BÉZIVIN, J.; JOUAULT, F.; BRETON, E.; GUELTAS, G. AMW: A Generic Model Weaver. In: *Proceedings of the 1ère Journée sur l'Ingénierie Dirigée par les Modèles (IDM05)*. Paris: [s.n.], 2005.

FOWLER, M. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. 3rd. ed. [S.l.]: Addison-Wesley, 2004.

FRANCE, R. B.; GHOSH, S.; DINH-TRONG, T.; SOLBERG, A. Model-Driven Development Using UML 2.0: Promises and Pitfalls. *Computer*, v. 39, n. 2, p. 59–66, Fevereiro 2006.

FRANKEL, D. S. *Model Driven Architecture - Applying MDA to Enterprise*. [S.l.]: Wiley, 2003.

FUENTES-FERNÁNDEZ, L.; VALLECILLO-MORENO, A. An Introduction to UML Profiles. *The European Journal for the Informatics Professional*, Upgrade, V, n. 2, p. 6–13, Abril 2004.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. [S.l.]: Addision-Wesley, 1995. 395 p.

GREENFIELD, J.; SHORT, K. *Software Factories - Assembling Applications with Patterns, Models, Frameworks, and Tools*. [S.l.]: Wiley, 2004.

GROSE, T. J.; DONEY, G. C.; BRODSKY, S. A. *Mastering XMI: Java Programming with XMI, XML, and UML*. [S.l.]: Wiley, 2002. 480 p.

HAY, D. C. *Data Model Patterns - Conventions of Thought*. [S.l.]: Dorset House Publishing, 1996.

INRIA. *ATL Transformation Example - UML to JAVA*. [S.l.], 2005. Disponível em: <http://www.eclipse.org/m2m/atl/atlTransformations/UML2Java/ExampleUML2Java[v00.01].pdf>. Acesso em: 08 fev. 2007.

JOUAULT, F.; BÉZIVIN, J. KM3: a DSL for Metamodel Specification. In: *Proceedings of 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems, LNCS 4037*. Bologna, Italy: [s.n.], 2006. p. 171–185.

JOUAULT, F.; KURTEV, I. Transforming Models with ATL. In: *Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005*. [S.l.: s.n.], 2005.

JOUAULT, F.; KURTEV, I. On the architectural alignment of ATL and QVT. In: *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*. [S.l.: s.n.], 2006. p. 1188–1195.

KLEPPE, A. *MDA Explained - The Model Driven Architecture: Practice and Promise*. [S.l.]: Addison-Wesley, 2003.

LOPES, D.; HAMMOUDI, S.; BÉZIVIN, J.; JOUAULT, F. Mapping Specification in MDA: from Theory to Practice. In: *Proceedings of the First International Conference on Interoperability of Enterprise Software and Applications (INTEROP-ESA'2005)*. [S.l.: s.n.], 2005. p. 253–264.

MELLOR, S. J. *MDA Distilled - Principles of Model Driven Architecture*. [S.l.]: Addison-Wesley, 2004.

MENS, T.; GORP, P. V. A Taxonomy of Model Transformation. In: *Proceedings of the International Workshop on Graph and Model Transformation (GraMoT)*. [S.l.: s.n.], 2005.

MERNIK, M.; HEERING, J.; SLOANA, A. M. When and how to Develop Domain-Specific Languages. *ACM Computing Surveys*, v. 37, n. 4, p. 316–344, Dezembro 2005.

MESERVY, T. O.; FENSTERMACHER, K. D. Transforming Software Development: An MDA Road Map. *Computer*, v. 38, n. 9, p. 52–58, Setembro 2005.

MILLER, J.; MUKERJI, J. (Ed.). *MDA Guide Version 1.0.1*. [S.l.], Junho 2003. Acesso em: 05 fev. 2007. Disponível em: <http://www.omg.org/docs/omg/03-06-01.pdf>.

OMG. *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification*. [S.l.], Novembro 2005. Acesso em: 05 fev. 2007. Disponível em: <http://www.omg.org/docs/formal/05-11-01.pdf>.

OMG. *MOF 2.0/XMI Mapping Specification, v2.1*. [S.l.], Setembro 2005. Acesso em: 05 fev. 2007. Disponível em: <http://www.omg.org/docs/formal/05-09-01.pdf>.

OMG. *Unified Modeling Language: Superstructure - version 2.0*. [S.l.], Agosto 2005. Acesso em: 05 fev. 2007. Disponível em: <http://www.omg.org/docs/formal/05-07-04.pdf>.

OMG. *Diagram Interchange - version 1.0*. [S.l.], Abril 2006. Acesso em: 05 fev. 2007. Disponível em: <http://www.omg.org/docs/formal/06-04-04.pdf>.

OMG. *Meta Object Facility (MOF) Core Specification - Version 2.0*. [S.l.], January 2006. Acesso em: 05 fev. 2007. Disponível em: <http://www.omg.org/docs/formal/06-01-01.pdf>.

OMG. *Object Constraint Language Specification - version 2.0*. [S.l.], May 2006. Acesso em: 05 fev. 2007. Disponível em: <http://www.omg.org/docs/formal/06-05-01>.

OMG. *Unified Modeling Language: Infrastructure - version 2.0*. [S.l.], Março 2006. Acesso em: 05 fev. 2007. Disponível em: <http://www.omg.org/docs/formal/05-07-05.pdf>.

RICHTER, M.; GOGOLLA, M. OCL: Syntax, semantics, and tools. *Lecture Notes in Computer Science*, v. 2263, p. 42, 2002.

RUMBAUGH, J.; JACOBSON, I.; BOOCH, G. *The Unified Modeling Language Reference Manual*. 2nd. ed. [S.l.]: Addison-Wesley, 2005.

SELIC, B. The pragmatics of model-driven development. *IEEE Software*, v. 20, n. 5, p. 19–25, Setembro/Outubro 2003.

TOVAL, A.; REQUENA, V.; FERNÁNDEZ, J. L. Emerging OCL tools. *Software and Systems Modeling*, v. 2, n. 4, p. 248–261, Dezembro 2003.

TRATT, L. *The Converge programming language*. [S.l.], Fevereiro 2005.

TRATT, L. The MT model transformation language. In: *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*. [S.l.]: ACM Press, 2006. p. 1296–1303.

WARMER, J.; KLEPPE, A. *The Object Constraint Language: Getting your models ready for MDA*. 2nd. ed. [S.l.]: Addison-Wesley, 2003.

Apêndice A – Especificação da DSL *OrdemServiço* em Linguagem KM3

Este apêndice apresenta a especificação, em linguagem KM3, da DSL *OrdemServiço* utilizada pelos experimentos do presente trabalho.

```
1.   package OrdemTrabalho {
2.     abstract class Nomeado {
3.       attribute nome :  String;
4.     }
5.     class OrdemTrabalho extends Nomeado { }
6.     abstract class Atividade extends Nomeado { }
7.     class PassoAtividade extends Atividade { }
8.     class OutraAtividade extends Atividade { }
9.     abstract class Procedimento extends Nomeado { }
10.    class PassoProcedimento extends Procedimento { }
11.    class OutroProcedimento extends Procedimento { }
12.    abstract class Grupo extends Nomeado { }
13.    class Pessoa extends Grupo { }
14.    class Organizacao extends Grupo { }
15.    class TipoAtivo extends Nomeado { }
16.    class Ativo extends Nomeado { }
18.    class EntregaProducao extends Nomeado { }
18.  }
19.  package TiposPrimitivos {
20.    datatype String;
21.    datatype Integer;
22.  }
```

Apêndice B – Especificação do Modelo PIM em XMI

Este apêndice apresenta a especificação, em formato XMI, do modelo PIM, apresentado na seção 6.2, utilizado para a realização dos experimentos propostos no presente trabalho.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns="OrdemTrabalho">
  <Pessoa nome="Pessoa"/>
  <OrdemTrabalho nome="OrdemProducao"/>
  <TipoAtivo nome="TipoMaterial"/>
  <Ativo nome="Material"/>
  <EntregaProducao nome="EntregaProducao"/>
  <PassoAtividade nome="PassoFabricacao"/>
  <PassoProcedimento nome="ProcedimentoFabricacao"/>
</xmi:XMI>
```

Apêndice C – Especificação do Meta-Modelo de Banco de Dados Relacional

Este apêndice apresenta a especificação, em linguagem KM3, do meta-modelo de banco de dados relacional utilizado no presente trabalho.

```
1.  package Relacional {
2.      abstract class Nomeado {
3.          attribute nome :  String;
4.      }
5.      class Tabela extends Nomeado {
6.          reference col[*] ordered container :  Coluna oppositeOf dono;
7.          reference chave[*] :  Coluna oppositeOf chaveDe;
8.      }
9.      class Coluna extends Nomeado {
10.          reference dono :  Tabela oppositeOf col;
11.          reference chaveDe[0-1] :  Tabela oppositeOf chave;
12.          reference tipo :  Tipo;
13.      }
14.      class Tipo extends Nomeado { }
15.  }

16.  package TiposPrimitivos {
17.      datatype Boolean;
18.      datatype Integer;
19.      datatype String;
20.  }
```

Apêndice D – Especificação do Modelo de Transformação do Experimento 1

Este apêndice apresenta a especificação, em linguagem ATL, da transformação de modelo utilizada na primeira etapa do primeiro experimento.

```
1.  module OrdemTrabalho2Relacional;
2.  create OUT : Relacional from IN : OrdemTrabalho;

3.  rule OrdemTrabalho2Tabela {
4.    from
5.      ot : OrdemTrabalho!OrdemTrabalho
6.    to
7.      saida : Relacional!Tabela (
8.        nome <- ot.nome,
9.        col <- Sequence {chave},
10.       chave <- Set{chave}
11.     ),
12.     chave : Relacional!Coluna (
13.       nome <- 'idObjeto'
14.     )
15. }

16. rule Pessoa2Tabela {
17.   from
18.     pessoa : OrdemTrabalho!Pessoa
19.   to
20.     saida : Relacional!Tabela (
21.       nome <- pessoa.nome,
22.       col <- Sequence {chave},
23.       chave <- Set{chave}
24.     ),
25.     chave : Relacional!Coluna (
26.       nome <- 'idObjeto'
27.     )
28. }

29. rule EntregaProducao2Tabela {
30.    from
31.      ep : OrdemTrabalho!EntregaProducao
32.    to
33.      saida : Relacional!Tabela (
34.        nome <- ep.nome,
35.        col <- Sequence {chave},
36.        chave <- Set{chave}
37.     ),
38.     chave : Relacional!Coluna (
39.       nome <- 'idObjeto'
40.     )
41. }

42. rule PassoAtividade2Tabela {
43.    from
44.      pa : OrdemTrabalho!PassoAtividade
45.    to
46.      saida : Relacional!Tabela (
47.        nome <- pa.nome,
48.        col <- Sequence {chave},
```

```
49.         chave <- Set{chave}
50.      ),
51.      chave : Relacional!Coluna (
52.        nome <- 'idObjeto'
53.      )
55. }

55. rule PassoProcedimento2Tabela {
56.    from
57.      pp : OrdemTrabalho!PassoProcedimento
58.    to
59.      saida : Relacional!Tabela (
60.        nome <- pp.nome,
61.        col <- Sequence {chave},
62.        chave <- Set{chave}
63.     ),
64.     chave : Relacional!Coluna (
65.       nome <- 'idObjeto'
66.     )
67. }

68. rule TipoAtivo2Tabela {
69.    from
70.      ta : OrdemTrabalho!TipoAtivo
71.    to
72.      saida : Relacional!Tabela (
73.        nome <- ta.nome,
74.        col <- Sequence {chave},
75.        chave <- Set{chave}
76.     ),
77.     chave : Relacional!Coluna (
78.       nome <- 'idObjeto'
79.     )
80. }

81. rule Ativo2Tabela {
82.    from
83.      a : OrdemTrabalho!TipoAtivo
84.    to
85.      saida : Relacional!Tabela (
86.        nome <- a.nome,
87.        col <- Sequence {chave},
88.        chave <- Set{chave}
89.     ),
90.     chave : Relacional!Coluna (
91.       nome <- 'idObjeto'
92.     )
93. }
```

Apêndice E – Descrição do Modelo PSM Gerado no Experimento 1

Este apêndice apresenta a descrição, em formato XMI, do modelo PSM gerado como resultado do processo de transformação do Experimento 1. Este modelo é igual para as etapas 1 e 2 do experimento.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns="Relacional">
  <Table name="OrdemProducao" key="/0/@col.0">
    <col name="idObjeto" keyOf="/0"/>
  </Table>
  <Table name="PassoFabricacao" key="/1/@col.0">
    <col name="idObjeto" keyOf="/1"/>
  </Table>
  <Table name="ProcedimentoFabricacao" key="/2/@col.0">
    <col name="idObjeto" keyOf="/2"/>
  </Table>
  <Table name="Pessoa" key="/3/@col.0">
    <col name="idObjeto" keyOf="/3"/>
  </Table>
  <Table name="EntregaProducao" key="/4/@col.0">
    <col name="idObjeto" keyOf="/4"/>
  </Table>
  <Table name="TipoMaterial" key="/5/@col.0">
    <col name="idObjeto" keyOf="/5"/>
  </Table>
    <Table name="Material" key="/6/@col.0">
    <col name="idObjeto" keyOf="/6"/>
  </Table>
</xmi:XMI>
```

Apêndice F – Especificação da Extensão do Meta-Modelo Base de Tecelagem de Modelos

Este apêndice apresenta a especificação, em linguagem KM3, da extensão do meta-modelo base de tecelagem de modelos utilizado no presente trabalho.

```
1.  package mmwot {
2.     class OrdemTrabalho extends WModel {
3.       reference modeloEsquerda container subsets wovenModel:  ModelRefXMIEsquerda;
4.       reference modeloDireita container subsets wovenModel:  ModelRefXMIRight;
5.     }
6.     abstract class ModelRefXMI extends WModelRef {
7.       attribute tipo :  String;
8.       reference ownedElementRef[0-*] container :  ElementRefXMI oppositeOf modelRef;
9.     }
10.    class ModelRefXMIEsquerda extends ModelRefXMI { }
11.    class ModelRefXMIDireita extends ModelRefXMI { }
12.    class ElementRefXMI extends WElementRef {
13.      reference modelRef:  WModelRef oppositeOf ownedElementRef;
14.    }
15.    class IgualdadeClasse extends WLink { }
16.    class Esquerda extends WLinkEnd { }
17.    class Direita extends WLinkEnd { }
18.  }
```

Apêndice G – Especificação da Transformação de Alta Ordem do Experimento 1

Este apêndice apresenta a especificação, em linguagem ATL, da especificação de transformação de modelos de alta ordem utilizada na etapa 2 do primeiro experimento.

```
1.  module AMW2ATL;
2.  create OUT : ATL from IN : AMW, esquerda : MOF, direita : MOF;

3.  helper def : contadorRegras : Integer = 0;
4.  helper def : nomeModulo  : String = 'OrdemTrabalho2BancoDados';


5.  rule Module {
6.    from
7.      amw : AMW!OrdemTrabalho
8.    to
9.      atl : ATL!Module (
10.       isRefining <- false,
11.       name <- thisModule.nomeModulo,
12.       inModels <- amw.modeloEsquerda,
13.       outModels <- amw.modeloDireita,
14.       elements <- Set {amw.ownedElement}
15.     )
16. }

17. rule ReferenciaModeloEsquerda {
18.   from
19.     amw : AMW!ModelRefXMIEsquerda
20.   to
21.     atl : ATL!OclModel (
22.       metamodel <- mm,
23.       name <- 'IN'
24.     ),
25.   mm : ATL!OclModel (
26.       elements <- MOF!EClassifier.allInstancesFrom('esquerda'),
27.       name <- 'OrdemTrabalho'
28.     )
29. }

30. rule ClassificadorFonte {
31.   from
32.     fonte : MOF!EClassifier (fonte.ePackage.name = 'OrdemTrabalho')
33.   to
34.     atl : ATL!OclModelElement (
35.       name <- fonte.name
36.     )
37. }

38. rule ReferenciaModeloDireita {
39.   from
40.     amw : AMW!ModelRefXMIDireita
```

```
41.   to
42.     atl : ATL!OclModel (
43.       metamodel <- mm,
44.       name <- 'OUT'
45.     ),
46.     mm : ATL!OclModel (
47.       elements <- MOF!EClassifier.allInstancesFrom('direita'),
48.       name <-'Relacional'
49.     )
50. }

51. rule ClassificadorAlvo {
52.   from
53.     alvo : MOF!EClassifier (alvo.ePackage.name ='Relacional')
54.   to
55.     atl : ATL!OclModelElement (
56.       name <- alvo.name
57.     )
58. }

59. rule IgualdadeClasse {
60.   from
61.     amw: AMW!IgualdadeClasse
62.   to
63.     atl : ATL!MatchedRule (
64.       inPattern <- amw.end->select(e | e.oclIsTypeOf(AMW!Esquerda))->first(),
65.       outPattern <- amw.end->select(e | e.oclIsTypeOf(AMW!Direita))->first()
66.     )
67.   do {
68.     thisModule.contadorRegras <- thisModule.contadorRegras + 1;
69.     atl.name <- 'Classe2Tabela_' + thisModule.contadorRegras.toString();
70.   }
71. }

72. rule PadraoEntradaIgualdadeClasse {
73.   from
74.     amw: AMW!Esquerda(amw.link.oclIsTypeOf(AMW!IgualdadeClasse))
75.   to
76.     atl: ATL!InPattern (
77.       elements <- elemento
78.     ),
79.     elemento: ATL!SimpleInPatternElement(
80.       varName <- 'ot',
81.       type <-tipo
82.     ),
83.     tipo : ATL!OclModelElement (
84.       name <-MOF!EClassifier.getInstanceById('left', amw.element.ref).name,
85.       model <- ATL!OclModel.allInstances()->select(e | e.name =
86.         'OrdemTrabalho')->first()
87.     )
88. }

89. rule PadraoSaidaIgualdadeClasse {
90.   from
91.     amw: AMW!Direita(amw.link.oclIsTypeOf(AMW!IgualdadeClasse))
92.   to
93.     atl : ATL!OutPattern (
94.       elements <- elementoSaida,
95.       elements <-  elementoChave
96.     ),
97.     elementoSaida : ATL!SimpleOutPatternElement(
98.       varName <- 'saida',
99.       type <- tipoSaida,
100.      bindings <- ligacaoSaida,
101.      bindings <- ligacaoColuna,
102.      bindings <- ligacaoConj
103.    ),
104.    tipoSaida : ATL!OclModelElement (
105.      name <- MOF!EClassifier.getInstanceById('right', amw.element.ref).name,
106.      model <- ATL!OclModel.allInstances()->select(e | e.name =
107.        'Relacional')->first()
108.    ),
109.    ligacaoSaida : ATL!Binding(
110.      propertyName <- 'name',
111.      value <- valorLigacaoSaida
112.    ),
113.    valorLigacaoSaida: ATL!NavigationOrAttributeCallExp (
```

```
114.      name <- 'nome',
115.      source <- fonteLigacaoSaida
116.    ),
117.    fonteLigacaoSaida : ATL!VariableExp(
118.      referredVariable <- thisModule.resolveTemp(amw.link.end->select (e |
119.        e.oclIsTypeOf(AMW!Esquerda))->first(), 'elemento')
120.    ),
121.    elementoChave : ATL!SimpleOutPatternElement(
122.      varName <- 'chave',
123.      type <- tipoChave,
124.      bindings <- ligacaoChave
125.    ),
126.    tipoChave : ATL!OclModelElement (
127.      name <- 'Coluna',
128.      model <- ATL!OclModel.allInstances()->select ( e | e.name =
129.        'Relacional')->first()
130.    ),
131.    ligacaoChave : ATL!Binding(
132.      propertyName <- 'name',
133.      value <- valorLigacaoChave
134.    ),
135.    valorLigacaoChave: ATL!StringExp (
136.      stringSymbol <- 'idObjeto'
137.    ),
138.    ligacaoColuna : ATL!Binding (
139.      propertyName <- 'col',
140.      value <- sequencia
141.    ),
142.    sequencia : ATL!SequenceExp (
143.      elements <- Sequence{aFonte}
144.    ),
145.    aFonte : ATL!VariableExp(
146.      referredVariable <- elementoChave
147.    ),
148.    ligacaoConj : ATL!Binding (
149.      propertyName <- 'chave',
150.      value <- sequence
151.    ),
152.    sequence : ATL!SetExp (
153.      elements <- Set{aSource}
154.    ),
155.    aSource : ATL!VariableExp(
156.      referredVariable <- elementoChave
157. )
158.}
```

Apêndice H – Especificação do Meta-Modelo da Linguagem JAVA

Este apêndice apresenta a especificação, em linguagem KM3, do meta-modelo simplificado da linguagem de programação JAVA utilizado no presente trabalho.

```
1.  package JAVA {
2.    abstract class ElementoJava {
3.      attribute nome : String;
4.    }

5.    abstract class FeatureClasse extends ElementoJava {
6.      attribute isFinal : Boolean;
7.    }

8.    abstract class MembroClasse extends FeatureClasse {
9.      reference tipo : JavaClass oppositeOf elementosTipados;
10.     reference dono : JavaClass oppositeOf membros;
11.     attribute isStatic : Boolean;
12.     attribute isPublic : Boolean;
13.   }

14.   class Campo extends MembroClasse { }

15.   class ClasseJava extends FeatureClasse {
16.     reference elementosTipados[*] : ClassMember oppositeOf tipo;
17.     reference membro[*] container : ClassMember oppositeOf dono;
18.     reference parametros[*] : FeatureParameter oppositeOf tipo;
19.     reference "package" : Package oppositeOf classes;
20.     attribute isAbstract : Boolean;
21.     attribute isStatic : Boolean;
22.     attribute isPublic : Boolean;
23.   }

24.   class Metodo extends MembroClasse {
25.     reference parametros[*] ordered container : FeatureParameter oppositeOf metodo;
26.   }

27.   class Pacote extends ElementoJava {
28.     reference classes[*] container : JavaClass oppositeOf "pacote";
29.   }

30.   class TipoPrimitivo extends ClasseJava { }

31.   class FeatureParameter extends ClassFeature {
32.     reference tipo : ClasseJava oppositeOf parametros;
33.     reference metodo : Metodo oppositeOf parametros;
34.   }
35. }

36. package TipoPrimitivos {
37.   datatype String;
38.   datatype Boolean;
39. }
```

Apêndice I – Especificação da Transformação de Modelo do Experimento 2

Este apêndice apresenta a especificação, em linguagem ATL, da especificação de transformação de modelos utilizada na primeira etapa do segundo experimento.

```
1.  module OT2JAVA;
2.  create OUT : JAVA from IN : OrdemTrabalho;

3.  rule OrdemTrabalho2Classe {
4.    from
5.      ot : OrdemTrabalho!OrdemTrabalho
6.    to
7.      saida : JAVA!ClasseJava (
8.        nome <- ot.nome
9.      )
10. }

11. rule Pessoa2Classe {
12.    from
13.      p : OrdemTrabalho!Pessoa
14.    to
15.      saida : JAVA!ClasseJava (
16.        nome <- p.nome
17.      )
18. }

19. rule EntregaProducao2Classe {
20.    from
21.      ep : OrdemTrabalho!EntregaProducao
22.    to
23.      saida : JAVA!ClasseJava (
24.        nome <- ep.nome
25.      )
26. }

27. rule PassoAtividade2Classe {
28.    from
29.      pa : OrdemTrabalho!PassoAtividade
30.    to
31.      saida : JAVA!ClasseJava (
32.        nome <- pa.nome
33.      )
34. }

35. rule PassoProcedimento2Classe {
36.    from
37.      pp : OrdemTrabalho!PassoProcedimento
38.    to
39.      saida : JAVA!ClasseJava (
40.        nome <- pp.nome
41.      )
42. }
```

```
43. rule TipoAtivo2Classe {
44.    from
45.      ta : OrdemTrabalho!TipoAtivo
46.    to
47.      saida : JAVA!ClasseJava (
48.        nome <- ta.nome
49.      )
50. }

51. rule Ativo2Classe {
52.    from
53.      a : OrdemTrabalho!Ativo
54.    to
55.      saida : JAVA!ClasseJava (
56.        nome <- a.nome
57.      )
58. }
```

Apêndice J – Descrição do Modelo PSM Gerado no Experimento 2

Este apêndice apresenta a descrição, em formato XMI, do modelo PSM gerado como resultado do processo de transformação do Experimento 2. Este modelo é igual para as etapas 1 e 2 do experimento.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns="JAVA">
  <ClasseJava name="OrdemProducao"/>
  <ClasseJava name="PassoFabricacao"/>
  <ClasseJava name="ProcedimentoFabricacao"/>
  <ClasseJava name="EntregaProducao"/>
  <ClasseJava name="Pessoa"/>
  <ClasseJava name="TipoMaterial"/>
  <ClasseJava name="Material"/>
</xmi:XMI>
```

Apêndice K – Especificação da Transformação de Alta Ordem do Experimento 2

Este apêndice apresenta a especificação, em linguagem ATL, da especificação de transformação de modelo de alta ordem utilizada na segunda etapa do segundo experimento.

```
1.  module AMW2ATL;
2.  create OUT : ATL from IN : AMW, esquerda : MOF, direita : MOF;

3.  helper def : contadorRegras : Integer = 0;
4.  helper def : nomeModulo  : String = 'OrdemTrabalho2Java';


5.  rule Module {
6.    from
7.      amw : AMW!OrdemTrabalho
8.    to
9.      atl : ATL!Module (
10.       isRefining <- false,
11.       name <- thisModule.nomeModulo,
12.       inModels <- amw.modeloEsquerda,
13.       outModels <- amw.modeloDireita,
14.       elements <- Set {amw.ownedElement}
15.     )
16. }

17. rule ReferenciaModeloEsquerda {
18.   from
19.     amw : AMW!ModelRefXMIEsquerda
20.   to
21.     atl : ATL!OclModel (
22.       metamodel <- mm,
23.       name <- 'IN'
24.     ),
25.   mm : ATL!OclModel (
26.       elements <- MOF!EClassifier.allInstancesFrom('esquerda'),
27.       name <- 'OrdemTrabalho'
28.     )
29. }

30. rule ClassificadorFonte {
31.   from
32.     fonte : MOF!EClassifier (fonte.ePackage.name = 'OrdemTrabalho')
33.   to
34.     atl : ATL!OclModelElement (
35.       name <- fonte.name
36.     )
37. }

38. rule ReferenciaModeloDireita {
39.   from
40.     amw : AMW!ModelRefXMIDireita
```

```
41.   to
42.     atl : ATL!OclModel (
43.       metamodel <- mm,
44.       name <- 'OUT'
45.     ),
46.     mm : ATL!OclModel (
47.       elements <- MOF!EClassifier.allInstancesFrom('direita'),
48.       name <-'JAVA'
49.     )
50. }

51. rule ClassificadorAlvo {
52.   from
53.     alvo : MOF!EClassifier (alvo.ePackage.name ='JAVA')
54.   to
55.     atl : ATL!OclModelElement (
56.       name <- alvo.name
57.     )
58. }

59. rule IgualdadeClasse {
60.   from
61.     amw: AMW!IgualdadeClasse
62.   to
63.     atl : ATL!MatchedRule (
64.       inPattern <- amw.end->select(e | e.oclIsTypeOf(AMW!Esquerda))->first(),
65.       outPattern <- amw.end->select(e | e.oclIsTypeOf(AMW!Direita))->first()
66.     )
67.   do {
68.     thisModule.contadorRegras <- thisModule.contadorRegras + 1;
69.     atl.name <- 'Classe2ClasseJava_' + thisModule.contadorRegras.toString();
70.   }
71. }

72. rule PadraoEntradaIgualdadeClasse {
73.   from
74.     amw: AMW!Esquerda(amw.link.oclIsTypeOf(AMW!IgualdadeClasse))
75.   to
76.     atl: ATL!InPattern (
77.       elements <- elemento
78.     ),
79.     elemento: ATL!SimpleInPatternElement(
80.       varName <- 'ot',
81.       type <-tipo
82.     ),
83.     tipo : ATL!OclModelElement (
84.       name <-MOF!EClassifier.getInstanceById('left', amw.element.ref).name,
85.       model <- ATL!OclModel.allInstances()->select(e | e.name =
86.         'OrdemTrabalho')->first()
87.     )
88. }

89. rule PadraoSaidaIgualdadeClasse {
90.   from
91.     amw: AMW!Direita(amw.link.oclIsTypeOf(AMW!IgualdadeClasse))
92.   to
93.     atl : ATL!OutPattern (
94.       elements <- elementoSaida,
95.       elements <-  elementoChave
96.     ),
97.     elementoSaida : ATL!SimpleOutPatternElement(
98.       varName <- 'saida',
99.       type <- tipoSaida,
100.      bindings <- ligacaoSaida
101.    ),
102.    tipoSaida : ATL!OclModelElement (
103.      name <- MOF!EClassifier.getInstanceById('right', amw.element.ref).name,
104.      model <- ATL!OclModel.allInstances()->select(e | e.name = 'JAVA')->first()
105.    ),
106.    ligacaoSaida : ATL!Binding(
107.      propertyName <- 'name',
108.      value <- valorLigacaoSaida
109.    ),
110.    valorLigacaoSaida: ATL!NavigationOrAttributeCallExp (
111.      name <- 'nome',
112.      source <- fonteLigacaoSaida
113.    ),
```

```
114.    fonteLigacaoSaida : ATL!VariableExp(
115.      referredVariable <- thisModule.resolveTemp(amw.link.end->select (e |
116.      e.oclIsTypeOf(AMW!Esquerda))->first(), 'elemento')
117.    )
118.}
```