Universidade Federal de Campina Grande

Centro de Engenharia Elétrica e Informática

Coordenação de Pós-Graduação em Ciência da Computação

Dissertação de Mestrado

MetaTT - Uma Abordagem Baseada em Metamodelos para a Escrita de Transformações de Modelo para Texto

Anderson Rodrigo Santos Bezerra Ledo

Campina Grande, Paraíba, Brasil
Agosto - 2012

Universidade Federal de Campina Grande

Centro de Engenharia Elétrica e Informática

Coordenação de Pós-Graduação em Ciência da Computação

MetaTT - Uma Abordagem Baseada em Metamodelos para a Escrita de Transformações de Modelo para Texto

Anderson Rodrigo Santos Bezerra Ledo

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação

Linha de Pesquisa: Engenharia de Software

Franklin de Souza Ramalho (Orientador)

Campina Grande, Paraíba, Brasil

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

L474m Ledo, Anderson Rodrigo Santos Bezerra.
MetaTT - Uma abordagem baseada em metamodelos para a escrita de transformações de modelo para texto / Anderson Rodrigo Santos Bezerra Ledo. - Campina Grande, 2012.
67f.

Dissertação (Mestrado em Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática.
Orientador: Prof. Dr. Franklin de Souza Ramalho.
Referências.

1. Engenharia de Software. 2. Metamodelos. 3. Desenvolvimento Dirigido por Modelos (DDM). I. Título.

CDU 004.41(043)

"METATT - UMA ABORDAGEM BASEADA EM METAMODELOS PARA A ESCRITA DE TRANSFORMAÇÕES DE MODELO PARA TEXTO"

ANDERSON RODRIGO SANTOS BEZERRA LEDO

DISSERTAÇÃO APROVADA EM 24/08/2012

FRANKLIN DE SOUZA RAMALHO, Dr.
Orientador(a)

ADALBERTO CAJUEIRO DE FARIAS, Dr.
Examinador(a)

VINICIUS CARDOSO GARCIA, Dr.
Examinador(a)

CAMPINA GRANDE - PB

Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Programa de Pós-Graduação em Ciência da Computação

Parece da Dissertação de Mestrado

ALUNO: Anderson Rodrigo Santos Bezerra Ledo.

Suas Recomendações e Seu Parecer

Eu, Vinicius Cardoso Garcia, Doutor, Professor Adjunto do Departamento de Informação e Sistemas do Centro de Informática da Universidade federal de Pernambuco, dou o parecer favorável a aprovação da Dissertação de Mestrado "MetaTT - Uma Abordagem Baseada em Metamodelos para a Escrita de Transformações de Modelo para Texto" do candidato Anderson Rodrigo Santos Bezerra Ledo, com pequenas correções e melhorias no texto, indicadas por mim no documento enviado a ele.

Ressalto que a minha participação se deu por vídeo conferência, via internet (utilizando o software Skype) mas que não teve qualquer influência na minha atuação durante o processo de defesa. Muito pelo contrário, tudo funcionou perfeitamente bem.

Gostaria de destacar a organização, preocupação e a atenção de todos os envolvidos para que a minha participação via vídeo ocorresse da melhor maneira possível.

Examinador (Assinatura):
Data: 24/08/2012

Resumo

O Desenvolvimento Dirigido por Modelos (DDM) é uma área bem estabelecida nas comunidades acadêmica e industrial da Engenharia de Software. Uma série de benefícios são inerentes à adoção de DDM, como um alto nível de automação e reutilização. Contudo, para alcançar tais benefícios é necessário que seus artefatos (modelos, metamodelos e transformações) sejam construídos, mantidos e empregados de maneira adequada, o que requer tempo e esforço da equipe de desenvolvimento envolvida.

Algumas técnicas e ferramentas para apoiar atividades de metamodelagem e elaboração de transformações têm sido propostas com a finalidade de orientar as atividades de DDM, permitindo sua aplicação e aproveitando melhor o seu potencial de automação e reutilização.

Com relação à escrita de transformações de modelo-para-texto (M2T), alguns trabalhos abordam o uso delas aplicadas em domínios específicos, mas não tratam de como projetar e escrever tais transformações independentemente do domínio e prezando por boas decisões de projeto na elaboração das transformações. Outros trabalhos abordam o problema de geração de sintaxe concreta a partir de modelos mas não têm foco em transformações M2T.

Neste trabalho, propomos MetaTT, uma abordagem que compreende a proposta de uma arquitetura comum para transformadores M2T e uma técnica operacional para guiar a escrita das regras de transformação com base na estrutura dos metamodelos. Através da análise de cenários de aplicação, verificamos que MetaTT diminui significativamente o esforço empregado pelo desenvolvedor na construção das transformações e se torna especialmente útil quando necessita-se lidar com metamodelos grandes. Além disso, fornecemos um suporte ferramental para automatizar as tarefas prescritas e avaliamos o seu uso em cenários diferentes e com metamodelos de características variadas.

Abstract

Model Driven Development (MDD) is a well-established area in the academic and industrial software engineering communities. A number of benefits are inherent from the adoption of MDD, *i.e.*, a high level of automation and reuse. However, to achieve these benefits it is necessary that its artifacts (models, metamodels and transformations) are constructed, maintained and used properly, which requires time and effort from the development team in charge.

Some techniques and tools to support metamodeling activities and the development of transformations have been proposed in order to guide the activities of MDD, enabling its application and a better use of its potential for automation and reuse.

With respect to the writing of model-to-text (M2T) transformations, some works discuss the use of such kind of transformations applied in specific areas, but they do not address how to design and write them regardless of the metamodeled domain and with focus on incorporating good design decisions in the elaboration of the transformations. Other works address the problem of concrete syntax generation from models but do not focus on M2T transformations.

In this dissertation, we propose MetaTT, an approach that includes the proposal of a common architecture for M2T syntax generators and a technique to guide the writing of the transformation rules based on the structure of the metamodels. MetaTT significantly reduces the effort made by the developer in the construction of transformations and is especially useful when you need to handle large metamodels. In addition, we provide a support tool to automate the required tasks and we evaluate its use within different scenarios and using metamodels with varied characteristics.

Agradecimentos

Primeiramente, gostaria de agradecer a Deus pelo meu bem estar, pela minha saúde física e mental e por ter condições e capacidade de trabalhar e buscar por minhas metas e objetivos.

Dedico o esforço empregado neste trabalho a minha família e, em especial, a minha mãe, Guia. Agradeço enormemente a Maria José e Guedes, que são tios/pais pra mim. Sem estas pessoas eu não teria acesso ao bem mais precioso que tenho hoje: educação cidadã e erudita. Agradeço por poder contar com a compreensão, apoio e amizade de toda a minha família: avós, tios, tias, primos, irmãos.

Agradeço ao Prof. Franklin por estes anos de trabalho dos quais tiro grande aprendizado. Agradeço por seu apoio, pela sua disponibilidade, por sua paciência, por sua dedicação e compromisso. Desejo-lhe, ainda mais, grandes projetos e resultados acadêmicos em sua carreira e grande felicidade em sua vida.

Agradeço pelo auxílio e presença dos meus colegas de laboratório durante toda minha estada nele: Everton, Júnior, Paulo, Fábio, Andreza, Carlos Artur, Caio Paes, Waldemar Neto, Jemão, João Felipe, Camila, Anne, Sebastião, Hugo e Arthur Marques, André Aranha, Daniel, Gabriel, Dudu, Gustavo e todos os demais.

Agradeço também aos meus amigos por seu apoio durante este trabalho: Mari, Eloá, Lorena, Larissa, Camilla, Solon, Edigley, Jaindson, Rafael, Nustenil, Mayanna e aos amigos do CGHS que me acompanharam no último ano (André, Thiago, Danilo, Worm, Nigini, Amaury, Arthur, Vítor etc.). Um agradecimento especial ao meu amigo Natã, que auxiliou este trabalho ajudando na escrita de artigos, contribuindo com sugestões e implementando boa parte do suporte ferramental do MetaTT.

Foi muito bom contar com o apoio e profissionalismo dos funcionários da COPIN. Obrigado às funcionárias Aninha, Vera e Rebeka e aos coordenadores Hyggo e Nazareno.

Por último, mas não menos importante, agradeço às instituições que apoiaram este trabalho: o apoio financeiro da CAPES em forma de bolsa; o apoio administrativo e financeiro da COPIN, com processos e custeio de viagens; e o apoio acadêmico e de infraestrutura do SPLab (antes GMF) durante todo este tempo.

Conteúdo

Lista de Símbolos

ATL - *Atlas Transformation Language*

BNF - *Backus-Naur Form*

CA - Complexidade Agregada

DDM - Desenvolvimento Dirigido por Modelos

DSL - *Domain Specific Language*

EMF - *Eclipse Modeling Framework*

EMOF - *Essentials MOF*

HTML - *HyperText Markup Language*

IDE - *Integrated Development Environment*

JET - *Java Emitter Templates*

MDA - *Model Driven Architecture*

MNRM - Média do Número de Referências do Metamodelo

MOF - *Meta Object Facility*

MPAH - Média da Profundidade das Árvores de Herança

M2M - *Model To Model*

M2T - *Model To Text*

NM - Número de Metaclasses

OCL - *Object Constraint Language*

OMG - *Object Management Group*

QVT - *Query Views Transformation Language*

UML - *Unified Modeling Language*

UP - *Unified Process*

XMI - *XML Metadata Interchange*

XML - *Extensible Markup Language*

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

A Engenharia de Software é uma disciplina que define métodos e processos a fim de introduzir melhorias na produção de software através da proposta de arquiteturas, arcabouços, ferramentas, técnicas, abordagens, etc. Dentro da Engenharia de Software, temos várias sub-áreas dedicadas ao estudo de tópicos específicos (processos de software, testes de software, linguagens de programação, etc.) e cada área possui tanto um corpo de conhecimento quanto um conjunto de ferramentas em particular. Uma das sub-áreas da Engenharia de Software que tiveram ascenção na última década foi o Desenvolvimento Dirigido por Modelos (DDM) [42]. O objetivo de DDM é atacar problemas inerentes às abordagens tradicionais, mais especificamente aqueles que estão ligados à falta de sincronização entre modelos de software e sua implementação dentro de um processo de software.

Ao se executar um processo iterativo comum, a exemplo do UP [22], o software passa primeiramente por fases de análise de requisitos, análise de viabilidade e, logo após, por fases de projeto arquitetural e projeto de software. Estes projetos são documentados através de modelos de software que desempenham papel fundamental na comunicação das decisões técnicas e como guia para a implementação do software ao longo do processo de desenvolvimento. Estes modelos podem ser de natureza estrutural (como diagramas de componentes, diagramas de classes, etc.) ou de natureza comportamental (como diagramas de sequência, diagramas de atividades, etc.). O ponto falho destes processos, do ponto de vista da abordagem DDM, é que os modelos não estão no centro das atividades e isto faz com que, ao longo do ciclo de vida do software, as mudanças e adições de funcionalidades passem a ser realizadas mais no código do que nos modelos, o que leva a uma inconsistência que diminui

cada vez mais a utilidade dos modelos neste contexto. Desta forma, o software deixa de ter uma representação mais abstrata que seja útil para a equipe e aspectos como comunicação e boa documentação são enfraquecidos.

Contrastando com este cenário, o Desenvolvimento Dirigido por Modelos (DDM), é uma metodologia de desenvolvimento de software na qual o foco está nos modelos de software ao invés do código de implementação. DDM é uma área que se estabeleceu nas comunidades acadêmica e industrial de engenharia de software abordando tópicos relativos à metamodelagem de variados domínios, transformações entre estes domínios, geração de código para várias plataformas, etc. Nesta linha, um conjunto de diferentes padrões, ferramentas e abordagens tem emergido com o intuito de concretizar a proposta do DDM. Dentre as alternativas disponíveis (*e.g.*, [46], [4], [17] e [35]), escolhemos utilizar MDA devido ao seu nível de padronização e detalhe.

MDA é um arcabouço que compreende um conjunto de padrões para a definição de seus principais artefatos, que são os metamodelos, os modelos, as transformações entre modelos (M2M) e as transformações de modelo para texto (M2T). Metamodelos são representações de um domínio de aplicação, que pode ser uma linguagem de programação, a descrição de um processo de software, um formato específico de documento, etc. Modelos são instâncias de um metamodelo. Por exemplo, um programa pode ser representado em forma de um modelo que seja uma instância de uma linguagem de programação descrita por um metamodelo. Transformações M2M são transformações que recebem um modelo como entrada e produzem um outro modelo como saída, um exemplo é uma transformação que recebe modelos UML como entrada e produz modelos Java como saída (uma representação em XMI [8] de programas Java). Transformações M2T são transformações que recebem um modelo como entrada e produzem uma saída textual, um exemplo é uma transformação que recebe modelos de programas Java (em XMI) como entrada e produz o código fonte correspondente a estes modelos.

A adoção do arcabouço MDA para a realização de DDM traz consigo um conjunto de vantagens. O nível de granularidade no arcabouço torna cada parte parcialmente independente das outras, o que permite que diferentes equipes trabalhem em diferentes partes do projeto simultaneamente e que integrem as partes facilmente porque tais partes são construídas sobre os padrões prescritos pelo Object Management Group (OMG) [45]. Por outro

lado, essa mesma granularidade do arcabouço frequentemente resulta em um alto número de artefatos que precisam ser produzidos e mantidos, o que demanda um esforço considerável das esquipes envolvidas, tempo de desenvolvimento e habilidades específicas. Por exemplo, para construir-se um software para gerenciamento financeiro seria necessário conceber um metamodelo que representasse os conceitos do domínio relativo ao gerenciamento financeiro, conceber um outro metamodelo que representasse os conceitos relacionados à plataforma na qual o software será depositado (um ambiente de computação em nuvem, um dispositivo móvel, um ambiente desktop, etc.) e/ou até mesmo um outro metamodelo para representar os conceitos da linguagem de programação em que o software deve ser codificado ao final do processo. Adicionalmente, também seria necessário elaborar as transformações para mapear os conceitos do domínio do problema para os vários ambientes específicos de plataforma e, finalmente, para a linguagem de programação na qual o código final seria gerado. No contexto apresentado, a nossa abordagem insere-se no últim passo, em que produz-se as transformações para a geração do código final.

Uma das observações que podem ser feitas sobre o cenário apresentado é que, se por um lado MDA especifica um conjunto de artefatos a fim de prezar pela consistência dos modelos e flexibilidade no processo de automação, por outro, também há uma adição de complexidade no processo de desenvolvimento. Com isto, faz-se necessário tornar o precesso DDM/MDA cada vez menos custoso a fim de facilitar sua adoção sem que para isto seja necessário abrir mão das vantagens de ter os artefatos prescritos na abordagem. Para atingir tal meta, uma das abordagens a serem seguidas é a elaboração de boas práticas para a produção e uso adequado dos artefatos, como padrões e diretrizes de projeto e construção.

Para a elaboração de transformações M2M, aquelas que transformam modelos de um domínio de aplicação para outros modelos em um outro domínio de aplicação, há algumas abordagens propostas, como em [29], [49], [48], [3] e [2]. Entretanto, há poucas técnicas que dão suporte à elaboração de transformações M2T, aquelas que transformam modelos de um domínio de aplicação para os seus correspondentes artefatos textuais, a fim de que se possa ter artefatos de documentação, código fonte, etc.

Hoje, além da abordagem padrão, o MOF Models to Text Transformation Language (MOF2Text) [18], lançado pelo OMG em 2008, há uma variedade de linguagens e ferramentas empregadas para atender à necessidade de geração de texto a partir de modelos.

Ferramentas de geração de texto baseadas em *templates*, como o Java Emitter Templates (JET) [14] e o Xpand [18], mecanismos presentes em linguagens de transformação entre modelos usados para a extração de valores primitivos (textuais neste caso) dos mesmos, como as *queries* de ATL [23] [1] e linguagens de transformação de modelo para texto baseadas em metamodelos, como parte do próprio processo de padronização, a exemplo de MOFScript [37] e Acceleo [15].

Apesar de padrões e tecnologias correspondentes terem sido incorporados à infraestrutura do arcabouço MDA, ainda há muito pouco conhecimento disponível sobre como projetar e escrever transformações de modelo para texto. Claramente, este campo demanda mais contribuições não apenas com novas linguagens e ferramentas, mas principalmente com boas práticas relativas ao uso das linguagens e ferramentas já existentes a fim de gerenciar a complexidade e evolução das transformações textuais. Ao elaborar-se transformações para um determinado metamodelo, há, frenquentemente, dúzias de metaclasses e seus relacionamentos que causam impacto nas características estruturais e comportamentais das transformações. A comunidade de MDA ainda possui um déficit de diretrizes efetivas no sentido de orientar como produzir e empregar transformações textuais.

Posto isso, verificamos que há poucos trabalhos na literatura abordando a geração de sintaxe concreta. Alguns deles têm foco em novos formalismos, tais como metamodelos ou linguagens específicas de domínio (DSLs - *Domain Specific Languages*), a fim de prover modelos de descrição para sintaxe concreta. Outros trabalhos apresentam abordagens nas quais as transformações textuais não são projetadas adequadamente. Por exemplo, elas misturam o código que diz respeito à lógica de transformação com o código que diz respeito à definição dos *templates* de sintaxe. Esta abordagem as torna muito difíceis de serem entendidas e mantidas. As formas sobre como deve-se lidar com estes problemas ainda são tópicos em aberto no campo das transformações de modelo para texto.

1.1 O Problema

O problema abordado neste trabalho é que os desenvolvedores não dispõem de um método claro para dar assistência na elaboração de transformações M2T no arcabouço DDM/MDA,

[1] Mecanismos para extração de valores primitivos.

de forma que este método possibilite a elaboração de transformações de forma que o código destas se torne mais simples e de legibilidade mais fácil do que na ausência do método, e este possa ser utilizado independentemente do domínio modelado. Desta maneira, os geradores são frequentemente produzidos de maneira *ad-hoc*, sem um padrão ou arquitetura de referência definidos, o que torna a tarefa de desenvolvimento ainda mais custosa conforme a complexidade dos metamodelos, e resulta, assim, em geradores de código de difícil evolução e manutenção.

1.2 Escopo

O trabalho está inserido no escopo do Desensolvimento Dirigido por Modelos (DDM), realizado através da Arquitetura Dirigida por Modelos (MDA). A modalidade de transformações analisada tanto no problema como na solução proposta são as transformações M2T escritas com linguagens do paradigma M2T, mais especificamente MOFScript. As implementações são baseadas nas ferramentas providas pelo EMF (*Eclipse Modeling Framework* [43]) ou baseadas nele, como metamodelos descritos em linguagem Ecore e transformações M2T escritas na linguagem MOFScript.

1.3 Solução Proposta

De acordo com o escopo e problema identificados, a solução proposta é o MetaTT, uma abordagem que compreende:

- Uma arquitetura comum para transformadores M2T, na qual especificamos diferentes módulos (e submódulos) e as preocupações que estes devem tratar bem como a estrutura das transformações e a maneira como as regras devem ser codificadas em cada módulo;
- Uma técnica operacional para guiar a escrita das regras de transformação conforme especificadas na arquitetura e a geração automática de parte das regras de transformação com base nas características estruturais dos metamodelos.

1.4 Contribuições Esperadas

Como contribuições, esperamos:

- Prover uma arquitetura de referência para a elaboração de transformações M2T, em que essa arquitetura incorpore boas decisões de projeto seguindo princípios de modularidade e separação de preocupações;
- Prover uma técnica de automação e geração de parte do código das transformações M2T de acordo com a arquitetura definida para as mesmas a fim de agilizar a implementação das mesmas;
- Contribuir com um suporte ferramental que automatize a técnica de geração das transformações e possa ajudar efetivamente na escrita delas.

1.5 Organização do Documento

O texto desta dissertação está organizado da seguinte forma:

- Capítulo 2. Fundamentação Teórica. Nesse capítulo, detalharemos conceitos que julgamos essenciais para a compreensão deste trabalho: Desenvolvimento Dirigido por Modelos (DDM); MOFScript, com informações sobre o histórico da linguagem dentro do contexto de MDD e explicações das suas características; Ecore, linguagem de metamodelagem exemplificada com um metamodelo que nos será útil ao longo da dissertação;
- Capítulo 3. MetaTT. Neste capítulo, especificamos, na Sec. 3.1, a arquitetura que propomos para a implementação dos geradores de código e, na Sec. 3.2, explicamos como gerar os artefatos arquiteturais através do uso de um modelo de referência, que, por sua vez, é obtido do metamodelo em questão. Na Sec. 3.3, explicamos como o suporte ferramental produzido pode ser utilizado para gerar os artefatos arquiteturais;
- Capítulo 4. Avaliação. Neste capítulo, na Sec. 4.1, especificamos as métricas que utilizamos como parâmetros de comparação, caracterizamos os cenários de avaliação e análisamos os resultados decorrentes da observação destes cenários; na Sec. 4.2,

avaliamos a aplicabilidade do MetaTT com relação à independência de domínios utilizando metamodelos variados e; na Sec. 4.3, analisamos as limitações e as ameaças à validade de nossas conclusões advindas do processo de avaliação;

- Capítulo 5. Trabalhos Relacionados. Neste capítulo, listamos os trabalhos relacionados que abordam o tópico da geração de sintaxe concreta, destacando as principais semelhanças e diferenças com relação ao nosso trabalho;

- Capítulo 6. Considerações Finais. Neste capítulo, elaboramos algumas conclusões sobre a abordagem proposta nesta dissertação e sobre os resultados alcançados com ela. Por fim, discutiremos as perspectivas de trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Neste capítulo, apresentamos uma visão geral sobre os conceitos importantes e tecnologias utilizadas neste trabalho.

2.1 Desenvolvimento Dirigido por Modelos (DDM)

O Desenvolvimento Dirigido por Modelos (DDM) [42] é uma abordagem da Engenharia de Software que tem como proposta fundamental uma mudança de ênfase do esforço e tempo aplicados ao longo do processo de desenvolvimento de software. Em DDM, foca-se mais em atividades de modelagem, metamodelagem e transformações de modelos, e menos em atividades de codificação como meio de obtenção do software, como é feito nas metodologias tradicionais.

Os principais elementos de uma abordagem DDM são: *(i)* Metamodelos, que descrevem como os modelos podem ser formados; *(ii)* Modelos, que são as instâncias dos metamodelos; e *(iii)* Transformações, que são regras que definem como modelos dados como entrada, e em conformidade com um dado metamodelo, devem ser transformados em modelos de saída, e em conformidade com um segundo metamodelo (geralmente diferente do primeiro). As transformações também podem definir como transformar um modelo de entrada diretamente em texto, como, por exemplo, em código fonte.

Uma visão da abordagem DDM, adaptada de [27] para incluir o conceito de transformações M2T, é mostrada na Fig. 2.1. Nela, a cadeia de transformações mostrada vai de um modelo de entrada com alto nível de abstração (*modelo 1*) até a *sintaxe concreta* resultante

da execução de todas as transformações. O primeiro modelo (*modelo 1*) está em conformidade com seu metamodelo correspondente (*metamodelo 1*) e é dado como entrada para uma *ferramenta de transformação* e transformado em um segundo modelo (*modelo 2*), que, por sua vez, está em conformidade com um segundo metamodelo (*metamodelo 2*). Uma ferramenta de transformação executa uma definição de transformação de modelo para modelo (M2M) que transforma os conceitos de um domínio, representado pelo primeiro metamodelo (*metamodelo 1*), em conceitos de um outro domínio, representado pelo segundo metamodelo (*metamodel 2*). Mais à frente, o modelo de saída dessa transformação serve como entrada para uma transformação de modelo para texto (M2T), que mapeia os conceitos desse modelo em suas representações textuais, ou seja, a sua sintaxe concreta.

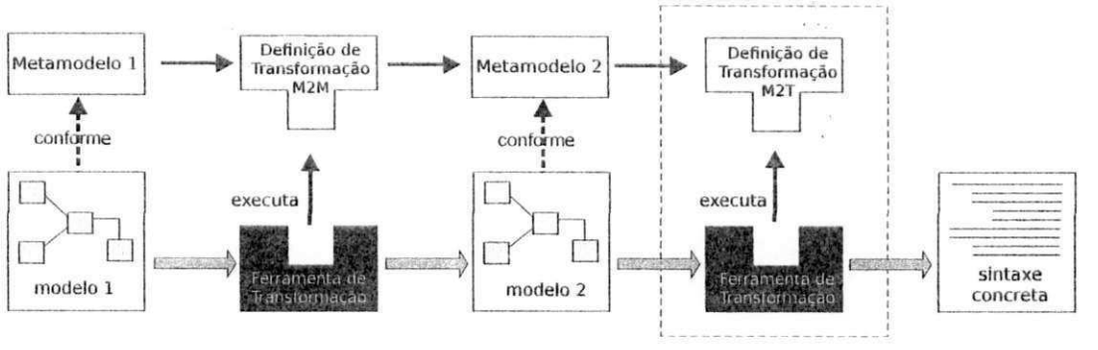

Figura 2.1: Uma visão simplificada da abordagem DDM.

Apesar de algumas similaridades entre os dois tipos de transformações, as transformações entre modelos (M2M) são essencialmente diferentes das transformações de modelo para texto (M2T). As primeiras usam pelo menos dois metamodelos (um que representa os conceitos dos modelos de entrada e outro que representa os conceitos dos modelos de saída) e elas são frequentemente escritas com linguagens híbridas, tais quais ATL [23] e QVT [39]. Por outro lado, as transformações de modelo para texto (M2T) usam apenas um metamodelo para representar os conceitos de um modelo de entrada e são frequentemente escritas em linguagens imperativas, tais quais MOFScript e MOF2Text.

A título de ilustração das diferenças entre as transformações M2M e M2T, ilustramos a seguir um exemplo de cada tipo de regra desempenhando papéis semelhantes. Na Fig. 2.2, exemplificamos uma regra M2M, presente em [7]. Esta regra de nome `C2C` utiliza casamento de padrões para mapear classes UML para classes Java de maneira declarativa estabelecendo

o padrão de origem `from` com um elemento `e` a ser casado e o padrão destino `to` com um elemento `out` a ser produzido. No elemento de destino, a regra mapeia cada valor de atributo da metaclasse *Class* do metamodelo de UML para a metaclasse *JavaClass* do metamodelo de Java. O resultado desta transformação é um novo modelo representado em XMI [8].

```
1 rule C2C {
2         from e : UML!Class
3         to out : JAVA!JavaClass (
4                 name <- e.name,
5                 isAbstract <- e.isAbstract,
6                 isPublic <- e.isPublic(),
7                 package <- e.namespace
8         )
9 }
```

Figura 2.2: Um exemplo de transformação M2M escrita em ATL.

Na Fig. 2.3, exemplificamos uma regra M2T que espelha, até certo ponto, o comportamento da regra M2M. A diferença entre as regras é que, enquanto a da Fig. 2.2 mapeia um modelo UML em um outro modelo, só que Java, a da Fig. 2.3 mapeia um modelo UML diretamente para a sintaxe concreta de Java. Vê que, enquanto a transformação M2M mapeia conceitos, a transformação M2T mapeia conceitos para estruturas sintáticas. Também fica evidente o caráter declarativo das regras M2M, mais simples neste contexto, comparado com o caráter imperativo das regras M2T, precisando checar várias condições explicitamente antes de imprimir a sintaxe concreta na saída padrão.

```
1 uml.Class::C2C() {
2         println("package " + e.namespace)
3         if(self.public = true){
4                 print("public ")
5         }
6         if(self.abstract = true){
7                 print("abstract ")
8         }
9         println("class " + self.name + "{ }")
10 }
```

Figura 2.3: Um exemplo de transformação M2T escrita em MOFScript.

2.2 MOFScript

Na Sec. 2.2.1, um breve histórico do processo de padronização e comparação com tecnologias correlatas é apresentado, já na Sec. 2.2.2 há uma descrição da sintaxe da linguagem.

2.2.1 Histórico e Características

MOFScript é uma linguagem para a especificação de transformações de modelo para texto (M2T) [37]. Ela tem vários mecanismos que permitem a navegação e consulta sobre os modelos de maneira que dados possam ser extraídos e traduzidos em texto. Com ela, qualquer tipo de texto pode ser gerado, a exemplo de código fonte, documentação ou linguagens de marcação. Ela foi uma das linguagens que fez parte do processo de padronização que resultou na especificação MOF Models To Text (MOF2Text ou MOFM2T) [18] da OMG. Apesar de suas particularidades, MOF2Text e MOFScript estão estreitamente ligadas por compartilherem o mesmo paradigma.

No período em que o MetaTT começou a ser concebido ainda não havia uma implementação de MOF2Text disponível, a exemplo de Acceleo [15], por isso a nossa abordagem foi construída com base em MOFScript, que segue o mesmo paradigma. Atualmente, Acceleo já está disponível, mas MOFScript ainda é considerada uma ferramenta mais robusta para este trabalho. Além do mais, ela tem sido amplamente utilizada na comunidade de DDM para a escrita de transformações de modelo-para-texto e ela tem uma semântica próxima à semântica da linguagem MOF2Text. Também é importante enfatizar que, de acordo com o plano do projeto Acceleo [1], algumas funcionalidades avançadas ainda não foram implementadas, como especificado em [18]. Isto significa que funcionalidades importantes, como extensão modular, sobrescrição de *templates*, *text mode switching* e macros, ainda não estão totalmente implementadas, o que ainda nos impede de realizar os conceitos discutidos neste trabalho na linguagem MOF2Text.

Com respeito às características de linguagem, MOFScript oferece mecanismos para abstração e reúso de código, tais como herança e construções semelhantes às de OCL que permitem percorrer os objetos de um modelo. Por outro lado, ao contrário de XPand [13] e Epsilon [28], o ambiente de MOFScript não dá suporte diretamente à integração de suas transformações em *workflows* de transformações, o que só pode ser alcançado através de

uma API.

Pelas razões supracitadas, MOFScript foi a linguagem adotada neste trabalho, principalmente devido ao seu alinhamento com o padrão MOF2Text, o que é essencial para facilitar uma futura migração de solução para o Acceleo.

2.2.2 A Linguagem

Em MOFScript, uma transformação é especificada com a declaração de uma *text-transformation*, que é formada por declarações de importação, um nome, a definição do metamodelo de entrada, algumas propriedades, um conjunto de regras e um conjunto de variáveis. Na Fig. 2.4, uma transformação de modelo-para-texto, chamada `JavaTexttransformation` e localizada em um pacote chamado `utilPkg`, é importada. Isto significa que todas as suas regras podem ser usadas dentro da transformação `JavaTT`. Na linha 3, há a declaração de uma transformação nomeada como `JavaTT`, da qual o metamodelo de entrada é `"Java"`, sendo referenciado dentro da transformação através do identificador `J`. As linhas 4-5 apresentam a definição de uma propriedade e uma variável, respectivamente. Uma propriedade é uma referência para valores constantes, enquanto que as variáveis podem referenciar diferentes valores ao longo das transformações. Ambas podem ser declaradas em um escopo global (a exemplo de `version` na linha 5) ou local (a exemplo de `typeDeclarations` na linha 7, declarada dentro da regra *getCompilationUnitCode()*).

As regras em MOFScript, quando têm um tipo de retorno especificado, são semelhantes às funções e são similares aos procedimentos quando não há tipo de retorno. As regras de MOFScript também podem ter um tipo de contexto, que é o tipo dos elementos do modelo ao qual a regra será aplicada. Um tipo de retorno pode ser um tipo embutido de MOFScript (*p.e.*, *String*, *Real*, etc.) ou um tipo do modelo. Parâmetros também pode ser declarados. Na Fig. 2.4, uma regra é illustrada nas linhas 6-9. Esta regra retorna um valor *String* e deve ser aplicada em instâncias do tipo *CompilationUnit*, que é seu tipo de contexto. Adicionalmente, sempre que uma regra MOFScript precisa retornar um valor, esta usa uma variável implícita e reservada da linguagem chamada de `result`, para a qual o resultado da computação deve ser atribuído, como ilustrado na linha 8.

Duas outras regras são mostradas na Fig. 2.4. A regra nomeada `toFile()` nas linhas

```
1  import ''utilPkg/JavaTexttransformation.m2t''
2
3  texttransformation JavaTT(in J:''Java''){
4   property dir : String = ''/genCode''
5   var version : Real = 1.0
6   J.CompilationUnit::getCompilationUnitCode():String{
7    var typeDeclarations: String = self.getTDCode()
8    result = typeDeclarations
9   }
10  module::toFile(d: String, v: Real, code : String){
11   file(d+''JavaFile''+v+''.java'')
12   println(code)
13  }
14  J.CompilationUnit::main(){
15   var code: String = self.getCompilationUnitCode()
16   toFile(dir, version, code)
17  }
18 ...
19 }
```

Figura 2.4: Um exemplo de transformação escrita em MOFScript.

10-13 são responsáveis pela persistência de *strings* para um arquivo. Nota-se que o tipo de contexto da regra não é um tipo específico. O seu contexto é a transformação na qual a regra está definida. Isto é especificado pela palavra chave `module` antes do nome da regra e significa que a regra pode ser invocada de qualquer lugar na tranformação sem a necessidade de um tipo de contexto. Nesta regra, há a declaração dos parâmetros `d` (o diretório no qual o arquivo deve ser salvo), `v` (o valor da versão do arquivo que vai ser salvo) e `code` (o código a ser armazenado em disco), em que `v` referencia valores do tipo *Real*, enquanto que as outras referenciam valores do tipo *String*. A regra denominada `main` nas linhas 14-17 é uma regra especial uma vez que ela é sempre a primeira a ser executada em uma transformação de modelo para texto e, desta maneira, é chamada de regra **ponto de entrada**. Toda transformação MOFScript só pode ser executada diretamente se possuir uma regra que seja um ponto de entrada, ou seja, uma regra *main*. Esta regra também pode ter um objeto de contexto ou não. Quando não há um contexto específico, a palavra *module* é utilizada. Alternativamente, o contexto de execução de regra pode ser um elemento do metamodelo, e o nome de tal elemento deve ser usado na declaração. Nesse caso, a regra é executada para

cada elemento do modelo de entrada que seja uma instância do mesmo tipo do contexto. Por exemplo, na Fig. 2.4 a regra *main* é executada para cada elemento *CompilationUnit* presente no modelo de entrada.

2.3 Ecore

A OMG definiu o MOF [38] como a metalinguagem padrão para especificar metamodelos. Com o mesmo objetivo, a comunidade Eclipse lançou o Ecore [44], subconjunto de MOF implementado dentro do *Eclipse Modeling Framework* (EMF) [44]. Ao longo do tempo, o projeto EMF também foi um contribuinte significativo para o *Essentials MOF* (EMOF) que é parte do padrão MOF2 [40]. Este processo permitiu um alinhamento entre os conceitos de Ecore e EMOF. Uma vez que todo modelo MOF2 pode ser representado como um modelo EMOF, é rezoável de se dizer que Ecore e MOF2 também estão alinhados.

Este trabalho aborda a escrita de geradores de código M2T para metamodelos descritos na metalinguagem Ecore, uma vez que a implementação de MOFScript é baseada no EMF e metamodelos especificados em Ecore são requeridos.

Com o intuito de ilustrar um cenário de aplicação para nosso trabalho, nós definimos PLang, um metamodelo simplificado que espelha conceitos simples de linguagens de programação do paradigma procedimental. Na Fig. 2.5, mostramos o metamodelo de PLang, representando apenas as associações de composição mais importantes a fim de manter o modelo simples. Abaixo, segue uma breve descrição dos elementos do metamodelo PLang:

- Modelos PLang (*AST*) são compostos de um conjunto de nodos (*ASTNode*). Como em várias linguagens de programação procedurais, PLang dá suporte a comandos (*Statement*), expressões (*Expression*), declarações (*Declaration*), tipos (*Type*) e declarações de tipos (*TypedDeclaration*);

- Um comando pode ser também classificado como um comando iterativo (*Iterative*), uma atribuição (*Assignment*), uma chamada de um procedimento (*ProcedureCall*) ou um comando condicional (*ConditionalSttment*);

[1] Um pacote pode ser implementado em MOFScript como um diretório no sistema de arquivos local.

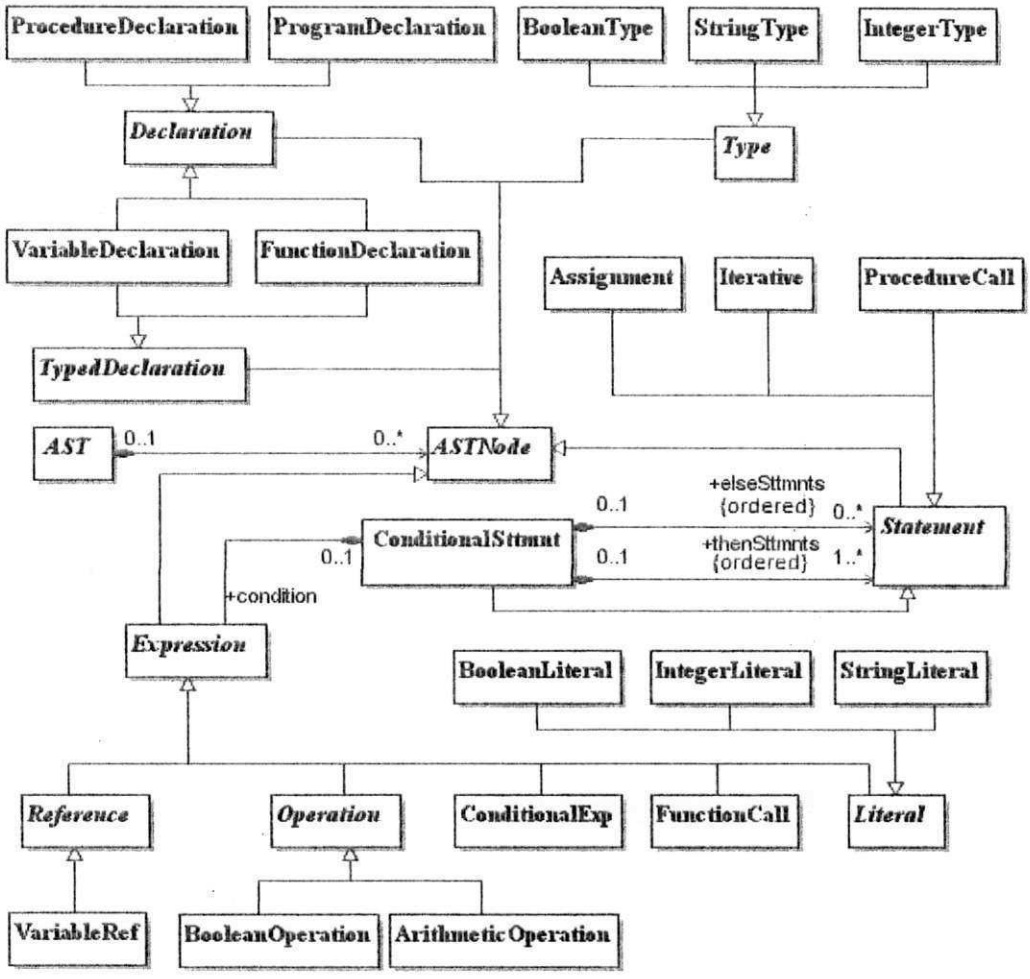

Figura 2.5: Metamodelo de PLang.

- Em PLang, pode-se declarar funções (*FunctionDeclaration*), programas (*ProgramDeclaration*), variáveis (*VariableDeclaration*) e procedimentos (*ProcedureDeclaration*);

- Uma declaração tipada (*TypedDeclaration*) pode ser classificada como (1) uma declaração de variável (*VariableDeclaration*) referenciando um tipo correspondente ao valores das variáveis; ou como (2) uma declaração de função (*FunctionDeclaration*) referenciando o tipo dos valores retornados como resultado;

- Uma expressão em PLang pode ser classificada como uma operação (*Operation*), um literal (*Literal*), uma chamada de função (*FunctionCall*), uma referência (*Reference*) ou uma expressão condicional (*ConditionalExp*);

- Neste exemplo, PLang dá suporte apenas a referências a variáveis (*VariableRef*);
- Dois tipos de operações são suportadas: operações booleanas (*BooleanOperation*) e aritméticas (*ArithmeticOperation*);
- Três tipos de literais podem aparecer em expressões de PLang: *StringLiteral*, *BooleanLiteral* e *IntegerLiteral* que modelam valores strings, booleanos e inteiros, respectivamente;
- Em PLang, um comando condicional (*ConditionalSttmnt*) é composto de (i) uma condição (*condition*) que é uma expressão (*Expression*); (ii) um conjunto ordenado de comandos a serem executados quando a condição é avaliada como verdadeira, o *thenSttmnts*; e, opcionalmente, (iii) um conjunto ordenado de comandos a serem executados quando a condição é avaliada como falsa, o *elseSttmnts*. Similarmente, uma chamada de função (*FunctionCall*) pode ter qualquer número de argumentos e uma declaração de programa (*ProgramDeclaration*) é composta de outras declarações. Contudo, com o intuito de manter a simplicidade do exemplo apenas algumas associações de composição envolvendo a metaclasse *ConditionalSttment* são ilustradas.

Por simplicidade, algumas associações e composições não são mostradas na Fig. 2.5. Por exemplo, uma declaração tipada (*TypedDeclaration*) tem uma associação com um tipo (*Type*) e uma atribuição (*Assignment*) é composta de um referência a uma variável (*VariableRef*) e uma expressão (*Expression*). Mostrar todas associações e composições adicionaria detalhes à Fig. 2.5 que não são essenciais para o entendimento do metamodelo, bem como o entendimento do trabalho como um todo.

Neste capítulo, explicamos a abordagem do Desenvolvimento Dirigido por Modelos e apresentamos os seus principais elementos, explicando o contexto em que está inserida. Mostramos os conceitos da linguagem de transformação MOFScript e apresentamos o metamodelo de PLang como metamodelo central para ilustração dos nossos exemplos nesta dissertação. As explicações e exemplos, nos próximos capítulos, abordarão transformações M2T escritas na linguagem MOFScript e transformando elementos do metamodelo PLang, apresentado aqui.

Capítulo 3

MetaTT

Neste trabalho propomos MetaTT, um abordagem com base em metamodelos para a escrita de transformações de modelo para texto (M2T). Ela guia a organização, especificação e fluxo de controle entre as transformações M2T a partir das informações providas pelo metamodelo. MetaTT consiste em:

1. Uma arquitetura a ser seguida pelas transformações M2T. Esta arquitetura é formada por módulos, submódulos e contratos entre artefatos que, juntos, objetivam alcançar um alto nível de reúso e automação no proceso de obtenção dos mesmos, separando claramente as preocupações identificadas no projeto de transformações M2T;

2. Uma abordagem operacional que indica como os artefatos propostos na arquitetura devem ser implementados, ou seja, que indica quais são as transformações e regras que devem aparecer em cada módulo precrito em (1). Para isto, utilizamos um modelo de referência de implementação e definimos padrões para a implementação das regras de acordo com o módulo e submódulo dos quais fazem parte.

O MetaTT conta com um suporte ferramental para apoiar a abordagem operacional em (2), através do qual parte das atividades são automatizadas, mas algumas ainda necessitam da interferência do desenvolvedor. Na Fig. 3.1 ilustramos como um desenvolvedor pode usar o MetaTT e como as transformações resultantes podem ser usadas para a geração da sintaxe concreta a partir dos modelos. Na *região 1*, ilustramos como os desenvolvedores precisam interagir com o suporte ferramental de MetaTT (explicado no Cap. 3.3) a fim de produzir transformações M2T:

1. O desenvolvedor provê um metamodelo para a ferramenta MetaTT;

2. MetaTT usa este metamodelo para gerar o conjunto de transformações que seguem a arquitetura padronizada conforme prescrito na Sec. 3.1. Neste processo, o módulo *Core* é gerado quase que completamente, e os demais módulos precisam de uma maior interferência do desenvolvedor;

3. O desenvolvedor complementa a implementação das regras que foram geradas como *stubs* pelo MetaTT no módulo *Templates*;

4. O desenvolvedor precisa inspecionar as transformações do módulo *Main* e ajustá-las a fim de que reflitam as decisões de projeto do próprio desenvolvedor no gerador M2T, tal como a escolha do elemento raiz a ser transformado.

Na *região* 2 da Fig. 3.1, ilustramos como as transformações M2T geradas são usadas para mapear modelos em sintaxe concreta. Este processo funciona da seguinte maneira: uma instância do metamodelo provido é dada como entrada para o gerador de sintaxe concreta, que foi obtido a partir do MetaTT. O gerador executa as transformações (parte delas geradas pelo MetaTT) sobre o modelo de entrada e, como resultado de sua execução, tem-se os arquivos gerados, tais como arquivos de documentação, código fonte Java, arquivos XML, etc.

Nosso trabalho é capaz de descrever como construir a infraestrutura de um gerador M2T independentemente do metamodelo para o qual a sintaxe concreta precisa ser gerada. Os detalhes sobre a arquitetura e a geração dos artefatos arquiteturais são discutidos nas próximas subseções.

3.1 Arquitetura

O MetaTT organiza a gerador M2T em três módulos principais: *Main*, *Core* e *Templates*, mostrados na Fig. 3.2. O módulo *Main* é responsável pelo começo do processo de transformação e por obter a sintaxe concreta resultante das tarefas realizadas no módulo *Core* bem como persistir este resultado. O módulo *Core* extrai informação dos modelos dados como entrada e usa a sintaxe definida no módulo *Templates*. O módulo *Templates* é responsável por manter as definições de sintaxe.

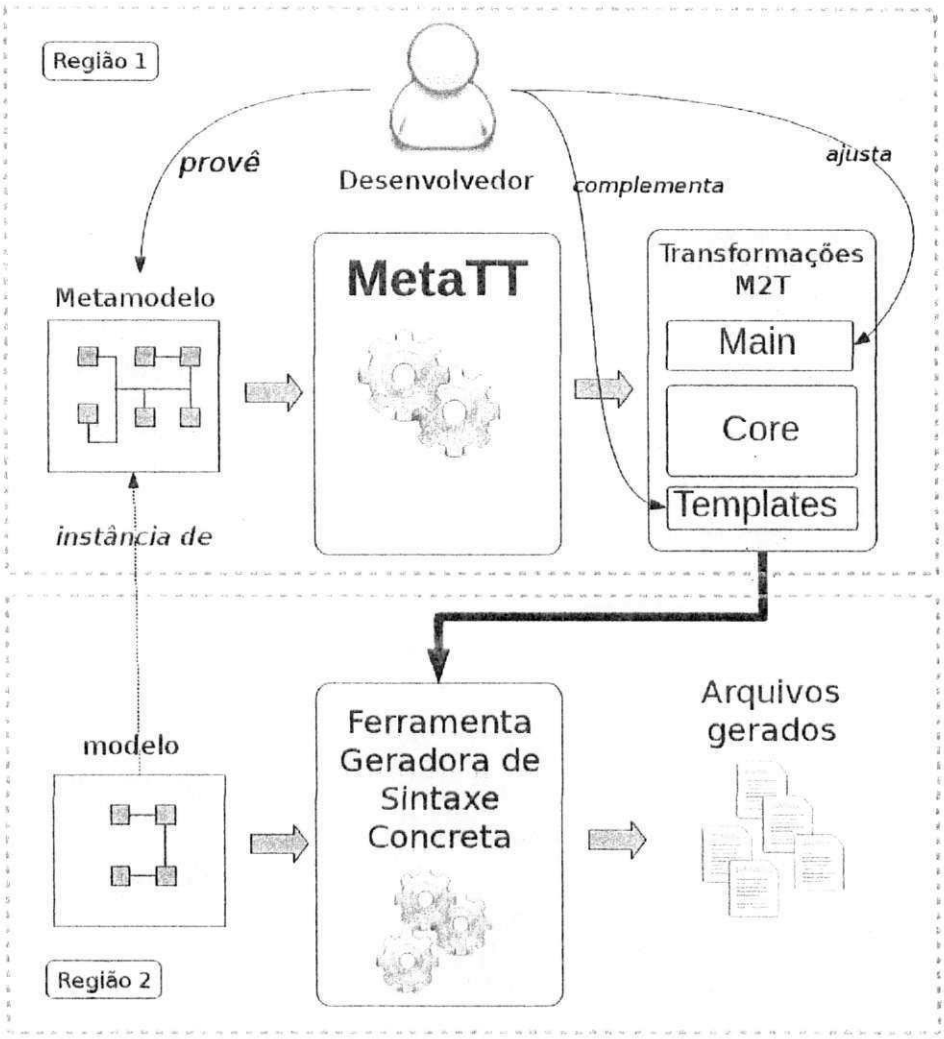

Figura 3.1: Visão geral do uso de MetaTT.

Como mostrado na Fig. 3.2, os módulos *Core* e *Templates* são compostos de outros submódulos responsáveis por tarefas específicas. O módulo *Templates* não depende de nenhum outro módulo, já o módulo *Main* atua como cliente do módulo *Core*, que concentra o processo de geração de código. Cada módulo, e submódulos correspondentes, são explicados nas próximas seções.

3.1.1 Módulo *Templates*

O módulo *Templates* provê a definição de sintaxe concreta para a linguagem alvo. Por exemplo, considerando a linguagem de programação Java como a linguagem alvo, este módulo

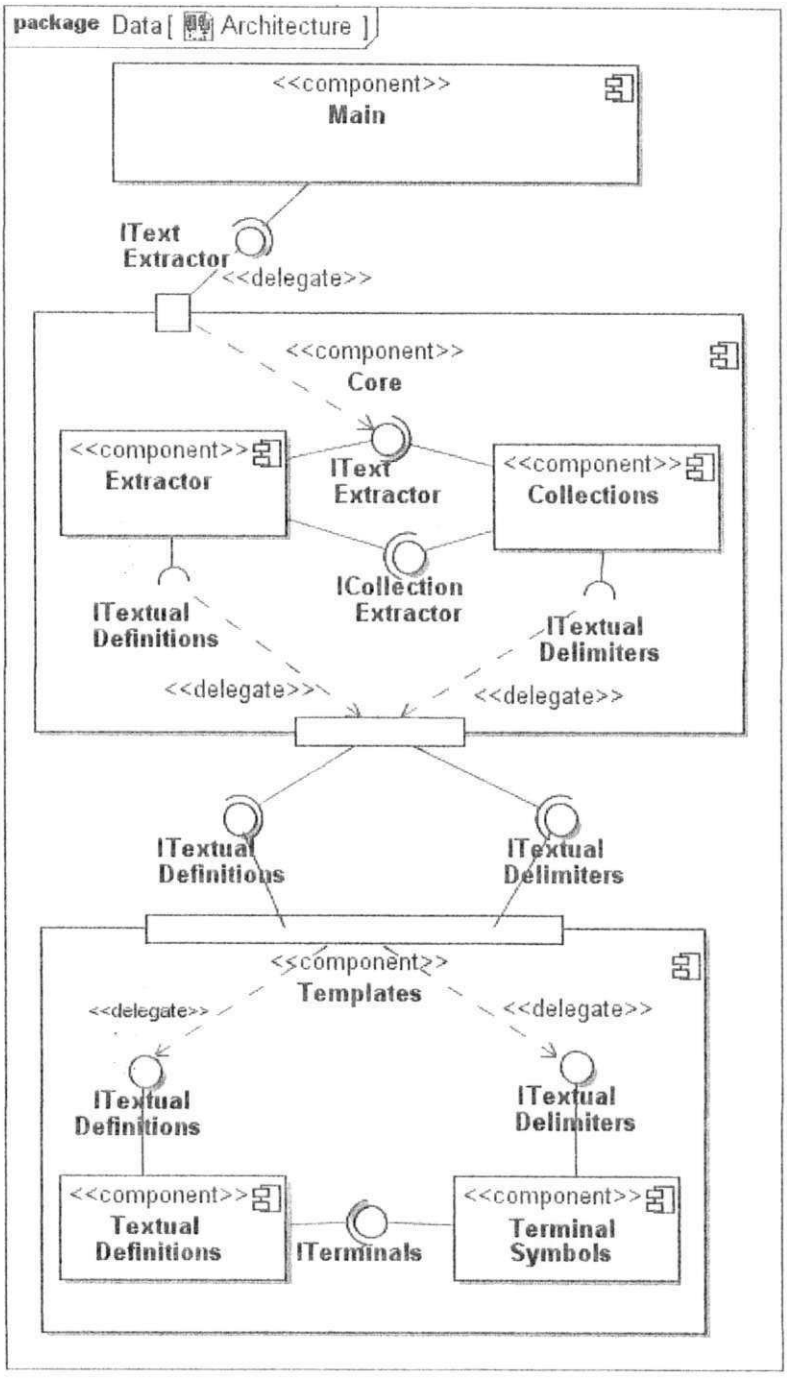

Figura 3.2: A arquitetura prescrita pelo MetaTT para geradores de código M2T.

conterá a especificação da sintaxe concreta para as assinaturas de métodos, declarações de tipos, as palavras chaves, etc.

O módulo *Templates* é formado por um conjunto de regras de templates (*template rules*) e tabelas de símbolos (*symbol tables*). As regras de templates provêem a definição da sintaxe para elementos que aparecem no metamodelo e para os quais precisa-se gerar sintaxe concreta. As tabelas de símbolos provêem partes individuais de sintaxe para elementos terminais que não estão metamodelados, ou seja, palavras chaves, caracteres separadores, delimitadores de bloco, etc.

Sempre que necessitamos de alguma informação referente à sintaxe concreta no processo de geração de texto, esta informação é obtida do módulo *Templates* através das interfaces *ITextualDefinitions* e *ITextualDelimiters*, que dão acesso aos submódulos *TextualDefinitions* e *TerminalSymbols*, respectivamente. O submódulo *TextualDefinitions* contém as regras de template (*template rules*), e cada uma delas define um template específico para cada metaclasse do metamodelo. O submódulo *TerminalSymbols* contém a tabela de símbolos (*symbol tables*) que contém as contantes *strings* correspondente aos terminais da linguagem (*p.e.*, chaves, separadores, ponto e vírgula, palavas chaves, etc.). Sempre que algum símbolo terminal é requerido em uma regra do submódulo *TextualDefinitions*, esta informação é acessada no submódulo *TerminalSymbols* através da interface *ITerminals*.

```
1  property LEFT_PARENTHESES: String= ''('' 
2  property RIGHT_PARENTHESES: String= '')''
3  property LEFT_CURLY_BRACES: String= ''{''
4  property RIGHT_CURLY_BRACES: String= ''}''
5  ...
6  property IF: String=''if''
7  property ELSE: String=''else''
8  ...
9  property DECLARATION_BEGIN_BLOCK_DELIMITER: String=
       LEFT_CURLY_BRACES
10 property DECLARATION_SEPARATOR: String=''\n\t''
11 property DECLARATION_END_BLOCK_DELIMITER: String=
       RIGHT_CURLY_BRACES
```

Figura 3.3: Delimitadores de bloco para uma declaração em PLang.

Na Fig. 3.3, ilustramos parte da tabela de símbolos provida para o caso do PLang, na

qual cada símbolo é definido como uma constante *string* (uma propriedade de MOFScript) que referencia caracteres que podem ser usados na definição da sintaxe. Nas linhas 1-4, caracteres de uso frequente são ilustrados (parênteses esquerdo e direito, chaves esquerda e direita). Tais caracteres podem ser reusados em várias regras de templates e outras definições (*p.e.*, na Fig. 3.5, linhas 10 e 12). Nas linhas 6 e 7, Fig. 3.3, são mostradas palavras chaves específicas de uma expressão condicional (*ConditionalExp*). Nas linhas 9-11, são mostrados alguns caracteres usados em declarações de blocos, tais como chaves esquerda (linha 9), caracter de fim de linha seguido por um caracter de *tab* (linha 10) ou chave direita (linha 11).

Quando um desenvolvedor de transformações M2T precisa especificar a definição da sintaxe para um dado elemento no MetaTT, ele precisa definir os valores dos elementos terminais na tabela de símbolos e definir como as regras de templates devem combinar esses elementos de maneira que formem a sintaxe para os elementos não-terminais. Por exemplo, supondo que o desenvolvedor decidiu que um comando condicional (*ConditionalSttmnt*) deve ter a sintaxe de acordo com o trecho da gramática BNF mostrado na Fig. 3.4, então ele deve definir a sintaxe concreta na regra de template como mostrado na Fig. 3.5.

sttmnt →	if (**exp**) then { **sttmnt** }
	if (**exp**) then { **sttmnt** } else { **sttmnt** }

Figura 3.4: Um trecho de uma gramática BNF, para um elemento *ConditionalSttmnt* de PLang.

A Fig. 3.5 ilustra a regra *conditionalSttmntTemplate*, que recebe a sintaxe concreta, extraída previamente, dos elementos realacionados: *(i)* uma condição (*condition*), *(ii)* um bloco de comandos para a condição avaliada como verdadeira (*thenSttmnts*), seguido *(iii)* de um bloco de comandos para a condição avaliada como falsa (*elseSttmnts*). Nas linhas 4-7, a sintaxe do *if*, mais a condição é acrescentada à saída do programa, bem como o bloco de comandos. Nas linhas 9-13, o bloco de comandos *else* é acrescentado à saída do programa apenas se o argumento *else* não for vazio.

Vale ressaltar que, apesar de termos separado bem os símbolos terminais na Fig. 3.5, usando constantes advindas do submódulo *SymbolTables*, o desenvolvedor poderia ter usando os terminais da linguagem diretamente na definição da regras, tornando-a ainda mais simples.

```
1  module::conditionalSttmntTemplate(condition:String,thenSttmnts:
       String,elseSttmnts:String):String{
2   var code: String = ''''
3
4   code += IF+LEFT_PARENTHESES+condition+RIGHT_PARENTHESES
5   code += LEFT_CURLY_BRACES+''\n''
6   code += thenSttmnts+''\n''
7   code += RIGHT_CURLY_BRACES
8
9   if(not elseSttmnts.trim() = ''''){
10   code += ELSE+LEFT_CURLY_BRACES+''\n''
11   code += elseSttmnts+''\n''
12   code += RIGHT_CURLY_BRACES
13  }
14  result = code
15 }
```

Figura 3.5: Exemplo de regra de template para o trecho de gramática descrito na Fig. 3.4.

Um exempo de saída textual para a regra de template na Fig. 3.5 é o código na Fig. 3.6.

```
1 if(a < b){
2  println(''a is smaller than b'')
3 }else{
4  println(''a is greater than or equal to b'')
5 }
```

Figura 3.6: Exemplo de código resultante da regra de template na Fig. 3.5.

Uma das principais características da abordagem do MetaTT é simplificar a definição da sintaxe concreta e desacoplar o módulo *Templates* dos outros módulos do gerador M2T. Dessa maneira, quando alguma mudança na especificação da sintaxe concreta precisa ser feita, ela é realizada com a modificação de um regra e uma tabela de símbolos bem definidas e localizadas. Isto fica evidente ao percerber-se que as transformações do módulo *Templates* não possuem *imports* para outros módulos, como consequência da independência entre este módulo e os demais. Como resultado, podemos alterar a sintaxe concreta a ser gerada trocando-se os artefatos do módulo *Templates* por outros (respeitando-se os contratos das interfaces *ITextualDefinitions* e *ITextualDelimiters*), sem precisar mudar outros módulos. Esta modificação consiste na inserção de um novo módulo com transformações e regras com os

mesmos identificadores, modificando-se apenas a sua implementaçao.

Apesar do exemplo apresentado estar relacionado ao PLang, que representa conceitos de linguagens de programação, nossa abordagem não está restrita a gerar texto apenas para linguagens de programação. O MetaTT depende de um metamodelo, mas não depende da semântica específica de um domínio meta-modelado. Também é importante destacar que apesar de usarmos uma gramática BNF como descrição da sintaxe no nosso exemplo, nós não obrigamos a adoção de qualquer regra de formação em nossos templates, de maneira que o desenvolvedor é livre pra mudar a sintaxe de acordo com o que for conveniente. Por exemplo, se o desenvolvedor precisa de um gerador M2T para a linguagem de marcação HTML, ele precisa (1) prover um metamodelo de HTML para o MetaTT, (2) ajustar o módulo Main para transformar o elemento raíz do metamodelo de HTML (que deve ser o metaelemento "html") e (3) preencher as regras de templates e definir os terminais. Com o objetivo de realizar a atividade (3) ele não depende de uma descrição em gramática, e a descrição da sintaxe pode ser feita pelo desenvolvedor com base em exemplos ou no conhecimento prévio que ele já tenham sobre a sintaxe de HTML.

3.1.2 Core

Este módulo provê regras que são responsáveis por cuidar da transformação de cada metaclasse do metamodelo. Ele é dividido em outros dois submódulos: *Extractor* e *Collections*.

O submódulo *Extractor* é responsável pela extração das informações de sintaxe concreta de cada metaclasse do metamodelo, enquanto que o submódulo *Collections* gerencia a extração das informações de sintaxe concreta para coleções de instâncias de uma dada metaclasse. Estes dois submódulos dependem um do outro uma vez que são responsáveis por funcionalidades complementares. Sempre que informações de sintaxe concreta para uma coleção de elementos são necessárias dentro do submódulo *Extractor*, ele acessa a funcionalidade provida pelo submódulo *Collections* através da interface *ICollectionExtractor* e sempre que a transformação de um único elemento precisa ser feita dentro do submódulo *Collections*, ele acessa a funcionalidade do submódulo *Extractor* através da interface *ITextExtractor*.

As regras no submódulo *Extractor* são nomeadas *regras extratoras*. Para cada metaclasse do metamodelo há uma regra extratora responsável por invocar outras regras capazes

de extrair as partes de sintaxe concreta (uma para cada atributo ou referência daquela metaclasse) que juntas formarão a sintaxe completa para aquela metaclasse. A tarefa de combinar esses pedaços de sintaxe é atribuída às regras de template (através da interface *ITextualDefinitions*) do módulo *Templates*. Por exemplo, para a metaclasse *ConditionalSttmnt*, ilustrada na Fig. 3.7, há uma regra extratora, apresentada na Fig. 3.8. Podemos perceber uma relação bastante próxima entre os relacionamentos de composição da metaclasse *ConditionalSttmnt* e a estrutura do código apresentado na Fig. 3.8. Para cada relacionamento saindo da metaclasse *ConditionalSttmnt* com destino para uma metaclasse *Statement* ou *Expression*, há a invocação de uma regra extratora ou de coleção, ou seja, há uma correspondência entre os papéis *condition*, *thenSttmnts* e *elseSttmnts* na Fig. 3.7 e as linhas 2, 3 e 4, respectivamente, na Fig. 3.8. Na linha 5, uma regra de template, chamada *conditionalSttmntTemplate*, é invocada para combinar os pedaços de sintaxe capturados nas linhas 2-4 formando a sintaxe completa do elemento *ConditionalSttmnt*.

Nas linhas 3 e 4, na Fig. 3.8, uma regra de coleção é invocada. Tal regra é provida pelo submódulo *Collections*. Ela é responsável por abstrair o processamento de coleções de elementos e adicionar a estes seus correspondentes delimitadores textuais, obtidos do módulo *Templates* através da interface *ITextualDelimiters*. Uma vez que os papéis *thenStatements* e *elseStatements* são multivalorados, tratar estes elementos dentro da própria regra *getConditionalSttmntCode()* a tornaria mais complexa e prejudicaria a sua legibilidade (p.e., seria necessário adicionar laços e variáveis de controle). Por separar estes três tipos de regras em três diferentes módulos, MetaTT ajuda a simplificar as regras extratoras.

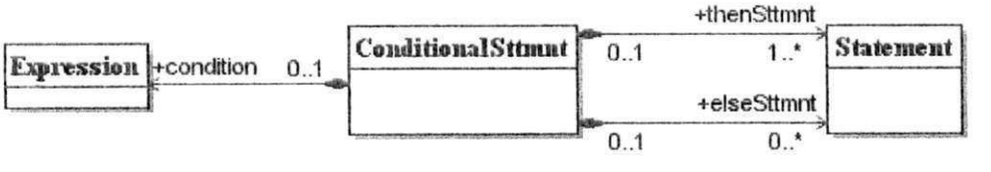

Figura 3.7: Um metaelemento *ConditionalSttmnt* e seus metaelementos relacionados.

```
1 PLang.ConditionalSttmnt::getConditionalSttmntCode() : String {
2   var condition:String= self.condition.getExpressionCode()
3   var thenSttmnts:String= getStatementCollectionCode(self.
        thenSttmnts)
4   var elseSttmnts:String= getStatementCollectionCode(self.
        elseSttmnts)
5   result= conditionalSttmntTemplate(condition, thenSttmnts,
        elseSttmnts)
6 }
```

Figura 3.8: Uma regra de extração para um comando condicional de PLang.

3.1.3 Main

Este módulo é responsável tanto pelo ponto inicial como pelo ponto final da transformação de modelo para texto. Ele contém uma regra de entrada que recebe um modelo de entrada, identifica a metaclasse raiz [1] deste metamodelo e invoca a regra correspondente do módulo *Core* que, por sua vez, retorna a sintaxe concreta para o dado elemento. Finalmente, a regra de entrada realiza a persistência da sintaxe concreta gerada, finalizando o processo de transformação.

Sempre que se quer gerar texto para um dado metaelemento, escreve-se uma regra principal que invoque a regra adequada àquele elemento (uma regra extratora já especificada no módulo *Core*). As figuras 3.9 e 3.10 são dois exemplos ligeiramente diferentes de regras principais. Em ambas, a declaração do tipo de contexto da regra (na linha 1) permitem que um elemento seja automaticamente selecionado no modelo de entrada. Na linha 1 da Fig. 3.9, a transformação detecta o elemento de contexto *ProgramDeclaration*, seleciona os elementos de mesmo tipo no modelo de entrada e invoca a regra de extração chamada *getProgramDeclarationCode()* (na linha 2) do módulo *Core*. A persistência é feita nas linhas 3-4. A Fig. 3.10 é análoga à Fig. 3.9, exceto pelo fato de que ele realiza a transformação de elementos do tipo *FunctionDeclaration*, ou seja, ela é responsável por gerar a sintaxe

[1] A metaclasse raiz de um metamodelo é aquela que representa o conceito fundamental do mesmo, aquela que agrega os demais conceitos. A definição da metaclasse raiz depende do domínio metamodelado. Por exemplo, em um metamodelo fidedigno aos conceitos da linguagem Java a metaclasse raiz pode ser um arquivo de código fonte, já em um metamodelo simplificado (que, por exemplo, ignore os conceitos de interface, enumerações, etc.) a metaclasse raiz pode uma classe simples.

concreta para declarações de funções em PLang.

```
1 plang.ProgramDeclaration::main(){
2   var code:String= self.getProgramDeclarationCode()
3   file(self.name + ".plang")
4   print(code)
5 }
```

Figura 3.9: Uma regra principal que gera código para todo elemento *ProgramDeclaration*.

```
1 plang.FunctionDeclaration::main(){
2   var code:String= self.getFunctionDeclarationCode()
3   file(self.name + ".plang")
4   print(code)
5 }
```

Figura 3.10: Uma regra principal que gera código para todo elemento *FunctionDeclaration*.

Um melhor controle do processo de geração é possível porque a persistência é isolada em um módulo, ou seja, *Main*, enquanto o processo de extração de texto fica isolado em outro módulo, ou seja, *Core*. O módulo *Main* é o único, dentre todos os módulos prescritos, que tem a presença de comandos *print*. Isto confere ao desenvolvedor de transformações um melhor controle sobre quais elementos do metamodelo devem ser selecionados para serem persistidos em arquivos.

3.1.4 Integração entre os Módulos

Nesta seção, nós apresentamos uma visão geral do fluxo de execução entre os módulos arquiteturais.

Na Fig. 3.11, nós apresentamos as principais atividades realizadas através dos módulos, que são representados por raias de UML[2]. Inicialmente, no módulo *Main*, os elementos a serem transformados são selecionados. Então, o fluxo progride para a atividade de extração de dados, executada no módulo *Core* (pelos submódulos *Extractor* e *Collections*) do qual omitimos os detalhes aqui por questões de simplificação. Depois que os dados necessários

[2]Raias podem ser utilizadas para expressar partições hierárquicas [47]. Neste caso, cada raia é utilizada como uma partição (módulo) da arquitetura do MetaTT.

são extraídos, eles são combinados pelo módulo *Templates*. O texto gerado é então persistido pelo módulo *Main* que finaliza o processo.

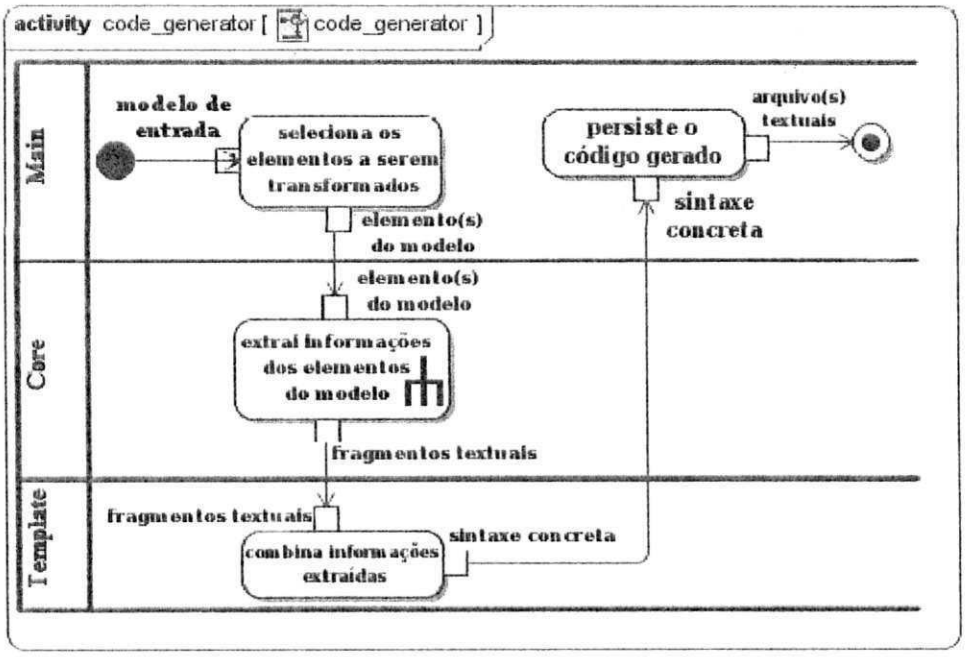

Figura 3.11: Visão geral das atividades realizadas no MetaTT.

Na Fig. 3.12, ilustramos as interações ocorridas entre os módulos durante o processo de geração de texto para um comando condicional. Neste contexto, a regra extratora *getConditionalStatementCode()* é invocada a partir do módulo *Main*. A partir desse ponto, enfatizamos a troca de mensagens entre as transformações em diferentes módulos (que podem ser obervadas pelas linhas de vida) para a regra apresentada na Fig. 3.8. As chamadas de mensagens progridem para as regras extratoras *getConditionalStatementCode()* e *getExpressionCode()*. Então, há duas chamadas (ilustradas num fragmento combinado *ref* de UML) para a regra *getStatementCollectionCode()* (uma para o conjunto de comandos a serem executados se a condição for avaliada como verdadeira e, para o caso oposto, uma outra chamada para um outro conjunto de comandos), ambas do submódulo *Collections*. Finalmente, os códigos parciais são combinados no módulo *Templates* e o resultado é retornado de volta ao módulo *Main* para ser persistido.

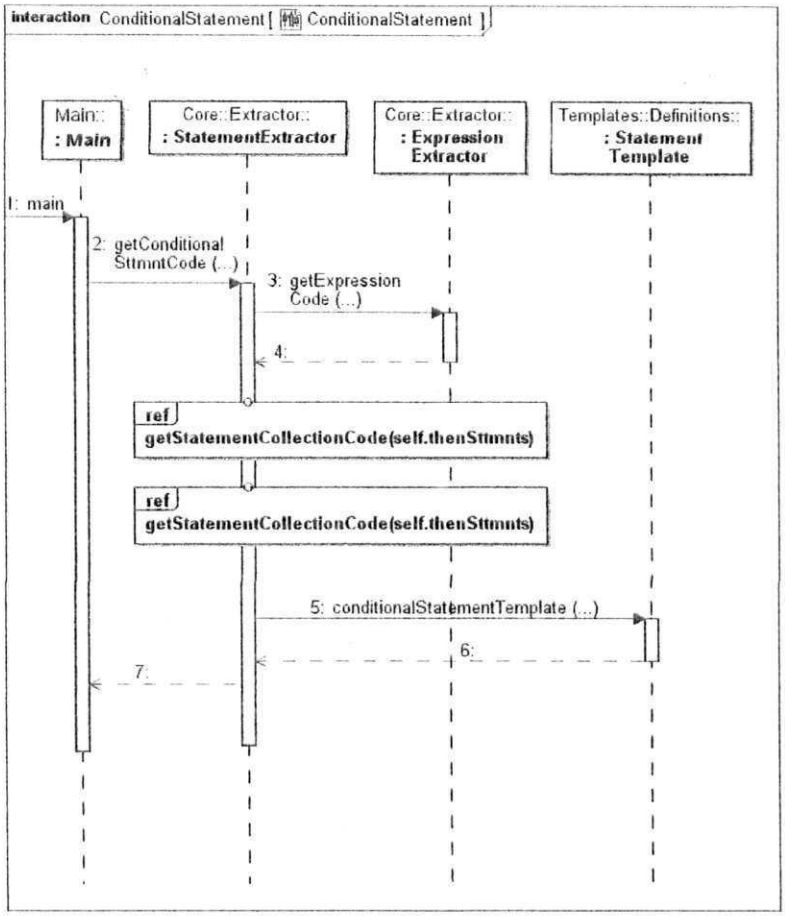

Figura 3.12: Diagrama de sequência ilustrando as interações entre os módulos dentro do gerador M2T de PLang.

3.2 Geração dos Artefatos Arquiteturais

Depois de prescrever os módulos arquiteturais e a maneira como eles interagem entre si, é preciso elaborar seus artefatos internos, ou seja, as transformações (e seus relacionamentos) e as assinaturas de regras. Questões importantes que surgem neste processo são: Quantas, quais e de que tipo devem ser as transformações a serem criadas para cada módulo? Quais são as regras que cada transformação deve conter? Como as regras serão arranjadas nas transformações de maneira coesa? Que relacionamentos devem existir entre as transformações?

A fim de responder essas perguntas de projeto e a fim de facilitar o processo de elaboração de tais artefatos, um *modelo de referência* deve ser adotado. Este modelo não deve ser confundido com a *arquitetura de referência*. Enquanto aquela contém a especificação

dos módulo se submódulos do nosso projeto de transformação, este contém informações que serão úteis para derivar artefatos específicos de cada módulo e submódulo (como transformações e regras). Tais informações são obtidas do metamodelo para o qual a sintaxe concreta deve ser gerada.

Os elementos do *modelo de referência* são usados, mais tarde, para guiar a obtenção dos artefatos, tais como transformações de modelo para texto e suas regras de transformação.

As próximas subseções descrevem o método para obtenção do *modelo de referência* e como os artefatos são elaborados com base nas informações providas por este modelo.

3.2.1 Obtenção do Modelo de Referência a partir do Metamodelo

A partir do metamodelo, deve-se seguir um processo com 4 passos a fim de se obter o modelo de referência. Este processo sintetiza a abordagem descrita neste trabalho para a obtenção de uma referência para implementação dos artefatos de maneira que eles se encaixem nos módulos propostos pela arquitetura do MetaTT. A seguir apresentamos os passos e a aplicação deles sobre o metamodelo PLang:

passo 1 Selecionar as informações (do metamodelo) necessárias para elaborar um modelo de referência. Abaixo, os elementos do metamodelo (na esquerda) originam os *elementos de referência* [3] (na direita).

- metaclasses → *transformações de referência* (identificadas pelo mesmo nome da metaclasse correspondente);
- relacionamentos de herança → *associações de referência* direcionadas (seguindo a direção filho-pai no metamodelo).

A partir deste ponto, o restante das informações no metamodelo (composições, atributos, etc.) não é mais usado, apenas os elementos de referências criados (ou seja, *associações de referência* e *transformações de referência*).

A razão para a existência do *passo 1* é simplificar o modelo de referência selecionando apenas as informações essenciais do metamodelo: metaclasses e relacionamentos de herança.

[3] A partir deste ponto, todo elemento será um *elemento de referência*.

Na Fig. 3.13, ilustramos o resultado da aplicação do passo 1 sobre o metamodelo de PLang. Depois disso, as metaclasses tornam-se transformações de referência e relacionamentos de herança tornam-se associações de referência direcionadas. Recomendamos ao leitor observar a Fig. 2.5 a fim de identificar a diferença entre o metamodelo e o resultado da aplicação do passo 1.

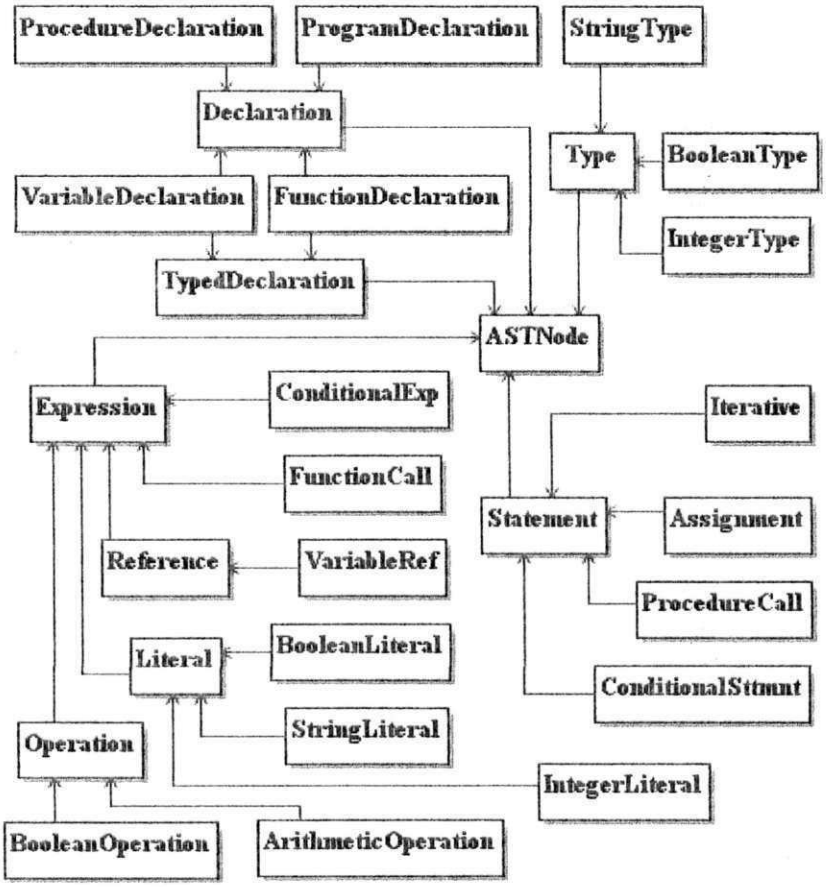

Figura 3.13: Uma versão preliminar do modelo de referência após a aplicação do *passo 1* sobre o metamodelo de PLang.

passo 2 Verificar se alguma *transformação de referência* na estrutura resultante (do passo 1) apresenta duas ou mais *associações de referência* em direção a qualquer outra *transformação de referência* (o que mostra se há herança múltipla no metamodelo). Se não, pula-se para o passo 3. Se sim, aplica-se o processo de normalização: Enquanto houver duas ou mais associações a partir de uma dada *transformação de referência*, deve-se tomar esta *transformação de referência* e mesclar todas as outras *transformações de*

referência apontadas por ela. Mesclar significa que as transformações de referência apontadas tornar-se-ão uma única transformação de referência e que toda *associação de referência* entre a transformação de referência que aponta e as que são apontadas tornar-se-ão uma única associação de referência[4]. Este passo garante que a estrutura do modelo de referência terá, em termos de grafos, a forma de uma árvore direcionada ou uma floresta direcionada.

No *passo 2*, como foi detectada a existência de um elemento, vamos chamá-lo de ***e***, com herança múltipla. Decide-se mesclar seus elementos ascendentes porque eles vão dar origem às *transformações de referência* e então o elemento ***e*** dará origem a uma *regra de referência*; esta regra pode ser adicionada para apenas uma das transformações porque observa-se que duplicar a regra em mais de um transformação pode causar problemas (tanto na fase de compilação como na fase de execução), então observa-se que mesclar os elementos ascendentes e manter apenas uma *regra de referência* relativa ao elemento ***e*** dentro da transformação mesclada é uma melhor escolha.

No exemplo do PLang, é detectada a presença de duas associações de referência a partir da mesma transformação de referência para outras, ou seja, ambas associações de referência a partir de uma *FunctionDeclaration* e de uma *VariableDeclaration* alcançando uma *Declaration* e uma *TypedDeclaration*. De acordo com a abordagem, as transformações de referência apontadas são mescladas em uma única, resultando na *Declaration_TypedDeclaration_Merged*, ilustrada na Fig. 3.14.

passo 3 Para toda associação de referência que tem uma transformação de referência folha [5], esta torna-se uma *regra de referência* contida na sua transformação de referência pai. Caso a *transformação de referência* não tenha uma *transformação de referência* ascendente, ela incorpora uma *regra de referência* referente a si mesma.

Agrupar as *regras de referência* que são derivadas a partir dos metaelementos que têm um elemento ascendente em comum é a maneira de manter boa coesão.

[4]Neste processo de mesclagem entre dois elementos do modelo de referência, não há problemas relativos ao conflito entre seus atributos, uma vez que, no *passo 1*, os atributos foram descartados e apenas as metaclasses e relacionamentos de herança foram mantidas, na forma de novos elementos e associações de referência.

[5]Denominamos de folha uma transformação de referência que não é alcançada por nenhuma associação de referência.

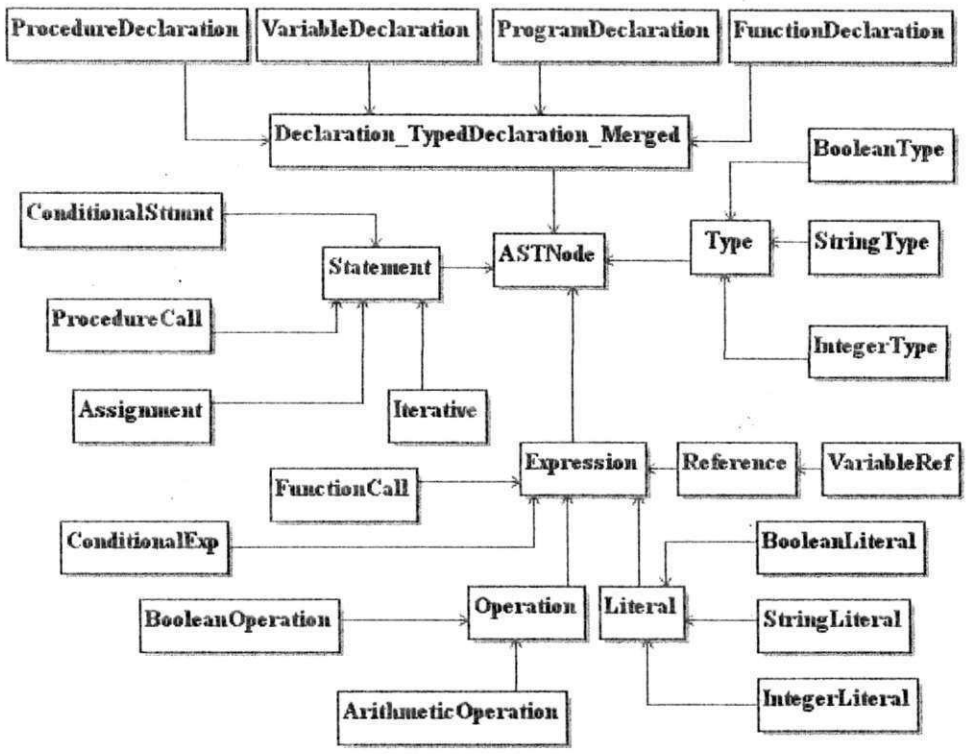

Figura 3.14: Uma versão preliminar do modelo de referência após a aplicação do *passo 2*.

Na Fig. 3.15, ilustramos o resultado da aplicação do passo 3. Por exemplo, os elementos folha das transformações referência *FunctionCall* e *ConditionalExp* tornam-se regras de referência que passam a fazer parte da transformação de referência *Expression*. Entretanto, a metaclasse *Operation* não é transformada em uma regra de referência uma vez que não é um elemento folha (*p.e.*, a metaclasse *BooleanOperation* é um elemento filho de *Operation*).

passo 4 Reverter a direção das associações de referência. Os membros que antes apontavam passam a ser apontados e os que eram apontados passam a apontar. Então, as associações de referência devem ser marcadas como uma relação de *import*[6].

No *passo 4*, alguns relacionamentos de *import* são derivados a partir das associações resultantes porque elas refletem o fato de que os elementos mais abstratos precisam da definição dos elementos menos abstratos.

[6]Uma associação marcada como *import* saindo de uma transformação de referência *A* para uma outra *B* significa que *A* poderá usar os conceitos de *B*.

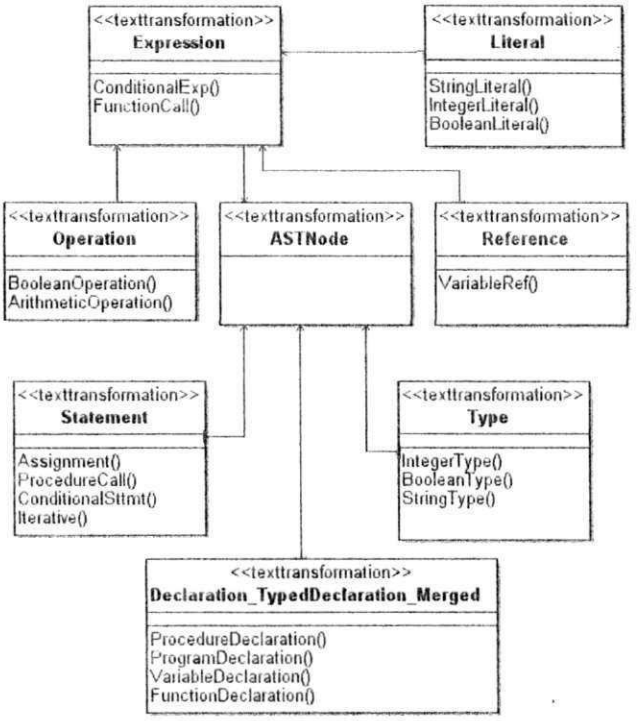

Figura 3.15: Uma versão preliminar do modelo de referência após a aplicação do *passo 3*.

Como prescrito, a direção das associações de referência estão invertidas e marcadas como associações de *import*, mostradas na Fig. 3.16.

Na Fig. 3.16, o modelo de referência para a implementação do gerador de código M2T para PLang é mostrado. Os retângulos marcados como *texttransformation* são as transformações de referência (*p.e.*, *ASTNode* e *Expression*). As setas marcadas como *import* entre as transformações de referência são associações de referência (*p.e.*, de *ASTNode* para *Expression* e de *Expression* para *Literal*). Adicionalmente, dentro de cada transformação de referência tem-se as regras de referência (*p.e.*, *ConditionalExp()* e *FunctionCall()*, ambas na transformação de referência *Expression*).

Cada um dos elementos de referência apresentados na Fig. 3.16 é usado para guiar a elaboração das transformações de modelo para texto e suas respectivas regras de transformação. Na abordagem proposta neste trabalho, o desenvolvedor de transformações usa o modelo de referência para saber quais transformações ele achará em cada módulo e as regras que o mesmo achará em cada transformação, bem como os relacionamentos entre elas.

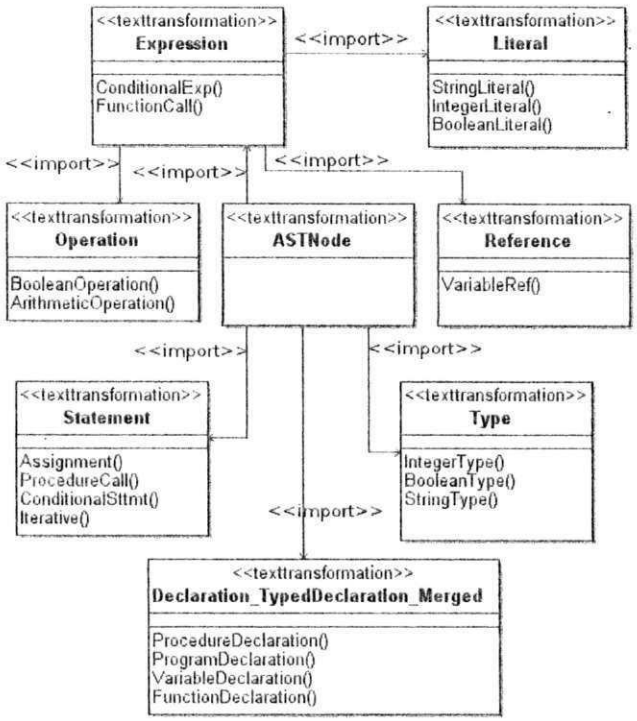

Figura 3.16: O modelo de referência obtido depois da aplicação do *passo 4*.

3.2.2 Obtenção dos Artefatos a partir do Modelo de Referência

Uma vez que sabemos separar as diferentes preocupações dentro de um gerador M2T por meio da arquitetura de referência proposta (tratando de persistência no módulo *Main*, definindo a sintaxe nos módulo *Templates* e controlando o processo de geração nos submódulos de *Core*), nós usamos o modelo de referência para projetar as transformações de modelo para texto, com as suas respectivas regras, dentro de cada módulo. Em geral, este procedimento é dado da seguinte maneira e vamos exemplificá-lo utilizando a transformação de referência *Declaration_TypedDeclaration_Merged* e mais especificamente a regra de referência *ProcedureDeclaration*. Para cada transformação de referência no modelo de referência, derivamos os seguintes artefatos:

- Uma transformação dentro do submódulo *Extractor* que, por sua vez, contém uma regra extratora para cada regra de referência desta transformação de referência. As regras extratoras são expostas pela interface *ITextExtractor*. Elas dependem da especi-

ficação das interfaces *ITextualDefinitions* e *ICollectionExtractor*. Apresentamos um exemplo de transformação e regra deste tipo na Fig. 3.17.

```
1
2  texttransformation
3  Declaration_TypedDeclaration_Merge_Extractor
4  (in plang:"http://gmf.ufcg.edu.br/mdd/plang"){
5    (...)
6
7    plang.ProcedureDeclaration::getProcedureDeclarationCode():
         String{
8
9    var name: String = self.name
10   var arguments: String =
11     getVariableDeclarationCollectionCode(self.arguments)
12   var statements: String =
13     getStatementCollectionCode(self.statements)
14   var calls: String =
15     getProcedureCallCollectionCode(self.calls)
16
17   result =
18     procedureDeclarationTemplate(
19                         name,
20                         arguments,
21                         statements,
22                         calls )
23   }
24
25 (...)
26
27 }
```

Figura 3.17: Exemplo de transformação derivada do modelo de referência com uma regra extratora derivada.

- Uma transformação dentro do submódulo *Collections* que, por sua vez, contém uma regra de coleção para cada regra de referência. As regras de coleção são expostas pela interface *ICollectionExtractor* e elas dependem da interface *ITextualDelimiters* para funcionar corretamente. Apresentamos um exemplo de transformação e regra deste tipo na Fig. 3.18.

```
texttransformation
Declaration_TypedDeclaration_Merge_Collection
(in plang:"http://gmf.ufcg.edu.br/mdd/plang"){

module::
  getProcedureDeclarationCollectionCode
    (procedureDeclarations : List):String{

    var startingEmbrace :
      String = procedureDeclarationStartingEmbrace
    var separator :
      String = procedureDeclarationSeparator
    var endingEmbrace :
      String = procedureDeclarationEndingEmbrace

    var code : String =""
    code += startingEmbrace
    procedureDeclarations->
      forEach(
        someProcedureDeclaration:
          plang.ProcedureDeclaration){
            code +=
              someProcedureDeclaration.
                getProcedureDeclarationCode()
            if(someProcedureDeclaration != procedureDeclarations
              .last())
              code += separator
          }
          code += endingEmbrace
          result = code
  }
}
```

Figura 3.18: Exemplo de transformação derivada do modelo de referência com uma regra de coleção derivada.

- Uma transformação no submódulo *Templates* que, por sua vez, contém um esqueleto de uma regra template para cada regra de referência. As transformações deste submódulo não dependem de nenhuma outra em outro submódulo e estão expostas para as demais através da interface *ITextualDefinitions*. Apresentamos um exemplo de transformação e regra deste tipo na Fig. 3.19.

```
1  texttransformation
2  Declaration_TypedDeclaration_Merge_Templates
3  (in plang:"http://gmf.ufcg.edu.br/mdd/plang"){
4
5  module::
6      procedureDeclarationTemplate
7          (name: String,
8              arguments: String,
9                  statements: String,
10                     calls: String)
11                                 :String{
12
13                         result += "\t"
14                         result += name
15                         result += " ( "
16                         result += arguments
17                         result +=" ) "
18                         result += statements
19         }
20
21 }
```

Figura 3.19: Exemplo de transformação derivada do modelo de referência com uma regra de template derivada.

- Três propriedades no submódulo *TerminalSymbols* para cada regra de referência (tais como símbolos para indicar começo e fim de coleções e um separador, como prescrito na Fig. 3.3). Tais delimitadores são expostos pela interface *ITextualDelimiters*.

O papel desempenhado pelas transformações geradas está de acordo com o papel de cada módulo (e submódulo) que a contém. Desta forma, a transformação *StatementExtractor*, do submódulo *Extractor*, realiza a extração de informações, enquanto que a transformação *StatementTemplates*, do submódulo *Templates*, especifica a sintaxe concreta para os elementos do tipo *Statement*. Casos análogos aplicam-se às demais transformações.

O desenvolvedor define as regras e transformações em cada módulo seguindo a abordagem descrita ou ele pode utilizar o suporte ferramental provido. Caso haja a ausência na implementação de alguma das regras previstas, alguns erros no comportamento das transformações devem ser observados. Caso falte uma regra extratora para um determinado ele-

mento, todas as regras extratoras dos elementos que dependem conceitualmente dele serão afetadas. Problemas equivalentes ocorrerão na falta de regras prescritas para os demais módulos e submódulos.

3.2.3 Resolução de Ciclos

Como explicado na Subseção 3.1.2, e observado pelas figuras 3.7 e 3.8, os relacionamentos de composição do metamodelo são refletidos na estrutura das regras extratoras. Apesar de esta estratégia ajudar na concepção e automação das regras extratoras, ela pode levar a um problema: a presença de ciclos nos relacionamentos de composição pode dar origem *a chamadas recursivas infinitas entre as regras de transformação*, muitas vezes difíceis de serem detectadas. Para evitar este problema, nós detectamos os possíveis ciclos a partir de um algoritmo que utiliza uma estratégia de caminhamento nas metaclasses do metamodelo a partir dos relacionamentos de composição seguindo uma abordagem de busca em profundidade. Isto permite que detectemos todos os possíveis ciclos a partir da estrutura do metamodelo, antes mesmo de implementar as transformações.

É importante destacar que um ciclo conceitual no metamodelo dá origem a chamadas cíclicas no gerador de código apenas se as instâncias dos metamodelos apresentarem o ciclo e só poderão ser observadas em tempo de execução. Alguns ciclos detectados não devem ser fonte de chamadas cíclicas. Por exemplo, para o metaelemento *ConditionalSttmnt* apresentado na Fig. 3.7, nosso algoritmo de detecção acusa que há um possível ciclo porque um *ConditionalSttmnt* é composto por dois grupos de comandos (*thenSttmnt* e *elseSttmnt*) e, nestes grupos, outros elementos *ConditionalSttmnt* podem ser encontrados. Entretanto, o ciclo só ocorrerá se a instância do modelo referenciar o próprio *ConditionalSttmnt* no grupo de comandos dos seus blocos, o que seria uma inconsistência no modelo e não nas transformações. Por conta disso, os ciclos precisam ser analisados caso a caso pelo desenvolvedor, que pode utilizar o nosso algoritmo de detecção como auxílio.

3.3 Suporte Ferramental

Apesar de definirmos a arquitetura de referência e os passos necessários para a escrita de geradores com MetaTT, realizá-los manualmente, em um cenário com um metamodelo com-

plexo, é uma tarefa intrisecamente cansativa e sujeita a erros devido a sua complexidade. A fim de abordar esta questão, implementamos um conjunto de artefatos e transformações (tanto M2M quanto M2T) com o intuito de dar suporte ferramental à nossa abordagem. Esses artefatos estão ilustrados na Fig. 3.20 e são os seguintes:

Entrada A entrada para a ferramenta do MetaTT é um *Metamodelo* descrito em Ecore.

Ferramenta MetaTT O suporte ferramental é composto por três módulos internos. Ao receber um metamodelo na entrada, este é usado pelos dois primeiros módulos (*Gerador de Modelo de Referência* e *Resolvedor de Ciclos*), e os resultados destes, juntamente com o metamodelo de entrada, são utilizados por um terceiro módulo (*Gerador de Transformações M2T*).

Gerador de Modelo de Referência Este módulo toma o mesmo metamodelo da entrada e aplica sobre ele os passos descritos na Seção 3.2. Ao final, obtemos um modelo de referência para o metamodelo em questão. Ele é implementado na forma de transformações M2M escritas na linguagem ATL.

Resolvedor de Ciclos Este módulo toma o mesmo metamodelo da entrada e verifica a existência de ciclos de composição nele, de acordo com o processo descrito na Seção 3.2.3. Ele é implementado na forma de transformações M2M escritas na linguagem ATL. Depois que a ferramenta detecta os ciclos, o modelo de saída deve passar por análise do desenvolvedor, que deve eliminar os ciclos manualmente utilizando alguma ferramenta de edição de modelos.

Gerador de Transformações M2T Este módulo toma três modelos como entrada: o *Metamodelo* dado como entrada para a ferramenta, o *modelo de referência* e o *modelo de ciclos* obtidos a partir dos dois módulos anteriores. O *gerador de transformações* utiliza o modelo de referência para, de acordo com a arquitetura do MetaTT, definir as transformações e as regras que devem ser geradas. Ao gerar as regras extratoras ele utiliza o *modelo de ciclos* como auxiliar para evitar a introdução de ciclos de recursão infinita no gerador de código resultante.

Saída A saída do processo é um gerador de código para o metamodelo dado como entrada e seguindo a arquitetura definida pelo MetaTT.

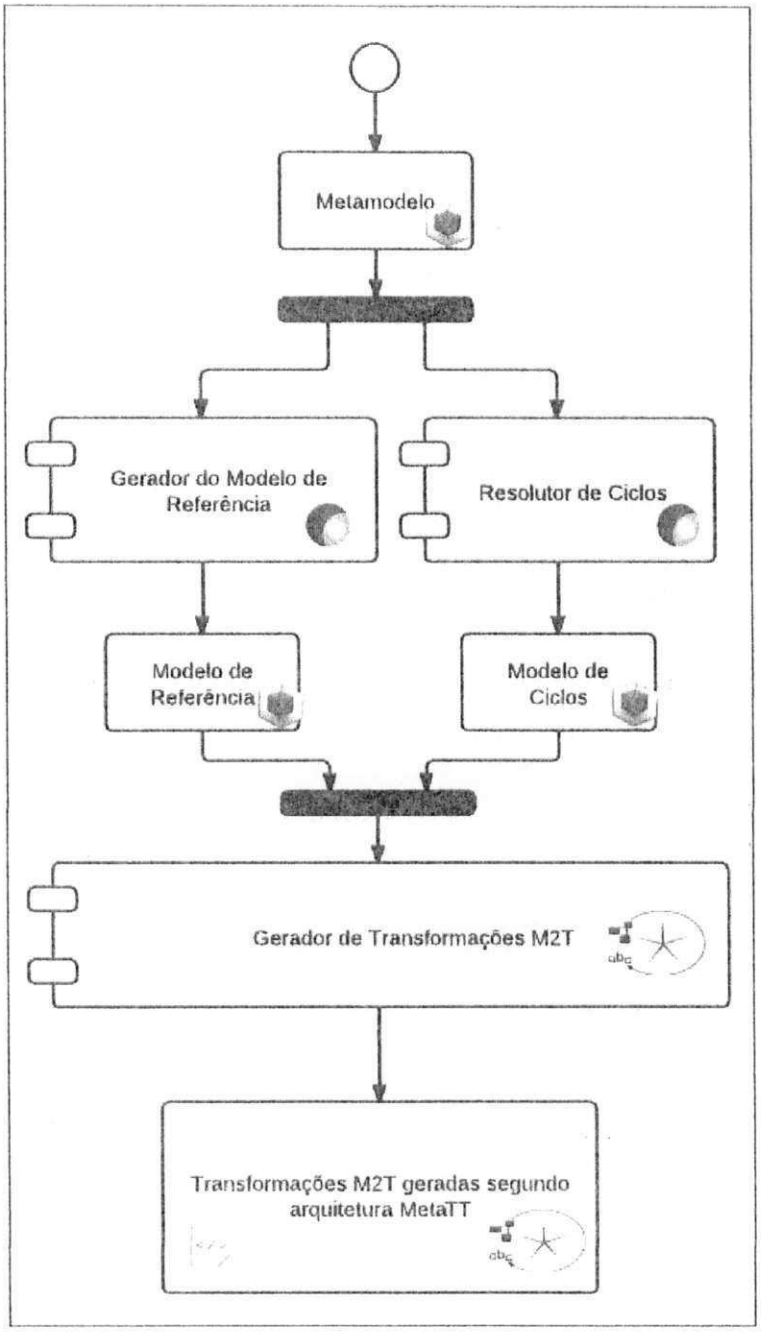

Figura 3.20: Arquitetura do suporte ferramental do MetaTT.

Apesar do fato de que o suporte ferramental é capaz de gerar a maioria da infraestrutura do gerador M2T e suas transformações, há algumas situações nas quais necessita-se da interferência do desenvolvedor, a exemplo de:

1. Configurar importações adicionais nas transformações (quando uma transformação de-

pende de outra para funcionar corretamente). [7]

2. Editar o modelo de ciclos para evitar introdução de recursão infinita no gerador de código.

3. Especificação a sintaxe concreta dentro do módulo *Templates*, como previamente indicado na Seção 3.1.1;

4. Definir (no módulo *Main*) quais são os elementos elementos raízes a serem transformados.

Limitações. O suporte ferramental ainda precisa ser melhor integrado para que funcione como uma suíte de ferramentas e possa ser integrado a um ambiente de desenvolvimento ou invocado por linha de comando. A edição do modelo de ciclos também depende de alguma ferramenta externa a nossa abordagem.

Mais detalhes sobre as transformações disponíveis e o projeto do suporte ferramental do MetaTT estão disponíveis em [30].

[7] Esta é uma tarefa simples, uma vez que o compilador da própria linguagem de implementação auxilia no processo, e não está automatizada ainda por limitação de tempo durante a implementação do suporte ferramental.

Capítulo 4

Avaliação

No processo de avaliação do MetaTT procuramos investigar a eficácia da abordagem sob o aspecto de melhoria de qualidade (modularidade e legibilidade) dos geradores de sintaxe concreta produzidos e sobre a independência do método com relação a diferentes domínios metamodelados. Para avaliar estes aspectos, seguimos por dois caminhos: *(i)* comparamos os geradores M2T construídos com (e sem) nossa abordagem para os mesmos metamodelos e *(ii)* implementamos geradores de código para metamodelos de diferentes domínios.

4.1 Qualidade dos Geradores M2T

Com a finalidade de avaliar o resultado da aplicação do MetaTT em termos de qualidade dos geradores M2T produzidos definimos os cenários a serem analisados e as métricas que auxiliarão na análise.

4.1.1 Caracterização dos Cenários

Os cenários em questão são caracterizados por duas dimensões, o metamodelo para o qual o gerador M2T é produzido e o método utilizado na produção deste. A ordem cronológica de execução dos cenários é 1, 3, 2, 4 e os cenários 1 e 3 foram realizados sem o conhecimento prévio do da existência do MetaTT, de fato, eles são anteriores à existência do MetaTT.

Metamodelos

Os metamodelos escolhidos para compor os cenários de análise foram os metamodelos da linguagem Java e de Redes de Petri. Eles foram escolhidos porque foram usados no contexto do laboratório de pesquisa em que este trabalho foi realizado, e, assim, podemos analisar o uso deles face à aplicação real dos mesmos em outros trabalhos. Por exemplo, o metamodelo de Java e respectivo gerador de código foram utilizados no contexto dos trabalhos [41] e [33] e o metamodelo e gerador de código de Redes de Petri foram utilizados no contexto do trabalho [6]. Outro aspecto bastante favorável à escolha de tais metamodelos é que eles são bastante diferentes entre si quanto ao tamanho e complexidade estrutural, desta forma podemos enquadrar o metamodelo de Redes de Petri como exemplo de metamodelos pequenos e simples e o metamodelo de Java como exemplo de metamodelos grandes e complexos.

Métodos

Com respeito aos métodos usados para a elaboração dos geradores M2T, pode-se usar o MetaTT ou um método *ad-hoc*. Observamos que, quando nenhum método prévio é fornecido para o desenvolvimento e codificação das transformações, identificamos um comportamento comum: o desenvolvedor começa a produzir transformações para os elementos raízes do metamodelo e adiciona regras de transformações iterativamente conforme os conceitos vão sendo requeridos nos novos elementos a serem transformados. Na falta de um método alternativo, a caracterização deste comportamento nos permite uma comparação entre o uso do MetaTT e uma abordagem *ad-hoc* observada.

Sujeitos

Os sujeitos que realizaram a avaliação foram dois estudantes do SPLab/UFCG: um graduado que realizou os cenários 1 e 2 e um graduando que realizou os cenários 3 e 4, ambos com conhecimento prévio sobre MDD/MDA e com experiência em transformações M2T, mais especificamente MOFScript. Eles receberam instruçõs sobre o MetaTT antes de usá-lo.

Cenários

Com base nos métodos e metamodelos elencados para uso nesta avaliação, obtivemos quatro cenários para análise, descritos na Tab. 4.1.

Cenário	Método	Metamodelo
1	Ad-hoc	Java
2	MetaTT	Java
3	Ad-hoc	PetriNet
4	MetaTT	PetriNet

Tabela 4.1: Os possíveis cenários que combinam os dois metamodelos e o uso do MetaTT ou de uma estratégia *ad-hoc*.

4.1.2 Métricas de avaliação

Para examinar as características dos geradores M2T obtidos em cada cenário e analisar os resultados, usamos o seguinte conjunto de métricas como apoio:

Número de Metaclasses (NM). Foi escolhida para indicar o tamanho do metamodelo e pode ser usada para comparar a quantidade de esforço requerido para construir geradores M2T para um determinado metamodelo;

Média do Número de Referências por Metaclasse (MNRM). Indica o nível de dependência entre metaclasses. Por exemplo, se uma metaclasse tem três referências para outra(s) metaclasse(s), tal dependência deve ser refletida na estrutura do gerador M2T porque a transformação da primeira metaclasse depende da transformação das últimas. MNRM captura o número médio de dependências conceituais que devem ser tratadas no gerador M2T. Dado um metamodelo e o seu conjunto de metaclasses $M = \{m_1, m_2, ..., m_i, ..., m_{n-1}, m_n\}$, definimos MNRM como:

$$MNRM = \frac{\sum_{i=1}^{n} ref(m_i)}{n} \tag{4.1}$$

Na fórmula 4.1, $ref(m_i)$ é uma função que informa o número de referências da i-ésima metaclasse de um metamodelo.

Média da Profundidade das Árvores de Herança (MPAH). A profundidade de árvore de herança (*DIT - Depth of inheritance tree*) é uma métrica amplamente utilizada no universo OO e está correlacionada a aspectos de qualidade de software, tais como reusabilidade e manutenabilidade. Ela provê uma medida de níveis de herança da classe mais alta na árvore de herança para uma classe particular [10]. Por exemplo, no diagrama de classes que tem uma classe C como a mais alta, todas as classes que são descendentes diretas de C têm um valor de *DIT* igual a 1; se uma classe adicional é descendente direta de uma das descendentes de C, então esta classe apresenta valor de *DIT* igual a 2. Em PLang, o elemento *ConditionalSttmnt* (ver Fig. 2.5) tem valor de *DIT* porque é descendente de *Statement* e, logo após, de *ASTNode*. Com o intuito de apresentar esta informação para o metamodelo como um todo, usamos ***MPAH**, a média dos valores da medida de DIT para cada metaclasse do metamodelo.* Segundo [34] (*apud* [25]), quanto maior o valor da métrica DIT, maior será a chance de reúso. Entretanto, um alto valor desta métrica também pode causar problemas em relação a compreensão dos programas. Dado um metamodelo e o seu conjunto de metaclasses $M = \{m_1, m_2, ..., m_i, ..., m_{n-1}, m_n\}$, definimos MPAH como:

$$MPAH = \frac{\sum_{i=1}^{n} DIT(m_i)}{n} \tag{4.2}$$

Na fórmula 4.2, $DIT(m_i)$ é uma função que informa o valor da métrica *DIT* da *i*-ésima metaclasse de um metamodelo.

Complexidade Agregada (CA). Combinamos as métricas NM, $MNRM$ e $MPAH$ a fim de analisar a complexidade agregada pelas características de tamanho e complexidade estrutural já consideradas. Dado um metamodelo A, a complexidade agregada do mesmo é definida como:

$$CA_A = NM_A \times (MNRM_A + MPAH_A) \tag{4.3}$$

Na fórmula 4.3, as métricas de complexidade estrutural ($MNRM$ e $MPAH$) são somadas e multiplicadas por NM, que denota o tamanho do metamodelo. O objetivo do uso das métricas de tamanho e complexidade conjuntamente na última coluna é mensurar a complexidade dos metamodelos e extrair daí um indicativo da quantidade de esforço a ser empregada na construção de um gerador de código.

Na Tab. 4.2, a coluna NM apresenta os valores indicativos do tamanho dos metamodelos e as colunas MNRM e MPAH apresentam os valores indicativos da complexidade média dos elementos dos mesmos. A coluna *CA* apresenta valores indicativos da complexidade total de cada metamodelo, ou agregada.

Metamodelo	NM	MNRM	MPAH	CA
Redes de Petri	7	1.57	0.71	**15.96**
Java	99	1.62	1.82	**340.56**

Tabela 4.2: Métricas descritivas dos metamodelos.

A partir dos valores da coluna *CA*, fica claro que os metamodelos apresentam complexidades diferentes e que o metamodelo de Java é significativamente mais complexo que o de Redes de Petri.

4.1.3 Análise dos cenários

Após definidos os métodos e justificadas as escolhas dos metamodelos, pode-se fazer uma análise dos cenários traçados (e resumidos na Tab. 4.1) face às métricas apresentadas.

Cenário 1. *Construção de um gerador de código para Java usando uma abordagem ad-hoc.*
A construção do gerador foi intrinsecamente difícil porque o desenvolvedor precisou, primeiramente, compreender os detalhes do metamodelo. Observamos que o código das transformações tornou-se difícil de ser compreendido e mantido à medida que novos conceitos do metamodelo passavam a ser transformados. As razões para tal dificuldade estão associadas à elevada complexidade do metamodelo, evidente pela taxa de dependência entre metaclasses bem como pela profundidade das árvores de herança. A dependência entre metaclasses levou a muitas ocorrências de transformações aninhadas dentro das regras, ou seja, os conceitos dos elementos relacionados foram sendo resolvidos dentro da mesma regra, levando à repetição de código (porque os mesmos conceitos também eram necessários em algumas outras regras de transformação e acabaram tendo sua transformação definida novamente nestas regras) e situações de inconsistência (porque não havia garantia de que o mesmo tipo de meta-elemento seria transformado da mesma maneira no código repetido, o que é, geralmente, desejável).

Evitar repetição de código por meio de práticas de modularização comuns (*p.e.*, a definição de uma regra adequada para cada metaclasse e a invocão desta sempre que necessário) não era algo que pudesse ser feito de maneira trivial no início do projeto porque a dependência conceitual entre metaclasses era muito complexa (*p.e.*, os elementos relacionados também tinha outros elementos relacionados, que por sua vez tinha outros elementos relacionados e assim por diante). A complexidade decorrente da estrutura hierárquica do metamodelo (representada pela métrica MPAH) levou ao uso de muitos trechos de código para verificar o tipo correto dos meta-elementos a fim de invocar suas respectivas regras corretamente. Esta adição de código nas regras dificulta a sua legibilidade. Uma boa abordagem para evitar a adição deste código para a verificação de tipo explícito seria o uso extensivo de polimorfismo. Para tal, o desenvolvedor precisaria identificar as relações de herança no metamodelo, definir regras de transformação abstratas para cada metaclasse ascendente e substituir essas regras para metaclasses descendentes, permitindo que o mecanismo polimórfico funcionasse corretamente. Por exemplo, no metamodelo Java tínhamos uma metaclasse *Expression* com suas 26 metaclasses descendentes. Para usar polimorfismo corretamente neste cenário foi preciso definir uma regra de transformação abstrata para a metaclasse *Expression*, *i.e.*, *getExpressionCode()*, e sobrescrevê-la para cada uma das 26 metaclasses.

Cenário 2. *Construção de um gerador de código para Java usando a abordagem MetaTT.* Para construir o gerador M2T, o desenvolvedor primeiro precisou entender os conceitos da abordagem MetaTT. Em seguida, o suporte ferramental pode ser usado. Uma vez que o desenvolvedor entendeu a estrutura das regras e preencheu iterativamente os esqueletos de regras presentes no sub-módulo *TextualDefinitions*, o processo de construção tornou-se mais fácil. O entendimento do gerador M2T está ligado à compreensão da abordagem MetaTT. A manutenção do gerador M2T foi mais fácil do que quando adota-se uma abordagem *ad-hoc*, uma vez que seguiu um projeto modular e foi automaticamente documentado através de nossa abordagem. O balanço entre o esforço de aprendizagem dos conceitos de MetaTT e a consequente melhoria na compreensão e manutenção do gerador de código demostrou-se bastante proveitoso.

O uso de MetaTT também levou a uma melhoria significativa na qualidade do código das regras. Constatamos que as regras de transformação ficaram mais simples e modulares porque diminuímos a redundância de código e melhoramos sua organização uma vez que evitamos transformações aninhadas (dois metalementos sendo transformados em uma mesma regra) e também simplificamos a lógica do código de cada regra. Em números, *(i)* a quantidade de metaclasses transformadas por regra diminuiu 44 %, o que indica uma diminuição na quantidade de repetição de código e melhoria da modularidade das regras e *(ii)* o número de instruções por regra diminuiu 39 %, o que indica que o código das transformações tornou-se mais simples de ser lido e entendido [32]. Além disso, o gerador de código Java produzido é utilizado como parte integrante de duas ferramentas de projetos com base em MDA e relacionados com as áreas de teste [33] e conformidade arquitetural [41].

Cenário 3. *Construção de um gerador para Redes de Petri seguindo uma abordagem ad-hoc.* Uma vez que o metamodelo era pequeno (7 metaclasses), juntamente com os conceitos mais simples (menor número de referências por metaclasse, 1,57, indicando menor dependência conceitual), foi fácil começar a construção do gerador M2T. Codificar as regras de transformação iterativamente funcionou bem neste caso. A baixa complexidade do problema tornou um design modular e bem estruturado um recurso não essencial. A manutenção das regras de transformação também não foi seriamente prejudicada. Entender o código também foi simples devido à baixa complexidade do artefato. Esta foi uma implementação utilizada no projeto MDAVeritas [6].

Cenário 4. *Construção de um gerador para Redes de Petri seguindo a abordagem MetaTT.* Para construir o gerador de código, houve a necessidade de, primeiramente, compreender os conceitos da abordagem e de compreender os conceitos do metamodelo. Devido à simplicidade do metamodelo a ser transformado, a sobrecarga de adicionar o entendimento da abordagem foi negativa por trazer benefícios que não impactavam em reais necessidades de melhoria de modularização e entendimento. A compreensão do gerador de código ficou sujeita à compreensão dos aspectos da abordagem (estrutura das regras, arquitetura e fluxo de execução). Embora a abordagem esteja documentada, o esforço necessário para compreendê-la e, efetivamente, empregá-la não demonstrou

ganhos significativos uma vez que o metamodelo era muito simples. A manutenção se tornou uma tarefa simples com a abordagem, mas não o suficiente para justificar seu uso. Esta versão do gerador M2T está disponível em [30].

A análise dos cenários descritos revela que a utilidade de MetaTT torna-se bastante evidente diante da tarefa de gerenciar a complexidade da escrita de geradores M2T para metamodelos grandes e complexos (elevado número de metaclasses e relações). Por outro lado, observou-se que para metamodelos bastante simples (baixo número de metaclasses e relações) uma abordagem *ad-hoc* pode ser suficiente e a adoção da abordagem MetaTT pode representar uma adição de complexidade desnecessária para a tarefa em questão. A adoção de MetaTT para projetos de geradores M2T com metamodelos pequenos torna-se uma opção a ser feita pelo desenvolvedor, caso este julgue pertinente seguir uma implementação padronizada. O MetaTT também pode ser mais interessante em casos onde o metamodelo é pequeno, mas muda e evolui frequentemente, porque torna-se mais fácil localizar onde as mudanças devem ser refletidas nas transformações. Por exemplo, se uma um atributo a de uma metaclasse M é alterado, precisaríamos apenas checar as regras específicas da metaclasse M dentro de cada módulo e realizar as devidas alterações (uma regra em cada módulo). Entretanto, se as transformações são feitas de maneira *ad-hoc* pode ser preciso checar todos os acessos ao atributo a da metaclasse M (que podem estar espalhados no código) e, então, verificar o efeito das alterações no comportamento das transformações.

De modo geral, MetaTT facilita a compreensão dos geradores M2T devido ao fato de que é um método documentado e as suas regras seguem padrões bem definidos e também facilita o processo de manutenção uma vez que os elementos estão melhor organizados.

Estudos adicionais são necessários a fim de definir limites relativos às características do metamodelo e à eficiência da nossa abordagem. Por exemplo, ainda não é possível determinar a partir de quais valores de métricas pode-se considerar o uso MetaTT como sendo mais ou menos efetivo. O que fica evidente é que os metamodelos utilizados permitem a análise de cenários bastante contrastantes, o que fica evidente quando observamos a métrica de *complexidade agregada* para ambos.

4.2 Independência de Domínio

A fim de avaliar a aplicabilidade da abordagem em vários domínios, implementamos protótipos de geradores M2T com metamodelos de vários domínios diferentes. Na Tab. 4.3, demonstra-se os metamodelos, o tamanho dos metamodelos representado pelo número de metaclasses para cada um deles e do tamanho dos geradores M2T medidos pelo número de linhas de código (LOC). Estes protótipos indicam que o método pode ser aplicado a diferentes domínios, tais como os experimentados: linguagens sistemas de controle (Redes de Petri), linguagens de programação (C# e Java), linguagens de banco de dados (SQLDDL e SQLDML) e linguagens de documentação (LaTeX). Como o MetaTT utiliza apenas características estruturais (não semânticas) dos metamodelos para a geração de artefatos é possível aplicar a técnica independentemente do domínio que esteja sendo transformado.

Metamodelo	NM	LoC do gerador
PetriNet	7	226
C#	10	292
SQLDDL	16	627
LaTeX	30	1009
SQLDML	30	1103
JavaAbstractSyntax	99	2718

Tabela 4.3: Dados coletados sobre os geradores M2T construídos usando-se a abordagem MetaTT.

O esforço empregado para a construção de transformações M2T, utilizando-se o MetaTT, deve ser independente do domínio do metamodelo (p.e., o paradigma da linguagem de programação abordada) porque o MetaTT baseia-se nas características estruturais e sintáticas dos metamodelos e não em características específicas ou semânticas. Para aplicar MetaTT para linguagens de outros paradigmas, diferentes do imperativo, o desenvolvedor precisa seguir o processo descrito na introdução do Cap. 3 e ilustrado na Fig. 3.1. Por exemplo, para construir transformações para um metamodelo de Prolog, o desenvolvedor necessitaria prover o metamodelo como entrada para a ferramenta do MetaTT e obter a versão inicial das transformações. Elas compreenderiam as regras prontas do módulo *Core* (tanto as do

submódulo *Extractor* quando do *Collections*) e os esqueletos das regras do módulo *Templates*. Dentre as regras geradas, ele precisaria completar as regras de template, adicionando os detalhes da sintaxe a ser gerada para cada elemento, e ajustar a regra principal do módulo *Main* para que filtre os elementos a serem transformados. Caso as transformações geradas no módulo *Core* precisem de ajustes, estes devem ser mínimos.

Os geradores de código da Tab. 4.3 estão disponíveis no diretório `evaluation/sample` do repositório de código do projeto MetaTT em [30]. Os geradores de C#, LaTeX, SQLDDL e SQLDML não foram considerados elegíveis para gerar novos cenários, a exemplo dos da subseção 4.1.3, porque nós não temos implementações *ad-hoc* de transformações M2T disponíveis para fazer a comparação.

4.3 Ameaças à validade

Identificamos as seguintes ameaças com relação às conclusões da nossa avaliação:

- O baixo número de amostras/cenários analisados em virtude do baixo número de metamodelos e participantes na nossa avaliação nos impede de fazer uma análise mais cuidadosa em que conclusões mais fortes possam ser feitas, inclusive, com aparato estatístico;

- A influência no grau de perícia dos participantes não foi avaliada porque não realizamos o processo com participantes com pouca ou nenhuma experiência no campo de transformações M2T;

- As métricas utilizadas para denotar tamanho e complexidade são ameaçadas pela falta de uma validação rigorosa delas enquanto métricas representativas desses aspectos. No entanto, essa ameaça é suavizada por: *(i)* o fato de que elas já são utilizados para avaliar diagramas de classe UML (como em [41]) e *(ii)* a correspondência entre os conceitos (dependência, composição, herança, etc) de diagramas de classe UML e as estruturas dos metamodelos.

- A representatividade dos metamodelos analisados como sendo instâncias "pequena" ou "grande". Apesar do fato de que eles são comparativamente diferentes em tamanho,

não podemos afirmar fortemente que os metamodelos de Redes de Petri e Java são exemplos estatisticamente representativos de um metamodelo pequeno e um grande, respectivamente. Para afirmar isso com uma maior segurança, seria necessária uma análise mais profunda sobre as características de um conjunto maior de metamodelos.

- Em relação à afirmação de que o MetaTT pode ser usado independentemente do domínio, a construção dos geradores adicionais como prova de conceito nos dá forte evidência de que ela possa estar correta. Entretanto, o estudo com um maior número de amostras, realização de atividades e análises seria necessário para dar suporte a uma análise mais profunda.

Capítulo 5

Trabalhos Relacionados

Os trabalhos relacionados abordam, em vários níveis, a geração de sintaxe concreta, mas, a maioria deles (1) usa uma abordagem de geração de código com foco em transformações de modelos pra texto, mas não é projetada para ser independente de domínio ([16] e [20]) ou (2) está focada na geração de artefatos de IDEs e na especificação de sintaxe através de novos modelos ou gramáticas ([36], [24], [11], [19] e [26]), o que foge do nosso escopo. Além disso, *nossa abordagem é a única que se baseia em uma arquitetura proposta com módulos bem definidos, responsabilidades e contratos.* Esta arquitetura tem como objetivo alcançar um maior nível de reutilização e manutenção nas transformações.

[16] apresenta uma abordagem para a geração de código Java a partir de modelos UML dentro da ferramenta FUJABA. Para isso, os modelos UML são processados e modelos de *tokens* intermediários são criados, otimizados e processados. A partir desses modelos, aplicam-se templates para gerar a sintaxe concreta. Apesar de a solução ser projetada para geração de código Java, os autores deixam claro que se pode gerar código para linguagens de mesmo paradigma utilizando-se novos templates com os mesmos modelo de *tokens*. Enquanto [16] apresenta soluções para o framework FUJABA, o nosso trabalho tem foco na escrita de transformações no paradigma de transformações M2T e não depende de uma ferramenta específica, mesmo que nossas implementações tenham sido feitas no ambiente de desenvolvimento Eclipse. Enquanto a abordagem de [16] está focada na transformação de modelos UML independentes de plataforma, o nosso trabalho não impõe restrições sobre o tipo dos modelos a serem transformados, e as características de seus metamodelos. O MetaTT pode ser utilizado para produzir transformações que atuem em modelos independentes de plataforma,

como UML, bem como em modelos de mais baixo nível, como modelos de linguagens de programação e de bancos de dados (Java, SQL, etc.).

[20] apresenta um estudo de caso de geração de código a partir de DSLs, refinando modelos gradualmente através de regras de reescrita e estratégias que integram transformações de modelo para modelo, de modelo para código e de código para código, diferentemente do nosso trabalho, que tem foco nas transformações de modelo para código (ou qualquer outro artefato textual). Também são discutidos como os princípios de modularização e extensibilidade foram aplicados na concepção da WebDSL (uma DSL para a elaboração de aplicações web dinâmicas), já no nosso trabalho nós discutimos aspectos de modularidade presentes nas decisões de projeto que adotamos e que são refletidas na arquitetura proposta do MetaTT. O trabalho de [20] aborda todo o conjunto de transformações (de níveis de abstração mais altos para níveis mais baixos), englobando várias partes de uma cadeia de transformação, enquanto que nós realizamos uma investigação aprofundada sobre o projeto de transformações de modelo-para-texto que pode ser aplicado em contextos diferentes ou incorporado à uma cadeia de transformações já existente. Ao contrário do nosso trabalho, [20] não propõe uma arquitetura para a escrita de geradores M2T e não discute aspectos relativos ao projeto de transformações M2T com foco na qualidade do código das transformações. Também há uma diferença na abordagem MDD seguida uma vez que [20] foca em DSLs e transformações com regras de reescritas na linguagem Stratego [50], já o nosso trabalho tem foco no arcabouço MDA com transformações do paradigma M2T, escritas em MOFScript [37].

[36] apresenta uma técnica para que seja possível mapear sintaxe abstrata e sintaxe concrete, ou seja, mapear modelos e texto, no contexto do projeto Kermeta [12]. Para tal, um metamodelo para representação de sintaxe concreta é proposto, com o qual pode-se instanciar modelos de sintaxe concreta. Nesta abordagem, pode-se utilizar os modelos de sintaxe concreta para gerar o seu texto correspondente ou pode-se utilizar um artefato textual para, a partir dele e das informações do metamodelo, instanciar um modelo de sintaxe concreta correspondente. Consequentemente, a abordagem em [36] permite mapeamentos bidirecionais de sintaxe (sintaxe abstrata $\leftrightarrow$ sintaxe concreta). A principal diferença com relação à nossa abordagem é que enquanto eles fazem mapeamentos através da descrição da sintaxe concreta através da elaboração de novos modelos, a nossa abordagem concentra-se

em atividades de escrita de transformações. Por consequência do foco em transformações, a nossa abordagem oferece o mapeamento de sintaxe abstrata (modelos) para sintaxe concreta (texto), mas não o mapeamento no sentido inverso.

TCS [24], XText [11] e EMFText [19] são ferramentas que suportam a definição de DSLs. Algumas delas incluem a geração de plugins, analisadores, editores de código, etc. Essas ferramentas também suportam mapeamentos declarativos entre modelos abstratos e sintaxe concreta por meio de linguagens de especificação ao estilo de gramáticas. Tais mapeamentos permitem a instanciação automática de modelos a partir da sintaxe concreta, bem como a geração automática de sintaxe concreta a partir dos modelos. Diferentemente delas, MetaTT não propõe uma nova ferramenta para a especificação de DSLs sintaxe ou uma linguagem M2T nova, mas propõe uma arquitetura para projetar e implementar transformações M2T para uma fácil manutenção e evolução.

[26] está inserido no contexto da criação, uso e manutenção de linguagens. Geralmente, DSLs descritas por meio de um modelo de sintaxe abstrata (ASM, ou um metamodelo). O principal problema tratado é o da falta de suporte ferramental para dar apoio às atividades de projeto e uso de novas DSLs. Para resolver tal problema, [26] propõe uma técnica para gerar alguns artefatos de IDE (como parsers, gramáticas e editores) para uma linguagem a partir do seu modelo de sintaxe abstrata. Esta técnica oferece uma grande ajuda no desenvolvimento e uso de DSLs. Para tal, os autores prescrevem o uso de metamodelos, mas com algumas restrições, como, p.e., o fato de que suas respectivas instâncias devam seguir uma estrutura de árvore, para que se possa fazer o mapeamento dos modelos abstratos com árvores sintáticas. A primeira diferença que se pode observar em relação ao MetaTT é que eles possuem focos diferentes. Enquanto o nosso trabalho procura atacar aspectos do projeto de transformações, o trabalho de [26] é focado na geração completa de artefatos para IDEs. Outra diferença está no fato de que nós não impomos restrições sobre a estrutura do metamodelo para o qual estamos gerando texto, p.e., nós não exigimos a ausência de ciclos no metamodelo. Por outro lado, nós eliminamos estes ciclos quando obtemos o modelo de referência, conforme especificado na Sec. 3.2.1.

Em todas as abordagens acima, propõe-se novas representações para especificar a sintaxe concreta, seja por meio de uma gramática, um metamodelo ou uma DSL. Isto facilita não apenas o processo de geração de texto, mas também o processo de instanciação de modelos

a partir da sintaxe concreta. É importante destacar que estes trabalhos tratam problemas correlatos, mas diferentes. Uns atacam o problema de incorporação ou especificação de sintaxe concreta aos metamodelos, outros atacam o problema de desenvolvimento e suporte ferramental para DSLs. Diferentemente deles, o nosso trabalho ataca o problema de projeto de transformações de modelo para texto.

Capítulo 6

Considerações Finais

Neste trabalho de mestrado investigamos métodos para produzir geradores de código com transformações M2T focando na qualidade das transformações produzidas e, como resultado desta análise, propusemos o MetaTT, uma abordagem que consiste na (*i*) definição de uma arquitetura comum para transformadores M2T e (*ii*) em uma técnica operacional para guiar a elaboração dos aretefatos arquiteturais e a geração automática de parte das regras de transformação com base nas características estruturais dos metamodelos.

A fim de tornar mais prática a aplicação do MetaTT, nós construímos um conjunto de artefatos para auxiliar o desenvolvedor na geração dos elementos arquiteturais, incluindo parte das regras de transformação. Para avaliar a efetividade do trabalho nós aplicamos MetaTT em quatro cenários diferentes formados por dois metamodelos (Java e Redes de Petri) combinados com duas alternativas de implementação, o uso do MetaTT ou uma abordagem *ad-hoc* (comumente observada).

Como resultado, constatamos que a aplicação do MetaTT (*ii*) ajudou a diminuir o esforço empregado na escrita de geradores de código para metamodelos complexos devido ao fato de que os passos para a realização da tarefa estavam bem definidos e de que parte das regras de transformação foi gerada automaticamente; (ii) as regras de transformação foram simplificadas e a quantidade de código redundante foi diminuída uma vez que as preocupações a serem tratadas foram bem identificadas e separadas em módulos adequados; e (iii) constatamos que o MetaTT pode ser utilizado em diferentes domínios uma vez que a solução foi projeta para ser genérica.

Adicionalmente, durante o trabalho da dissertação, abordamos outras questões ligadas ao

projeto de transformação como forma de obter resultados auxiliares que pudessem contribuir com as linhas gerais deste trabalho. Por exemplo, antes de elaboramos a proposta de uma arquitetura para o projeto dos geradores de código, nós definimos alguns aspectos, em forma de diretrizes, que nos guiaram em direção à tal arquitetura [32]. Estas diretrizes podem ser utilizadas separadamente em geradores de código sem que se precise adotar toda a arquitetura e, mesmo assim, introduzir melhorias.

Além das diretrizes, nós investigamos métricas destinadas à análise de qualidade de código e não encontramos métricas dedicadas para transformações M2T, o que nos fez testar a adaptação de algumas métricas utilizadas para avaliação de programas do paradigma de Orientação a Objetos (OO), a exemplo de CBO e LCOM. Contudo, verificamos que estas métricas não se aplicavam ao nosso contexto, uma vez que elas utilizam características estruturais de código ligadas ao design do modelo OO e, no nosso contexto, isso requereria uma análise de qualidade muito mais ligada às características dos metamodelos do que às características das transformações em si. Desta maneira, escolhemos métricas mais simples, porém úteis, como o número de instruções e o número de metaelementos manipulados por regra. O número de instruções por regra (*NSR - Number of Instructions per Rule*) foi elaborado com base em trabalhos que apresentam o número de instruções de um método Java como um dos fatores de maior impacto na legibilidade do seu código, dentre outros [9]. Desta maneira, escolhemos este fator para uso em nossas regras de transformação, que são similares aos métodos Java e são escritos em linguagem imperativa também. Já o número de metaelementos manipulados (*NMT - Number of Metaelements Tackled*) foi uma métrica elaborada de acordo com a nossa observação de que quanto menos elementos são transformados em uma única regra, mais simples ela será. Apesar de parecer uma constatação bastante simples, é frequente encontrar vários elementos sendo transformados em uma única regra quando não há orientação contrária. Em [31] nós mostramos como melhorias foram implementadas em transformações M2T e como essas melhorias foram medidas com as métricas descritas. Também produzimos suporte ferramental para medição.

Apesar de bastante importantes, questões sobre a completude e/ou corretude do código gerado não são abordadas em nosso trabalho. Tais aspectos estão ligados a questões de metamodelagem e da semântica das transformações enquanto que o foco do nosso trabalho está na organização do projeto das transformações M2T.

Como limitação, verificamos que, apesar dos benefícios, o uso de MetaTT exige a aprendizagem de um novo método e que, nos casos em que o metamodelo é pouco complexo, uma abordagem *ad-hoc*, mesmo que sem qualquer padronização, pode ser adequada para a escrita das transformações sem que se tenha grandes prejuízos quanto à sua evolução futura.

6.1 Contribuições

Como principais contribuições deste trabalho, podemos elencar:

- **Exploração de aspectos do projeto de transformações de modelo para texto**. A partir do trabalho pudemos verificar formas de implementação das transformações e maneiras de tratar os problemas que surgiram.

- **Arquitetura de referência para elaboração de geradores de modelo para texto**. A arquitetura proposta é resultado do nosso esforço para melhoria constante das transformações e pode ser utilizada como ponto de partida para a proposição de outras arquiteturas diferentes e com tecnologias diferentes.

- **Técnica para geração dos artefatos arquiteturais**. A fim de facilitar a adoção da arquitetura elaborada, apresentamos um processo de automação da geração de seus artefatos a partir do uso de um modelo de referência criado a partir das informações estruturais do metamodelo. Este processo inclui um conjunto de passos explicativos sobre como obter o modelo de referência e um conjunto de passos sobre como obter os artefatos de transformação a partir do modelo de referência, ou seja, como obter as diferentes transformações, os relacionamentos de dependência entre elas e suas regras.

- **Suporte ferramental para a geração de transformações de modelo para texto**. A fim de automatizar o processo de geração das regras de transformação, produzimos um suporte ferramental que realiza o processo de automação proposto e, assim, deve auxiliar os desenvolvedores de transformação a obterem as primeiras versões de seus geradores de código de maneira prática.

6.2 Limitações e Trabalhos Futuros

Durante o desenvolvimento do nosso trabalho, surgiram algumas lacunas devido à restrição de escopo, tempo ou disponibilidade de recursos, que merecem ser consideradas como candidatas a trabalhos futuros em virtude da contribuição que podem trazer para a área e para a comunidade MDD/MDA. Estes pontos são:

As limitações são as seguintes:

- **Métricas para transformações M2T**. A área de MDD/MDA ainda carece de métricas efetivamente validadas para que se possa medir a qualidade do código das transformações M2T. Apesar de termos iniciado alguns estudos neste sentido, a elaboração de métricas para a avaliação de transformações requer estudos mais aprofundados com um número considerável de amostras e de avaliações de especialistas a fim de se mapear tais avaliações para as características das transformações, identificar-se as características que têm maior impacto e elaborar-se um modelo de avaliação que possa ser encapsulado e integrado a ambientes de desenvolvimento.

- **Avaliação mais robusta do impacto do MetaTT**. Sentimos a necessidade de avaliar com mais robustez os efeitos de MetaTT na produtividade de desenvolvedores MDD/MDA, entretanto, a indisponibilidade de recursos humanos em quantidade suficiente para que pudéssemos ter resultados estatisticamente válidos impossibilitou esta opção. Acreditamos que uma abordagem experimental com tais recursos pode trazer melhores avaliações sobre a efetividade do MetaTT. Caso a realização de experimentos adicionais demonstrem resultados a favor das ameaças levantadas na Secão 4.3, precisaremos rever as decisões de projeto tomadas e melhorar o MetaTT com base nos novos resultados.

- **Finalização do Suporte Ferramental**. Apesar de provermos um conjunto de artefatos que viabilizam a geração das transformações M2T e auxiliam no processo de adoção do MetaTT, estes artefatos ainda podem ser melhor integrados em forma de uma suíte de ferramentas a fim de que o seu uso se torne mais prático. Por exemplo, as transformações providas no suporte ferramental podem ser encapsuladas para funcionar independente do ambiente de desenvolvimento em que é executado hoje (IDE Eclipse) por

meio de linha de comando, ou podem ser integradas a uma IDE em forma de *plugin*.

Os trabalhos futuros são os seguintes:

- **Repositório de transformações M2T**. Assim como a comunidade tem disponibilizado repositórios de metamodelos [21] e de transformações M2M [5], acreditamos que, utilizando-se do método em MetaTT, pode-se criar, também, um repositório de transformações M2T. Desta maneira, a construção de cadeias de transformações seria facilitada ainda mais com a disponibilidade de geradores de código e os desenvolvedores MDD/MDA poderia se preocupar mais com as atividades de transformações entre modelos.

- **Rastreabilidade de mudanças**. O aspecto de rastreabilidade de mudanças nos geradores de código pode ser um tópico de estudos futuros. Quando geramos o código para as transformações, não criamos mecanismos para preservar mudanças em casos em que estas transformações forem re-geradas uma vez que este é um tópico além do escopo do nosso trabalho. Entretanto, há trabalhos que abordam este aspecto dentro do contexto MDD/MDA e acreditamos que uma investigação sobre estes trabalhos no contexto do MetaTT pode ser interessante para ajudar a evoluir o nosso trabalho.

- **Implementação da arquitetura com outras linguagens**. A implementação da arquitetura proposta com outras linguagens, como o MOF2Text, ou outras tecnologias, como o JET e o XPand, pode revelar pontos de melhoria na mesma, bem como verificar se as decisões tomadas se aplicam a todas as linguagems de transformção M2T ou se há particularidades de MOFScript na arquitetura que precisem ser reavaliadas. Como os conceitos trabalhados na elaboração da arquitetura são comuns ao paradigma de transformações M2T, é sensato dizer que pode-se aplicar a arquitetura a qualquer linguagem ou tecnologia que siga esta paradigma, entretanto uma reimplementação em uma destas tecnologias poderia prover um parecer mais preciso.

Bibliografia

[1] Accelo project plan. http://www.eclipse.org/projects/project-plan.php?projectid=modeling.m2t.acceleo, 2009.

[2] A. Agrawal, G. Karsai, and F. Shi. Graph transformations on domain-specific models. 2003.

[3] D. Akehurst and S. Kent. A relational approach to defining transformations in a metamodel. *«UML» 2002 The Unified Modeling Language*, pages 155–178, 2002.

[4] S.W. Ambler. *Agile modeling*. Wiley, 2002.

[5] Atl transformations - atl eclipse project page. http://www.eclipse.org/m2m/atl/atlTransformations/.

[6] P. Barbosa, J. Figueiredo, F. Ramalho, A. Dias Jr., A. Costa, A. Aranha, and F. Moutinho. Mda veritas, jan 2011.

[7] M. Bohlen. Andromda-from uml to deployable components, version 2.1. 2, 2003.

[8] S. Brodsky. Xmi opens application interchange. *IBM www-4. ibm. com/software/ad/standards/xmiwhite0399. pdf*, 1999.

[9] R.P.L. Buse and W.R. Weimer. Learning a metric for code readability. *Software Engineering, IEEE Transactions on*, 36(4):546–558, 2010.

[10] S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on*, 20(6):476–493, 2002.

[11] S. Efftinge and M. Völter. oaw xtext: A framework for textual dsls. In *Workshop on Modeling Symposium at Eclipse Summit*, volume 32, 2006.

[12] F. Fleurey, Z. Drey, D. Vojtisek, C. Faucher, and V. Mahé. Kermeta language, reference manual. *Internet: http://www. kermeta. org/docs/KerMeta-Manual. pdf. IRISA*, 2006.

[13] Eclipse Foundation. Xpand project. http://www.eclipse.org/modeling/m2t/?project=xpand. 2008.

[14] Eclipse Foundation. Eclipse Modeling: Java Emitter Templates. http://www. eclipse. org/emft/projects/jet, 2007.

[15] Eclipse Foundation. Acceleo project. http://www.eclipse.org/modeling/m2t/?project=acceleo, 2009.

[16] L. Geiger, C. Schneider, and C. Reckord. Template-and modelbased code generation for MDA-Tools. In *Proc. of the 3rd International Fujaba Days*, pages 57–62, 2005.

[17] J. Greenfield and K. Short. Software factories: assembling applications with patterns, models, frameworks and tools. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 16–27. ACM, 2003.

[18] Object Management Group. Mof models to text transformation language. http://www.omg.org/spec/MOFM2T/1.0/, 2008.

[19] F. Heidenreich, J. Johannes, S. Karol, M. Seifert, and C. Wende. Derivation and refinement of textual syntax for models. In *Model Driven Architecture-Foundations and Applications*, pages 114–129. Springer, 2009.

[20] Z. Hemel, L.C.L. Kats, D.M. Groenewegen, and E. Visser. Code generation by model transformation: a case study in transformation modularity. *Software and Systems Modeling*, pages 1–28, 2008.

[21] INRIA. Altanmod project home page. http://www.emn.fr/z-info/atlanmod/index.php/Zoos.

[22] I. Jacobson, G. Booch, J. Rumbaugh, et al. Unified process. *IEEE Software*, 16(3):96–102, 1999.

[23] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, and P. Valduriez. ATL: a QVT-like transformation language. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, page 720. ACM, 2006.

[24] F. Jouault, J. Bézivin, and I. Kurtev. TCS:: a DSL for the specification of textual concrete syntaxes in model engineering. In *Proceedings of the 5th international conference on Generative programming and component engineering*, page 254. ACM, 2006.

[25] H. Kim and C. Boldyreff. Developing software metrics applicable to uml models. In *6th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, 2002.

[26] A. Kleppe. Towards the generation of a text-based IDE from a language metamodel. In *Model Driven Architecture-Foundations and Applications*, pages 114–129. Springer, 2007.

[27] A.G. Kleppe, J. Warmer, and W. Bast. *MDA explained: the model driven architecture: practice and promise*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2003.

[28] D. Kolovos, R. Paige, and F. Polack. The epsilon transformation language. *Theory and Practice of Model Transformations*, pages 46–60, 2008.

[29] M. Lawley and J. Steel. Practical declarative model transformation with Tefkat. In *Satellite Events at the MoDELS 2005 Conference*, pages 139–150. Springer, 2005.

[30] Anderson Ledo, Franklin Ramalho, and Nat a Melo. The metatt project, jan 2010.

[31] Anderson Ledo, Franklin Ramalho, and Nat a Venâncio de Melo. Metatt - a metamodel based approach for writing textual transformations. In *CBSoft 2012 - SBCARS*, Natal - RN - Brazil, sep 2012.

[32] F Ledo, Anderson. Ramalho and N. Melo. Guidelines for Improving Model-to-Text Transformations. *CBSoft - Congresso Brasileiro de Software*, 2010.

[33] H.S. Lima, F. Ramalho, P.D.L. Machado, and E.L. Galdino. Automatic Generation of Platform Independent Built-in Contract Testers. 2007.

[34] L. Mark and K. Jeff. Object-oriented software metrics: A practical guide. ptr prentice hall.

[35] J. Miller, J. Mukerji, et al. MDA guide version 1.0. *OMG Document: omg/2003-05-01, http://www. omg. org/mda/mda_files/MDA_Guide_Version1-0. pdf*, 2003.

[36] P.A. Muller, F. Fleurey, F. Fondement, M. Hassenforder, R. Schneckenburger, S. Gérard, and J.M. Jézéquel. Model-driven analysis and synthesis of concrete syntax. *Model Driven Engineering Languages and Systems*, pages 98–110, 2006.

[37] J. Oldevik. MOFScript Eclipse Plug-In: Metamodel-Based Code Generation. In *Eclipse Technology Workshop (EtX) at ECOOP*, volume 2006, 2006.

[38] Object Management Group (OMG). Meta object facility specification. http://www.omg.org/technology/documents/formal/mof.htm, 2002.

[39] Object Management Group (OMG). Query views transformation language. http://www.omg.org/cgi-bin/doc?ptc/2005-11-01, 2005.

[40] Object Management Group (OMG). Meta object facility specification version 2.0. http://www.omg.org/spec/MOF/2.0/, 2006.

[41] W. Pires, J. Brunet, and F. Ramalho. UML-based design test generation. In *Proceedings of the 2008 ACM symposium on Applied computing*, pages 735–740. ACM, 2008.

[42] B. Selic. The pragmatics of model-driven development. *IEEE software*, 20(5):19–25, 2003.

[43] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro. *EMF: Eclipse Modeling Framework*. Addison-Wesley Professional, 2008.

[44] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2009.

[45] C.M. Stone. The Object Management Group Standardization of Object Technology. In *Advances in database technology–EDBT'94: 4th International Conference on Extending Database Technology, Cambridge, United Kingdom, March 28-31, 1994: proceedings*, page 1. Springer, 1994.

[46] D. Thomas and B.M. Barry. Model driven development: the case for domain oriented programming. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, page 7. ACM, 2003.

[47] Documents associated with unified modeling language (uml), v2.4.1. August 2011.

[48] D. Varró and A. Pataricza. Generic and meta-transformations for model transformation engineering. *UML 2004-The Unified Modelling Language*, pages 290–304, 2004.

[49] D. Varró, G. Varró, and A. Pataricza. Designing the automatic transformation of visual languages. *Science of Computer Programming*, 44(2):205–227, 2002.

[50] E. Visser. Stratego: A Language for Program Transformation Based on Rewriting Strategies System Description of Stratego 0.5. In *Rewriting Techniques and Applications*, pages 357–361. Springer, 1999.