Universidade Federal do Maranhão
Centro de Ciências Exatas e Tecnologia
Curso de Ciência da Computação

PAULO ROBERTO JANSEN DOS REIS

GERAÇÃO SEMIAUTOMÁTICA DE ESPECIFICAÇÃO DE CORRESPONDÊNCIAS ENTRE METAMODELOS: aplicação em QVT

São Luís
2012

PAULO ROBERTO JANSEN DOS REIS

GERAÇÃO SEMIAUTOMÁTICA DE ESPECIFICAÇÃO DE CORRESPONDÊNCIAS ENTRE METAMODELOS: aplicação em QVT

Monografia apresentada ao Curso de Ciência da Computação da Universidade Federal do Maranhão, como parte dos requisitos necessários para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Denivaldo Cicero Pavão Lopes

São Luís

2012

Reis, Paulo Roberto Jansen dos

Geração semiautomática de especificação de correspondências entre metamodelos: aplicação em QVT / Paulo Roberto Jansen dos Reis. - 2012.

56.p

Impresso por computador (Fotocópia).

Orientador: Denivaldo Cicero Pavão Lopes.

Monografia (Graduação) - Universidade Federal do Maranhão, Curso de Ciência da Computação, 2012.

1. Software - Modelagem 2. MDE 3. QVT. I.Título.

CDU 004.414.23

PAULO ROBERTO JANSEN DOS REIS

GERAÇÃO SEMIAUTOMÁTICA DE ESPECIFICAÇÃO DE CORRESPONDÊNCIAS ENTRE METAMODELOS:
aplicação em QVT

Monografia apresentada ao Curso de Ciência da Computação da Universidade Federal do Maranhão, como parte dos requisitos necessários para obtenção do grau de Bacharel em Ciência da Computação.

Aprovado em 26 de junho de 2012.

BANCA EXAMINADORA

Denivaldo Cicero Pavão Lopes

Denivaldo Cicero Pavão Lopes

Doutor em Informática

Samyr Béliche Vale

Doutor em Ciência da Computação

Zair Abdelouahab

Ph.D. em Ciência da Computação

RESUMO

Atualmente, a indústria de software tem se defrontado com uma complexidade cada vez maior no processo de desenvolvimento, manutenção e evolução de software. Visando oferecer soluções para o gerenciamento dessa complexidade, foi proposta a Engenharia Dirigida por Modelos *(MDE - Model Driven Engeneering)*. Apesar de a MDE oferecer suporte ao gerenciamento desta complexidade, a etapa de transformação entre metamodelos que é uma das principais etapas deste processo, ainda é uma tarefa manual. Este trabalho propõe uma ferramenta para gerar automaticamente as definições de transformação em QVT a partir de modelos de correspondências.

Palavras-chave: MDE. QVT. Metamodelos.

ABSTRACT

Currently, the software industry has been facing an increasing complexity in the development process, maintenance and evolution of software. In order to provide solutions for managing this complexity, the Model Driven Engineering (MDE) was proposed. Although MDE support the management of this complexity, the process of model transformation is still a manual task of codification. This work proposes a tool to generate automatically transformation definitions in QVT from mapping models.

Keywords: MDE. QVT. Metamodels.

Agradecimentos

Agradeço a toda a minha família, em especial ao meu pai por ter me concedido a oportunidade que ele e minha mãe tiveram de fazer um curso superior, e a minha mãe por suas orações e pelo auxílio dado durante todos esses anos de estudo.

Agradeço ao professor Anselmo Paiva pela confiança no meu trabalho e pela recomendação e orientação concedida para o mestrado.

Agradeço ao professor Denivaldo Lopes pela paciência, orientação e pelas aulas ministradas durante o período de projeto de pesquisa e de monografia.

Agradeço ao professor Geraldo Braz pelo acompanhamento, confiança e pelas oportunidades geradas durante o período de estágio.

Agradeço também a todos os outros professores da graduação pela contribuição direta e indireta no meu crescimento e capacitação profissional.

Agradeço ainda aos amigos de graduação Bruno Froz, Luciano Carácas, Júlio Filho, Pedro Henrique, Suellen Motta e Tássio Luz pelos bons momentos de estudo e entretenimento.

Agradeço aos amigos do Labmint Breno Nascimento, Antônio Busson, Raphael Gomes e Tiago Ramos pelo companherismo e compartilhamento de conhecimento durante longas jornadas de desenvolvimento no estágio e na Voxel Game Studio.

Agradeço aos amigos do Leserc Eduardo Bezerra, Guilherme Farias, Kleberson Pereira, Michael Costa, Ruy Oliveira e Wesley Leite pelas relevantes informações trocadas no decorrer do projeto de pesquisa.

E, por fim, agradeço à Deus, pois para Ele a ordem em que Seu nome aparece nesta folha não é importante, mas sim se minhas intenções e atitudes O agradecem verdadeiramente.

"Você quer passar o resto da sua vida vendendo água com açúcar ou você quer uma chance de mudar o mundo?".

Steve Jobs

Sumário

Lista de Figuras

Lista de Códigos

1 INTRODUÇÃO

Este capítulo descreve o contexto do problema, a problemática, a solução proposta, os objetivos geral e específico deste trabalho, a metodologia utilizada para o desenvolvimento do mesmo e ao final apresenta os capítulos que compõem a monografia.

1.1 Contexto

Atualmente, a indústria de software tem se defrontado cada vez mais com a complexidade no desenvolvimento, manutenção e evolução de sistemas de software (FRANKEL, 2003). Esta complexidade não se deve somente porque a quantidade de dados e código cresceu, mas também porque os sistemas de softwares passaram a ser distribuídos e implementados em diferentes plataformas. Com o advento da Internet, os sistemas de software passaram a existir em um ambiente largamente distribuído e heterogêneo que estimulou ainda mais a difusão de sistemas de software. Neste novo contexto, novos requisitos começaram a aparecer e novas oportunidades surgiram para a utilização de sistemas de software. Novas tecnologias e plataformas aparecem constantemente enquanto outras antigas podem ser substituídas completamente, tornando necessário o processo de migração de sistemas antigos para novas tecnologias, ou um processo de adaptação das tecnologias antigas para que possam ser integradas com as novas. Se por um lado, a indústria de software passou a receber uma alta demanda para a criação de novos projetos de sistemas de software ou a ser desafiada para evoluir sistemas de software existentes, por outro lado a indústria de software se deparou com um grau de complexidade para o desenvolvimento, manutenção e evolução de sistemas de software para o qual os paradigmas existentes não podiam mais oferecer soluções viáveis (FRANKEL, 2003).

Com o aumento da demanda, cresceu também a competitividade no mercado de desenvolvimento de softwares, criando nas empresas a necessidade de desenvolver e adaptar sistemas de software em um tempo cada vez menor e com altos padrões de qualidade, confiabilidade, segurança e adaptabilidade. Dessa forma, o tempo que uma empresa leva para desenvolver ou adaptar um software passou a ser vital para a permanência da mesma

no mercado.

Apesar de a Engenharia de Software possuir modelos no processo de desenvolvimento de software desde os anos 80 (LOPES; HAMMOUDI; ABDELOUAHAB, 2005), esses modelos são utilizados apenas nas fases iniciais do processo, tais como, análise de requisitos e modelagem do sistema, e servindo apenas como documentação do mesmo nas fases posteriores, tornando necessário implementar o sistema novamente caso haja a necessidade de migração de tecnologia ou plataforma.

A fim de fornecer soluções racionais para suportar o gerenciamento desta complexidade, a academia e organizações propuseram abordagens baseadas em modelos. Um modelo pode ser definido como "uma abstração de um sistema físico que distingue o que é pertinente do que não é com o intuito de simplificar a realidade" (KLEPPE; WARMER; BAST, 2003). Esta nova tendência tem resultado em MDE *(Model Driven Engineering)* que pode ser definida como uma abordagem na qual modelos estão no centro do desenvolvimento, manutenção e evolução de software (SCHMIDT, 2006).

1.2 Problemática

Embora a MDE forneça suporte para gerenciar a complexidade no desenvolvimento, manutenção e evolução de sistemas de software, esta é baseada na criação manual de definição de transformação entre modelos, que ainda é uma tarefa de codificação, ou seja, o problema da codificação em linguagens de programação tradicionais como Java, C++ e C#, é movido para o problema da codificação em linguagens de transformação como ATL e QVT (LOPES; HAMMOUDI; ABDELOUAHAB, 2005), que se constitui em uma tarefa enfadonha, propensa a erros e com muita verbosidade.

1.3 Solução Proposta

Uma solução possível para os problemas citados é a proposta de uma abordagem para geração semiautomática de especificações de correspondências e utilizá-las para gerar automaticamente as definições de transformação. Assim, algoritmos de *matching* de metamodelos podem gerar especificações de correspondências a partir de dois metamodelos, e estas podem ser utilizadas para gerar uma definição de transformação em QVT.

1.4 Objetivo

Aplicar a MDE para suportar a geração semiautomática de especificações de correspondências e a geração de definições de transformação em QVT-O.

1.4.1 Objetivos Específicos

Os objetivos específicos deste trabalho são:

- Estudar e desenvolver conceitos relacionados à MDE;
- Propor a extensão de um *framework* para criação e edição de modelos de correspondências e geração de definições de transformação como *plug-ins* do IDE-Eclipse.

1.5 Metodologia

Este trabalho foi desenvolvido segundo a metodologia descrita a seguir:

- Levantamento e análise bibliográfica sobre conceitos de MDE, MDA, EMF, Linguagens Específicas de Domínio (DSL), Linguagens de Transformação de Modelos, Ferramentas MDE, algoritmos de *matching* de metamodelos e a linguagem de transformação QVT;
- Estudo dos princípios de separação entre especificação de correspondência e definição de transformação;
- Determinação dos requisitos de um *framework* para a modelagem de correspondências e geração da definição de transformação;
- Aprimoramento da ferramenta MT4MDE[1] para a geração da definição de transformação em QVT-O como *plug-in* para Eclipse utilizando o EMF. Esta ferramenta foi estendida seguindo as seguintes fases: análise de requisitos, modelagem, implementação, validação e teste, e implantação;
- A ferramenta proposta foi desenvolvida na linguagem de programação Java;

[1]Esta ferramenta será apresentada na sessão 2.3.1.

- Este trabalho privilegia ferramentas e tecnologias baseadas em padrões abertos e softwares livres tais como Java, Linux, Eclipse e EMF.

1.6 Apresentação dos Capítulos

Esse trabalho esta organizado em 6 (seis) capítulos da seguinte forma.

O Capítulo 2 apresenta um levantamento bibliográfico sobre MDE, juntamente com suas principais tecnologias e abordagens, e ainda um levantamento de alguns trabalhos relacionados a este.

O Capítulo 3 apresenta a modelagem das ferramentas utilizadas para o desenvolvimento do *plug-in*, juntamente com as alterações realizadas nas mesmas para sua geração.

O Capítulo 4 exibe detalhes de como foi implementado o *plug-in*, especificando as principais classes e métodos.

O Capítulo 5 mostra alguns dos testes realizados para a validação dos resultados.

Por fim, o Capítulo 6 conclui o trabalho e faz as considerações finais sobre os resultados alcançados e sobre possíveis trabalhos vindouros nessa linha de pesquisa.

2 REVISÃO BIBLIOGRÁFICA

Este capítulo apresenta um levantamento de conceitos, tecnologias e trabalhos relacionados ao trabalho desenvolvido nessa monografia.

2.1 Model Driven Engineering (MDE)

A Engenharia Dirigida por Modelos (MDE) é uma abordagem que define modelos como foco principal e presente em todas as etapas do processo de desenvolvimento de software, com o objetivo de prover benefícios como o gerenciamento da complexidade dos softwares, a harmonização entre as diversas tecnologias existentes no mercado, redução de custos, diminuição de tempo no processo de criação do software, manutenção e/ou adaptação do software e aumento da qualidade do mesmo (LOPES, 2008) (SCHMIDT, 2006).

Modelos em MDE não são apenas documentações de software, mas também modelos interpretáveis por computadores, sendo assim precisam ser descritos formalmente para evitar ambiguidades (SOUZA Jr et al., 2009). A especificação dos elementos da construção de modelos são chamadas de linguagem de modelagem (LOPES, 2008). UML *(Unified Modeling Language)* é um dos exemplos mais usados de linguagem de modelagem (OMG, 2011c).

Uma linguagem de criação de modelos é descrita por um metamodelo, ou seja, um metamodelo é o modelo da linguagem de modelagem, sendo assim um modelo é conforme um metamodelo, por sua vez um metamodelo é criado segundo uma linguagem de metamodelagem e um metametamodelo é o modelo que define esta linguagem, logo um metamodelo é conforme um metametamodelo (LOPES, 2008). Colocando fim ao fluxo, temos que um metametamodelo tem a capacidade de descrever a si mesmo. A relação anteriormente citada é representada na Figura 2.1, juntamente com a relação de um sistema que é representado por um modelo.

As linguagens de metamodelagem mais comuns são MOF *(Meta-Object Facility)* (OMG, 2006) da OMG *(Object Management Group)* e Ecore do Projeto Eclipse (STEIN-

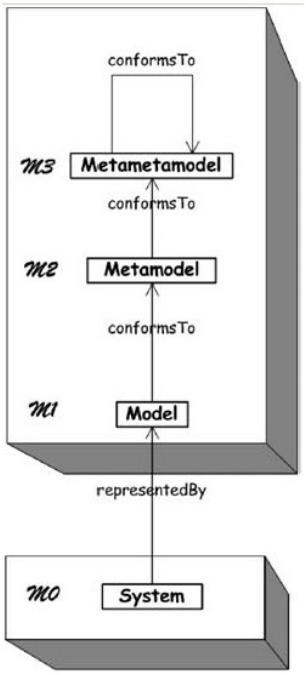

Figura 2.1: Organização de Modelos (BÉZIVIN, 2005)

BERG; BUDINSKY; PATERNOSTRO, 2008).

A técnica que permite obter um modelo através de outros modelos é chamada de transformação de modelos *(Model Transformation)*. A especificação de quais elementos do metamodelo de entrada correspondem a outros elementos do metamodelo de saída é chamada de especificação de correspondência *(Mapping Specification)*, e a descrição de forma operacional da especificação de correspondência é chamada de definição de transformação *(Transformation Definition)*. As definições de transformação são escritas usando linguagens de transformação *(Transformation Language)* (LOPES, 2008). Na Figura 2.2, "*mapping M*" é a especificação de correspondência entre o metamodelo fonte *(source MM)* e o metamodelo alvo *(target MM)*, e "*transformation M*" é a definição de transformação gerada a partir da especificação de correspondência.

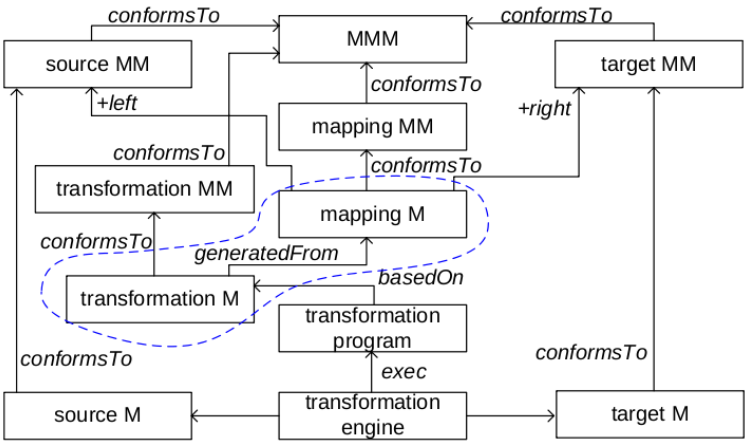

Figura 2.2: Demonstração da especificação de correspondência e da definição de transformação (LOPES; HAMMOUDI; ABDELOUAHAB, 2005)

2.1.1 Model Driven Architecture (MDA)

A Arquitetura Dirigida por Modelos (MDA) é uma proposta definida pela OMG para desenvolvimento de software, onde, seguindo a abordagem MDE, modelos são o foco principal do processo de desenvolvimento de software (KLEPPE; WARMER; BAST, 2003).

MDA dividiu modelos em *Platform Independent Model (PIM)* e *Platform Specific Model (PSM)*. Modelos Independentes de Plataforma são modelos genéricos que não dependem da plataforma onde o software será implementado e são focados na lógica e processo do negócio, já os Modelos Específicos de Plataforma contém informações específicas da plataforma e da tecnologia de implementação (OMG, 2003) (FRANKEL, 2003).

Os principais benefícios da abordagem MDA são o reuso, a facilidade de integração com outros softwares e a redução de esforços no processo de migração para outras plataformas ou tecnologias.

A *Object Management Group (OMG)* é uma associação que desenvolve e mantém normas e especificações relacionadas à interoperabilidade, portabilidade e reuso de softwares. Em (LOPES, 2008) são citadas algumas das principais tecnologias e especificações da MDA definidas pela OMG, dentre elas:

- *Unified Modeling Language (UML)*: é uma linguagem de modelagem usada para especificar, construir e documentar modelos, com uma série de diagramas padronizados para modelar aspectos estruturais e aspectos comportamentais. UML é

baseada na Orientação a Objetos e é também a especificação da OMG mais usada (OMG, 2011c);

- *XML Metadata Interchange (XMI)*: é um formato utilizado para descrever modelos usando XML, permitindo que os modelos sejam interpretáveis por computadores e possibilitando ainda o compartilhamento, a interoperabilidade e a serialização dos modelos (OMG, 2011b);

- *Object Constraint Language (OCL)*: é uma linguagem formal usada para descrever expressões em modelos UML, permitindo expressar restrições que não poderiam ser expressas apenas através dos diagramas, especificar operações e realizar requisições sobre objetos descritos em um modelo. A versão 2.0 da linguagem OCL alinhou UML 2.0 e MOF 2.0, pois compartilham um núcleo em comum, que permite que um subconjunto de OCL seja usado em ambos (OMG, 2012);

- *Meta-Object Facility (MOF)*: é uma linguagem de metamodelagem que fornece um framework para gerenciamento de metadados e um conjunto de serviços para auxiliar desenvolvedores no desenvolvimento e na interoperabilidade de sistemas dirigidos por modelos (OMG, 2006);

- *Query/View/Transformation (QVT)*: Será apresentado mais detalhadamente na sessão 2.2.2.

2.1.2 Eclipse Modeling Framework (EMF)

O EMF é um *framework* definido pelo projeto Eclipse para auxiliar no gerenciamento da complexidade de sistemas de softwares, visando dispor ferramentas de modelagem e geração de código para desenvolvimento de software segundo a abordagem MDE (STEINBERG; BUDINSKY; PATERNOSTRO, 2008).

Os principais recursos do EMF são a possibilidade de geração de código, o suporte à criação e edição de ferramentas como *plug-ins* para o Eclipse e a linguagem de metamodelagem Ecore inserida em seu núcleo.

EMF permite criar metamodelos conforme o metametamodelo Ecore através de uma ferramenta de edição gráfica para a criação de modelos e esses metamodelos podem

ser usados para gerar *plug-ins* que permitiram criar modelos conforme o novo metamodelo (LOPES, 2008).

2.2 Linguagens de Transformação

Esta sessão apresentará as linguagens ATL e QVT, que são atualmente as duas principais linguagens de transformação entre modelos.

2.2.1 ATLAS Transformation Language (ATL)

ATL é uma linguagem de transformação de modelos desenvolvida pelo ATLAS Group e LINA & INRIA, que disponibiliza aos desenvolvedores um conjunto de conceitos que tornam possíveis gerar uma série de modelos alvos (*target models*), a partir de um conjunto de modelos fonte (*source models*) (Atlas group; LINA; INRIA, 2006).

ATL é uma linguagem híbrida, pois possui estruturas de programação declarativas e imperativas. O paradigma de programação declarativo é o mais usado, pois permite expressar mapeamentos entre o os elementos do modelo fonte e alvo de forma simplificada, porém estruturas imperativas facilitam alguns mapeamentos que poderiam ser escritos apenas de forma complexa através do paradigma declarativo (Atlas group; LINA; INRIA, 2006).

Os aspectos imperativos da linguagem são realizados através de expressões OCL, estruturas de controle de fluxo, variáveis e atribuições.

ATL oferece aos desenvolvedores três tipos diferentes de unidades (Atlas group; LINA; INRIA, 2006), e estas são:

- *Modules*: Correspondem a transformação modelo-a-modelo e possibilitam especificar a forma como será gerado o modelo alvo a partir do modelo fonte. Podem utilizar *Helpers* e importar *ATL Libraries*. *Helpers* são procedimentos e podem ser vistos de forma equivalente aos métodos em Java. *ATL Libraries* são importadas através da palavra reservada *uses*, semelhante ao *import* em Java;
- *Queries*: permitem realizar requisições nos modelos e são feitas através de expressões OCL;

- *Libraries*: possibilitam a fatoração de código, permitindo aos desenvolvedores definir um conjunto de *Helpers* e utilizá-los em diferentes arquivos.

2.2.2 Query/View/Transformation (QVT)

QVT é uma linguagem de transformação padronizada pela OMG, e possui esse nome por possuir recursos para a realização de requisições *(Queries)*, visões *(Views)* e transformações *(Transformations)* de modelos (GARDNER et al., 2003), detalhadas a seguir:

- *Query*: é uma expressão avaliada sobre o modelo, e seu resultado consiste em uma ou mais instâncias ou tipos definidos no modelo fonte, ou definidos pela *Query*. Em QVT as *Queries*, são feitas através de expressões OCL;

- *View*: é um modelo completamente derivado de outro, e não pode ser modificado de forma independente do modelo derivado, ou seja, caso um dos dois sejam alterados, as mudanças serão refletidas no outro. *Views* podem ser completas, quando possuem as mesmas informações do modelo fonte, ou parciais, quando possuem apenas um subconjunto do mesmo. *Views* são geradas através de transformações. *Queries* são um tipo restrito de *View*;

- *Transformation*: é a transformação em si, que gera um modelo alvo a partir de um modelo fonte. Transformações podem ser unidirecionais ou bidirecionais.

QVT combina dois tipos diferentes de paradigmas de programação, o declarativo e o imperativo.

A Figura 2.3 mostra a arquitetura da linguagem QVT, que possui uma estrutura híbrida, com módulos responsáveis pela implementação declarativa e outros referentes à implementação imperativa.

Conforme mostrado na Figura 2.3, os elementos da arquitetura da linguagem QVT são os seguintes:

- *Relations*: A linguagem *Relations* fornece recursos para a especificação de forma declarativa de transformações como um conjunto de relações entre os modelos. *Re-*

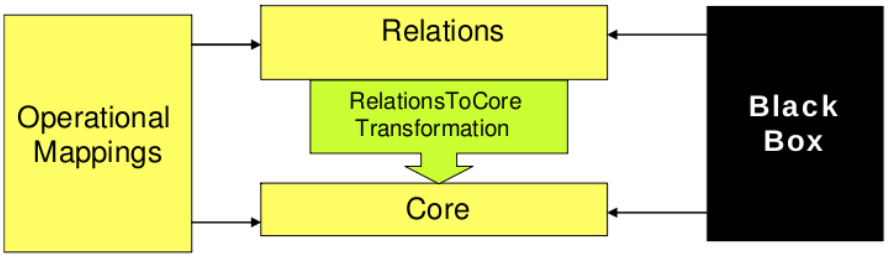

Figura 2.3: Arquiterura da linguagem QVT (OMG, 2011a)

lations possui suporte para transformações bidirecionais, complexos padrões de *matching* entre objetos, e de forma automática cria links de rastreabilidade de classes e suas instâncias, ou seja, permite o registro de informações sobre as transformações para gravar o que ocorreu durante sua execução e possibilitar ações reversas. É comumente chamada de *Query/View/Transformation Relations*, ou simplesmente QVT-R;

- *Core*: é uma linguagem declarativa com a mesma expressividade da linguagem *Relations*, porém definições de transformações escritas nessa linguagem se tornam mais verbosas, pois sua linguagem é relativamente mais simples e seus links de rastreabilidade devem ser definidos explicitamente;

- *Operational Mappings*: É a linguagem padrão para prover implementações imperativas em QVT e estende *Relations* com construções imperativas e construções OCL, permitindo a definição de transformações completamente imperativas ou a definição de abordagens híbridas, na qual operações imperativas implementam as relações em transformações relacionais. *Operational Mapping* é chamada de *Query/View/-Transformation Operational*, ou simplesmente QVT-O. Esta é mais familiar para programadores por ser similar a diversas linguagens de programação orientadas a objeto, pois possibilitam ao desenvolvedor manipular os elementos dos modelos de forma semelhante a classes e atributos, além de possuir estruturas de controle comuns como: *while*, *foreach*, *if-then-else*, etc. Uma das desvantagens desta linguagem em relação à linguagem QVT-R é o fato de ela não possuir suporte à bidirecionalidade;

- *Black Box*: é um mecanismo que permite executar código externo durante a execução da transformação, permitindo reuso de bibliotecas de domínios específicos, e o reuso de algoritmos complexos escritos em qualquer linguagem de programação ligada

ao MOF sejam executadas, ambos dificilmente implementados através do uso de expressões OCL.

2.3 Trabalhos Relacionados

Esta sessão apresenta o levantamento de alguns trabalhos relacionados a este, em relação à geração semiautomática da especificação de correspondência e em relação à comparação entre as linguagens QVT-O e ATL.

2.3.1 Geração semiautomática da especificação de correspondência

No trabalho (LOPES; HAMMOUDI; ABDELOUAHAB, 2005), uma abordagem para a geração da especificação de correspondência de forma semiautomática é proposta. Nesse trabalho, duas ferramentas são apresentadas, a MT4MDE *(Mapping Tool for Model Driven Engineering)* e a SAMT4MDE *(Semi-Automatic Matching Tool for Model Driven Engineering)*. SAMT4MDE estende MT4MDE com a geração da especificação de correspondência de forma semiautomática através de um algoritmo de *matching* entre os metamodelos (LOPES; HAMMOUDI; ABDELOUAHAB, 2005) (LOPES et al., 2006) (SOUZA Jr et al., 2009), e MT4MDE gera a definição de transformação em ATL a partir desta especificação de correspondência gerada. MT4MDE disponibiliza ainda uma interface gráfica para a exibição dos metamodelos usados na transformação e para a edição da especificação de correspondência (ver Figura 2.4).

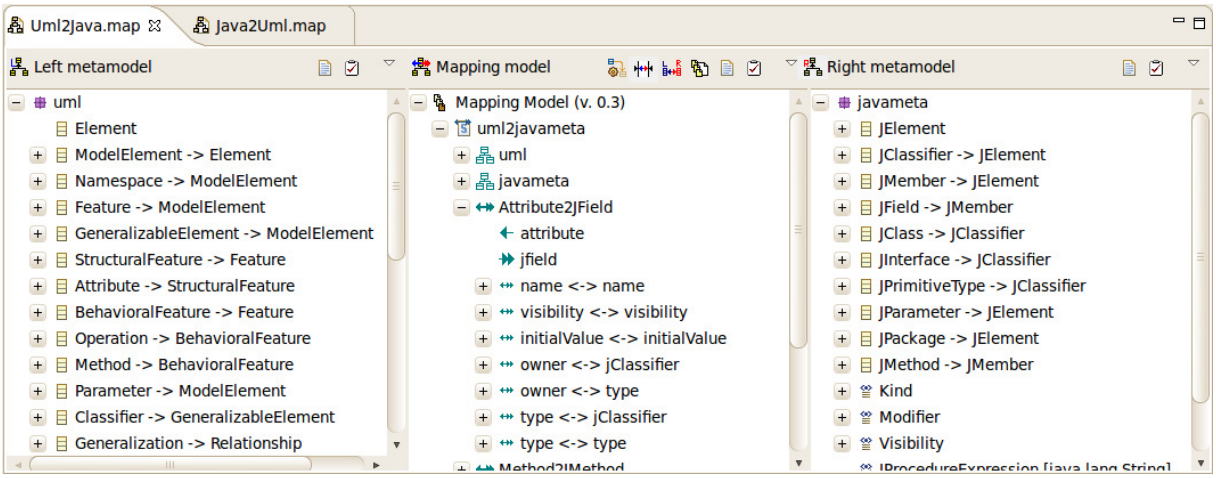

Figura 2.4: Interface gráfica da ferramenta MT4MDE

O algoritmo de *matching* consiste em encontrar elementos equivalentes ou semelhantes entre dois metamodelos. Os passos deste algoritmo são descritos a seguir, conforme citado em (LOPES; HAMMOUDI; ABDELOUAHAB, 2005):

1. Criar um conjunto C;
2. Inicializar o conjunto C como Vazio;
3. Procurar as classes que não possuem classes filhas em ambos os metamodelos;
4. Selecionar classes equivalentes ou similares de acordo com a função ϕ;

   ϕ retorna 1 se as classes forem equivalentes, 0 se forem similares e -1 se forem distintas. Essa função compara atributos e relacionamentos entre as classes dos metamodelos para realizar a classificação.
5. Colocar o conjunto de classes equivalentes ou similares em C;
6. Selecionar tipos de dados equivalentes ou similares de acordo com a função ϕ';

   ϕ' retorna 1 se os tipos de dados forem o mesmo, 0 se um tipo de dado puder ser representado pelo outro, e -1 caso não seja possível representá-lo.
7. Colocar o conjunto de tipos de dados equivalentes ou similares em C;
8. Selecionar enumerações iguais ou similares de acordo com a função ϕ";

   ϕ" retorna 1 se as enumerações forem equivalentes, 0 se forem similares e -1 se forem distintos.
9. Colocar o conjunto de enumerações equivalentes ou similares em C.

De acordo com a avaliação feita em (LOPES; HAMMOUDI; ABDELOUAHAB, 2005), SAMT4MDE e MT4MDE possuem os seguintes valores para as medidas de qualidade de acerto na geração da especificação de correspondência para o caso de teste envolvendo a transformação de UML para Java:

- *Precision*: 0.86;
- *Recall*: 0.68;
- *F-Measure*: 0.76;

- *Overall*: 0.57.

Apesar de SAMT4MDE possuir bons resultados na geração da especificação de correspondência, esta é semiautomática, pois comete erros e precisa ser validada pelo usuário, que pode remover ou adicionar mapeamentos entre os elementos.

2.3.2 Comparativo entre QVT-O e ATL

Em (JOUAULT; KURTEV, 2006), as arquiteturas das linguagens QVT e ATL são descritas e analisadas, mostrando como elas podem ser alinhadas em diversas categorias e descrevendo como códigos em ATL podem ser executados em motores QVT e como códigos em QVT podem ser executados na ATLVM *(ATL Virtual Machine)*.

O trabalho (LAARMAN, 2009) implementa a proposta mostrada no trabalho (JOUAULT; KURTEV, 2006), projetando, implementando e descrevendo um compilador baseado em técnicas de transformações entre modelos *(ACG - ATLVM Code Generator)* (ver Figura 2.5). Esse compilador permite que códigos-fonte escritos em QVT-O possam ser convertidos para ATLVM e consequentemente executados em ambientes ATL.

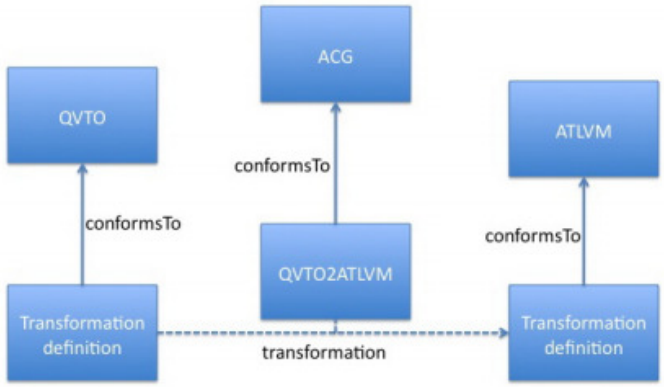

Figura 2.5: Esquema da transformação de QVT-O para ATL (LAARMAN, 2009)

2.4 Síntese

Nesse capítulo, foram apresentados conceitos relacionados à MDE como a transformação entre modelos e a separação entre especificação de correspondência e definição de transformação, juntamente com as principais tecnologias usadas atualmente como EMF,

ATL e QVT, e ainda foram mostrados trabalhos relacionados a este quanto à geração semiautomática da especificação de correspondência e a comparação entre as linguagens ATL e QVT-O.

3 MODELAGEM DO FRAMEWORK

Este capítulo apresenta a modelagem da ferramenta desenvolvida, dividindo-a em: modelagem do processo de geração de código, modelagem da especificação de correspondência utilizada para gerar o código e modelagem do funcionamento da ferramenta.

3.1 Modelagem da Geração de Código

Diante das diferenças entre as linguagens QVT-R e QVT-O, foi necessário concentrar as pesquisas em apenas uma dessas duas abordagens, para essa tomada de decisão foram levados em consideração alguns aspectos, tais como: a expressividade das linguagens, a simplicidade, o ambiente de programação e a semelhança com ATL.

Transformações em QVT-R suportam complexos padrões de *matching*, permitem alto nível de abstração e bidirecionalidade entre os modelos, enquanto transformações em QVT-O possuem um baixo nível de abstração e são unidirecionais, mas apesar disso QVT-O é mais atrativo por possuir sintaxe familiar aos programadores que usam linguagens imperativas (GUDURIC; PUDER; TODTENHOFER, 2009). Atualmente o Projeto Eclipse provê suporte à linguagem QVT-O através da instalação de um *plug-in*, enquanto QVT-R carece de boas implementações, possuindo o Eclipse MEDINI como um dos ambientes de programação mais utilizados. Dentre as duas abordagens, QVT-O assemelha-se mais com ATL e possui maior potencial de interoperabilidade com esta (LAARMAN, 2009) (JOUAULT; KURTEV, 2006).

Por esses motivos, QVT-O foi a linguagem escolhida para a geração da definição de transformação no *plug-in* proposto.

QVT-Core não foi citado por ser a mais simples das linguagens, a mais verbosa e por não registrar os links de rastreabilidade automaticamente.

Como já citado na sessão 2.3.1, a ferramenta MT4MDE é uma ferramenta que permite a geração automática da definição de transformação através de uma interface visual, e a ferramenta SAMT4MDE estende MT4MDE com a geração semiautomática da

especificação de correspondência, através de um algoritmo de *matching* entre metamodelos. Essas duas ferramentas foram as escolhidas para a implementação deste trabalho, pelos seguintes motivos:

- Por apresentarem bons resultados na geração da especificação de correspondência, como anteriormente citado em 2.3.1;
- Por ser modelada de forma a ser facilmente estendida para novas linguagens de transformação;
- Por realizar a geração da definição de transformação apenas em ATL.

Um dos motivos que nos levaram a estender as ferramentas MT4MDE e SAMT4-MDE foi o fato de possibilitar ao desenvolvedor escolher qual linguagem de transformação deseja usar, baseado no seu conhecimento sobre cada uma, no ambiente de transformação disponível para a realização de transformação, e no desempenho de cada uma das linguagens (AMSTEL et al., 2011).

A Figura 3.1 mostra a arquitetura dessas ferramentas, possibilitando uma melhor compreensão de seu funcionamento e do processo de acréscimo de novas linguagens de transformação como opção para a geração das definições de transformação.

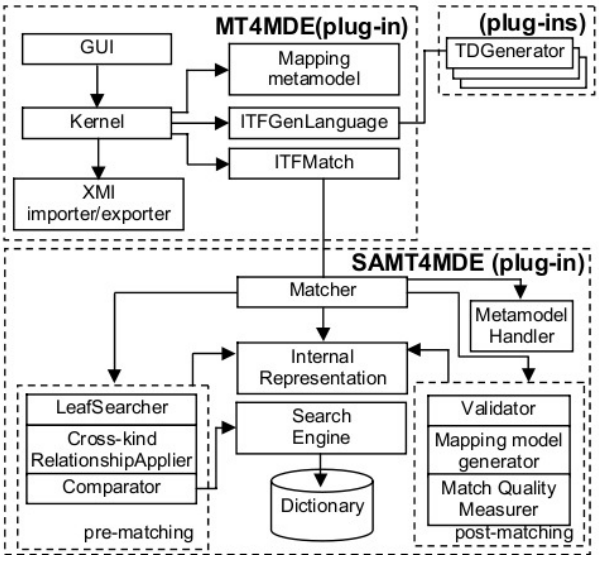

Figura 3.1: Arquitetura das ferramentas MT4MDE e SAMT4MDE (LOPES; HAMMOUDI; ABDELOUAHAB, 2005)

MT4MDE possui um módulo chamado ITFGenLanguage (Interface de Geração de Linguagem), que é uma interface que manipula TDGenerator (Gerador da Definição

de Transformação), permitindo adicionar novos *plug-ins* geradores da definição de transformação em outras linguagens de transformação através da implementação de ITFGenLanguage.

Nas versões das ferramentas apresentadas em (LOPES; HAMMOUDI; ABDELOUAHAB, 2005), há um TDGenerator que realiza a geração da definição de transformação em ATL, chamado de ATLGen. Visando estender esta ferramenta para a linguagem QVT-O, criou-se a classe QVTGen. Na Figura 3.2, é exibida a modelagem das classes QVTGen e ATLGen, que implementam ITFGenLanguage.

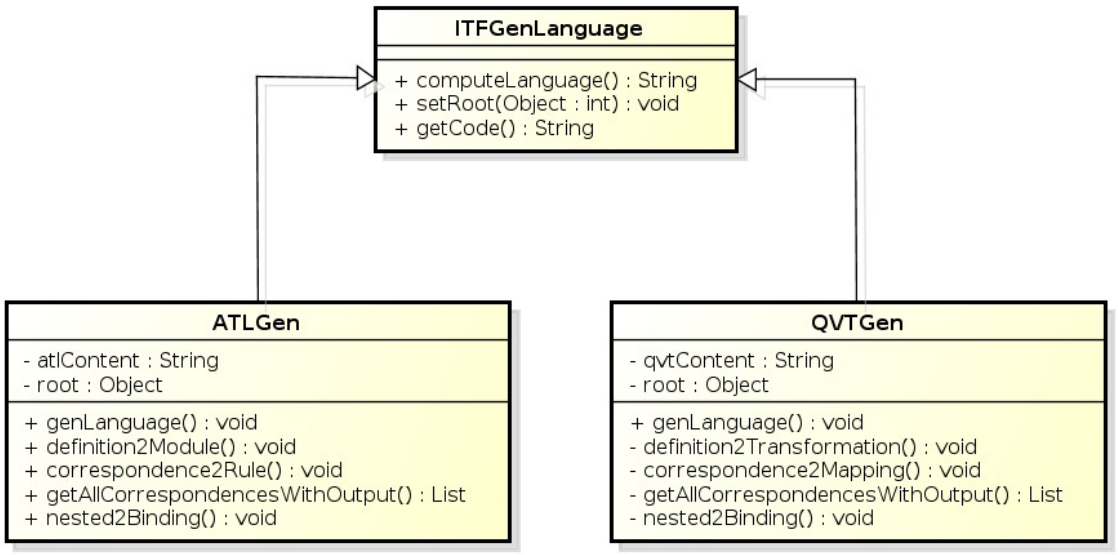

Figura 3.2: Modelagem da classe QVTGen

3.2 Modelagem da Especificação de Correspondência

Como a geração da definição de transformação é realizada através da especificação de correspondência gerada em *Mapping Model Generator* (ver Figura 3.1), então faz-se necessário compreender como o *mapping* é modelado. A Figura 3.3 mostra a representação dos elementos que constituem a especificação de correspondência, e estes são:

- *Element*: é apenas uma generalização para os outros elementos do modelo;
- *Historic*: é o elemento que possibilita manter um histórico das decisões tomadas durante o processo de criação do mapeamento. *Historic* possui uma coleção de *Definitions*;

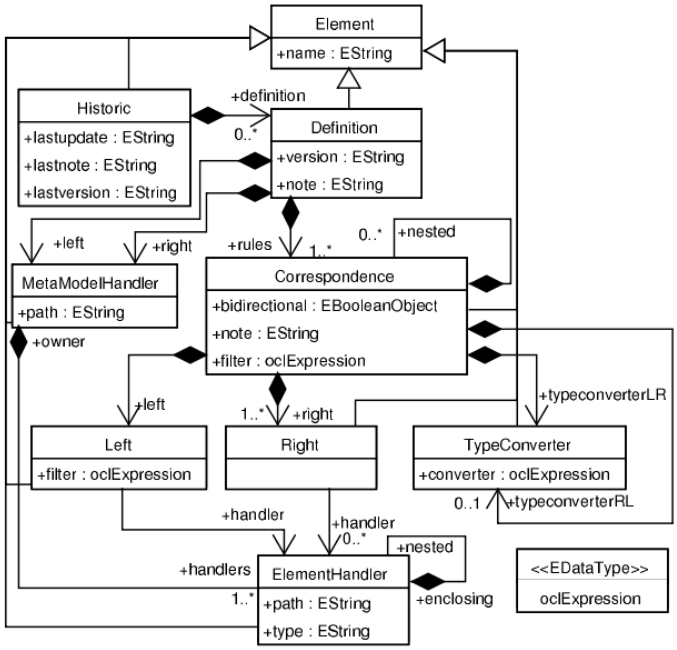

Figura 3.3: Metamodelo da especificação de correspondência (LOPES et al., 2005)

- *Definition*: é o elemento que contém as correspondências entre os elementos dos metamodelos fonte e alvo. *Definitions* possuem um elemento *Left* e um ou mais elementos *Right*;

- *Correspondence*: especifica a relação entre um elemento do metamodelo fonte e um ou mais elementos do metamodelo alvo. Correspondences possuem dois *TypeConverters* que permitem a conversão de tipos de dados de elementos de um metamodelo para o outro;

- *Left*: é usado para representar elementos do metamodelo fonte;

- *Right*: é usado para representar elementos do metamodelo alvo;

- *MetaModelHandler*: elemento usado para navegar nos metamodelos, sem realizar alterações nos mesmos;

- *ElementHandler*: é usado para acessar os elementos que estão sendo mapeados sem modificá-los;

- *TypeConverter*: permitem a conversão de tipos de dados entre os elementos dos metamodelos. Se os tipos de dados forem iguais o mapeamento é direto, mas se forem apenas similares será feita uma conversão de tipos através do uso da expressão em OCL contida no *TypeConverter*.

3.3 Modelagem do Funcionamento da Ferramenta

As possíveis ações realizadas pelo usuário do *plug-in* desenvolvido estão representadas no diagrama de caso de uso da Figura 3.4.

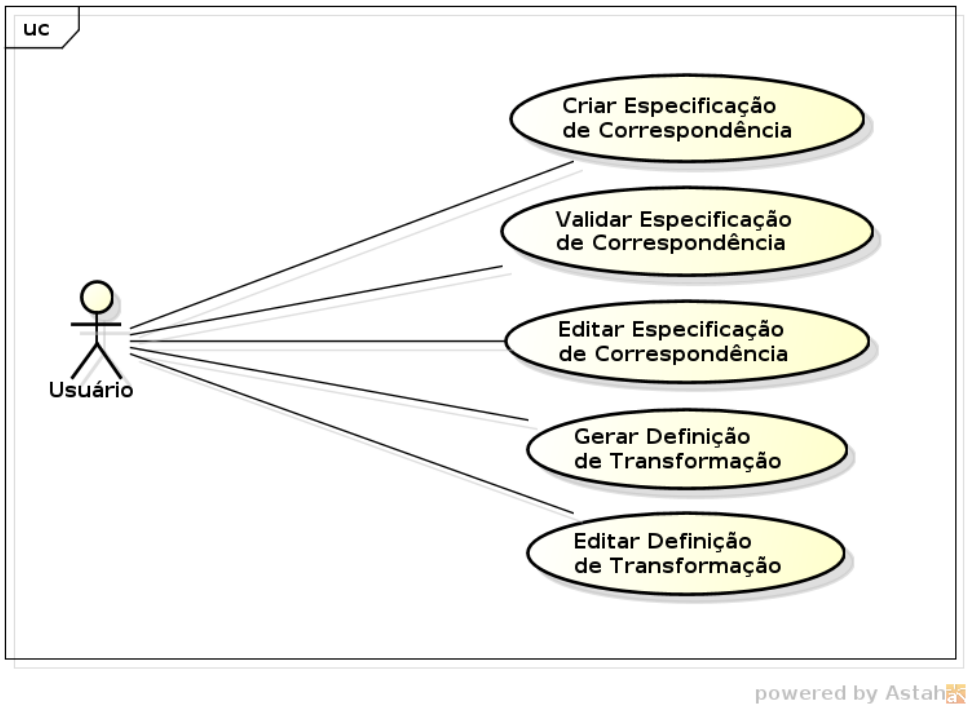

Figura 3.4: Diagrama de Caso de Uso da Ferramenta

O funcionamento da ferramenta foi modelado de acordo com o diagrama de atividades da Figura 3.5, que descreve as ações do usuário e as atividades realizadas por cada uma das ferramentas utilizadas até a geração da definição de transformação.

O início do processo é dado pelo usuário que cria uma nova especificação de correspondência, dando como entrada dois metamodelos e especificando qual destes é o metamodelo fonte e qual é o metamodelo alvo. Após a leitura dos metamodelos, a ferramenta SAMT4MDE realiza a geração da especificação de correspondência de forma semiautomática de acordo com o algoritmo de *matching* apresentado na sessão 2.3.1, esta especificação de correspondência é apresentada ao usuário, que pode realizar alterações caso esta não esteja completamente correta. Com a validação da especificação de correspondência o usuário pode então escolher a linguagem a qual deseja gerar a definição de transformação, recebendo-a em um editor de texto.

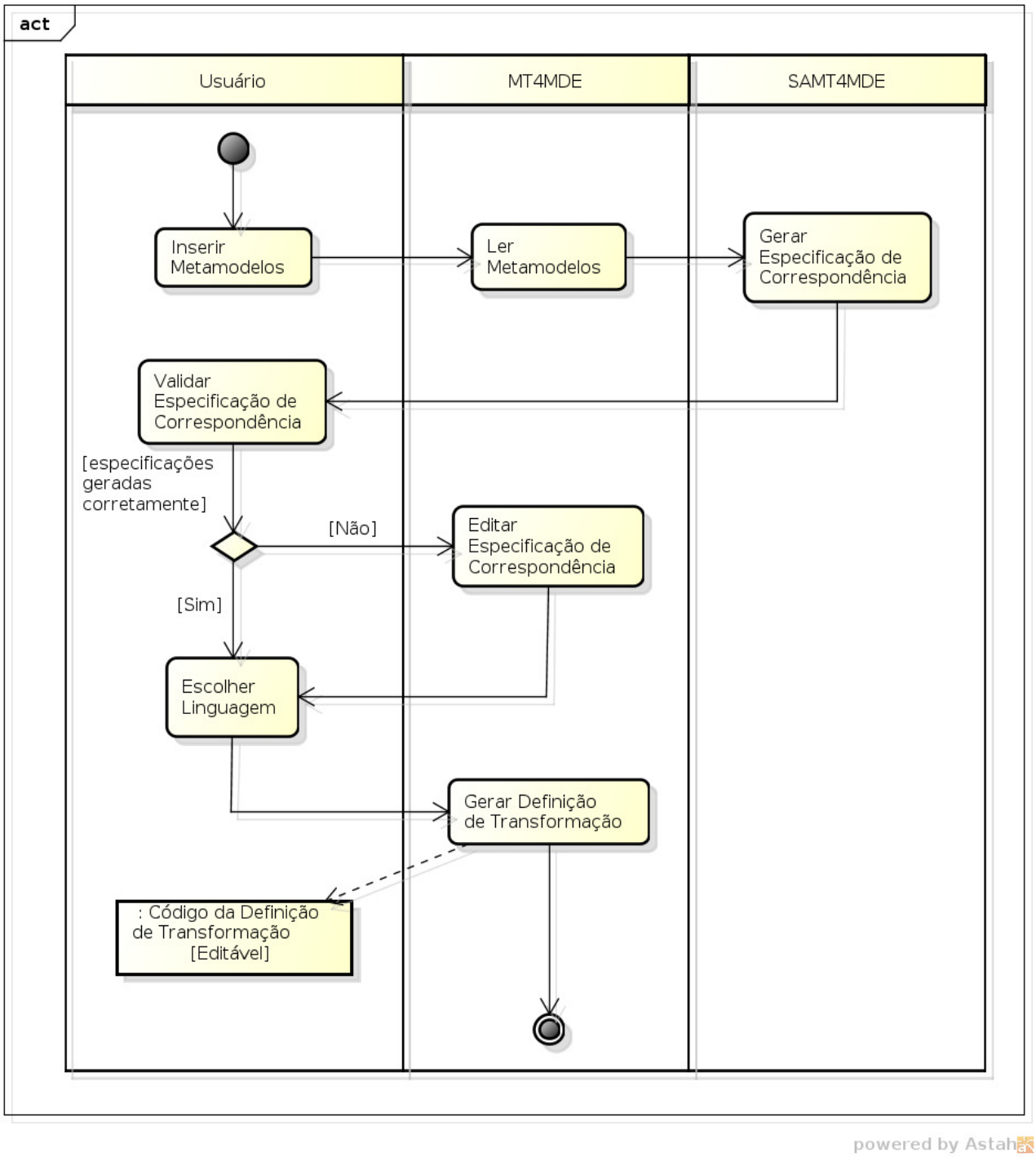

Figura 3.5: Diagrama de Atividades da Ferramenta

3.4 Síntese

Nesse capítulo, foram apresentados os motivos pelos quais a linguagem QVT-O foi a linguagem escolhida para a geração da definição de transformação e a modelagem do módulo QVTGen.

Foram apresentados ainda como estender a ferramenta MT4MDE para novas

linguagens de transformação, e como esta ferramenta modela a especificação de correspondência usada na geração de código.

Ao final, foram apresentados os diagramas de caso de uso e de atividades para modelar as funcionalidades disponíveis ao usuário e o funcionamento da ferramenta.

4 IMPLEMENTAÇÃO DO *PLUG-IN* PARA O ECLIPSE

ATL e QVT possuem semelhanças devido ambas derivarem da OCL, e como o módulo que gerava as regras de transformação em ATL já estava implementado, este foi tomado como modelo para a implementação da geração de código em QVT-O. Dessa forma, foram utilizados os protótipos dos métodos da classe ATLGen e modificados para a geração da classe QVTGen.

Para a implementação do *plug-in*, a linguagem de programação Java foi a escolhida, pois as duas ferramentas citadas na sessão 2.3.1 são implementadas nessa linguagem.

4.1 Estendendo MT4MDE com QVTGen

ITFGenLanguage é uma interface que contém os métodos necessários para o processo de geração de linguagem.

O conteúdo de ITFGenLanguage está especificado na Listagem 4.1.

Listagem 4.1: Código-fonte do arquivo ITFGenLanguage.java

```
1 package mapping.presentation;
2
3 public interface ITFGenLanguage {
4         public void setRoot(Object Root);
5         public String computelanguage();
6         public String getCode();
7 }
```

Esses métodos são essenciais e são chamados após o usuário iniciar o processo de geração da linguagem, onde o método *setRoot()* é chamado para atribuir um elemento do tipo Historic ao elemento raiz, e o método *computelanguage()* inicia o processo de geração de código e ao final retorna uma String contendo todo o código gerado pela classe, e este é exibido ao usuário em um editor de texto.

Como mostrado na Figura 3.2, QVTGen utiliza métodos e atributos principais semelhantes aos usados em ATLGen, com implementações internas distintas, pois apesar da semelhança entre as linguagens, QVT-O e ATL possuem diferenças na forma em que

são escritas as regras de transformação, entre essas estão:

- A forma de referenciar os metamodelos;
- A forma de definir a ordem de execução das regras de transformação;
- A forma de referenciar os literais;
- A forma de mapear os relacionamentos.

Essas diferenças são mais detalhadas a seguir na descrição dos métodos da classe QVTGen em comparação aos métodos da classe ATLGen:

1. Método: *genLanguage()*
   - Parâmetros de Entrada: Objeto do tipo *Historic*;
   - Acão executada: Percorre todas as *Definitions* contidas em *Historic*, chamando o método *definition2transformation()* para cada uma delas.
2. Método: *definition2Transformation()*
   - Parâmetros de Entrada: Objeto do tipo *Definition*;
   - Ação executada: Escreve os *modelTypes*, as *transformations* e a *main*, e percorre as *Correspondences* contidas em *Definition*, chamando o método *correspondence2Mapping()* para cada delas.
3. Método: *correspondence2Mapping()*
   - Parâmetros de Entrada: Objeto do tipo Correspondence;
   - Ação executada: Escreve o *mapping*, adiciona a chamada do mesmo na *main* e chama o método *nested2Binding()*, passando como parâmetro uma lista que contêm o retorno do método *getAllCorrespondencesWithOutput()* e o nome do elemento de entrada do *mapping*.
4. Método: *nested2Binding()*
   - Parâmetros de Entrada: Uma variável *String* e uma lista de objetos do tipo *Correspondences*;

- Ação executada: Escreve todo o conteúdo dos *mappings*, ou seja, as atribuições dos elementos do metamodelo fonte aos elementos do metamodelo alvo, como atributos, literais, relacionamentos e expressões OCL's.

4.2 Detalhando QVTGen

As expressões regulares citadas a seguir estão conforme a notação EBNF e correspondem às regras de como a linguagem QVT-O foi gerada nesse trabalho, sendo estas apenas uma parte simplificada das regras gerais. As regras gerais possuem mais opções e a utilização de mais recursos da linguagem, essas podem ser encontradas em (OMG, 2011a).

Nessas expressões foi escolhido não utilizar diversos símbolos não-terminais opcionais, ou seja, os elementos da expressão definidos como "<nome_do_simbolo>?", não são elementos obrigatórios e muitos destes não estão sendo utilizados nessa versão do *plug-in*.

A expressão *modeltype* é a declaração do metamodelo que contém o conjunto de classes e atributos a serem utilizados na transformação, esses são definidos conforme as expressões regulares abaixo:

<*modeltype*>::= 'modeltype' <*identifier*>'uses' <*packageid*>';'
<*packageid*>::= <*scoped_identifier*>'(' <*uri*>')'
<*uri*>::= <*STRING*>

São geradas duas expressões *modeltype*, uma para o metamodelo fonte e outra para o alvo. A URI gerada por padrão é endereço do *namespace*, contido no atributo *nsURI* do elemento raiz do arquivo Ecore do metamodelo.

ATL diferencia-se de QVT-O nesse aspecto, por não referenciar os metamodelos através de uma URI no código.

Transformation representa a definição de uma transformação unidirecional que é expressa imperativamente. Esta declara os metamodelos definidos como *modeltypes*, definindo qual deles é o metamodelo fonte (*in*) e qual é o alvo (*out*). *Tranformations* são escritas de acordo com as seguintes regras:

<*transformation*>::= 'transformation' <*identifier*><*transformation_signature*>
<*transformation_signature*>::= '(' <*param_list*>? ')'

*<param_list>::= <param>(',' <param>)**

<param>::= <param_direction>? <declarator>

<param_direction>::= 'in' | 'inout' | 'out'

<declarator>::= <identifier>':' <scoped_identifier>

Nesse trabalho, o parâmetro de direção '*inout*' não está sendo usado.

Transformations em QVT-O são semelhantes a *Modules* em ATL e são mostrados suas semelhanças na Listagem 4.2 a seguir:

Listagem 4.2: Comparação entre *Modules* em ATL e *Tranformations* em QVT-O

```
1 --Module in ATL
2 module module_name;
3 create output_id : output_model from input_id : input_model;
4
5 --Tranformation in QVT-O
6 transformation transf_name ( in input_id : input_model, out output_id : output_model);
```

A *main* é o ponto de entrada da execução *(entry operation)* e em seu corpo deve conter uma lista de expressões que serão executadas sequencialmente. Esta não possui parâmetros, mas pode acessar qualquer elemento global dos metamodelos. Sua simples escrita é definida abaixo:

<entry>::= <entry_header><expression_block>';'?

<entry_header>::= 'main()'

<expression_block>::= '{' <expression_list>? '}'

ATL possui um ponto de entrada chamado de *Called Rules*, porém a classe ATL-Gen não gera código com esse ponto de entrada, tornando-se diferente da forma como o método *definition2Transformation()* foi escrito em QVTGen.

A Listagem 4.3 mostra o trecho de código do método *definition2Transformation()*, que mostra como são gerados os *modeltypes*, a *transformation* e a *main*.

Listagem 4.3: Trecho de código do método *definition2Transformation* em Java

```
1  private void definition2Transformation(Definition definition) {
2          String path = ResourcesPlugin.getWorkspace().getRoot().getRawLocation().toString();
3          String sourceModelPath, targetModelPath;
4          String targetName = definition.getTarget().getName();
5          String sourceName = definition.getSource().getName();
6
7          String transformation =
8                  "transformation " + definition.getName() +
9                  "( in " + sourceName + " : " + sourceName.toUpperCase() +
10                 ", out " + targetName + " : " + targetName.toUpperCase() + ");\n\n";
11
12         main = "main(){\n";
13         auxMain = "\t\t" + sourceName + ".objectsOfType(";
```

```
14
15          ...
16
17          String modeltype = (inIsLoaded && outIsLoaded)?
18                          "modeltype " + sourceName.toUpperCase() + " uses " + sourceName + "('" +
                                inEcore.getURI() + "');\n" +
19                          "modeltype " + targetName.toUpperCase() + " uses " + targetName + "('" +
                                outEcore.getURI() + "');\n\n":
20                          "modeltype " + sourceName.toUpperCase() + " uses " + sourceName + "('');\n" +
21                          "modeltype " + targetName.toUpperCase() + " uses " + targetName + "('');\n\n";
22          ...
```

Em QVT-O, *mapping* é uma operação que implementa o mapeamento entre um ou mais elementos do modelo fonte e um ou mais elementos do modelo alvo e são definidos conforme as seguintes regras:

<mapping>::= <mapping_header>'{' <mapping_body>'}' ';'?

<mapping_header>::= 'mapping' <scoped_identifier>'::' <identifier>'() :' <scoped_identifier>

<mapping_body>::= <init_section>? <population_section>? <end_section>?

<population_section>::= <expression_list>|'population' <expression_block>

Nesse trabalho as sessões *init* e *end* não estão sendo utilizados e a sessão *population* está sendo usada de forma implícita.

Mappings em QVT-O são semelhantes a *Rules* em ATL e um comparativo entre os dois está expresso na Listagem 4.4 a seguir:

Listagem 4.4: Comparação entre *Rules* em ATL e *Mappings* em QVT-O

```
1  --Rule in ATL
2  rule rule_name{
3          from source_identifier : input_model!source_element
4          to target_identifier : output_model!target_element
5          (
6                  --Body of rule
7          )
8  }
9
10 --Mapping in QVT-O
11 mapping input_model.source_element :: map_name () : output_model.target_element {
12         --Body of mapping
13 }
```

Para a escrita da lista de expressões da *main* foi escolhido utilizar uma operação sobre os elementos do modelo de entrada que é uma operação que foi adicionada à biblioteca padrão da OCL (OMG, 2011a). Essa operação é definida como:

Model.objectsOfType(OclType) : Set(Element)

Em *Model* é colocado o modelo fonte e em *OclType* é colocado um tipo do elemento

desse modelo.

Essa operação retorna uma lista de objetos do modelo *Model* do tipo *OclType*, e para cada elemento dessa lista é realizado o mapeamento especificado ao lado.

A Listagem 4.5 mostra o trecho de código do método *correspondence2Mapping()*, que mostra como são gerados os *mappings* e como são adicionadas suas chamadas na *main*.

Listagem 4.5: Trecho de código do método *correspondence2Mapping* em Java

```
1  private void correspondence2Mapping(Correspondence correspondence){
2          ...
3
4          String name = correspondence.getName();
5
6          if (name == null || name.length() == 0){
7                  if (correspondence.getInput() != null && correspondence.getOutput() != null){
8                          if (correspondence.getOutput().size() > 0){
9                                  if (correspondence.getInput().getName() != null &&
10                                         ((Output)correspondence.getOutput().get(0)).getName()
                                               != null)
11
12                                 name = correspondence.getInput().getName() + "2" +
13                                 ((Output)correspondence.getOutput().get(0)).getName();
14                         }
15                 }
16         }
17
18         ...
19
20         Input input = correspondence.getInput();
21         String inputName = input.getName();
22         String outputName = ((Output)correspondence.getOutput().get(0)).getName();
23         if (input != null){
24                 if (input.getHandler() != null){
25                         inputName = input.getHandler().getName();
26
27                         if (correspondence.getOutput() != null){
28                                 if (correspondence.getOutput().size() > 0){
29                                         int i=0;
30                                         for (Iterator it = correspondence.getOutput().iterator(); it.
                                                       hasNext(); i++){
31                                             Output output = (Output)it.next();
32                                             if (output.getHandler() == null)
33                                                     continue;
34                                             if (output.getHandler().getOwner() == null)
35                                                     continue;
36                                             if (i==0)
37                                                     outputName = output.getHandler().getName();
38
39                                             auxContent += "\nmapping " + inputName + "::" + name +
                                                       "() : " + outputName + "{\n";
40                                             main += auxMain + inputName + ") -> map " + name + "()
                                                       ;\n";
41         ...
```

Expressões de atribuição são feitas da mesma forma em ATL e QVT-O, modificando-se apenas o símbolo de atribuição "<-" pelo ":=", porém a forma como os literais são referenciados e a forma como os relacionamentos são mapeados são distintas. Os literais

em QVT-O são referenciados da mesma forma que em OclType, pois em ambas é colocado o nome do *enumeration* seguido de "::" e do nome do literal, já em ATL não se faz necessário colocar o nome do *enumeration*, precisando apenas do prefixo "#" e do nome do literal. Essa pequena diferença, gerou um problema, pois as ferramentas utilizadas, MT4MDE e SAMT4MDE, não guardam em sua especificação de correspondência a informação do atributo *eType* (tipo do elemento) contido no arquivo .ecore do metamodelo, e o nome do *enumeration* está no conteúdo desse atributo. Para adquirir esse elemento se fez necessário ler o arquivo do metamodelo ao invés de usar apenas os dados contidos na especificação de correspondência, e para este foi utilizado a biblioteca JDOM (HUNTER, 2002) do Java, que possibilita a leitura e a realização de uma série de operações em arquivos XML.

Os relacionamentos foram resolvidos utilizando uma das operações *resolve* de QVT-O. Esse tipo de operação permite inspecionar o rastreamento de objetos, para recuperar objetos alvos criados ou atualizados por mapeamentos executados em objetos fonte. A operação usada foi a *resolveone*. A leitura do metamodelo também foi necessária nesse caso, pois o nome do elemento no qual o elemento submetido à operação deve ser resolvido, também estava dentro do atributo *eType*.

O atributo *nsURI* utilizado na expressão *modeltype* também é lido diretamente do arquivo .ecore através do JDOM, e caso haja algum problema na leitura do arquivo, será apresentada uma mensagem de erro ao usuário e o código será gerado incompleto, com comentários no lugar do conteúdo dos atributos não lidos. A manipulação dos arquivos .ecore através da utilização do JDOM são feitos de forma separada da classe geradora de código através da classe XMIModelReader.java (ver Anexo II), que possui métodos que realizam a leitura de cada um dos atributos necessários.

Para acrescentar a geração da linguagem QVT-O às ferramentas MT4MDE e SAMT4MDE, basta colocar o arquivo responsável pela geração da linguagem (QVTGen.java) juntamente com o arquivo responsável pela leitura do metamodelo (XMIModelReader.java) na pasta *PluginMapping.editor*, no package language.generator, adicionar a biblioteca do JDOM no CLASSPATH do Java e editar o arquivo *plugin.properties* para exibir na interface gráfica a opção referente à nova linguagem. Nesse arquivo, deve ser adicionado o nome da linguagem no conteúdo de *_UI_Languages* e o nome do arquivo responsável por gerar aquela linguagem em *_UI_Languages_classes*.

O código-fonte completo do arquivo QVTGen.java é apresentado no Anexo I.

4.3 Síntese

Nesse capítulo, foram mostrados detalhes da implementação do módulo QVTGen, descrevendo seus principais métodos, e como estes estão gerando código na linguagem QVT-O.

Por fim, é mostrado como adicionar o módulo desenvolvido às ferramentas MT4-MDE e SAMT4MDE.

5 VALIDAÇÃO E TESTE

Para a validação da ferramenta, testes foram realizados para verificar se o código gerado corresponde ou aproxima-se de um código em QVT-O escrito manualmente por um programador. Nos testes, a sintaxe, a semelhança do código gerado automaticamente com o manual, e os resultados das transformações foram analisados.

A sintaxe do código gerado foi avaliada usando o próprio analisador sintático do *plug-in Operational QVT* no Eclipse, e neste, o código gerado não apresenta nenhum erro sintático, e nenhum *warning* para os casos de testes usados na validação. É importante ressaltar que, caso a leitura do arquivo do metamodelo não seja realizada com sucesso, o código gerado conterá erros nos *modeltypes*, literais e relacionamentos. Nesse caso serão geradas instruções em forma de comentários no código para auxiliar o usuário na correção dos erros.

O código da definição de transformação é gerado com endentação semelhante à usada pela maioria dos programadores para facilitar a compreensão do código e simplificar o processo de edição do mesmo, caso haja necessidade.

Para analisar se o código gerado estava realizando corretamente a transformação entre os dois metamodelos, os resultados de transformações com o código gerado em QVT-O, e os resultados de transformações com o código gerado em ATL foram comparados. Pois como o código gerado em ATL já realiza a transformação de forma correta, então basta verificar se o PSM resultante desta transformação é equivalente ao PSM resultante da transformação em QVT-O, para mostrar que a classe geradora de código em QVT-O está correta.

5.1 Estudo de Caso (Uml2Java)

Uma das transformações realizadas foi a transformação de UML para Java, ambos conforme o metamodelo Ecore. Os itens a seguir mostram os passos do processo de realização dessa transformação:

1. Criar um novo projeto;
2. Colocar os metamodelos na pasta do projeto;
3. Criar um novo *Mapping* (*New->Other->New Mapping Model*);
4. Especificar o metamodelo fonte (UML) e o alvo (Java);
5. Realizar o *Matching* entre os dois metamodelos;
6. Validar a especificação de correspondência (ver Figura 5.1);
7. Gerar a definição de transformação em QVT-O ou em ATL (ver Figura 5.2);
8. Utilizar os motores de transformação para a geração do modelo alvo em Java, dado um diagrama de classe conforme UML (ver Figura 5.3).

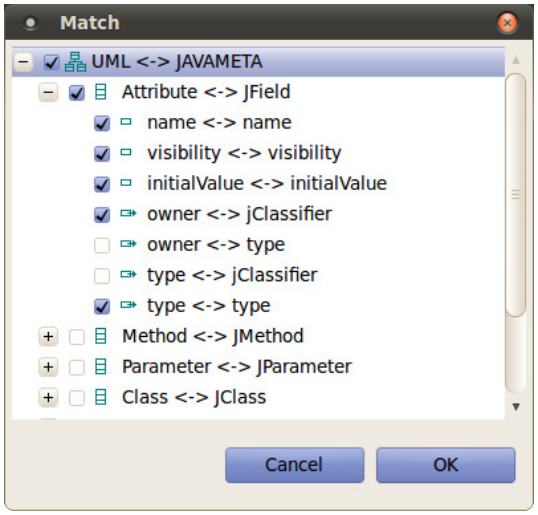

Figura 5.1: Interface gráfica para a edição da especificação de correspondência gerada

O Anexo III apresenta o código-fonte completo gerado através desta transformação.

Após a execução das transformações em QVT-O e em ATL, utilizando o modelo fonte citado, comparou-se os modelos alvos gerados, e identificou-se a equivalência entre eles, diferenciando-se apenas na codificação do arquivo XML. A Listagem 5.1 mostra os PSMs conforme o metamodelo Java gerados pelas transformações.

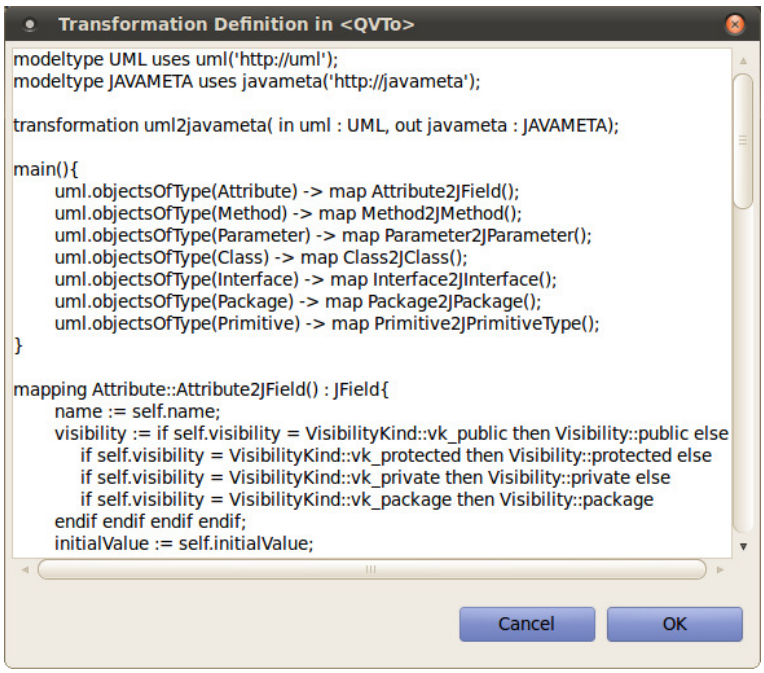

Figura 5.2: Definição de transformação em QVT-O gerada através do *plug-in* construído

Figura 5.3: Diagrama de classe usado como modelo fonte nos testes realizados

Listagem 5.1: Modelos gerados pela transformação em QVT-O e em ATL

```
1  --PSM generated by QVT-O transformation
2  <?xml version="1.0" encoding="UTF-8"?>
3  <javameta:JPackage xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.org
       /2001/XMLSchema-instance" xmlns:javameta="http://javameta" xsi:schemaLocation="http://javameta
       javameta.ecore" name="Model_IN">
4    <jElement xsi:type="javameta:JClass" name="Author">
5      <jMember xsi:type="javameta:JField" name="name"/>
6      <jMember xsi:type="javameta:JField" name="surname"/>
7      <jMember xsi:type="javameta:JField" name="gender"/>
8      <jMember xsi:type="javameta:JField" name="age"/>
9    </jElement>
10 </javameta:JPackage>
11
12
13 --PSM generated by ATL transformation
14 <?xml version="1.0" encoding="ISO-8859-1"?>
15 <javameta:JPackage xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.org
       /2001/XMLSchema-instance" xmlns:javameta="http://javameta" name="Model_IN">
16   <jElement xsi:type="javameta:JClass" name="Author">
17     <jMember xsi:type="javameta:JField" name="name"/>
18     <jMember xsi:type="javameta:JField" name="surname"/>
19     <jMember xsi:type="javameta:JField" name="gender"/>
20     <jMember xsi:type="javameta:JField" name="age"/>
21   </jElement>
22 </javameta:JPackage>
```

Para a realização dos testes citados acima, foi utilizado o *Eclipse Indigo Modeling Tools* com os seguintes *plug-ins*:

- *Operational QVT* e *ATL - ATLAS Transformation Language*, usados como motores de transformação e analisadores sintáticos para QVT-O e ATL;
- *Ecore Tools*, usado para criar metamodelos conforme Ecore;
- *UML2 Tools SDK*, usado como editor dos modelos conforme UML, usados como PSMs na realização das transformações.

5.2 Síntese

Nesse capítulo foram apresentados como o módulo QVTGen foi avaliado, juntamente com um passo-a-passo de como realizar transformações na ferramenta, e as ferramentas usadas na realização dos testes.

Nos testes, foi mostrado um caso de estudo da transformação de UML para Java, mostrando a sintaxe do código gerado e o resultado da transformação para um diagrama de classes conforme UML.

6 CONCLUSÃO

Este trabalho apresentou a aplicação de conceitos relacionados à MDE na extensão da ferramenta MT4MDE, que funciona como *plug-in* para o Eclipse, tornando-a capaz de gerar o código da definição de transformação na linguagem QVT-O através da especificação de correspondência entre dois metamodelos, que por sua vez, é gerada de forma semiautomática pela ferramenta SAMT4MDE.

6.1 Contribuições

A geração da definição de transformação em QVT-O através da ferramenta desenvolvida, contribui com a abordagem de Engenharia Dirigida por Modelos e com o mercado em aspectos relevantes como:

- A automatização de etapas do processo de desenvolvimento e/ou adaptação de softwares;
- A aceleração no tempo de desenvolvimento e/ou adaptação de sistemas;
- A redução de erros inseridos durante o processo manual de codificação da definição de transformação.

6.2 Trabalhos Futuros

Como propostas para trabalhos futuros temos:

- A criação do módulo QVTRGen, capaz de realizar a geração da definição de transformação em QVT-R;
- Possibilitar ao usuário escolher entre diferentes formas que o código é gerado;
- A criação de uma extensão da ferramenta para auxiliar a criação de *queries*;

- Expandir os recursos da linguagem e o número de palavras reservadas usadas nas gerações de código;

- A integração com um motor de transformação para permitir que após a validação da especificação de correspondência seja possível passar como entrada um modelo fonte e gerar o modelo alvo de forma automatizada e sem a necessidade de sair do programa.

Referências Bibliográficas

AMSTEL, M. et al. Performance in Model Transformations: Experiments with ATL and QVT. Proceedings of the Fourth International Conference on Model Transformation (ICMT2011), 2011.

Atlas group; LINA; INRIA. ATL: Atlas Transformation Language - ATL, User Manual version 0.7. 2006.

BÉZIVIN, J. On the unification power of models. Springer-Verlag, May 2005.

FRANKEL, D. S. Model Driven Architeture: Applying MDA to Enterprise Computing. Wiley and OMG Press, 2003.

GARDNER, T. et al. A review of OMG MOF 2.0 Query/Views/Transformations submissions and recommendations towards the final standard. 1st International Workshop on Metamodeling for MDA, 2003.

GUDURIC, P.; PUDER, A.; TODTENHOFER, R. A Comparison Between Relational and Operational QVT Mappings. Sixth International Conference on Information Technology: New Generation, 2009.

HUNTER, J. JDOM Makes XML Easy. Sun's 2002 Worldwide Java Developer Conference, 2002.

JOUAULT, F.; KURTEV, I. On the architectural alignment of ATL and QVT. In SAC'06: Proceedings of the 2006 ACM symposium on Applied Computing, p. 1188–1195, 2006.

KLEPPE, A.; WARMER, J.; BAST, W. MDA Explained: the model driven architeture: Practice and Promise. Addison-Wesley, 2003.

LAARMAN, A. Achieving QVTO & ATL Interoperability: An Experience Report on the Realization of a QVTO to ATL Computer. 1st International Workshop on Model Transformation with ATL, MtATL, p. 119–133, 2009.

LOPES, D. Capítulo 2 - Engenharia Dirigida por Modelos: Abordagem e Aplicações. ERCEMAPI, p. 13–35, 2008.

LOPES, D. et al. Generating Transformation Definition from Mapping Specification: Application to Web Service Platform. The 17th Conference on Advanced Information Systems Engineering (CAiSE'05), no LNCS 3520, p. 309–325, 2005.

LOPES, D.; HAMMOUDI, S.; ABDELOUAHAB, Z. Schema Matching in the context of Model Driven Engineering: From Theory to Practice. Proceeding of the International Conference on Systems, Computing Sciences and Software Engineering, p. 219–227, 2005.

LOPES, D. et al. Metamodel Matching: experiments and comparison. IEEE International Conference on Software Engineering Advances, 2006.

OMG. MDA guide. Acessado em: 20/06/2012, Disponível em: http://www.omg.org/cgi-bin/doc?omg/03-06-01.pdf, 2003.

OMG. Meta Object Facility (MOF) Core Specification. formal/2006-01-01, 2006.

OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification - Final Adopted Specification. formal/2011-01-01, 2011.

OMG. OMG MOF 2 XMI Mapping Specification. formal/2011-08-09, 2011.

OMG. OMG Unified Modeling LanguageTM (OMG UML), Infrastructure. formal/2011-08-05, August 2011.

OMG. OMG Object Constraint Language (OCL). formal/2012-01-01, 2012.

SCHMIDT, D. C. Model-Driven Engineering. IEEE Computer, February 2006.

SOUZA Jr, G. et al. A Step Forward in Semi-automatic Metamodel Matching: Algorithms and Tool. Springer, v. 24, p. 137–148, 2009.

STEINBERG, D.; BUDINSKY, F.; PATERNOSTRO, M. Ed Merks, EMF: Eclipse Modeling Framework. Addison-Wesley Professional. 2nd edition, 2008.

I Código-fonte do arquivo QVTGen.java

Listagem I.1: Implementação completa da classe QVTGen em Java

```
1  package language.generator;
2
3  import java.util.Iterator;
4  import java.util.LinkedList;
5  import java.util.List;
6
7  import org.eclipse.emf.common.util.EList;
8  import org.eclipse.jface.dialogs.MessageDialog;
9  import org.eclipse.core.resources.ResourcesPlugin;
10
11 import mapping.impl.CorrespondenceImpl;
12 import mapping.presentation.ITFGenLanguage;
13 import mapping.Correspondence;
14 import mapping.Definition;
15 import mapping.ElementHandler;
16 import mapping.Historic;
17 import mapping.Output;
18 import mapping.Input;
19 import mapping.TypeConverter;
20 import mapping.MetaModelHandler;
21
22 /**
23  * @author Paulo Jansen
24  */
25 public class QVTGen implements ITFGenLanguage{
26
27         private String qvtContent;
28         private Object root;
29
30         public String main;
31         public String auxMain;
32         public String auxContent = "";
33         public Boolean inIsLoaded;
34         public Boolean outIsLoaded;
35         public XMIModelReader inEcore;
36         public XMIModelReader outEcore;
37
38         //Constructor
39         public QVTGen() {
40                 qvtContent = new String();
41         }
42
43         public Object getRoot() { return root; }
44
45         public void setRoot(Object root) { this.root = root; }
46
47         public String getCode() { return qvtContent; }
48
49         public String computelanguage() {
50                 Object object = getRoot();
51                 Historic historic = null;
52                 if (!(object instanceof Historic))
53                         return null;
54                 historic = (Historic) object;
55                 genLanguage(historic);
56                 return qvtContent;
57         }
58
59         /**
```

```
60          * @param historic
61          */
62         private void genLanguage(Historic historic) {
63                 for (Iterator it = historic.getDefinition().iterator();it.hasNext();){
64                         Definition definition = (Definition) it.next();
65                         definition2Transformation(definition);
66                 }
67                 return;
68         }
69
70         /**
71          * @param definition
72          */
73         private void definition2Transformation(Definition definition) {
74                 String path = ResourcesPlugin.getWorkspace().getRoot().getRawLocation().toString();
75                 String sourceModelPath, targetModelPath;
76                 String targetName = definition.getTarget().getName();
77                 String sourceName = definition.getSource().getName();
78
79                 String transformation =
80                         "transformation " + definition.getName() +
81                         "( in " + sourceName + " : " + sourceName.toUpperCase() +
82                         ", out " + targetName + " : " + targetName.toUpperCase() + ");\n\n";
83
84                 main = "main(){\n";
85                 auxMain = "\t\t" + sourceName + ".objectsOfType(";
86
87                 sourceModelPath = path + definition.getSource().getMetaModelname();
88                 targetModelPath = path + ((MetaModelHandler)definition.getTarget()).getMetaModelname();
89
90                 inEcore = new XMIModelReader();
91                 inIsLoaded = inEcore.build(sourceModelPath);
92                 if (!inIsLoaded){
93                         MessageDialog.openError(null, "QVT Generator",
94                                         "Cannot build the source metamodel file.");
95                 }
96                 outEcore = new XMIModelReader();
97                 outIsLoaded = outEcore.build(targetModelPath);
98                 if (!outIsLoaded){
99                         MessageDialog.openError(null, "QVT Generator",
100                                        "Cannot build the target metamodel file.");
101                }
102
103                String modeltype = (inIsLoaded && outIsLoaded)?
104                                "modeltype " + sourceName.toUpperCase() + " uses " + sourceName + "('"
                                    + inEcore.getURI() + "');\n" +
105                                "modeltype " + targetName.toUpperCase() + " uses " + targetName + "('"
                                    + outEcore.getURI() + "');\n\n":
106                                "modeltype " + sourceName.toUpperCase() + " uses " + sourceName + "('')
                                    ;\n" +
107                                "modeltype " + targetName.toUpperCase() + " uses " + targetName + "('')
                                    ;\n\n";
108
109
110                for (Iterator maps = definition.getRules().iterator(); maps.hasNext();){
111                        Correspondence correspondence = (Correspondence) maps.next();
112                        correspondence2Mapping(correspondence);
113                }
114
115                qvtContent += modeltype + transformation + main + "}\n" + auxContent;
116        }
117
118        /**
119         * @param correspondence
120         */
121        private void correspondence2Mapping(Correspondence correspondence){
122                if (correspondence == null){
123                        MessageDialog.openError(null, "QVT Generator",
```

```
124                                     "Please check this mapping model!\n" + "A correspondence cannot
                                            be null!");
125                     return;
126             }
127
128             String name = correspondence.getName();
129
130             if (name == null || name.length() == 0){
131                     if (correspondence.getInput() != null && correspondence.getOutput() != null){
132                             if (correspondence.getOutput().size() > 0){
133                                     if (correspondence.getInput().getName() != null &&
134                                                     ((Output)correspondence.getOutput().get(0)).
                                                         getName() != null)
135
136                                             name = correspondence.getInput().getName() + "2" +
137                                             ((Output)correspondence.getOutput().get(0)).getName();
138                             }
139                     }
140             }
141
142             if (name == null || correspondence.getInput() == null || correspondence.getOutput().
                    size() == 0){
143                     MessageDialog.openError(null, "QVT Generator", "Please check this mapping model
                            !\n" +
144                                     "A correspondence must have one input and at least one output
                                            whit specified handlers.\n" +
145                                     "This correspondence will not be generated.");
146                     return;
147             }
148
149             Input input = correspondence.getInput();
150             String inputName = input.getName();
151             String outputName = ((Output)correspondence.getOutput().get(0)).getName();
152             if (input != null){
153                     if (input.getHandler() != null){
154                             inputName = input.getHandler().getName();
155
156                             if (input.getHandler().getType().toString().contains("EEnum"))
157                                     return;
158
159                             if (correspondence.getOutput() != null){
160                                     if (correspondence.getOutput().size() > 0){
161                                             int i=0;
162                                             for (Iterator it = correspondence.getOutput().iterator
                                                    (); it.hasNext(); i++){
163                                                     Output output = (Output)it.next();
164                                                     if (output.getHandler() == null)
165                                                             continue;
166                                                     if (output.getHandler().getOwner() == null)
167                                                             continue;
168                                                     if (i==0)
169                                                             outputName = output.getHandler().
                                                                    getName();
170
171                                                     auxContent += "\nmapping " + inputName + "::" +
                                                            name + "() : " + outputName + "{\n";
172                                                     main += auxMain + inputName + ") -> map " +
                                                            name + "();\n";
173
174                                                     //Begin nested2binding
175                                                     List list = getAllCorrespondencesWithOutput(
                                                            correspondence, output);
176                                                     if (list!= null)
177                                                             nested2Binding(0, inputName, list);
178                                                     //End nested2binding
179
180                                                     if (it.hasNext()){
181                                                             auxContent += "\t" + "";
```

```
182                                                     }
183
184                                                     auxContent += "\n}\n";
185                                                 }
186                                             }
187                                         }
188                                     }
189                             }
190                     }
191
192                     /**
193                      * @param tab
194                      * @param inputName
195                      * @param list
196                      */
197                     private void nested2Binding(int tab, String inputName, List list){
198                             if (list == null || inputName == null){
199                                     MessageDialog.openError(null, "QVT Generator", "Please check this mapping model
                                            !\n" +
200                                                     "List of nested correspondences and input name cannot be null!"
                                                            );
201                                     return;
202                             }
203
204                             for (Iterator it = list.iterator(); it.hasNext();){
205                                     Correspondence correspondence = (Correspondence)it.next();
206
207                                     if (correspondence.eContainer() instanceof CorrespondenceImpl
208                                                     && correspondence.getInput() != null && correspondence.
                                                            getOutput() != null
209                                                     && correspondence.getOutput().size() > 0){
210                                             Input input = correspondence.getInput();
211                                             Output output = (Output)correspondence.getOutput().get(0);
212
213                                             TypeConverter tc = correspondence.getTypeconverterLR();
214                                             String ocl = null;
215                                             if (tc != null && tc.getOclExpression() != null && tc.getOclExpression
                                                    ().length() != 0)
216                                                     ocl = tc.getOclExpression();
217
218                                             String conversion;
219                                             String relationship;
220                                             if (outIsLoaded){
221                                                     String tmp = outEcore.getETypeName(output.getHandler().getName
                                                            ());
222                                                     relationship = (tmp!=null)? ".resolveone(" + tmp + ")":
223                                                             ".resolveone(/*Type of " + input.getHandler().getName()
                                                                    + "*/)";
224                                             }
225                                             else
226                                                     relationship = ".resolveone(/*Type of " + input.getHandler().
                                                            getName() + "*/)";
227
228                                             if (ocl == null){
229                                                     conversion = " := self." + input.getHandler().getName();
230
231                                                     if (input.getHandler().getType().toString().contains("
                                                            EReference"))
232                                                             conversion += relationship;
233                                             }
234                                             else{
235                                                     if (ocl.indexOf("@") > -1){
236                                                             String str = ocl;
237                                                             str = str.replaceAll("if", "").replaceAll("else", "");
238                                                             str = str.replaceAll("then", "").replaceAll("end", "");
239                                                             str = str.replaceAll("@=", "").replaceAll(" ", "").
                                                                    replaceFirst("#", "");
240
```

```
241                                             String[] l = str.split("#");
242                                             if (inIsLoaded && outIsLoaded){
243                                                     for (int i=0; i<l.length; i++){
244                                                             if (i%2 == 0)
245                                                                     ocl = ocl.replaceAll("#"+l[i],
                                                                             inEcore.
                                                                             getEnumNameOfThisLiteral(l
                                                                             [i])+"::"+l[i]);
246                                                             else
247                                                                     ocl = ocl.replaceAll("#"+l[i],
                                                                             outEcore.
                                                                             getEnumNameOfThisLiteral(l
                                                                             [i])+"::"+l[i]);
248                                                     }
249                                             } else ocl = ocl.replaceAll("#", "/*Enum's name of this
                                                    literal*/::");
250
251                                             ocl = ocl.replaceAll("else", "else\n\t\t\t");
252                                             ocl = ocl.replaceFirst("endif", "\n\t\tendif");
253
254                                             conversion = " := " + ocl.replaceAll("@", "self." +
                                                      input.getHandler().getName() + " ");
255                                     }
256                                     else if (ocl.length() > 0)
257                                             conversion = " := " + ocl;
258                                     else
259                                             conversion = " := self" + "." + input.getHandler().
                                                  getName();
260                             }
261
262                             if (it.hasNext()){
263                                     auxContent += "\t\t" + output.getHandler().getName() +
                                          conversion + ";\n";
264                             }
265                             else auxContent += "\t\t" + output.getHandler().getName() + conversion+
                                  ";";
266                     }
267                     else{
268                             MessageDialog.openError(null, "QVT generator", "Please check this
                                  mapping model!\n"
269                                             + "A nested correspondence must have one input and at
                                                  least one output whith specified handlers!");
270                     }
271             }
272     }
273
274     /**
275      * @param corresp
276      * @param output
277      */
278     private List getAllCorrespondencesWithOutput(Correspondence corresp, Output output){
279                     if (corresp.getNested() == null || output == null)
280                             return null;
281                     LinkedList lklist = new LinkedList();
282
283                     ElementHandler parent = output.getHandler();
284                     if (parent == null) {
285                             MessageDialog.openError(null, "QVT generator", "Please check this
                                  mapping model!"
286                                             + "\nA parent handler cannot be null!");
287                             return null;
288                     }
289
290                     for (Iterator iter = corresp.getNested().iterator(); iter.hasNext();) {
291                             Correspondence nested = (Correspondence) iter.next();
292                             if (((EList) nested.getOutput()).size() > 0) {
293                                     Output tmp = (Output) nested.getOutput().get(0);
```

```
294                                     if (tmp.getHandler() != null && tmp.getHandler().getEnclosing()
                                            == parent) {
295                                         lklist.add(nested);
296                                     }
297                                     else if (tmp.getHandler().getEnclosing() ==null) {
298                                         MessageDialog.openError(null, "QVT generator", "Please
                                                check this mapping model!"
299                                                     + "\nA nested correspondence must have
                                                          one input and at least one output
                                                          whit specified handlers.");
300                                     }
301                                 } else {
302                                     MessageDialog.openError(null, "QVT generator", "Please check
                                            this mapping model!"
303                                                 + "\nA nested correspondence must have one
                                                      input and at least one output whit
                                                      specified handlers.");
304                                 }
305                             }
306                             return lklist;
307         }
308 }
```

II Código-fonte do arquivo XMIModelReader.java

Listagem II.1: Implementação da classe XMIModelReader em Java

```
1  package language.generator;
2
3  import java.util.Iterator;
4
5  import org.jdom2.Document;
6  import org.jdom2.Element;
7  import org.jdom2.input.SAXBuilder;
8
9  public class XMIModelReader {
10         public SAXBuilder sax;
11         public Document doc;
12
13         public XMIModelReader(){
14                 sax = new SAXBuilder();
15                 doc = null;
16         }
17
18         public Boolean build(String path){
19                 try {
20                         doc = sax.build(path);
21                         return true;
22                 }
23                 catch(Exception e) {
24                         e.printStackTrace();
25                         return false;
26                 }
27         }
28
29         public String getURI(){
30                 return doc.getRootElement().getAttributeValue("nsURI");
31         }
32
33         public String getEnumNameOfThisLiteral(String literal){
34                 Iterator<Element> iter, it = doc.getRootElement().getChildren().iterator();
35
36                 while (it.hasNext()){
37                         Element elem = it.next();
38                         iter = elem.getChildren().iterator();
39
40                         while (iter.hasNext()){
41                                 Element e = iter.next();
42                                 if (e.getAttribute("name") != null &&
43                                                 e.getAttributeValue("name").equals(literal)){
44                                         return elem.getAttributeValue("name");
45                                 }
46                         }
47                 }
48                 return null;
49         }
50
51         public String getETypeName(String name){
52                 Iterator<Element> iter, it = doc.getRootElement().getChildren().iterator();
53
54                 while (it.hasNext()){
55                         iter = it.next().getChildren().iterator();
56
57                         while (iter.hasNext()){
58                                 Element e = iter.next();
59                                 if (e.getAttribute("name") != null &&
```

```
60                                              e.getAttributeValue("name").equals(name)){
61                                          String str = e.getAttributeValue("eType").toString();
62                                          return str.substring(3);
63                                      }
64                                  }
65                              }
66                              return null;
67                      }
68              }
```

III Código-fonte da transformação Uml2Java

Listagem III.1: Definição de transformação em QVT-O gerada pelo *plug-in* construído

```
1  modeltype UML uses uml('http://uml');
2  modeltype JAVAMETA uses javameta('http://javameta');
3
4  transformation UML2JAVA( in uml : UML, out javameta : JAVAMETA);
5
6  main(){
7          uml.objectsOfType(Class) -> map Class2JClass();
8          uml.objectsOfType(Property) -> map Property2JField();
9          uml.objectsOfType(Method) -> map Method2JMethod();
10         uml.objectsOfType(Parameter) -> map Parameter2JParameter();
11         uml.objectsOfType(Interface) -> map Interface2JInterface();
12         uml.objectsOfType(PrimitiveType) -> map Primitive2JPrimitiveType();
13         uml.objectsOfType(Package) -> map Package2JPackage();
14 }
15
16 mapping Class::Class2JClass() : JClass{
17         name := self.name;
18         visibility := if self.visibility = VisibilityKind::public then Visibility::public else
19                 if self.visibility = VisibilityKind::protected then Visibility::protected else
20                 if self.visibility = VisibilityKind::private then Visibility::private else
21                 if self.visibility = VisibilityKind::package then Visibility::package
22         endif endif endif endif;
23 }
24
25 mapping Property::Property2JField() : JField{
26         name := self.name;
27         visibility := if self.visibility = VisibilityKind::public then Visibility::public else
28                 if self.visibility = VisibilityKind::protected then Visibility::protected else
29                 if self.visibility = VisibilityKind::private then Visibility::private else
30                 if self.visibility = VisibilityKind::package then Visibility::package
31         endif endif endif endif;
32         initialValue := self.initialValue;
33         jClassifier := self.owner.resolveone(JClassifier);
34         type := self.type.resolveone(JClassifier);
35 }
36
37 mapping Method::Method2JMethod() : JMethod{
38         name := self.name;
39         visibility := if self.visibility = VisibilityKind::public then Visibility::public else
40                 if self.visibility = VisibilityKind::protected then Visibility::protected else
41                 if self.visibility = VisibilityKind::private then Visibility::private else
42                 if self.visibility = VisibilityKind::package then Visibility::package
43         endif endif endif endif;
44         body := self.body;
45         jClassifier := self.owner.resolveone(JClassifier);
46         parameters := self.parameter.resolveone(JParameter);
47 }
48
49 mapping Parameter::Parameter2JParameter() : JParameter{
50         name := self.name;
51         type := self.type.resolveone(JClassifier);
52 }
53
54 mapping Interface::Interface2JInterface() : JInterface{
55         name := self.name;
56         visibility := if self.visibility = VisibilityKind::public then Visibility::public else
57                 if self.visibility = VisibilityKind::protected then Visibility::protected else
58                 if self.visibility = VisibilityKind::private then Visibility::private else
59                 if self.visibility = VisibilityKind::package then Visibility::package
```

```
60          endif endif endif endif;
61  }
62
63  mapping PrimitiveType::Primitive2JPrimitiveType() : JPrimitiveType{
64          name := self.name;
65          visibility := if self.visibility = VisibilityKind::public then Visibility::public else
66                  if self.visibility = VisibilityKind::protected then Visibility::protected else
67                  if self.visibility = VisibilityKind::private then Visibility::private else
68                  if self.visibility = VisibilityKind::package then Visibility::package
69          endif endif endif endif;
70  }
71
72  mapping Package::Package2JPackage() : JPackage{
73          name := self.name;
74          jElement := self.ownedElement.resolveone(JElement);
75  }
```