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 Textuais

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

Resumo

Seu resumo aqui

Abstract

Abstract Here

Agradecimentos

Agradecimentos

Conteúdo

Lista de Símbolos

UE - *Um Exemplo*

OE - *Outro Exemplo*

Lista de Figuras

Lista de Tabelas

Lista de Códigos Fonte

Capítulo 1

Introdução

1.1 Seção 1 do Capítulo 1

1.1.1 Subseção

Subsubseção

A Figura **??**. A Tabela 1.1. A Equação (1.1). O trabalho de fulano [1]. O Código Fonte 1.1.

coluna 1	coluna 2	coluna 3
valor 1,1	valor 1,2	valor 1,3
valor 2,1	valor 2,2	valor 2,3

Tabela 1.1: Primeira tabela.

$$E = m \times c^2 \tag{1.1}$$

Código Fonte 1.1: Loop simples

```
for(int x=1; x<10; x++){
  cout << x << "\n";
}
```

1.2 Seção 2 do Capítulo 1

1.2.1 Subseção

Subsubseção

Capítulo 2

Fundamentação Teórica

Neste capítulo, apresenta-se 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) 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 MDD, 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 em texto.

Uma visão simplificada da abordagem DDM é mostrada na Fig. 2.1. Nela, a cadeia de transformações mostrada vai de um modelo de entrada com alto nível de abstração (*model 1*) até a sintaxe concreta (*concrete syntax*) resultante da execução de todas as transformações. O primeiro modelo (*model 1*) está em conformidade com seu metamodelo correspondente

(*metamodelo 1*) e é dado como entrada de uma ferramenta de transformação (*transformation tool*) e transformado em um segundo modelo (*model 2*), que, por sua vez, está em conformidade com um segundo metamodelo (*metamodel 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 (*metamodel 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, *i.e.*, a sua sintaxe concreta.

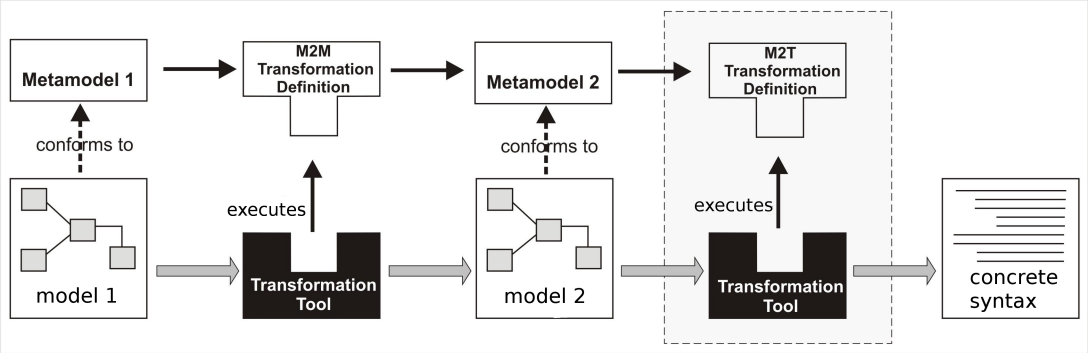

Figura 2.1: Uma visão simplificada da abordagem DDM.

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 [**?**] e QVT [**?**]. 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.

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 sucinta da sintaxe da lin-

guagem.

2.2.1 Histórico e Características

MOFScript é uma linguagem para a especificação de transformações de modelo para texto (M2T). 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) [**?**] da OMG. Apesar de suas particularidades, MOF2Text e MOFScript estão intimamente 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, 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 MDD 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 [**?**], algumas funcionalidades avançadas ainda não foram implementadas, como especificado em [**?**]. Isto significa que funcionalidades importantes, como extensão modular, sobescrição 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 as de OCL que permitem percorrer os objetos de um modelo. Por outro lado, ao contrário de XPand e Epsilon, 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 por conta do 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. 2.2 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 transformaçõo 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 escopoo global (a exemplo de `version` na linha 5) ou local (a exemplo de `typeDeclarations` na linha 7). Esta última é 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 um tipo do modelo ao qual a regra será aplicada. Uma regra também pode ter um tipo de retorno, que pode ser um tipo embutido de MOFScript (*e.g.*, *String*, *Real*, etc.) ou um tipo do modelo. Parâmetros também pode ser declarados. Na Fig. 2.2, 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, este usa uma variável implícita e reservada da linhagem 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. 2.2. A regra nomeada como `toFile()` nas linhas 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 signfica 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. Quando o elemento de contexto de uma regra *main* é um elemento de um metamodelo, a regra é executada para cada elemento do modelo de entrada que seja uma instância do mesmo tipo do contexto. Por exemplo, na Fig. 2.2 a regra *main* é executada para cada elemento *CompilationUnit* presente no modelo de entrada.

```
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 2.2: Um exemplo de transformação escrita em MOFScript.

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

2.3 Ecore

A OMG definiu o MOF como a metalinguagem padrão para especificar metamodelos. Com o mesmo objetivo, a comunidade Eclipse lançou o Ecore, subconjunto de MOF implementado dentro do *Eclipse Modeling Framework* (EMF). Ao longo do tempo, o projeto EMF também foi um contribuidor significativo para o *Essentials MOF* (EMOF) que é parte do padrão MOF2. Este processo levou a um alinhamento entre os conceitos de Ecore e EMOF. Uma vez que todo modelo MOF2 pode ser representado como um modelo EMOF, é resoável 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. Na Fig. 2.3, mostramos o metamodelo de PLang, representando apenas os 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*);

- 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

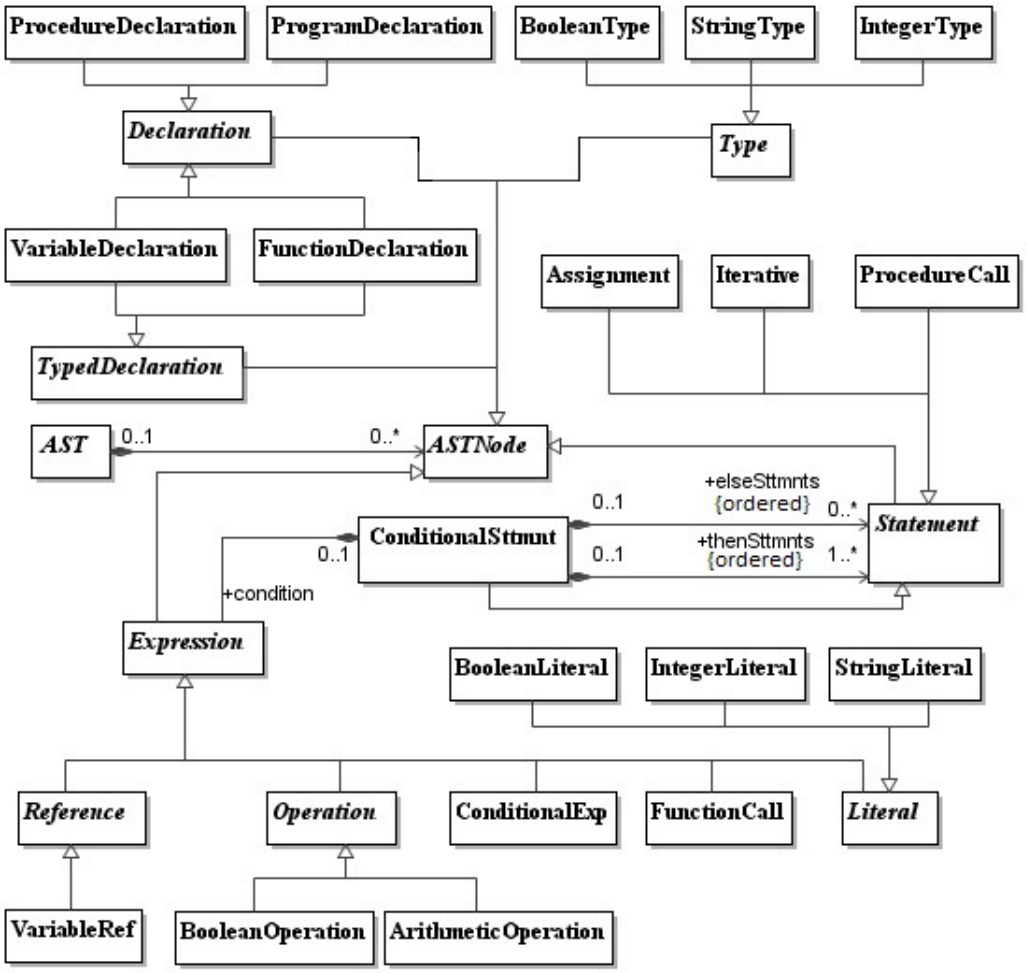

Figura 2.3: Metamodelo de PLang.

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 (*BooleanLiteral*) e aritméticas (*ArithmeticOperation*);
- Três tipos de literais podem aparecer em expressões de PLang: *StringLiteral*, *Boolean-*

Literal 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. 2.3. 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. 2.3 que não são essenciais para o entendimento do metamodelo, bem como o entendimento do trabalho como um todo.

Capítulo 3

MetaTT

Este trabalho propõe MetaTT, um abordagem com base em metamodelos para a escrita de geradores de código 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 arqutetura que deve ser seguida pelas transformações M2T geradas. Esta arquitetura é formada por módulos, sub-módulos e contratos entre esses artefatos que, juntos, objetivam alcancçar um auto nível de reúso de automação;

2. Uma abordagem operacional que indica como os artefatos propostos na arquitetura devem ser implementados, *i.e.*, que indica quais são as transformações e regras devem aparecer em cada módulo precrito em (1).

Na Fig. 3.1 ilustra-se como um desenvolvedor pode usar o MetaTT e como as transformações resultantes pode ser udadas para a geração da sintaxe concreta a partir dos modelos. Na regição 1, ilustra-se como os desenvolvedores precisam interagir com o suporte ferramental de MetaTT (explicado no Cap. **??**) a fim de produzir transformações M2T:

1. O desenvolvedor provê um metamode 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. O desenvolvedor complementa a implementação das regras que foram geradas como *stubs* pelo MetaTT no móudulo *Templates*;

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

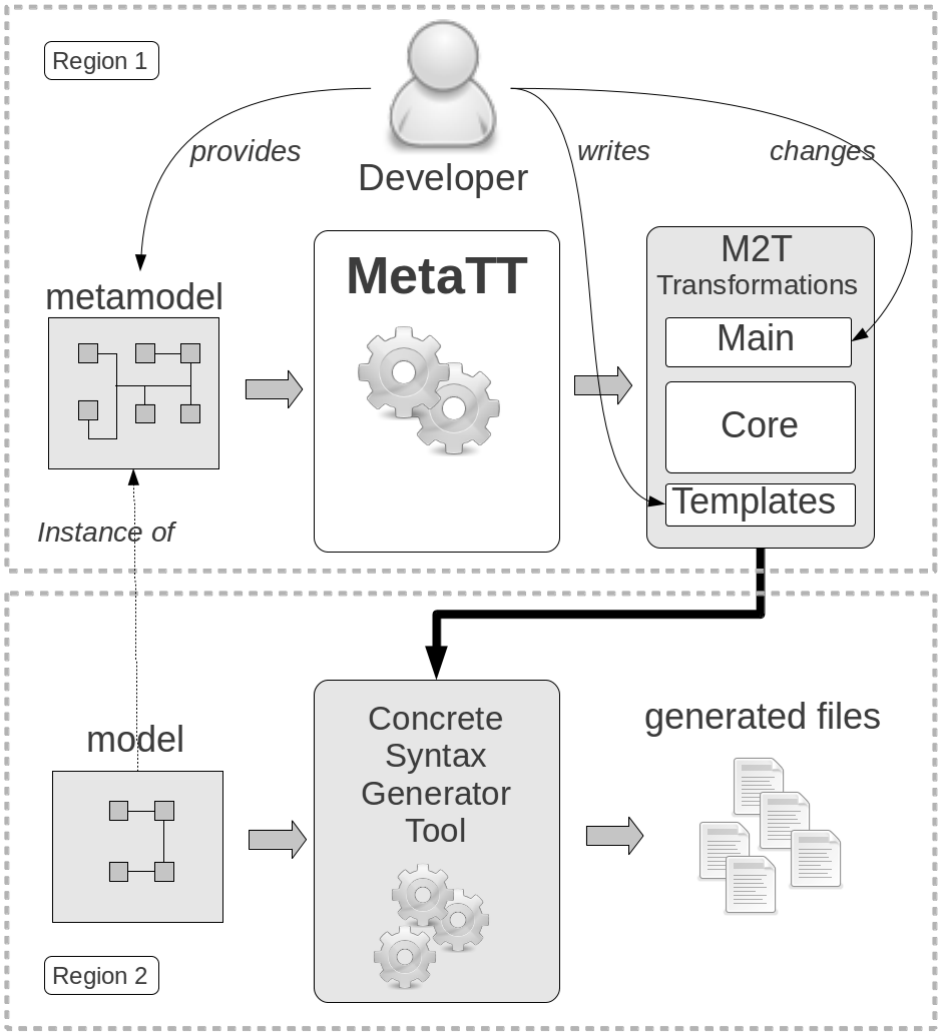

Figura 3.1: MetaTT usage overview.

Na região 2 (*region 2*) da Fig. 3.1, ilustra-se 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 apartir do MetaTT. O gerador executa as transformações 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 seções.

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 dos processos realizados 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.

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*

Este módulo provê a definição de sintaxe concreta para a linguagem alvo. Por exemplo, considerando a linguagem de programação Java como a linguagem alvo, este módulo conterá a especificação da sintaxe concreta para as assinaturas de métodos, declarações de tipos, as palavras chaves, etc.

Este módulo é formado por um conjutno de regras de templates (*template rules*) e tabelas de símbolos (*symbol tables*). As regras de templates provêem a definição da sintaxe para elementos não-terminais 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, *e.g.*, palavras chaves, caracteres separadores, delimitadores de bloco, etc.

Sempre que necessita-se de alguma informação referente à sintaxe concreta no processo

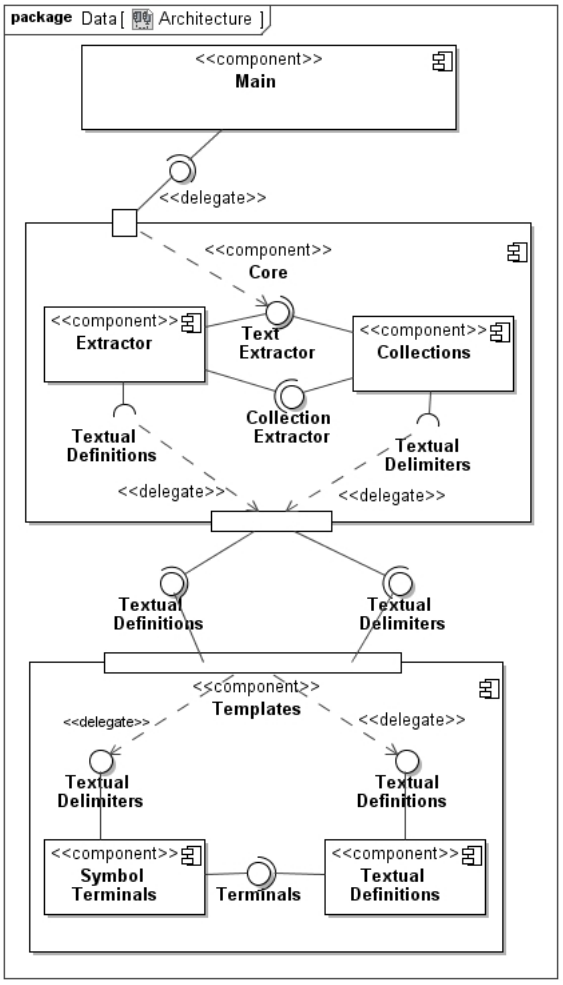

Figura 3.2: A arquitetura prescrita por nosso trabalho para geradores de código M2T.

de geração de texto, esta informação é obtida do módulo *Templates* através das interfaces *ITextDefinitions* e *ITextDelimiters*, que dão acesso aos submódulos *TextualDefinitions* e *TerminalSymbols*, respectivamente. O submódulo *TextualDefinitions* contém as regras de template (*template rules*), e cada uma delas define a um template específico para cada metaclasse do metamodelo. O submódulo *TerminalSymbols* contém as tabelas de símbolos (*simbol tables*) que contém as contantes *strings* correspondente aos terminais da linguagem (*e.g.*, 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. Um exemplo de uma tabela de símbolos.

Na Fig. 3.3 ilustra-se parte da tabela de símbolos provida para o caso do PLang, na qual cada símbolo é definido como uma constante *string* (definido como 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 (chaves esquerda e direita, parênteses esquerdo e direito). Tais caracteres podem ser reusados em várias regras de templates e outras definições (*e.g.*, nas linhas 9 e 11 e, mais à frente, na Fig. 3.5). Nas linhas 6 e 7, 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 ou ela precisa definir os valores do 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 ou ela define 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: An excerpt of a BNF grammar for the PLang ConditionalSttmnt with non-terminal elements in normal font and terminal elements in bold font.

```
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: Implementation of a template rule for the BNF grammar described in Fig. 3.4.

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 9-13, the bloco de comandos *else* é acrescentado à saída do programa apenas se o argumento *else* não for vazio.

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: Example of a code outputted from the *template rule* in 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 dedfinidas e localizadas. Como resultado, pode-se alterar a sintaxe concreta a ser gerada trocando-se os artefatos do módulo *Templates* por outros (respeitando-se os contratos das interfaces????), sem precisar mudar outros módulos.

Apesar de o exemplo apresentado estar relacionado ao PLang, que representa conceitos de linguagens de programação, nosa 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 metamodelo. 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 nosso templates, de maneira que o desenvolvedor fica livre pra mudar a sintaxe de acordo com o que o desenvolvedor achar conveniente. Por exemplo, se o desenvolvedor precisa de um gerador M2T para a linguagem de marcação HTML, ele ou ela 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 ou ela não depende de uma descrição em gramática, e a descrição da sintaxe pode seer feita pelo desenvolvedor com base em exemplos ou conhecimento prévio que ele ou ela já tem sobre a sintaxe de HTML.

3.1.2 Core

This module provides rules for carrying out the textual transformation of each metaclass of the metamodel. It is further divided into two sub-modules: *Extractor* and *Collections*.

Extractor sub-module is responsible for the extraction of the concrete syntax information of each metaclass of the metamodel, whereas *Collections* sub-module manages the extraction of the concrete syntax information for collections of instances of a given metaclass. Both sub-modules depend on each other since they are responsible for complementary functionalities. Whenever textual information for a collection of elements is required inside the *Extractor* sub-module, it accesses the functionality provided by the *Collections* sub-module through the *ICollectionExtractor* interface and whenever the transformation of a single element is required inside the *Collections* sub-module it accesses the functionality of the *Extractor* sub-module through the *ITextExtractor* interface.

The rules in *Extractor* sub-module are named *extractor rules*. For each metaclass of the metamodel there is an extractor rule responsible for invoking rules capable of extracting the concrete syntax pieces (one for each attribute or reference of that metaclass) that together will form the whole syntax to that metaclass. The task of combining these syntax pieces is the role of the template rules (through the *ITextualDefinitions* interface) from the *Templates* module. For instance, for the *ConditionalSttmnt* metaclass, illustrated in Fig. 3.7, there is an extractor rule, presented in Fig. 3.8. One can perceive the close relationship between the composition relationships of the *ConditionalSttmnt* and the structure of the code presented in Fig. 3.8. For each relationship from the *ConditionalSttmnt* metaclass targeting a *Statement* or an *Expression*, there is a rule invocation in the extractor rule, *i.e.*, there is a correspondence between the member ends *condition*, *thenSttmnts* and *elseSttmnts* from Fig. 3.7 and lines 2, 3 and 4, respectively, from Fig. 3.8. At line 5, a template rule, named *conditionalStatement-Template*, is invoked to combine the syntax pieces captured at lines 2-4 forming the whole concrete syntax of the *ConditionalSttmnt*.

At lines 3 and 4, in Fig. 3.8, some collection rules are invoked. Such rules are provided by the *Collections* sub-module. They are responsible for abstracting the processing of collections of elements and enclosing them with their corresponding textual delimiters, obtained from the *Templates* module through the required interface *ITextualDelimiters*. In this example, the collections are required since a collection of *then* and *else* statements may

be involved in a conditional statement. By separating these kinds of rules into a different module, MetaTT helps simplify the extractor rules.

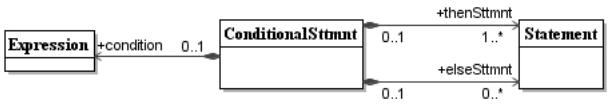

Figura 3.7: A ConditionalSttmnt and its related metaelements.

```
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: An extractor rule for a conditional statement in PLang.

3.1.3 Main.

This module comprises the start and the finish point of the model-to-text transformation. It contains an *entry point rule* that receives an input model, identifies the metaclass this model is rooted and invokes the corresponding rule from Core module that in turn returns the concrete syntax for the given element. Finally, the *entry point rule* performs the persistence of this syntax, finishing the process of model-to-text transformation.

Whenever one wants to generate text for a given metaelement he or she needs to write a main rule that invokes the corresponding rule (an extractor rule already specified in the *Core* module) for that metaelement. Fig. 3.9 and Fig. 3.10 are two slightly different examples of main rules. In both, the declaration of the context type of the rule (at line 1) allows an element to be automatically selected from the input model. At line 1, in Fig. 3.9, the transformation matches with *ProgramDeclaration* elements and invokes the *extractor rule* named *getProgramDeclarationCode()* (at line 2) from the *Core* module. Persistence is made at lines

3-4. Fig. 3.10 is analogous to Fig. 3.9, but it performs the transformation for *FunctionDeclaration* elements, *i.e.*, it is responsible for generating the concrete syntax for the function declarations in PLang.

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

Figura 3.9: A main rule that generates the code of every *ProgramDeclaration* element.

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

Figura 3.10: A main rule that generates the code of every *FunctionDeclaration* element.

A better control of the generation process is possible because the persistence is isolated in one module, *i.e. Main*, whereas the extraction of the text is isolated in another one, *i.e. Core*. The Main module is the only place in the whole text generator where there are print statements. This gives the transformer a better control over what should be printed to files.

3.1.4 Putting It All Together.

In this section we give an overview of the execution flow between the architectural modules.

In Fig. 3.11, we present the main activities performed through the modules which are represented by swimlanes. Initially, in the *Main* module, model elements are matched with the input model to be transformed. Then, the flow progresses to the activity of data extraction executed in the *Core* module (by the *Extractor* and *Collections* sub-modules) whose details are omitted here for the sake of simplicity. After the needed data is extracted, they are combined by the *Templates* module. The generated text is then persisted by the *Main* module and the process finishes.

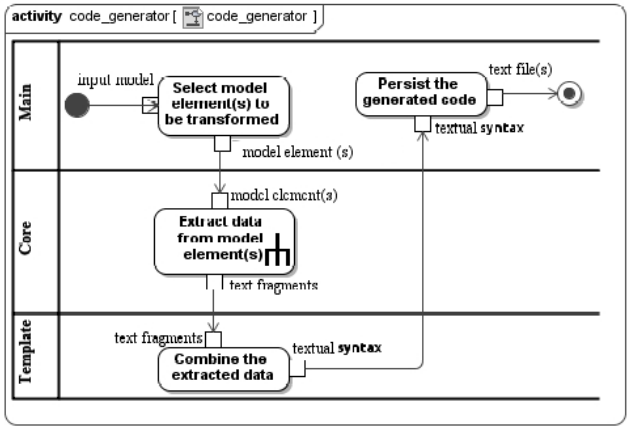

Figura 3.11: Overview of the activities performed in MetaTT.

In Fig. 3.12, we illustrate interactions occurring between the modules during the process of text generation to a conditional statement. In this context, the extractor rule *getConditionalStatementCode()* is invoked from the module *Main*. From this point, we emphasize the message exchanges among the transformations in different modules (that can be observed by the lifelines) for the rule presented in Fig. 3.8. The message callings progresses to the *getConditionalStatementCode()* and the *getExpressionCode()* extractor rules. Then, there are two callings (illustrated in the UML *ref* combined fragments) for the *getStatementCollectionCode()* rule (one for the set of statements to be executed if the condition evaluates to true and another one otherwise), both from *Collections* sub-module. Finally, the partial codes are combined into the *Templates* module and the result is returned back to the *Main* module to be persisted.

3.2 Generation of the Architectural Artifacts

After showing the prescribed modules and how they interact each other, we need to elaborate their internal artifacts, *i.e.*, the transformations and their relationships and rule signatures. Important questions that arise in this process are: How many and what transformations should be created for each module? What rules should each transformation comprise? How will the rules be arranged into the transformations? What relationships should exist between the transformations?

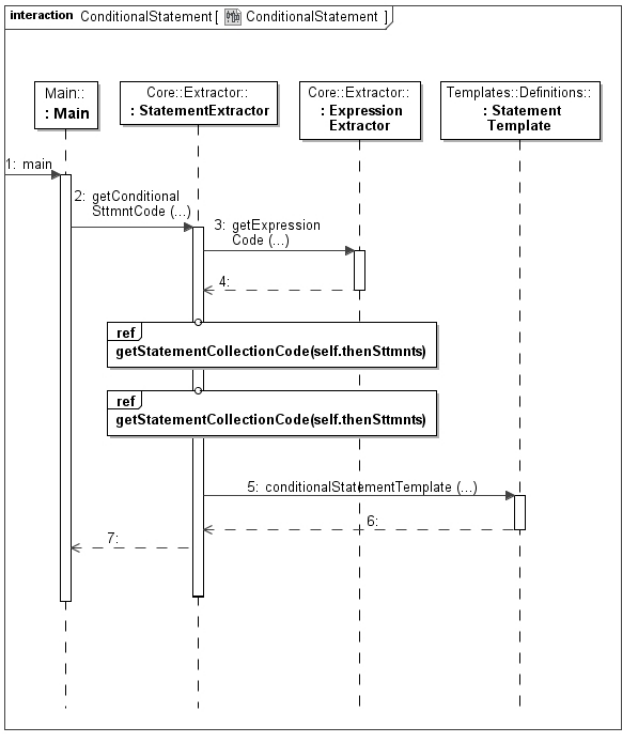

Figura 3.12: Sequence diagram illustrating the interactions among modules inside a PLang M2T generator.

In order to answer the design questions and to ease the process of elaboration of the artifacts we adopt a *reference model*, that contains the essential information for designing the M2T transformation artifacts. Such information is obtained from the metamodel to which the concrete syntax must be generated.

The elements of the *reference model* are used later to guide the derivation of artifacts, such as model-to-text transformations and transformation rules.

The next subsections describe our approach to obtain the *reference model* and how we elaborate the artifacts based on the information provided by such model.

3.2.1 From the Metamodel to the Reference Model.

From the metamodel, one must follow a stepwise process in order to obtain the reference model. This process synthesizes our approach for obtaining a reference for the implementation of the artifacts and the way they are comprised in the modules prescribed by the MetaTT

architecture. The steps are:

step 1 To select the information (from the metamodel) needed to elaborate the reference model. Below, the metamodel elements (on the left) originate the *reference elements* [1] (on the right):

- metaclasses → *reference transformations* (identified by the same name of the corresponding metaclasses);
- inheritance relationships → directed *reference associations* (following the direction child-to-parent in the metamodel).

From this point, the remaining information in the metamodel (compositions, attributes, etc.) are not used anymore, only the reference elements created (*i.e.*, *reference associations* and *reference transformations*).

step 2 To verify if some *reference transformation* in the resulting structure (from step 1) has two or more *reference associations* targeting any other *reference transformation* (it reveals whether there is multiple inheritance in the metamodel). If no, skip to step 3. If yes, then apply the normalization process: while there is two or more associations from a given *reference transformation*, take it and merge all the *reference transformations* it targets. Merging means that the targeted reference transformations will become only one and every reference association between the source reference transformation and the targeted ones becomes an unique reference association. This step guarantees that the structure of the reference model will have the form of a directed tree or a directed forest graph.

step 3 For every reference association which has a leaf *reference transformation* in one of the ends, this leaf *reference transformation* is turned into a *reference rule* contained in its parent reference transformation.

step 4 To reverse the directions of the reference associations. The targets become sources and vice-versa. Then, tag every reference association as an *import* relationship.

[1] From this point, every element will be a *reference element*.

The rationale for *step 1* is to simplify the reference model selecting only essential information from the metamodel: Metaclasses and inheritance relationships. In *step 2*, as we detect an element *e* with multiple inheritance, we decided to merge its parents because they will further give origin to *reference transformations* and then *e* will give origin to a *reference rule*; this rule could be added to only one of the transformations because we observed that duplicating the rule into more than one transformation could cause problems (in compilation and execution), then we observed that merging the parents and keeping only one *reference rule* relative to *e* inside the merged transformation is a better choice. In *step 3*, grouping the *reference rules* that are derived from metaelements which have a parent element in common is the way to keep good cohesion. In *step 4*, some import relationships are derived from the resulting associations because they reflect the fact that the more abstract elements need the definitions of the less abstract ones. For instance, the actual rules that will be further derived from the *reference rules* in the *Expression reference transformation* will need the definitions of the rules derived from the *reference rules* into *Operation*, *Reference* and *Literal*.

In Fig. 3.13, the result of the application of the step 1 over the PLang metamodel is illustrated. After that, metaclasses become reference transformations and inheritance relationships become directed reference associations. It is worth taking a look at Fig. 2.3 to note the difference from the metamodel and the result of applying the step 1.

During the application of the step 2, the presence of two reference associations from the same reference transformation to another ones is detected, *i.e.*, both reference associations from *FunctionDeclaration* and *VariableDeclaration* point to *Declaration* and *TypedDeclaration*. According to our approach, the targeted transformation references are merged into a single one, resulting into *Declaration_TypedDeclaration_Merged*, illustrated in Fig. 3.14.

Fig. 3.15 illustrates the result of applying the step 3. For instance, the leafs *FunctionCall* and *ConditionalExp* transformation references become rule references in the *Expression* reference transformation. Conversely, *Operation* is not turned into one rule reference since it is not a leaf element (*e.g.*, *BooleanOperation* is its child element).

As prescribed by the step 4, the direction of the reference associations are inverted and tagged as *import* associations, shown in Fig. 3.16.

Fig. 3.16 shows the reference model for the implementation of the M2T generator for PLang. The rectangles tagged as *texttransformation* are the reference transformations (*e.g.*,

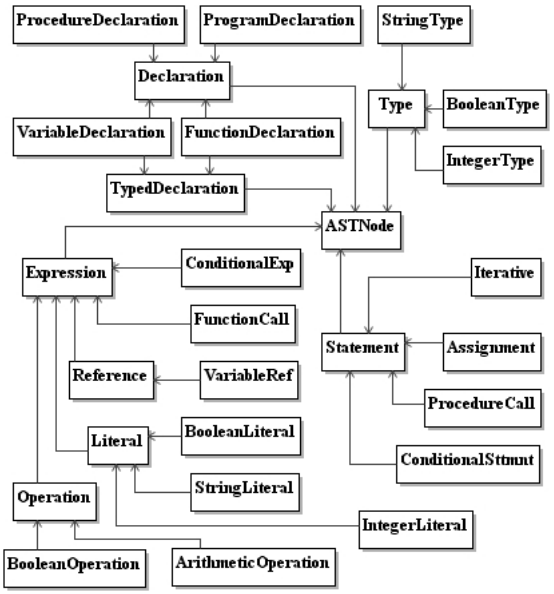

Figura 3.13: A preliminary version of the reference model after the application of the *step 1* on the PLang metamodel.

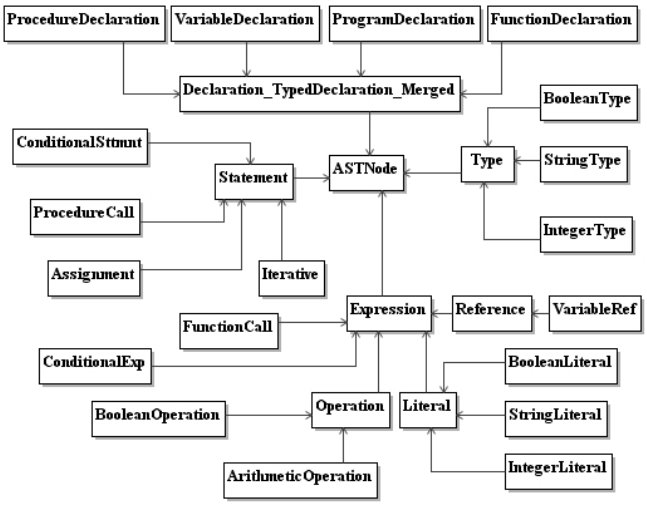

Figura 3.14: A preliminary version of the reference model after the application of the *step 2*.

ASTNode and *Expression*). The arrows tagged as *import* between the reference transformations are reference associations (*e.g.*, from *ASTNode* to *Expression* and from *Expression* to *Literal*). Additionally, inside each reference transformation we have the reference rules (*e.g*, *ConditionalExp()* and *FunctionCall()*, both in the *Expression* reference transformation).

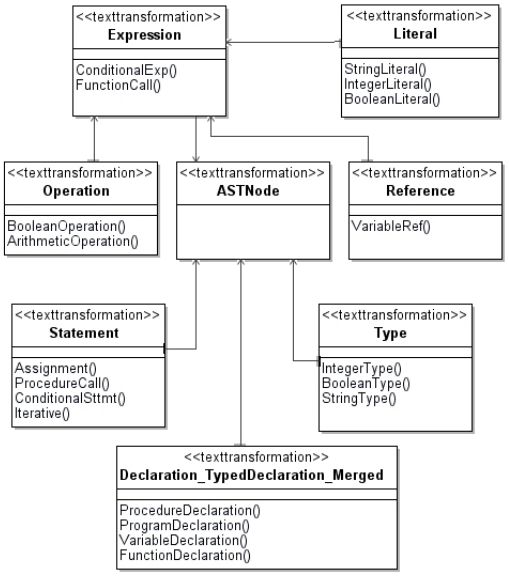

Figura 3.15: A preliminary version of the reference model after the application of the *step 3*.

Each one of the reference elements presented in Fig. 3.16 is used to guide the elaboration of the model-to-text transformations and their respective transformation rules. In our approach, the transformation developer uses the reference model to know which transformations he or she will find in each module and the rules he or she will find in each transformation, as well as the relationships between them.

3.2.2 Deriving Artifacts from the Reference Model.

Once we know how to separate the different concerns inside the M2T generator by means of the proposed architecture, we use the reference model to design the model-to-text transformations, with their respective rules, inside each module. In general, this procedure is given as follows. For each reference transformation in the reference model, derive the following artifacts:

- A transformation into the *Extractor* sub-module, that, in turn, comprises one extractor rule for each reference rule. The extractor rules are exposed by the *ITextExtractor* interface. They depend on the specification of the *ITextualDefinitions* and the *ICollectionExtractor* interfaces.

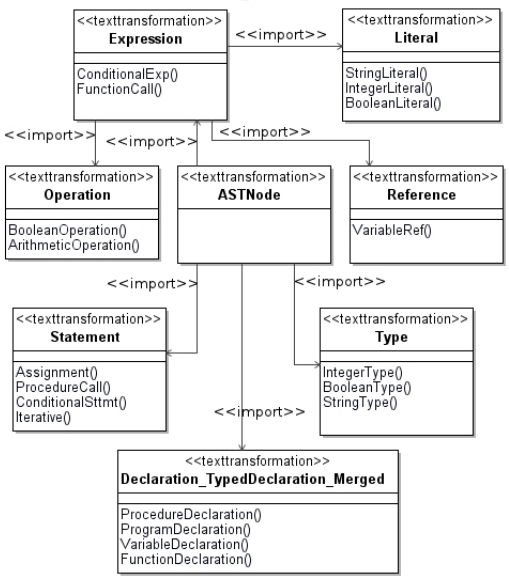

Figura 3.16: The reference model obtained after the application of the *step 4*.

- A transformation into the *Collections* sub-module, that, in turn, comprises one collection rule for each reference rule. The collection rules are exposed by the *ICollectionExtractor* interface and they depend on the *ITextualDelimiters* interface to work properly.

- A transformation into the *Templates* sub-module, that, in turn, comprises one template rule stub for each reference rule. The transformations in this sub-module do not depend on any other module and they are exposed by the *ITextualDefinitions* interface.

- Three properties in the *TerminalSymbols* sub-module for each reference rule (such as two embraces and one separator, as prescribed in Fig. 3.3). Such delimiters are exposed by the *ITextualDelimiters* interface.

The role each transformation plays in the process of M2T transformations generation is according to the role played by its container modules (and sub-modules). Therefore, the transformation *StatementExtractor*, from the *Extractor* sub-module, performs data extraction, whereas the transformation *StatementTemplates*, from the *Templates* sub-module, specifies the concrete syntax for *Statement* model elements. Analogous cases hold for the remaining transformations.

Bibliografia

[1] Nome do Autor. Titulo, 2006.

Apêndice A

Meu primeiro apêndice

Apêndice B

Meu segundo apêndice