SERGIO ROBERTO DE MELLO CANOVAS

Uma proposta de formalismo como arcabouço teórico para Engenharia Dirigida por Modelos e aplicações

São Paulo

2016

SERGIO ROBERTO DE MELLO CANOVAS

Uma proposta de formalismo como arcabouço teórico para Engenharia Dirigida por Modelos e aplicações

Tese apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do título de Doutor em Ciências

Orientador: Prof. Livre-Docente Carlos Eduardo Cugnasca

São Paulo

2016

SERGIO ROBERTO DE MELLO CANOVAS

Uma proposta de formalismo como arcabouço teórico para Engenharia Dirigida por Modelos e aplicações

Tese apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do título de Doutor em Ciências

Área de Concentração: Engenharia de Computação

Orientador: Prof. Livre-Docente Carlos Eduardo Cugnasca

São Paulo

2016

Este exemplar foi revisado e corrigido em relação à versão original, sob responsabilidade única do autor e com a anuência de seu orientador.

São Paulo, ______ de ____________________ de __________

Assinatura do autor: ________________________

Assinatura do orientador: ________________________

Catalogação-na-publicação

Canovas, Sergio Roberto de Mello
Uma proposta de formalismo como arcabouço teórico para Engenharia Dirigida por Modelos e aplicações / S. R. M. Canovas -- versão corr. -- São Paulo, 2016.
281 p.

Tese (Doutorado) - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Computação e Sistemas Digitais.

1.Engenharia dirigida por modelos 2.Engenharia de software I.Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Computação e Sistemas Digitais II.t.

AGRADECIMENTOS

Ao Prof. Dr. Carlos Eduardo Cugnasca, pela orientação desde o trabalho de conclusão de graduação até o doutorado, e pelo constante estímulo transmitido durante todo esse tempo, bem como sua pronta disponibilidade e atenção em todos os momentos necessários.

Ao Prof. Dr. João José Neto e ao Prof. Dr. Edson Satoshi Gomi, por terem participado da banca de qualificação deste trabalho e terem contribuído com importantes comentários e críticas.

Aos meus pais, Sérgio Luiz e Tercina, pelo incondicional e firme apoio durante esses árduos anos de trabalho.

À minha namorada Rebecca Sotelo por toda a compreensão e incentivo durante esse período.

Ao meu sócio e companheiro diário de trabalho Marlon Gripp Chermont pela compreensão durante minhas ausências para a realização deste trabalho.

A todos que colaboraram direta ou indiretamente na execução deste trabalho.

A mais alta das torres começa no solo.

Provérbio chinês

RESUMO

Engenharia Dirigida por Modelos, ou *Model Driven Engineering* (MDE), é uma abordagem para desenvolvimento de software a partir de modelos. Código-fonte ou artefatos executáveis são gerados de forma automática, total ou parcialmente, a partir de transformações descritas por funções de mapeamento. Com isso, obtém-se as vantagens de desenvolver software em nível de abstração maior em relação às linguagens de programação tradicionais e da possibilidade de gerar implementações do mesmo sistema para diversas plataformas a partir do mesmo modelo. Uma das áreas de pesquisa da MDE é a formalização de teorias e conceitos relacionados a essa abordagem, tais como modelos, metamodelos, relação de conformidade, operações sobre metamodelos, etc. Embora existam na literatura propostas de formalização, observam-se lacunas e falta de consenso geral, o que leva autores a introduzir suas próprias definições quando desejam apresentar desenvolvimentos ou deduções, as quais nem sempre se encaixam entre si. Alguns autores consideram que, enquanto uma completa formalização de conceitos e relações da MDE não for estabelecida, seus potenciais podem não ser plenamente atingidos. A partir de estudos de proposições existentes na literatura, identificação de lacunas e necessidades, este trabalho apresenta uma proposta de arcabouço teórico para MDE, baseada nas teorias de conjuntos e linguagens, e em lógica de primeira ordem. Conceitos e operações relacionados à MDE são definidos sobre uma base comum e, a partir dela, algumas formulações originais são desenvolvidas, tais como a função de mesclagem de metamodelos e uma definição de modelos executáveis, que podem servir como base para a construção de motores de execução de modelos, consolidando e ampliando o conhecimento da área teórica da MDE. Do ponto de vista de aplicabilidade prática, a proposta é validada por meio de uma ferramenta de prova de conceito, criada também como parte desta pesquisa, e pela apresentação de exemplos de aplicações.

Palavras-Chave: Engenharia Dirigida por Modelos. Set Based Meta Modeling. Metamodelos. Modelos.

ABSTRACT

Model Driven Engineering (MDE) is a software development approach in which models are essential artifacts to build software systems. Source code or executable artifacts are automatically generated, completely or partially, by transformations described by mapping functions. Two main advantages can be obtained: software development in a higher level of abstraction than that of traditional programming languages and the possibility of generating implementations of the same system for multiple platforms from the same source model. One of the research fields of MDE is the formalization of theories and concepts related to this approach, such as models, metamodels, conformity relationship, operations over metamodels, etc. Although there are proposals of formalization in the literature, some gaps and a lack of general consensus can be identified, which leads some authors to introduce their own definitions when they want to present their work. These definitions not always fit to each other. Some authors claim that unless a complete formalization of MDE concepts and relations is given, the potentials of this approach may not be fully unfolded. Starting from studies on existing proposals, gaps and requirements, this work proposes a theoretical framework for MDE based on the set theory, language theory and first order logic. Concepts and operations related to MDE are defined over a common basis and some original formulations are developed, such as the metamodel merging function and definitions about executable models, which can be used as a foundation to build model execution engines, consolidating and expanding the theoretical field of MDE. From the applicability point of view, the proposal is validated by a proof-of-concept tool, created as part of this research, and by examples of applications.

Keywords: Model Driven Engineering. Set Based Meta Modeling. Metamodels. Models.

LISTA DE ILUSTRAÇÕES

Figura 1 – Aumento histórico do nível de abstração 15
Figura 2 – Diagrama clássico de transformação de modelos da MDA......................20
Figura 3 – Classificação de modelos de forma contínua em duas dimensões..........22
Figura 4 – Diagrama de classes do sistema escolar exemplo 25
Figura 5 – Modelo para especificação da interface do cadastro de aluno................26
Figura 6 – A arquitetura das quatro camadas 30
Figura 7 – Conceitos principais da MDE e suas relações 40
Figura 8 – Parte da especificação do EMOF definida nela mesma..........................45
Figura 9 – Sintaxe de uma classe em NEREUS 60
Figura 10 – Associações em NEREUS 61
Figura 11 – Sintaxe de uma associação em NEREUS..61
Figura 12 – Sintaxe de um pacote em NEREUS...62
Figura 13 – Representação de $MM_{CLASSES}$ em notação gráfica................................84
Figura 14 – Representação de uma metaclasse com as propriedades em compartimento próprio 86
Figura 15 – Exemplo de importação de pacotes na UML.......................................108
Figura 16 – Exemplo de mesclagem de pacotes na UML 110
Figura 17 – (a) Rede de Petri com uma transição habilitada; (b) A mesma rede após o disparo...150
Figura 18 – Metamodelo para Redes de Petri (DDMM) 151
Figura 19 – Metaclasse *Place* com a propriedade de estado *numberOfTokens* (SDMM) 153
Figura 20 – Metaclasse *Firing* (EDMM)..154
Figura 21 – Construção do metamodelo executável autocontido...........................155
Figura 22 – Metamodelo autocontido para uma x-DSML de Redes de Petri 156
Figura 23 – Contextualização da função de sincronização 175
Figura 24 – Editor de metamodelo da ferramenta SBMMTool 184
Figura 25 – Editor de propriedades do metamodelo da ferramenta SBMMTool 185
Figura 26 – Editor de metaclasse da ferramenta SBMMTool.................................185
Figura 27 – Editor de propriedade de metaclasse da ferramenta SBMMTool 186
Figura 28 – Editor de generalização da ferramenta SBMMTool.............................186
Figura 29 – Editor de enumeração da ferramenta SBMMTool 187

Figura 30 – Editor de restrição da ferramenta SBMMTool 188
Figura 31 – Botão para checagem de metamodelo bem formado na ferramenta SBMMTool..189
Figura 32 – Tela de seleção de metamodelos para mesclar da ferramenta SBMMTool..191
Figura 33 – Editor de modelo da ferramenta SBMMTool 191
Figura 34 – Editor de propriedades do modelo da ferramenta SBMMTool 192
Figura 35 – Editor de instância da ferramenta SBMMTool.....................................192
Figura 36 – Editor de slot da ferramenta SBMMTool 193
Figura 37 – Botões para checagem de modelo bem formado e conformidade na SBMMTool..194
Figura 38 – Editor de função de mapeamento da ferramenta SBMMTool 195
Figura 39 – Exemplo de gramática de grafo tripla...219
Figura 40 – Exemplo de regra de gramática de grafo tripla 219
Figura 41 – Relações entre as linguagens ou camadas do QVT 222
Figura 42 – Exemplo de diagrama de classes UML para conversão em código-fonte Java...229
Figura 43 – Autômato MOFM2T_ROOT 238
Figura 44 – Autômato TEMPLATE..238
Figura 45 – Autômato COMMAND..239
Figura 46 – Autômato EXPRESSION..240
Figura 47 – Autômato VAR 240
Figura 48 – Autômato FOL..241
Figura 49 – Arquitetura de um programa não-adaptativo na forma de um dispositivo guiado por regras 243
Figura 50 – Arquitetura de um programa adaptativo 244
Figura 51 – Metamodelo de uma DSML para representação de programas adaptativos..246
Figura 52 – Tela da ferramenta SBMMTool com o metamodelo AdaptiveProgramMetamodel..249
Figura 53 – Modelo de programa adaptativo exemplo 250
Figura 54 – Modelo de programa adaptativo exemplo editado na ferramenta SBMMTool..251
Figura 55 – Edição da instância bloco1_saida na ferramenta SBMMTool 253

Figura 56 – Exemplo de interface de navegação de objetos....................................257
Figura 57 – Exemplo de interface de edição de objeto 258
Figura 58 – Exemplo de relatório de aplicativo de negócio 259
Figura 59 – Metamodelo de classes de uma DSML para aplicativos de negócio....263
Figura 60 – Metamodelo de interfaces de edição de objeto de uma DSML para aplicativos de negócio 264
Figura 61 – Metamodelo de interfaces de navegação de objetos de uma DSML para aplicativos de negócio 265
Figura 62 – Metamodelo de aplicações client de uma DSML para aplicativos de negócio...266
Figura 63 – Metamodelo de permissões de usuário de uma DSML para aplicativos de negócio..267
Figura 64 – Metamodelo de permissões de usuário de uma DSML para aplicativos de negócio..268

LISTA DE QUADROS

Quadro 1 – Modelo de domínio de negócio de uma escola em linguagem natural...24
Quadro 2 – EBNF descrito em EBNF...48
Quadro 3 – Comparativo de formalismos e linguagens segundo especificação 66
Quadro 4 – Comparativo de formalismos e linguagens sobre recursos...67
Quadro 5 – Comparativo de formalismos e linguagens sobre sintaxe concreta e utilização ...68
Quadro 6 – Função de comportamento de uma rede de Petri específica ...160
Quadro 7 – Algoritmo que especifica a função de comportamento da semântica de redes de Petri...161
Quadro 8 – Operações de modificação distintas...176
Quadro 9 – Palavras reservadas do interpretador de restrições correspondentes a símbolos de LPO...189
Quadro 10 – Exemplo de uso do operador # ...197
Quadro 11 – Comparativo de formalismos e linguagens segundo especificação, incluindo o SBMM ...200
Quadro 12 – Comparativo de formalismos e linguagens sobre recursos, incluindo o SBMM ...201
Quadro 13 – Comparativo de formalismos sobre sintaxe concreta e utilização, incluindo o SBMM ...202
Quadro 14 – Exemplo de declaração de função de mapeamento em QVT ...222
Quadro 15 – Exemplo de relação em QVT ...223
Quadro 16 – Exemplo de relação em QVT com cláusulas when e where...224
Quadro 17 – Exemplo de relações topo e não-topo...225
Quadro 18 – Exemplo de domínios apenas checagem e forçados ...225
Quadro 19 – Relação ClassToTable ...226
Quadro 20 – Exemplo de template gerador de classes Java em MOFM2T...229
Quadro 21 – Exemplo de classe Java gerada automaticamente ...230
Quadro 22 – Exemplo de chamada de template em MOFM2T ...230
Quadro 23 – Exemplo de classe Java gerada automaticamente com chamada de template ...231
Quadro 24 – Exemplo de utilização do laço for em MOFM2T...232
Quadro 25 – Exemplo de guarda em MOFM2T ...232

Quadro 26 – Exemplo de área protegida em template MOFM2T............................233
Quadro 27 – Exemplo de bloco file em MOFM2T 234
Quadro 28 – Tokens do analisador léxico MOFM2T e LPO...................................236
Quadro 29 – Código-fonte de bloco1 e imprime...252
Quadro 30 – Código-fonte da função adaptativa c 254
Quadro 31 – Função de mapeamento de modelos de programas adaptativos para linguagem BADAL..254
Quadro 32 – Função de mapeamento do bloco de classes e regras de negócio....271

LISTA DE ABREVIATURA E SIGLAS

ALF	Action Language for Foundational UML
ATL	ATL Transformation Language
ADT	Algebraic Data Type
BADAL	Basic Adaptive Language
BNF	Backus-Naur Form
CASL	Common Algebraic Specification Language
CLP	Constraint Logic Programming
CMOF	Complete Meta Object Facility
CRUD	Create/Retrieve/Update/Delete
CWA	Closed World Assumption
DDMM	Domain Definition MetaModel
DSL	Domain Specific Language
DSML	Domain Specific Modeling Language
EBNF	Extended Backus-Naur Form
EDMM	Event Definition MetaModel
EMOF	Essential Meta Object Facility
fUML	Foundational Subset for Executable UML Models
GPL	General Purpose Language
GPML	General Purpose Modeling Language
HTML	HyperText Markup Language
IDE	Integrated Development Environment
i-DSML	Interpreted Domain Specific Modeling Language
LPO	Lógica de Primeira Ordem
MDA	Model Driven Architecture
MDD	Model Driven Development
MDE	Model Driven Engineering
MML	Metamodeling Language Calculus
MoC	Model of Computation
MOF	Meta Object Facility
MOFM2T	MOF Model To Text Transformation Language
OCL	Object Constraint Language

OPL	Object Persistence Layer
ORB	Object Request Broker
OWA	Open World Assumption
PC	Personal Computer
PIM	Platform Independent Model
PSM	Platform Specific Model
QVT	Query / View / Transformation
RPC	Remote Procedure Call
SBMM	Set Based Meta Modeling
SDMM	State Definition MetaModel
SQL	Structured Query Language
TM3	Trace Management MetaModel
UML	Unified Modeling Language
URI	Uniform Resource Indicator
W3C	World Wide Web Consortium
x-DSML	Executable Domain Specific Modeling Language
XHTML	Extensible HyperText Markup Language
XMI	XML Metadata Interchange
XML	Extensible Markup Language
XSLT	Extensible Stylesheet Language Transformations
xUML	Executable UML

SUMÁRIO

1 INTRODUÇÃO 7
1.1 CONTEXTUALIZAÇÃO 7
1.2 MOTIVAÇÃO 9
1.3 OBJETIVO 11
1.4 METODOLOGIA 11
1.5 ORGANIZAÇÃO DO DOCUMENTO...12
2 CONCEITOS PRINCIPAIS SOBRE MDE E TENDÊNCIAS 14
2.1 AUMENTANDO O NÍVEL DE ABSTRAÇÃO 14
2.2 MODELOS DE SOFTWARE...16
2.3 TRANSFORMAÇÃO DE MODELOS NO CONTEXTO DA MDE 17
2.3.1 Transformação entre diferentes domínios de descrição...........................17
2.3.2 Transformação para diferentes plataformas...18
2.3.3 Classificação de modelos como PIM e PSM 19
2.3.4 Classificação contínua de modelos segundo Stephen Mellor 21
2.4 TRANSFORMAÇÃO MANUAL DE MODELOS 23
2.5 METAMODELOS 27
2.6 TRANSFORMAÇÃO AUTOMÁTICA DE MODELOS.....................................28
2.7 A ARQUITETURA DAS QUATRO CAMADAS 30
2.8 MODELOS DE MARCAÇÃO 34
2.9 REFINAMENTO DE MODELOS..35
2.10 PROCESSOS MDE 36
2.11 APLICAÇÃO PRÁTICA DA MDE 36
2.12 ABORDAGENS DE MODELAGEM 38
2.12.1 Linguagens de modelagem de propósito geral (GPML) 38
2.12.2 Linguagens de modelagem específicas de domínio (DSML) 39
2.13 RESUMO DAS DEFINIÇÕES E CONCEITOS PRINCIPAIS SOBRE MDE...40
2.14 DESAFIOS E TÓPICOS DE PESQUISA DA MDE 41
3 FORMALISMOS DE SUPORTE À METAMODELAGEM E MDE 42
3.1 INTRODUÇÃO...42
3.2 MOF 42
3.2.1 Conceitos principais 42
3.2.2 Aspectos teóricos da definição do MOF 46

3.2.3 Dificuldades na adoção do MOF 49
3.3 KM3 51
3.4 O FORMALISMO DE ALANEN E PORRES 55
3.5 NEREUS...59
3.6 A ABORDAGEM DE JACKSON, LEVENDOVSZKY E BALASUBRAMANIAN 63
3.7 QUADROS COMPARATIVOS..66
3.8 OUTROS FORMALISMOS E LINGUAGENS 68
3.9 MOTIVAÇÕES PARA O SBMM..69
4 SET BASED META MODELING (SBMM) 71
4.1 INTRODUÇÃO..71
4.2 METAMODELOS EM SBMM..71
4.3 EXEMPLO DE METAMODELO DESCRITO EM SBMM.......................................80
4.4 NOTAÇÃO GRÁFICA PARA METAMODELOS EM SBMM..................................83
4.5 METAMODELOS BEM FORMADOS..87
4.6 RESTRIÇÕES SOBRE MODELOS 88
4.7 MODELOS EM SBMM..92
4.8 EXEMPLO DE MODELO DESCRITO EM SBMM 97
4.9 RELAÇÃO DE CONFORMIDADE 101
4.10 MESCLAGEM E IMPORTAÇÃO DE METAMODELOS.....................................107
4.10.1 Mesclagem e importação no MOF e na UML..107
4.10.2 Mesclagem e importação de metamodelos no SBMM 110
4.10.3 Exemplo de mesclagem de metamodelos no SBMM.....................................122
4.11 MODELOS DE MARCAÇÃO 128
4.12 TRANSFORMAÇÃO DE MODELOS 132
4.13 FUNÇÕES PARA CONSTRUÇÃO E EDIÇÃO DE MODELOS 138
4.14 FUNÇÕES PARA CONSULTA SOBRE MODELOS...142
4.15 DESENVOLVIMENTOS ADICIONAIS SOBRE O SBMM 144
5 APLICAÇÕES TEÓRICAS 145
5.1 INTRODUÇÃO...145
5.2 MODELOS EXECUTÁVEIS...145
5.2.1 Introdução e Trabalhos Relacionados...145
5.2.2 Exemplo de Metamodelo para uma x-DSML de Redes de Petri 149
5.2.3 Exemplo de Modelo de Rede de Petri...157
5.2.4 Semântica de Execução..158

5.2.5 Motores de Execução....163
5.2.6 Considerações sobre Adaptação de Modelos em Tempo de Execução .164
5.2.7 Formalização de Conceitos Utilizando SBMM 165
5.3 DEFINIÇÕES E PROPRIEDADES SOBRE SINCRONIZAÇÃO DE MODELOS 174
6 UMA FERRAMENTA PARA APLICAÇÕES PRÁTICAS 183
6.1 INTRODUÇÃO....183
6.2 A FERRAMENTA....183
6.2.1 Editor de metamodelo 184
6.2.2 Mesclagem de metamodelos 190
6.2.3 Editor de modelo 190
6.2.4 Editor de função de mapeamento do tipo modelo-para-código....195
6.2.5 Motor de execução da função de mapeamento modelo-para-código 197
6.2.6 Interpretador de sentenças em lógica de primeira ordem 198
6.3 EXEMPLOS DE APLICAÇÃO....199
7 CONSIDERAÇÕES FINAIS 200
7.1 RESULTADOS E DISCUSSÃO 200
7.2 CONTRIBUIÇÕES....203
7.3 CONCLUSÃO 205
7.4 TRABALHOS FUTUROS....206
REFERÊNCIAS....208
APÊNDICE A – Métodos e Linguagens de Transformação com o SBMM 217
A.1 GRAMÁTICAS DE GRAFO TRIPLAS....217
A.2 QUERY / VIEW / TRANSFORMATION (QVT)....220
A.3 MOF MODEL TO TEXT TRANSFORMATION (MOFM2T) 228
APÊNDICE B – Autômatos dos Analisadores Sintáticos e Interpretadores 236
APÊNDICE C – Exemplo de Aplicação para Programas Adaptativos....242
APÊNDICE D – Exemplo de Aplicação para Aplicativos de Negócio....256

1 INTRODUÇÃO

1.1 CONTEXTUALIZAÇÃO

Engenharia Dirigida por Modelos, ou *Model Driven Engineering* (MDE), é uma abordagem para desenvolvimento de software baseada no uso de modelos como artefatos essenciais. Programas são construídos a partir de modelos de alto nível de abstração que podem ser transformados em código-fonte através de um processo automatizado. Em essência, a abordagem traz dois benefícios potenciais: o aumento do nível de abstração para o desenvolvedor e a possibilidade de gerar um programa para múltiplas plataformas, a partir do mesmo conjunto de modelos. Kolovos et al. (2013) afirmam que a MDE é considerada a mais recente mudança de paradigma na Engenharia de Software.

Entretanto, aplicar a MDE na prática ainda não é uma atividade consolidada. Ela requer o apoio de tecnologias e definições mais específicas para de fato trazer resultados satisfatórios.

O *Object Management Group* (OMG) mantém a *Model Driven Architecture* (MDA), que especializa o conceito de MDE e traz consigo um conjunto de elementos e tecnologias de apoio que é conhecido como infraestrutura da MDA. Dentre esses elementos, destaca-se o *Meta Object Facility* (MOF), uma linguagem para definição de linguagens de modelagem (OMG, 2015a). Os modelos escritos em MOF, por definirem linguagens para criar modelos de sistemas, são chamados de metamodelos. Como exemplo, existem metamodelos para a *Unified Modeling Language* (UML) escritos em MOF.

Embora a MDA seja marca registrada do OMG e se refira especificamente à MDE aplicada com suas tecnologias, é possível encontrar na literatura o uso desses termos sem distinção. Existe também um terceiro termo: *Model Driven Development* (MDD), que considera exclusivamente atividades de criação de sistemas baseada em modelos. De acordo com Cabot (2009), o termo MDE é mais geral que MDD por abranger também outras tarefas baseadas em modelos dentro da Engenharia de Software, tal como engenharia reversa de sistemas legados.

Para que seja manipulado por um programa e utilizado em transformações automáticas dentro do contexto da MDE, um modelo de software de alto nível de

abstração, tal como um diagrama de classes UML, deve ser escrito conforme o metamodelo da linguagem de modelagem utilizada. Isso ocorre de forma análoga a um programa de computador que deve ser escrito em conformidade com a gramática que define a sintaxe da linguagem de programação. Decorre daí da importância dos metamodelos na MDE. Mas a MDE como um todo ainda sofre com obstáculos decorrentes do pobre suporte de ferramentas, falta de suporte teórico e metodologias (YANG; LIU; WANG, 2012). Também carece de métodos ou processos de modelagem efetivos e bem conhecidos (YANG; LIU; LI, 2014).

Uma vez que a UML é o atual padrão de fato para modelagem de software orientado a objetos (SHAH; IBRAHIM, 2014), o caminho natural para a adoção da MDE é a utilização de ferramentas que ou transformam modelos UML em código-fonte, ou sejam capazes de simulá-los.

A falta de semânticas claras e bem definidas em diagramas e elementos existentes na UML constituem impedimentos graves para a simulação ou geração de código automática a partir de modelos UML (RODRIGUES; 2009). Outra dificuldade está no fato de que a UML depende de linguagem natural para descrever certos aspectos, sendo muito difícil utilizá-la como entrada para ferramentas de geração de código (MARCHETTI; 2012).

Embora a transformação de diagramas de classes UML em código-fonte com a estrutura das classes seja relativamente bem estabelecida (SILVA, 2005), diversas iniciativas visam resolver outros aspectos ainda não tão bem consolidados. Isso ocorre principalmente com relação aos aspectos dinâmicos da modelagem, tais como a conversão de diagramas de atividades UML em código-fonte, que determina o comportamento da aplicação (GESSENHARTER; RAUSCHER, 2011), ou a transformação de modelos de processos de negócio (PADILLA, 2014).

A *Executable UML* (xUML), originalmente proposta por Mellor e Balcer (2002), e a *Foundational Subset for Executable UML Models* (fUML) (OMG, 2016c) têm por objetivo estabelecer subconjuntos da UML com semântica precisamente definida e padronizada, de modo que permitam a transformação de modelos xUML ou fUML em código-fonte com comportamento definido, sem ambiguidades.

Entretanto, mesmo que essas dificuldades sejam superadas, muito provavelmente o uso apenas da UML como linguagem de modelagem de software não será suficiente

para aplicação e utilização eficiente da MDE em todos os casos. Isso porque a UML não é completa, ou seja, não provê diagramas apropriados para modelagem de todos os aspectos de um software em todos os contextos. Como exemplos, Ambler (2004) cita que a UML não possui um diagrama para modelar interfaces de usuário, enquanto Shah e Ibrahim (2014) destacam, mais recentemente, que a UML não provê conceitos para expressar recursos e características peculiares do sistema operacional móvel Android, resultando em dificuldades de modelagem.

Em resumo, a UML é uma linguagem de propósito geral, ou *General Purpose Language* (GPL). Uma vez que a proposta central da MDE é criar software a partir de modelos de alto nível de abstração, dificilmente a UML será capaz de prover os elementos necessários para modelagem em alto nível de abstração em todas as classes de aplicação consideradas.

Por outro lado, as linguagens específicas de domínio, ou *Domain Specific Languages* (DSLs), existem para diminuir essa distância. Trata-se de linguagens projetadas para serem úteis para um conjunto limitado de tarefas ou para uma classe bem específica de aplicações, em contraste com as GPLs, que precisam ser úteis para muitas tarefas genéricas e cruzar múltiplos domínios de aplicação (JOUAULT; BÉZIVIN, 2006) (JÉZÉQUEL et al., 2012).

DSLs para modelagem de software podem ser criadas a partir de extensões de GPLs (como a própria UML) ou definidas por completo, a partir da criação de novos metamodelos. Esse tipo de DSL também é comumente denominado *Domain Specific Modeling Language* (DSML).

As DSMLs despontam como um meio para suprir carências da UML na adoção e utilização prática da MDE nas organizações, funcionando de forma complementar ou até mesmo por si só. Isso atribui importância aos atos de criar, estender e manipular os artefatos que descrevem e implementam DSMLs, bem como os modelos descritos em conformidade com elas.

1.2 MOTIVAÇÃO

Embora o MOF seja o padrão para metamodelagem estabelecido pelo OMG, podendo ser usado para definir DSMLs, ele apresenta alguns aspectos negativos

que serão detalhados na subseção 3.2.3, tais como não prover semântica formal, ser bastante extenso a ponto de ter de ser dividido em dois grupos, apresentar ambiguidades e ser especificado utilizando ele mesmo, ou seja, de modo metacircular. Apesar de esta última característica não ser necessariamente um ponto negativo, constitui-se em uma barreira de entrada adicional para seu entendimento, segundo Bézivin e Gerbé (2001). Há também um consenso atual de que o MOF está subformalizado em suas definições (JACKSON; LEVENDOVSZKY; BALASUBRAMANIAN, 2013), o que traz desvantagens para definição de operações, propriedades e provas. Xiong et al. (2007), por exemplo, apresenta propriedades sobre sincronização de modelos. Porém, sem um formalismo subjacente adequado, precisa introduzir diversas definições próprias para conseguir expressá-las, e essas definições acabam existindo de forma isolada, não necessariamente se encaixando com facilidade em um arcabouço mais geral. No mesmo contexto, Rutle et al. (2012) observa que a formalização da correspondência entre linguagens de modelagem e metamodelos, bem como a relação de conformidade entre modelos e metamodelos, são aspectos não incluídos nos padrões do OMG.

Esses aspectos motivaram o surgimento de outros formalismos e linguagens alternativas ao MOF, tais como o KM3 (JOUAULT; BÉZIVIN, 2006), MOMENT2 (BORONAT; MESEGUER, 2008), NEREUS (FAVRE, 2009), e a abordagem de Jackson, Levendovszky e Balasubramanian (2013), entre outros.

Apesar de esses e outros formalismos apresentarem aspectos ou características não presentes diretamente no MOF, tais como a utilização de definições em lógica de primeira ordem e a aplicação de Prolog (ou outra linguagem lógica) para realizar deduções com modelos e metamodelos, outros requisitos fundamentais da MDE como a extensão de metamodelos e o refinamento de modelos não são tratados nesses formalismos, ou não são tratados com o devido detalhe para aplicações práticas no contexto da MDE.

De acordo com Rutle et al. (2012), muitos pesquisadores consideram que, enquanto uma completa formalização dessas relações não for estabelecida, os potenciais da MDE podem não ser plenamente atingidos.

1.3 OBJETIVO

O objetivo deste trabalho é propor um formalismo que sirva como arcabouço teórico para a MDE, ou seja, que permita estabelecer definições, descrever metamodelos, modelos, relações e operações sobre os mesmos. Também faz parte do objetivo deste trabalho avaliar esse formalismo no contexto de suas aplicações teóricas e práticas, bem como compará-lo com outras soluções, inclusive o MOF.

Deseja-se um formalismo não metacircular, que seja definido com base em conceitos primitivos, que possa servir tanto para aplicação direta quanto como base para desenvolvimentos mais complexos. Este formalismo é denominado *Set Based Meta Modeling* (SBMM), por ser baseado principalmente em conjuntos e relações, embora se baseie também em lógica de primeira ordem.

O SBMM deve ser encarado como uma alternativa ao MOF e outros formalismos ou linguagens encontradas na literatura, tendo como diretriz cobrir aspectos importantes da MDE que não são cobertos ou não são tratados suficientemente ou eficientemente por esses outros formalismos. Por outro lado, não se tem a pretensão de que o SBMM resolva todos os problemas, mantendo a validade e vantagens de outras soluções, principalmente aquelas com foco distinto, por exemplo a realização de inferência lógica.

1.4 METODOLOGIA

Para cumprir o objetivo proposto, estabeleceu-se a seguinte metodologia para este trabalho de pesquisa:

- Revisão bibliográfica sobre MOF e outros formalismos encontrados na literatura para metamodelagem e modelagem, avaliando vantagens, desvantagens e limitações de cada um;
- Identificação de pontos de melhoria dos formalismos e lacunas na aplicação para MDE;
- Formulação do SBMM e operações relacionadas;

- Avaliação do SBMM do ponto de vista teórico, por meio de sua aplicação em duas áreas de pesquisa da MDE encontradas na literatura, a saber: modelos executáveis e sincronização de modelos;
- Avaliação do SBMM do ponto de vista prático, por meio da criação de uma ferramenta para prova de conceito, que aplica diretamente o SBMM para fins criação de metamodelos, modelos e transformações de modelos em código-fonte. Para testar a aplicação prática na MDE, entende-se que uma avaliação apenas teórica não seja possível para esta finalidade;
- Estudos de caso de aplicação da ferramenta em dois exemplos de classes de software distintas, a saber: programas adaptativos e aplicativos de negócio, utilizando o SBMM como formalismo subjacente, com o propósito de avaliar a real aplicabilidade dentro do paradigma da MDE.

1.5 ORGANIZAÇÃO DO DOCUMENTO

O Capítulo 2 apresenta os termos e conceitos principais sobre MDE, permitindo o levantamento e entendimento dos requisitos que um arcabouço ou formalismo teórico de suporte à MDE, bem como ferramentas de apoio, devem prover. Algumas tendências observadas no contexto atual também são apresentadas.

O Capítulo 3 introduz formalismos para metamodelagem e modelagem encontrados na literatura, identificando os principais aspectos de cada um e comparando-os em função de critérios que representam requisitos ou características da MDE.

O Capítulo 4 detalha o SBMM, formalismo criado como parte desta pesquisa, e mostra como metamodelos e modelos, bem como operações sobre eles, podem ser expressos por meio de sua proposta. Também são apresentados conceitos complementares como relação de conformidade e extensão de metamodelos.

O Capítulo 5 ilustra a utilização do SBMM em duas aplicações teóricas: a caracterização formal de modelos executáveis (não-adaptativos e adaptativos) e a apresentação de definições e propriedades sobre sincronização de modelos.

O Capítulo 6 apresenta a SBMMTool, a ferramenta de software para aplicação da MDE, apoiada no formalismo SBMM, criada como prova de conceito para esta

pesquisa. Com ela é possível editar metamodelos, modelos e transformá-los em programas. Tudo isso usando o SBMM como fundação teórica subjacente.

O Capítulo 7 apresenta e discute os resultados desta pesquisa, concluindo o trabalho e citando propostas de trabalho futuro.

O Apêndice A fornece um resumo sobre alguns métodos e linguagens de transformação de modelos, originalmente concebidos para o MOF, e considerações sobre sua aplicação com o SBMM. O Apêndice B detalha os autômatos utilizados na construção do motor de transformação da ferramenta SBMMTool, enquanto os Apêndices C e D apresentam dois exemplos de utilização desta ferramenta para a criação de aplicativos segundo a abordagem MDE. Respectivamente, tratam de programas adaptativos e aplicativos de negócio.

2 CONCEITOS PRINCIPAIS SOBRE MDE E TENDÊNCIAS

2.1 AUMENTANDO O NÍVEL DE ABSTRAÇÃO

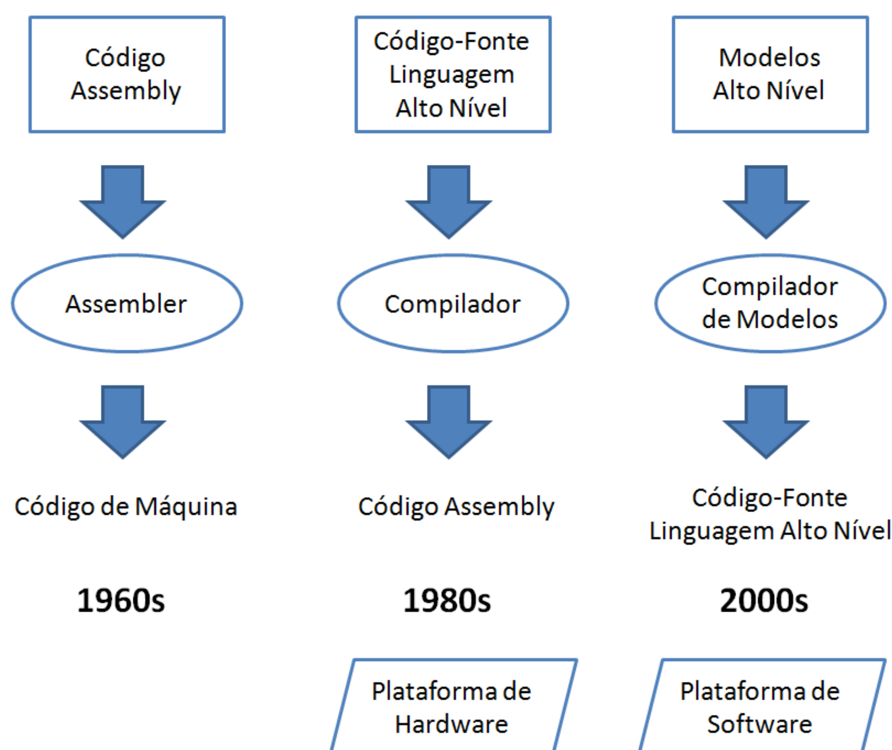
Na evolução dos procedimentos e técnicas para o desenvolvimento de software, pode-se constatar o contínuo aumento de nível de abstração. Nos primórdios da era da computação, programas eram transferidos por meio da entrada direta de seu código de máquina por manipulação física de chaves. Posteriormente, o código elaborado por meio de linguagem de montagem (*assembly*) substituiu a codificação direta de instruções representadas em código binário por mnemônicos específicos para cada processador (WAGNER et al., 2006). Já no final dos anos de 1950, começaram a surgir as linguagens de programação, como ALGOL e COBOL, que permitiram a criação de programas estruturados baseados em blocos, mais fáceis de serem escritos, entendidos e mantidos, além de permitirem portabilidade para diversas plataformas de hardware devido à disponibilidade de compiladores para elas (WAGNER et al., 2006). Posteriormente, surgiram as linguagens baseadas no paradigma orientado a objetos, tais como Smalltalk, C++, Eiffel, Java, etc. (MELLOR et al., 2004). Java e C++, por exemplo, são extensivamente utilizadas em diversos tipos de aplicação.

Nessa evolução descrita, de forma geral o nível de abstração no qual o desenvolvedor trabalha foi elevado, o que requereu o aprendizado de novas linguagens de mais alto nível que pudessem ser mapeadas em linguagens conhecidas de menor nível de abstração, por exemplo, C++ para C, e C para *assembly*. De início, cada camada de abstração mais elevada foi introduzida apenas como um conceito, para somente depois ser formalizada e implementada através de ferramentas e compiladores (MELLOR et al., 2004).

Observa-se, então, um padrão nesta evolução: o conhecimento de uma aplicação é expresso em uma linguagem de nível mais alto em relação às conhecidas, mas ainda com poucos formalismos, definições e padronizações. Com o tempo, aprende-se a usar esta linguagem e um conjunto de convenções se forma, permitindo a consolidação desta formalização e o estabelecimento de padrões. Tem-se, então, uma linguagem de mais alto nível que pode ser mapeada automaticamente em uma linguagem de mais baixo nível. Em um próximo ciclo, esta linguagem passa a ser

vista como de nível mais baixo para servir como base a uma nova de nível ainda mais alto. O processo se repete e o nível de abstração com o qual os desenvolvedores lidam diretamente aumenta. O próximo passo nesse caminho de elevação de nível de abstração é o desenvolvimento baseado em modelos, conforme Figura 1.

Figura 1 – Aumento histórico do nível de abstração



Fonte: Mellor et al. (2004) (traduzido)

Embora a Figura 1 destaque a década de 2000 como a década do desenvolvimento dirigido a modelos, a MDE ainda não é uma realidade amplamente disseminada na indústria de software, carecendo de ferramentas (YANG; LIU; WANG, 2012), de suporte teórico e de metodologias (YANG; LIU; LI, 2014). Mais detalhes são apresentados na seção 2.11. De qualquer modo, esta abordagem tem como intenção possibilitar a construção de sistemas através de modelos independentes de plataforma de software, sendo sua ideia central.

2.2 MODELOS DE SOFTWARE

Segundo Mellor et al. (2004), um modelo consiste em um conjunto de elementos que descreve alguma realidade física, abstrata ou hipotética. São descritores dessa realidade modelada, em algum nível de abstração que serve para algum propósito. Bons modelos servem como meios de comunicação: são mais baratos de construir do que a realidade modelada, e permitem que a equipe de trabalho os utilize para desenvolver e discutir estratégias para resolver o problema em questão. Modelos podem ser rascunhos simples, plantas detalhadas ou até mesmo modelos executáveis de software.

Um modelo de sistema de software é um modelo que descreve ou especifica o sistema, podendo considerar aspectos de seu ambiente. É frequentemente apresentado como uma combinação de desenhos e texto. O modelo pode estar expresso em uma linguagem de modelagem ou em linguagem natural (OMG, 2003).

Complementarmente, de acordo com Rutle et al. (2012), um modelo de software é uma abstração que pode não representar todos os aspectos e propriedades do sistema real, mas apenas aquelas que são relevantes para um dado contexto. A UML é considerada o padrão de fato de linguagem de modelagem de software (GESSENHARTER; RAUSCHER, 2011) (SHAH; IBRAHIM, 2014), provendo um conjunto de diagramas e elementos que permitem descrever e especificar diversos aspectos de um software.

A UML é uma linguagem gráfica com o objetivo de ser facilmente interpretada pelos desenvolvedores. Entende-se por linguagens gráficas aquelas que utilizam notação baseada em elementos visuais, ou desenhos, e não apenas em texto.

O código-fonte de um software também é um modelo, em geral de nível de abstração mais baixo que a UML, mais próximo à plataforma de software e/ou hardware. Códigos-fonte são expressos em linguagens baseadas em texto, tais como C++, Java, etc.

Para manter a generalidade, o software construído em sua forma final (seja por meio de códigos-fontes interpretáveis ou códigos binários compilados) também será considerado um modelo. Embora ele não seja mais um modelo de acordo com a

definição acima, e sim a própria realidade construída, será possível posicioná-lo em uma classificação de modelos a ser apresentada na subseção 2.3.4.

Para a MDE, um modelo de software não deve ser visto apenas como um descritor para documentação e comunicação sobre um sistema de software a ser construído, mas sim como artefato fundamental de primeira classe para sua própria construção através da transformação de modelos (KOLOVOS et al., 2013). Uma das metas principais da MDE é a manipulação de modelos como artefatos exclusivos da construção do software (CARIOU et al., 2013).

2.3 TRANSFORMAÇÃO DE MODELOS NO CONTEXTO DA MDE

2.3.1 **Transformação entre diferentes domínios de descrição**

Um processo, metodologia ou técnica de desenvolvimento de software, seja qual for, é composto por diversas etapas, independentemente do nome que recebam. No Processo Unificado, por exemplo, divide-se o tempo em fases que por sua vez são subdivididas em iterações (BOOCH; RUMBAUGH; JACOBSON, 1999b).

Em cada etapa, modelos são criados ou atualizados e devem manter coerência com outros modelos gerados no processo. Quase sempre a alteração em um modelo (p. ex., especificação de requisitos) implica obrigatoriamente em alterações em outros modelos (p. ex., diagrama de classes e modelo entidade-relacionamento). A alteração dos modelos de forma organizada para manter essa coerência é denominada sincronização de modelos.

Em outras palavras, quando um modelo de certo domínio de descrição é atualizado, modelos de outros domínios que compartilham ou dependem de elementos do primeiro modelo devem ser sincronizados para refletir as mudanças. Esses diferentes modelos usualmente coexistem e evoluem de forma independente (XIONG et al., 2007).

Manter os modelos corretamente sincronizados de forma manual em um projeto de software requer boa comunicação e entendimento da equipe, sendo, em geral, caro e suscetível a erros (AMBLER, 2004), ainda mais levando em conta as técnicas e metodologias mais modernas, que são iterativas e apresentam evoluções incrementais sucessivas dos modelos de todos os níveis de abstração, à medida

que o software é construído. As metodologias ágeis têm sido levadas aos seus limites devido às pressões atuais por construir software rapidamente, cenário característico principalmente na área de aplicativos móveis (MAXIMILIEN; CAMPOS, 2012), e isso demanda novas soluções na Engenharia de Software.

São desejáveis técnicas que possibilitem, de alguma forma, a transformação e sincronização automática de modelos de domínios de descrição distintos quando os mesmos possuem relações. Como benefícios, o processo de desenvolvimento seria mais rápido e o software teria mais qualidade (MELLOR et al., 2004).

Embora existam técnicas específicas para executar transformação e sincronização de modelos, a propagação de modificações entre modelos que coexistem e expressam conceitos diferentes, porém inter-relacionados, ainda é um problema típico na MDE (ZAN; PACHECO; HU, 2014).

2.3.2 **Transformação para diferentes plataformas**

Na década de 2010, na área de desenvolvimento de software, tornou-se significativo o problema do grande número de plataformas nas quais as aplicações potencialmente precisam ser executadas: hardwares e sistemas operacionais distintos, ambiente *web*, ambiente *desktop*, *tablets*, *smartphones*, etc. Novas plataformas surgem cada vez mais rapidamente com o avanço tecnológico. Algumas adquirem sucesso no mercado e se estabelecem em longo prazo, enquanto outras não têm a mesma sorte e rapidamente ficam fora do mercado.

Além da variedade de plataformas em si, cada uma possui características referentes à sua modalidade. Um dispositivo móvel, por exemplo, apresenta características de processamento, usabilidade e interfaces distintas das de um computador pessoal. Isso faz com que o uso de máquinas virtuais para executar o mesmo código em múltiplas plataformas nem sempre seja suficiente ou adequado para portar aplicações entre diferentes plataformas. A aplicação deve respeitar as características de cada uma, inclusive sua modalidade.

Mesmo considerando uma única plataforma como base (p. ex., computador pessoal com o sistema operacional Windows), existem diversas opções de escolha no nível de implementação (p. ex., .NET, Delphi, C++, Java). Neste nível se encontram as linguagens de programação, compiladores, ambientes de desenvolvimento ou

Integrated Development Environments (IDEs), *frameworks*, bibliotecas, *middlewares*, protocolos, etc.

A existência de múltiplas plataformas e possibilidades no nível de implementação cria um cenário no qual as organizações precisam enfrentar uma decisão difícil no momento de adotar uma plataforma e tecnologia de implementação para seus projetos de software. A escolha, de modo a preservar o investimento pelo maior tempo possível, e de modo a garantir que a utilização e manutenção de sistemas sobrevivam frente às novas tendências e tecnologias que rapidamente surgem e substituem tecnologias anteriores são questões importantes enfrentadas pelas organizações (OMG, 2014a).

A MDE também busca resolver este problema na medida em que propõe partir de modelos de software independentes de plataforma e, por meio de transformações automáticas, obter implementações específicas da aplicação para diferentes plataformas respeitando suas características.

2.3.3 **Classificação de modelos como PIM e PSM**

O diagrama clássico da MDA do OMG prevê dois níveis de abstração do ponto de vista de plataformas: o modelo independente de plataforma ou *Platform Independent Model* (PIM) e o modelo específico de plataforma ou *Platform Specifc Model* (PSM), dependente da mesma (AMBLER, 2004). A MDA propõe que o PSM seja gerado automaticamente a partir do PIM através de uma transformação.

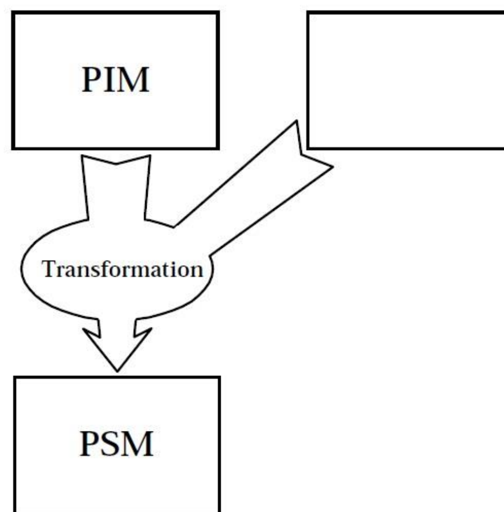
A Figura 2 ilustra esse esquema: um PIM, junto com informações adicionais (quadro em branco), passa por um mecanismo de transformação para gerar um PSM.

As informações adicionais representadas pelo quadro em branco da Figura 2 podem ser tanto para utilização do mecanismo de transformação (p. ex., mapeamentos, regras, *templates*, padrões, procedimentos) quanto dados extras para complementar o PIM e prover informações suficientes para gerar o PSM (p. ex., conjunto de marcações, conforme será definido mais adiante) em determinada plataforma. Isso pode ser necessário, pois, como o PIM não considera aspectos específicos da plataforma para a qual o sistema será gerado, muitas vezes é preciso introduzir informações extras para cobrir os novos detalhes que se fazem necessários. Este

tipo de transformação é denominado modelo-para-modelo (*model-to-model transformation*)

A próxima etapa de transformação a partir do PSM seria a geração do código-fonte do sistema desejado, que também deve ser da forma mais automática possível para trazer os benefícios desejados (OMG, 2014b). Esse tipo de transformação é classificado como modelo-para-código (*model-to-code transformation*).

Figura 2 – Diagrama clássico de transformação de modelos da MDA



Fonte: OMG (2003)

Na situação ideal, em que o sistema pode ser gerado completamente a partir de transformações, qualquer manutenção no sistema se reduz a uma manutenção no PIM e, eventualmente, nas informações adicionais. Por meio da reaplicação da transformação, os PSMs e implementações específicas para cada plataforma de interesse são gerados novamente.

Dessa forma, os arquitetos e desenvolvedores de software podem concentrar seus esforços em desenvolver e manter um modelo conceitual abstrato da aplicação (PIM), o qual contém as regras de negócio e seus aspectos fundamentais, sem precisar se basear em alguma plataforma ou tecnologia de implementação específica.

Os investimentos das organizações em projetos de software seriam então preservados na medida em que um mesmo PIM, já criado, poderia dar origem

automaticamente a novas implementações para outras plataformas que surgem no contexto da organização (OMG, 2014a). Mais interessante ainda, esta abordagem permite gerar sistemas existentes através de seu PIM para novas plataformas que ainda estão por ser inventadas, aproveitando-se os modelos de alto nível de abstração já construídos.

Além dessas vantagens por si só, a retirada do foco dos esforços da implementação e seu redirecionamento para o modelo conceitual da aplicação é bem-vinda no sentido de que muitos projetos falham por má definição de requisitos do usuário ou área de negócio contratante (KÖBLER et al., 2008). Assim, essas questões podem ser tratadas com melhor cuidado e atenção, trazendo outro ganho.

2.3.4 **Classificação contínua de modelos segundo Stephen Mellor**

Apesar de elucidativa, a classificação de modelos em PIM e PSM nem sempre é prática. Dados dois modelos arbitrários, é mais realista classificá-los em mais independente de plataforma ou menos independente de plataforma. Isto é, os modelos não são necessariamente PIMs puros (totalmente independentes de plataforma) ou PSMs puros (totalmente suficientes para descrever o software em uma plataforma). Eles têm certo grau de dependência ou independência de plataforma. Sendo assim, não obrigatoriamente precisam ser aplicados exatamente dois passos de transformação (PIM para PSM e PSM para código-fonte). Pode ser aplicado um número arbitrário de passos de transformação.

Existem linguagens mais adequadas para expressar modelos mais independentes de plataformas (ex: UML), e outras mais adequadas para expressar modelos mais dependentes de plataformas (p. ex., linguagem C). As primeiras são chamadas de linguagens mais abstratas e as últimas de linguagens mais concretas. Essas últimas, em geral, apresentam elementos cuja semântica faz referência mais direta a recursos, dispositivos e processos computacionais. Os modelos mais independentes de plataforma são expressos de melhor forma em linguagens mais abstratas e os modelos menos independentes de plataformas em linguagens mais concretas (MELLOR et al., 2004).

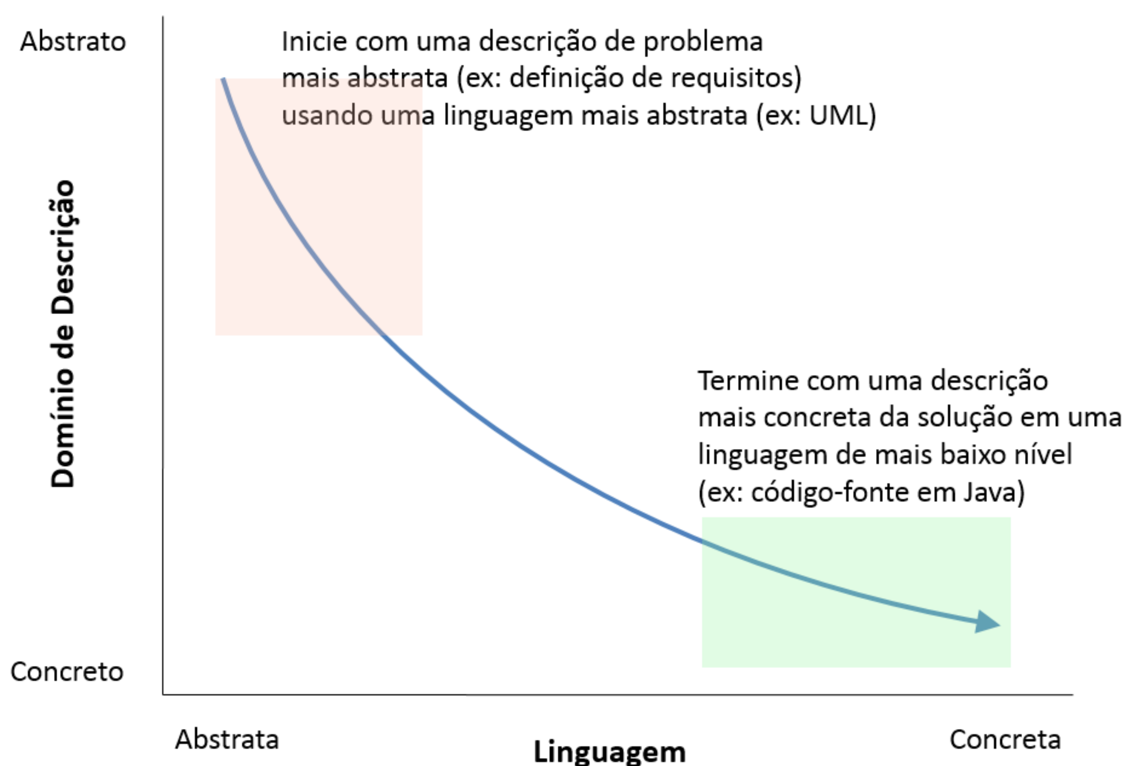
Assim como ocorre do ponto de vista de plataforma, também não existe uma classificação precisa para os domínios de descrição. Por exemplo, não é proibido

que um modelo de requisitos contenha alguns aspectos de construção da solução. Isso porque a descrição de um caso de uso da UML, por exemplo, pode trazer uma sequência de eventos que implique em alguma característica concreta da interface de usuário, que teoricamente faria parte de outro domínio de descrição.

Em resumo, o nível de abstração de um modelo tanto sob o ponto de vista de dependência de plataforma (como visto, também relacionada com a linguagem utilizada) quanto sob o ponto de vista de domínio de descrição pode variar dentro de uma faixa, não necessariamente estando dentro de níveis discretos bem definidos.

A Figura 3 apresenta uma curva que generaliza o modelo de transformação da Figura 2. Segundo ela, a abordagem geral da MDE deve iniciar a partir de um modelo mais independente de plataforma e de mais alto nível de domínio de descrição, indo em direção a modelos menos independentes de plataforma e de domínios de descrição mais próximos da solução computacional final.

Figura 3 – Classificação de modelos de forma contínua em duas dimensões



Fonte: Mellor et al. (2004) (adaptado)

As transformações podem ocorrer em um número arbitrário de passos, e o grau de classificação geral de cada modelo quanto à dependência de plataforma e quanto a domínio de descrição pertence a uma faixa contínua.

A continuidade desta curva, assim como a possibilidade de mistura de diferentes domínios de descrição em um mesmo modelo, é a fonte principal de discussão sobre o que exatamente constitui ou deve ser um modelo de requisitos, um modelo de análise, um modelo de *design*, etc. (MELLOR et al., 2004).

Entendendo-se a existência e possibilidade deste contínuo, a discussão torna-se desnecessária e podem-se criar os modelos mais convenientes para a prática sem a preocupação de classificá-lo exatamente como um PIM ou PSM puros, ou como um modelo pertencente exclusivamente a um único domínio de descrição.

Um modelo arbitrário pode estar em qualquer ponto da área do gráfico da Figura 3. A curva ilustra de forma geral a direção que a MDE propõe seguir: da esquerda para a direita, de cima para baixo.

Antes de abordar as transformações automáticas de modelos, será visto um exemplo de transformação manual com o intuito de identificar problemas, dificuldades e requisitos a serem tratados em transformações automáticas.

2.4 TRANSFORMAÇÃO MANUAL DE MODELOS

Será considerado, a seguir, um exemplo prático de transformação manual de modelos. Supondo que um analista estudou o domínio de negócio de uma escola e definiu um modelo simples em linguagem textual natural (facilmente interpretada por humanos e dificilmente interpretada por máquinas) que descreve um problema a ser resolvido através de um sistema de software, conforme o Quadro 1.

Evidentemente, este modelo, que serve para o propósito ilustrativo, é simples quando comparado com casos típicos, pois nem mesmo menciona notas ou critérios de aprovação. O modelo está expresso em linguagem natural, sem fazer referência a nenhuma plataforma computacional ou recurso específico. Está, portanto, descrito em uma linguagem abstrata, e descreve aspectos sobre como o negócio funciona e também sobre os requisitos desejados para a solução. Não descreve, por exemplo, como deve ser a interface de usuário ou a sequência de operações do sistema,

pertencendo, então, a domínios de descrição abstratos. Assim sendo, o modelo se localiza em algum ponto na parte superior à esquerda, dentro da área vermelha da Figura 3.

Quadro 1 – Modelo de domínio de negócio de uma escola em linguagem natural

Em uma escola, existem cursos, turmas e alunos. Em um dado semestre, alunos matriculam-se em cursos, formando-se turmas. Uma turma pode ter de 1 a 40 alunos. Havendo mais alunos matriculados do que o permitido, a escola pode optar por abrir uma nova turma para o mesmo curso no mesmo semestre. Um único professor ministra um curso para uma turma, mas o mesmo professor pode ministrar um ou mais cursos para mais de uma turma. Um professor também pode estar desalocado no semestre, não ministrando nenhum curso para nenhuma turma. Deseja-se construir um sistema que permita armazenar, cadastrar e consultar alunos, turmas, cursos, professores e notas.

Fonte: autor

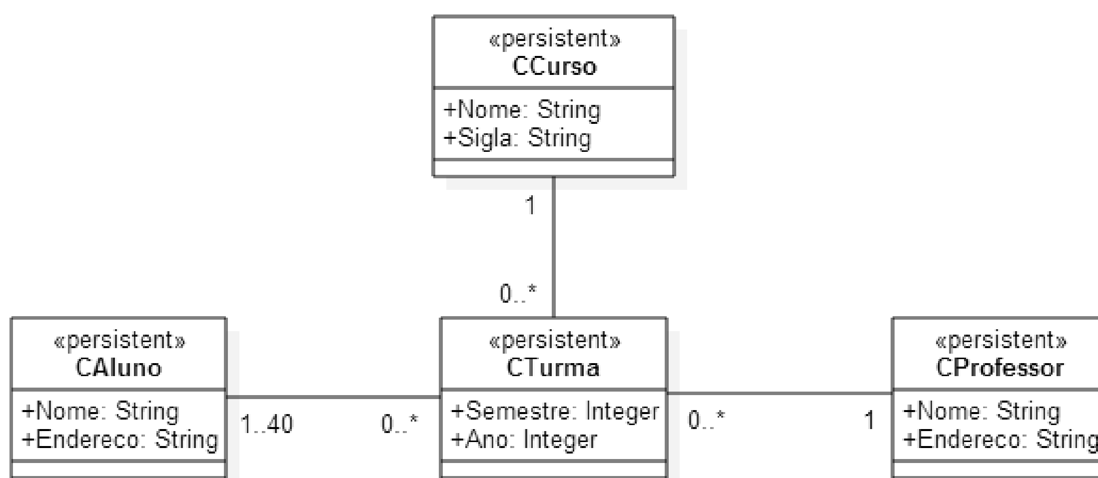
Sabe-se que, por estar descrito em linguagem natural, é muito difícil aplicar transformações automáticas com esse modelo. Mas, para fins de exemplo, aplicar-se-á a estratégia da MDE manualmente.

De acordo com a Figura 3, deseja-se partir desse modelo em direção a uma implementação concreta de sistema, seguindo a direção da curva.

O analista poderia, então, propor o modelo da Figura 4, expresso por meio de um diagrama de classes UML, que descreve os tipos de dados com os quais o sistema lida e seus relacionamentos.

Esse modelo reflete apenas um domínio de descrição específico da construção do sistema (classes de negócio e seus relacionamentos), sendo necessários outros modelos complementares para prover informações suficientes para uma implementação final em código-fonte. Mesmo assim, já é possível observar que surgiram informações de implementação da solução que não existiam explicitamente no modelo anterior.

Figura 4 – Diagrama de classes do sistema escolar exemplo



Fonte: autor

Pode-se dar um passo adicional e criar então mais um modelo. Desta vez, um modelo de interface de usuário para cada classe persistente, ou seja, para as classes cujos objetos precisam ser armazenados em disco para persistirem após o término da execução do sistema e serem recuperados na próxima execução. No diagrama UML, essas classes estão indicadas com o estereótipo *«persistent»*. Trata-se de um modelo de outro domínio de descrição.

Se considerada a implementação do sistema em plataforma *web*, por exemplo, pode-se expressar o modelo da interface de usuário para cadastro de alunos através do código de uma página HTML com um formulário de entrada de dados, conforme Figura 5.

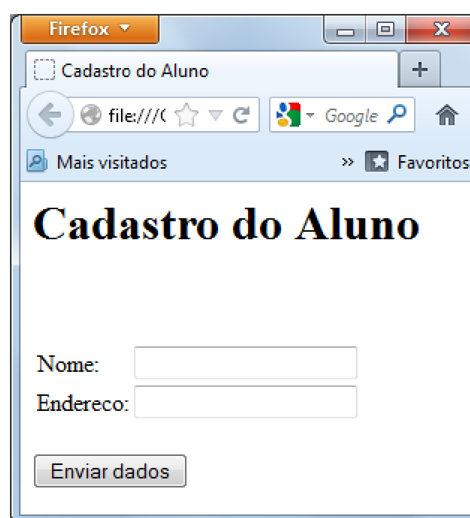
Esse modelo está à direita no gráfico, por ser expresso por meio de uma linguagem mais concreta e apresentar dependência da plataforma *web* (já sendo inclusive parte do código-fonte final do sistema). Também está abaixo, por estar em um domínio de descrição concreto, já que a interface está sendo definida com precisão. Portanto, esse modelo estaria dentro da área verde da Figura 3.

De volta ao modelo original do Quadro 1, outro analista poderia ter criado um diagrama de classes diferente da Figura 4 com base no mesmo modelo original. Poderia, por exemplo, ter criado uma classe *CPessoa* com os atributos *Nome* e

Endereco. As classes *CAluno* e *CProfessor* poderiam ser subclasses *CPessoa*, herdando esses atributos. Outro modo possível seria enxergar a classe *CProfessor* como classe associativa.

Figura 5 – Modelo para especificação da interface do cadastro de aluno
(a) Código HTML (b) Renderização da interface no navegador

```
<html>
   <head>
      <title>Cadastro do Aluno</title>
   </head>
   <body>
      <h1>Cadastro do Aluno</h1>
      <br /><br />
      <form name="CadastroAluno">
         <table border="0">
            <tr>
               <td>Nome:</td>
               <td><input type="text" /></td>
            </tr>
            <tr>
               <td>Endereco:</td>
               <td><input type="text" /></td>
            </tr>
         </table>
         <br />
         <input type="submit" />
      </form>
   </body>
</html>
```



(a) (b)

Fonte: autor

De maneira análoga, outro analista poderia ter criado o modelo de interface de usuário da Figura 5 contemplando botões e controles para realizar algumas operações relativas ao aluno, tais como efetuar a matrícula em um curso e inseri-lo em uma turma.

Pelo exemplo apresentado é possível observar que, se não houver uma forma de sincronização automática de modelos, um trabalho manual deve ser realizado caso haja alterações no modelo inicial, aumentando o custo, prazo e susceptibilidade a erros no projeto. Linguagens textuais naturais podem ser práticas para descrever modelos em qualquer nível, mas por serem difíceis de serem interpretadas por máquinas, elas não servem para a aplicação de transformação automática.

Se existissem funções com regras pré-definidas para gerar um modelo a partir de outros, seria garantida a repetibilidade das decisões quando se transformam modelos contendo instâncias do mesmo tipo. Se for necessário trocar uma regra por

outra mais adequada em determinado contexto, basta reexecutar a transformação e obtêm-se novos modelos a partir dos originais. Ou seja, as regras padronizam a maneira de construir modelos a partir de outros modelos. Se posteriormente for percebida a necessidade de melhoria e evolução nessas regras, o trabalho manual ficará centralizado na atualização dessas regras. Feito isso, é possível reaplicá-las de forma automática sobre os modelos desejados. Com a ausência desse mecanismo automático, os analistas ou desenvolvedores precisariam atualizar e checar manualmente todos esses modelos, tornando a manutenção do sistema mais cara e demorada.

A seguir, será retomado o conceito de metamodelos, necessário para o entendimento de como podem ser definidas as funções de mapeamento para realizar transformações automáticas de modelos.

2.5 METAMODELOS

Assim como as linguagens baseadas em texto (cadeias de símbolos), que sintaticamente podem ser definidas formalmente por meio de dispositivos matemáticos, tais como gramáticas (RAMOS; NETO; VEGA, 2009), as linguagens de modelagem também podem ser descritas com base em um tipo específico de modelo, os chamados metamodelos. Um metamodelo é, portanto, um artefato que define a sintaxe abstrata de uma linguagem de modelagem (ALANEN; PORRES, 2008). A notação na qual o metamodelo é expresso é definida por uma sintaxe concreta associada.

Definir metamodelos é um dos pontos-chave da MDE, sendo que existem diversas linguagens e formalismos propostos na literatura para isso, tais como o KM3 (JOUAULT; BÉZIVIN, 2006), o formalismo de Alanen e Porres (2008), entre outros. O padrão do OMG para descrever metamodelos é o MOF (OMG, 2016b), em conjunto com a *Object Constraint Language* (OCL) para estabelecer restrições. A sintaxe abstrata da UML, por exemplo, pode ser definida por meio de metamodelos escritos em MOF.

Diz-se que um modelo apresenta uma relação de conformidade com um metamodelo quando o primeiro é escrito de modo a respeitar os tipos e restrições do segundo (BORONAT; MESEGUER, 2008).

O conceito de metamodelo é de fundamental importância para descrever funções de mapeamento para transformações automáticas de modelos.

2.6 TRANSFORMAÇÃO AUTOMÁTICA DE MODELOS

Um mapeamento, ou transformação, é definido como uma aplicação de uma função de mapeamento. Uma função de mapeamento é uma função que toma como entrada um ou mais modelos e produz um modelo de saída (MELLOR et al., 2004). Ou seja, a função de mapeamento define a transformação.

O estabelecimento de funções de mapeamento permite a execução de transformações de interesse de forma automática, quantas vezes forem necessárias. Uma função de mapeamento pode ser escrita em uma linguagem de programação convencional ou em uma linguagem de transformação. Linguagens de transformação são linguagens especializadas para escrita de regras de transformação.

Em resumo, as abordagens para implementação de funções de mapeamento são (MELLOR et al., 2004):

- **Imperativa:** significa implementar a função de mapeamento em uma linguagem de programação procedural;
- **Baseada em arquétipo:** define uma função de mapeamento como um conjunto de *templates* que mistura código ou regras a um padrão de texto a ser gerado. Pode ser implementado como uma linguagem de acesso a dados que seleciona os elementos apropriados dos modelos de entrada, escolhe qual arquétipo utilizar para cada elemento e então decide como transformar esta informação em uma saída a ser inserida no modelo de saída. Esta abordagem é particularmente adequada para modelos expressos em linguagens textuais. A natureza sequencial e visível de uma cadeia de caracteres destaca a importância de ordem e aspectos de formatação. Um exemplo desta abordagem é o *eXtensible Stylesheet Language Transformations* (XSLT), uma linguagem de transformação de arquivos XML

(W3C, 1999). Serve, por exemplo, para transformar documentos XML de dados puros em exibições formatadas para o usuário em HTML.

- **Declarativa:** define uma função de mapeamento como um conjunto de regras de transformação, as quais especificam o que deve ser produzido, e não como. Esta abordagem é mais apropriada (em relação à abordagem imperativa) quando se tem a intenção de aplicar engenharia reversa nos modelos. Embora não garanta a reversibilidade, em outras palavras esta abordagem permitiria aplicar a função de mapeamento inversa f^{-1}, se existir, a partir da declaração de f. Exemplos de linguagens de transformação declarativas são *Query / View / Transformation* (QVT) (OMG, 2015b) e *ATL Transformation Language* (ATL) (JOUAULT; KURTEV, 2005).

Um detalhe fundamental é que as funções de mapeamento devem ser definidas sobre metamodelos, e não sobre modelos. Do contrário, a função de mapeamento só teria aplicação para a instância sobre a qual foi definida, e não sobre casos mais gerais. Em outras palavras, as regras de transformação que compõem uma função de mapeamento referenciam elementos de metamodelos, e não elementos de modelos.

Tomando o exemplo do sistema escolar, se fosse definida uma função de mapeamento usando a abordagem imperativa, ou seja, construindo um algoritmo em uma linguagem de programação procedural que aplicasse a seguinte regra: “Para a classe *CAluno*, criar um arquivo HTML contendo um formulário com campos de entrada para os atributos *Nome* e *Endereco* e um botão para envio dos dados”, esta regra só valeria para um pequeno trecho deste modelo em especial. Seria necessário definir diversas regras para cobrir cada classe presente no diagrama. Além disso, essas regras precisariam ser sempre revistas à medida que houvesse alteração nas classes, tais como a criação de novos atributos.

Definindo regras sobre os elementos previstos no metamodelo, ou seja, sobre classes e atributos, elas valem de forma geral para as instâncias desses elementos a que a regra se aplica. Ou seja, se a regra acima fosse reescrita da seguinte forma: “Para cada classe com o estereótipo *«persistent»*, criar um arquivo HTML contendo um formulário com campos de entrada para seus atributos públicos e um botão para

envio dos dados", ela teria muito mais valor prático, pois a função de mapeamento poderia ser aplicada a todas as classes do diagrama da Figura 4.

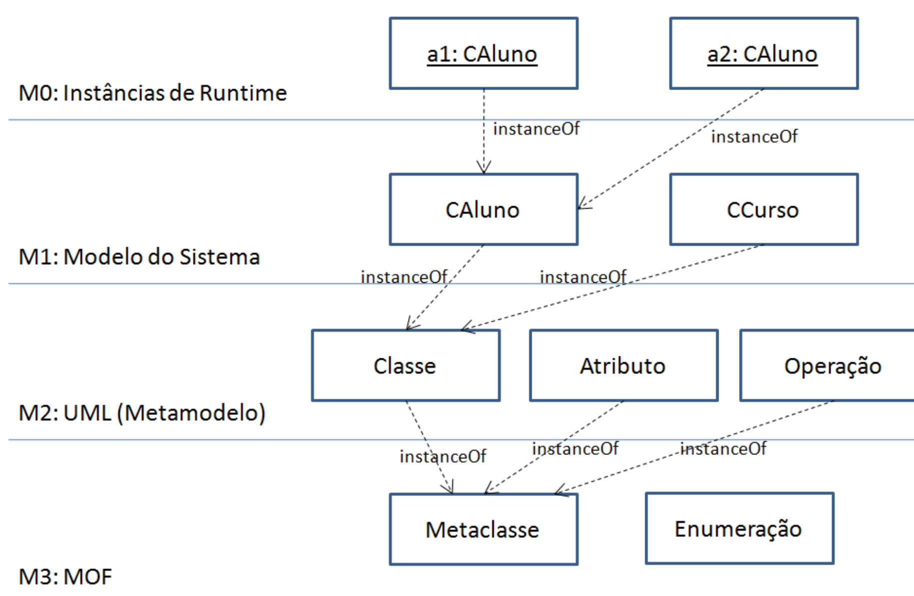
Essa constatação pode parecer óbvia, mas provê um encaminhamento importante para se estabelecer uma arquitetura de definição de linguagens conveniente e seus respectivos formalismos. Funções de mapeamento leem e escrevem modelos que podem estar expressos em linguagens arbitrárias que, por sua vez, são definidas através de metamodelos. Então, se houver uma maneira conceitualmente única para se definir metamodelos, as funções de mapeamento poderão ser definidas sobre um arcabouço comum geral.

Este raciocínio leva a uma arquitetura conceitual conhecida como arquitetura das quatro camadas (MELLOR et al., 2004).

2.7 A ARQUITETURA DAS QUATRO CAMADAS

A arquitetura das quatro camadas é um modelo conceitual que contempla objetos de tempo de execução, modelos, metamodelos e um último nível conhecido como "meta-metamodelo", como pode ser visto na Figura 6.

Figura 6 – A arquitetura das quatro camadas



Fonte: Mellor et al. (2004) (adaptado)

Essa arquitetura permite classificar e visualizar precisamente a qual nível descritivo cada elemento pertence, bem como identificar suas relações horizontais e verticais, facilitando a leitura e tratamento desses elementos por programas.

A camada M0 refere-se aos dados de um sistema em tempo de execução. No exemplo do sistema escolar, seriam objetos instanciados de *CAluno*, *CCurso*, etc. durante a execução do software. Registros em tabelas de um banco de dados relacional que armazenam esses objetos de forma persistente também pertencem a esta camada.

A camada M1 diz respeito aos modelos do sistema considerado: diagramas UML de classes, definições de tabelas de um banco de dados relacional, diagramas de casos de uso, códigos-fonte, etc. É o nível no qual a modelagem de software ocorre e, portanto, no qual trabalham os analistas e desenvolvedores. As transformações de modelos da Figura 2 e Figura 3 ocorrem dentro deste nível.

A camada M2 refere-se aos metamodelos, isto é, os metadados que capturam as linguagens de modelagem. Neste nível estão as definições da UML, ou seja, as especificações de seus diagramas e elementos: classes, atributos, operações, casos de uso, atores, etc. Definições sintáticas e semânticas das linguagens de programação também ocorrem neste nível. Portanto, é aqui que atuam os desenvolvedores de linguagens. As funções de mapeamento também são definidas neste nível, entre os metamodelos. Dessa forma, como foi visto, passam a ter utilidade geral pois podem ser aplicadas sobre quaisquer instâncias desses metamodelos (ou seja, os modelos de software do nível M1).

A camada M3 corresponde aos “meta-metamodelos”, que servem para descrever os metamodelos. Este nível contém apenas os conceitos mais simples requeridos para capturar modelos e metamodelos, sendo uma constante para suportar todas as possibilidades de modelagem das camadas acima. O OMG dedica esforços de padronização neste nível, e com isso criou o MOF (OMG, 2015a).

Embora a nomenclatura “meta-metamodelo” aparente ser informal, ela é utilizada na literatura. Por isso, daqui em diante, ela será utilizada sem aspas ao longo do texto.

Com relação à camada M0, embora seus elementos estejam relacionados ao tempo de execução de um programa, observa-se que não necessariamente se tratam de elementos executáveis. O exemplo apresentado acima sobre as linhas de uma

tabela de um banco de dados relacional, que estão classificadas na camada M0, ilustra esse fato. Registros específicos de uma tabela ocorrem em tempo de execução, mas registros em si não têm uma interpretação executável. Em outros casos, elementos da camada M1 possuirão interpretação executável, tais como em máquinas de estado ou redes de Petri (nas quais existe o conceito de estado corrente, passo de execução, etc.). De qualquer modo, pressupõe-se uma máquina de execução subjacente para criar, modificar, eliminar e operar sobre as instâncias na camada M0, sendo guiada conforme as especificações do modelo da camada M1. No caso geral, tal máquina terá a capacidade de uma máquina de Turing.

Visto à luz da arquitetura das quatro camadas, o objetivo deste trabalho, descrito na seção 1.3, consiste em propor um formalismo para a camada M3 e, com base nele, definir os entes da camada M2 (metamodelos) e M1 (modelos), bem como estabelecer relações entre instâncias de diferentes camadas, operações e propriedades sobre os mesmos, tendo a MDE como pano de fundo.

Um erro comum que se pode cometer à primeira vista é enxergar as transformações da MDE ocorrendo entre as camadas. Na verdade, as transformações da MDE ocorrem dentro da mesma camada. Em particular, as transformações de interesse para desenvolvimento de software (Figura 2 e Figura 3) ocorrem na camada M1. A relação entre as camadas é de instância e tipo. O nível M3 possui elementos para descrever metamodelos do nível M2, que por sua vez oferecem meios para descrever modelos do nível M1. Esses últimos descrevem programas cujos elementos de tempo de execução são classificados na camada M0.

Tal arquitetura implica que o conjunto de decisões tomadas em uma certa camada direciona as decisões que podem ser tomadas na camada superior. Por exemplo, os elementos presentes em um metamodelo (M2) determinam e restringem a existência e características dos elementos de modelos (M1) conformes a tal metamodelo. Ao elaborar um modelo, o desenvolvedor não pode instanciar conceitos que não estão previstos no metamodelo utilizado. Ademais, as instâncias só podem carregar informações previstas por seu tipo na camada inferior adjacente. Um sistema de tipos tem como seu maior propósito evitar problemas sobre respresentações, bem como proibir situações em que esses problemas aparecem. Nesse contexto, um tipo pode ser visto como uma capa que protege uma representação não-tipada subjacente, impedindo violações e usos não previstos de tal representação

(CARDELLI; WEGNER, 1985). No exemplo das linhas de uma tabela de um banco de dados relacional, cada linha possui uma representação “bruta” em tempo de execução, que seria a resperentação não-tipada subjacente. Os elementos de M1 que especificam a tabela relativa a esta linha funcionam como tipos que permitem interpretar tal representação com respeito ao conteúdo de cada campo ou coluna, bem como verificar se ela é válida e identificar violações.

De modo geral, a existência da camada Mn direciona e restringe as decisões subsequentes na camada Mn-1. Em particular, M3 direciona as decisões no momento da elaboração de uma linguagem de modelagem através do seu metamodelo. M2, por sua vez, direciona as decisões referentes à arquitetura no modelo, pois impõe uma estrutura de tipos estáticos que irá determinar as possibilidades de como o modelo pode ser organizado. Isso significa que um modelo na camada M1 pode ser validado em tempo de compilação, em que se pode verificar o respeito aos tipos e restrições impostas pela camada M2 antes que o modelo seja utilizado em uma transformação ou execução direta. Esse conceito de conformidade será revisto em detalhes na seção 4.9. Já M1, por fim, determina os conceitos que podem existir em tempo de execução de um sistema, isto é, na camada M0. De acordo com a visão de Cardelli e Wegner (1985), os elementos de M1 funcionariam como “capas” para interpretar, moldar e impedir violações sobre os elementos que existem em tempo de execução em M0, os quais são instanciados dinamicamente.

Como as funções de mapeamento devem ser definidas sobre metamodelos, então a existência de conceitos e blocos básicos comuns na camada M3 (que são utilizados para definir diferentes metamodelos da camada M2) permitiria a definição de funções de mapeamento entre metamodelos através de técnicas comuns. Se surgirem novos metamodelos, ou seja, novas linguagens de modelagem, as mesmas técnicas ainda poderiam ser usadas para criar funções de mapeamento que atuam sobre modelos escritos nessas novas linguagens de modelagem. Daí a importância de existir uma especificação de linguagem para a camada M3 com sintaxe e semântica suficientemente simples para facilitar a sua implementação e utilização, mas ao mesmo tempo poderosa para permitir que novos metamodelos surjam conforme necessidades práticas.

2.8 MODELOS DE MARCAÇÃO

Às vezes, uma função de mapeamento requer informações adicionais que não estão presentes nos modelos de entrada, não porque essas informações deixaram de ser inseridas pelo desenvolvedor, mas sim porque nem mesmo estão previstas para serem expressas nos modelos de entrada. Define-se como marcação uma informação adicional que pode complementar um modelo, mas que não faz parte dele. Em outras palavras, marcações são extensões leves e não intrusivas de modelos que capturam dados adicionais requeridos para uma função de mapeamento, sem poluir ou alterar o modelo original, preservando-o (MELLOR et al., 2004). Um conjunto de marcações sobre um modelo pode ser comparado a um plástico transparente com informações complementares colocado sobre um desenho em papel. O plástico adiciona conteúdo ao desenho original, sem alterá-lo. Mas, quando aplicado, enxerga-se o desenho (modelo) original com modificações.

As marcações são um dos tipos possíveis de informações adicionais utilizadas no processo clássico de transformação de modelos da MDE conforme ilustra a Figura 2.

Um modelo de marcações define uma lista de nomes, tipos e valores padrões das marcações que podem ser aplicadas sobre outros modelos e seus elementos. Os modelos de marcações estão para os conjuntos de marcações assim como os metamodelos estão para os modelos (MELLOR et al., 2004)[1].

Uma função de mapeamento que toma classes persistentes para gerar interfaces de usuário para edição em HTML, por exemplo, poderia contar com uma marcação indicando a cor de fundo a ser utilizada na interface gerada. Essa informação não faz parte do diagrama de classes, mas é mantida junto com o modelo e ao mesmo tempo não dentro dele, para prover informação necessária para a função de mapeamento.

Em outro exemplo, um sistema bancário que deve lidar com cerca de 100.000 clientes e 500.000 contas pode usar essa informação para definir o número de espaços pré-alocados para cada tabela em um banco de dados, e também para gerar código de acesso a dados de maneira otimizada. Essa informação pode ser

---

[1] Para manter o padrão da nomenclatura de acordo com a analogia, uma alternativa seria que o modelo de marcação fosse denominado *metamodelo de marcação*, e que o conjunto de marcações fosse denominado *modelo de marcações*. Porém, isso não ocorre, e utiliza-se aqui as mesmas nomenclaturas da literatura.

capturada como uma marcação *InstanceQuantity* do tipo número inteiro associada a cada classe persistente, a ser definida pelo modelador do sistema como uma estimativa (MELLOR et al., 2004).

2.9 REFINAMENTO DE MODELOS

Diferentes modelos de um mesmo sistema de software usualmente coexistem e evoluem de forma independente (XIONG et al., 2007), mesmo que apresentem dependências que precisam ser sincronizadas.

Ao se gerar um modelo via transformação automática a partir de outros modelos de entrada, pode ser desejável editar partes desse modelo diretamente para introduzir mais informações, como parte do processo de criação do software. Por exemplo, ao gerar um código-fonte em Java com definições estruturais de classes a partir de um diagrama de classes UML, o desenvolvedor pode escrever as implementações dos métodos de cada classe diretamente no arquivo de código-fonte Java, que eram informações não existentes originalmente no modelo do diagrama de classes. Entretanto, se alterações são feitas no diagrama de classes original e uma nova transformação automática em código Java é reaplicada, não é desejável que as implementações introduzidas pelo desenvolvedor sejam sobrescritas.

O ato de introduzir informações manualmente em um modelo gerado automaticamente é denominado *model elaboration* em inglês (MELLOR et al., 2004), aqui traduzido como refinamento de modelos[2].

Conclui-se, então, que um requisito importante para descrever funções de mapeamento é que seja prevista alguma maneira de preservar informações introduzidas diretamente no modelo alvo, sem que a geração da próxima versão do mesmo modelo as sobrescreva.

---

[2] Optou-se pela palavra *refinamento* em vez de *elaboração* para evitar que ela seja confundida com a criação de modelos, já que a palavra *elaboração* é mais próxima deste significado.

2.10 PROCESSOS MDE

Os sistemas atuais são, em geral, complexos o suficiente para se construir utilizando transformação de modelos em um único passo (*hop*). Múltiplos passos, encadeando e compondo diversas funções de mapeamento, utilizando diversos tipos distintos de modelos que se combinam, podem ser necessários.

Um processo MDE consiste em uma composição de funções de mapeamento que define o caminho dos modelos de origem até os artefatos finais, que podem ser o software pronto para ser compilado ou executado (no caso de um processo MDE completo), ou simplesmente modelos mais próximos da implementação final a serem trabalhados pelo desenvolvedor (no caso de um processo MDE parcial).

Ao se definir as funções de mapeamento que compõem o processo MDE, conjuntamente estão sendo definidos também os metamodelos do processo e, por consequência, os tipos de modelos que podem ser criados, transformados e gerados.

Uma vez definido o processo MDE de interesse, o mesmo deve possuir suporte prático de ferramentas para ser utilizado em um processo de desenvolvimento de software real.

2.11 APLICAÇÃO PRÁTICA DA MDE

Apesar da possibilidade de vários benefícios, aplicar a MDE na prática ainda não é uma tarefa fácil e consolidada, havendo obstáculos e desafios relacionados à imaturidade de ferramentas (QUINTERO et al., 2012), falta de suporte teórico (YANG; LIU; WANG, 2012) e metodologias (YANG; LIU; LI, 2014). Para a MDA especificamente, que se refere às tecnologias do OMG, a proposta de aplicação sugerida é através de uma cadeia de ferramentas que suportem os padrões da UML, MOF, XMI, QVT, etc., de modo a compor um ambiente de desenvolvimento orientado a modelos (MELLOR et al., 2004). Kolovos et al. (2013) cita diversas ferramentas para MDE com distintas finalidades: gerenciamento de repositórios de modelos, controle de versão de modelos, edição e transformação, etc., servindo como boa referência geral.

Organizações que desenvolvem software de acordo com a MDE têm integrado e adaptado ferramentas por conta própria (MOHAGHEGHI et al., 2008). Em concordância, Karna, Tolvanen e Kelly (2009) observam que as soluções mais efetivas para aplicar MDE são usualmente especializadas para uma única companhia. Destaca-se a utilização de DSMLs para resolver mais eficientemente problemas em domínios específicos. Há poucas evidências empíricas sobre a aceitação da MDE na indústria. Mohagheghi et al. (2013) apresenta um estudo empírico para investigar o estado da prática de aplicação da MDE neste contexto, coletando dados de múltiplas fontes, tais como avaliação de ferramentas, entrevistas e pesquisas. São destacados três fatores importantes como determinantes para a adoção da MDE: utilidade percebida, facilidade de uso e maturidade das ferramentas.

Quanto às pequenas empresas, de acordo com Cuadrado, Izquierdo e Molina (2014), o relato de experiências com MDE é ainda raro, apesar de elas representarem uma parcela significativa das empresas de software.

Em resumo, a utilização prática da MDE nas organizações, de forma geral, ainda ocorre de maneira incipiente, difusa e não consolidada.

Por outro lado, estudos mostram que a MDE, quando aplicada, pode aumentar a produtividade em um fator de 10 (JAAKSI, 2002) (KARNA; TOLVANEN; KELLY, 2009). Mais recentemente, o trabalho de Papotti et al. (2013) obtém uma redução de aproximadamente 90% no tempo de desenvolvimento de software em um experimento controlado, concordando com o fator acima. Embora não seja possível generalizar os resultados para qualquer caso de aplicação da MDE, esta parece ser a ordem de grandeza do ganho proporcionado por esta abordagem.

Apesar de a adoção da MDE na indústria estar longe do sucesso, a MDE vem crescentemente ganhando aceitação na comunidade da Engenharia de Software (CUADRADO; IZQUIERDO; MOLINA, 2014).

2.12 ABORDAGENS DE MODELAGEM

2.12.1 **Linguagens de modelagem de propósito geral (GPML)**

Um processo MDE candidato a ter sucesso deve contar com metamodelos que permitam modelar a aplicação em um alto nível de abstração e também com funções de mapeamento capazes de transformar os modelos em artefatos próximos da implementação final, se não completos, tal como código-fonte pronto para ser compilado.

Uma das abordagens para isso é a utilização de uma linguagem de modelagem de propósito geral, ou *General Purpose Modeling Language* (GPML), tal como a UML, e utilizar uma biblioteca de funções de mapeamento prontas e padronizadas, que confiam em uma semântica bem definida dos elementos da linguagem. A UML, entretanto, depende de linguagem natural para descrever certos modelos e possui ambiguidades, não sendo adequada para isso em sua plenitude (MARCHETTI; 2012).

Outro problema relacionado à UML está associado à sua complexidade, devido à grande quantidade de conceitos em cada um de seus diagramas. Na prática, apenas uma pequena porcentagem dos conceitos da UML é usada na resolução de problemas reais (PASTOR; MOLINA, 2010). Por isso, reduzir e racionalizar a linguagem para os conceitos que são realmente necessários para especificar sistemas é uma iniciativa positiva. Trabalhos mais recentes concordam com esta análise (FONDEMENT et al., 2013).

Iniciativas como a xUML (MELLOR; BALCER, 2002) e a fUML (OMG, 2016c) têm por objetivo estabelecer subconjuntos da UML com semântica precisamente definida e padronizada de modo que permitam a transformação de seus modelos em código-fonte com comportamento definido, sem ambiguidades ou indefinições. Outras soluções encontradas na literatura visam resolver problemas específicos de transformação de modelos para certos diagramas ou elementos específicos da UML, tais como Gessenharter e Rauscher (2011), que propõem a transformação de atividades UML em código-fonte Java para controle de fluxo de aplicações.

Wagner et al. (2006), por sua vez, abordam a utilização de máquinas de estado para modelagem de software, com exemplos de aplicação em diferentes domínios.

No entanto, a abordagem de usar uma linguagem de modelagem de propósito geral com funções de mapeamento preestabelecidas e padronizadas pode não ser adequada para prover o nível de abstração necessário para qualquer classe de aplicação como uma solução geral. Nesse caso, vale considerar a adoção de linguagens de modelagem mais específicas.

2.12.2 **Linguagens de modelagem específicas de domínio (DSML)**

Esta abordagem parte do pressuposto de que não existe uma linguagem de modelagem de propósito geral capaz de resolver todos os problemas de forma eficiente para qualquer classe de aplicação, nem mesmo a UML.

Surgem então, nesse contexto, as linguagens de modelagem específicas de domínio, ou *Domain Specific Modeling Languages* (DSML). Uma linguagem específica de domínio é projetada para ser útil para um conjunto limitado de tarefas, em contraste com uma linguagem de propósito geral, cujo pressuposto é ser útil para muitas tarefas genéricas e cruzar múltiplos domínios de aplicação (JOUAULT; BÉZIVIN, 2006) (JÉZÉQUEL et al., 2012).

Um processo MDE para criar um aplicativo de negócio, por exemplo, tende a envolver modelos de diferentes tipos de um processo MDE para criar um jogo. Em outras palavras, os metamodelos e funções de mapeamento adequados a um domínio de aplicação tendem a ser diferentes dos metamodelos e funções de mapeamento adequados a um domínio de aplicação distinto.

Um domínio pode ser um domínio de aplicação (p. ex., seguradoras e empresas de saúde), um domínio técnico (p. ex., dados, lógica de negócio e fluxos de trabalho) ou um domínio organizacional (p. ex., vendas e serviços ao cliente). Pode-se interpretar um domínio como uma família de sistemas de software exibindo funcionalidade similar, sendo a funcionalidade desta família codificada como conhecimento de domínio (WU et al., 2010).

As linguagens específicas de domínio podem ser padronizadas para um domínio, ou podem ser criadas sob demanda. Isto é, uma organização poderia criar uma linguagem de domínio específica para sua área de interesse, considerando também as especificidades da organização, e utilizá-la em seus processos MDE. Tal

abordagem foi utilizada com sucesso em Karna, Tolvanen e Kelly (2009), em que a organização criou uma linguagem de modelagem própria e funções de mapeamento específicas para desenvolvimento de software para relógios esportivos, obtendo um retorno de investimento significativo.

Segundo Kolovos et al. (2013), a MDE advoga soluções específicas de domínio e permite sua utilização combinada com linguagens de propósito geral. Isso permite dispor de arcabouços e ferramentas mais adequados para atacar cada tipo de problema e evitar a armadilha das ferramentas *Computer Aided Software Engineering* (CASE) que se propõem a resolver todos os problemas.

2.13 RESUMO DAS DEFINIÇÕES E CONCEITOS PRINCIPAIS SOBRE MDE

A Figura 7 apresenta as relações entre os principais conceitos sobre MDE apresentados neste capítulo.

Figura 7 – Conceitos principais da MDE e suas relações

Fonte: autor

2.14 DESAFIOS E TÓPICOS DE PESQUISA DA MDE

A MDE é uma área de pesquisa bastante ampla, contendo diversos tópicos de pesquisa mais específicos. Alguns desses tópicos são: criação e extensão de DSMLs, computação de transformações de modelos e modelagem colaborativa (incluindo repositórios de modelos, modelagem *online* e versionamento de modelos). Kolovos et al. (2013) apresentam um bom panorama dessas áreas, dando enfoque ao aspecto de escalabilidade da MDE. São resumidos o estado da arte de cada tópico, bem como as direções de pesquisa observadas. Para a modelagem colaborativa, também é importante o tópico de pesquisa sobre sincronização de modelos, que será retomado com mais detalhes na seção 5.3.

A formalização dos conceitos sobre MDE, tópico de pesquisa no qual este trabalho se insere, também é um problema em aberto. O Capítulo 3 apresenta uma revisão bibliográfica mais detalhada sobre o assunto, destacando as lacunas nas soluções existentes e fazendo a ponte para o Capítulo 4, onde se inicia a contribuição original desta pesquisa.

Outra área recente e importante da MDE é denominada *Models at Runtime*, e refere-se ao estudo do uso de modelos em tempo de execução, ou seja, a utilização de representações internas de um sistema, ou parte dele, durante sua execução para tomada de decisões. Uma revisão sistemática sobre o assunto é apresentada por Szvetits e Zdun (2013). Esse assunto é retomado na seção 5.2, onde os conceitos apresentados no Capítulo 4 são utilizados para formalizar modelos executáveis e conceitos relacionados. Outras referências da área são apresentadas.

3 FORMALISMOS DE SUPORTE À METAMODELAGEM E MDE

3.1 INTRODUÇÃO

Metamodelagem continua a ser um tópico extensivamente pesquisado com muitas abordagens para formalização (JACKSON; LEVENDOVSZKY; BALASUBRAMANIAN, 2013). Conforme citado, a base para metamodelagem do OMG é o MOF (OMG, 2015a). No entanto, alguns autores pesquisam alternativas ao MOF como suporte teórico para metamodelagem e MDE em geral, elencando desvantagens, limitações e inadequações do MOF como motivação para isso, conforme será visto na subseção 3.2.3.

Essa busca por alternativas ao MOF levou à criação de outros formalismos tais como o KM3 (JOUAULT; BÉZIVIN, 2006), o formalismo de Alanen e Porres (2008), NEREUS (FAVRE, 2009), a abordagem de Jackson, Levendovszky e Balasubramanian (2013), entre outros.

Neste capítulo, serão apresentados o MOF e os formalismos supracitados encontrados na literatura, identificando as vantagens e desvantagens de cada um e servindo como base para a posterior apresentação do SBMM.

Considerando o caráter de revisão bibliográfica deste capítulo, alguns trechos das seguintes seções, no que se refere às definições apresentadas pelos trabalhos citados, consistem na tradução de tais definições, evitando-se assim o risco de introduzir perdas ou imprecisões pela reescrita com outras palavras.

3.2 MOF

3.2.1 **Conceitos principais**

MOF é uma linguagem enquadrada na camada M3, que permite definir metamodelos, ou seja, especificações de linguagens de modelagem, as quais se relacionam à camada M2. É uma das tecnologias que compõem a infraestrutura da MDA do OMG. A última versão considerada neste trabalho é a 2.5, publicada em 2015, e é especificada por um conjunto de sete documentos (OMG, 2015a):

- MOF 2 Core Specification;
- MOF 2 XMI Mapping;
- MOF 2 Facility and Object Lifecycle;
- MOF 2 Versioning and Development Lifecycle;
- MOF 2 Query/View/Transformations;
- MOF Model To Text;
- Object Constraint Language (OCL);

A especificação principal é a MOF 2 Core Specification (OMG, 2015a), que descreve a arquitetura da linguagem, provas de conceito e suas capacidades principais. O MOF é apresentado através de descrições textuais e gráficas, usando uma combinação de linguagens, a saber: um subconjunto da UML, uma linguagem de restrições de objetos (OCL) e linguagem natural.

As outras seis especificações mostram como o MOF se relaciona com outras tecnologias que fazem parte da infraestrutura da MDA. *XML Metadata Interchange* (XMI), por exemplo, é uma forma padronizada de salvar modelos MOF em arquivos XML, o que possibilitaria integração entre diferentes ferramentas.

Foge ao escopo deste trabalho apresentar uma descrição detalhada sobre o MOF; contudo, a seguir são apresentados alguns conceitos gerais e a forma com que ele é especificado.

Um dos aspectos mais importantes é que a especificação do MOF faz reuso de muitos elementos da biblioteca de infraestrutura da UML 2. Ou seja, importa explicitamente conceitos fundamentais como classes, propriedades, operações e pacotes, fazendo uso desses conceitos do ponto de vista da camada M3. Sobre esses conceitos importados, o MOF é definido por meio de diagramas UML de classes, nos quais os conceitos e capacidades são hierarquizados com a utilização de pacotes (*packages*) e relações de dependência de importação (*import*) e mesclagem (*merge*).

A especificação estabelece três capacidades para o MOF:

- **Reflexão:** habilidade de um metamodelo escrito em MOF de ter acesso aos seus próprios metadados. Por exemplo, uma metaclasse poderia ter acesso a suas próprias definições, tais como seu nome ou lista de operações com seus respectivos argumentos. Esse conceito não é exclusivo da camada M2. Linguagens de programação comuns orientadas a objeto, tais como Java e Free Pascal, oferecem esse recurso na camada M1;
- **Identificadores:** capacidade de atribuir um identificador único a cada objeto do metamodelo escrito em MOF, sem precisar contar com atributos cujos valores estão sujeitos a mudanças;
- **Extensão:** consiste em um mecanismo simples de estender um metamodelo escrito em MOF com pares nome/valor. Está de acordo com a definição vista para modelos de marcação na seção 2.8.

O MOF é dividido em duas partes: *Essential MOF* (EMOF) e *Complete MOF* (CMOF). O CMOF designa o MOF em sua totalidade. O EMOF é um subconjunto do CMOF que corresponde proximamente aos recursos encontrados nas linguagens de programação orientadas a objeto e no XML. A motivação do EMOF é reduzir a barreira de entrada para desenvolvimento e integração de ferramentas dirigidas a modelos, que dão suporte prático para utilização da MDA (OMG, 2015a).

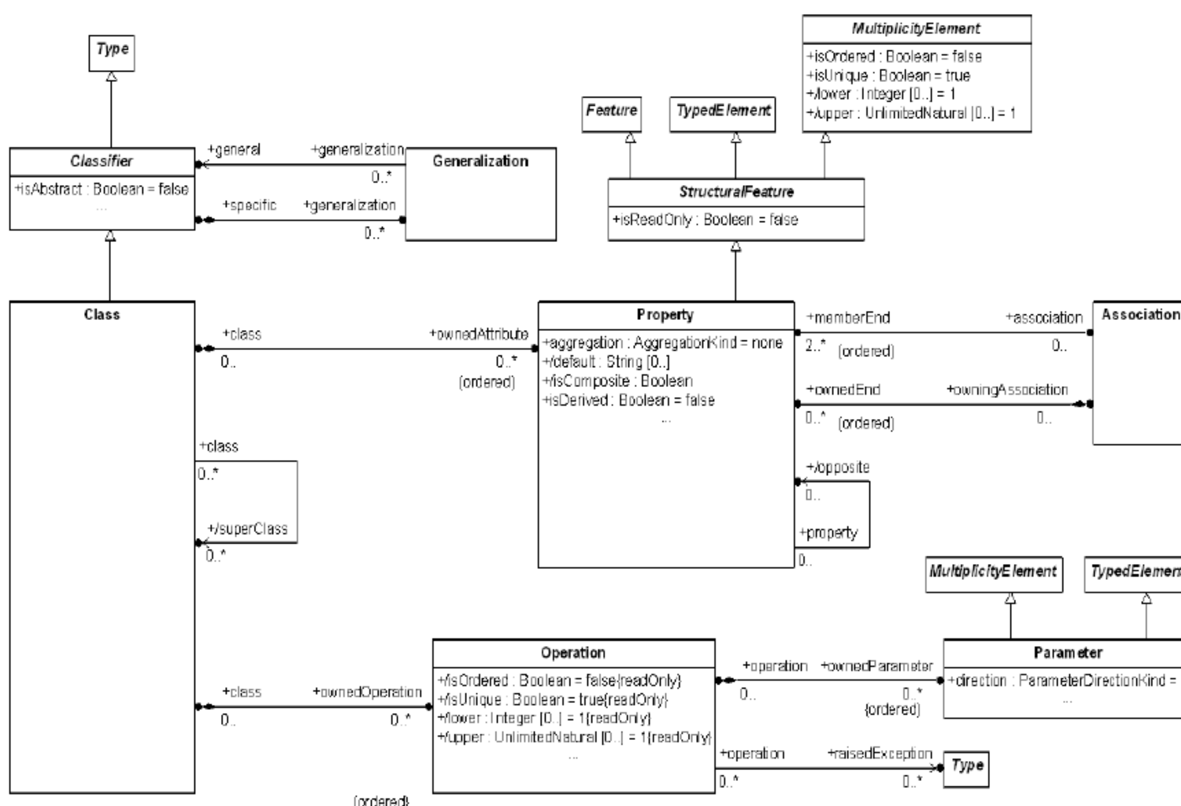
O EMOF é descrito inicialmente através de modelos de classes e tipos primitivos de dados importados da UML 2. Mas, segundo a própria especificação, o suporte completo do EMOF requer que ele seja descrito utilizando-se ele mesmo, removendo mesclagem de pacotes e redefinições que possam ter sido especificadas no modelo UML (OMG, 2015a). Apresenta-se, então, um modelo completo do EMOF autossuficiente, descrito por meio dele mesmo, sem dependências com relação a quaisquer outros pacotes ou capacidades de metamodelagem não suportadas por si próprio. Esse modelo consolidado do EMOF tem por intenção permitir o desenvolvimento de ferramentas de apoio à MDA com raízes no próprio EMOF sem requerer uma implementação mais completa do CMOF.

Alguns recursos não presentes no EMOF, mas presentes no CMOF, são: visibilidade de propriedades e redefinição de propriedades em subclasses (OMG, 2015a).

EMOF e CMOF também são pontos de conformidade da especificação. Assim, pode-se rotular determinada ferramenta como conforme ao EMOF (*EMOF-compliant*), por exemplo, e assim se sabe que atende plenamente ao EMOF.

Para ilustrar as explicações apresentadas, a Figura 8 apresenta um dos diagramas que definem o EMOF através dele mesmo (OMG, 2015a). Nessa figura, está descrita a estrutura de classes tradicional da orientação a objetos (uma classe é composta por múltiplas propriedades e operações, podendo participar de associações binárias). Cada bloco do diagrama é uma metaclasse, instância do próprio elemento *Class* do EMOF, e cada linha é uma associação, instância do próprio elemento *Association* do EMOF. Essa é a essência para o entendimento de que o EMOF pode ser descrito nele mesmo, justificando a arquitetura de quatro camadas do OMG na medida em que não são necessárias mais camadas a partir da M3. Assim como o EMOF, o CMOF também pode ser descrito por meio dele mesmo.

Figura 8 – Parte da especificação do EMOF definida nela mesma



Fonte: OMG (2015a)

A especificação do MOF também utiliza a linguagem OCL para estabelecer restrições em seus modelos, tais como unicidade de valores de propriedades e valores permitidos de propriedades em função de outros aspectos do modelo.

3.2.2 **Aspectos teóricos da definição do MOF**

De acordo com Bézivin e Gerbé (2001), a forma com a qual o MOF é definido origina confusões. Uma delas é o fato de se aproveitar elementos básicos da UML para reutilizar no MOF, estando a UML na camada M2 e o MOF na camada M3. Uma vez que, conceitualmente, a camada Mn é definida através de instanciação de conceitos da camada Mn+1, essa forma de definir o MOF parece violar a arquitetura. Essa forma de definir o MOF persiste em sua especificação (OMG, 2015a).

O primeiro ponto para explicar que a violação não ocorre é que os elementos da UML são copiados para a camada do MOF, e não instanciados. Essa cópia significa que os elementos são replicados para a outra camada e passam assim a pertencer de forma independente a ela, devendo ser enxergados sob a ótica de uma camada descritiva distinta.

A cópia é feita para não ser necessário redefinir os conceitos básicos de classe, propriedade, operação, etc. na camada M3, aproveitando as definições que já existem na camada M2 pela UML em sua especificação. Ou seja, uma vez que esses elementos são replicados, deve-se enxergá-los como cópias distintas nas duas camadas. Para evitar dúvidas, pode-se por exemplo chamar o elemento *Class* da camada M3 de *MOF::Class* e o da camada M2 de *UML::Class*. O *UML::Class*, por sua vez, poderia ser redescrito como uma instância de *MOF::Class*, de acordo com a arquitetura das quatro camadas. Indo além, no modelo de software do sistema escolar, a classe *CCurso* da camada M1 é uma instância de *UML::Class*.

Pode-se encontrar na literatura a utilização da palavra “importação” no lugar de “cópia”. Ou seja, diz-se que o MOF importa elementos da UML. Essa terminologia gera mais confusão, pois uma relação de importação significa que os elementos importados podem ser referenciados em outro módulo (SZYPERSKI, 1992). Quando se usa o conceito de metaclasse para compor metamodelos, embora sintaticamente parecido, não se trata do conceito de classe para compor modelos. E, portanto, não

é adequado dizer que houve uma importação do conceito de classe para ser usado como metaclasse no contexto de outra camada.

Ou seja, embora seja utilizado o conceito de classe em camadas distintas, tratam-se de interpretações diferentes. O conceito de classe quando enxergado na camada M1 (parte de um modelo de software específico) não é o mesmo conceito de classe na camada de metamodelos M2 (definido pela UML), que também não é o mesmo quando comparado com a classe da camada M3 (definido pelo MOF) (SIQUEIRA, 2011). O relacionamento entre elas é de instância e tipo, representado pela seta *instanceOf* no diagrama da arquitetura das quatro camadas da Figura 6.

Em resumo, a cópia (e não importação nem instanciação) que o MOF faz das classes da UML é para reutilizar suas definições, que já existiam previamente, e a partir daí obtém-se *MOF:Class* que pode ser usada para redefinir *UML::Class* como instância da primeira.

Para não haver dúvidas sobre as camadas que estão sendo referidas, utiliza-se nomenclaturas diferentes para os conceitos de classe e modelo. Em M1, os nomes são esses mesmos. Em M2, diz-se metaclasses e metamodelos, respectivamente. Em M3, meta-metaclasses e meta-metamodelos.

Mesmo quando se observa o modelo autodescrito do EMOF ou CMOF, sem depender da UML, também pode causar estranheza o fato de parecer uma recursão sem base. É como estabelecer uma definição que faz referência a ela mesma. Por um lado, se for suposto que a camada Mn deva sempre ser descrita por uma camada Mn+1, a arquitetura teria então infinitas e impraticáveis camadas. Por outro lado, ao conseguir que uma camada Mn seja usada para definir ela mesma, não são necessárias mais camadas a partir dela.

Linguagens textuais são suportadas por extensa e consolidada teoria, que provê formalismos para sua definição e tratamento. Podem-se vislumbrar conceitos da teoria dentro da arquitetura das quatro camadas.

Um código-fonte de um programa expresso em uma linguagem de programação qualquer (C++, Java, etc.) se encaixa no nível M1, representando o modelo de um sistema. Esse sistema em execução se enquadra no nível M0. A linguagem de programação em si, cuja sintaxe pode ser descrita por uma gramática (usualmente livre de contexto com apoio de tabelas de símbolos) é classificada no nível dos

metamodelos, ou seja, M2. Gramáticas livres de contexto podem ser definidas através de notação *Extended Backus-Naur Form* (EBNF), por exemplo. A notação EBNF, na camada M3, pode ser descrita através dela mesma conforme o Quadro 2.

Por isso, o fato de o MOF poder ser definido através dele mesmo não é um problema. É uma situação já conhecida no campo das linguagens e gramáticas. Mas, do ponto de vista prático, há algumas considerações.

A questão se assemelha também a escrever um compilador de linguagem C em linguagem C. Se já se dispõe de um compilador de linguagem C inicial (escrito em outra linguagem previamente disponível ou em *assembly*, código de máquina, etc.), pode-se usá-lo para compilar o novo compilador. Se o compilador inicial for descartado do cenário, surge a dúvida sobre como se conseguiu compilar o compilador da primeira vez, já que era necessário ter um compilador de linguagem C. Essa técnica, referente a escrever o primeiro compilador em uma linguagem já disponível, normalmente mais primitiva, e depois escrever compiladores mais avançados na própria linguagem, criando-se linguagens de mais alto nível, é conhecida como *bootstrapping*. Foi muito utilizada nos primórdios da computação quando não se tinha linguagens de alto nível disponíveis. A técnica permitiu o aumento sucessivo no nível de abstração das linguagens de programação, partindo-se do ponto inicial onde se programava os computadores diretamente a partir de código de máquina.

Quadro 2 – EBNF descrito em EBNF

```
lower ⇐ a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
upper ⇐ A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
digit ⇐ 0|1|2|3|4|5|6|7|8|9
special ⇐ -|_|"|#|&|'|(|)|*|+|,|.|/|:|;|<|>
character ⇐ lower | upper | digit | special
empty ⇐ " "

lhs ⇐ lower {[_] lower}
option ⇐ "[" rhs "]"
repetition ⇐ "{" rhs "}"
sequence ⇐ empty | {character | lhs | option | repetition}
rhs ⇐ sequence {"|" sequence}

ebnf_rule ⇐ lhs "⇐" rhs
ebnf_description ⇐ {ebnf_rule}
```

Fonte: Feynman (2012)

Analogamente, precisa-se ter uma ferramenta de metamodelagem inicial que suporte a edição de metamodelos em MOF, mas não escrita em MOF. Essa ferramenta deve implementar os blocos de construção fundamentais do MOF em outra linguagem, respeitando sua semântica exata. A partir daí, seria possível criar o modelo do MOF usando a própria ferramenta e na própria notação MOF. Se a ferramenta permitir a construção de modelos sobre os metamodelos criados, podem-se construir tudo sobre a “pedra fundamental” do modelo MOF, obtida através de *bootstrapping* a partir de uma implementação não criada em MOF.

O EMOF tem o objetivo explícito de reduzir as barreiras de entrada, em relação ao CMOF, para iniciar esse *bootstrapping* (OMG, 2015a).

3.2.3 **Dificuldades na adoção do MOF**

As especificações do MOF 1.1 e da UML 1.1 foram disponibilizadas pelo OMG em 1997 (OMG, 2015a). Desde então, até a década de 2010, a UML vem sendo considerada um padrão de fato para modelagem de software (SHAH; IBRAHIM, 2014), sendo amplamente utilizada (GESSENHARTER; RAUSCHER, 2011).

Embora existam estudos empíricos sobre a utilização da MDE na indústria (MOHAGHEGHI et al., 2013), ao proceder revisão bibliográfica no *Google Scholar* no período dos últimos dez anos, de 2006 a 2016, não foram encontrados trabalhos que abordassem especificamente a adoção do MOF no mercado como tecnologia de apoio para aplicação da MDE. Entretanto, encontra-se na literatura apontamentos sobre dificuldades e problemas relacionados ao MOF que podem ter sido limitadores de sua adoção.

Pastor e Molina (2010), por exemplo, definem diversos metamodelos para criação de modelos de software distintos: interfaces, funcionalidades, estrutura, etc. Funções de mapeamento que combinam modelos para geração de aplicativos são descritas. A UML é aproveitada em grande escala junto com linguagem natural para explicar esses metamodelos, mas o MOF nem mesmo é citado.

Jouault e Bézivin (2006) argumentam que a falta de um ambiente prático de suporte para o MOF levou à solução de utilizar ferramentas CASE baseadas em UML para esse propósito. O “preço a pagar” foi que isso reforçou o alinhamento do MOF como

um subconjunto da UML enxergado no contexto da camada M3, principalmente diagramas de classes, e desde então esse alinhamento foi mantido através das versões seguintes da UML e MOF. Em outras palavras, a UML pode ser considerada como uma linguagem de propósito geral que permite não só modelar software orientado a objetos, mas também definir metamodelos MOF. A construção de um metamodelo MOF consiste então na construção de um diagrama de classes UML respeitando certas propriedades e restrições. A conversão do modelo UML em metamodelo MOF é denominada promoção e é implementada por ferramentas de transformação tais como UML2MOF disponível no Netbeans MDR. De acordo com Jouault e Bézivin (2006), quando o número de metamodelos envolvido é limitado, não há grandes problemas. Entretanto, quando múltiplos metamodelos que evoluem com o tempo são necessários, essa abordagem passou a ser difícil de gerenciar e inconveniente. Isso motivou os autores a elaborar uma linguagem de criação de metamodelos denominada KM3.

Alanen e Porres (2008) reforçam a observação de que o MOF não é especificado de maneira formal o suficiente, o que possivelmente explica a dificuldade na construção de ferramentas para editar, ler e transformar modelos. Especificamente, os autores observam que as versões 2.0 da UML e do MOF trazem novos conceitos, tais como propriedades de subconjuntos e derivadas de união. Alguns desses conceitos são amplamente utilizados nessas especificações, mas carecem de definições precisas, o que é considerado uma omissão crítica. Com o objetivo de prover meios para definições precisas de metamodelos, e consequentemente viabilizar a construção de ferramentas, os autores criaram um formalismo não-metacircular para a especificação de metamodelos que consideram as principais características estruturais do MOF 2.0 e UML 2.0.

Favre (2009) também chama atenção para a falta de formalismo na definição do MOF, e considera que uma especificação formal permitiria produzir especificação de software precisa e analisável, clarificando o significado pretendido dos metamodelos e auxiliando na validação de transformações, além de servir como referência para implementações de ferramentas. É apontado que uma técnica de especificação formal precisa pelo menos prover sintaxe, alguma semântica e um sistema de inferência. A semântica descreve os modelos associados a uma dada especificação. O sistema de inferência permite realizar deduções a partir da especificação formal

original, as quais abrem caminho para novas fórmulas que podem ser derivadas e checadas. Assim, o sistema de inferência pode ajudar a automatizar testes, prototipação ou verificação. Com essa base, a autora apresenta uma linguagem chamada NEREUS, com o objetivo de prover esse suporte para metamodelagem.

Jackson, Levendovszky e Balasubramanian (2013) afirmam que o estado da arte em metamodelagem ainda está abaixo do ideal. Por um lado, consideram que há um consenso geral de que o MOF está subformalizado e merece atenção especial. Por outro lado, tentativas de formalização completa do MOF ou de alternativas ao MOF também ainda não habilitaram extensivamente raciocínio lógico automatizado sobre arcabouços de metamodelagem. Esses autores propõem uma abordagem lógica para formalizar e realizar inferências sobre arcabouços de metamodelagem.

Nas próximas seções, serão apresentados detalhes sobre cada um desses formalismos citados, bem como um quadro comparativo entre eles.

3.3 KM3

Kernel MetaMetaModel (KM3) consiste em uma linguagem para descrever metamodelos apresentada por Jouault e Bézivin (2006). De acordo com suas definições, um modelo é uma designação geral para três tipos específicos de modelo: meta-metamodelo, metamodelo e modelo terminal. As definições para modelo se aplicam a todos os seus tipos específicos. Antes de definir um modelo, define-se um multigrafo direcionado. Um multigrafo direcionado G é uma tripla:

$$G = (N_G, E_G, \Gamma_G)$$

Em que N_G é um conjunto finito de nós, E_G é um conjunto finito de arestas e Γ_G: $E_G \rightarrow N_G \times N_G$ é uma função que mapeia arestas em seus pares de nós origem e destino.

Um modelo M é uma tripla:

$$M = (G, \omega, \mu)$$

Sendo G um multigrafo direcionado, ω um outro modelo, chamado de modelo de referência de M, e μ é uma função que associa os elementos de G aos nós de G_ω, onde $G_\omega = (N_\omega, E_\omega, \Gamma_\omega)$ é o grafo associado ao modelo ω. Ou seja, $\mu: N_G \cup E_G \rightarrow N_\omega$.

A relação entre um modelo e seu modelo de referência é denominada relação de conformidade. Diz-se que M é escrito conforme ω.

Essa definição permite um número arbitrário de camadas de modelagem. Para fins práticos, consideram-se as camadas M3, M2 e M1 da arquitetura das quatro camadas, nas quais se encaixam os três tipos de modelos específicos citados acima: meta-metamodelo, metamodelo e modelo terminal, respectivamente. Por definição, um modelo terminal é aquele cujo modelo de referência é um metamodelo. Um metamodelo, por sua vez, é aquele cujo modelo de referência é um meta-metamodelo. Em concordância com a metacircularidade também existente no MOF, o meta-metamodelo é definido como aquele cujo modelo de referência é ele mesmo.

De acordo com as definições do KM3, para um modelo M são escritas sentenças em lógica de primeira ordem baseadas em dois predicados principais, a saber:

- **Node(x,y):** estabelece que um nó $x \in N_G$ é associado a um nó $y \in N_\omega$ pela função μ.
- **Edge(x,y,z):** estabelece que uma aresta entre o nó $x \in N_G$ e o nó $y \in N_G$ está associada ao nó $z \in N_\omega$ pela função μ. No KM3, múltiplas arestas entre dois nós só podem existir se eles estão associados a metaelementos distintos. Desse modo, a tripla (x,y,z) identifica unicamente uma aresta.

Ou seja, um conjunto de sentenças com esses predicados definem a função μ da tripla do modelo.

No KM3, restrições em modelos são expressas através de fórmulas em lógica de primeira ordem. O meta-metamodelo do KM3 é definido a partir de uma versão simplificada do mesmo, denominada SimpleKM3, apenas com os conceitos de

classes e referências. Em seguida, conceitos adicionais são introduzidos. O SimpleKM3 é definido pelas seguintes sentenças:

- Node(class, class);
- Node(reference, class);
- Node(features, reference);
- Node(type, reference);
- Edge(class, features, features);
- Edge(features, reference, type);
- Edge(reference, type, features);
- Edge(type, class, type).

Em que *class*, *reference*, *features*, *type* são pertencentes à N_G e, portanto, nós do grafo G. A metacircularidade fica evidente já na primeira sentença, na qual *class* é associado ao tipo *class*. Em outras palavras, o mesmo elemento possui interpretação em duas camadas distintas.

Um exemplo de restrição descrita em lógica de primeira ordem é a unicidade do metaelemento, definida pela seguinte sentença:

$$\forall x, y, z\ Node(x,y) \wedge Node(x,z) \Rightarrow y = z$$

Ou seja, μ como função pode associar apenas um único metaelemento a um dado nó do modelo. Outro exemplo de restrição é que todo nó utilizado como metaelemento de outro deve possuir o nó *class* como seu metaelemento:

$$\forall x, y\ Node(x,y) \Rightarrow Node(y,class)$$

Além de outras restrições, o SimpleKM3 é estendido por meio de novos nós e arestas em seu grafo G, juntamente com sentenças em lógica de primeira ordem

adicionais, até formar o meta-metamodelo KM3 completo. O conceito de herança da orientação a objetos é introduzido pela seguinte sentença:

$$\forall x, y\ IsKindOf(x,y) \Leftrightarrow Node(x,y) \vee (\exists z\ Node(x,z) \wedge ConformsTo(z,y))$$

Em que:

$$\forall x, y\ ConformsTo(x,y) \Leftrightarrow (x = y) \vee (\exists z\ Edge(x,z,supertypes) \wedge ConformsTo(z,y))$$

Sendo que *supertypes* denota um tipo de referência, com o significado de que x é submetaclasse de z quando Edge(x, z, supertypes) é verdadeiro, e é introduzido por meio de três sentenças:

- Node(supertypes, reference);
- Edge(class, supertypes, features);
- Edge(supertypes, class, type).

Herança circular é proibida, caso contrário o predicado ConformsTo(x,y) não poderia ser definido. Os autores fornecem um link com uma especificação completa do KM3 em Prolog, podendo ser consultado em Jouault e Bézivin (2006).

A especificação do KM3 não provê uma linguagem ou mecanismo nativo para definir transformações, mas os autores sugerem a utilização da linguagem ATL para este fim. Modelos de marcação não são previstos. Operações de mesclagem (*merge*) e importação (*import*) entre metamodelos também não são mencionadas ou definidas.

A especificação do KM3 não define nada sobre tipos primitivos de dados (p. ex., inteiro, string e booleano) que possam ser usados nas definições dos modelos. Eles são assumidos como existentes, porém sem maiores explicações ou inclusão explícita de enumerações nas definições.

3.4 O FORMALISMO DE ALANEN E PORRES

O trabalho de Alanen e Porres (2008) identifica que as construções encontradas frequentemente em linguagens de metamodelagem, dentre elas o MOF, são muito similares uma vez que são todas baseadas, de algum modo, no paradigma da orientação a objetos. Nessa abordagem, o metamodelo é definido como uma coleção de metaclasses e propriedades, enquanto o modelo consiste em uma instanciação desses conceitos.

Partindo desse pressuposto, o trabalho apresenta uma formalização não metacircular de uma linguagem de metamodelagem que suporta os principais conceitos encontrados no MOF 2.0. Um metamodelo MM é dado por uma quíntupla:

$$MM = (C, P, generalizations, properties, characteristics)$$

Em que C é o conjunto de metaclasses, P é um conjunto de propriedades tais que $C \cap P = \emptyset$. As generalizações são definidas como uma função generalizations: $C \rightarrow 2^C$, em que 2^C é o conjunto potência de C. A relação de generalizações g é definida como:

$$g = \{ (c_1,c_2) \mid c_2 \in generalizations(c_1) \}$$

Ou seja, se $(a,b) \in g$, então a é submetaclasse de b. É requerido que o grafo direcionado que representa a relação de generalização, composto por nós de C e arestas de g, seja acíclico.

Cada propriedade possui uma metaclasse como sua proprietária (*owner*), e este fato é representado pela função owner: $P \rightarrow C$. As propriedades pertencentes a cada classe são dadas pela função properties: $C \rightarrow 2^P$, tal que:

$$\forall c \in C \ (properties(c) = \{ p \mid c = owner(p) \})$$

As propriedades efetivas de uma metaclasse são aquelas definidas pela própria metaclasse mais aquelas definidas transitivamente por meio de suas generalizações.

As características de uma propriedade representam restrições em elementos que podem ser inseridos nos *slots* correspondentes à propriedade. Alanen e Porres (2008) definem as características como uma tupla de funções detalhando essas propriedades:

characteristics = (lower, upper, opposite, ordered, composite, derived, supersets)

Em que:

- **lower:** $P \rightarrow \mathbb{Z}^{0+}$ representa o limite inferior da multiplicidade da propriedade;
- **upper:** $P \rightarrow \mathbb{Z}^{+} \cup \{\infty\}$ representa o limite superior da multiplicidade, incluindo o símbolo de infinito;
- **opposite:** $P \rightarrow P$ é função bijetora que relaciona uma propriedade à sua propriedade oposta. A oposta de uma propriedade não pode ser ela mesma, e a oposta da oposta é ela mesma;
- **ordered:** $P \rightarrow \mathcal{B}$ é verdadeiro se a propriedade é ordenada, isto é, se existe relação de ordem entre os múltiplos valores que o *slot* correspondente pode conter. $\mathcal{B}$ é o conjunto dos valores lógicos booleanos, ou seja, $\mathcal{B}$ = {true, false};
- **composite:** $P \rightarrow \mathcal{B}$ é verdadeiro se a propriedade é composta;
- **derived:** $P \rightarrow \mathcal{B}$ é verdadeiro se a propriedade é derivada;
- **supersets:** $P \rightarrow 2^{P}$ representa o conjunto de propriedades da qual a propriedade é um subconjunto. O grafo representando a relação de *superset* é formado com os nós do conjunto P e as arestas (p,q) tais que $q \in$ supersets(p), devendo ser livre de ciclos.

Um modelo M é então definido como uma sêxtupla:

$$M = (E, type, slots, S, property, elements)$$

Em que E é um conjunto finito de elementos e S é um conjunto finito de *slots*. Cada elemento em E tem um tipo definido por uma metaclasse no metamodelo, dado pela função type: $E \rightarrow C$. Cada *slot* tem um elemento como seu proprietário (*owner*) e esse fato é representado pela função slotowner: $S \rightarrow E$. Por conveniência, também é definida a função *slots* que fornece os *slots* de determinado elemento, tal que slots: $E \rightarrow 2^S$. Cada *slot* corresponde a uma propriedade, estabelecida pela função property: $S \rightarrow P$.

A função elements: $S \rightarrow (2^E, <)$ retorna um conjunto de elementos com relação de ordem total se a propriedade do argumento s é ordenada, ou seja, se ordered(property(s)) = true. Caso contrário, elements: $S \rightarrow 2^E$ retorna um conjunto de elementos sem relação de ordem. O *slot* descreve então a conexão entre o elemento proprietário e os elementos dentro de si.

Existe um *slot* para cada propriedade efetiva no tipo de um elemento. O tamanho de um *slot* s é definido como a quantidade de elementos nesse *slot*, e é denotado por #s, ou seja #s = #elements(s).

O trabalho define dez restrições para os modelos através de sentenças em lógica de primeira ordem. Um exemplo de restrição refere-se à multiplicidade, que estabelece que o tamanho de um *slot* é restringido pelos limites inferior e superior de sua propriedade:

$$\forall s \in S \; lower(property(s)) \leq \#s \leq upper(property(s))$$

Essas dez restrições podem ser usadas para determinar quando um modelo é válido no contexto do arcabouço de modelagem proposto, independentemente de qual metamodelo o modelo deva obedecer. Ou seja, são restrições invariantes. Entretanto, é necessário estabelecer quando um modelo M é uma instância válida de um metamodelo MM. Isso ocorre quando M é um modelo válido e o tipo de cada elemento de M está em MM. Seja C o conjunto de metaclasses de MM, tem-se:

$$\forall e \in E \ type(e) \in C$$

Por outro lado, também se pode permitir M como uma instância de uma extensão de MM. Diz-se que M está em conformidade com MM se M é um modelo válido e o tipo de cada elemento de M é uma submetaclasse de uma metaclasse em MM. Assim se define a relação de conformidade nesse formalismo, ideia similar ao predicado *ConformsTo* do KM3. Dessa forma, supondo que M seja válido, M está em conformidade com MM se:

$$\forall e \in E \ (\exists c \in C \ (type(e) \subseteq_c c))$$

Em que $\subseteq_c$ é definido como g*, ou seja, o fecho transitivo e reflexivo da relação de generalizações g. Isso significa que o tipo de e pode ser uma submetaclasse de alguma metaclasse de MM.

Não é definido explicitamente como um metamodelo pode utilizar metaclasses de outro metamodelo (*import*) ou como dois metamodelos podem ser mesclados (*merge*), mas a explicação da relação de conformidade indica que as submetaclasses podem estar em outro metamodelo. Com isso, implicitamente tem-se de algum modo a possibilidade de criar modelos com tipos provenientes de mais de um metamodelo.

Por outro lado, as referências às metaclasses que aparecem no modelo são referências diretas a esses objetos (*hard references*). Isso implica que o modelo não pode existir sem um metamodelo, e seus elementos ficam diretamente vinculados aos elementos de metaclasse do metamodelo pela função type. A rigor, em uma troca de versão do metamodelo, por exemplo, o modelo precisaria ser reescrito para passar a se referir aos objetos do metamodelo mais recente. Se as referências fossem fracas, pelo identificador da metaclasse (*soft references*), isso não seria necessário desde que o novo metamodelo mantivesse os identificadores para as metaclasses atualizadas.

Embora não tenham sido apresentados nesse resumo, o formalismo também define três conceitos que surgiram no MOF 2.0, a saber: subconjuntos, uniões derivadas e redefinições de propriedades. A motivação é que as especificações do OMG não

descrevem esses conceitos em detalhe, nem mesmo informalmente, e, portanto, não poderiam ser aplicados na prática segundo os próprios autores.

O trabalho não trata de transformações de modelos, e também não fala sobre como definir enumerações. Entretanto, dá a entender que uma enumeração pode ser definida como algum tipo específico de metaclasse, pois considera que o contra-domínio da função type é C. Como são mencionados tipos primitivos para propriedades tais como *string* e inteiro, então assume-se que eles devem pertencer a C.

3.5 NEREUS

NEREUS é uma linguagem específica de domínio para definir metamodelos formais, servindo também como alternativa ao MOF. Provê classes, associações e pacotes como conceitos de modelagem. Sua especificação pode ser encontrada em Favre (2009).

Uma classe pode declarar tipos, atributos, operações e axiomas, que são fórmulas em lógica de primeira ordem. As classes estão estruturadas em diferentes tipos de relações: importação, herança, subtipagem e associações.

A Figura 9 apresenta a estrutura da sintaxe da declaração de uma classe em NEREUS.

A cláusula *IMPORTS* expressa relação de utilização. A especificação da nova classe é baseada nas especificações importadas declaradas em *<importList>* e suas operações públicas podem ser usadas na nova especificação.

A cláusula *ATTRIBUTES* introduz, como no MOF, um atributo com as seguintes propriedades: nome, tipo, multiplicidade e um marcador denominado *isDerived*. Por sua vez, *OPERATIONS* introduz as assinaturas das operações, suas listas de argumentos e tipos de retorno.

Operações podem ser declaradas como totais ou parciais. Funções parciais devem especificar seu domínio por meio de cláusulas *PRE* que indicam que condições os argumentos precisam satisfazer para pertencer ao domínio da função.

Figura 9 – Sintaxe de uma classe em NEREUS

CLASS className [<parameterList>]
IMPORTS <importsList>
INHERITS <inheritsList>
IS-SUBTYPE-OF <is-subtype-of-List>
GENERATED-BY <basicConstructor>
ASSOCIATES <associatesList>
DEFERRED
TYPES <typeList>
ATTRIBUTES <attributeList>
OPERATIONS <operationList>
EFFECTIVE
TYPES <typeList>
OPERATIONS <operationList>
AXIOMS <varList>
<axiomList>
END-CLASS

Fonte: Favre (2009)

Em NEREUS é possível especificar os três níveis usuais de visibilidade para operações encontrados nas linguagens orientadas a objeto: público (*public*), protegido (*protected*) e privado (*private*). Os significados de outras cláusulas além das aqui apresentadas podem ser encontrados em Favre (2009).

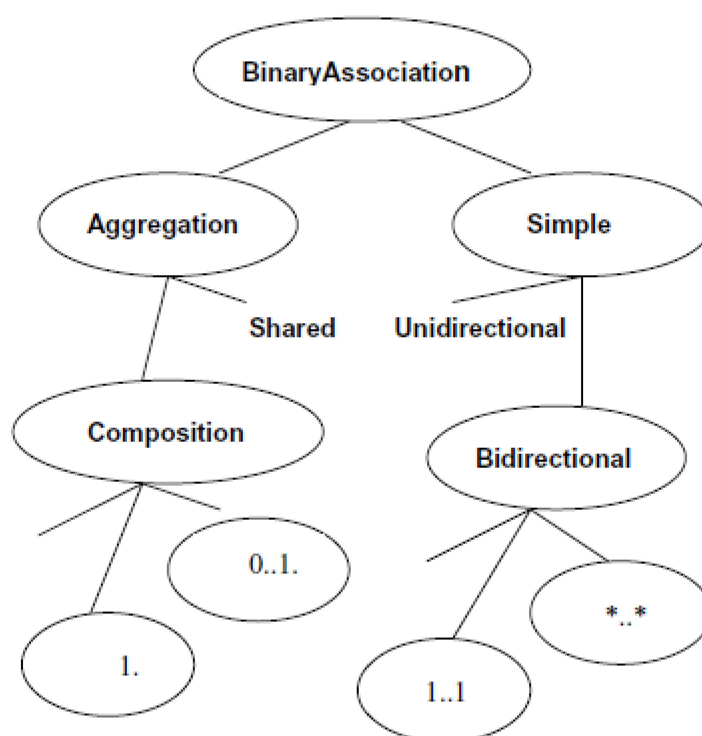
NEREUS provê um componente de associação e uma taxonomia de tipos de construtores, que classificam associações binárias de acordo com seu tipo (agregação, composição e associação normal), grau (unária, binária), navegabilidade (unidirecional, bidirecional) e conectividade (um-para-um, um-para-muitos, muitos-para-muitos). A Figura 10 apresenta um diagrama com as classificações possíveis para associações, enquanto a Figura 11 apresenta a estrutura da sintaxe da declaração de uma associação.

A cláusula *CONSTRAINED-BY* permite a especificação de restrições estáticas em lógica de primeira ordem. Relações são definidas nas classes por meio da cláusula *ASSOCIATES*.

Por fim, pacotes são o mecanismo oferecido por NEREUS para agrupar elementos. A Figura 12 apresenta a estrutura da sintaxe da declaração de um pacote. Como no MOF, NEREUS provê mecanismos para composição de metamodelos e reuso. A

cláusula IMPORTING lista os pacotes importados. São suportados generalização e aninhamento de pacotes, mas nada é comentado sobre mesclagem (*merge*) de metamodelos ou de pacotes.

Figura 10 – Associações em NEREUS



Fonte: Favre (2009)

Figura 11 – Sintaxe de uma associação em NEREUS

```
ASSOCIATION <relationName>
IS <typeConstructorName> […:class1;…:class2;…:role1; …:role2;
…:mult1;…:mult2; …:visibility1;…:visibility2]
CONSTRAINED-BY <constraintList>
END-ASSOCIATION
```

Fonte: Favre (2009)

Figura 12 – Sintaxe de um pacote em NEREUS

```
PACKAGE packageName
IMPORTING <importsList> GENERALIZATION <inheritsList>
NESTING <nestingList>  CLUSTERING <clusteringList>
<elements>
END-PACKAGE
```

Fonte: Favre (2009)

O trabalho de Favre (2009) também apresenta indicações sobre como transformar um metamodelo MOF, incluindo suas restrições em OCL, em um metamodelo denotado em NEREUS. Essa transformação é descrita em linguagem natural. É suportada a EssentialOCL, que consiste em um subconjunto da OCL completa. São definidas, também em linguagem natural, enumerações básicas que representam tipos primitivos tais como *Integer*, *Real*, *Boolean* e *String*. Não é explicado como definir outros tipos enumerados.

Common Algebraic Specification Language (CASL) é uma linguagem expressiva baseada em algumas construções tais como funções, lógica de primeira ordem e especificações arquiteturais estruturadas (FAVRE, 2009). Especificações são estruturadas por meio de operadores, extensões e combinações. Especificações arquiteturais impõem estruturas em implementações. A semântica do NEREUS é fornecida através de uma tradução em CASL, escolhida como base com semântica pré-conhecida. Um exemplo de mapeamento de associações em NEREUS para CASL é apresentado em Favre (2009).

A especificação de NEREUS tem como característica negativa ser definida em grande parte em linguagem natural, não deixando plenamente claro ou preciso o significado de todos os seus conceitos. Um exemplo para isso é a diferença entre subtipagem e herança, que não fica bem explicada. Por isso, NEREUS parece não se diferenciar muito em relação ao que o MOF já oferece, sendo basicamente uma linguagem textual específica de domínio que, em tese, facilita a expressão de metamodelos MOF, os quais são denotados graficamente em linguagem similar ao diagrama de classes UML com o auxílio de OCL para estabelecer restrições.

Embora o trabalho mencione um possível mecanismo de inferência para efetuar raciocínio lógico sobre metamodelos, nenhum é apresentado. Também não foram encontrados até o momento em outros trabalhos da mesma autora.

3.6 A ABORDAGEM DE JACKSON, LEVENDOVSZKY E BALASUBRAMANIAN

O trabalho de Jackson, Levendovszky e Balasubramanian (2013) concentra seus esforços em uma abordagem para o problema da especificação formal de arcabouços de metamodelagem de forma geral, buscando maneiras para a realização de raciocínio lógico automatizado (*reasoning*) sobre os mesmos. O ponto central da abordagem utiliza tipos de dados algébricos, ou *Algebraic Data Types* (ADTs) e programação lógica por restrções, ou C*onstraint Logic Programming* (CLP). Os autores constróem uma especificação de um arcabouço de metamodelagem baseado em grafos tipados, que pode servir como ponto de partida para arcabouços mais complexos.

As especificações são escritas em linguagem FORMULA, que é uma linguagem de especificação lógica de alto nível que implementa a hipótese do mundo aberto, ou O*pen World Assumption* (OWA). A hipótese do mundo aberto considera que o valor verdade de uma proposição é independente do conhecimento do agente, em contraposição à hipótese do mundo fechado, ou C*losed World Assumption* (CWA), que supõe que uma proposição não conhecida é falsa.

Sob a OWA, a falha em derivar um fato não implica o oposto. Por exemplo, assumindo que se sabe apenas que Maria é cidadã da França pela sentença lógica Cidadão(Maria,França), ao perguntar para a máquina de inferência se Paulo é cidadão da França através da questão Cidadão(Paulo,França), sob a CWA a resposta é não, enquanto sob OWA a resposta é desconhecida.

Os conceitos principais que são propostos para a construção de um arcabouço de metamodelagem são: repositório de modelos, operações de edição e metainterpretador.

O repositório de modelos serve para codificar o conjunto de todos os pares metamodelo/instância em conformidade. Instâncias peculiares desses pares podem ser construídas através de questões OWA pela máquina de inferência da linguagem

FORMULA. O repositório de modelos é especificado em dois níveis descritivos, sendo o par de camadas M2/M1 (metamodelo/modelo) um caso particular. É permitido, portanto, um número arbitrário de camadas na definição.

Basicamente, um grafo direcionado G é definido como uma quádrupla:

$$G = (V, E, src, dst)$$

Em que V e E são os conjuntos de nós e arestas, respectivamente, e src e dst são funções tais que src: $E \rightarrow V$ e dst: $E \rightarrow V$. Essas funções associam cada aresta aos seus nós origem e destino. Percebe-se aqui o mesmo conceito da definição de multigrafo direcionado do KM3, mas com uma diferença na formalização: os nós origem e destino são dados por duas funções, e não uma função que resulta em um par ordenado.

Um grafo tipado T é definido então como outra quádrupla:

$$T = (G, H, vt, et)$$

Sendo G e H grafos direcionados, vt e et funções tais que vt: $V_H \rightarrow V_G$ e et: $E_H \rightarrow E_G$, em que V_H é o conjunto de nós de H, V_G o conjunto de nós de G, E_H o conjunto de arestas de H e E_G o conjunto de arestas de G.

A interpretação de T como um par metamodelo/instância é que o grafo G age como o metamodelo provendo o conjunto de tipos. O grafo H representa uma instância desse metamodelo, cujos elementos são associados aos tipos de G por vt e et. O conceito é muito similar ao KM3, com a diferença de que aqui é permitido que as arestas de G também atuem como tipos, enquanto no KM3 apenas os nós do grafo G seriam permitidos como tipos. O KM3 é, inclusive, citado no trabalho de Jackson, Levendovszky e Balasubramanian (2013).

É dito que um modelo H está em conformidade com o metamodelo G se as arestas e vértices de H estão conectadas de acordo com seus tipos, de acordo com a equação abaixo:

$$conforms(T) = \forall e \in E_H,\ src_G(et(e)) = vt(src_H(e)) \wedge dst_G(et(e)) = vt(dst_H(e))$$

O repositório de modelos Store(U) é então definido como o conjunto de todos os possíveis pares metamodelo/instância em conformidade construídos sobre um universo U de nós e arestas.

$$Store(U) = \{\ T \mid conforms(T) \wedge V_G, V_H, E_G, E_H \subseteq U\ \}$$

As operações de edição consistem em transformações para editar elementos no nível da instância do par considerado. Essas transformações são definidas sobre o repositório de modelos. A formalização de editores, ou operações de edição, permite gerar casos de teste nos quais a edição destrói a conformidade. Para o arcabouço definido no trabalho, são especificadas apenas operações de edição que sempre mantêm a conformidade. Exemplos de operações de edição são a inserção ou exclusão de nós ou arestas no repositório de modelos.

O metainterpretador é uma transformação que promove elementos do nível de instância ao seu metanível. Diz-se que um arcabouço é metacircular se existe um modelo de entrada para o metainterpretador que, quando promovido ao seu metanível, resulta em um modelo igual. O trabalho aproveita esta propriedade para formular uma questão OWA que deduz automaticamente um meta-metamodelo descrito nele mesmo, como o caso do MOF.

O trabalho de Jackson, Levendovszky e Balasubramanian (2013) é bastante rico com relação a raciocínio e inferência lógica em metamodelagem, introduzindo muitos conceitos precisos. Por outro lado, baseia-se muito na linguagem FORMULA, deixando a operação de certa forma presa a esse contexto. Também pouco é considerado sobre o tema da MDE em geral, principalmente com relação a transformações entre modelos, transformações de modelos para código, como descrever essas transformações, e operações entre metamodelos. Também não se aborda como definir enumerações ou operações de importação (*import*) e mesclagem (*merge*) entre metamodelos dentro do contexto do formalismo.

3.7 QUADROS COMPARATIVOS

O Quadro 3 apresenta um comparativo entre os cinco formalismos apresentados com base na metacircularidade e natureza predominante de sua especificação. O Quadro 4, por sua vez, faz a comparação com respeito a recursos de metamodelagem, modelagem e transformação de modelos, que são os aspectos centrais dos processos MDE. Por fim, o Quadro 5 compara com referência às sintaxes concretas (linguagens gráficas ou textuais) de utilização prática dos formalismos ou linguagens.

Quadro 3 – Comparativo de formalismos e linguagens segundo especificação

Característica	MOF	KM3	Alanen	NEREUS	Jackson
Metacircular	Sim	Sim	Não	Não	Sim
Natureza predominante da especificação	Linguagem natural	Formal	Formal	Linguagem natural	Formal
Principais conceitos primitivos subjacentes usados na especificação	Metaclasses	Multigrafos direcionados e funções	Conjuntos, relações e funções	Metaclasses	Grafos direcionados e funções

Fonte: autor

Quadro 4 – Comparativo de formalismos e linguagens sobre recursos

Característica	MOF	KM3	Alanen	NEREUS	Jackson
Suporte a operações nas metaclasses	Sim	Não	Não	Sim	Não
Suporte a definição de enumerações e tipos primitivos	Sim	Não*	Não**	Não*	Sim
Suporte à capacidade de reflexão do MOF	Sim	Não	Não	Não	Não
Suporte à capacidade de identificação do MOF	Sim	Não	Não	Sim	Não
Suporte à capacidade de extensão do MOF ou modelos de marcação	Sim	Não	Não	Não	Não
Suporte à mesclagem e importação de metamodelos	Sim	Não	Não***	Somente importação	Não
Suporte à transformação de modelos	Sim	Sim	Não	Sim****	Não*****
Descreve modelos executáveis	Não	Não	Não	Não	Não
Descreve sincronização de modelos	Não	Não	Não	Não	Não

* Utiliza como recurso implícito mas não especifica definições.
** Não, mas sugere que é possível.
*** Apenas assume implicitamente esta possibilidade.
**** Menciona QVT como uma possibilidade para transformar modelos mas não especifica sua interação com a linguagem apresentada.
***** Refere-se à transformação como operações no contexto de um mesmo modelo para editá-lo, mas não no contexto da MDE.

Fonte: autor

Quadro 5 – Comparativo de formalismos e linguagens sobre sintaxe concreta e utilização

Característica	MOF	KM3	Alanen	NEREUS	Jackson
Linguagem ou notação de referência para metamodelagem	Notação gráfica própria similar a classes UML	Conjuntos, relações, sentenças em LPO e linguagem própria	Conjuntos, relações e sentenças em LPO	Própria	FORMULA
Linguagem ou notação de referência para modelagem	UML	Conjuntos, relações, sentenças em LPO e linguagem própria	Conjuntos, relações e sentenças em LPO	Não especifica	FORMULA
Linguagem ou notação de referência para restrições sobre modelos	OCL	Sentenças em LPO e linguagem própria	LPO*	Própria	FORMULA
Linguagem ou notação de referência para transformações	QVT e MOFM2T	ATL	Não especifica	QVT	Não especifica
Mecanismo de referência para inferência e raciocínio (*reasoning*)	Não especifica	LPO	LPO	Não especifica	FORMULA
Ferramentas de referência	Diversas. Ex: Papyrus e Acceleo	Compilador Prolog, Eclipse	Não especifica	Não especifica	FORMULA

* Não especifica mecanismo explícito de inserir restrições específicas sobre modelos nos metamodelos, mas como usa LPO para definir restrições comuns e invariantes para todos os modelos, então subentende-se que pode ser usada LPO para isso.

Fonte: autor

3.8 OUTROS FORMALISMOS E LINGUAGENS

Metamodelagem tem sido um tópico extensivamente pesquisado. Diversas outras abordagens de formalização, além das apresentadas, podem ser encontradas na literatura. Alguns outros representantes são: *Metamodeling Language Calculus* (MML), baseada em cálculo-ζ (CLARK; EVANS; KENT, 2001); VPM, baseada em grafos como o KM3 (VARRÓ; PATARICZA, 2003); MOMENT2, que tem como base formal a *membership equational logic* (BORONAT; MESEGUER, 2008) e CINV, uma das primeiras tentativas de formalizar metamodelagem sem metacircularidade, baseada em conjuntos e relações mas sem suporte a generalizações (BAAR, 2003).

Favre (2004)[3] afirma que, uma vez que a MDE propõe modelagem precisa para se obter automaticamente código-fonte ou artefatos executáveis, seus conceitos fundamentais também precisam ser definidos de forma precisa. O autor propõe a utilização das teorias de conjuntos e linguagens, consideradas “raízes” da Ciência da Computação, para atingir este fim.

3.9 MOTIVAÇÕES PARA O SBMM

O estudo do MOF e de outras abordagens de formalização para metamodelagem permitiu identificar vantagens e lacunas de cada uma. Por exemplo, KM3 e a abordagem de Jackson, Levendovszky e Balasubramanian (2013) possuem alto grau de formalismo, mas não mencionam extensão de metamodelos ou modelos de marcação. Alanen e Porres (2008) usam referências diretas aos objetos do metamodelo no modelo, o que dificulta sua aplicação tal como é na construção de ferramentas, pois as estruturas precisam ser modificadas em relação às originais para que um modelo não tenha que carregar seu metamodelo. Nenhum desses formalismos também aborda precisamente definições de enumerações como parte do metamodelo. Apenas supõem que enumerações básicas existem e podem ser usadas. Isso deixa uma lacuna para operações de mesclagem de metamodelos, por exemplo, que precisariam operar também sobre as enumerações presentes nos mesmos.

Com relação à transformação de modelos, os formalismos estudados fazem referência a linguagens ou mecanismos já conhecidos, mas não apresentam uma conceituação e definições formais sobre transformações, explicando-as em linguagem natural. Para tratar esse aspecto, utilizou-se ideias de trabalhos mais específicos sobre sincronização e transformação de modelos (XIONG et al., 2007) (RUTLE et al., 2012).

Um outro tópico mais recente da MDE, a execução direta de modelos sem passar por transformações, não é abordado por nenhum dos trabalhos citados, sendo outra lacuna a ser explorada.

---

[3] É importante observar que autor do trabalho referenciado por Favre (2004) não é a mesma autora do trabalho referenciado por Favre (2009), apesar do mesmo sobrenome. Favre (2004) advoga por uma formalidade muito maior e a utilização de conceitos mais primitivos que os utilizados em Favre (2009) para fundamentar a MDE.

A própria existência de múltiplas tentativas para formalização de conceitos da MDE indica que não houve um consenso até então. Se, por um lado, propor mais uma possibilidade pode tornar esse cenário ainda mais diverso, por outro lado a agregação de ideias de várias fontes em um único formalismo e sua utilização como arcabouço subjacente comum para aplicações teóricas e práticas pode contribuir com a consolidação e entendimento da MDE, seus conceitos e fenômenos.

Visando a contribuir com o campo da formalização de conceitos da MDE, o SBMM não se propõe a ser um formalismo construído do zero, embora suas definições sejam autocontidas e não recorram a referências a outros formalismos, mas sim reunir e aproveitar conceitos e ideias já existentes na literatura e preencher lacunas, servindo como uma proposta alternativa que encontra seu nicho de aplicação teórica e prática. Inicia-se a apresentação do SBMM no Capítulo 4.

4 SET BASED META MODELING (SBMM)

4.1 INTRODUÇÃO

Este capítulo apresenta o formalismo SBMM por completo. Inicia-se com a conceituação de metamodelos, abordando detalhadamente todos os seus componentes. É apresentada sua notação gráfica associada, e em seguida são abordados modelos, restrições sobre modelos, relação de conformidade, modelos de marcação, operação de mesclagem de metamodelos, funções de mapeamento que definem transformações e funções para construção de modelos, tendo esta última especial utilidade como especificação para construção de ferramentas.

Devido ao caráter teórico deste capítulo, muitas definições e afirmações são apresentadas na forma de sentenças em lógica de primeira ordem. Russel e Norvig (2004) apresentam uma gramática que define a sintaxe desse tipo de sentença. Sempre que possível, uma leitura em linguagem natural de cada sentença é apresentada em seguida para facilitar o entendimento.

Para simplificar algumas expressões quantificadas, adota-se a seguinte convenção. Expressões com o quantificador universal $\forall$ do tipo "$\forall x\ (x \in A) \Rightarrow$ *Sentença*", onde x é uma variável, A é um conjunto e *Sentença* é uma sentença interna podem ser expressas como "$\forall x \in A$ (*Sentença*)", sem prejuízo ao significado e sem introduzir ambiguidades. Esse tipo de expressão faz a afirmação *Sentença* para todos os elementos do conjunto A. Analogamente, expressões com o quantificador existencial $\exists$ do tipo "$\exists x\ (x \in A) \wedge$ *Sentença*" podem ser expressas como "$\exists x \in A$ (*Sentença*)". Esse tipo de expressão afirma que existe algum elemento de A para o qual *Sentença* é válida.

4.2 METAMODELOS EM SBMM

Um metamodelo consiste em um nome identificador (cadeia de símbolos) $name_{MM}$, um conjunto C de metaclasses, um conjunto Γ de generalizações, um conjunto E de enumerações, um conjunto R de restrições, uma função $descriptor_c$ para mapear metaclasses em seus descritores e uma função $descriptor_e$ para mapear

enumerações em seus descritores. Ou seja, um metamodelo é uma sétupla MM conforme a eq. (1):

$$MM = (name_{MM}, C, \Gamma, E, R, descriptor_c, descriptor_e) \quad (1)$$

Em que:

- $name_{MM} \in \Sigma^+$, em que Σ é um conjunto finito não-vazio de símbolos pré-definido (fora da definição de MM) para a formação de cadeias que representam nomes. Por exemplo, Σ pode ser o conjunto das letras alfabeto romano com maiúsculas e minúsculas, dígitos de 0 a 9 e símbolos adicionais como *underscore*. O elemento $name_{MM}$ representa o nome do metamodelo, servindo para implementar capacidade equivalente àquela dos identificadores do MOF. O conjunto Σ é suposto ser o mesmo para todos os metamodelos dentro de um contexto;
- $C = \{c_1, c_2, ..., c_{|C|}\}$ é o conjunto finito, eventualmente vazio, de metaclasses. $|C|$ denota a cardinalidade, ou número de elementos, de C;
- $\Gamma \subseteq C \times C$ é uma relação não reflexiva livre de ciclos que mapeia submetaclasses em suas supermetaclasses, representando o conceito de herança da orientação a objetos. Não pode ser reflexiva, pois uma metaclasse não é submetaclasse nem supermetaclasse dela mesma. Por não permitir ciclos e nem reflexão, então Γ é subconjunto próprio de $C \times C$ quando $C \neq \emptyset$. A relação Γ indica as relações diretas de herança, enquanto seu fecho transitivo Γ^+ contém também as relações indiretas. Ou seja, se c_1 é submetaclasse de c_2 e c_2 é submetaclasse de c_3, então c_1 é submetaclasse de c_3, embora não diretamente, de modo que $(c_1, c_3) \in \Gamma^+$;
- $E = \{e_1, e_2, ..., e_{|E|}\}$ é o conjunto finito, eventualmente vazio, de enumerações;
- $R = \{r_1, r_2, ..., r_{|R|}\}$ é o conjunto finito, eventualmente vazio, de restrições sobre modelos de MM (importante frisar que não são restrições sobre o metamodelo MM) tais que $r_1 \wedge r_2 \wedge ... \wedge r_{|R|}$ não consistam em uma falácia. Mais adiante será visto que os elementos de R são sentenças em lógica de primeira ordem;

- descriptor$_c$: $C \rightarrow \Sigma^+ \times 2^{U_{P,MM}}$ é uma função que mapeia cada metaclasse $c_j \in C$ em seu descritor, que é dado por um par ordenado composto por um nome identificador metaclassname$_j \in \Sigma^+$ e um conjunto de propriedades. $U_{P,MM}$, a ser detalhado na eq. (4), denota o conjunto universo de todas as propriedades possíveis no contexto do metamodelo MM. O descritor de metaclasse é detalhado e exemplificado mais adiante;
- descriptor$_e$: $E \rightarrow \Sigma^+ \times 2^{U_V}$ é uma função que mapeia cada enumeração $e_j \in E$ em seu descritor, que é dado por um par ordenado composto por um nome identificador enumname$_j \in \Sigma^+$ e um conjunto de valores possíveis. U_V denota o conjunto universo de todos os valores possíveis. O descritor de enumeração é detalhado e exemplificado mais adiante.

Analisando sob a luz da arquitetura das quatro camadas, as definições e restrições sobre a sétupla apresentada acima formam o meta-metamodelo, correspondente à camada M3. Um metamodelo consiste em uma instância da sétupla, e se encaixa na camada M2. Um modelo é uma instância de um metamodelo, e esses são posicionados no nível M1. Mais adiante, na seção 4.7, é apresentada uma definição de modelo e, na seção 4.9, sua relação de conformidade com seu metamodelo. É no nível dos modelos em que o desenvolvedor de software trabalha.

Metaclasses funcionam como tipos para os elementos de um modelo. Na UML, por exemplo, o conceito de caso de uso consiste em uma metaclasse, enquanto um caso de uso específico é uma instância do mesmo e consiste em um elemento do modelo.

A metaclasse a partir da qual um elemento de modelo foi instanciado é denominada de seu tipo base (*base type*). As supermetaclasses dessa metaclasse, e também ela mesma, são denominadas de tipos do elemento (*type*). Logo, o tipo base é também um tipo do elemento.

Para facilitar o entendimento das expressões, a partir deste ponto convenciona-se que as letras j, k, x e y são usadas como indexadores genéricos, salvo quando expresso algo diferente em algum caso específico, sendo, portanto, números naturais positivos. Seus limitadores máximos dependem do contexto. Por exemplo,

se j estiver indexando uma metaclasse, como em $c_j \in C$, então $j \leq |C|$. Analogamente, se j estiver indexando uma enumeração, então $j \leq |E|$. As letras u e w, ao menos que expresso o contrário, denotam cadeias de Σ^+.

Cada metaclasse $c_j \in C$ é descrita por um nome identificador $metaclassname_j$ e um conjunto de propriedades P_j. Sendo assim, pode-se mapeá-las a pares ordenados por meio da função $descriptor_c$ de acordo com a eq. (2):

$$descriptor_c(c_j) = (metaclassname_j, P_j) \quad (2)$$

Em que:

- $metaclassname_j \in \Sigma^+$ é a cadeia de símbolos que identifica c_j dentro do metamodelo;
- $P_j = \{p_{j1}, ..., p_{j|Pj|}\} \in 2^{U_{P,MM}}$ é o conjunto finito de propriedades da metaclasse c_j, eventualmente vazio.

Os nomes devem ser únicos dentro do metamodelo, tais que se $c_j \neq c_k$ então $metaclassname_j \neq metaclassname_k$.

Do ponto de vista semântico, as propriedades $p_{jx} \in P_j$ definem *slots* de informação para elementos de modelos (instâncias) da metaclasse c_j e de suas submetaclasses. Esses *slots* aparecem na definição de modelo, mais adiante, na seção 4.7.

Cada propriedade p_{jx}, por sua vez, consiste em um nome identificador, um tipo alvo (que pode ser uma metaclasse ou enumeração) e uma multiplicidade que restringe quantos elementos o *slot* de informação representado pode armazenar. Ou seja, propriedades são triplas conforme a eq. (3).

$$p_{jx} = (propertyname_{jx}, propertytype_{jx}, mult_{jx}) \quad (3)$$

Em que:

- $propertyname_{jx} \in \Sigma^+$ é a cadeia de símbolos que identifica p_{jx} dentro de sua metaclasse;
- $propertytype_{jx} \in C \cup E$ é o tipo alvo da propriedade, ou seja, pode ser uma metaclasse ou uma enumeração do metamodelo;
- $mult_{jx} \in \mathbb{N} \times (\mathbb{N}^+ \cup \{*\})$ é a multiplicidade da propriedade, sendo um par ordenado cujo primeiro elemento é um número natural que denota o limite inferior de *slots* que a propriedade é capaz de armazenar. O segundo elemento pode ser um número natural positivo ou um símbolo *, que representa infinito, denotando o limite superior de *slots*. Se $mult_{jx} = (1, 1)$, por exemplo, significa que a propriedade possui um e apenas um *slot* de informação. Se $mult_{jx} = (0, *)$, então a propriedade representa uma lista com um número arbitrário de *slots*. Para tornar a notação mais semelhante àquela da UML e MOF, um par ordenado de multiplicidade (lower, upper) também será denotado por lower..upper. Na UML e no MOF, as multiplicidades não necessariamente estão dentro de uma faixa de valores inicial e final, podendo ser valores arbitrários tais como 1, 3 e 5 (sem incluir 2 e 4). Porém, para nossos propósitos a definição apresentada aqui é suficiente. Essa simplificação também foi adotada no formalismo de Alanen e Porres (2008).

Os nomes das propriedades devem ser únicos dentro da metaclasse, tais que se $p_{jx} \neq p_{jy}$ então $propertyname_{jx} \neq propertyname_{jy}$.

Desse modo, pode-se então verificar que $U_{P,MM}$, o conjunto de propriedades possíveis no contexto do metamodelo MM, é dado pela eq. (4).

$$U_{P,MM} = \Sigma^+ \times (C \cup E) \times (\mathbb{N} \times (\mathbb{N}^+ \cup \{*\})) \quad (4)$$

Ou seja, $U_{P,MM}$ é construído a partir de conjuntos gerais ou já presentes em MM e, portanto, não precisa ser incluído na sétupla que define o metamodelo.

A relação $PEF_{MM} \subseteq C \times (P_1 \cup P_2 \cup ... \cup P_{|C|})$ de um metamodelo MM estabelece as propriedades efetivas de cada metaclasse, que são aquelas definidas por elas

mesmas mais aquelas definidas por suas supermetaclasses, transitivamente, conforme a eq. (5):

$$PEF_{MM} = \left\{ (c_j, p) \mid (c_j \in C) \wedge \left(p \in P_j \cup \bigcup_{(c_j, c_k) \in \Gamma^+} P_k \right) \right\} \quad (5)$$

A eq. (5) inclui na relação os pares ordenados (c_j, p) tais que o primeiro objeto é uma metaclasse e o segundo uma propriedade, de modo a incluir as propriedades pertencentes à própria metaclasse (as do conjunto P_j) e também aquelas pertencentes a qualquer supermetaclasse (as dos conjuntos P_k), transitivamente, quando $(c_j, c_k) \in \Gamma^+$.

Como, pela definição de PEF_{MM}, é possível obtê-la a partir dos conjuntos que compõem a definição de MM, então PEF_{MM} não é uma relação livre a ser definida em MM, mas sim decorrência da sétupla que o define, assim como ocorreu para o conjunto $U_{P,MM}$.

Pode-se utilizar a relação PEF_{MM} para definir a função $PEFC_{MM}(c_j)$, a qual toma uma metaclasse $c_j \in C$ como argumento e a mapeia no conjunto de propriedades $\{pef_{j1}, ..., pef_{jn}\}$ tais que $(c_j, pef_{jx}) \in PEF_{MM}$, em que n, nesse contexto, é o número de propriedades efetivas de c_j. Ou seja, $PEFC_{MM}$ é uma função conforme (6) definida pela expressão da eq. (7).

$$PEFC_{MM}: C \rightarrow 2^{P_1 \cup P_2 \cup \ldots \cup P_n} \quad (6)$$

$$PEFC_{MM}(c_j) = \{pef_{jx} \mid (c_j, pef_{jx}) \in PEF_{MM}\} \quad (7)$$

As enumerações $e_j \in E$ servem para definir tipos de dados básicos cujas instâncias podem assumir um valor de um conjunto finito, definido pela própria enumeração. Cada enumeração é então descrita por um par ordenado mapeado pela função $descriptor_e$ conforme a eq. (8).

$$descriptor_e(e_j) = (enumname_j, L_j) \quad (8)$$

Em que:

- $enumname_j \in \Sigma^+$ é a cadeia de símbolos que identifica e_j dentro do metamodelo;
- L_j é o conjunto finito de valores que as instâncias de e_j podem assumir.

Os nomes devem ser únicos dentro do metamodelo, tais que se $e_j \neq e_k$ então $enumname_j \neq enumname_k$. Também não pode haver enumeração com o mesmo nome de qualquer metaclasse do mesmo metamodelo.

Por exemplo, pode-se definir uma enumeração e_1 de um tipo básico booleano como:

$$descriptor_e(e_1) = (\text{“Boolean”}, \{true, false\}) \tag{9}$$

A eq. (9) define uma enumeração denominada *Boolean* cujas instâncias podem assumir os valores *true* ou *false*.

As cadeias de símbolos que representam nomes identificadores são denotadas entre aspas para evitar ambiguidade com referências a outros objetos do metamodelo ou valores permitidos em enumerações, ou seja, não se deve confundir a propriedade designada por p em uma equação com o eventual nome “p” (cadeia de símbolos) que identifica um elemento do metamodelo.

Estender a definição de enumeração para aceitar conjuntos L_j infinitos enumeráveis é simples, mas não faz sentido na prática pois uma variável de computador é sempre armazenada em um número finito de bits, e, portanto, uma instância de enumeração não pode assumir um dentre infinitos valores. Sendo assim, um tipo básico de número inteiro de 64 bits com sinal pode ser definido pela enumeração e_2 da eq. (10):

$$descriptor_e(e_2) = (\text{ “Integer64”}, \{x \mid (x \in \mathbb{Z}) \wedge (-2^{63} \leq x \leq 2^{63} - 1)\}) \tag{10}$$

Uma enumeração e_3 para o tipo *String* sobre o alfabeto Σ pode ser definida pela eq. (11).

$$descriptor_e(e_3) = (\text{“String”}, \{x \mid (x \in \Sigma^*) \wedge (|x| \leq h) \}) \quad (11)$$

A restrição $|x| \leq h$ para algum limitante superior h garante que a lista seja finita. Na prática, as linguagens de programação permitem variáveis do tipo *String* de tamanhos bastante elevados, de forma que h é muito grande. Normalmente é limitado pela memória disponível ou alguma característica da linguagem de programação utilizada.

Entretanto, como o SBMM é independente de implementação, para facilitar definições, poderão ser apresentados conjuntos infinitos enumeráveis de valores permitidos, tais como em e_4:

$$descriptor_e(e_4) = (\text{“Integer”}, \mathbb{Z})$$

O leitor deve subentender que, ao serem transportados para uma implementação computacional, os eventuais conjuntos infinitos de valores de enumerações devem ser substituídos por equivalentes práticos. Por exemplo, a enumeração e_2 da eq. (10) é uma implementação usual de e_4, já que inteiros de 64 bits resolvem a maioria dos problemas que precisam manipular inteiros em geral.

Ademais, até mesmo conjuntos infinitos não-enumeráveis poderiam ser usados para fins de simplificação das expressões, por exemplo em e_5, desde que o leitor também subentenda que, na prática, uma aproximação finita do conjunto será de fato utilizada:

$$descriptor_e(e_5) = (\text{“Real”}, \mathbb{R})$$

As enumerações não contêm propriedades e nem possuem relação de generalização.

O conjunto U_V é uma abstração para denotar o conjunto de todos os valores possíveis que podem pertencer a conjuntos de valores de enumerações. Trata-se de uma abstração única para todos os metamodelos e, portanto, esse conjunto não se encontra definido dentro da sétupla do metamodelo.

Cada restrição $r_j \in R$ é uma sentença em lógica de primeira ordem. Cada uma representa uma restrição nos modelos conformes ao metamodelo MM, relação esta a ser detalhada na seção 4.6. Uma das condições para que um modelo esteja em conformidade com o metamodelo MM é respeitar todas as restrições $r_j \in R$. Outra forma de interpretar R, do ponto de vista de implementação, é que cada r_j é uma sub-rotina que toma um modelo como argumento e retorna verdadeiro ou falso de acordo com o respeito à restrição correspondente. Se a avaliação falhar para algum r_j, então o modelo não está em relação de conformidade com o metamodelo.

Essa interpretação corresponde ao fato de Rodrigues (2009) observar que linguagens de ação também servem para descrever restrições na UML. Assim, pode-se escolher uma única linguagem de ação para denotar ações e restrições, sem precisar da OCL, que é especializada em restrições.

As funções $descriptor_c$ e $descriptor_e$ são auxiliares formais para mapear os objetos correspondentes a metaclasses e enumerações, respectivamente, em seus descritores. Para fins de simplificação de notação, usar-se-á os operadores $=^c$ e $=^e$ para indicar os descritores de metaclasses e enumerações. Dessa forma, a eq. (2) pode ser denotada de forma simplificada por (12).

$$c_j =^c (metaclassname_j, P_j) \qquad (12)$$

E, também como exemplo, a eq. (9) pode ser denotada de forma simplificada por (13).

$$e_1 =^e (\text{“Boolean”}, \{true, false\}) \qquad (13)$$

Para clarificar o entendimento das definições, convém apresentar um exemplo de um metamodelo aplicável na prática.

4.3 EXEMPLO DE METAMODELO DESCRITO EM SBMM

Considere-se um metamodelo simples que define uma parte do conhecido diagrama de classes UML. Em um diagrama de classes UML, os elementos mais comuns são classes, atributos, operações, associações e generalizações. Em um modelo de um sistema de software específico, esses elementos (camada M1) são instâncias de metaclasses (camada M2). O metamodelo a ser apresentado como exemplo, para fins ilustrativos, não considera todos os aspectos existentes no diagrama de classes da UML, deixando de fora conceitos como agregação, anotações e outros.

Seja então o metamodelo $MM_{CLASSES}$ definido pela eq (14).

$$MM_{CLASSES} = (\text{“ClassMetamodel”}, C_{CLASSES}, \Gamma_{CLASSES}, E_{CLASSES}, R_{CLASSES}, descriptor_{c_CLASSES}, descriptor_{e_CLASSES}) \quad (14)$$

Em que:

- $C_{CLASSES} = \{c_1, c_2, c_3, c_4, c_5, c_6\}$
  - $c_1 =^{c_CLASSES}$ (“Class”, P_1)
    - $P_1 = \{p_{11}, p_{12}, p_{13}, p_{14}\}$
      - p_{11} = (“Name”, e_1, 1..1)
      - p_{12} = (“Attributes”, c_2, 0..*)
      - p_{13} = (“Operations”, c_3, 0..*)
      - p_{14} = (“Stereotype”, e_1, 0..1)
  - $c_2 =^{c_CLASSES}$ (“Attribute”, P_2)
    - $P_2 = \{p_{21}, p_{22}, p_{23}, p_{24}\}$
      - p_{21} = (“Name”, e_1, 1..1)
      - p_{22} = (“Type”, e_1, 1..1)
      - p_{23} = (“Multiplicity”, e_2, 1..1)
      - p_{24} = (“Visibility”, e_3, 1..1)

- $c_3 =^{c_CLASSES}$ (“Operation”, P_3)
  - $P_3 = \{p_{31}, p_{32}, p_{33}, p_{34}, p_{35}\}$
    - p_{31} = (“Name”, e_1, 1..1)
    - p_{32} = (“Type”, e_1, 1..1)
    - p_{33} = (“Visibility”, e_3, 1..1)
    - p_{34} = (“Arguments”, c_4, 0..*)
    - p_{35} = (“Implementation”, e_1, 1..1)
- $c_4 =^{c_CLASSES}$ (“Argument”, P_4)
  - $P_4 = \{p_{41}, p_{42}, p_{43}\}$
    - p_{41} = (“Name”, e_1, 1..1)
    - p_{42} = (“Type”, e_1, 1..1)
    - p_{43} = (“ValRef”, e_4, 1..1)
- $c_5 =^{c_CLASSES}$ (“Association”, P_5)
  - $P_5 = \{p_{51}, p_{52}, p_{53}, p_{54}, p_{55}\}$
    - p_{51} = (“Name”, e_1, 1..1)
    - p_{52} = (“AssociationEnd1”, c_1, 1..1)
    - p_{53} = (“MultiplicityEnd1”, e_2, 1..1)
    - p_{54} = (“AssociationEnd2”, c_1, 1..1)
    - p_{55} = (“MultiplicityEnd2”, e_2, 1..1)
- $c_6 =^{c_CLASSES}$ (“Generalization”, P_6)
  - $P_6 = \{p_{61}, p_{62}\}$
    - p_{61} = (“SubClass”, c_1, 1..1)
    - p_{62} = (“SuperClass”, c_2, 1..1)

- $\Gamma_{CLASSES} = \varnothing$
- $E_{CLASSES} = \{e_1, e_2, e_3, e_4\}$
  - $e_1 =^{e_CLASSES}$ (“String”, Σ^*)

- $e_2 =^{e_CLASSES}$ (“EMultiplicity”, $\{(lower, upper) \mid (lower, upper) \in \mathbb{N} \times (\mathbb{N}^+ \cup \{*\})\}$)
- $e_3 =^{e_CLASSES}$ (“EVisibility”, {private, protected, public})
- $e_4 =^{e_CLASSES}$ (“EValueReference”, {value, reference})

O conjunto de restrições $R_{CLASSES}$ não foi apresentado por enquanto, sendo retomado mais adiante na seção 4.6.

Nesse metamodelo têm-se seis metaclasses, nomeadas como *Class*, *Attribute*, *Operation*, *Argument*, *Association* e *Generalization*. Tratam-se dos conceitos mais comuns encontrados no diagrama de classes UML, respectivamente: classe, atributo, operação, argumento, associação e generalização.

É possível enxergar no metamodelo $MM_{CLASSES}$ que a metaclasse *Class* está relacionada a 0, 1 ou vários atributos pela propriedade *Attributes* (vide multiplicidade de p_{12}), e a 0, 1 ou várias operações através da propriedade *Operations* (vide multiplicidade de p_{13}).

Cada atributo (*Attribute*) possui um nome, tipo, multiplicidade e visibilidade, que são as propriedades do conjunto P_2. Observar que esta multiplicidade é uma propriedade da metaclasse *Attribute* que, embora tenha o mesmo significado, não se confunde com a multiplicidade que é o terceiro elemento da tripla que define cada propriedade no metamodelo.

Cada operação (*Operation*) possui essas mesmas propriedades e também um conjunto com um número arbitrário de argumentos, conforme conjunto P_3. A implementação de uma operação foi modelada aqui como uma cadeia de símbolos (*String*), que representa o *slot* de informação no modelo onde se poderia, por exemplo, armazenar o código-fonte em alguma linguagem da implementação da operação. Cada argumento pode ser passado por valor ou por referência, aspecto modelado pela propriedade p_{43} e enumeração e_4.

Uma associação (*Association*) tem a semântica de relacionar duas classes, determinadas pelas propriedades p_{52} e p_{54}. Essas classes são conhecidas como terminais da associação, ou *association-end*. Cada terminal possui uma

multiplicidade, determinadas por p_{53} e p_{55}. Apenas associações binárias são permitidas neste metamodelo.

A metaclasse de generalização (*Generalization*) representa o conceito de herança da orientação a objetos. As propriedades p_{61} e p_{62} identificam a subclasse e a superclasse, respectivamente. A metaclasse *Generalization* tem o mesmo significado que a relação Γ na definição de metamodelo da eq. (1). Entretanto, ambas não se confundem, pois a primeira representa o conceito no nível dos modelos (camada M1), enquanto a segunda o faz no nível dos metamodelos (camada M2).

Nota-se que a definição de metamodelo do SBMM incorpora o conceito de propriedades nas metaclasses, mas não o conceito de operações. Apesar disso, foi possível utilizar a definição para estabelecer o conceito de operações no metamodelo de classes da UML. Ou seja, as operações passaram a ficar disponíveis para as classes (camada M2), mas não estão disponíveis para as metaclasses (camada M3). Isso significa que o SBMM não pode incorporar a capacidade de reflexão por meio de operações, presente no MOF, na camada M3. Mas o fato de esta possibilidade existir na camada M2 e M1 é suficiente para resolver os problemas práticos dos desenvolvedores de software que necessitam de reflexão. Observa-se que outros formalismos como o KM3 e o de Alanen e Porres (2008) também não apresentam essa capacidade.

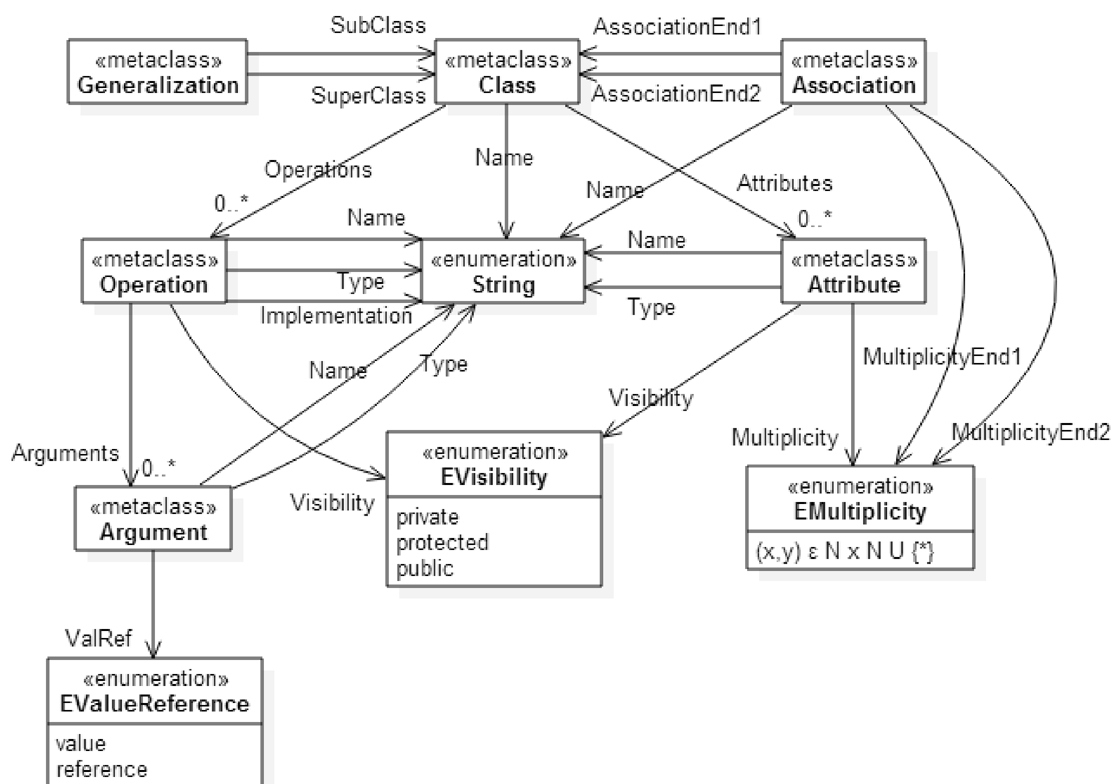
4.4 NOTAÇÃO GRÁFICA PARA METAMODELOS EM SBMM

Embora também sirva como sintaxe concreta, escrever metamodelos como equações de conjuntos e relações não é prático para apresentação e concepção de projetos. A notação apresentada até então tem como foco expressar conceitos e definições precisas que consistem nos fundamentos do formalismo, funcionando como teoria subjacente, e não como ferramenta do dia-a-dia do desenvolvedor.

Pode-se então associar à notação de conjuntos e relações utilizada para definir metamodelos uma sintaxe concreta gráfica semelhante à do MOF, facilitando a visualização e o entendimento dos mesmos, além de aproveitar o padrão já existente

Representam-se as metaclasses e enumerações como retângulos nomeados. As propriedades são denotadas por linhas que ligam a metaclasse ao tipo alvo, que por sua vez pode ser também uma metaclasse ou enumeração. As linhas são rotuladas pelo nome da propriedade e direcionadas com uma seta no tipo alvo. Nesta notação, o metamodelo $MM_{CLASSES}$ seria representado pela Figura 13.

Figura 13 – Representação de $MM_{CLASSES}$ em notação gráfica



Fonte: autor

Os retângulos correspondentes a metaclasses são identificados pelo estereótipo *«metaclass»* acima do nome, enquanto os retângulos correspondentes a enumerações são indicados com *«enumeration»*. Os valores permitidos em cada enumeração são apresentados, um por linha, em um compartimento abaixo do nome no retângulo da enumeração. Nas enumerações com um grande número de valores ou infinitos valores, eles são descritos através de uma expressão matemática, ou foram omitidos quando o entendimento é trivial.

Generalizações, não presentes neste exemplo, são denotadas por uma linha com uma flecha fechada, partindo da submetaclasse em direção à supermetaclasse.

Poderiam ser utilizadas generalizações para definir tipos específicos de associações, tais como agregação, adicionando uma nova metaclasse (submetaclasse de c_5) ao metamodelo, e nesta introduzindo as propriedades necessárias.

Outra ideia para utilizar generalizações seria definir uma metaclasse que representa uma relação genérica, ligando duas metaclasses, e definir c_5 (associação) e c_6 (generalização) como submetaclasses da primeira.

Para estabelecer mais precisamente essa sintaxe concreta na forma de notação gráfica, apresenta-se inicialmente a definição de um grafo direcionado G relacionado ao metamodelo MM, ou seja, G(MM). Utilizando a mesma forma de definição do KM3, um grafo direcionado é uma tripla:

$$G = (N_G, E_G, \rho_G)$$

Em que N_G é um conjunto finito de nós, E_G um conjunto finito de arestas e $\rho_G: E_G \rightarrow N_G \times N_G$ uma função que mapeia arestas em seus pares de nós origem e destino. Toma-se como ponto de partida um metamodelo genérico MM para construir G(MM):

$$MM = (name_{MM}, C, \Gamma, E, R, descriptor_c, descriptor_e)$$

Para cada metaclasse $c_j \in C$, introduz-se um nó no grafo G(MM) denotado por $n_G(c_j)$. Faz-se o mesmo para cada enumeração $e_j \in E$, de modo que $N_G = \{n_G(c_1), n_G(c_2), ..., n_G(c_{|C|})\} \cup \{n_G(e_1), n_G(e_2), ..., n_G(e_{|E|})\}$.

Para cada generalização $\gamma_j = (c_x, c_y) \in \Gamma$, introduz-se uma aresta no grafo G(MM) denotada por $e_G(\gamma_j)$ de modo que $\rho_G(e_G(\gamma_j)) = (n_G(c_x), n_G(c_y))$.

Para cada propriedade $p_{jx} \in P_j$ (com $j \leq |C|$ e $x \leq |P_j|$), adiciona-se nova aresta denotada por $e_G(p_{jx})$ tal que $\rho_G(e_G(p_{jx})) = (n_G(c_j), n_G(t_{jx}))$, lembrando que c_j é a metaclasse que possui a propriedade p_{jx} e t_{jx} é o tipo alvo de p_{jx}. Desse modo, tem-se:

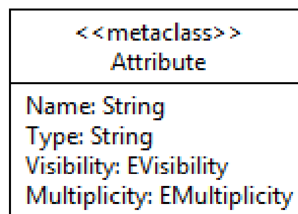
$$E_G = \{e_G(\gamma_1), \dots, e_G(\gamma_{|\Gamma|})\} \cup \{e_G(p_{11}), \dots, e_G(p_{1|P_1|})\} \cup \dots \cup \{e_G(p_{|C|1}), \dots, e_G(p_{|C||P_{|C|}|})\}$$

Através da rotulagem apropriada dos nós e arestas de G(MM), conforme os elementos que levaram à sua construção, e também da introdução de informações de posicionamento dos símbolos gráficos que compõem a sintaxe concreta da representação visual do grafo, uma ferramenta de software pode renderizar o diagrama do metamodelo MM.

Observa-se que o conjunto de restrições R não participa da composição do grafo G(MM). Isso é equivalente ao fato de as restrições em UML e MOF serem definidas à parte através da linguagem OCL, não aparecendo nos diagramas de classes.

Para evitar uma grande quantidade de linhas representando propriedades, a notação da UML e do MOF permite inseri-las em um compartimento dentro do mesmo retângulo da metaclasse, separada de seu nome por uma linha, da mesma forma que os literais permitidos na enumeração *EVisibility* estão representados na Figura 13. A metaclasse ou enumeração de destino é indicada após o nome da propriedade, separada por dois-pontos. Na verdade, essa é a forma mais usual de denotar propriedades, e pode ser adotada também na notação gráfica do SBMM conforme exemplo da Figura 14.

Figura 14 – Representação de uma metaclasse com as propriedades em compartimento próprio



Fonte: autor

4.5 METAMODELOS BEM FORMADOS

Ao longo das definições apresentadas nas seções anteriores, diversas restrições na composição do metamodelo e seus conjuntos e elementos foram estabelecidas. Uma sétupla de metamodelo só é válida se cumprir todas essas condições. Nesse caso, diz-se então que o metamodelo está bem formado, usando nomenclatura análoga àquela dos arquivos XML para conceito semelhante. Esta seção apresenta um resumo dessas restrições, sendo útil para a construção de ferramentas de metamodelagem que utilizam o SBMM como formalismo subjacente.

- $name_{MM} \in \Sigma^+$: o nome do metamodelo é uma cadeia não-vazia;
- $\forall c_j \in C\ ((c_j, c_j) \notin \Gamma^+)$: uma metaclasse não pode ser submetaclasse de si mesma, direta ou transitivamente;
- $\forall c_j \in C\ (metaclassname_j \in \Sigma^+)$: o nome de toda metaclasse é uma cadeia não-vazia;
- $\forall c_j, c_k \in C\ ((metaclassname_j = metaclassname_k) \Rightarrow (c_j = c_k))$: não pode haver metaclasses com o mesmo nome no mesmo metamodelo;
- $\forall propertyname, propertytype, mult\ (((propertyname, propertytype, mult) \in P_1 \cup ... \cup P_{|C|}) \Rightarrow (propertyname \in \Sigma^+))$: o nome de toda propriedade é uma cadeia não-vazia;
- $\forall metaclassname, P\ ((metaclassname, P) \in Im(descriptor_c)) \Rightarrow (\forall propertyname_x, propertyname_y, propertytype_x, propertytype_y, mult_x, mult_y\ (((propertyname_x, propertytype_x, mult_x) \in P) \wedge ((propertyname_y, propertytype_y, mult_y) \in P) \wedge (propertyname_x = propertyname_y)) \Rightarrow (propertytype_x = propertytype_y) \wedge (mult_x = mult_y))$: não pode haver propriedades com o mesmo nome na mesma metaclasse. $Im(descriptor_c)$ denota o conjunto imagem da função $descriptor_c$;
- $\forall e_j, e_k \in E\ ((enumname_j = enumname_k) \Rightarrow (e_j = e_k))$: não pode haver enumerações com o mesmo nome no mesmo metamodelo;

- $\forall c_j \in C, e_k \in E$ ($metaclassname_j \neq enumname_k$) : não pode haver metaclasses e enumerações com o mesmo nome no mesmo metamodelo;
- $descriptor_c$ precisa estar definida para toda metaclasse $c \in C$, ou seja, $descriptor_c$ é uma função total sobre C;
- $descriptor_e$ precisa estar definida para toda enumeração $e \in E$, ou seja, $descriptor_e$ é uma função total sobre E;

4.6 RESTRIÇÕES SOBRE MODELOS

Foi mencionado na definição de metamodelo que o conjunto R serve para estabelecer restrições sobre os modelos conformes ao metamodelo. Cada restrição $r_j \in R$ consiste em uma sentença em lógica de primeira ordem na qual metaclasses e enumerações são representadas por predicados unários a serem aplicados sobre elementos existentes no modelo. Propriedades são representadas por funções que associam o valor contido no respectivo *slot* de informação do elemento do modelo em seu argumento. Se a propriedade possui multiplicidade que permite múltiplos valores, a função retorna um conjunto com todos os valores e não um único valor, no caso de o tipo ser uma metaclasse. Caso o tipo seja uma enumeração, é retornada uma lista com os múltiplos valores. Constantes que são valores de qualquer enumeração presente no metamodelo são permitidas nas expressões.

Ou seja, cada sentença $r_j \in R$ é uma cadeia válida dentro das regras das sentenças de lógica de primeira ordem. O universo de objetos, nesse contexto, são as instâncias dos modelos, a serem detalhadas na seção 4.7, e também números, cadeias, valores de enumerações, conjuntos e tuplas. Convenciona-se que os predicados são denotados pelos nomes das metaclasses e enumerações, e as funções são denotadas pelos nomes das propriedades precedidos pelo nome da metaclasse à qual pertence separado por um ponto. Isso porque pode haver propriedades de mesmo nome em metaclasses diferentes, e dessa forma não há ambiguidade.

Se a sentença contiver predicados ou funções que não correspondam às metaclasses, enumerações e propriedades, então a restrição é inválida no

metamodelo, pois ela não pode ser resolvida, mesmo que a sentença seja sintaticamente correta.

Se aplicada uma função correspondente a uma propriedade em um objeto cujo tipo não corresponda à metaclasse dona da propriedade, ou se aplicada sobre um objeto que seja uma enumeração, a função retorna um conjunto vazio. Russel e Norvig (2004) mencionam que essa é uma solução técnica para que todas as funções que possam ser usadas nas sentenças sejam totais, isto é, estejam definidas para quaisquer objetos, sendo um requisito da lógica de primeira ordem.

Por exemplo, a seguinte sentença denota que não é permitido que dois argumentos de uma mesma operação tenham o mesmo nome em modelos conformes ao metamodelo $MM_{CLASSES}$:

$$\forall o, a_1, a_2\ Operation(o) \wedge Argument(a_1) \wedge Argument(a_2) \wedge \\ (a_1 \in Operation.Arguments(o)) \wedge (a_2 \in Operation.Arguments(o)) \wedge \\ (Argument.Name(a_1) = Argument.Name(a_2)) \Rightarrow a_1 = a_2$$

A leitura em linguagem natural para a sentença acima é: para toda operação o e argumentos a_1 e a_2, se a_1 e a_2 são argumentos de o e possuem o mesmo nome, então a_1 deve ser igual a a_2.

Seguindo esse raciocínio, estabelecem-se outras restrições para o metamodelo de exemplo $MM_{CLASSES}$ que, enfatizando, aplicam-se sobre os modelos que são instâncias deste metamodelo. Seja $R_{CLASSES} = \{r_1, r_2, r_3, r_4, r_5, r_6, r_7, r_8, r_9, r_{10}\}$. Uma interpretação em linguagem natural é apresentada para cada sentença.

$$r_1: \forall c\ Class(c) \Rightarrow \neg(Class.Name(c) = \varepsilon)$$

Interpretação de r_1: Nenhuma classe pode ter o nome vazio, em que ε é a constante que denota a cadeia (*string*) vazia.

$$r_2: \forall c_1, c_2\ Class(c_1) \wedge Class(c_2) \wedge (Class.Name(c_1) = Class.Name(c_2)) \Rightarrow c_1 = c_2$$

Interpretação de r_2: Não pode haver mais de uma classe com o mesmo nome.

$$r_3: \forall a\ Attribute(a) \Rightarrow \neg(Attribute.Name(a) = \varepsilon)$$

Interpretação de r_3: Nenhum atributo pode ter o nome vazio.

$$r_4: \forall c, a_1, a_2\ Class(c) \wedge Attribute(a_1) \wedge Attribute(a_2) \wedge (a_1 \in Class.Attributes(c)) \wedge (a_2 \in Class.Attributes(c)) \wedge (Attribute.Name(a_1) = Attribute.Name(a_2)) \Rightarrow a_1 = a_2$$

Interpretação de r_4: Não pode haver mais de um atributo com o mesmo nome na mesma classe.

$$r_5: \forall o\ Operation(o) \Rightarrow \neg(Operation.Name(o) = \varepsilon)$$

Interpretação de r_5: Nenhuma operação pode ter o nome vazio.

$$r_6: \forall c, a_1, a_2\ Class(c) \wedge Operation(a_1) \wedge Operation(a_2) \wedge (a_1 \in Class.Operations(c)) \wedge (a_2 \in Class.Operations(c)) \wedge (Operation.Name(a_1) = Operation.Name(a_2)) \Rightarrow a_1 = a_2$$

Interpretação de r_6: Não pode haver mais de uma operação com o mesmo nome na mesma classe. Embora na orientação a objetos exista o conceito de sobrecarga (*overload*), que permite mais de uma operação com o mesmo nome na mesma classe, desde que com argumentos distintos, para fins de simplificação neste metamodelo a sobrecarga não é permitida pela restrição r_6.

$$r_7: \forall a\ Argument(a) \Rightarrow \neg(Argument.Name(a) = \varepsilon)$$

Interpretação de r_7: Nenhum argumento de operação pode ter o nome vazio.

$$r_8: \forall o, a_1, a_2\ Operation(o) \wedge Argument(a_1) \wedge Argument(a_2) \wedge (a_1 \in Operation.Arguments(o)) \wedge (a_2 \in Operation.Arguments(o)) \wedge (Argument.Name(a_1) = Argument.Name(a_2)) \Rightarrow a_1 = a_2$$

Interpretação de r_8: Não pode haver mais de um argumento com o mesmo nome em uma mesma operação (já visto anteriormente).

$$r_9: \forall a\ Association(a) \Rightarrow \neg(Association.Name(a) = \varepsilon)$$

Interpretação de r_9: Nenhuma associação pode ter o nome vazio.

$$r_{10}: \forall g\ Generalization(g) \Rightarrow \neg(Generalization.SubClass(g) = Generalization.SuperClass(g))$$

Interpretação de r_{10}: Uma classe não pode ser subclasse de si mesma.

Em uma aplicação real, mais algumas restrições seriam necessárias, tais como não permitir nomes de classes, atributos, etc. que contenham espaços ou que comecem com caracteres correspondentes a dígitos, como é usual nas linguagens de programação. Para fins de simplificação deste exemplo, restrições deste tipo não são apresentadas. Percebe-se que uma biblioteca de funções de manipulação de cadeias, ou operadores provenientes da teoria de linguagens, seriam de grande utilidade para compor essas expressões.

Foi mencionado que metaclasses e enumerações são vistas nas restrições como predicados unários sobre elementos do modelo. Define-se que o símbolo * (asterisco) seguido do nome de uma metaclasse refere-se apenas a instâncias diretas, e não de submetaclasses. Por exemplo, a expressão MyMetaclass(x) é verdadeira se *x* for um elemento do modelo que é instância de *MyMetaclass* ou qualquer uma de suas submetaclasses, transitivamente. Já a expressão MyMetaclass*(x) é verdadeira apenas se *x* for uma instância direta de *MyMetaclass*, sendo falsa mesmo se for instância de alguma de suas eventuais submetaclasses.

Esse recurso permite o SBMM suportar o conceito de metaclasses abstratas do MOF. Uma metaclasse abstrata não pode ter instâncias diretas, sendo usada apenas como base para definir outras metaclasses. Desse modo, a restrição estabelecida pela expressão $\neg\exists$x MyMetaclass*(x) significa que *MyMetaclass* é uma metaclasse abstrata. No MOF, uma metaclasse abstrata é representada pela utilização de fonte itálica em seu nome, dentro do bloco que a representa em sua notação gráfica.

4.7 MODELOS EM SBMM

Um modelo M é um objeto definido como uma quádrupla conforme a eq. (15).

$$M = (name_M, name_{MM}, I, descriptor_I) \quad (15)$$

Em que:

- $name_M \in \Sigma^+$ é uma cadeia não-vazia que representa o nome do modelo. O conjunto Σ é suposto ser o mesmo daquele apresentado no contexto dos metamodelos;
- $name_{MM} \in \Sigma^+$ é uma cadeia não-vazia que representa o nome do metamodelo alvo pretendido. Diferentemente de outros formalismos apresentados, cujas definições de modelo fazem referência direta (*hard reference*) ao objeto que representa seu metamodelo, aqui o metamodelo correspondente é apenas referenciado pelo seu nome como uma cadeia de símbolos (*soft reference*). Isso permite que um modelo M exista mesmo sem um metamodelo, já que a localização do objeto do metamodelo correspondente ao nome $name_{MM}$ é um problema à parte, a ser atribuído a alguma função. Enquanto existe sem metamodelo, o modelo carece de uma interpretação, já que a semântica de um modelo está atrelada à semântica das metaclasses de seu metamodelo. Entretanto, a existência de um modelo de maneira independente permite maior facilidade na implementação de ferramentas nas quais um mesmo modelo pode ser interpretado de diferentes formas quando enxergados sob o

contexto de diferentes metamodelos, ou sob o contexto de diferentes versões do mesmo metamodelo que evolui com o tempo;

- $I = \{i_1, ..., i_a\}$ é o conjunto finito, eventualmente vazio, de instâncias ou elementos do modelo. Será utilizada a nomenclatura instância ou elemento indistintamente no texto, mas optou-se pela letra I para não haver ambiguidades com o conjunto E de enumerações do metamodelo. O número de instâncias no conjunto é dado por $a = | I |$, em que $| I |$ é a cardinalidade de I;
- $descriptor_i$ é uma função que mapeia cada instância $i_j \in I$ em seu descritor, que é dado por uma tripla composta por um nome identificador único, o nome do tipo base e um conjunto de *slots*, a ser detalhado e exemplificado a seguir.

Cada instância $i_j \in I$, com $1 \leq j \leq a$ possui um tipo base (*base type*), conforme definido anteriormente na seção 4.2, que corresponde a uma metaclasse do metamodelo alvo.

Para preservar a definição de modelo independente de qualquer metamodelo, ou seja, sem fazer referência direta a conjuntos ou objetos externos a M (com exceção do conjunto de símbolos Σ, que é universal), uma instância i_j é mapeada a uma tripla ordenada descritiva conforme a eq. (16):

$$descriptor_i(i_j) = (instancename_j, metaclassname_j, S_j) \qquad (16)$$

Em que:

- $instancename_j \in \Sigma^+$ é uma cadeia não-vazia que representa o nome da instância. Esse nome deve ser único para as instâncias do conjunto I e funciona como identificador da instância no modelo;
- $metaclassname_j \in \Sigma^+$ é uma cadeia não-vazia que representa o nome da metaclasse do tipo base. Quando o modelo é contextualizado sob a

interpretação de algum metamodelo específico, é tarefa de alguma função retornar o objeto correspondente à metaclasse a partir desse nome;

- $S_j = \{s_{j1}, ..., s_{jb}\}$ é o conjunto finito, eventualmente vazio, de *slots* de dados. Cada *slot* s_{jx} representa um valor atribuído a alguma propriedade efetiva da metaclasse tipo base para a instância em questão i_j. Recordando, as propriedades efetivas de uma metaclasse são as definidas por ela mesma mais aquelas definidas por suas supermetaclasses, transitivamente, conforme as eqs. (5) e (7). O número de *slots* no conjunto S_j é dado por $b = |S_j|$.

Cada *slot* s_{jx} deve referenciar a propriedade efetiva à qual corresponde, bem como indicar o valor que o modelo carrega para ela. Ou seja, um *slot* é um objeto definido pelo par da eq. (17).

$$s_{jx} = (propertyname_{jx}, val_{jx}) \quad (17)$$

Em que:

- $propertyname_{jx} \in \Sigma^+$ é uma cadeia não-vazia que representa o nome da propriedade efetiva do tipo base ao qual o *slot* s_{ij} se refere. Não podem ocorrer dois *slots* s_{jx} e $s_{jy} \in S_j$ tais que $propertyname_{jx} = propertyname_{jy}$, ou seja, só pode haver no máximo um *slot* por propriedade efetiva na mesma instância. Quando o modelo é contextualizado sob a interpretação de algum metamodelo específico, é tarefa de alguma função retornar o objeto correspondente à propriedade a partir desse nome;
- $val_{jx} \in (\Sigma \cup \{\#\})^* \cup 2^I$ é o valor que o *slot* s_{jx} carrega, atribuído à propriedade identificada por $propertyname_{jx}$, tal que $\# \notin \Sigma$.

É fornecida a seguir uma explicação mais detalhada para val_{jx}, buscando melhor compreensão do significado da expressão que determina o conjunto ao qual ele pertence.

O termo val_{jx} se trata do valor atribuído à propriedade de nome $propertyname_{jx}$ para a instância do modelo i_j. A propriedade referenciada por $propertyname_{jx}$ está definida na metaclasse referenciada por $metaclassname_j$ (ou em alguma de suas supermetaclasses) no metamodelo correspondente. A propriedade tem um tipo alvo, que pode ser uma enumeração ou metaclasse. Além disso, essa propriedade tem uma multiplicidade, que define os limites inferior e superior da quantidade de valores que pode conter. Sendo assim, val_{jx} representa um valor de enumeração (ou uma lista deles, se a propriedade tiver multiplicidade não unitária), ou então uma instância do modelo que tem uma metaclasse como tipo (ou um conjunto de instâncias, se a multiplicidade for não unitária), não necessariamente tipo base. Isso determina que $val_{jx} \in (\Sigma \cup \{\#\})^* \cup 2^I$, conforme detalhado nos parágrafos seguintes.

Como a definição de modelo é construída de modo que eles existam independentemente de metamodelo, embora enfatizando que a interpretação do modelo requer obrigatoriamente um metamodelo, então nota-se que o modelo não tem "conhecimento" sobre os valores de enumerações que podem ser referenciados.

Dessa forma, deve-se ter uma convenção geral para codificar valores de enumerações que, sob determinada interpretação, podem ser convertidos nos valores de fato da enumeração correspondente. Esse fato é similar ao caso das máquinas de Turing universais. Uma máquina de Turing é capaz de operar sobre uma cadeia cujos símbolos pertencem a um alfabeto de fita. Uma máquina de Turing universal emula outra máquina de Turing, e para isso existe uma convenção para codificar máquinas de Turing como sequencias de símbolos, que por sua vez servem como entradas para as máquinas de Turing universais (SIPSER, 2007). Cada sequência desse tipo, quando sozinha, é apenas uma cadeia de símbolos. Porém, quando interpretada à luz da convenção, pode ser entendida como uma máquina de Turing e até mesmo convertida em sua representação formal usual.

Os valores de qualquer enumeração são representados por uma cadeia de símbolos. Por exemplo, o conjunto dos booleanos $\mathcal{B}$ = {true, false} possui dois valores atômicos. As cadeias "true" e "false", por sua vez, são representações textuais para esses valores. Para o conjunto dos números naturais $\mathbb{N}$, também tem-se a representação usual na forma de dígitos decimais utilizando cadeias sobre o alfabeto {"0", "1", ..., "9"}, no qual cada dígito está representado entre aspas para explicitar que se tratam dos símbolos, e não dos números de 0 a 9.

Como os valores de cada enumeração sempre podem ser codificados na forma de cadeias de símbolos, resta então determinar uma codificação para listas, afinal as propriedades podem ter multiplicidade não unitária. Para isso, é introduzido um símbolo especial # não pertencente a Σ, que serve como separador. Por exemplo, para codificar uma lista de três números inteiros composta por 52, 68 e 327, é utilizada a cadeia "52#68#327". Dessa forma, $val_{jx} \in (\Sigma \cup \{\#\})^*$ para enumerações.

Convenientemente, define-se uma função auxiliar específica para cada metamodelo MM, denominada $decode_{MM}$, capaz de decodificar uma cadeia na sequência de valores de enumeração que representa, de modo que:

$$decode_{MM}: (\Sigma \cup \{\#\})^* \rightarrow L_1^* \cup L_2^* \cup \ldots \cup L_{|E|}^* \tag{18}$$

Sendo |E| o número de enumerações do metamodelo MM e, recordando, L_j o conjunto dos valores permitidos para a enumeração $e_j \in E$. L_j^* é o fecho transitivo e reflexivo de L, representando o conjunto de todas as listas que podem ser formadas pelos valores de L, com um número arbitrário de elementos. A função $decode_{MM}$ volta a ser utilizada mais adiante, na seção 4.9, dentro do tópico sobre conformidade de modelos.

Por outro lado, se o tipo da propriedade referenciada por $propertyname_{jx}$ for uma metaclasse, então val_{jx} deve ser alguma instância existente no modelo e, portanto, pertencente ao conjunto I. Entretanto, como a multiplicidade pode ser não unitária, então val_{jx} pode carregar um subconjunto das instâncias, inclusive o conjunto vazio. No caso de multiplicidade unitária, então obrigatoriamente val_{jx} deve ser um conjunto unitário. Sendo assim, então $val_{jx} \in 2^I$ para instâncias de metaclasses. Observa-se que a mesma instância não pode aparecer duas vezes dentro de val_{jx}, ao contrário do caso das enumerações, em que o mesmo valor pode aparecer mais de uma vez na lista, tal como no exemplo codificado pela cadeia "52#68#52#327".

Conclui-se então que, no caso geral, $val_{jx} \in (\Sigma \cup \{\#\})^* \cup 2^I$, lembrando que $\# \notin \Sigma$.

O valor do *slot* também tem seu equivalente no formalismo definido por Alanen e Porres (2008). Entretanto, suas definições não incluem as enumerações, permitindo apenas instâncias como valores para os *slots*.

Assim como se definiram os operadores $=^c$ e $=^e$ na seção 4.2 para simplificar a notação e tornar implícitas as funções $descriptor_c$ e $descriptor_e$, também se define aqui de maneira análoga o operador $=^i$ que torna implícita a função $descriptor_i$.

4.8 EXEMPLO DE MODELO DESCRITO EM SBMM

Considere-se o modelo representado pelo diagrama de classes UML do sistema escolar da Figura 4. A seguir, ele é apresentado através do SBMM, referenciando o metamodelo $MM_{CLASSES}$ como base.

$$M_{ESCOLAR} = (\text{“ClassModel”}, \text{“ClassMetamodel”}, I_{ESCOLAR}, descriptor_{i_ESCOLAR}) \quad (19)$$

Em que:

- $I_{ESCOLAR} = \{i_1, i_2, i_3, i_4, i_5, i_6, i_7, i_8, i_9, i_{10}, i_{11}, i_{12}, i_{13}, i_{14}, i_{15}\}$
  - $i_1 =^{i_ESCOLAR}$ (“CCurso”, “Class”, S_1)
    - $S_1 = \{s_{11}, s_{12}, s_{13}, s_{14}\}$
      - s_{11} = (“Name”, “CCurso”)
      - s_{12} = (“Attributes”, $\{i_2, i_3\}$)
      - s_{13} = (“Operations”, $\emptyset$)
      - s_{14} = (“Stereotype”, “persistent”)
  - $i_2 =^{i_ESCOLAR}$ (“CCurso.Nome”,“Attribute”, S_2)
    - $S_2 = \{s_{21}, s_{22}, s_{23}, s_{24}\}$
      - s_{21} = (“Name”, “Nome”)
      - s_{22} = (“Type”, “String”)
      - s_{23} = (“Multiplicity”, “(1,1)”)
      - s_{24} = (“Visibility”, “public”)
  - $i_3 =^{i_ESCOLAR}$ (“CCurso.Sigla”,“Attribute”, S_3)

- $S_3 = \{s_{31}, s_{32}, s_{33}, s_{34}\}$
  - s_{31} = (“Name”, “Sigla”)
  - s_{32} = (“Type”, “String”)
  - s_{33} = (“Multiplicity”, “(1,1)”)
  - s_{34} = (“Visibility”, “public”)

- $i_4 =^{i_ESCOLAR}$ (“CAluno”,“Class”, S_4)
  - $S_4 = \{s_{41}, s_{42}, s_{43}\}$
    - s_{41} = (“Name”, “CAluno”)
    - s_{42} = (“Attributes”, $\{i_5, i_6\}$)
    - s_{43} = (“Operations”, $\varnothing$)
    - s_{44} = (“Stereotype”, “persistent”)
- $i_5 =^{i_ESCOLAR}$ (“CAluno.Nome”,“Attribute”, S_5)
  - $S_5 = \{s_{51}, s_{52}, s_{53}, s_{54}\}$
    - s_{51} = (“Name”, “Nome”)
    - s_{52} = (“Type”, “String”)
    - s_{53} = (“Multiplicity”, “(1,1)”)
    - s_{54} = (“Visibility”, “public”)
- $i_6 =^{i_ESCOLAR}$ (“CAluno.Endereco”,“Attribute”, S_6)
  - $S_6 = \{s_{61}, s_{62}, s_{63}, s_{64}\}$
    - s_{61} = (“Name”, “Endereco”)
    - s_{62} = (“Type”, “String”)
    - s_{63} = (“Multiplicity”, “(1,1)”)
    - s_{64} = (“Visibility”, “public”)
- $i_7 =^{i_ESCOLAR}$ (“CTurma”,“Class”, S_7)
  - $S_7 = \{s_{71}, s_{72}, s_{73}\}$
    - s_{71} = (“Name”, “CTurma”)

- s_{72} = (“Attributes”, {i_8, i_9})
- s_{73} = (“Operations”, ∅)
- s_{74} = (“Stereotype”, “persistent”)

- $i_8 =^{i_ESCOLAR}$ (“CTurma.Semestre”,“Attribute”, S_8)
  - S_8 = {s_{81}, s_{82}, s_{83}, s_{84}}
    - s_{81} = (“Name”, “Semestre”)
    - s_{82} = (“Type”, “Integer”)
    - s_{83} = (“Multiplicity”, “(1,1)”)
    - s_{84} = (“Visibility”, “public”)
- $i_9 =^{i_ESCOLAR}$ (“CTurma.Ano”,“Attribute”, S_9)
  - S_9 = {s_{91}, s_{92}, s_{93}, s_{94}}
    - s_{91} = (“Name”, “Ano”)
    - s_{92} = (“Type”, “Integer”)
    - s_{93} = (“Multiplicity”, “(1,1)”)
    - s_{94} = (“Visibility”, “public”)
- $i_{10} =^{i_ESCOLAR}$ (“CProfessor”,“Class”, S_{10})
  - S_{10} = {$s_{10,1}$, $s_{10,2}$, $s_{10,3}$}
    - $s_{10,1}$ = (“Name”, “CProfessor”)
    - $s_{10,2}$ = (“Attributes”, {i_{11}, i_{12}})
    - $s_{10,3}$ = (“Operations”, ∅)
    - $s_{10,4}$ = (“Stereotype”, “persistent”)
- $i_{11} =^{i_ESCOLAR}$ (“CProfessor.Nome”,“Attribute”, S_{11})
  - S_{11} = {$s_{11,1}$, $s_{11,2}$, $s_{11,3}$, $s_{11,4}$}
    - $s_{11,1}$ = (“Name”, “Nome”)
    - $s_{11,2}$ = (“Type”, “String”)
    - $s_{11,3}$ = (“Multiplicity”, “(1,1)”)

- $s_{11,4}$ = (“Visibility”, “public”)

- $i_{12} =^{i_ESCOLAR}$ (“CProfessor.Endereco”,“Attribute”, S_{12})
  - $S_{12} = \{s_{12,1}, s_{12,2}, s_{12,3}, s_{12,4}\}$
    - $s_{12,1}$ = (“Name”, “Endereco”)
    - $s_{12,2}$ = (“Type”, “String”)
    - $s_{12,3}$ = (“Multiplicity”, “(1,1)”)
    - $s_{12,4}$ = (“Visibility”, “public”)
- $i_{13} =^{i_ESCOLAR}$ (“CCurso_CTurma”,“Association”, S_{13})
  - $S_{13} = \{s_{13,1}, s_{13,2}, s_{13,3}, s_{13,4}, s_{13,5}\}$
    - $s_{13,1}$ = (“Name”, “CCurso_CTurma”)
    - $s_{13,2}$ = (“AssociationEnd1”, $\{i_1\}$)
    - $s_{13,3}$ = (“MultiplicityEnd1”, “(1,1)”)
    - $s_{13,4}$ = (“AssociationEnd2”, $\{i_7\}$)
    - $s_{13,5}$ = (“MultiplicityEnd2”, “(0,*)”)
- $i_{14} =^{i_ESCOLAR}$ (“CAluno_CTurma”,“Association”, S_{14})
  - $S_{14} = \{s_{14,1}, s_{14,2}, s_{14,3}, s_{14,4}, s_{14,5}\}$
    - $s_{14,1}$ = (“Name”, “CAluno_CTurma”)
    - $s_{14,2}$ = (“AssociationEnd1”, $\{i_4\}$)
    - $s_{14,3}$ = (“MultiplicityEnd1”, “(1,40)”)
    - $s_{14,4}$ = (“AssociationEnd2”, $\{i_7\}$)
    - $s_{14,5}$ = (“MultiplicityEnd2”, “(0,*)”)
- $i_{15} =^{i_ESCOLAR}$ (“CTurma_CProfessor”,“Association”, S_{15})
  - $S_{15} = \{s_{15,1}, s_{15,2}, s_{15,3}, s_{15,4}, s_{15,5}\}$
    - $s_{15,1}$ = (“Name”, “CTurma_CProfessor”)
    - $s_{15,2}$ = (“AssociationEnd1”, $\{i_7\}$)
    - $s_{15,3}$ = (“MultiplicityEnd1”, “(0,*)”)

- $s_{15,4}$ = (“AssociationEnd2”, {i_{10}})
- $s_{15,5}$ = (“MultiplicityEnd2”, “(1,1)”)

Pode-se observar que as quatro classes presentes no modelo, *CCurso*, *CAluno*, *CTurma* e *CProfessor*, estão representadas pelas instâncias i_1, i_4, i_7 e i_{10}, respectivamente. Todas elas possuem a metaclasse referenciada pela cadeia “Class” como tipo base. Cada uma dessas instâncias possui dois atributos e nenhuma operação. Por isso, os *slots* s_{12}, s_{42}, s_{72} e $s_{10,2}$ possuem dois elementos no conjunto de valores cada, referenciando as instâncias correspondentes aos atributos. Enquanto isso, o conjunto de valores de s_{13}, s_{43}, s_{73} e $s_{10,3}$, que referenciariam as operações, é o conjunto vazio.

As associações presentes no diagrama da Figura 4 estão representadas pelas instâncias i_{12} a i_{15}. Apesar de não aparecer nenhum nome de associação no diagrama, como a restrição r_9 do metamodelo estabelece que as associações não podem ter o nome vazio, então em cada *slot* correspondente a nome de associação ($s_{12,1}$, $s_{13,1}$, $s_{14,1}$ e $s_{15,1}$) foi introduzido um nome.

4.9 RELAÇÃO DE CONFORMIDADE

A relação de conformidade entre modelo e metamodelo estabelece se o primeiro está descrito conforme as regras definidas pelo segundo. Outros formalismos apresentados no capítulo 3 também definem suas relações de conformidade dentro de seus contextos. Essa relação é importante para a construção de ferramentas: assim como um compilador precisa analisar sintaticamente um código-fonte com respeito à gramática da linguagem de programação, uma ferramenta de modelagem precisa garantir que os modelos criados estão de acordo com os respectivos metamodelos desejados. Jackson, Levendovszky e Balasubramanian (2013) definem operadores de construção de modelos que adicionam ou removem elementos de modo que o modelo sempre permanece conforme a cada alteração. Dessa forma, uma ferramenta que aplica esse conceito garante que o modelo em construção está sempre em conformidade ao metamodelo.

O conceito de conformidade também é importante para ferramentas de transformação de modelos. É necessário garantir que existe conformidade entre modelo e metamodelo antes de operar sobre o modelo.

Nesta seção é apresentada a relação de conformidade do SBMM. Sejam M um modelo e MM um metamodelo quaisquer tais que:

$$MM = (name_{MM}, C, \Gamma, E, R, descriptor_c, descriptor_e)$$

$$M = (name_M, name_{MM}, I, descriptor_i)$$

A relação de conformidade é definida como uma função denominada conformsTo.

$$conformsTo: U_M \times U_{MM} \rightarrow \mathcal{B}$$

Em que U_M representa o universo de modelos, U_{MM} o universo de metamodelos e $\mathcal{B}$ = {true, false} o conjunto de valores booleanos.

Antes de um modelo estar em conformidade com um metamodelo, ele deve ser bem formado por si só. Um modelo é bem formado quando respeita as condições descritas ao longo da seção 4.7, resumidas a seguir:

- $name_M \in \Sigma^+$: o nome do modelo é uma cadeia não-vazia;
- $name_{MM} \in \Sigma^+$: o nome do metamodelo referenciado é uma cadeia não-vazia;
- $instancename_j \in \Sigma^+$: o nome de toda instância é uma cadeia não-vazia;
- $\forall i_j, i_k ((i_j \in I) \wedge (i_k \in I) \wedge (instancename_j = instancename_k)) \Rightarrow i_j = i_k$: não pode haver instâncias com o mesmo nome no mesmo modelo;
- $metaclassname_j \in \Sigma^+$: o nome de toda metaclasse referenciada como tipo base de instância é uma cadeia não-vazia;
- $propertyname_{jx} \in \Sigma^+$: o nome de toda propriedade referenciada em *slots* é uma cadeia não-vazia;

- $\forall$ (instancename, metaclassname, S) $\in$ Im($descriptor_i$) ($\forall$ ($propertyname_1$, val_1), ($propertyname_2$, val_2) $\in$ S (($propertyname_1$ = $propertyname_2$) $\Rightarrow$ (val_1 = val_2))): não pode haver mais de um *slot* referenciando o mesmo nome de propriedade na mesma instância. Im($descriptor_i$) denota o conjunto imagem da função $descriptor_i$;

Por definição, um modelo M está em conformidade com um metamodelo MM se e somente se, além de ser bem formado, as quatro seguintes condições forem satisfeitas: (a) existe a metaclasse referenciada como tipo base para todas as instâncias pertencentes a I; (b) existe propriedade correspondente para todos os *slots* pertencentes aos conjuntos S_j de cada instância de I; (c) os valores de cada *slot* são válidos conforme o tipo base da instância; (d) M satisfaz todas as restrições de R.

A avaliação de cada restrição de R, bem como sua representação, já foi discutida na seção 4.6. Serão, então, detalhadas as condições (a), (b) e (c).

A condição (a) pode ser representada através da seguinte sentença, uma vez que os tipos das instâncias $i_j \in I$ são referenciados por uma cadeia de símbolos que indica o nome identificador da metaclasse no metamodelo. Com as definições e notação das seções 4.2 e 4.7 em mente, tem-se:

$$\forall \text{ (instancename, metaclassname, S)} \in \text{Im}(descriptor_i)$$
$$(\exists \text{ (w, P)} \in \text{Im}(descriptor_c) \text{ (metaclassname = w))}$$

Interpretação: para todo descritor de instância de I (instancename, metaclassname, S), pertencente, portanto, ao conjunto imagem de $descriptor_i$, existe um par (w, P) que é descritor de alguma metaclasse de C tal que o nome do tipo referenciado por metaclassname é igual ao nome w da metaclasse.

A condição (b), por sua vez, requer a introdução da função auxiliar metaclass(MM, metaclassname), que toma um metamodelo e uma cadeia não-vazia para associar o objeto correspondente à metaclasse de mesmo nome que existe dentro do metamodelo. Essa função pode ser definida pela expressão da eq. (20).

$$\text{metaclass}(MM, \text{metaclassname}) = \{ c \mid (c \in C) \wedge ((w, P) = \text{descriptor}_c(c)) \wedge (\text{metaclassname} = w)\} \quad (20)$$

Em um metamodelo bem formado, não há metaclasses com nomes repetidos e, portanto, a função metaclass retorna um conjunto unitário quando existir metaclasse com o nome metaclassname, ou o conjunto vazio quando não existir. A condição (b) pode então ser estabelecida por:

$$\forall\ (\text{instancename}, \text{metaclassname}, S) \in \text{Im}(\text{descriptor}_i)$$
$$(\forall\ (\text{propertyname}, \text{val}) \in S\ (\ \exists\ (w, t, m) \in$$
$$\text{PEFC}_{MM}(\text{metaclass}(MM, \text{metaclassname}))))$$

Interpretação: Para todo *slot* (propertyname, val) de todo descritor de instância (instancename, metaclassname, S), existe uma tripla (w, t, m) que é propriedade efetiva da metaclasse referenciada por metaclassname. Ou seja, o *slot* pode conter valor de uma propriedade direta da metaclasse ou de uma de suas supermetaclasses, transitivamente. A função PEFC_{MM} foi definida pela eq. (7). Observa-se aqui que a função metaclass retorna um conjunto unitário cujo elemento é uma metaclasse. Esse conjunto foi utilizado como argumento da função PEFC_{MM}. Subentende-se que é este único elemento que está sendo passado de fato para a função, já que o domínio de PEFC_{MM} é o conjunto de metaclasses C conforme a eq. (6).

Por fim, a condição (c) determina que todos os valores em *slots* devem ser válidos de acordo com os tipos no metamodelo. Se o tipo for uma enumeração, o valor deve ser uma cadeia codificada que representa um conjunto cujos valores pertencem ao conjunto de valores permitidos da enumeração. Se o tipo for uma metaclasse, o valor deve ser um conjunto de instâncias de tipo (não necessariamente o tipo base) dessa metaclasse. Também deve ser respeitada a multiplicidade de valores permitida pela propriedade.

Introduz-se a função auxiliar property(MM, metaclassname, propertyname), dada pela eq. (21), que retorna a propriedade efetiva de nome propertyname da metaclasse de nome metaclassname do metamodelo MM.

$$\text{property}(MM, \text{metaclassname}, \text{propertyname}) = \{ (w, t, m) \mid ((w, t, m) \in \text{PEFC}_{MM}(\text{metaclass}(MM, \text{metaclassname}))) \wedge (\text{propertyname} = w) \} \quad (21)$$

Em um metamodelo bem formado, não há propriedades com nomes repetidos dentro da mesma metaclasse e, portanto, a função acima retorna um conjunto unitário quando existir propriedade com o nome propertyname na metaclasse de nome metaclassname, ou o conjunto vazio quando não existir. Assim como no caso da função metaclass, em que seu resultado foi subentendido como o único elemento de seu conjunto unitário, o mesmo será feito para property, de modo a simplificar a expressão. A condição (c) pode então ser denotada por:

$$\forall (\text{instancename}, \text{metaclassname}, S) \in \text{Im}(\text{descriptor}_i) (\forall (\text{propertyname}, \text{val}) \in S\ ((\text{propertyname}, \text{propertytype}, \text{mult}) = \text{property}(MM, \text{metaclassname}, \text{propertyname}) \wedge (sc_1 \wedge sc_2 \wedge sc_3 \wedge sc_4)))$$

Interpretação: Para todo *slot* ou par (propertyname, val) de todo descritor de instância ou tripla (instancename, metaclassname, S), a propriedade efetiva (propertyname, propertytype, mult) correspondente ao *slot* deve atender às subcondições sc_1, sc_2, sc_3 ou sc_4, sendo sc_1 definida por:

$$(\text{propertytype} \in E) \wedge (\text{descriptor}_e(\text{propertytype}) = (\text{enumname}, L)) \wedge (\text{mult} = (\text{lower}, \text{upper})) \wedge (\text{upper} \neq *) \Rightarrow (\text{val} \in \Sigma^*) \wedge (\text{decode}_{MM}(\text{val}) \in L^*) \wedge (|\text{decode}_{MM}(\text{val})| \geq \text{lower}) \wedge (|\text{decode}_{MM}(\text{val})| \leq \text{upper})$$

Interpretação: Se o tipo propertytype for uma enumeração descrita por (enumname, L) com multiplicidade (lower, upper), então o valor decodificado do *slot* deve pertencer a uma lista finita, eventualmente vazia, de elementos de L, que são os

valores permitidos para a enumeração propertytype. O número de elementos deve estar entre lower e upper. A função $decode_{MM}$ foi definida pela eq. (18).

A subcondição sc_2, por sua vez, é estabelecida pela seguinte expressão:

$$(propertytype \in E) \wedge (descriptor_e(propertytype) = (enumname, L)) \wedge$$
$$(mult = (lower, *)) \Rightarrow (val \in \Sigma^*) \wedge (decode_{MM}(val) \in L^*) \wedge$$
$$(|decode_{MM}(val)| \geq lower)$$

Interpretação: Caso similar ao da subcondição sc_1, mas se aplica quando a multiplicidade da propriedade do *slot* possui limite superior infinito. Nesse caso, não é verificado o limite superior do número de elementos do valor.

Para as subcondições seguintes, convém definir a função auxiliar types, que toma um metamodelo, um modelo e uma instância do modelo para retornar o conjunto de metaclasses que são tipos dessa instância. Lembrando que os tipos de uma instância são seu tipo base mais suas supermetaclasses. Essa função pode ser definida pela expressão da eq. (22):

$$\begin{gathered} types(MM, M, i) = \{c \mid (c \in C) \wedge \\ (descriptor_i(i) = (instancename, metaclassname, S)) \wedge \\ ((c = metaclass(MM, metaclassname)) \vee \\ (((metaclass(MM, metaclassname), c) \in \Gamma^+))\} \end{gathered} \qquad (22)$$

Seja então a subcondição sc_3:

$$(propertytype \in C) \wedge (mult = (lower, upper)) \wedge (y \neq *) \Rightarrow$$
$$(val \subseteq \{ i \mid (i \in I) \wedge (propertytype \in types(MM, M, i))\}) \wedge$$
$$(|val| \geq lower) \wedge (|val| \leq upper)$$

Interpretação: Se o tipo propertytype for uma metaclasse com multiplicidade (lower, upper), então o valor do *slot* deve ser um conjunto finito, eventualmente vazio,

contido no subconjunto das instâncias de I em que propertytype é um de seus tipos, que são os valores permitidos para a metaclasse propertytype. O número de elementos deve estar entre lower e upper.

Por fim, a subcondição sc_4 é dada por:

$$(propertytype \in C) \wedge (mult = (lower, *)) \Rightarrow$$
$$(val \subseteq \{ i \mid (i \in I) \wedge (propertytype \in types(MM, M, i))\}) \wedge (|val| \geq lower)$$

Interpretação: Caso similar ao da subcondição sc_3, mas se aplica para quando a multiplicidade da propriedade do *slot* possui limite superior infinito. Nesse caso, não é verificado o limite superior do número de instâncias no conjunto val.

O conjunto de todos os modelos possíveis conformes a um metamodelo MM é denotado por $\mathcal{M}_{MM}$, enquanto o conjunto de todas as possíveis instâncias da metaclasse $c \in C$ de MM é denotado por $\mathcal{I}_{MM}(c)$. Algumas vezes é conveniente usar $\mathcal{I}_{MM}(c)$ para obter o conjunto de todas as possíveis instâncias de um conjunto de metaclasses. Assim, usar um conjunto de metaclasses como argumento de $\mathcal{I}_{MM}(c)$, sem alterar a notação, também é aceito.

4.10 MESCLAGEM E IMPORTAÇÃO DE METAMODELOS

4.10.1 **Mesclagem e importação no MOF e na UML**

O MOF e a UML apresentam um elemento denominado pacote, ou *package*, que permite agrupar outros elementos do metamodelo ou modelo, respectivamente, inclusive permitindo a existência de pacotes dentro de pacotes, hierarquizando o agrupamento desses elementos. Existem dois tipos de relacionamentos específicos entre pacotes que possibilitam a extensão e reuso de metamodelos e modelos, que são a importação e mesclagem, ou *import* e *merge*, respectivamente.

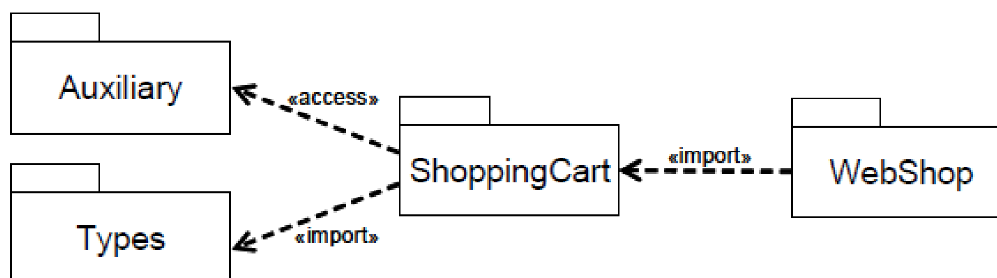
A especificação do MOF (OMG, 2015a) faz referência à especificação da UML (OMG, 2015c) para explicar a semântica da importação e mesclagem de pacotes. A diferença fundamental é que o MOF as interpreta no nível dos metamodelos, no qual os elementos são metaclasses e enumerações, enquanto a UML as interpreta no

nível dos modelos, em que os elementos são classes, associações, casos de uso, atores, etc.

A importação de pacotes no MOF e UML consiste em uma relação binária que permite que elementos de um pacote façam referências a membros de outro pacote pelo seu nome. É representada por uma linha tracejada com uma flecha aberta partindo do pacote que está importando em direção ao pacote importado. Um estereótipo é utilizado para identificar qual tipo de importação é pretendida: *«import»* para pública ou *«access»* para privada. A importação pública significa que se um pacote B importar o pacote A, então os elementos de A também estarão visíveis para um pacote C que eventualmente importar o pacote B. Na importação privada, essa visibilidade não ocorre (OMG, 2015c).

No exemplo da Figura 15, os elementos no pacote *Types* são importados por *ShoppingCart* e, posteriormente, importados por *WebShop*. Por outro lado, os elementos de *Auxiliary* são apenas acessados por *ShoppingCart* e não podem ser referenciados por *WebShop*.

Figura 15 – Exemplo de importação de pacotes na UML



Fonte: OMG (2015c)

A mesclagem de pacotes define como o conteúdo de um pacote é estendido pelo conteúdo de outro pacote. Trata-se de uma relação binária que indica que os conteúdos dos dois pacotes devem ser combinados. É similar à generalização de classes na medida em que a subclasse adiciona características a uma outra classe, resultando em um elemento que combina as características de ambas (OMG, 2015c).

A mesclagem pode ser utilizada para estender um dado conceito em incrementos. Cada incremento deste conceito é definido em um pacote separado, com seu próprio propósito ou enfoque. Selecionando quais pacotes mesclar, é possível obter uma definição adaptada do conceito em questão para um fim específico. A mesclagem de pacotes é extensivamente utilizada na definição do metamodelo da UML (OMG, 2015c).

Conceitualmente, a mesclagem de pacotes pode ser vista como uma operação que obtém os conteúdos de dois pacotes, denominados de pacote a ser mesclado e pacote receptor, e produz um terceiro que combina esses conteúdos. Semanticamente, não há diferença entre um modelo ou metamodelo que exibe explicitamente a operação de mesclagem de pacotes e um modelo ou metamodelo em que a mesclagem já foi realizada.

A mesclagem entre dois pacotes implica em um conjunto de transformações, em que o conteúdo do pacote a ser mesclado é combinado com o conteúdo do pacote receptor. Nos casos em que certos elementos nos dois pacotes representam a mesma entidade ou conceito, seu conteúdo é mesclado em um único elemento resultante de acordo com regras pré-determinadas para o tipo de elemento em questão. Uma relação de mesclagem denotada em um diagrama MOF ou UML implica nessas transformações, mas seus resultados finais não são explicitados na representação gráfica do metamodelo ou modelo, respectivamente. Entretanto, o desenho do pacote receptor no diagrama tem a intenção de representar o resultado da mesclagem, e não apenas o incremento que ela adiciona. Assim, qualquer referência a um elemento dentro pacote receptor implica em uma referência aos resultados da mesclagem aplicada a este elemento, e não apenas ao incremento que está “fisicamente” contido neste pacote.

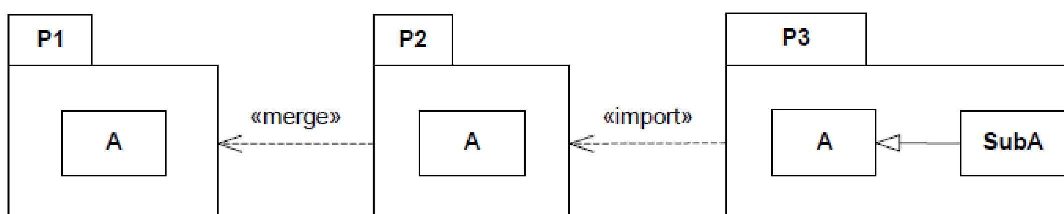
Essa situação é exemplificada na Figura 16, na qual o pacote *P1* (a ser mesclado) e o pacote *P2* (receptor) definem diferentes incrementos para a mesma classe *A* (identificadas por *P1::A* e *P2::A* respectivamente). O pacote *P2* mescla o conteúdo do pacote *P1*, o que implica na mesclagem do incremento *P1::A* no incremento *P2::A*. O pacote *P3* importa o conteúdo de *P2* de modo que pode definir uma subclasse de *A* chamada *SubA*. Nesse caso, o elemento *A* no pacote *P3* (*P3::A*) representa o resultado da mesclagem de *P1::A* em *P2::A*, e não apenas o incremento denotado em *P2::A*. Se um outro pacote importasse *P1*, então a

referência a *A* neste outro pacote seria exclusivamente ao incremento *P1::A* e não à classe *A* resultante da mesclagem.

Como pode ser observado, a relação de mesclagem é denotada no MOF e UML por uma linha tracejada com flecha aberta e o estereótipo *«merge»*. A flecha parte do pacote receptor em direção ao pacote a ser mesclado. Apesar da nomenclatura, que pode sugerir o sentido oposto, o termo receptor significa que o pacote recebe os incrementos provenientes do pacote a ser mesclado.

A especificação da UML (OMG, 2015c) descreve, em linguagem natural, regras para mesclar diversos tipos de elementos, tais como classes, propriedades, associações, enumerações, etc. Com isso, supõe-se como uma ferramenta deve processar esta relação de mesclagem e obter os elementos resultantes. O MOF utiliza a semântica de mesclagem de classes UML para suas metaclasses, interpretadas no seu nível descritivo.

Figura 16 – Exemplo de mesclagem de pacotes na UML



Fonte: OMG (2015c)

4.10.2 Mesclagem e importação de metamodelos no SBMM

Baseando-se nas semânticas de importação e mesclagem do MOF, definem-se no SBMM operações semelhantes para extensão e reuso de metamodelos. Enquanto a especificação da UML (OMG, 2015c), referenciada pela especificação do MOF (OMG, 2015a), apresenta esse conceito em linguagem natural com alguns exemplos, esta subseção visa a descrever precisamente através de equações qual deve ser o resultado da mesclagem de dois metamodelos baseados em SBMM. Definições desse tipo não foram encontradas nos outros formalismos apresentados no Capítulo 3 deste trabalho.

Antes de iniciar, faz-se um paralelo com o conceito de módulo apresentado por Szyperski (1992). No contexto de programas de computador (camada M1), um módulo é uma "cápsula" contendo definições de itens, sendo que o termo item é usado para denotar elementos individuais que podem ser declarados e operados em uma dada linguagem de programação (variáveis, tipos, procedimentos, métodos, classes, etc.). Um módulo estabelece uma fronteira entre itens definidos dentro dele e itens definidos em outros módulos.

No contexto da camada M2, o conceito que mais se aproxima da definição de módulo é o conceito de pacote do MOF, haja vista que ele estabelece uma fronteira entre os itens que contém e os externos. A mesclagem e a importação são definidas primariamente entre pacotes. No SBMM, não foi previsto explicitamente o conceito de módulo ou pacote. Porém, como os itens possíveis da camada M2 (metaclasses, enumerações, generalizações, propriedades, etc.) são definidos obrigatoriamente dentro de um metamodelo, pode-se enxergar um metamodelo também como um módulo. Desse modo, a mesclagem e importação na camada M2 são definidas entre metamodelos.

Inicia-se aqui com a operação de mesclagem. Seja $@_{MM}$ uma função que toma um nome e dois metamodelos como entrada, resultando no metamodelo mesclado. MM_1 e MM_2 são os metamodelos de entrada tais que:

$$MM_1 = (name_{MM1}, C_1, \Gamma_1, E_1, R_1, descriptor_{c1}, descriptor_{e1})$$

$$MM_2 = (name_{MM2}, C_2, \Gamma_2, E_2, R_2, descriptor_{c2}, descriptor_{e2})$$

Define-se $@_{MM}$ conforme a eq. (23).

$$\begin{gathered} @_{MM}(newname, MM_1, MM_2) = \\ (newname, C_1 \ @_C \ C_2, \Gamma_1 \ @_\Gamma \ \Gamma_2, E_1 \ @_E \ E_2, R_1 @_R \ R_2, \\ descriptor_{c1} \ @_{dc} \ descriptor_{c2}, descriptor_{e1} \ @_{de} \ descriptor_{e2}) \end{gathered} \tag{23}$$

Em que:

- newname $\in \Sigma^+$ é a cadeia de símbolos que nomeia o metamodelo resultante;
- $@_C$ é o operador de mesclagem para conjuntos de metaclasses;
- $@_\Gamma$ é o operador de mesclagem para conjuntos de generalizações;
- $@_E$ é o operador de mesclagem para conjuntos de enumerações;
- $@_R$ é o operador de mesclagem para conjuntos de restrições;
- $@_{dc}$ é o operador de mesclagem para funções descritoras de metaclasses;
- $@_{de}$ é o operador de mesclagem para funções descritoras de enumerações.

Os operadores acima são auxiliares para a definição da função $@_{MM}$, e operam dentro de seu contexto. Isso significa que suas especificações têm acesso aos elementos de MM_1 e MM_2.

O operador de mesclagem para conjuntos de metaclasses $@_C$ é definido de acordo com a eq. (24).

$$\begin{gathered} C_1 \ @_C \ C_2 = C_1 \cup C_2 \setminus \\ \{c \mid (c \in C_2) \wedge (descriptor_{c2}(c) = (metaclassname, P)) \wedge \\ (\exists \ (u, Q) \in Im(descriptor_{c1}) \mid u = metaclassname)\} \end{gathered} \tag{24}$$

Interpretação da eq. (24): A fórmula consiste na união dos conjuntos de metaclasses de cada metamodelo $C_1 \cup C_2$ subtraídos das metaclasses pertencentes a C_2 que possuem o mesmo nome de alguma metaclasse de C_1. Em outras palavras, junta-se os elementos de C_1 e C_2 e retira-se aqueles de C_2 com algum correspondente em C_1. Ou seja, quando há metaclasses de mesmo nome em C_1 e C_2, apenas o objeto proveniente de C_1 é mantido no metamodelo resultante como representante da metaclasse mesclada. A informação carregada pelos descritores de elementos de C_2 descartados é preservada pela mesclagem das funções $descriptor_{c1}$ e $descriptor_{c2}$, vista a seguir.

Define-se então o operador de mesclagem de funções descritoras de metaclasses $@_{dc}$ por meio de (25).

$$
\begin{array}{c}
descriptor_{c1} \ @_{dc} \ descriptor_{c2} = \\
\{(c, (w, P)) \mid ((c, (w, P)) \in descriptor_{c1}) \wedge \\
\neg(\exists \ (u, Q) \ ((u, Q) \in Im(descriptor_{c2}) \wedge (w = u)))\} \\
\cup \\
\{(c, (w, update_target_types(P))) \mid ((c, (w, P)) \in descriptor_{c2}) \wedge \\
\neg(\exists \ (u, Q) \ ((u, Q) \in Im(descriptor_{c1}) \wedge (w = u)))\} \\
\cup \\
\{(c, (w, update_target_types(P \ @_P \ Q))) \mid \\
((c, (w, P)) \in descriptor_{c1}) \wedge ((w, Q) \in Im(descriptor_{c2}))\}
\end{array}
\qquad (25)
$$

Interpretação da eq. (25): A construção dessa equação é baseada na união de três conjuntos. A equação foi disposta com quebras de linha de modo a facilitar a visualização de cada um deles. Os dois primeiros conjuntos se tratam das funções $descriptor_{c1}$ e $descriptor_{c2}$ descartando os descritores de metaclasses que têm alguma correspondente de mesmo nome no outro metamodelo. O descarte é compensado pela união com o terceiro conjunto da expressão, que produz um novo descritor para cada metaclasse de C_1 que possua uma correspondente de mesmo nome em C_2. O conjunto de propriedades desse novo descritor é produzido pelo operador $@_P$, definido adiante. Um detalhe importante é que os descritores de metaclasses provenientes dos dois últimos conjuntos passam por uma transformação realizada pela função auxiliar update_target_types. Essa função tem por objetivo atualizar os tipos alvos que passam por mesclagem. Por exemplo: se y é metaclasse que é tipo alvo de alguma propriedade de qualquer metaclasse no metamodelo MM_2, então esse tipo alvo deve ser trocado para x caso a metaclasse y seja mesclada com a metaclasse x de mesmo nome proveniente de MM_1. A mesma regra vale para enumerações. Mais adiante é apresentada uma fórmula para essa função na eq. (34).

O operador $@_P$ de mesclagem para os conjuntos de propriedades é definido pela eq. (26):

$$
\begin{aligned}
P_x @_P P_y = \{p_x @_p p_y \mid \\
(p_x \in P_x) \wedge (p_x = (propertyname_x, propertytype_x, mult_x)) \wedge \\
(p_y \in P_y) \wedge (p_y = (propertyname_y, propertytype_y, mult_y)) \wedge \\
(propertyname_x = propertyname_y)\} \\
\cup \\
\{p_x \mid (p_x \in P_x) \wedge (p_x = (propertyname_x, propertytype_x, mult_x)) \wedge \\
\neg\exists p_y ((p_y \in P_y) \wedge (p_y = (propertyname_y, propertytype_y, mult_y)) \wedge \\
(propertyname_x = propertyname_y))\} \\
\cup \\
\{p_y \mid (p_y \in P_y) \wedge (p_y = (propertyname_y, propertytype_y, mult_y)) \wedge \\
\neg\exists p_x ((p_x \in P_x) \wedge (p_x = (propertyname_x, propertytype_x, mult_x)) \wedge \\
(propertyname_y = propertyname_x))\}
\end{aligned}
\quad (26)
$$

Interpretação da eq. (26): A construção do operador $@_P$ baseia-se na união de três conjuntos. O primeiro indica que propriedades de mesmo nome devem ser mescladas de alguma maneira como resultado do operador $@_p$, definido abaixo. O segundo indica que o conjunto final de propriedades mesclado deve conter todas as propriedades da metaclasse x, sem alterações, que não tem correspondência em y. O terceiro indica o mesmo com relação às propriedades de y sem correspondência em x.

Por fim, para concluir a definição completa do operador de mesclagem de conjuntos de metaclasses $@_C$, resta definir o operador de mesclagem de propriedades $@_p$, usado na definição do operador de mesclagem de conjuntos de propriedades $@_P$. Sejam:

$$p_x = (propertyname_x, propertytype_x, mult_x)$$

$$p_y = (propertyname_y, propertytype_y, mult_y)$$

O termo $mult_x = (lower_x, upper_x)$ é a multiplicidade de p_x determinada por um limite inferior $lower_x$ e um limite superior $upper_x$, que pode ser infinito.

O termo $mult_y = (lower_y, upper_y)$ é a multiplicidade de p_y determinada por um limite inferior $lower_y$ e um limite superior $upper_y$, que pode ser infinito.

Como o operador $@_p$ foi concebido para operar sobre propriedades de mesmo nome, então se pode supor que $propertyname_x = propertyname_y$. O operador $@_p$ é definido pela eq. (27).

$$p_x \; @_p \; p_y = (propertyname_x,\ maxtype(propertytype_x,\ propertytype_y),\ (min(lower_x,\ lower_y),\ max(upper_x,\ upper_y))) \tag{27}$$

A função maxtype tem por objetivo retornar o tipo mais geral dentre seus operandos. Quando são enumerações com nomes distintos, isto é, se $propertytype_x \in E_1 \wedge propertytype_y \in E_2 \wedge enumname_x \neq enumname_y$, então o tipo mais geral é aquela enumeração cujo conjunto de valores permitidos contém o conjunto correspondente da outra. Isso permite que o tipo alvo da propriedade resultante da operação $p_x \; @_p \; p_y$ seja capaz de suportar valores de ambas as enumerações $propertytype_x$ e $propertytype_y$. Quando isso não ocorre, maxtype não é definida, ou seja, as propriedades p_x e p_y possuem tipos incompatíveis e não podem ser mescladas. As expressões das eqs. (28) e (29) definem maxtype para enumerações de nomes diferentes.

$$\begin{gathered} maxtype(propertytype_x,\ propertytype_y) = propertytype_x \\ \text{se } (propertytype_x \in E_1) \wedge (descriptor_{e1}(propertytype_x) = (enumname_x,\ L_x)) \wedge \\ (propertytype_y \in E_2) \wedge (descriptor_{e2}(propertytype_y) = (enumname_y,\ L_y)) \wedge \\ (enumname_x \neq enumname_y) \wedge (L_y \subseteq L_x) \end{gathered} \tag{28}$$

$$\begin{gathered} maxtype(propertytype_x,\ propertytype_y) = propertytype_y \\ \text{se } (propertytype_x \in E_1) \wedge (descriptor_{e1}(propertytype_x) = (enumname_x,\ L_x)) \wedge \\ (propertytype_y \in E_2) \wedge (descriptor_{e2}(propertytype_y) = (enumname_y, L_y)) \wedge \\ (enumname_x \neq enumname_y) \wedge (L_x \subset L_y) \end{gathered} \tag{29}$$

Por exemplo, seja a enumeração e_1 descrita por ("Integer", $\mathbb{Z}$) e a enumeração e_2 descrita por ("Real", $\mathbb{R}$). O resultado de $maxtype(e_1,\ e_2)$ é igual a e_2, pois $\mathbb{Z} \subset \mathbb{R}$, encaixando-se no segundo caso acima. De fato, o tipo correspondente aos números reais é mais abrangente, podendo suportar também os inteiros, sendo escolhido

como tipo alvo para a mesclagem de uma propriedade inteira com uma propriedade real.

Caso as enumerações $propertytype_x$ e $propertytype_y$ tenham o mesmo nome, então elas são interpretadas como modelagens do mesmo conceito e, eventualmente, uma delas possui valores permitidos que a outra não permite, ou vice-versa. Nesse caso, deve-se mesclá-las para expandir sua capacidade na enumeração de mesmo nome do metamodelo resultante, suportando todos os valores. Isso é feito pelo operador de união, conforme será visto mais adiante. Entretanto, por enquanto, é necessário saber que quando há enumerações de mesmo nome nos dois metamodelos, o objeto da enumeração do primeiro metamodelo é escolhido como representante da enumeração mesclada, assim como foi feito para metaclasses. Dessa forma, define-se a função maxtype conforme a eq. (30).

$$\begin{gathered} maxtype(propertytype_x, propertytype_y) = propertytype_x \\ \text{se } (propertytype_x \in E_1) \wedge (descriptor_{e1}(propertytype_x) = (enumname_x, L_x)) \wedge \\ (propertytype_y \in E_2) \wedge (descriptor_{e2}(propertytype_y) = (enumname_y, L_y)) \wedge \\ (enumname_x = enumname_y) \end{gathered} \quad (30)$$

Por outro lado, se os tipos alvos de p_x e p_y são metaclasses, isto é, se $propertytype_x \in C_1$ e $propertytype_y \in C_2$, então o tipo mais geral é a metaclasse $propertytype_x$ proveniente de C_1 conforme (31), desde que tenham o mesmo nome, pois a eq. (24) estabeleceu que, quando há metaclasses de mesmo nome em C_1 e C_2, aquela de C_1 permanece como representante no metamodelo mesclado resultante. Lembrando que o seu descritor mapeado pela função $descriptor_{c1}$ $@_{dc}$ $descriptor_{c2}$ foi atualizado para suportar as propriedades de ambas conforme a eq. (25).

$$\begin{gathered} maxtype(propertytype_x, propertytype_y) = propertytype_x \\ \text{se } (propertytype_x \in C_1) \wedge \\ (descriptor_{c1}(propertytype_x) = (metaclassname_x, P_x)) \wedge \\ (propertytype_y \in C_2) \wedge \\ (descriptor_{c2}(propertytype_y) = (metaclassname_y, P_y)) \wedge \\ (metaclassname_x = metaclassname_y) \end{gathered} \quad (31)$$

Se os tipos alvos de p_x e p_y, ou seja, $propertytype_x$ e $propertytype_y$, forem metaclasses de nomes diferentes, o tipo mais geral será aquele que for supermetaclasse do outro no metamodelo resultante. Ou seja, nesse caso maxtype é definido pelas eqs. (32) e (33).

$$\begin{array}{c} maxtype(propertytype_x, propertytype_y) = propertytype_x \\ se\ (propertytype_x \in C_1) \wedge \\ (descriptor_{c1}(propertytype_x) = (metaclassname_x, P_x)) \wedge \\ (propertytype_y \in C_2) \wedge \\ (descriptor_{c2}(propertytype_y) = (metaclassname_y, P_y)) \wedge \\ (metaclassname_x \neq metaclassname_y) \wedge \\ ((update_type(propertytype_y), update_type(propertytype_x)) \in (\Gamma_1\ @_\Gamma\ \Gamma_2)^+) \end{array} \quad (32)$$

$$\begin{array}{c} maxtype(propertytype_x, propertytype_y) = propertytype_y \\ se\ (propertytype_x \in C_1) \wedge \\ (descriptor_{c1}(propertytype_x) = (metaclassname_x, P_x)) \wedge \\ (propertytype_y \in C_2) \wedge \\ (descriptor_{c2}(propertytype_y) = (metaclassname_y, P_y)) \wedge \\ (metaclassname_x \neq metaclassname_y) \wedge \\ ((update_type(propertytype_x), update_type(propertytype_y)) \in (\Gamma_1\ @_\Gamma\ \Gamma_2)^+) \end{array} \quad (33)$$

Para facilitar o entendimento das eqs. (32) e (33), a ideia básica é que se as metaclasses $propertytype_x$ e $propertytype_y$ tiverem relação de generalização no metamodelo resultante, direta ou transitivamente, então deve ser preservada como tipo alvo resultante da propriedade aquela que for supermetaclasse da outra. A função update_type é definida mais adiante nas eqs. (36) a (38) e serve para obter o tipo (metaclasse ou enumeração) correspondente no metamodelo resultante da mesclagem. Já $\Gamma_1\ @_\Gamma\ \Gamma_2$ denota o conjunto de generalizações do modelo resultante, cujo operador $@_\Gamma$ foi citado anteriormente e será apresentado na eq. (39).

Quando $propertytype_x$ e $propertytype_y$ forem metaclasses de nomes diferentes e não possuírem relação de generalização no metamodelo resultante, então são tipos alvo incompatíveis e as propriedades não podem ser mescladas.

Quando $propertytype_x$ for uma enumeração e $propertytype_y$ uma metaclasse, ou vice-versa, maxtype não é definida, considerando que enumerações e metaclasses são tipos incompatíveis.

A função update_target_types, utilizada anteriormente, toma um conjunto de propriedades $P = \{p_1, p_2, ..., p_{|P|}\}$ como argumento e é definida pela expressão da eq. (34).

$$\text{update_target_types}(P) = \{\text{update_target_type}(p_1), \text{update_target_type}(p_2), \ldots, \text{update_target_type}(p_{|P|})\} \quad (34)$$

A função update_target_type, por sua vez, toma uma propriedade p = (propertyname, propertytype, mult) como argumento e a transforma em uma propriedade com o tipo alvo corrigido, se necessário. É dada pela eq. (35).

$$\text{update_target_type}(p) = (\text{propertyname}, \text{update_type}(\text{propertytype}), \text{mult}) \quad (35)$$

Por fim, update_type(t) pode ser definida pelas eqs. (36) a (38).

$$\begin{gathered}\text{update_type}(t) = x \\ \text{se } (t \in C_2) \wedge (\text{descriptor}_{c2}(t) = (\text{metaclassname}_t, P_t)) \wedge \\ (\exists x\ (x \in C_1) \wedge (\text{descriptor}_{c1}(x) = (\text{metaclassname}_x, P_x)) \wedge \\ (\text{metaclassname}_t = \text{metaclassname}_x))\end{gathered} \quad (36)$$

$$\begin{gathered}\text{update_type}(t) = y \\ \text{se } (t \in E_2) \wedge (\text{descriptor}_{e2}(t) = (\text{enumname}_t, L_t)) \wedge \\ (\exists y\ (y \in E_1) \wedge (\text{descriptor}_{e1}(y) = (\text{enumname}_y, L_y)) \wedge \\ (\text{enumname}_t = \text{enumname}_y))\end{gathered} \quad (37)$$

$$\text{update_type}(t) = t \text{ caso contrário} \quad (38)$$

Ou seja, update_type(t) toma o tipo t, que pode ser uma metaclasse ou enumeração, e retorna seu tipo correspondente no metamodelo mesclado caso t seja do segundo metamodelo e tenha um correspondente de mesmo nome no primeiro metamodelo. Caso contrário, não é necessária nenhuma mudança no tipo t.

O operador $@_\Gamma$ de mesclagem de conjuntos de generalizações é definido sobre Γ_1 e Γ_2 pela união da eq. (39).

$$\Gamma_1 \; @_\Gamma \; \Gamma_2 = \Gamma_1 \cup \Theta_2 \qquad (39)$$

Em que:

$$\Theta_2 = \{(\text{update_type}(x), \text{update_type}(y)) \mid (x, y) \in \Gamma_2 \}$$

Interpretação de Θ_2: Consiste nos pares de generalizações de Γ_2 atualizados para quando alguma metaclasse $x \in C_2$ ou $y \in C_2$ possua correspondente de mesmo nome em C_1.

O operador $@_E$ de mesclagem de conjuntos de enumerações é definido pela expressão sobre E_1 e E_2 da eq. (40).

$$\begin{gathered} E_1 \; @_E \; E_2 = E_1 \cup E_2 \setminus \\ \{e \mid (e \in E_2) \wedge (\text{descriptor}_{e2}(e) = (\text{enumname}, L)) \wedge \\ (\exists \, (u, V) \in \text{Im}(\text{descriptor}_{e1}) \mid u = \text{enumname}) \} \end{gathered} \qquad (40)$$

Interpretação da eq. (40): A fórmula consiste na união dos conjuntos de enumerações de cada metamodelo $E_1 \cup E_2$ subtraídos das enumerações pertencentes a E_2 que possuem o mesmo nome de alguma enumeração pertencente a E_1. Em outras palavras, junta-se os elementos de E_1 e E_2 e retira-se aqueles de E_2 com algum correspondente em E_1. Ou seja, quando há enumerações de mesmo nome em E_1 e E_2, apenas o objeto proveniente de E_1 é mantido no metamodelo

mesclado como representante da enumeração. A informação carregada pelos descritores de elementos de E_2 descartados é preservada pela mesclagem das funções $descriptor_{e1}$ e $descriptor_{e2}$, vista a seguir.

Define-se então o operador de mesclagem de funções descritoras de enumeração $@_{de}$ através de (41).

$$\begin{gathered} descriptor_{e1} \; @_{de} \; descriptor_{e2} = \\ \{(e, (w, L)) \mid ((e, (w, L)) \in descriptor_{e1}) \wedge \\ \neg\exists\, (u, V)\, (((u, V) \in Im(descriptor_{e2})) \wedge (w = u))\} \\ \cup \\ \{(e, (w, L)) \mid ((e, (w, L)) \in descriptor_{e2}) \wedge \\ \neg\exists\, (u, V)\, (((u, V) \in Im(descriptor_{e1})) \wedge (w = u))\} \\ \cup \\ \{(e, (w, L \cup V)) \mid ((e, (w, L)) \in descriptor_{e1}) \wedge \\ (w, V) \in Im(descriptor_{e2})\} \end{gathered} \tag{41}$$

Interpretação da eq. (41): A construção dessa equação é baseada na união de três conjuntos. Os dois primeiros se tratam das funções $descriptor_{e1}$ e $descriptor_{e2}$ descartando os descritores de enumerações que tem alguma correspondente de mesmo nome no outro metamodelo. O descarte é compensado pela união com o terceiro conjunto da expressão, que produz um novo descritor para cada enumeração de E_1 que possua uma correspondente de mesmo nome em E_2. Esse novo descritor é produzido pelo operador de união. Ou seja, duas enumerações mescladas resultam em uma enumeração cujos valores permitidos são a união dos conjuntos de valores permitidos para cada operando.

O operador $@_R$ de mesclagem de conjuntos de restrições é definido pela união entre R_1 e R_2 conforme a eq. (42).

$$R_1 \; @_R \; R_2 = R_1 \cup R_2 \tag{42}$$

Interpretação da eq. (42): Como R_1 impõe restrições para os modelos conformes a MM_1 e R_2 impõe restrições para os modelos conformes a MM_2, os modelos

conformes a $@_{MM}(MM_1, MM_2)$ devem obedecer a todas as restrições provenientes de MM_1 e MM_2.

Como os predicados e funções são denotados pelos nomes das metaclasses, enumerações e propriedades, e eles permanecem os mesmos quando qualquer elemento do metamodelo é mesclado, então não é necessário fazer nenhuma alteração nas sentenças.

Pode ser que existam restrições incompatíveis em $R_1 = \{r_{11}, r_{12}, \ldots r_{1|R1|}\}$ e $R_2 = \{r_{21}, r_{22}, \ldots r_{2|R2|}\}$, que resultem em uma falácia para a expressão $r_{11} \wedge r_{12} \wedge \ldots \wedge r_{1|R1|} \wedge r_{21} \wedge r_{22} \wedge \ldots \wedge r_{2|R2|}$. Nesse caso, provavelmente há alguma incoerência conceitual entre os dois metamodelos que se deseja mesclar, cabendo ao desenvolvedor dos metamodelos analisar.

O conceito de importação é apresentado por Szyperski (1992) como uma relação entre módulos. Sejam m e m’ dois módulos de um programa. A relação *imp* entre m e m’, ou seja, m *imp* m’, denota que m importa m’. Isso significa que os itens de m’ passam a ser acessíveis por m.

A importação de metaclasses ou enumerações pode ser obtida no SBMM com auxílio da mesclagem. A definição de propriedade de (3) não permite tipos que não estejam no mesmo metamodelo. Sendo assim, caso seja desejada a utilização de um tipo *T* (metaclasse ou enumeração) no metamodelo MM_1, mas *T* está definido no metamodelo MM_2, pode-se criar o tipo *T* com o mesmo nome em MM_1, porém vazio (metaclasse sem propriedades ou enumeração com conjunto de valores vazio). Isso é suficiente para permitir referências a *T* em propriedades de MM_1 (importação), mesmo que os detalhes sobre *T* estejam especificados em MM_2. Uma ferramenta que permita criar ou editar instâncias (modelos) de MM_1 com referências a tipos de outro metamodelo (no caso, MM_2) pode internamente efetuar a mesclagem entre MM_1 e MM_2 para obter um metamodelo consolidado com os tipos completos e assim verificar relações de conformidade.

4.10.3 **Exemplo de mesclagem de metamodelos no SBMM**

Para ilustrar a mesclagem de metamodelos no SBMM, parte-se de um simples metamodelo de exemplo que representa uma versão simplificada do diagrama de casos de uso da UML, denotado por $MM_{USECASES}$.

$$MM_{USECASES} = (\text{“UseCasesMetamodel”}, C_{USECASES}, \Gamma_{USECASES}, E_{USECASES}, R_{USECASES}, descriptor_{c_USECASES}, descriptor_{e_USECASES})$$

Em que:

- $C_{USECASES} = \{c_1, c_2, c_3\}$
    - $c_1 =^{c_USECASES}$ (“UseCase”, P_1)
        - $P_1 = \{p_{11}, p_{12}, p_{13}, p_{14}, p_{15}\}$
            - p_{11} = (“Name”, e_1, 1..1)
            - p_{12} = (“PreConditions”, e_1, 1..1)
            - p_{13} = (“Description”, e_1, 1..1)
            - p_{14} = (“PostConditions”, e_1, 1..1)
            - p_{15} = (“Exceptions”, e_1, 1..1)
    - $c_2 =^{c_USECASES}$ (“Actor”, P_2)
        - $P_2 = \{p_{21}\}$
            - p_{21} = (“Name”, e_1, 1..1)
    - $c_3 =^{c_USECASES}$ (“AssociationUseCaseActor”, P_3)
        - $P_3 = \{p_{31}, p_{32}, p_{33}\}$
            - p_{31} = (“Name”, e_1, 1..1)
            - p_{32} = (“UseCase”, c_1, 1..1)
            - p_{33} = (“Actor”, c_2, 1..1)
- $\Gamma_{USECASES} = \emptyset$
- $E_{USECASES} = \{e_1\}$

- $e_1 =^{e_USECASES}$ (“String”, Σ^*)

- $R_{USECASES} = \{r_1, r_2, r_3, r_4\}$
  - r_1: $\forall u\ UseCase(u) \Rightarrow \neg(UseCase.Name(u) = \varepsilon)$
    - Interpretação: Nenhum caso de uso pode ter o nome vazio.
  - r_2: $\forall u_1, u_2\ UseCase(u_1) \wedge UseCase(u_2) \wedge (UseCase.Name(u_1) = UseCase.Name(u_2)) \Rightarrow u_1 = u_2$
    - Interpretação: Não pode haver mais de um caso de uso com o mesmo nome.
  - r_3: $\forall a\ Actor(a) \Rightarrow \neg(Actor.Name(a) = \varepsilon)$
    - Interpretação: Nenhum ator pode ter o nome vazio.
  - r_4: $\forall a_1, a_2\ Actor(a_1) \wedge Actor(a_2) \wedge (Actor.Name(a_1) = Actor.Name(a_2)) \Rightarrow a_1 = a_2$
    - Interpretação: Não pode haver mais de um ator com o mesmo nome.

Nesse metamodelo, as pré-condições, descrição, pós-condições e exceções dos casos de uso são representadas como *strings*. Para o propósito deste exemplo é suficiente.

Considere-se que se deseja estender $MM_{USECASES}$ para admitir associações entre casos de uso, que permitem também estereótipos como *«use»* ou *«import»*. Também é desejado acrescentar um atributo de estereótipo nos atores. Para isso, cria-se um segundo metamodelo $MM_{USECASES}$’ que, posteriormente, será mesclado com o primeiro para estendê-lo.

$$MM_{USECASES}' = (\text{“UseCasesMetamodelExt”}, C_{USECASES}', \Gamma_{USECASES}', E_{USECASES}', R_{USECASES}', descriptor_{c_USECASES'}, descriptor_{e_USECASES'})$$

Em que:

- $C_{USECASES}$' = {c_1', c_2', c_3'}
  - $c_1' =^{c_USECASES'}$ (“UseCase”, ∅)
  - $c_2' =^{c_USECASES'}$ (“Actor”, P_2')
    - P_2' = {p_{21}'}
      - p_{21}' = (“Stereotype”, e_1', 1..1)
  - $c_3' =^{c_USECASES'}$ (“AssociationUseCases”, P_3')
    - P_3' = {p_{31}', p_{32}', p_{33}'}
      - p_{31}' = (“UseCase1”, c_1', 1..1)
      - p_{32}' = (“UseCase2”, c_1', 1..1)
      - p_{33}' = (“Stereotype”, e_1', 1..1)
- $\Gamma_{USECASES}$' = ∅
- $E_{USECASES}$' = {e_1'}
  - $e_1' =^{e_USECASES'}$ (“String”, Σ^*)
- $R_{USECASES}$' = ∅

Aplicando a função de mesclagem $@_{MM}$ sobre $MM_{USECASES}$ e $MM_{USECASES}$', obtém-se um metamodelo resultante estendido. Aplicando a eq. (23) sobre $MM_{USECASES}$ e $MM_{USECASES}$', tem-se:

$$@_{MM}(\text{“NewUseCasesMetamodel”}, MM_{USECASES}, MM_{USECASES}') =$$
$$(\text{“NewUseCasesMetamodel”}, C_{USECASES} \ @_C \ C_{USECASES}', \Gamma_{USECASES} \ @_\Gamma \ \Gamma_{USECASES}',$$
$$E_{USECASES} \ @_E \ E_{USECASES}', R_{USECASES} \ @_R \ R_{USECASES}',$$
$$descriptor_{c_USECASES} \ @_{dc} \ descriptor_{c_USECASES}',$$
$$descriptor_{e_USECASES} \ @_{de} \ descriptor_{e_USECASES}')$$

Recorrendo-se às definições dos operadores apresentados na subseção 4.10.2, tem-se os resultados a seguir. Foram destacados em negrito a aplicação dos

operadores e seus operandos, bem como os resultados obtidos. Os passos intermediários são apresentados sem destaque.

$$\mathbf{C_{USECASES}\ @_C\ C_{USECASES}' =} \{c_1, c_2, c_3\} \cup \{c_1', c_2', c_3'\} \setminus \{c_1', c_2'\} = \mathbf{\{c_1, c_2, c_3, c_3'\}}$$

$$\mathbf{P_1\ @_P\ P_1' =} \varnothing \cup \{p_{11}, p_{12}, p_{13}, p_{14}, p_{15}\} \cup \varnothing = \mathbf{\{p_{11}, p_{12}, p_{13}, p_{14}, p_{15}\}}$$

$$\mathbf{update_target_types(P_1\ @_P\ P_1') =}$$
$$\{update_target_type(p_{11}), update_target_type(p_{12}), update_target_type(p_{13}),$$
$$update_target_type(p_{14}), update_target_type(p_{15})\} =$$
$$\mathbf{\{(\text{“Name”}, e_1, 1..1), (\text{“PreConditions”}, e_1, 1..1), (\text{“Description”}, e_1, 1..1),}$$
$$\mathbf{(\text{“PostConditions”}, e_1, 1..1), (\text{“Exceptions”}, e_1, 1..1)\}}$$

$$\mathbf{P_2\ @_P\ P_2' =} \varnothing \cup \{p_{21}\} \cup \{p_{21}'\} = \mathbf{\{p_{21}, p_{21}'\}}$$

$$\mathbf{update_target_types(P_2\ @_P\ P_2') =}$$
$$\{update_target_type(p_{21}), update_target_type(p_{21}')\} =$$
$$\mathbf{\{(\text{“Name”}, e_1, 1..1), (\text{“Stereotype”}, e_1, 1..1)\}}$$

$$\mathbf{update_target_types(P_3') =}$$
$$\{update_target_type(p_{31}'), update_target_type(p_{32}'), update_target_type(p_{33}')\} =$$
$$\mathbf{\{(\text{“UseCase1”}, c_1, 1..1), (\text{“UseCase2”}, c_1, 1..1), (\text{“Stereotype”}, e_1, 1..1)\}}$$

$$\mathbf{descriptor_{c_USECASES}\ @_{dc}\ descriptor_{c_USECASES'} =}$$
$$\{(c_3, (\text{“AssociationUseCaseActor”}, P_3))\} \cup$$
$$\{(c_3', (\text{“AssociationUseCases”}, update_target_types(P_3')))\} \cup$$
$$\{(c_1, (\text{“UseCase”}, update_target_types(P_1\ @_P\ P_1'))),$$
$$(c_2, (\text{“Actor”}, update_target_types(P_2\ @_P\ P_2')))\} =$$
$$\{(c_3, (\text{“AssociationUseCaseActor”}, \{(\text{“Name”}, e_1, 1..1),$$
$$(\text{“UseCase”}, c_1, 1..1), (\text{“Actor”}, c_2, 1..1)\}))\} \cup$$
$$\{(c_3', (\text{“AssociationUseCases”}, \{(\text{“UseCase1”}, c_1, 1..1), (\text{“UseCase2”}, c_1, 1..1),$$
$$(\text{“Stereotype”}, e_1, 1..1)\}))\} \cup$$
$$\{(c_1, (\text{“UseCase”}, \{(\text{“Name”}, e_1, 1..1), (\text{“PreConditions”}, e_1, 1..1),$$
$$(\text{“Description”}, e_1, 1..1), (\text{“PostConditions”}, e_1, 1..1), (\text{“Exceptions”}, e_1, 1..1)\})),$$

$(c_2$, ("Actor", {("Name", e_1, 1..1), ("Stereotype", e_1, 1..1)}))} =

{(c_3, ("AssociationUseCaseActor", {("Name", e_1, 1..1), ("UseCase", c_1, 1..1), ("Actor", c_2, 1..1)})),

(c_3', ("AssociationUseCases", {("UseCase1", c_1, 1..1), ("UseCase2", c_1, 1..1), ("Stereotype", e_1, 1..1)})),

(c_1, ("UseCase", {("Name", e_1, 1..1), ("PreConditions", e_1, 1..1), ("Description", e_1, 1..1), ("PostConditions", e_1, 1..1), ("Exceptions", e_1, 1..1)})),

(c_2, ("Actor", {("Name", e_1, 1..1), ("Stereotype", e_1, 1..1)}))}

$$\Gamma_{USECASES} \ @_{\Gamma} \ \Gamma_{USECASES}' = \varnothing \ @_{\Gamma} \ \varnothing = \varnothing = \Gamma_{USECASES}$$

$$E_{USECASES} \ @_{E} \ E_{USECASES}' = \{e_1\} \cup \{e_1'\} \setminus \{e_1'\} = \{e_1\} = E_{USECASES}$$

$$descriptor_{e_USECASES} \ @_{de} \ descriptor_{e_USECASES}' = \varnothing \cup \varnothing \cup \{(e_1, (\text{"String"}, \Sigma^* \cup \Sigma^*))\} = \{(e_1, (\text{"String"}, \Sigma^*))\}$$

$$R_{USECASES} \ @_{R} \ R_{USECASES}' = R_{USECASES} \cup R_{USECASES}' = \{r_1, r_2, r_3, r_4\} \cup \varnothing = \{r_1, r_2, r_3, r_4\} = R_{USECASES}$$

O metamodelo final resultante da mesclagem pode então ser reescrito a seguir, transpondo os resultados das aplicações dos operadores. Foi adicionado o sufixo '' para representar os objetos resultantes da operação de mesclagem.

$$MM_{USECASES}'' = (\text{"NewUseCasesMetamodel"}, C_{USECASES}'', \Gamma_{USECASES}'', E_{USECASES}'', R_{USECASES}'', descriptor_{c_USECASES}'', descriptor_{e_USECASES}'')$$

Em que:

- $C_{USECASES}'' = \{c_1, c_2, c_3, c_3'\}$
  - $c_1 =^{c_USECASES''}$ ("UseCase", P_1'')
    - $P_1'' = \{p_{11}, p_{12}, p_{13}, p_{14}, p_{15}\}$

- p_{11} = (“Name”, e_1, 1..1)
- p_{12} = (“PreConditions”, e_1, 1..1)
- p_{13} = (“Description”, e_1, 1..1)
- p_{14} = (“PostConditions”, e_1, 1..1)
- p_{15} = (“Exceptions”, e_1, 1..1)

  - $c_2 =^{c_USECASES''}$ (“Actor”, P_2’’)
    - P_2’’ = {p_{21}, p_{21}’’}
      - p_{21} = (“Name”, e_1, 1..1)
      - p_{21}’’ = (“Stereotype”, e_1, 1..1) = update_target_type(p_{21}’)
  - $c_3 =^{c_USECASES''}$ (“AssociationUseCaseActor”, P_3’’)
    - P_3’’ = {p_{31}, p_{32}, p_{33}}
      - p_{31} = (“Name”, e_1, 1..1)
      - p_{32} = (“UseCase”, c_1, 1..1)
      - p_{33} = (“Actor”, c_2, 1..1)
  - $c_3' =^{c_USECASES''}$ (“AssociationUseCases”, P_3’’’)
    - P_3’’’ = {p_{31}’’’, p_{32}’’’, p_{33}’’’}
      - p_{31}’’’ = (“UseCase1”, c_1, 1..1) = update_target_type(p_{31}’)
      - p_{32}’’’ = (“UseCase2”, c_1, 1..1) = update_target_type(p_{32}’)
      - p_{33}’’’ = (“Stereotype”, e_1, 1..1) = update_target_type(p_{33}’)

- $\Gamma_{USECASES}$’’ = $\Gamma_{USECASES}$ = $\varnothing$
- $E_{USECASES}$’’ = $E_{USECASES}$ = {e_1}
  - $e_1 =^{e_USECASES''}$ (“String”, Σ^*)
- $R_{USECASES}$’’ = $R_{USECASES}$ = {r_1, r_2, r_3, r_4}
  - r_1: $\forall o$ UseCase(o) $\Rightarrow \neg$(UseCase.Name(o) = ε)
  - r_2: $\forall o_1, o_2$ UseCase(o_1) $\wedge$ UseCase(o_2) $\wedge$ (UseCase.Name(o_1) = UseCase.Name(o_2)) $\Rightarrow o_1 = o_2$

- r_3: $\forall o\ Actor(o) \Rightarrow \neg(Actor.Name(o) = \varepsilon)$
- r_4: $\forall o_1, o_2\ Actor(o_1) \wedge Actor(o_2) \wedge (Actor.Name(o_1) = Actor.Name(o_2)) \Rightarrow o_1 = o_2$

Observa-se no metamodelo resultante que a propriedade *Stereotype* foi adicionada à metaclasse *Actor* em relação ao metamodelo original. Também foi adicionada a metaclasse que permite instanciar associações entre casos de uso, denominada *AssociationUseCases*.

Observa-se que o mecanismo de mesclagem de metamodelos apresentado baseia-se em incrementos que geram ampliações. Subtrações, ou seja, incrementos que removem partes do metamodelo a ser mesclado, não são previstas. Fondement et al. (2013) apresentam uma discussão sobre o assunto, apresentando o conceito de *Package Unmerge*, que pode servir como um caminho para futuras extensões do mecanismo de mesclagem de metamodelos do SBMM para subtrações.

4.11 MODELOS DE MARCAÇÃO

Conforme visto na seção 2.8, um modelo de marcação estabelece variáveis extras permitidas em um metamodelo. Um conjunto de atribuição de valores para essas variáveis serve para complementar um modelo, sem, no entanto, modificá-lo.

O MOF aborda esse conceito através da capacidade de extensão descrita em sua especificação (OMG, 2015a), predominantemente em linguagem natural. Os outros formalismos estudados no Capítulo 3 não abordam esse tema.

Deseja-se formalmente definir um modelo de marcação no SBMM, mas observa-se que seu conceito é muito similar a propriedades cujos tipos são enumerações. Para evitar a introdução de novas definições similares às já existentes, define-se um modelo de marcação como um metamodelo com uma restrição adicional: o tipo alvo de todas as propriedades p_{jx} de todas as metaclasses $c_j \in C$ devem ser obrigatoriamente enumerações.

Ou seja, um modelo de marcação é um metamodelo tal como definido pela eq. (1) com a restrição adicional de que para toda propriedade p_{jx} = (propertyname$_{jx}$,

$propertytype_{jx}$, $mult_{jx}$), deve-se ter $propertytype_{ix} \in E$ em vez de $propertytype_{jx} \in C \cup E$.

Quando um modelo de marcação é observado por si só, talvez não seja possível identificar se tem a intenção de ser usado como metamodelo ou como modelo de marcação, afinal metamodelos cujos tipos alvo de propriedades sejam exclusivamente enumerações também são permitidos.

Os modelos de marcação são usados no contexto de moldar conjuntos de marcações, que por sua vez proveem informações complementares a modelos para funções de mapeamento responsáveis por definir transformações. Sob essa ótica, a interpretação de um modelo de marcação difere daquela de um metamodelo, embora sejam definidos como o mesmo tipo de sétupla.

Enquanto nos metamodelos as metaclasses são abstrações de conceitos de modelagem (classe, associação, caso de uso, atividade, etc.), nos modelos de marcação as metaclasses servem para introduzir variáveis (propriedades) complementares cujos tipos são enumerações. Os modelos de marcação devem ser interpretados em conjunto com o metamodelo para o qual foram definidos. As metaclasses do modelo de marcação levam o nome das metaclasses "reais" do metamodelo correspondente.

Para clarificar esta interpretação, é apresentado um exemplo a seguir. Considerando o metamodelo de diagrama de classes simplificado apresentado na seção 4.3, cria-se a marcação *InstanceQuantity* exemplificada na seção 2.8 para a metaclasse *Class*. Seja $MMark_{CLASSES}$ o modelo de marcação da eq. (43).

$$MMark_{CLASSES} = (\text{"MarkingModelClasses"}, C_{MMARK}, \Gamma_{MMARK}, E_{MMARK}, R_{MMARK}, descriptor_{c_MARK}, descriptor_{e_MARK}) \quad (43)$$

Em que:

- $C_{MMARK} = \{c_1\}$
  - $c_1 =^{c_MARK} (\text{"Class"}, P_1)$

- $P_1 = \{p_{11}\}$
  - p_{11} = (“InstanceQuantity”, e_1, 1..1)
- $\Gamma_{MMARK} = \varnothing$
- $E_{MMARK} = \{e_1\}$
  - $e_1 =^{e_MARK}$ (“Integer”, $\mathbb{Z}$)
- $R_{MMARK} = \varnothing$

A metaclasse $c_1 \in C$ não é uma metaclasse a ser mesclada com aquela de mesmo nome de $MM_{CLASSES}$ da seção 4.3, eq. (14). Ou seja, não se trata de um novo metamodelo para gerar uma modificação sobre primeiro. Sob o ponto de vista de modelos de marcação, permitirá definir um tipo específico de modelo, chamado de conjunto de marcações, que provê informações adicionais para funções de mapeamento que trabalham com o modelo. Entretanto, essas informações adicionais não fazem parte do modelo, funcionando como a analogia do “plástico transparente” colocado sobre o mesmo (MELLOR et al., 2004), conforme seção 2.8.

Por outro lado, nada impede que o modelo de marcação seja mesclado com o metamodelo pretendido, gerando um metamodelo estendido. Nesse caso, os valores das marcações ficariam armazenados junto com os modelos conformes ao metamodelo estendido, não sendo mais o “plástico transparente”.

Um conjunto de marcações, por sua vez, consiste em uma instância de um modelo de marcação. Sendo assim, uma vez que modelo de marcação foi definido como um metamodelo, pode-se então definir um conjunto de marcações SMark como um modelo específico de acordo com as definições da eq. (15), o qual é apresentado na eq. (44).

$$SMark = (name_{SMark}, name_{MMark}, I_{SMark}, descriptor_{i_SMark}) \qquad (44)$$

Em que:

- $name_{SMark} \in \Sigma^+$ é uma cadeia não-vazia que representa o nome do conjunto de marcações;
- $name_{MMark}$ identifica o modelo de marcação pretendido, que define as variáveis adicionais cujos valores serão carregados por SMark;
- I_{SMark} é o conjunto de elementos que carregam valores, não sendo instâncias "reais";
- $descriptor_{i_SMark}$ é a função que mapeia cada $i \in I_{SMark}$ em seu descritor que de fato carrega os *slots* com valores.

De forma análoga ao que foi visto para modelos de marcação, enquanto nos modelos as instâncias são elementos de modelagem correspondentes a entidades de sua metaclasse, nos conjuntos de marcações as instâncias servem para carregar valores de variáveis (*slots*) complementares cujas propriedades têm enumerações como tipos. Os conjuntos de marcação devem ser interpretados juntamente com o modelo para o qual foram definidos. As instâncias do conjunto de marcações levam o nome das instâncias "reais" do modelo correspondente sobre as quais os valores carregados serão aplicados.

Seja um exemplo de conjunto de marcações relacionado ao modelo de marcação $MMark_{CLASSES}$ da eq. (43) a ser aplicado sobre o modelo do sistema escolar apresentado na seção 4.8 pela eq. (19). Seja $SMark_{CLASSES}$ o conjunto de marcações estabelecido pela eq. (45).

$$SMark_{ESCOLAR} = (\text{"MarkingSetClasses"}, \text{"MarkingModelClasses"}, I_{SMarkESCOLAR}, descriptor_{i_SMarkESCOLAR}) \quad (45)$$

Em que:

- $I = \{i_1\}$
  - $i_1 =^{i_SMarkESCOLAR}$ ("CAluno","Class", S_1)
    - $S_1 = \{s_{11}\}$
      - s_{11} = ("InstanceQuantity", "1000")

O conjunto de marcações acima atribui o valor 1000 para a variável *InstanceQuantity* vinculada à classe *CAluno*. Conforme visto, esse valor poderia ser usado por uma função de mapeamento, por exemplo, para gerar uma transformação adequada que prevê em torno de 1000 alunos cadastrados no sistema, tratando esse volume de dados da forma mais apropriada.

4.12 TRANSFORMAÇÃO DE MODELOS

A transformação de modelos tem papel fundamental na MDE, haja vista que o objetivo final de um processo MDE é transformar os modelos de software em código-fonte ou em alguma forma idealmente pronta para compilação ou execução. Nesta seção são apresentadas definições sobre o conceito de transformações de modelos dentro do contexto do formalismo SBMM. Essas definições servem para dar base teórica à construção de ferramentas de transformações de modelos que o utilizam como base.

Transformações são definidas por funções de mapeamento. Existem basicamente dois tipos de transformações de interesse da MDE: modelo-para-modelo e modelo-para-código.

Seja U_M o universo de todos os modelos possíveis de acordo com a definição da eq. (15). Uma função de mapeamento f, do tipo modelo-para-modelo, toma n modelos como entrada e produz um modelo de saída. A expressão (46) reflete este fato.

$$f: U_M{}^n \rightarrow U_M \tag{46}$$

Do ponto de vista da implementação de f, seja uma fórmula, uma máquina de Turing, um programa de computador, etc., devem ser de “conhecimento” da função os metamodelos para os quais seja esperada conformidade para cada modelo de entrada. Isso porque a função precisa reconhecer os elementos que aparecem em cada modelo para processá-los, combiná-los e transformar todo o conjunto no modelo de saída. Ou seja, por essa definição o metamodelo está presente de alguma forma intrínseca na definição da função de mapeamento. Presume-se que

os modelos de entrada estejam conformes aos metamodelos esperados, caso contrário a função pode ser indefinida para os valores de entrada ou apresentar problemas durante sua computação em uma implementação concreta.

Seja Σ um alfabeto de interesse para código-fonte, tais como o conjunto de caracteres ASCII ou Unicode. Uma função de mapeamento g, do tipo modelo-para-código, toma n modelos como entrada e produz uma cadeia de símbolos de Σ como saída, de acordo com a eq. (47).

$$g: U_M{}^n \rightarrow \Sigma^* \tag{47}$$

Vale também para este tipo de transformação a observação sobre o “conhecimento” que a função precisa ter sobre os metamodelos de entrada.

As funções de mapeamento eventualmente podem apresentar função inversa, permitindo a execução de transformações bidirecionais. Embora nem sempre possíveis ou simples de serem obtidas, as inversas das funções de mapeamento permitem a realização de engenharia reversa. Um simples exemplo que impede a obtenção da função inversa exata é o descarte de comentários. Considere-se uma função de mapeamento do tipo modelo-para-código que toma um diagrama de classes UML como entrada e gera código Java com a estrutura das classes como saída. Se a função é definida para descartar todos os comentários inseridos no diagrama UML na forma de notas, uma eventual implementação da função inversa não conseguiria recuperar esses comentários e, portanto, não geraria um modelo exatamente igual ao original.

Uma das características desejadas para a MDE é a capacidade de refinamento de modelos, conforme apresentado na seção 2.9. As definições apresentadas até aqui para funções de mapeamento não tratam explicitamente desse conceito, pois a reexecução de uma transformação sobrescreve eventuais alterações que o desenvolvedor tenha feito sobre um modelo ou código gerado anteriormente.

Para permitir que partes do modelo modificado manualmente sejam preservadas após a reexecução da transformação, Xiong et al. (2007) sugere a inclusão do modelo (ou cadeia) resultante da aplicação anterior da função de mapeamento, após passar pelas modificações manuais, como mais uma entrada para a mesma. Desta

maneira, a função poderia ter alguma capacidade de localizar as alterações feitas manualmente pelo desenvolvedor no último modelo (ou cadeia) gerado(a) para conseguir preservá-las. Sendo assim, pode-se reescrever as definições apresentadas para explicitar essa capacidade, conforme (48) e (49).

$$f: U_M{}^n \times U_M \rightarrow U_M \quad (48)$$

$$g: U_M{}^n \times \Sigma^* \rightarrow \Sigma^* \quad (49)$$

Foi introduzido mais um produto cartesiano no domínio de f, dando espaço ao último modelo gerado, com as modificações manuais, para servir como argumento. Idem para g, mas nesse caso trata-se da última cadeia gerada.

Para exemplificar, seja um modelo M criado por um desenvolvedor. Deseja-se realizar uma transformação sobre o mesmo, definida por uma função h: $U_M \times \Sigma^* \rightarrow \Sigma^*$, ou seja, que toma um único modelo como entrada e produz uma cadeia de saída. A cadeia que faz parte do par dos argumentos de entrada tem por intenção suportar a cadeia gerada na última aplicação da função, ou seja, na última transformação.

Denomina-se a primeira versão de M como M_{v1}. A primeira aplicação de h utiliza a cadeia vazia ε como entrada, já que não houve aplicação prévia da mesma. Essa aplicação de h gera a cadeia α_{v1}, isto é, transforma M_{v1} em α_{v1}.

$$\alpha_{v1} = h\ (M_{v1}, \varepsilon)$$

Suponha-se que o desenvolvedor modifique manualmente a cadeia α_{v1}, que se trata de um código-fonte. Essa modificação pode ser, por exemplo, a implementação de operações em Java sobre a estrutura de classes gerada a partir do diagrama UML de classes M_{v1}. Essa modificação resulta na cadeia α_{v1}'. Considera-se também que o desenvolvedor fez novas mudanças no modelo gerando a segunda versão M_{v2}, através da introdução de novas classes, por exemplo. A segunda aplicação de h deve receber α_{v1}' como argumento para conseguir preservar as alterações manuais no código-fonte.

$$\alpha_{v2} = h\ (M_{v2},\ \alpha_{v1}')$$

Até aqui não foi abordado o mérito de como essa preservação é feita, ou quais tipos de preservação de alterações manuais no modelo ou código são possíveis. Esse aspecto será abordado mais adiante, especificamente na linguagem *MOF Models To Text Transformation* (MOFM2T), apresentada no Apêndice A. Apenas introduziu-se até então, na modelagem das funções de mapeamento, a capacidade de trabalhar com a versão anterior do modelo ou código modificado pelo desenvolvedor.

É interessante observar que as definições de funções de mapeamento aqui apresentadas automaticamente suportam o recebimento de conjuntos de marcações como argumentos. Isso porque o SBMM define conjuntos de marcações como tipos específicos de modelos, de acordo com a seção 4.11, mas que também pertencem ao universo de modelos possíveis U_M. Assim, duas funções de mapeamento h_1 e h_2 que recebem o mesmo tipo de modelo mas servem para gerar código-fonte para plataformas ou linguagens distintas, podem ser definidas para receber conjuntos de marcações além do modelo principal. Por exemplo, se h_1 deve receber dois conjuntos de marcações com informações adicionais para gerar código para a plataforma A, e h_2 deve receber um único conjunto de marcações específico para a plataforma B, então:

$$h_1: U_M{}^3 \times \Sigma^* \rightarrow \Sigma^*$$

$$h_2: U_M{}^2 \times \Sigma^* \rightarrow \Sigma^*$$

A função h_1 recebe três modelos: o modelo principal e seus dois conjuntos de marcações. Enquanto isso, h_2 recebe o modelo principal e um conjunto de marcação.

Outra capacidade desejável para funções de mapeamento é variar a saída gerada conforme o incremento do modelo em relação à execução anterior. Por exemplo, seja uma função de mapeamento uml_to_ddl do tipo modelo-para-código que toma como entrada um modelo correspondente a um diagrama de classes UML e produz

sentenças *Structured Query Language* (SQL) para criação de tabelas em um banco de dados relacional. Deseja-se que, na primeira execução da transformação, sejam produzidas sentenças do tipo *CREATE TABLE* para todas as tabelas necessárias. A partir da segunda execução, passa a ser desejada a produção de sentenças do tipo *ALTER TABLE*, ou vazias, no que se aplica às classes previamente existentes que sofreram alguma ou nenhuma alteração, respectivamente. Também se deseja sentenças do tipo *DROP TABLE* no que se refere a classes que existiam na versão anterior do modelo mas que não existem mais na última versão. Para isso, a função de mapeamento deve ter acesso à versão anterior de cada modelo e conjunto de marcações de entrada de modo a conseguir computar os incrementos e decidir sobre a saída a ser gerada. No exemplo anterior que definiu as funções h_1 e h_2, ao passar a receber a versão anterior do modelo correspondente ao diagrama de classes e seus conjuntos de marcações, elas passariam a ser:

$$h_1: U_M{}^6 \times \Sigma^* \rightarrow \Sigma^*$$

$$h_2: U_M{}^4 \times \Sigma^* \rightarrow \Sigma^*$$

Um dos problemas mais relevantes para aplicação prática da MDE é como descrever as transformações que cada função de mapeamento deve executar. Ferramentas para MDE que suportam transformações devem prover ao usuário algum mecanismo que o permita descrever as funções de mapeamento de seu interesse, eventualmente fornecendo uma biblioteca de funções pré-definidas. Em outras palavras, é desejável que as ferramentas para MDE disponibilizem um motor (*engine*) de transformação capaz de ler descrições de funções de mapeamento e executá-las. Sendo assim, define-se motores de transformação de modo que o “conhecimento” sobre os metamodelos não seja intrínseco ao motor, mas sim passado na forma de argumentos.

Seja U_{MM} o universo de metamodelos possíveis de acordo com a definição da eq. (1). Um motor de transformação do tipo modelo-para-modelo te_model_to_model toma n modelos como entrada e m metamodelos. A função resulta em um modelo. Para não limitar n e m, o domínio de te_model_to_model será definido com base em uma tupla com um número arbitrário de modelos pertencente a $U_M{}^*$ e uma tupla com

um número arbitrário de metamodelos pertencente a U_{MM}^*. U_D denota o universo de descritores de funções de mapeamento. Ainda não se abordou o tipo de estrutura de dados que é necessário para descrever uma função de mapeamento. Isso dependerá da técnica escolhida e será discutido nas seções seguintes. Por enquanto, assume-se apenas que o universo U_D contém todos os objetos descritores possíveis dentro de certo contexto. A expressão (50) reflete a função te_model_to_model.

$$\text{te_model_to_model}: U_M^* \times U_{MM}^* \times U_D \rightarrow U_M \tag{50}$$

Cada modelo $M \in U_M$, por definição da eq. (15), referencia seu metamodelo por meio do nome. Todos os n modelos passados como argumentos de f devem ter seu metamodelo correspondente passados como argumentos também, para que o motor de transformação consiga encontrar as informações necessárias dos metamodelos utilizados. É observada a condição $m \leq n$ pois mais de um modelo pode referenciar um mesmo metamodelo. Por exemplo, quando as versões anterior e atual de um modelo são passadas para o motor, elas referenciam o mesmo metamodelo e, portanto, $m < n$. Dessa forma, no caso limite cada modelo referenciará um metamodelo distinto e, portanto, $m = n$.

O suporte a refinamento de modelos e conjuntos de marcações é implícito, pois os modelos adicionais de entrada que dão suporte a esses dois recursos são elementos adicionais da tupla pertencente a U_M^*.

Um motor do tipo modelo-para-código requer algumas adaptações na definição. Primeiramente por prover como saída uma cadeia de símbolos. E também por requerer uma cadeia de símbolos como entrada referente à versão anterior eventualmente modificada pelo desenvolvedor. Seja te_model_to_code um motor de transformação do tipo modelo-para-código de acordo com (51).

$$\text{te_model_to_code}: U_M^* \times U_{MM}^* \times \Sigma^* \times U_D \rightarrow \Sigma^* \tag{51}$$

Para todo modelo passado como argumento da função, o metamodelo referenciado também deve ser passado como argumento. Além disso, deve haver relação de

conformidade entre todos os modelos e seus metamodelos. O mesmo vale para conjuntos de marcações e seus modelos de marcação, que também são modelos e metamodelos, respectivamente. Caso essas restrições não sejam respeitadas, o motor de transformação pode ter problemas durante a computação da função de mapeamento, e em uma implementação prática isso se traduz em erros em tempo de execução ou geração de saídas inconsistentes. O descritor passado para o motor também deve estar relacionado aos metamodelos de entrada, e não a outros metamodelos.

O SBMM estabelece meios para descrever modelos e metamodelos. Cabe então investigar as possibilidades para descrever funções de mapeamento que compõem o universo U_D. Conforme visto na seção 2.6, existem basicamente três abordagens para descrição de funções de mapeamento: imperativa, baseada em arquétipo (ou *template*), e declarativa. O Apêndice A aborda algumas técnicas e linguagens já existentes para a descrição de funções de mapeamento, e também comenta sua aplicação com modelos SBMM.

4.13 FUNÇÕES PARA CONSTRUÇÃO E EDIÇÃO DE MODELOS

Uma ferramenta de edição de modelos construída utilizando o SBMM como formalismo subjacente deve manter estruturas de dados equivalentes às da definição de modelo apresentada na seção 4.7. Para especificar precisamente operações que possam ser feitas sobre modelos, esta seção apresenta definições de funções que servem como especificação para a construção de ferramentas.

A primeira operação de interesse é a inserção de uma instância no modelo. Ocorre, por exemplo, quando o desenvolvedor adiciona um caso de uso ou uma classe em um diagrama UML. A eq. (52) define a função insert_empty_instance que toma como entrada um modelo M = ($name_M$, $name_{MM}$, I, $descriptor_I$), o objeto de instância i $\notin$ I a ser adicionado, o nome da instância a ser adicionada $name_i \in \Sigma^+$ e o nome da metaclasse desejada metaclassname $\in \Sigma^+$. A função produz um modelo de saída M' = ($name_M$, $name_{MM}$, I', $descriptor_I$') correspondente ao original com a adição da nova instância sem nenhum *slot*.

$$\text{insert_empty_instance}(M, i, \text{instancename}, \text{metaclassname}) = M' = (\text{name}_M, \text{name}_{MM}, I', \text{descriptor}_i') \quad (52)$$

Tal que:

$$I' = I \cup \{ i \} \quad (53)$$

$$\text{descriptor}_i' = \text{descriptor}_i \cup \{ (i, (\text{instancename}, \text{metaclassname}, \emptyset)) \} \quad (54)$$

A eq. (53) produz o conjunto resultante de instâncias e a eq. (54) adiciona o mapeamento entre i e sua tripla descritiva na função descriptor_i. A aplicação de insert_empty_instance é uma operação atômica, mesmo estando descrita em mais de um passo. Ela não pode ser executada se já existir outra instância em I com o nome instancename, caso contrário o modelo resultante seria inválido. Sendo assim, uma ferramenta que implementa esta operação deve checar antes a condição estabelecida em (55).

$$\neg\exists\ (i', (w, u, S))\ ((i', (w, u, S)) \in \text{descriptor}_i) \wedge (w = \text{instancename}) \wedge (i \neq i') \quad (55)$$

Por outro lado, se a mesma instância for inserida no modelo duas vezes em sequência, sem edições intermediárias, ela não desrespeita a condição (55). Além disso, nesse caso, as operações (53) e (54) produzem I' e $\text{descriptor}_i'$ iguais a I e descriptor_i, respectivamente. Ou seja, supondo-se que a condição (55) é respeitada inicialmente, então insert_empty_instance(insert_empty_instance(M, i, name_i, name_c), i, name_i, name_c) = insert_empty_instance(M, i, instancename, metaclassname). Essa propriedade é importante para o estudo de sincronização de modelos e é retomada mais adiante na seção 5.3.

Já a função insert_or_edit_slot definida pela eq. (56) insere ou edita um *slot* referente à propriedade de nome propertyname na instância identificada por instancename com o valor val. Ela faz uso da função auxiliar remove_slot, que é apresentada na eq. (58), e atua sobre um conjunto de *slots* S.

$$\text{insert_or_edit_slot}(M, \text{instancename}, \text{propertyname}, \text{val}) = M'' = \quad (56)$$

$$(name_M, name_{MM}, I, descriptor_i'')$$

Tal que:

$$descriptor_i'' = \{ (i, (w, u, S)) \mid ((i, (w, u, S)) \in descriptor_i) \wedge (w \neq instancename)\} \cup \{(i, (w, u, remove_slot(S, propertyname) \cup \{(propertyname, val)\})) \mid ((i, (w, u, S)) \in descriptor_i) \wedge (w = instancename)\} \quad (57)$$

$$remove_slot(S, propertyname) = \{(w, v) \mid ((w, v) \in S) \wedge (w \neq propertyname)\} \quad (58)$$

A equação (57) monta o conjunto correspondente à função $descriptor_i''$ com base na união de dois conjuntos. O primeiro remove o descritor da instância de nome instancename de $descriptor_i$, enquanto o segundo o readiciona trocando o *slot* da propriedade identificada por propertyname (se existir tal *slot*) por um novo contendo o valor val. Caso não exista, ele é apenas adicionado pela operação de união que ocorre na terceira linha da equação. Observar que a dupla aplicação de insert_or_edit_slot equivale a uma única aplicação, como também ocorre para insert_empty_instance. Ou seja, insert_or_edit_slot(insert_or_edit_slot(M, instancename, propertyname, val), instancename, propertyname, val) = insert_or_edit_slot(M, instancename, propertyname, val).

A exclusão de um *slot* previamente adicionado é de interesse para limpar valores atribuídos a uma propriedade em uma instância. Ela é definida pela função delete_slot da eq. (59), que atua sobre um modelo. Também recorre à remove_slot, que atua em um conjunto de *slots*.

$$delete_slot(M, instancename, propertyname) = M''' = (name_M, name_{MM}, I, descriptor_i''') \quad (59)$$

Tal que:

$$descriptor_i''' = \{(i, (w, u, S)) \mid ((i, (w, u, S)) \in descriptor_i) \wedge \quad (60)$$

$$(w \neq instancename) \} \cup$$
$$\{(i, (w, u, remove_slot(S, propertyname)) \mid$$
$$((i, (w, u, S)) \in descriptor_i) \wedge (w = instancename) \}$$

A construção da eq. (60) é muito similar àquela da eq. (57), mas sem adicionar o *slot*.

A exclusão de uma instância do modelo também é importante, equivale ao desenvolvedor remover um caso de uso ou uma classe de um diagrama UML, por exemplo. Ela é definida pela função delete_instance da eq. (61).

$$delete_instance(M, instancename) = M'''' = (name_M, name_{MM}, I'''', descriptor_i'''') \quad (61)$$

Em que:

$$I'''' = I \setminus \{i \mid (i \in I) \wedge (descriptor_i(i) = (w, u, S)) \wedge (w = instancename)\} \quad (62)$$

$$descriptor_i'''' = \{(i, (w, u, S)) \mid ((i, (w, u, S)) \in descriptor_i) \wedge (w \neq instancename) \} \quad (63)$$

Ou seja, pela eq. (62) são incluídas no conjunto I'''' todas as instâncias de I e removida aquela cujo nome é igual a instancename. Pela eq. (63), complementarmente, são mantidos no conjunto correspondente à função $descriptor_i''''$ os mapeamentos de todas as instâncias exceto aquela de nome instancename. Caso não exista instância de nome instancename no modelo M, as expressões (62) e (63) não alteram os conjuntos I'''' e $descriptor_i''''$ e, portanto, nesse caso M'''' = M. Assim como para insert_empty_instance, a aplicação de delete_instance também é uma operação atômica.

Uma ferramenta que implementa delete_instance deve considerar que o modelo pode ficar inconsistente se for excluída uma instância referenciada como valor em algum *slot* de outra instância. Por isso, deve ser feita uma verificação prévia de referência e sugere-se um aviso ao usuário informando que a instância referenciada não pode ser excluída ou que o valor de algum *slot* do modelo perderá a referência.

Pelas equações, verifica-se que as funções delete_slot e delete_instance também podem ser aplicadas duas vezes em sequência sem afetar o resultado. Ou seja, delete_slot(delete_slot(M, instancename, propertyname), instancename, propertyname) = delete_slot(M, instancename, propertyname) e delete_instance(delete_instance(M, instancename), instancename) = delete_instance(M, instancename).

Dentre os outros formalismos estudados, funções de edição de modelos são propostas apenas por Jackson, Levendovszky e Balasubramanian (2013), adaptadas para seu contexto. Em especial, elas têm utilidade na especificação de métodos de sincronização de modelos, tais como em Xiong et al. (2007), em que uma operação sobre um modelo implica na execução de outra operação sobre outro modelo para mantê-los sincronizados.

4.14 FUNÇÕES PARA CONSULTA SOBRE MODELOS

Assim como funções para edição de modelos têm sua utilidade, funções que representam consultas sobre modelos também têm a sua. Na seção 5.2, mais adiante, elas serão usadas em um exemplo de aplicação teórica do SBMM, a descrição de modelos executáveis.

A função query_instance_by_name(M, instancename), definida em (64), retorna um subconjunto de instâncias de M cujo nome é instancename. Como um nome de instância é suposto ser único no modelo, então o resultado esperado da função é um conjunto unitário ou um conjunto vazio.

$$\begin{gathered}\text{query_instance_by_name}(M, \text{instancename}) = \\ \{i \mid (i \in I) \wedge (\text{descriptor}_i(i) = (w, u, S)) \wedge (w = \text{instancename})\}\end{gathered} \tag{64}$$

Já a função query_instance_by_prop(M, metaclassname, propertyname, val) retorna um subconjunto de instâncias de I cujo tipo é a metaclasse identificada por metaclassname e cujos *slots* que se referem à propriedade de nome propertyname possuem val como seu valor. Essa função é especificada em (65).

$$\begin{aligned}&\text{query_instance_by_prop(M, metaclassname, propertyname, val)} = \\ &\{i \mid (i \in I) \wedge (descriptor_i(i) = (w, u, S)) \wedge (u = metaclassname) \wedge \\ &(\exists\ (y, v)\ ((y, v) \in S) \wedge (y = propertyname) \wedge (v = val)\ )\ \}\end{aligned} \quad (65)$$

A função query_prop_value(M, instancename, propertyname), por sua vez, retorna o valor contido pelo *slot* relacionado à propriedade identificada por propertyname na instância identificada por instancename do modelo M. Como explicado na seção 4.7, o valor pode ser um conjunto quando o limite superior da multiplicidade da propriedade for maior que 1. A função query_prop_value é especificada por (66).

$$\text{query_prop_value(M, instancename, propertyname)} = val \quad (66)$$

Tal que:

$$\begin{aligned}&\exists\ (w, u, S)\ ((w, u, S) \in Im(descriptor_i)) \wedge (w = instancename) \wedge \\ &(\exists\ (y, v)\ ((y, v) \in S) \wedge (y = propertyname)) \Rightarrow (val = v)\end{aligned} \quad (67)$$

A expressão (67) assume que existe no máximo um *slot* para cada valor de propriedade, o que é válido para modelos SBMM bem formados. Se não houver instância nomeada por instancename no modelo M, então query_prop_value retorna um conjunto vazio conforme (68).

$$\begin{aligned}&\neg\exists\ (w, u, S)\ ((w, u, S) \in Im(descriptor_i)) \wedge (w = instancename) \wedge \\ &(\exists\ (y, v)\ ((y, v) \in S) \wedge (y = propertyname)) \\ &\Rightarrow (val = \emptyset)\end{aligned} \quad (68)$$

Observe que algumas definições de funções de consulta sobre modelos apresentadas nesta seção foram usadas em definições de funções de construção de modelos. Por exemplo, a eq. (62) poderia ser reescrita em termos de (64).

4.15 DESENVOLVIMENTOS ADICIONAIS SOBRE O SBMM

As construções apresentadas nesse capítulo não se limitam em si mesmas, podendo ser utilizadas em novos desenvolvimentos teóricos sobre conceitos da MDE. Por exemplo, considerando as definições algébricas para a função de mesclagem de metamodelos $@_{MM}$, é possível apresentar uma prova de que se trata de uma operação comutativa, ou seja, que a ordem dos metamodelos pode ser trocada sem alterar o resultado. Essa prova não está apresentada neste texto, sugerindo-se como trabalho futuro ou exercício para o leitor.

O Capítulo 5 dá continuidade à elaboração de novas construções com utilização do SBMM, em particular abordando dois tópicos de pesquisa da MDE: modelos executáveis e sincronização de modelos, servindo também como forma de validar a real utilidade teórica do formalismo proposto nessa pesquisa.

5 APLICAÇÕES TEÓRICAS

5.1 INTRODUÇÃO

Este capítulo apresenta duas aplicações teóricas do formalismo SBMM. A primeira delas refere-se a uma conceituação precisa sobre modelos executáveis, uma alternativa às transformações de modelos para aplicação da MDE. Inclui-se também o conceito de modelo executável adaptativo. A segunda aplicação diz respeito a definições e propriedades sobre sincronização de modelos.

5.2 MODELOS EXECUTÁVEIS

5.2.1 **Introdução e Trabalhos Relacionados**

Um modelo executável é um tipo de modelo que pode ser executado. Algumas DSMLs permitem a criação de modelos executáveis e, por isso, são conhecidas como *Executable DSMLs* (x-DSML). A execução de modelos pode ser alcançada não apenas por transformações modelo-para-código, mas também por interpretação direta de modelos. Modelos escritos em uma *Interpreted DSML* (i-DSML) podem ser interpretados de acordo com uma semântica por um motor de execução (CARIOU et al., 2013). Com essa abordagem, nenhuma transformação precisa ocorrer e o modelo é diretamente executado. A executabilidade de modelos se tornou um tópico chave de pesquisa na MDE (COMBEMALE; CRÉGUT; PANTEL, 2012).

Esta seção apresenta uma caracterização precisa de modelos executáveis, e também de semânticas de execução, utilizando o SBMM como formalismo subjacente.

O termo “metamodelo executável” é apenas um atalho para expressar que modelos conformes a ele são executáveis (CARIOU et al., 2011). O metamodelo em si não é um artefato executável.

Modelos executáveis são discutidos em vários trabalhos, sendo que o seguinte consenso é identificado por Cariou et al. (2013):

- Alguns tipos de modelo são executáveis, outros não;
- Um motor (*engine*) é responsável por executar o modelo;
- Se o modelo apresenta em si todas as informações necessárias para sua execução por um motor, então é dito ser autocontido.

Um modelo autocontido apresenta *slots* de propriedades que explicitamente definem o estado de execução do modelo (p. ex., o estado de uma máquina de estados pode ter uma propriedade *booleana isCurrent* para determinar qual é o estado corrente em um dado passo de execução).

Assim como observado para o termo “metamodelo executável”, o termo “metamodelo autocontido” também é um atalho para expressar que, na verdade, os modelos conformes ao metamodelo são autocontidos.

Existe uma classificação geral que pode ajudar a identificar modelos executáveis: a classificação produto ou processo. Segundo ela, modelos podem expressar produtos ou processos, independentemente do tipo de sistema sob consideração. A ideia central é que modelos de processo possibilitam a executabilidade de seu conteúdo porque contêm conceitos diretamente relacionados à execução: ponto de início, ponto de término, passo de evolução, etc. (CARIOU et al., 2013). Por outro lado, modelos de produto são mais estruturais, tais como diagramas de classes UML, e esses não expressam diretamente aspectos de comportamento e interação.

Essa classificação pode não ser muito precisa para determinar modelos executáveis. Observa-se um exemplo de modelo que descreve uma interface de usuário do tipo *Create/Retrieve/Update/Delete* (CRUD) para objetos persistentes em aplicativos de negócio (CANOVAS; CUGNASCA, 2015b). O metamodelo correspondente define uma DSML que prevê apenas elementos estruturais da interface de usuário, podendo seus modelos ser classificados como modelos de produto. Entretanto, não se pode afirmar que esses modelos não são executáveis. Um motor, que implementa a semântica de execução, pode ser capaz de renderizar a interface e processar todas as interações com o usuário, tais como o clique em um botão para criar um objeto. Em todo o caso, essa classificação ainda provê alguma evidência sobre a executabilidade de um dado tipo de modelo.

Para dar continuidade à conceituação de modelos executáveis utilizando o SBMM como formalismo subjacente, consideram-se apenas os modelos executáveis de processo. Sendo assim, deve-se pressupor que um modelo de produto executável pode ser mapeado em um modelo de processo equivalente. Sempre pode ser criado um modelo de processo de mais baixo nível de abstração (em outra linguagem de modelagem) que incorpore explicitamente conceitos implícitos no modelo de mais alto nível. Mesmo que esse mapeamento não seja necessariamente de interesse da MDE, pois essa abordagem clama pelo aumento do nível de abstração, é de interesse para a conceituação proposta nesta seção. Esse princípio pode ser argumentado pela analogia com programas de computador: um programa escrito em alguma linguagem de alto nível, que torna implícitos detalhes computacionais de execução, sempre pode ser reescrito ou compilado em um código equivalente em uma linguagem de mais baixo nível, no qual detalhes computacionais de execução são mais evidentes (CANOVAS; CUGNASCA, 2016).

Cariou et al. (2011) apresentam uma visão de execução de modelos como transformações de modelos. É assumido que os modelos executáveis sob consideração são autocontidos. Um passo de execução é descrito como uma evolução de um modelo de origem (o modelo em seu estado de origem) para um modelo alvo (o modelo em seu próximo estado de execução). Tanto o modelo de origem como o modelo alvo são do mesmo tipo. O trabalho usa OCL para descrever a semântica de execução de máquinas de estado UML. Markovic e Baar (2008) usam QVT para expressar regras de reescrita que gradualmente computam os valores de expressões OCL.

Essas abordagens permitem uma definição mais precisa de modelos executáveis e mostram como um modelo gradualmente evolui durante sua execução. Entretanto, elas são baseadas em linguagens ou implementações computacionais, e seus fundamentos subjacentes não são tão explorados.

Combemale et al. (2012) propõem a modelagem do domínio semântico de uma x-DSML em duas partes: ações específicas de domínio e modelo de computação, ou *Model of Computation* (MoC). A primeira se refere a conceitos de execução específicos do domínio da DSML e suas semânticas, enquanto a segunda é responsável por agendar e orquestrar as ações em um modelo executável. O artigo apresenta experimentações desta proposta utilizando o *Kermeta Workbench*

(MULLER; FLEUREY; JÉZÉQUEL, 2005), ferramenta especializada em metamodelagem, e o ambiente ModHel'X (BOULANGER; HARDEBOLLE, 2008), que suporta a definição de MoCs. O artigo apresenta uma proposta sobre como combinar artefatos produzidos por essas duas ferramentas para obter a definição do domínio semântico de uma x-DSML. Dois benefícios dessa abordagem são a possibilidade de reusar o mesmo MoC com diferentes x-DSMLs e reusar ações específicas de domínio com diferentes MoCs para implementar pontos de variação semânticos na x-DSML. Embora traga contribuições, o trabalho foca em aspectos práticos e nem tanto em aspectos teóricos de modelos executáveis.

A semântica de execução de uma x-DSML usualmente está embutida de forma *hard coded* nas funções de execução ou de mapeamento presentes nas ferramentas de desenvolvimento de sistemas baseadas em modelos. Para tornar esta definição explícita, Combemale, Crégut e Pantel (2012) propõem um padrão de projeto para x-DSMLs. De acordo com esse padrão, metaelementos estruturais são separados de metaelementos de estados e eventos em diferentes pacotes (*packages*) MOF. Por ser um padrão relacionado à metamodelagem, também é conhecido como padrão de metamodelagem.

Metaelementos estruturais estão relacionados aos conceitos chave da linguagem e seus relacionamentos. O padrão estabelece que eles devem ser parte de um pacote denominado *Domain Definition MetaModel* (DDMM). Já a definição da informação que deve ser gerenciada em tempo de execução, isto é, aquela que torna o modelo autocontido (tal como a noção de estado corrente em uma máquina de estados), compõe o pacote *State Definition MetaModel* (SDMM). Tais dados são explicitamente definidos no metamodelo e devem ser manipulados e registrados no modelo na forma de instâncias de metaclasses e valores de propriedades em *slots*, ao invés de ficarem ocultos e implícitos na ferramenta de execução de modelos. O pacote *Event Definition MetaModel* (EDMM), por sua vez, especifica eventos concretos que direcionam a execução dos modelos conformes ao metamodelo da x-DSML. Ainda de acordo com o padrão, o MoC deve ser especificado no pacote *Trace Management MetaModel* (TM3), sendo que este é reusado por todas as x-DSMLs baseadas no mesmo MoC. Por fim, tanto o mapeamento semântico quanto as interações com o ambiente são abstraídos pelo pacote *Semantics*. Ele descreve como o modelo em execução (cujo estado de execução é especificado pelo SDMM)

evolui de acordo com os estímulos especificados pelo EDMM através de eventos. O pacote *Semantics* pode ser especificado por uma função de transição através de uma linguagem de ação (semântica operacional) ou implicitamente definido pelo mapeamento em outra linguagem (semântica translacional).

Sendo assim, de acordo com esse padrão, um metamodelo que define uma x-DSML é composto por cinco pacotes: DDMM, SDMM, EDMM, TM3 e *Semantics*. O trabalho de Combemale, Crégut e Pantel (2012) provê alguns *insights* para formalmente descrever modelos executáveis, apesar de não o fazer por não ser seu foco.

Considerando i-DSMLs, o foco deve ser colocado sobre a semântica operacional, pois uma semântica translacional requer a tradução para outra linguagem, por definição, e neste caso o modelo não é diretamente interpretado.

As próximas subseções apresentam um exemplo de metamodelo que define uma x-DSML para redes de Petri, e também o exemplo de um modelo específico que é instância desse metamodelo. Em seguida, propõe-se uma caracterização formal para semânticas de execução utilizando o SBMM como formalismo subjacente. Motores de execução também são caracterizados. Por fim, são apresentadas quinze definições sobre metamodelos e modelos executáveis, incluindo o conceito de modelos executáveis adaptativos, ainda utilizando o SBMM como base.

5.2.2 **Exemplo de Metamodelo para uma x-DSML de Redes de Petri**

Esta subseção apresenta a construção de um metamodelo para uma linguagem de modelagem executável de Redes de Petri. O padrão de projeto apresentado na subseção 5.2.1, embora tenha sido criado com base no MOF, é aqui aplicado com o SBMM.

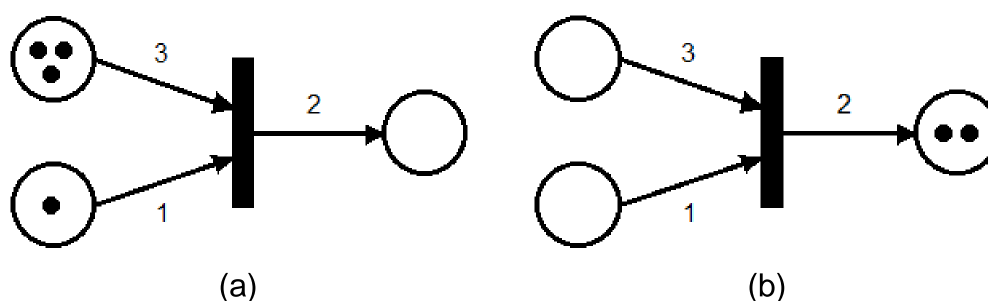
Redes de Petri se referem a uma linguagem de modelagem executável que, entre outras aplicações, podem descrever sistemas distribuídos discretos. Uma rede de Petri consiste em posições, transições e arcos direcionados. Arcos partem de uma posição para uma transição ou vice-versa, não ligam posições a posições ou transições a transições (BRETON; BÉZIVIN, 2001). As posições das quais algum arco parte para uma transição são denominadas posições de entrada da transição,

enquanto as posições para as quais arcos chegam de uma transição são chamadas de posições de saída da transição.

A semântica de execução de uma rede de Petri é descrita a seguir. Graficamente, posições em uma rede de Petri podem conter um número discreto de marcações denominadas *tokens*. Uma dada distribuição de *tokens* pelas posições da rede representa uma configuração. Uma transição de uma rede de Petri pode ser disparada se estiver habilitada, ou seja, se existirem *tokens* suficientes em todas as suas posições de entrada. Quando a transição é disparada, ou executada, ela consome os *tokens* requeridos de suas posições de entrada, e produz *tokens* nas posições de saída. Um disparo é uma operação atômica, isto é, um único passo que não pode ser interrompido.

O número requerido de *tokens* em cada posição de entrada é dado pelo peso do arco que parte para a transição. O número de *tokens* a serem produzidos em cada posição de saída é dado pelo peso do arco que parte da transição para a posição. A Figura 17 (a) apresenta um exemplo de rede de Petri utilizando sua notação gráfica usual (ou seja, sua sintaxe concreta). Posições são representadas por círculos, transições são representadas por retângulos preenchidos e setas representam arcos, que são rotulados por seus pesos. *Tokens* são representados por pontos dentro das posições. A mesma rede após o disparo da única transição está representada na Figura 17 (b).

Figura 17 – (a) Rede de Petri com uma transição habilitada; (b) A mesma rede após o disparo.

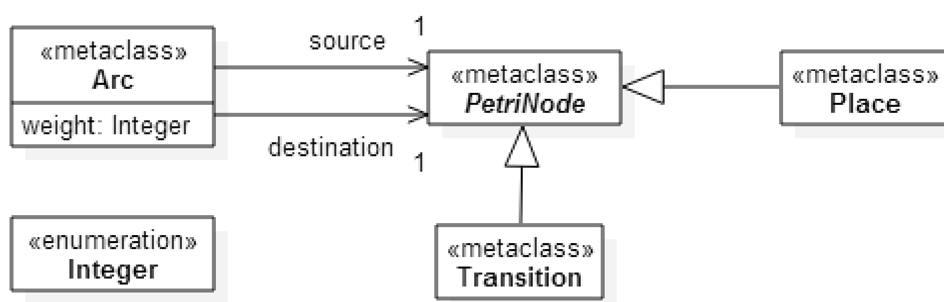


Fonte: autor

Ao menos que uma política de execução seja definida, a execução das redes de Petri é não determinística: quando múltiplas transições estão habilitadas ao mesmo tempo, qualquer uma delas pode ser disparada.

Considerando os conceitos aqui apresentados, a Figura 18 apresenta um metamodelo para redes de Petri.

Figura 18 – Metamodelo para Redes de Petri (DDMM)



Fonte: autor

A restrição de que um arco não pode ligar duas posições ou duas transições, juntamente com outras, estão representadas mais adiante através de expressões em lógica de primeira ordem.

Deve-se observar que um modelo conforme a esse metamodelo, isto é, uma instância de rede de Petri, não é autocontido porque não carrega informações que descrevem seu estado de execução. O metamodelo apresentado na Figura 18 é a parte DDMM do metamodelo executável de acordo com o padrão descrito por Combemale, Crégut e Pantel (2012) e apresentado na subseção 5.2.1. O mesmo metamodelo é representado em SBMM pela sétupla mostrada em (69).

$$Petri_{DDMM} = (\text{“PetriDDMM”}, C_{DDMM}, \Gamma_{DDMM}, E_{DDMM}, R_{DDMM}, descriptor_{c_DDMM}, descriptor_{e_DDMM}) \quad (69)$$

Em que:

- $C_{DDMM} = \{c_1, c_2, c_3, c_4\}$

- $c_1 =^{c_DDMM}$ ("Arc", P_1)
  - $P_1 = \{p_{11}, p_{12}, p_{13}\}$
    - p_{11} = ("weight", e_1, 1..1)
    - p_{12} = ("source", c_2, 1..1)
    - p_{13} = ("destination", c_2, 1..1)
- $c_2 =^{c_DDMM}$ ("PetriNode", $\varnothing$)
- $c_3 =^{c_DDMM}$ ("Place", $\varnothing$)
- $c_4 =^{c_DDMM}$ ("Transition", $\varnothing$)

- $\Gamma_{DDMM} = \{ (c_3,c_2), (c_4,c_2) \}$
- $E_{DDMM} = \{e_1\}$
  - $e_1 =^{e_DDMM}$ ("Integer", $\mathbb{Z}$)
- $R_{DDMM} = \{r_1, r_2, r_3, r_4\}$
  - r_1: $\forall$a Arc(a) $\Rightarrow$ (Transition(Arc.source(a)) $\Rightarrow$ Place(Arc.destination(a)))
  - r_2: $\forall$a Arc(a) $\Rightarrow$ (Place(Arc.source(a)) $\Rightarrow$ Transition(Arc.destination(a)))
  - r_3: $\neg\exists$x PetriNode*(x)
  - r_4: $\forall$a Arc(a) $\Rightarrow$ Arc.weight(a) > 0

A restrição r_1 estabelece que, para todos os elementos de modelo que forem instâncias de *Arc*, se *source* for uma transição então *destination* deve ser uma posição. A restrição r_2 estabelece a mesma regra para a situação oposta. O fato de *PetriNode* ser uma metaclasse abstrata é representado por r_3. Isso significa que uma instância direta de *PetriNode* não pode existir em um modelo conforme ao metamodelo. A restrição r_4 garante que os pesos dos arcos são números positivos.

Para adicionar informação relacionada a estado de execução nesse metamodelo, pode-se incluir a propriedade *numberOfTokens* na metaclasse *Place*, como mostrado na Figura 19, que corresponde à parte SDMM.

Figura 19 – Metaclasse *Place* com a propriedade de estado *numberOfTokens* (SDMM)

Fonte: autor

O metamodelo da Figura 19 é composto por uma única metaclasse e, em SBMM, é denotada por (70).

$$Petri_{SDMM} = (\text{“PetriSDMM”}, C_{SDMM}, \Gamma_{SDMM}, E_{SDMM}, R_{SDMM}, descriptor_{c_SDMM}, descriptor_{e_SDMM}) \quad (70)$$

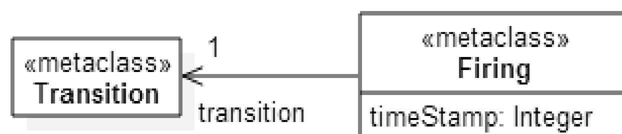
Em que:

- $C_{SDMM} = \{c_5\}$
  - $c_5 =^{c_SDMM} (\text{“Place”}, P_5)$
    - $P_5 = \{p_{51}\}$
      - $p_{51} = (\text{“numberOfTokens”}, e_2, 1..1)$
- $\Gamma_{SDMM} = \varnothing$
- $E_{SDMM} = \{e_2\}$
  - $e_2 =^{e_SDMM} (\text{“Integer”}, \mathbb{Z})$
- $R_{SDMM} = \{r_5\}$
  - r_5: $\forall p\ Place(p) \Rightarrow Place.numberOfTokens(p) \geq 0$

Para adicionar informações de evento, ou seja, a parte EDMM, cria-se a metaclasse *Firing*, que representa um evento de disparo. Ela é associada a uma transição e contém um carimbo de tempo (*timestamp*), modelado aqui como um número inteiro

representando o instante em que o evento ocorreu. Na verdade, Combemale, Crégut e Pantel (2012) propõem que uma metaclasse de evento deve ser filha de uma metaclasse mais geral chamada *RuntimeEvent* do pacote TM3. Entretanto, para os nossos propósitos aqui, apenas criar a metaclasse *Firing* sem herança já é suficiente. A Figura 20 mostra o conteúdo da parte EDMM.

Figura 20 – Metaclasse *Firing* (EDMM)



Fonte: autor

O metamodelo da Figura 27 é denotado em SBMM por (71).

$$Petri_{EDMM} = (\text{"PetriEDMM"}, C_{EDMM}, \Gamma_{EDMM}, E_{EDMM}, R_{EDMM}, descriptor_{c_EDMM}, descriptor_{e_EDMM}) \quad (71)$$

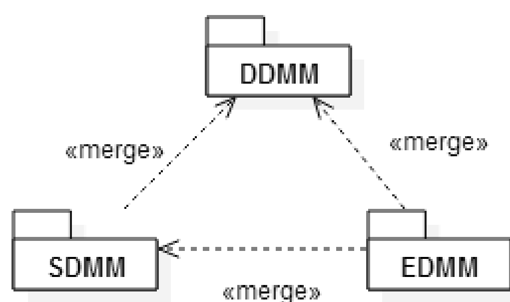
Em que:

- $C_{EDMM} = \{c_6, c_7\}$
    - $c_6 =^{c_EDMM} (\text{"Firing"}, P_6)$
        - $P_6 = \{p_{61}, p_{62}\}$
            - $p_{61} = (\text{"timeStamp"}, e_3, 1..1)$
            - $p_{62} = (\text{"transition"}, c_7, 1..1)$
    - $c_7 =^{c_EDMM} (\text{"Transition"}, \varnothing)$
- $\Gamma_{EDMM} = \varnothing$
- $E_{EDMM} = \{e_3\}$
    - $e_3 =^{e_EDMM} (\text{"Integer"}, \mathbb{Z})$

- $R_{EDMM} = \varnothing$

O metamodelo executável autocontido é então construído pela mesclagem de DDMM com SDMM e EDMM, operação representada pela Figura 21 em notação MOF.

Figura 21 – Construção do metamodelo executável autocontido



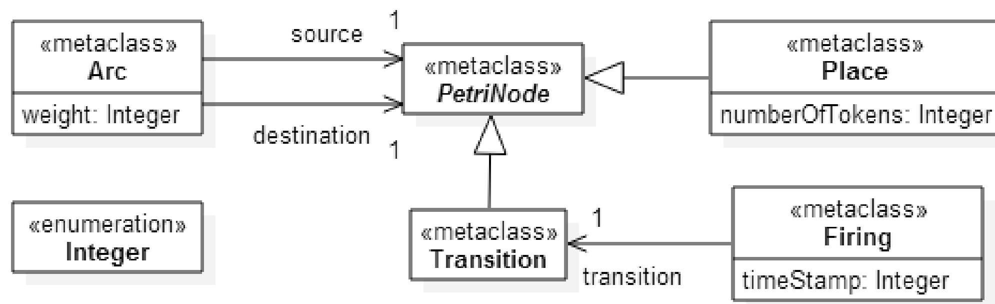
Fonte: Combemale, Crégut e Pantel (2012) (adaptado)

Como visto na subseção 4.10.2, o SBMM provê a função de mesclagem de metamodelos $@_{MM}$. Sendo assim, o metamodelo executável autocontido é expresso em SBMM por (72).

$$Petri_{MM} = @_{MM}(\text{"PetriMM"}, @_{MM}(\text{"TempPetriMM"}, Petri_{DDMM}, Petri_{SDMM}), Petri_{EDMM}) \quad (72)$$

Essa operação resulta no metamodelo autocontido final apresentado na Figura 22 (notação gráfica) e na eq. (73) (notação de conjuntos). Observe que as metaclasses e enumerações com o mesmo nome não são duplicadas no metamodelo resultante. Elas são mescladas conforme definido pela função $@_{MM}$.

Figura 22 – Metamodelo autocontido para uma x-DSML de Redes de Petri



Fonte: autor

$$Petri_{MM} = (\text{"PetriMetamodel"}, C_{MM}, \Gamma_{MM}, E_{MM}, R_{MM}, descriptor_{c_MM}, descriptor_{e_MM}) \quad (73)$$

Em que:

- $C_{MM} = \{c_1, c_2, c_3, c_4, c_6\}$
  - $c_1 =^{c_MM}$ ("Arc", P_1)
    - $P_1 = \{p_{11}, p_{12}, p_{13}\}$
      - p_{11} = ("weight", e_1, 1..1)
      - p_{12} = ("source", c_2, 1..1)
      - p_{13} = ("destination", c_2, 1..1)
  - $c_2 =^{c_MM}$ ("PetriNode", $\varnothing$)
  - $c_3 =^{c_MM}$ ("Place", P_3)
    - $P_3 = \{p_{31}\}$
      - p_{31} = ("numberOfTokens", e_1, 1..1)
  - $c_4 =^{c_MM}$ ("Transition", $\varnothing$)
  - $c_6 =^{c_MM}$ ("Firing", P_6)
    - $P_6 = \{p_{61}, p_{62}\}$
      - p_{61} = ("timeStamp", e_1, 1..1)

- p_{62} = (“transition”, c_4, 1..1)

- $\Gamma_{MM} = \{ (c_3,c_2), (c_4,c_2) \}$
- $E_{MM} = \{e_1\}$
  - $e_1 =^{e_MM}$ (“Integer”, $\mathbb{Z}$)
- $R_{MM} = \{r_1, r_2, r_3, r_4, r_5\}$
  - r_1: $\forall a\ Arc(a) \Rightarrow (Transition(Arc.source(a)) \rightarrow Place(Arc.destination(a)))$
  - r_2: $\forall a\ Arc(a) \Rightarrow (Place(Arc.source(a)) \rightarrow Transition(Arc.destination(a)))$
  - r_3: $\neg\exists x\ PetriNode^*(x)$
  - r_4: $\forall a\ Arc(a) \Rightarrow Arc.weight(a) > 0$
  - r_5: $\forall p\ Place(p) \Rightarrow Place.numberOfTokens(p) \geq 0$

O objeto p_{31} era originalmente p_{51}. Uma vez que as metaclasses c_3 e c_5 foram mescladas, c_3 ficou no metamodelo resultante e incorporou esta propriedade de c_5 (conforme definições da função $@_{MM}$). Sendo assim, as referências P_5 e p_{51} foram renomeadas para P_3 e p_{31}, respectivamente, para manter a convenção de indexação.

5.2.3 **Exemplo de Modelo de Rede de Petri**

A rede de Petri da Figura 17 (a) é descrita pela eq. (74).

$$Petri_M = (\text{“MyPetriNet01”}, \text{“PetriMetamodel”}, I_{PN01}, descriptor_{i_PN01}) \quad (74)$$

Em que:

- $I_{PN01} = \{i_1, i_2, i_3, i_4, i_5, i_6, i_7\}$
  - $i_1 =^{i_PN01}$ (“place01”, “Place”, { (“numberOfTokens”, “3”) })
  - $i_2 =^{i_PN01}$ (“place02”, “Place”, { (“numberOfTokens”, “1”) })
  - $i_3 =^{i_PN01}$ (“place03”, “Place”, { (“numberOfTokens”, “0”) })

- $i_4 =^{i_PN01}$ (“transition01”, “Transition”, ∅)
- $i_5 =^{i_PN01}$ (“arc01”, “Arc”, {(“weight”, “3”), (“source”, i_1), (“destination”, i_4)})
- $i_6 =^{i_PN01}$ (“arc02”, “Arc”, {(“weight”, “1”), (“source”, i_2), (“destination”, i_4)})
- $i_7 =^{i_PN01}$ (“arc03”, “Arc”, {(“weight”, “2”), (“source”, i_4), (“destination”, i_3)})

Note que o modelo $Petri_M$, cujo nome identificador é “MyPetriNet01”, respeita todas as restrições do metamodelo identificado por “PetriMetamodel”. Caso contrário, não seria um modelo em conformidade com o metamodelo.

5.2.4 **Semântica de Execução**

O comportamento executável de modelos conformes a um metamodelo executável é definido por uma semântica de execução. Por exemplo, a subseção 5.2.2 explicou a semântica de execução de redes de Petri em português. Combemale, Crégut e Pantel (2012) propõem que a semântica seja definida por uma tripla (SDMM, EDMM, Behavior), em que SDMM e EDMM foram exemplificados na subseção 5.2.2 e *Behavior* abstrai o comportamento executável do modelo, isto é, como ele evolui após receber estímulos. Em uma semântica translacional, *Behavior* é baseado no mapeamento do metamodelo para outra linguagem executável enquanto em uma semântica operacional *Behavior* descreve esta evolução no mesmo domínio. Por isso, ela é especialmente importante para interpretação de modelos. Na verdade, Combemale, Crégut e Pantel (2012) usam o nome *Semantics* em vez de *Behavior*, mas optou-se aqui pela troca para evitar ambiguidades entre a semântica de execução, que é a tripla, e a abstração *Semantics*, que é o terceiro elemento da tripla.

SDMM e EDMM são pacotes do padrão de metamodelagem, mas pode haver metamodelos executáveis não construídos com base nesse padrão. Dessa forma, esses pacotes podem não estar explicitamente definidos. Sendo assim, propõe-se aqui a definição de uma semântica para o metamodelo MM = ($name_{MM}$, C, Γ, E, R, $descriptor_c$, $descriptor_e$) como uma tripla de acordo com (75).

$$\text{Semantics}_{MM} = (\text{state_props}_{MM}, \text{event_class}_{MM}, b_{MM}) \quad (75)$$

Em que:

- state_props$_{MM}$: $C \rightarrow 2^{\{P_1 \cup P_2 \cup \ldots \cup P_n\}}$ é uma função que identifica as propriedades que descrevem o estado de execução do modelo de acordo com Semantics$_{MM}$. Ela toma uma metaclasse como argumento e a mapeia em seu subconjunto de propriedades relacionadas à descrição de estado. No exemplo da subseção 5.2.2, tem-se state_props$_{PetriMM}(c_3) = \{p_{31}\}$ e state_props$_{PetriMM}(c) = \emptyset$ para todo $c \in C \setminus \{c_3\}$. A tupla p_{31} refere-se à propriedade *numberOfTokens*, que é a única relacionada à descrição de estado de execução;
- event_class$_{MM} \subseteq C$ é o subconjunto de metaclasses que representam eventos. Considerando o exemplo da subseção 5.2.2, tem-se event_class$_{MM}$ = $\{c_6\}$. O objeto c_6 é a metaclasse *Firing*, que é a única que representa um evento. Este subconjunto deve respeitar a transitividade de Γ^+, ou seja, se x é submetaclasse de y, direta ou transitivamente, e se y é uma metaclasse de evento, então x também é uma metaclasse de evento;
- b_{MM} é chamada de função de comportamento da semântica de execução Semantics$_{MM}$. Pode ser uma função de mapeamento (semântica translacional) ou uma função de evolução (semântica operacional). Funções de mapeamento foram descritas, dentro do contexto do SBMM, na seção 4.12. Neste ponto, nosso objetivo é descrever uma semântica operacional e, portanto, nesta subseção b_{MM} sempre irá se referir a uma função de evolução. Nesse caso, b_{MM} toma dois argumentos: um modelo conforme a MM e uma instância de evento concreto, resultando então em um modelo atualizado conforme a MM, ou seja, $b_{MM}: \mathcal{M}_{MM} \times \mathcal{I}_{MM}(\text{event_class}_{MM}) \rightarrow \mathcal{M}_{MM}$.

Diz-se que um modelo executável M_k evolui para outro modelo M_{k+1} após receber um estímulo, o que caracteriza um passo de execução de acordo com a semântica considerada tal que $M_{k+1} = b_{MM}(M_k, ev)$, onde $ev \in \mathcal{I}_{MM}(\text{event_class}_{MM})$ é o estímulo

representado por uma instância de metaclasse de evento (evento concreto). Denota-se este passo de execução por $M_k \Rightarrow^{ev} M_{k+1}$.

Mesmo se descrito por passos intermediários auxiliares, como é apresentado mais adiante na subseção 5.2.5, um passo de execução é uma operação atômica. M_{k+1} pode ser visto como uma versão editada de M_k, e as alterações correspondentes a esta edição são computadas por b_{MM}. Essa função pode ser considerada uma abstração genérica a ser implementada, tal como *Behavior* (*Semantics*), apresentada em Combemale, Crégut e Pantel (2012). Uma das formas de implementá-la é através de um algoritmo.

Deve-se notar que descrever uma função de comportamento para um metamodelo específico não é necessariamente uma tarefa difícil. Por exemplo, considera-se o modelo de rede de Petri apresentado na subseção 5.2.3. O Quadro 6 poderia ser usado para descrever os efeitos da aplicação de b_{MM} para aquele modelo. Cada célula denota o número de *tokens* a ser adicionado em cada posição quando uma transição é disparada, de acordo com os pesos dos arcos.

Quadro 6 – Função de comportamento de uma rede de Petri específica

	transition01
place01	-3
place02	-1
place03	+2

Fonte: autor

Quando *transition01* é disparada (evento representado por uma instância de *Firing* cujo *slot* para a propriedade *transition* aponta para a instância *transition01*), o número de *tokens* de cada posição deve ser editado de acordo com o Quadro 6 para gerar uma nova versão do modelo correspondente ao resultado do passo de execução. Esta técnica para representar b_{MM} pode ser usada para qualquer rede de Petri através da adaptação do número de linhas e colunas da tabela. Trata-se, portanto, de uma técnica de especificação de b_{MM} específica para este metamodelo (mesmo que sirva para outros). Na verdade, matrizes são de fato utilizadas para representar redes de Petri.

Propõe-se aqui a utilização das funções de edição e consulta de modelos apresentadas nas seções 4.13 e 4.14 como biblioteca para especificar b_{MM} para

qualquer metamodelo executável na forma de algoritmos. A especificação de uma função de comportamento para a semântica do metamodelo de redes de Petri é apresentada no Quadro 7. O algoritmo não é baseado em nenhuma implementação de orientação a objetos subjacente. Seus tipos básicos são apenas conjuntos, tuplas, *strings* e inteiros, de acordo com a base do SBMM. A conversão de *strings* para inteiros e vice-versa são abstraídas respectivamente pelas funções *StrToInt* e *IntToStr*. Elementos de uma tupla são indexados por colchetes. Por exemplo, o primeiro elemento da tupla *myInstanceDescriptor* representando um descritor de instância de modelo é denotado por *myInstanceDescriptor*[1], que é o nome da instância de acordo com a eq. (16).

Quadro 7 – Algoritmo que especifica a função de comportamento da semântica de redes de Petri

```
1  Função b_PetriMM (inputModel: Tupla, eventInstance: Tupla)
2  Saída outputModel: Tupla;
3  // Considera-se as seguintes expansões das variáveis de entrada,
4  // cujos termos são acessíveis por este algoritmo.
5  // inputModel = (nameM, ‘PetriNetMetamodel’, I, descriptor_i)
6  // eventInstance = (nameEI, ‘Firing’, S_EI)
7  var
8    I’: Conjunto;
9    descriptor_i’: Conjunto; // Funções também são conjuntos.
10   firedTransition, a: Instância;
11   descr_a: Tupla;
12   inputArcs, outputArcs: Conjunto;
13   inputPlace, outputPlace: Instância;
14   descr_inputPlace, descr_outputPlace: Tupla;
15   n,w: String;
16 begin
17   // Inicializa o modelo de saída.
18   I’ := I;   // I’ é inicializado com uma cópia de I.
19   descriptor_i’ := descriptor_i; // Idem para descriptor_i’.
20   outputModel := (nameM, ‘PetriNetMetamodel’, I’, descriptor_i’);
21   // Como há uma única metaclasse de evento no metamodelo,
22   // não precisamos testar que tipo de evento ocorreu.
23   firedTransition := query_prop_value(inputModel,
24                                       nameEI, ‘transition’);
```

```
25    // Obtém conjunto dos arcos de entrada de firedTransition
26    inputArcs := query_instance_by_prop(inputModel, 'Arc',
27                              'destination', firedTransition);
28    // Varre os arcos de entrada atualizando o número de tokens
29    // de cada posição conforme os pesos.
30    for each a in inputArcs {
31      descr_a := descriptor_i(a); // descr_a = (nameI,nameC,S)
32      w := query_prop_value(inputModel, descr_a[1], 'weight');
33      inputPlace := query_prop_value(inputModel, descr_a[1],
34                                     'source');
35      descr_inputPlace := descriptor_i(inputPlace);
36      n := query_prop_value(inputModel, descr_inputPlace[1],
37                            'numberOfTokens');
38      insert_or_edit_slot(outputModel, descr_a[1],
39                          'numberOfTokens',
40                          IntToStr(StrToInt(n) - StrToInt(w)));
41    }
42    // Obtém o conjunto dos arcos de saída de firedTransition
43    outputArcs := query_instance_by_prop(inputModel, 'Arc',
44                                     'source', firedTransition);
45    // Varre os arcos de saída atualizando o número de tokens
46    // de cada posição conforme os pesos.
47    for each a in outputArcs {
48      descr_a := descriptor_i(a); // descr_a = (nameI,nameC,S)
49      w := query_prop_value(inputModel, descr_a[1], 'weight');
50      outputPlace := query_prop_value(inputModel, descr_a[1],
51                                      'destination');
52      descr_outputPlace := descriptor_i(outputPlace);
53      n := query_prop_value(inputModel, descr_outputPlace[1],
54                            'numberOfTokens');
55      insert_or_edit_slot(outputModel, descr_a[1],
56                          'numberOfTokens',
57                          IntToStr(StrToInt(n) + StrToInt(w)));
58    }
59  end;
```

Fonte: autor

5.2.5 **Motores de Execução**

O núcleo de um motor de execução de uma i-DSML definida por um metamodelo executável é um algoritmo que toma três entradas: um metamodelo, uma semântica de execução e um modelo em seu estado inicial. A saída deste algoritmo é o modelo em seu estado final. O motor de execução interage com o ambiente e instancia eventos concretos (p. ex., representando eventos de comunicação, de leitura e escrita, de relógio, de interações com usuário, etc.) e os insere no modelo. Sendo assim, a computação relacionada ao passo de execução $M_k \Rightarrow^{ev} M_{k+1}$ é realizada pelo motor de acordo com os seguintes subpassos:

1. ev := generate_instance_object();
2. $name_{ev}$:= generate_event_name();
3. M_k' := insert_empty_instance(M_k, ev, $name_{ev}$, $name_{ev_metaclass}$); P. ex.,
    i. M_k' := insert_empty_instance(M_k, ev, $name_{ev}$, “Firing”);
4. Preencher os valores de propriedades do evento concreto; P. ex.,
    i. M_k' := insert_or_edit_slot(M_k', $name_{ev}$, “timeStamp”, ‘1’);
    ii. M_k' := insert_or_edit_slot(M_k', $name_{ev}$, “transition”, i_4);
5. M_{k+1} := b_{MM}(M_k', ev);

No subpasso 1, a sub-rotina *generate_instance_object* abstrai a criação de um novo objeto ev para ser adicionado ao conjunto de instâncias do modelo. No subpasso 2, a função *generate_event_name* abstrai a geração do nome identificador da nova instância de evento concreto. Poderia ser, por exemplo, a *string* “ev” concatenada com um número sequencial.

Como observado anteriormente, o passo de execução é atômico, mesmo que sua descrição tenha sido apresentada na forma de subpassos e que a função b_{MM} tenha sido especificada por um algoritmo com vários passos internos.

Um desenvolvedor de x-DSML possui muitos graus de liberdade para criar metaclasses de evento em metamodelos executáveis. O interpretador de execução deve conhecer o significado pretendido de cada metaclasse de evento para corretamente instanciá-las e preencher os valores de suas propriedades. Por

exemplo, uma metaclasse de evento de nome *KeyboardEvent* poderia representar a interação de usuário correspondente ao pressionamento de uma tecla. Quando isso acontece, um evento concreto deve ser instanciado e o *slot* correspondente à propriedade *key* deve ser preenchido corretamente. As definições de metamodelo e semântica de execução aqui apresentadas não fornecem elementos suficientes para representar o significado de cada metaclasse. Assume-se que o motor de execução possui este conhecimento de algum modo. Uma opção é fazer uma implementação *hard coded*, mas o motor de execução perde generalidade. Especificar o significado pretendido para cada metaclasse de evento está além do escopo deste trabalho. Entretanto, prover uma biblioteca de metaclasses de evento conhecidas cujos significados são implementados no motor de execução é um caminho para superar este problema.

Analogamente, ações específicas de domínio não são explicitamente suportadas pelas definições até aqui. Uma ação específica de domínio representa um procedimento ou sub-rotina que deve ser chamada pelo motor quando o modelo em execução atinge determinada configuração. Por exemplo, quando certo estado em uma máquina de estados UML se torna ativo, uma requisição HTTP deve ser realizada.

5.2.6 **Considerações sobre Adaptação de Modelos em Tempo de Execução**

Na MDE, existe uma área de pesquisa denominada *Models at Runtime*[4], a qual estuda a utilização de modelos de software em tempo de execução. Uma de suas utilidades é prover uma base para tomada de decisão em tempo de execução relacionada ao próprio sistema sendo executado, que pode ser autogerenciado, monitorado ou modificado conforme estímulos recebidos do ambiente e, portanto, apresentar capacidades *self-**[5] (BENCOMO et al., 2014).

De acordo com a revisão sistemática da literatura sobre *Models at Runtime* apresentada por Szvetits e Zdun (2013), a autoadaptação de modelos em tempo de execução é um dos principais objetivos da área. Um dos exemplos apresentados é a

---

[4] Essa área de pesquisa também é frequentemente referenciada na literatura pela expressão *Models@run.time* e, por essa razão, optou-se por não traduzir.

[5] O termo *self-**, nesse contexto, designa de forma geral as capacidades que o sistema em execução tem sobre si mesmo (p. ex., auto-otimização, automoficação e autogerenciamento).

autoadaptação em tempo de execução de interfaces de usuário. Considere que um sistema mantenha em tempo de execução alguma representação interna (modelo) de sua interface. De acordo com sua utilização por parte do usuário (estímulos externos), em tempo de execução, essa representação pode sofrer alterações conforme regras predefinidas que refletem uma mudança na própria interface apresentada ao usuário, tal como mover campos ou botões pouco utilizados para uma aba separada, deixando predominantemente visíveis apenas aqueles elementos que o usuário acessa mais frequentemente. Observe que esse tipo de modelo não é necessariamente um modelo executável tal como caracterizado nas subseções anteriores, e ainda assim pode-se falar na adaptação de modelos. Uma arquitetura de execução adequada, com uma camada de controle, pode operar sobre os modelos passíveis de serem adaptados em tempo de execução.

A execução direta de modelos é apresentada na revisão sistemática de Szvetits e Zdun (2013) como uma técnica de *Models at Runtime*. Diz-se que os modelos executáveis possuem uma relação causal com o sistema de software em execução, e que mudanças no modelo sendo interpretado implicam diretamente em alterações no comportamento do sistema em execução.

A seguinte subseção propõe uma formalização dos conceitos de modelos executáveis e modelos executáveis adaptativos, baseada no SBMM. Uma formalização desse tipo não foi encontrada na revisão sistemática de Szvetits e Zdun (2013).

5.2.7 **Formalização de Conceitos Utilizando SBMM**

Nesta subseção são apresentadas definições que precisamente caracterizam metamodelos e modelos executáveis, incluindo adaptativos, assim como conceitos relacionados, tendo o SBMM como formalismo subjacente.

Basicamente, a ideia por trás dos conceitos definidos a seguir é que um modelo executável é não-adaptativo quando sua parte estrutural nunca pode ser alterada durante sua execução. Caso contrário, ou seja, quando há a possibilidade de alteração na parte estrutural (p. ex., criação de novos estados e transições em uma máquina de estados), então ele é adaptativo. Nos dois casos há a possibilidade de mudança de estado no passo de execução, o que não caracteriza uma alteração

estrutural (p. ex., quando o estado corrente de uma máquina de estados muda devido a uma transição). Dessa forma, é necessário estabelecer o que exatamente corresponde à parte estrutural de um modelo e o que exatamente corresponde a descrição de estado. Isso decorre da semântica de execução considerada, como será apresentado a seguir ao longo das definições.

Seja um metamodelo arbitrário MM = ($name_{MM}$, C, Γ, E, R, $descriptor_c$, $descriptor_e$) e $Semantics_{MM}$ = ($state_props_{MM}$, $event_class_{MM}$, b_{MM}) uma semântica de execução para MM. Seja C = {c_1, c_2, ... c_n} com $descriptor_c(c_k)$ = ($metaclassname_k$, P_k) para k = 1, 2, ..., n. O metamodelo $Petri_{MM}$ de nome "PetriMetamodel" da eq. (73) e o modelo $Petri_M$ de nome "MyPetriNet01" da eq. (74) são utilizados como exemplos ao longo das definições. Para esse metamodelo, considera-se a semântica $Semantics_{PetriMM}$ = ($state_props_{PetriMM}$, $event_class_{PetriMM}$, $b_{PetriMM}$), em que:

- $state_props_{PetriMM}(c_1) = \emptyset$;
- $state_props_{PetriMM}(c_2) = \emptyset$;
- $state_props_{PetriMM}(c_3)$ = { ("numberOfTokens", e_1, 1..1) };
- $state_props_{PetriMM}(c_4) = \emptyset$;
- $state_props_{PetriMM}(c_6) = \emptyset$;
- $event_class_{PetriMM} = \{c_6\}$;
- $b_{PetriMM}$: especificado no Quadro 7.

Alguns dos conceitos apresentados abaixo já foram mencionados. Entretanto, suas definições são reescritas de forma mais precisa através de fórmulas e sentenças em lógica de primeira ordem.

Definição 1. Propriedades de Estado. $p_{kj} \in P_k$ é uma propriedade de estado com relação à metaclasse c_k de acordo com a semântica $Semantics_{MM}$ se e somente se $p_{kj} \in state_props_{MM}(c_k)$. Uma propriedade de estado tem o objetivo de descrever, parcial ou totalmente, o estado de execução dos modelos executáveis conformes a MM. A função auxiliar $state_props_all_{MM}(c)$ pode ser obtida a partir de $state_props_{MM}$.

Ela mapeia a metaclasse c em suas propriedades de estado e também naquelas de suas supermetaclasses, transitivamente, de acordo com (76).

$$\mathrm{state_props_all}_{\mathrm{MM}}(\mathrm{c}) = \mathrm{state_props}_{\mathrm{MM}}(\mathrm{c}) \cup \bigcup_{(\mathrm{c},\mathrm{x})\in\Gamma^{+}} \mathrm{state_props}_{\mathrm{MM}}(\mathrm{x}) \quad (76)$$

Não se permite propriedades de estado em metaclasses de evento (ver Definição 5 adiante).

Exemplo da Definição 1. Em $Petri_{MM}$, p_{31} = (“numberOfTokens”, e_1, 1..1) é a única propriedade de estado, e ocorre com relação a c_3.

Definição 2. *Slots* de Estado. Seja M = ($name_M$, $name_{MM}$, **I**, $descriptor_I$) um modelo arbitrário conforme a MM. Seja I = {i_1, i_2, ... i_q} com $descriptor_I(i_k)$ = ($instancename_k$, $metaclassname_k$, S_k) para k = 1, 2, ..., q. O *slot* s_{kj} = ($propertyname_{kj}$, val_{kj}) $\in S_k$ é dito ser um *slot* de estado de acordo com a semântica $Semantics_{MM}$ se e somente se $\exists$ (w, t, m) $\in$ $state_props_all_{MM}$(metaclass(MM, $metaclassname_k$)) (w = $propertyname_{kj}$). Em outras palavras, s_{kj} é um *slot* de estado se e somente se ele corresponder a uma propriedade de estado da metaclasse identificada por $propertyname_k$ ou qualquer uma de suas supermetaclasses em MM. O conjunto de todos os *slots* de estado de uma instância i_k é denotado por $state_slots_{MM}(i_k)$.

Exemplo da Definição 2. Em $Petri_M$, tem-se $state_slots_{PetriM}(i_1)$ = {(“numberOfTokens”, “3”)}, $state_slots_{PetriM}(i_2)$ = {(“numberOfTokens”, “1”)}, $state_slots_{PetriM}(i_3)$ = {(“numberOfTokens”, “0”)} e $state_slots_{PetriM}(i_k) = \emptyset$ para todos os outros k.

Definição 3. Metamodelos Executáveis por Estado. MM é dito ser executável por estado de acordo com a semântica $Semantics_{MM}$ se e somente se $state_props_{MM}(c_1) \cup \ldots \cup state_props_{MM}(c_n) \neq \emptyset$. Em outras palavras, é necessário haver pelo menos uma propriedade de estado em qualquer metaclasse.

Exemplo da Definição 3. De acordo com a definição, $Petri_{MM}$ é um metamodelo executável por estado.

Definição 4. Estado de Execução de um Modelo. Seja M = ($name_M$, $name_{MM}$, I, $descriptor_I$) um modelo arbitrário conforme a MM. Seja I = {i_1, i_2, ... i_q} com $descriptor_I(i_k)$ = ($instancename_k$, $metaclassname_k$, S_k) para k = 1, 2, ..., q. O estado de execução de M de acordo com a semântica $Semantics_{MM}$ é o conjunto de todos os *slots* de estado de todas as instâncias combinados com os nomes das instâncias em um par ordenado, excluindo-se aquelas instâncias que não tem *slots* de estado. A eq. (77) expressa esse conceito através da função ExecState. O estado de execução descreve um *snapshot* do modelo em execução. Uma vez que os *slots* não contêm informação sobre suas instâncias proprietárias em si mesmos, os nomes das instâncias foram incluídos nos pares para identificar completamente os “carregadores” de cada informação de estado.

$$\mathrm{ExecState(M)} = \bigcup_{(\mathrm{i_k} \in \mathrm{I}) \wedge (\mathrm{state_slots_{MM}(i_k)} \neq \emptyset)} \{(\mathrm{instancename_k}, \mathrm{state_slots_{MM}(i_k)})\} \quad (77)$$

Exemplo da Definição 4. ExecState($Petri_M$) = { (“place01”, {(“numberOfTokens”, “3”)}), (“place02”, {(“numberOfTokens”, ”1”)}), (“place03”, {(“numberOfTokens”, “0”)}) }. O resultado da função descreve completamente o estado de execução de $Petri_M$.

Definição 5. Metaclasses de Evento. $c \in C$ é dita ser uma metaclasse de evento de acordo com a semântica $Semantics_{MM}$ se e somente se $c \in event_metaclass_{MM}$. Observe que se x é uma submetaclasse de c, isto é, $(x,c) \in \Gamma^+$, então x também pertence a $event_class_{MM}$. Caso contrário, $event_class_{MM}$ não é considerado um conjunto válido para compor $Semantics_{MM}$. Como outra restrição, metaclasses de evento não podem conter propriedades de estado, isto é, $\forall c \in C$ ($c \in event_class_{MM}$ $\Rightarrow state_props_all_{MM}(c) = \emptyset$).

Exemplo da Definição 5. Em $Petri_{MM}$, c_6 (*Firing*) é a única metaclasse de evento.

Definição 6. Eventos Concretos. Seja M = ($name_M$, $name_{MM}$, I, $descriptor_I$) um modelo arbitrário conforme a MM. ev $\in$ I é dito ser um evento concreto de acordo com a semântica $Semantics_{MM}$ se e somente se pelo menos um de seus tipos é um elemento de $event_class_{MM}$, ou seja, $\exists$ t $\in$ types(MM,M,ev) (t $\in$ $event_class_{MM}$).

Exemplo da Definição 6. $Petri_M$ não possui nenhum evento concreto tal como apresentado na subseção 5.2.3. Entretanto, se o motor de execução instanciar ev tal que $descriptor_I$(ev) = (“firing01”, “Firing”, {(“timeStamp”, “1”), (“transition”, i_4)}), então ev é um exemplo de evento concreto pois *Firing* é seu tipo.

Definição 7. Propriedades Estruturais. $p_{kj} \in P_k$ é uma propriedade estrutural com relação à metaclasse c_k de acordo com a semântica $Semantics_{MM}$ se e somente se ($p_{kj} \notin state_props_{MM}(c_k)$) $\wedge$ ($c_k \notin event_class_{MM}$). Em outras palavras, propriedades estruturais são aquelas que não são propriedades de estado e nem pertencem a uma metaclasse de evento. Também se define a função auxiliar $strucutral_props_{MM}(c)$ que mapeia o conjunto de propriedades estruturais da metaclasse c. Esta função não é uma nova parte de $Semantics_{MM}$ porque pode ser obtida a partir de $state_props_{MM}$ e $event_class_{MM}$. Outra função $strucutral_props_all_{MM}(c)$ é definida como um mapeamento da metaclasse c para suas propriedades estruturais considerando também aquelas de suas supermetaclasses, transitivamente. Sua definição é análoga à eq. (76).

Exemplo da Definição 7. Em $Petri_{MM}$, $structural_props_{MM}(c_1)$ = {(“weight”, e_1, 1..1), (“source”, c_2, 1..1), (“destination”, c_2, 1..1)}, $strucutral_props_{MM}(c_2) = \emptyset$, $strucutral_props_{MM}(c_3) = \emptyset$, $structural_props_{MM}(c_4)$ e $structural_props(c_6) = \emptyset$.

Definição 8. *Slots* Estruturais. Seja M = ($name_M$, $name_{MM}$, I, $descriptor_I$) um modelo arbitrário conforme a MM. Seja I = {i_1, i_2, ... i_q} com $descriptor_I(i_k)$ = ($instancename_k$,

$metaclassname_k$, S_k) para k = 1, 2, ..., q. $s_{kj} = (propertyname_{kj}, val_{kj}) \in S_k$ é dito ser um *slot* estrutural de acordo com $Semantics_{MM}$ se e somente se $\exists$ (w, t, m) $\in$ $structural_props_all_{MM}(metaclass(MM, metaclassname_k))$ (w = $propertyname_{kj}$). Em outras palavras, s_{kj} é um *slot* estrutural se e somente se corresponder a uma propriedade estrutural da metaclasse identificada por $metaclassname_k$ em MM, ou de uma de suas supermetaclasses. O conjunto de todos os *slots* estruturais da instância i_k é denotado por $structural_slots_{MM}(i_k)$.

Exemplo da Definição 8. Em $Petri_M$, tem-se $structural_slots_{PetriM}(i_5)$ = {("weight", "3"), ("source", i_1), ("destination", i_4)}, $strucutral_slots_{PetriM}(i_6)$ = {("weight", "1"), ("source", i_2), ("destination", i_4)} e $structural_slots_{PetriM}(i_7)$ = {("weight", "2"), ("source", i_4), ("destination", i_3)}. $structural_slots_{PetriM}(i_k) = \emptyset$ para todos os outros k.

Definição 9. Parte Estrutural do Modelo. Seja M = ($name_M$, $name_{MM}$, I, $descriptor_I$) um modelo arbitrário conforme a MM. Seja I = {i_1, i_2, ... i_q} com $descriptor_I(i_k)$ = ($instancename_k$, $metaclassname_k$, S_k) para k = 1, 2, ..., q. A parte estrutural de M de acordo com a semântica $Semantics_{MM}$ é o conjunto de *slots* estruturais de todas as instâncias combinados com os nomes das instâncias em pares ordenados de acordo com (78). Essa definição é análoga à definição de estado de execução de um modelo. A parte estrutural está relacionada aos elementos do modelo que permanecem inalterados durante a execução não-adaptativa do modelo (ver Definição 10 mais adiante). Uma vez que os *slots* não contêm informação sobre suas instâncias proprietárias em si mesmos, os nomes das instâncias foram incluídos nos pares para identificar completamente os "carregadores" de cada informação estrutural.

$$\mathrm{StructuralPart}(M) = \bigcup_{\substack{(i_k \in I) \wedge \\ (metaclass(MM, metaclassname_k) \notin event_class_{MM})}} \{(instancename_k, structural_slots_{MM}(i_k))\} \quad (78)$$

Diferentemente da definição de ExecState, em StructuralPart até mesmo as instâncias que não possuem *slots* estruturais são incluídas na definição. Isso acontece porque se considera que a existência de uma instância, mesmo que só tenha propriedades de estado, é parte estrutural do modelo. Por exemplo, a metaclasse *Place* possui exclusivamente uma propriedade de estado, mas a existência de cada instância de *Place* em uma rede de Petri compõe sua parte estrutural. Por outro lado, instâncias de metaclasses de evento são excluídas da definição, pois apenas representam ocorrências reais de eventos em tempo de execução, não tendo relação com a parte estrutural do modelo.

Exemplo da Definição 9. StructuralPart($Petri_M$) = {(“arc01”, {(“weight”, “3”), (“source”, i_1), (“destination”, i_4)}), (“arc02”, {(“weight”, “1”), (“source”, i_2), (“destination”, i_4)}), (“arc03”, {(“weight”, “2”), (“source”, i_4), (“destination”, i_3)}), (“transition01”, ∅), (“place01”, ∅), (“place02”, ∅), (“place03”, ∅)}. Ela descreve completamente a parte estrutural do modelo.

Definição 10. Metamodelos Executáveis Não-Adaptativos. MM é um metamodelo executável não-adaptativo de acordo com a semântica $Semantics_{MM}$ se e somente se ele é executável por estado (Definição 3) e o modelo de saída computado por b_{MM} tem parte estrutural (Definição 9) igual àquela do modelo de entrada, para qualquer modelo e evento concreto de entrada. Essa condição é representada em (79). Seja M = ($name_M$, $name_{MM}$, I, $descriptor_I$) um modelo arbitrário conforme a MM em qualquer passo de execução.

$$\forall M \in \mathcal{M}(MM)\ (\ \forall ev \in \mathcal{I}_{MM}(event_class_{MM})\ (\ StructuralPart(\ b_{MM}(M, ev)\ ) = StructuralPart(M)\ )\ ) \tag{79}$$

Exemplo da Definição 10. $Petri_{MM}$ é um metamodelo executável não-adaptativo de acordo com $Semantics_{PetriMM}$ porque $b_{PetriMM}$, como especificado no Quadro 7, nunca altera a parte estrutural do modelo de entrada. A função é capaz apenas de alterar *slots* relacionados à propriedade *numberOfTokens*, que é uma propriedade de

estado. Além disso, esta função não insere novas instâncias de qualquer metaclasse no modelo.

Definição 11. Modelos Executáveis Não-Adaptativos. M é um modelo executável não-adaptativo de acordo com a semântica $Semantics_{MM}$ se e somente se seu metamodelo MM é executável não-adaptativo (Definição 10) de acordo com a mesma semântica e se M é conforme a MM.

Exemplo da Definição 11. $Petri_M$ é um modelo executável não-adaptativo de acordo com $Semantics_{PetriMM}$ porque ele é conforme a $Petri_{MM}$, que é um metamodelo executável não-adaptativo.

Neste ponto, a definição de metamodelos e modelos executáveis adaptativos aparecem quase que naturalmente com base nos conceitos apresentados acima.

Definição 12. Metamodelos Executáveis Adaptativos. MM é um metamodelo executável adaptativo de acordo com a semântica $Semantics_{MM}$ se e somente se o modelo de saída computado por b_{MM} tem parte estrutural (Definição 9) diferente da do modelo de entrada para pelo menos um modelo de entrada e evento concreto possíveis. Esta situação está representada por (80). Seja M = ($name_M$, $name_{MM}$, I, $descriptor_I$) um modelo arbitrário conforme a MM em qualquer passo de execução.

$$\exists M \in \mathcal{M}(MM) \; (\exists ev \in \mathcal{I}_{MM}(event_class_{MM}) \; ((StructuralPart(b_{MM} (M, ev)) \neq StructuralPart(M)))) \quad (80)$$

Definição 13. Modelos Executáveis Adaptativos. M é um modelo executável adaptativo de acordo com a semântica $Semantics_{MM}$ se e somente se seu metamodelo MM é executável adaptativo (Definição 12) de acordo com a mesma semântica e se M é conforme a MM.

Exemplo da Definição 13. $Petri_{M}$ não é um modelo adaptativo de acordo com $Semantics_{PetriMM}$ porque ele é conforme a $Petri_{MM}$, que não é um metamodelo executável adaptativo.

Um metamodelo executável não-adaptativo não possibilita alterações estruturais em nenhum caso, enquanto o metamodelo executável adaptativo possibilita, mesmo que tais alterações não ocorram em alguma execução particular de um modelo conforme.

Definição 14. Metamodelos Executáveis. MM é um metamodelo executável de acordo com a semântica $Semantics_{MM}$ se e somente se ele for executável não-adaptativo (Definição 10) ou executável adaptativo (Definição 12) de acordo com a mesma semântica. Alternativamente, pode-se dizer que a DSML especificada por MM é executável.

Exemplo da Definição 14. $Petri_{MM}$ é um metamodelo executável de acordo com a semântica $Semantics_{PetriMM}$ porque ele é executável não-adaptativo.

Observa-se, pelas definições, que um metamodelo pode não apresentar propriedades de estado e, portanto, seus modelos não são executáveis por estado. Mas, ainda assim, ele pode ser executável exclusivamente por adaptatividade. Ou seja, em todo e qualquer passo de execução não ocorrem mudanças de estado, mas somente alterações estruturais.

Definição 15. Modelos Executáveis. M é um modelo executável de acordo com a semântica $Semantics_{MM}$ se e somente se seu metamodelo MM é executável de acordo com a mesma semântica e M é conforme a MM.

Exemplo da Definição 15. $Petri_M$ é um modelo executável de acordo com a semântica $Semantics_{PetriMM}$ porque ele é conforme a $Petri_{MM}$, que é um metamodelo executável.

5.3 DEFINIÇÕES E PROPRIEDADES SOBRE SINCRONIZAÇÃO DE MODELOS

Em um estudo sobre sincronização de modelos, Xiong et al. (2007) apresentam definições e propriedades sobre esse tipo de operação. No entanto, por não dispor de um formalismo subjacente, os autores introduzem seus próprios termos para em seguida estabelecer essas definições e propriedades. Esta seção apresenta, como outra aplicação teórica do SBMM, as definições e propriedades sobre sincronização de modelos de Xiong et al. (2007) com base no SBMM.

Conceitos básicos sobre sincronização de modelos foram apresentados na subseção 2.3.1.

Definição 16. Função de Sincronização. Seja MM_{origem} um metamodelo que define uma linguagem de modelagem de origem. Seja MM_{alvo} um metamodelo que define uma linguagem de modelagem alvo. Seja f: $\mathcal{M}(MM_{origem}) \rightarrow \mathcal{M}(MM_{alvo})$ uma função de mapeamento do tipo modelo-para-modelo, respeitando a definição geral de (46) para um modelo de entrada. Uma função de sincronização para o mapeamento f é uma função $sync_f$ com a assinatura apresentada em (81).

$$sync_f : \mathcal{M}(MM_{origem}) \times \mathcal{M}(MM_{origem}) \times \mathcal{M}(MM_{alvo}) \rightarrow \mathcal{M}(MM_{origem}) \times \mathcal{M}(MM_{alvo}) \quad (81)$$

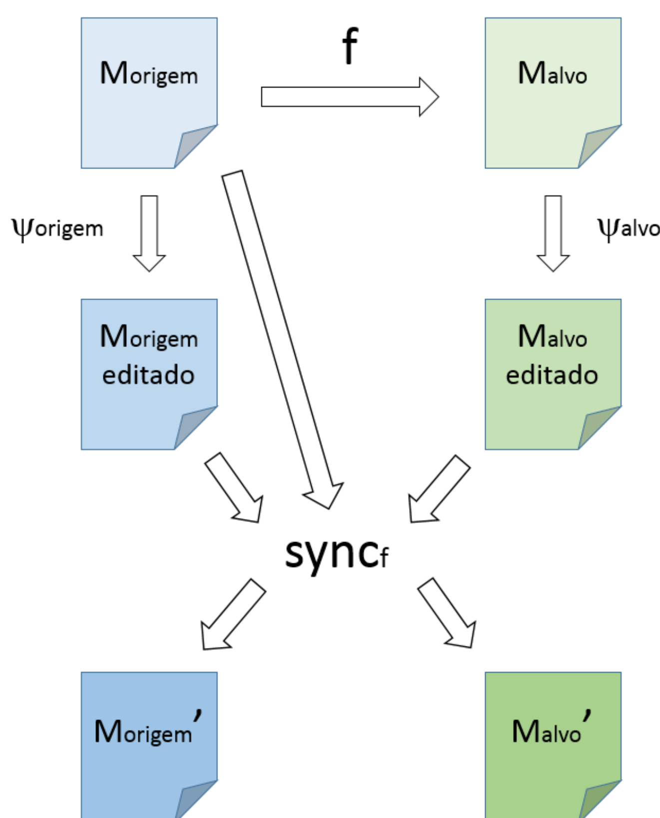
A função $sync_f$ toma como entradas o modelo na linguagem de origem, o modelo na linguagem de origem com as alterações que se deseja propagar para o modelo alvo e o modelo na linguagem alvo com as modificações que se deseja propagar para o modelo de origem. Observe que $sync_f$ não precisa do modelo na linguagem alvo antes das modificações pois ele pode ser obtido aplicando-se f ao modelo na linguagem de origem.

Verifica-se que a função de sincronização, em teoria, é capaz de gerar atualizações nos dois sentidos. Ou seja, um modelo em um domínio alvo também pode ser

editado pelo desenvolvedor e suas alterações propagadas de volta para o modelo no domínio de origem.

A Figura 23 ilustra o contexto apresentado. Sejam dois modelos inter-relacionados M_{origem} (p. ex., um diagrama de classes UML) e M_{alvo} (p. ex., um diagrama Entidade-Relacionamento), sendo que $M_{alvo} = f(M_{origem})$. Ambos podem passar por edições simultâneas por parte do desenvolvedor, resultando em $\psi_{origem}(M_{origem})$ e $\psi_{alvo}(M_{alvo})$. Essa notação é apresentada mais adiante na Definição 19. As alterações em M_{origem} precisam ser propagadas de algum modo para M_{alvo}, e vice-versa, resultando nos modelos sincronizados M_{origem}' e M_{alvo}'. A função $sync_f$ é responsável por realizar essa sincronização.

Figura 23 – Contextualização da função de sincronização



Fonte: autor

Definição 17. Operação de Modificação. Seja MM um metamodelo arbitrário. Uma operação de modificação ϕ é uma função ϕ: $U_M \rightarrow U_M$ que toma um modelo conforme a MM como entrada e produz um modelo bem formado como saída,

representando a aplicação de uma das funções de edição de modelos apresentadas na seção 4.13 (a saber: insert_empty_instance, insert_or_edit_slot, delete_slot e delete_instance), com valores atribuídos às outras entradas além do modelo. Diz-se que uma operação de modificação ϕ especializa uma função de edição de modelo.

Exemplo da Definição 17. A operação de modificação $\phi(M)$ = delete_instance(M, ”arc01”) exclui a instância de nome “arc01” do modelo M.

Definição 18. Operações de Modificação Distintas. Se duas operações de modificação ϕ_1 e ϕ_2 afetam diferentes partes do modelo, diz-se que elas são distintas ou independentes. Isso acontece quando os valores atribuídos às entradas da função que está sendo especializada respeitam as condições do Quadro 8. O índice 1 indica o parâmetro de entrada para ϕ_1 e o índice 2 a entrada para ϕ_2. O significado de cada argumento de cada função foi descrito na seção 4.13.

Quadro 8 – Operações de modificação distintas

$\phi_1 \downarrow \quad \phi_2 \rightarrow$	insert_empty_instance	insert_or_edit_slot	delete_slot	delete_instance
insert_empty_instance	$i_1 \neq i_2 \wedge$ $instancename_1 \neq$ $instancename_2$	$instancename_1 \neq$ $instancename_2 \wedge$ $i_1 \notin val_2$	$instancename_{i1}$ $\neq$ $instancename_{i2}$	$instancename_1 \neq$ $instancename_2$
insert_or_edit_slot	$instancename_1 \neq$ $instancename_2 \wedge$ $i_2 \notin val_1$	$instancename_{i1} \neq$ $instancename_2 \vee$ $propertyname_1 \neq$ $propertyname_2$	$instancename_1 \neq$ $instancename_2 \vee$ $propertyname_1 \neq$ $propertyname_2$	$instancename_1 \neq$ $instancename_2$
delete_slot	$instancename_1 \neq$ $instancename_2$	$instancename_1 \neq$ $instancename_2$	$instancename_1 \neq$ $instancename_2 \vee$ $propertyname_1 \neq$ $propertyname_2$	$instancename_1 \neq$ $instancename_2$
delete_instance	$instancename_1 \neq$ $instancename_2$	$instancename_1 \neq$ $instancename_2$	$instancename_1 \neq$ $instancename_2$	$instancename_1 \neq$ $instancename_2$

Fonte: autor

Duas operações de modificação distintas podem ter sua ordem de aplicação trocada sem afetar o resultado final, pois elas afetam diferentes partes do modelo. Isso significa que se ϕ_1 e ϕ_2 são distintas, então $\phi_1 \circ \phi_2 = \phi_2 \circ \phi_1$.

Exemplo da Definição 18. Sejam $\phi_1(M)$ = insert_or_edit_slot(M, “place01”, “numberOfTokens”, “3”) e $\phi_2(M)$ = insert_or_edit_slot(M, “place02”, “numberOfTokens”, “1”). ϕ_1 e ϕ_2 são distintas pois respeitam a condição correspondente do Quadro 8, ou seja “place01” ≠ “place02” $\vee$ “numberOfTokens” ≠ “numberOfTokens”. Intuitivamente, essas operações editam valores de *slots* de diferentes instâncias, sendo que $\phi_1 \circ \phi_2$ e $\phi_2 \circ \phi_1$ fornecem o mesmo resultado final.

Definição 19. Sequência de Operações de Modificação. Uma sequência de operações de modificação ψ é uma composição de operações de modificação tal que $\psi = \phi_1 \circ \phi_2 \circ ... \circ \phi_n$. Uma sequência de operações de modificação distintas é uma sequência tal que todas as operações são distintas entre si. Ψ denota o conjunto de todas as sequências desse tipo.

Xiong et al. (2007) argumentam que uma sequência de operações de modificação não-distintas pode ser convertida em uma sequência que contém apenas operações distintas. Por exemplo, se uma operação que altera o valor do *slot* de uma propriedade de certa instância para “a” é seguida de outra operação que altera o mesmo *slot* para “b”, então é suficiente usar apenas a segunda operação para representar a sequência. Se uma instância passa por operações de edição de *slots* e posteriormente é excluída, então é suficiente usar apenas a operação de exclusão para respresentar a sequência. Se a instância foi criada dentro da própria sequência e posteriormente excluída, então todas as operações sobre essa instância podem ser removidas para gerar a sequência distinta, afinal a instância não existe ao seu término, o que é equivalente a nunca a ter criado. O que Xiong et al. (2007) não comentam é o caso de operações de inserção de instâncias seguidas por edição de seus *slots*. Nesse caso, o *slot* de uma instância não pode ser editado antes de sua criação e, portanto, a operação de inserção de instância no modelo (insert_empty_instance) deve sempre preceder as operações de edição de seus *slots* (insert_or_edit_slot), não sendo possível gerar uma sequência equivalente de operações distintas nesse caso. O mesmo vale para uma instância que é inserida no modelo e posteriormente referenciada no valor de um *slot* de outra instância. Para

resolver esse problema, divide-se a sequência equivalente em duas partes distintas, de modo diferente ao apresentado por Xiong et al. (2007).

Seja uma sequência arbitrária ψ de operações de modificação. Qualquer ψ pode ser representada por uma sequência equivalente composta por até duas sequências distintas $\psi_{insertions}$ e ψ_{others} tal que $\psi = \psi_{others} \circ \psi_{insertions}$, ou $\psi(M) = \psi_{others}(\psi_{insertions}(M))$. Ou seja, as sequências ψ_{others} e $\psi_{insertions}$ são distintas em si mesmas, mas não distintas entre si. $\psi_{insertions}$, que contém apenas operações do tipo insert_empty_instance, é a sequência a ser aplicada antes no modelo, criando todas as instâncias necessárias. ψ_{others} pode conter operações dos tipos insert_or_edit_slot, delete_slot e delete_instance. ψ_{others} ou $\psi_{insertions}$ podem ser vazias.

A possibilidade dessa equivalência é importante, pois a seguir serão enunciadas propriedades sobre sequências de operações distintas. Caso se disponha de uma sequência não-distinta, sabe-se que ela pode ser convertida em uma composição de até duas sequências distintas.

Definição 20. Idempotência das Sequências de Operações de Modificação Distintas. Aplicar uma sequência de operações distintas gera o mesmo efeito de aplicá-la duas vezes. Esse fato está representado em (82). Diz-se que esse tipo de sequência é idempotente.

$$\forall\psi \in \Psi\ (\forall M \in U_M\ (\psi(\psi(M)) = \psi(M))) \qquad (82)$$

A prova desta propriedade, não apresentada por Xiong et al. (2007), é verificada a seguir. Seja $\psi = \phi_1 \circ \phi_2 \circ ... \circ \phi_n$. A aplicação dupla de ψ em um modelo, $\psi(\psi(M))$, é, portanto, igual a $\phi_1 \circ \phi_2 \circ ... \circ \phi_n \circ \phi_1 \circ \phi_2 \circ ... \circ \phi_n$. Mas as operações são distintas por hipótese, pois $\psi \in \Psi$, e, sendo assim, pode-se reescrever a expressão como $\phi_1 \circ \phi_1 \circ \phi_2 \circ \phi_2 \circ ... \circ \phi_n \circ \phi_n$. Na seção 4.13, foi visto que a aplicação dupla das quatro funções de edição de modelos equivale a uma só aplicação. Desse modo, $\phi_k \circ \phi_k = \phi_k$ com $1 \leq k \leq n$. Assim, a expressão se reduz novamente a $\phi_1 \circ \phi_2 \circ ... \circ \phi_n$,

que é igual a ψ. Algebricamente, essa propriedade é importante para verificar se uma sequência já foi aplicada em um modelo, fato verificado quando $M = \psi(M)$.

Definição 21. Operação de Modificação Refletível. Algumas modificações no modelo de um domínio não podem ser propagadas para o modelo do outro domínio. Por exemplo, em uma função de mapeamento que mapeia um diagrama de classes UML para código Java, por exemplo, se esta função não é capaz de gerar comentários no código Java, então introduzir ou editar comentários no código é uma operação não-refletível com respeito à função de mapeamento. Caso contrário, a operação é refletível. Ou seja, dada uma função de mapeamento $f: \mathcal{M}(MM_{origem}) \rightarrow \mathcal{M}(MM_{alvo})$, diz-se que a operação de modificação ϕ_{alvo} é refletível com respeito a f se para todo $M_{origem} \in \mathcal{M}(MM_{origem})$ existe uma operação de modificação ϕ_{origem} tal que $f(\phi_{origem}(M_{origem})) = \phi_{alvo}(f(M_{origem}))$. Em outras palavras, são operações feitas no modelo alvo que podem ser refletidas para o modelo origem por meio de outra operação ϕ_{origem}, de tal modo que a aplicação de f no modelo origem alterado pela operação refletida ϕ_{origem} resulte no modelo alvo gerado por $f(M_{origem})$ e posteriormente alterado por ϕ_{alvo}.

Definição 22. Sequências de Operações Não-Refletíveis. Seja $\psi = \phi_1 \circ \phi_2 \circ ... \circ \phi_n$ uma sequência de operações distintas. A notação $\psi|_f$ denota o conjunto das operações não-refletíveis com respeito a f que compõem ψ.

Definição 23. Propriedade de Estabilidade. Se nem o modelo na linguagem origem nem o modelo na linguagem alvo são modificados, então a função de sincronização não deve modificar nenhum deles. A eq. (83) ilustra esta propriedade.

$$sync_f\ (M_{origem}, M_{origem}, f(M_{origem})) = (M_{origem}, f(M_{origem})) \qquad (83)$$

A partir daqui, ψ_{origem} denota uma sequência de operações de modificação distintas no domínio da linguagem de origem, ou seja, $\psi_{origem}: U_M \rightarrow U_M$, e ψ_{alvo} uma

sequência de operações de modificação distintas no domínio da linguagem alvo, tal que ψ_{alvo}: $U_M \rightarrow U_M$. M_{origem} denota um modelo na linguagem de origem e M_{alvo} denota um modelo na linguagem alvo.

Definição 24. Propriedade de Preservação. Se o resultado da aplicação de $sync_f$ sobre modelos origem e alvo editados por ψ_{origem} e ψ_{alvo}, respectivamente, passar por nova aplicação de ψ_{origem} e ψ_{alvo}, então não ocorrem novas alterações. Ou seja:

$$sync_f\ (M_{origem},\ \psi_{origem}(M_{origem}),\ \psi_{alvo}(f(M_{origem}))) = (M_{origem}',\ M_{alvo}') \Rightarrow \\ \psi_{origem}(M_{origem}') = M_{origem}' \wedge \psi_{alvo}(M_{alvo}') = M_{alvo}' \quad (84)$$

Isso quer dizer que as alterações feitas em M_{origem} (representadas por ψ_{origem}) e aquelas feitas em $M_{alvo} = f(M_{origem})$ (representadas por ψ_{alvo}) antes da aplicação de $sync_f$ devem ser preservadas em M_{origem}' e M_{alvo}', respectivamente, que são os resultados da sincronização. Isso quer dizer que a sincronização não “destrói” o que o desenvolvedor criou em cada modelo.

Definição 25. Propriedade de Propagação. O modelo alvo sincronizado M_{alvo}' contém todas as modificações de ψ_{origem} em M_{origem} após a transformação de f, e o modelo sincronizado de origem M_{origem}' contém todas as modificações refletíveis de ψ_{alvo} em M_{alvo}. O racional por trás desta propriedade, indicada por (85), é que se uma modificação refletível em ψ_{alvo} não estiver refletida em M_{origem}', então ela não poderia ser gerada aplicando-se f em M_{origem}', e então a equação $\psi_{alvo}|_f\ (f(M_{origem}')) = M_{alvo}'$ não poderia valer, pois não seria possível reobter as modificações das operações refletíveis de ψ_{alvo} em M_{alvo}' via f.

$$sync_f\ (M_{origem},\ \psi_{origem}(M_{origem}),\ \psi_{alvo}(f(M_{origem}))) = (M_{origem}',\ M_{alvo}') \Rightarrow \\ \psi_{alvo}|_f\ (f(M_{origem}')) = M_{alvo}' \quad (85)$$

Se uma modificação em ψ_{origem} não estiver em M_{alvo}' mas for levada pela aplicação de f, então M_{alvo}' não poderia ser igual a $\psi_{alvo}|_f\ (f(M_{origem}'))$ porque $f(M_{origem}')$ inclui esta

modificação por hipótese. Esta propriedade se refere a propagação de modificações nas duas direções e permite modificações não-refletíveis nos modelos alvo sem que sejam sobrescritas ou perdidas na próxima sincronização.

Definição 26. Propriedade de Composição. Esta propriedade indica que aplicar a função de sincronização $sync_f$ duas vezes com duas sequências de operações deve produzir o mesmo efeito que sincronizar uma vez uma única sequência composta pelas duas sequências originais. A eq. (86) representa esta propriedade. Sejam ψ_{origem}' e ψ_{alvo}' sequências adicionais a ψ_{origem} e ψ_{alvo} a serem aplicadas após a primeira sincronização. Uma consequência desta propriedade é a possibilidade de sincronização após um período off-line. Isto é, desenvolvedores podem trabalhar em M_{origem} e M_{alvo} de forma independente, gerando modificações em cada modelo. O resultado de sincronizar a cada operação ou de sincronizar "em lote" após o acúmulo de modificações deve ser o mesmo.

$$\begin{gathered}(sync_f\ (M_{origem}, \psi_{origem}(M_{origem}), \psi_{alvo}(f(M_{origem}))) = (M_{origem}', M_{alvo}')) \wedge \\ (sync_f\ (M_{origem}, \psi_{origem}'(M_{origem}'), \psi_{alvo}'(M_{alvo}')) = (M_{origem}'', M_{alvo}'')) \Rightarrow \\ sync_f\ (M_{origem}, \psi_{origem}'(\psi_{origem}(M_{origem})), \psi_{alvo}'(\psi_{alvo}(f(M_{origem})))) = \\ (M_{origem}'', M_{alvo}'')\end{gathered} \tag{86}$$

A formalização da semântica e das propriedades de sincronização oferece um arcabouço para a construção de ferramentas ou rotinas de sincronização de modelos. Também ajuda a entender requisitos e potencialidades de um processo de sincronização, bem como suas limitações, por exemplo, no caso de sequências de operações não-refletíveis.

A edição simultânea de diferentes modelos de um mesmo sistema, porém inter-relacionados, é um dos problemas da MDE, requerendo um processo de sincronização automático para se obter produtividade e eliminação de incoerências entre modelos. O estudo de conflitos, ou seja, alterações conflitantes em M_{origem} e M_{alvo}, é um tópico em aberto. Uma das maneiras de resolver é definir um modelo como prioritário, cujas alterações prevalecem se conflitarem com alterações do outro modelo.

Esta subseção apresenta apenas as propriedades que uma função de sincronização $sync_f$, a ser obtida a partir da função de mapeamento f, deve possuir. O processo de obtenção da função de sincronização não é abordado neste trabalho, mas o leitor pode recorrer a Xiong et al. (2007) para maiores explicações. Nem mesmo há garantias de que existe uma $sync_f$ para qualquer f e, ademais, pode haver ambiguidades no mapeamento definido por f de modo que haja mais de uma possibilidade de função de sincronização correspondente.

O Capítulo 6 parte para a apresentação de aplicações práticas do SBMM, afinal o objetivo principal da MDE é o desenvolvimento de sistemas a partir de modelos. Uma forma de validar a real aplicabilidade prática do formalismo aqui proposto é através de uma ferramenta criada como prova de conceito.

6 UMA FERRAMENTA PARA APLICAÇÕES PRÁTICAS

6.1 INTRODUÇÃO

Com o objetivo de realizar uma prova de conceito e testar na prática a capacidade de aplicação do formalismo SBMM para a MDE, foi desenvolvida como parte desta pesquisa a ferramenta SBMMTool.

A SBMMTool tem por objetivo a criação e edição de metamodelos, modelos e funções de mapeamento. A execução das funções de mapeamento, que correspondem às transformações, também faz parte do escopo da ferramenta. Dessa forma, pode-se utilizar a ferramenta como um ambiente geral para desenvolvimento MDE, mesmo que em um estágio simples de usabilidade.

No caso das funções de mapeamento, previu-se na ferramenta apenas transformações do tipo modelo-para-código. Por isso, foi implementado um interpretador MOFM2T simplificado, com os aspectos e recursos principais da linguagem. Um resumo sobre a linguagem MOFM2T é apresentado no Apêndice A.

A ferramenta foi desenvolvida utilizando o compilador Free Pascal (FPT, 2010) e o ambiente de desenvolvimento Lazarus (FPT, 2015). Trata-se de ferramentas gratuitas e com amplos recursos disponíveis na Internet. Uma das vantagens dessa escolha é que tanto o compilador Free Pascal quanto o ambiente Lazarus estão disponíveis para várias plataformas, podendo gerar código executável nativo para Windows, Mac OS X e Linux sem a necessidade de máquina virtual para a execução do código compilado.

A seção 6.2 descreve com mais detalhes os recursos e funcionamento da ferramenta, enquanto os Apêndices C e D apresentam estudos de caso ilustrando a aplicação da SBMMTool como ferramenta CASE para classes de programas distintas.

6.2 A FERRAMENTA

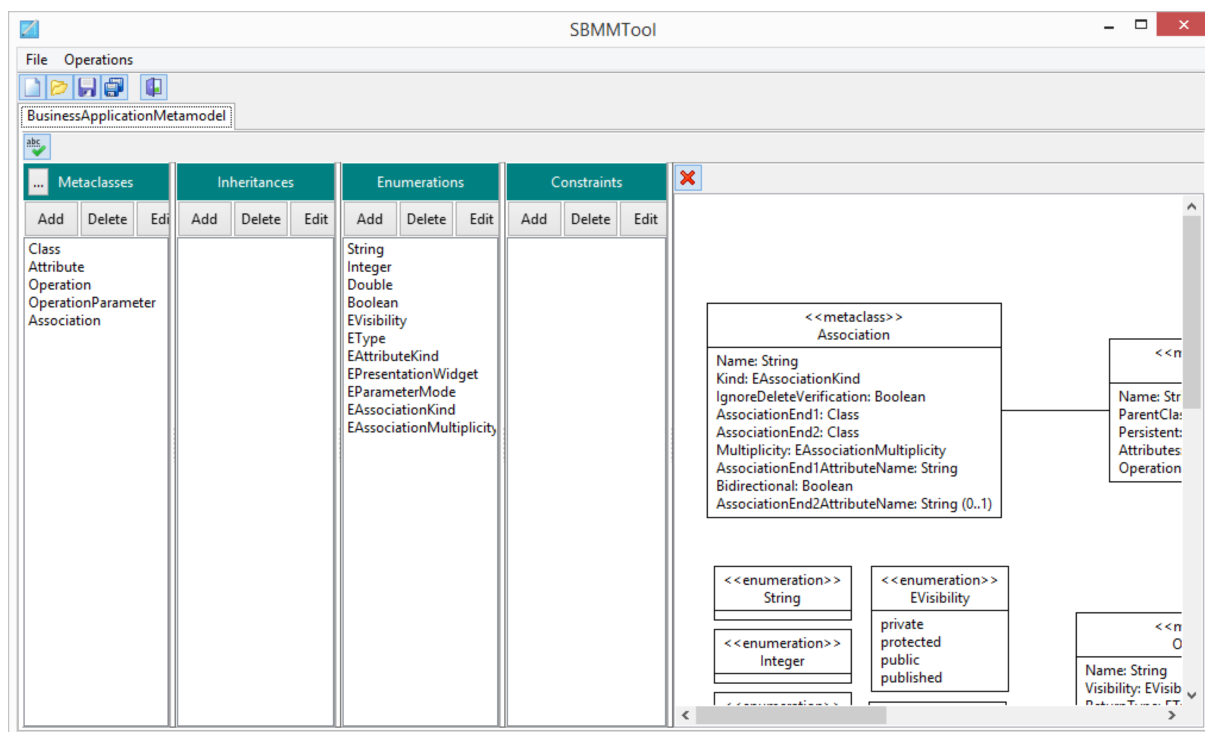
Basicamente, a ferramenta SBMMTool provê cinco macrofuncionalidades: editor de metamodelo, mesclagem de metamodelos, editor de modelo, editor de função de

mapeamento e motor de execução de função de mapeamento. Até a conclusão deste trabalho, as funções de mapeamento suportadas são apenas as do tipo modelo-para-código.

6.2.1 **Editor de metamodelo**

Baseado na definição de metamodelo apresentada na seção 4.2, o editor de metamodelo da SBMMTool permite a inserção, exclusão e edição de metaclasses, generalizações, enumerações e restrições. A Figura 24 apresenta uma tela de exemplo deste editor, onde é possível observar quatro listas na parte esquerda correspondendo a esses quatro conjuntos do formalismo.

Figura 24 – Editor de metamodelo da ferramenta SBMMTool

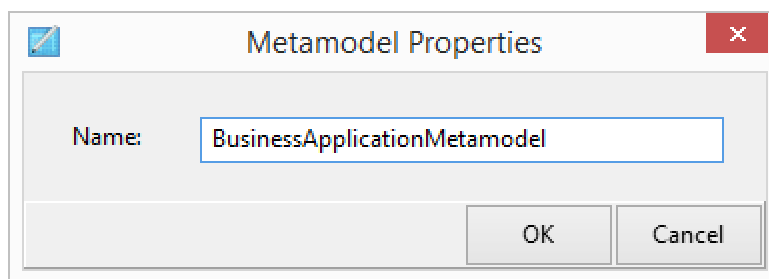


Fonte: autor

O nome do metamodelo, que seria o primeiro elemento da sétupla apresentada na seção 4.2, pode ser editado pelo botão de propriedades (com o rótulo "...") que aparece no canto superior esquerdo do editor de metamodelo. Ao clicar neste botão, é exibida a tela da Figura 25.

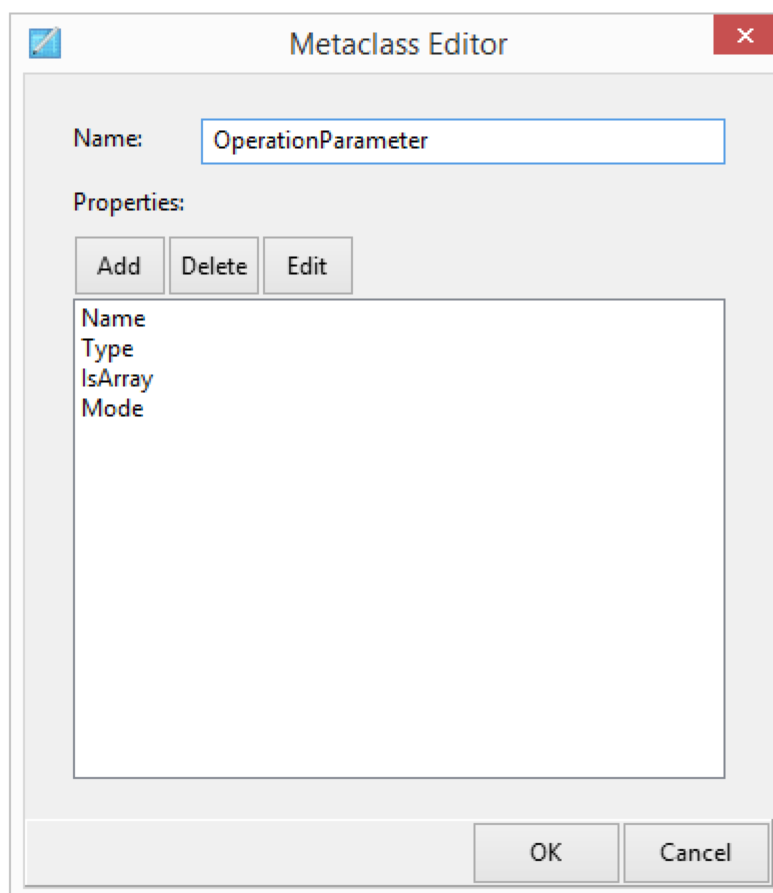
Cada uma das listas apresenta três botões: *Add, Delete* e *Edit*. Eles permitem, respectivamente, inserir, excluir e editar um elemento. Para uma metaclasse, quando inserida ou editada, a tela da Figura 26 aparece para o usuário.

Figura 25 – Editor de propriedades do metamodelo da ferramenta SBMMTool



Fonte: autor

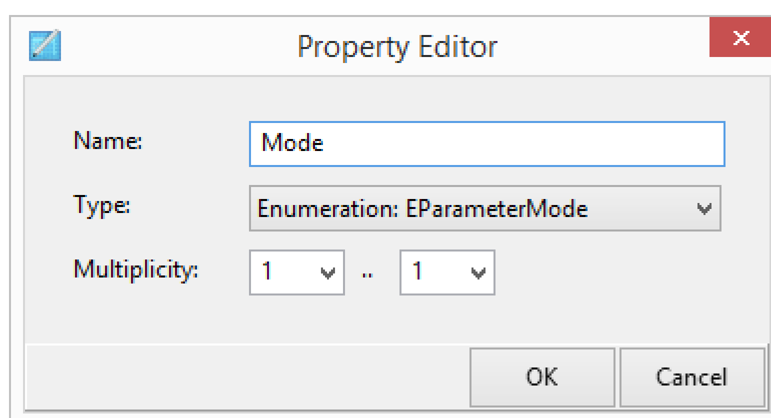
Figura 26 – Editor de metaclasse da ferramenta SBMMTool



Fonte: autor

Na Figura 26, o exemplo é de uma metaclasse chamada *OperationParameter* com quatro propriedades: *Name*, *Type, IsArray* e *Mode*. A tela da Figura 27, por sua vez, serve para editar cada propriedade, que é composta por seu nome, tipo base e multiplicidade, de acordo com a definição da eq. (3). No exemplo, é a propriedade *Mode* que está sendo editada. Ela é do tipo *EParameterMode*, uma enumeração, e tem mutiplicidade 1..1.

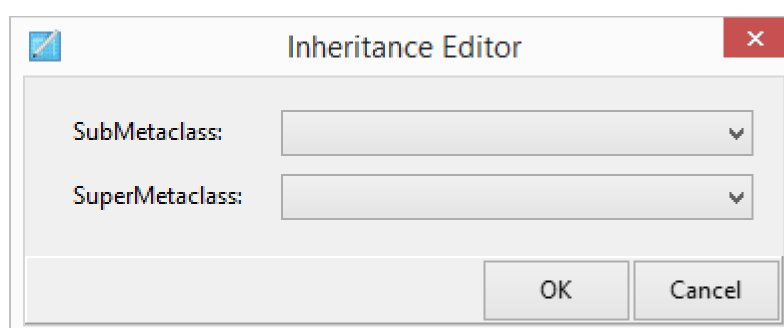
Figura 27 – Editor de propriedade de metaclasse da ferramenta SBMMTool



Fonte: autor

Quando uma generalização é inserida ou editada, a tela da Figura 28 é exibida para o usuário.

Figura 28 – Editor de generalização da ferramenta SBMMTool

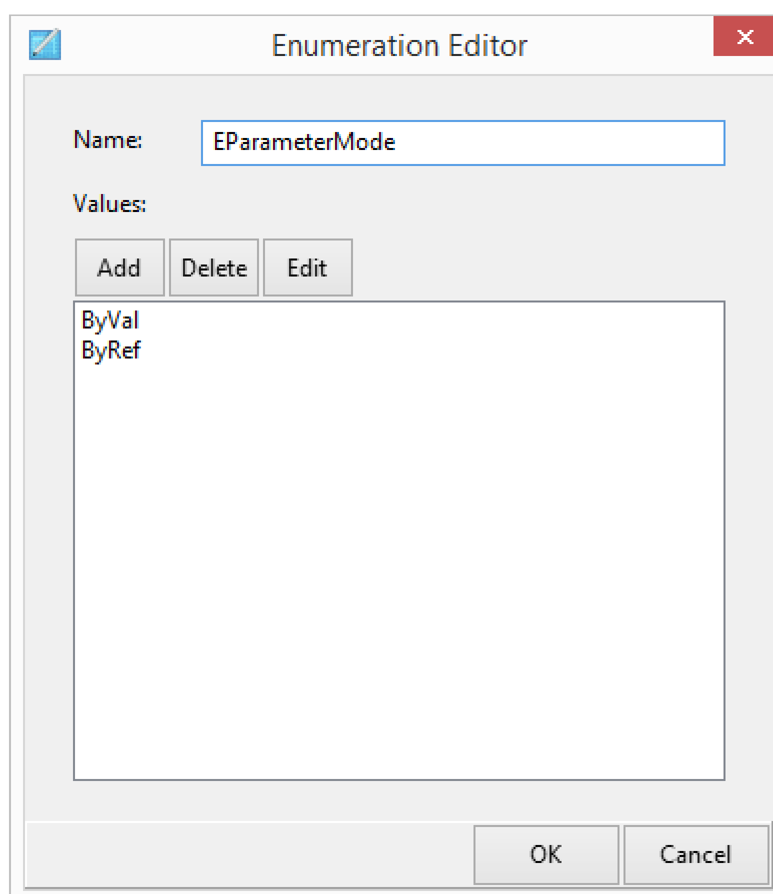


Fonte: autor

Como as generalizações são pares ordenados compostos por duas metaclasses do metamodelo, a tela simplesmente pede para o usuário informar qual é a submetaclasse e a supermetaclasse. A ferramenta checa por ciclos e impedem que os mesmos se formem no metamodelo, conforme especificado nas definições da seção 4.2.

Com relação às enumerações, é a tela da Figura 29 que permite sua inserção e edição. No exemplo, é a enumeração *EParameterMode* que está sendo editada. Ela foi utilizada como tipo base da propriedade *Mode* e possui dois valores permitidos: *ByVal* e *ByRef*.

Figura 29 – Editor de enumeração da ferramenta SBMMTool



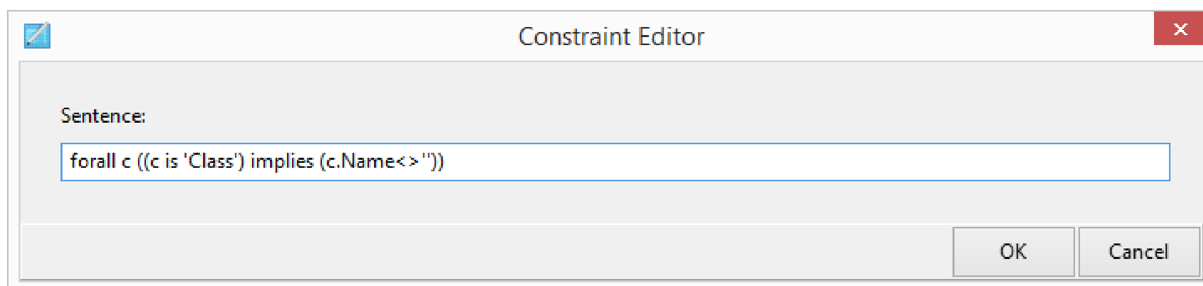
Fonte: autor

Uma das limitações da ferramenta é não permitir editar a lista de valores da enumeração na forma de uma expressão matemática, o que seria conveniente para

enumerações com muitos ou infinitos valores, como foi utilizado nos exemplos teóricos da seção 4.2. Isso poderia ser objeto de um refinamento futuro desse trabalho. Entretanto, a ferramenta inclui sempre três enumerações nativas em todos os metamodelos criados: *String*, *Integer* e *Double*, que correspondem respectivamente aos tipos de dados de cadeias alfanuméricas, números inteiros e números em ponto flutuante com dupla precisão. Essas enumerações são tratadas internamente da forma usual das linguagens de programação. Portanto, se o usuário precisar de alguma enumeração com muitos ou infinitos valores, uma dessas três pode ser utilizada em sua substituição, conforme conveniência.

A tela da Figura 30 é mostrada quando o usuário insere ou edita uma restrição. Ela permite a inserção de uma sentença em lógica de primeira ordem que irá compor o conjunto R da sétupla correspondente ao metamodelo, conforme eq. (1).

Figura 30 – Editor de restrição da ferramenta SBMMTool



Fonte: autor

Como as sentenças em lógica de primeira ordem possuem símbolos específicos não disponíveis nos dispositivos de entrada de dados comuns, foram utilizadas palavras reservadas específicas para os quantificadores e operadores, conforme o Quadro 9. O interpretador de sentenças em lógica de primeira ordem da ferramenta, que avalia as restrições sobre os modelos, foi implementado através de autômatos de pilha estruturados adaptativos (RAMOS; NETO; VEGA, 2009). Mais detalhes são apresentados adiante neste capítulo e também no Apêndice B.

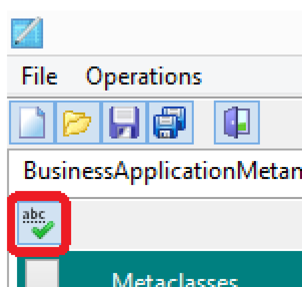
Quadro 9 – Palavras reservadas do interpretador de restrições correspondentes a símbolos de LPO

Símbolo LPO	Palavra
$\forall$	forall
$\exists$	exists
$\neg$	not
$\wedge$	and
$\vee$	or
$\Rightarrow$	implies

Fonte: autor

O editor de metamodelos também apresenta um botão que verifica se o metamodelo está bem formado, conforme visto na seção 4.5. A Figura 31 destaca esse botão, que também pode ser encontrado na Figura 24.

Figura 31 – Botão para checagem de metamodelo bem formado na ferramenta SBMMTool



Fonte: autor

Para finalizar, o editor de metamodelo, na parte direita da tela da Figura 24 encontra-se um editor de diagramas que permite dispor os elementos dos conjuntos do formalismo de forma gráfica, respeitando a notação apresentada na seção 4.4. Basta que o usuário clique sobre um elemento previamente inserido em uma das listas e o arraste para a área do diagrama. Automaticamente a ferramenta renderizará o bloco gráfico correspondente ao elemento selecionado, seja uma metaclasse ou enumeração. Para uma generalização, é renderizada uma linha com seta fechada partindo da submetaclasse em direção à supermetaclasse. Como as restrições não possuem notação gráfica prevista no diagrama, elas são os únicos elementos que não causam alterações no diagrama quando arrastadas para ele. Uma vez que um bloco correspondente a uma metaclasse ou enumeração foi

colocado no diagrama, o usuário pode arrastá-lo para reposicioná-lo conforme conveniência visual.

É importante frisar que o diagrama do metamodelo é apenas para facilitar a visualização, sendo opcional para o usuário. O metamodelo em si é composto pelos elementos inseridos nas quatro listas apresentadas e seus descritores. Se o usuário não arrastou algum desses elementos para o diagrama, isso não influencia em nada sua existência no metamodelo.

Nota-se também que, como um modelo de marcação é um tipo específico de metamodelo, pode-se usar este mesmo editor para criar modelos de marcação.

6.2.2 **Mesclagem de metamodelos**

Baseada na definição de mesclagem de metamodelos apresentada na subseção 4.10.2, a ferramenta SBMMTool dispõe de uma tela que permite selecionar múltiplos metamodelos e gerar um metamodelo resultante da operação de mesclagem. Embora essa operação seja definida sobre dois metamodelos, quando aplicada sobre três ou mais, é feita sua aplicação sucessiva compondo a função $@_{MM}$ com ela mesma, tal que a mesclagem de MM_1, MM_2 e MM_3 seja igual a:

$$@_{MM} (\text{“NewName2”}, @_{MM} (\text{“NewName1”}, MM_1, MM_2), MM_3)$$

A Figura 32 apresenta uma tela de exemplo de seleção de metamodelos para mesclar da ferramenta SBMMTool.

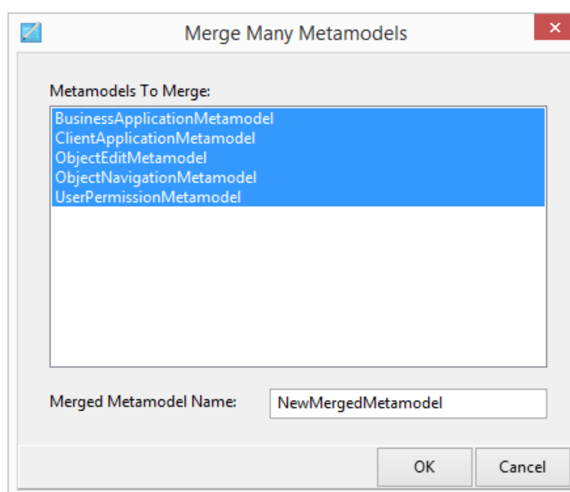
6.2.3 **Editor de modelo**

Baseado na definição de modelo apresentada pela eq. (15), o editor de modelo da SBMMTool permite a inserção, exclusão e edição de instâncias. A Figura 33 apresenta uma tela de exemplo desse editor, onde é possível observar uma lista na parte esquerda correspondendo ao conjunto I do formalismo do modelo.

A imagem da Figura 33 apresenta o exemplo de modelo da seção 4.8 inserido na ferramenta. É possível observar que as instâncias são apresentadas de forma

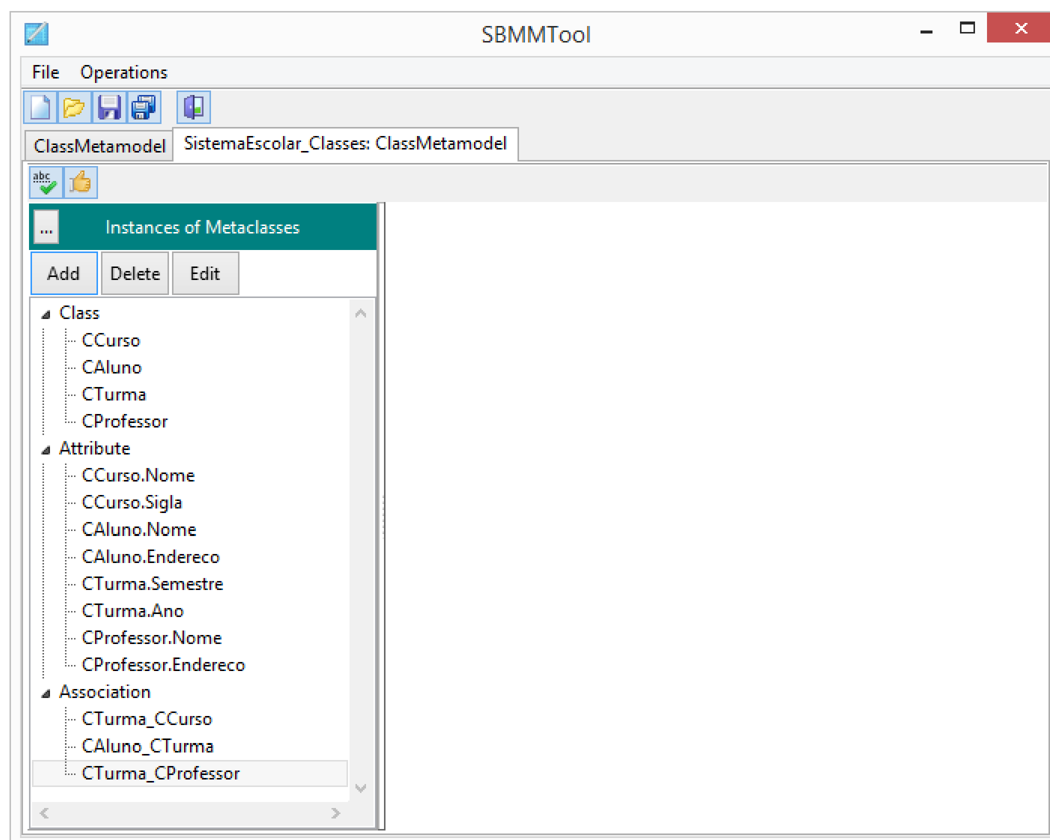
agrupada por metaclasse, em uma estrutura de árvore. Assim, o usuário pode enxergar com mais facilidade as instâncias de cada tipo.

Figura 32 – Tela de seleção de metamodelos para mesclar da ferramenta SBMMTool



Fonte: autor

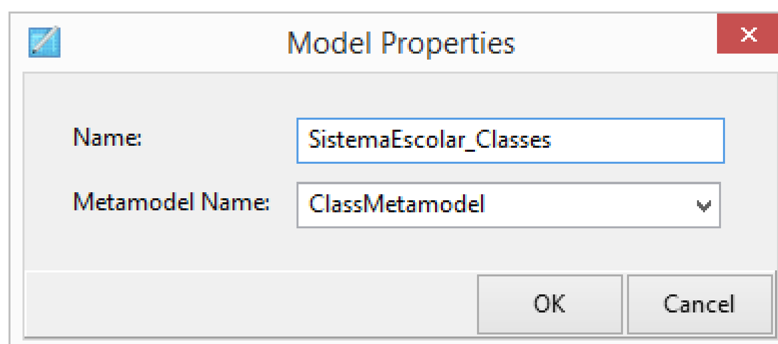
Figura 33 – Editor de modelo da ferramenta SBMMTool



Fonte: autor

O nome do modelo, que seria o primeiro elemento da tripla apresentada na eq. (15), pode ser editado pelo botão de propriedades (com o rótulo "...") que aparece no canto superior esquerdo do editor de modelo. Ao clicar neste botão, é exibida a tela da Figura 34. Nessa mesma tela também é possível selecionar o nome do metamodelo alvo, que corresponde ao segundo elemento da tripla.

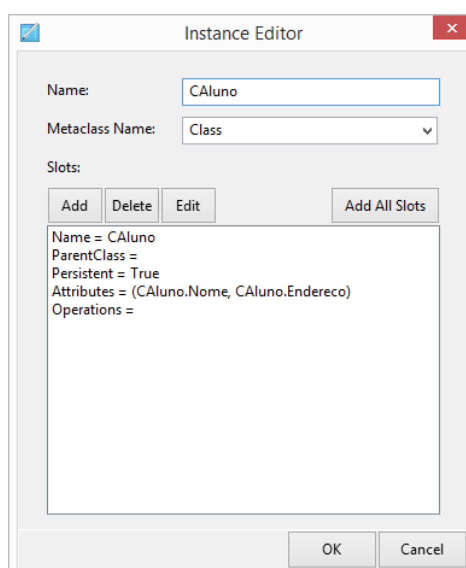
Figura 34 – Editor de propriedades do modelo da ferramenta SBMMTool



Fonte: autor

Ao inserir ou editar uma instância, a tela da Figura 35 é mostrada ao usuário, a qual corresponde ao editor de instância.

Figura 35 – Editor de instância da ferramenta SBMMTool



Fonte: autor

O exemplo da Figura 35 apresenta a classe *CAluno* do modelo da seção 4.8. É possível observar o nome da metaclasse que é seu tipo (*Class*) e também o conteúdo de cada *slot* na lista entitulada *Slots*. Botões para inserir, excluir e editar *slots* também estão presentes no topo da lista.

Quando um *slot* é inserido ou editado, a ferramenta apresenta a tela da Figura 36. Neste exemplo, a tela está carregada com os dados do *slot Attributes* da instância *CAluno*. Por corresponder a uma propriedade com multiplicidade maior que 1, a tela habilita a inserção de múltiplos valores neste *slot*. Neste caso, pode-se observar que o *slot* contém duas instâncias: *CAluno.Nome* e *CAluno.Endereco*.

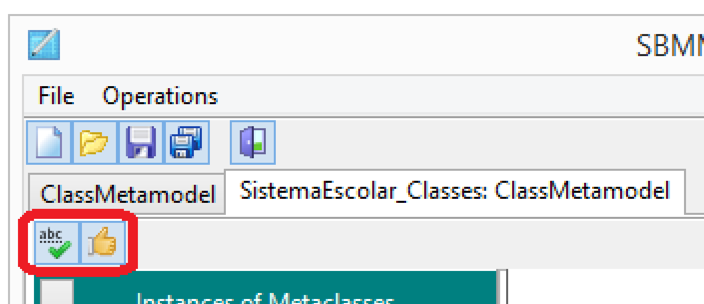
O botão *Add* insere uma posição no *slot*, desde que ele corresponda a uma propriedade com multiplicidade maior que 1, e permite a seleção de uma instância pré-existente para ocupá-la. O botão *Create and Add*, por sua vez, cria uma nova instância, exibe a tela da Figura 35 para que o usuário possa preencher seus dados, e só depois a adiciona em uma nova posição do *slot*.

Figura 36 – Editor de slot da ferramenta SBMMTool

Fonte: autor

As checagens de modelo bem formado e da relação de conformidade com seu metamodelo podem ser realizadas através dos botões destacados na Figura 37, respectivamente. Conforme definição, a relação de conformidade requer primeiramente que o modelo esteja bem formado. Portanto, um modelo não bem formado nunca resultará positivo no teste da relação de conformidade.

Figura 37 – Botões para checagem de modelo bem formado e conformidade na SBMMTool



Fonte: autor

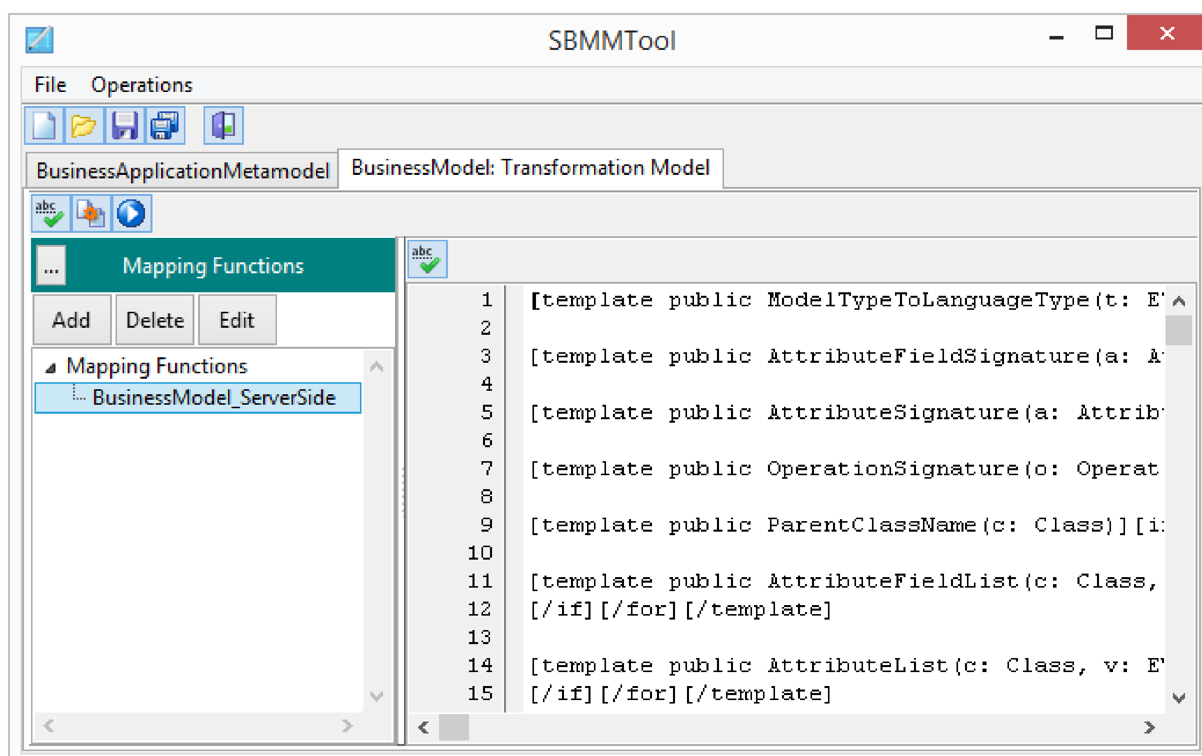
Tal como no editor de metamodelo, na parte direita da tela da Figura 33 encontra-se um editor de diagramas que permite dispor as instâncias do modelo de forma gráfica. Porém, no caso dos metamodelos a notação é única. No caso dos modelos, cada tipo de modelo possui uma notação gráfica própria. Por exemplo, um diagrama de classes UML possui uma notação gráfica para classes distinta daquela para casos de uso e atores em um diagrama de casos de uso. Como a definição de metamodelo do SBMM não prevê mecanismo para a associação de símbolos gráficos para cada metaclasse, foi usada a notação de objetos de diagramas de classes. Cada instância de metaclasse, independente de seu tipo, é representada por um bloco igual ao de um objeto UML. Embora permita visualizar qualquer modelo de forma gráfica com uma notação única, pode causar estranheza por não estar na notação própria. Nesse caso, por exemplo, casos de uso e atores seriam representados como blocos quadrados correspondentes a objetos, e não pela sua notação usual de elipse e boneco, respectivamente. A introdução de um mecanismo de associação de símbolos gráficos, ou sintaxe concreta, a metaclasses pode ser uma melhoria futura na ferramenta, o que permitiria personalizar a notação gráfica de cada metamodelo para ser aplicada no editor de modelo.

Basta que o usuário clique sobre uma instância previamente inserida e a arraste para a área do diagrama para que a mesma seja renderizada na forma de um bloco de objeto. Uma vez que um bloco foi colocado no diagrama, o usuário pode arrastá-lo para reposicioná-lo conforme conveniência visual.

6.2.4 Editor de função de mapeamento do tipo modelo-para-código

A seção 4.12 apresentou definições para funções de mapeamento dos tipos modelo-para-modelo e modelo-para-código. A ferramenta SBMMTool implementa a possibilidade de editar funções de mapeamento modelo-para-código utilizando um subconjunto do MOFM2T como linguagem de especificação. A tela da Figura 38 mostra uma função de mapeamento aberta na ferramenta, em que é possível observar o editor MOFM2T com uma especificação de exemplo.

Figura 38 – Editor de função de mapeamento da ferramenta SBMMTool



Fonte: autor

Apenas os seguintes recursos do MOFM2T são suportados pelo interpretador simplificado criado para a ferramenta: *template*, *file*, *if*, *for*, *protected*, exibição de valores e chamadas de outros *templates*. Para compor expressões, a especificação MOFM2T prevê a possibilidade de chamar funções da OCL. Essa implementação não contempla funções OCL, mas apresenta a possibilidade de montar expressões básicas com *strings*. Um resumo sobre a linguagem MOFM2T é apresentado no Apêndice A.

Duas alterações foram feitas em relação à especificação original do MOFM2T (OMG, 2008) para fins exclusivos de simplificação do projeto do analisador sintático. Como o objetivo principal desta pesquisa não foi criar uma implementação de referência precisa do MOFM2T, optou-se por essas alterações sem prejuízo ao funcionamento proposto.

Essas alterações permitiram que o analisador sintático pudesse operar com *look ahead* de no máximo um *token*. A primeira dessas alterações é a introdução da função value para a exibição de valores. Por exemplo, a expressão *[a.Name /]* no MOF2MT seria substituída por *[value(a.Name) /]* nessa implementação. A segunda delas é a introdução da palavra reservada *call* nas chamadas de outros templates. Desse modo, a expressão *[anotherTemplate(a,b) /]* seria substituída por *[call anotherTemplate(a,b) /]*.

Uma terceira alteração introduz uma funcionalidade prevista na definição de motor de transformação do tipo modelo-para-código da eq. (51). Trata-se da utilização da versão anterior do modelo como entrada para o motor, mas a especificação do MOFM2T (OMG, 2008) não provê expressões para obter o valor de uma propriedade de uma instância da versão anterior do modelo. Por isso, foi introduzido o operador # como indicador de versão anterior. Enquanto a expressão *[value(a.Name) /]* reproduz na cadeia de saída o valor da propriedade *Name* da instância *a* com relação ao modelo vigente, a expressão *[value(#a.Name) /]* reproduz o valor com relação ao modelo anterior. Isso permite que a função de mapeamento calcule diferenças e produza saídas conforme as mesmas. Por exemplo, o trecho do Quadro 10 produz o texto “propriedade Name alterada” na saída somente se a propriedade *Name* passou por alterações em relação à versão anterior do modelo.

Quadro 10 – Exemplo de uso do operador #

```
1 [if(a.Name <> #a.Name)]
2    propriedade Name alterada
3 [/if]
```

Fonte: autor

6.2.5 **Motor de execução da função de mapeamento modelo-para-código**

A tela correspondente ao editor de função de mapeamento modelo-para-código apresentada na Figura 38 possui botões para acionar as funcionalidades de checagem de sintaxe e de execução da transformação.

Um analisador sintático e um interpretador MOFM2T foram implementados como parte da ferramenta SBMMTool. Sua implementação é baseada em autômatos de pilha estruturados adaptativos. Um autômato raiz inicia a análise do arquivo de entrada, que corresponde à especificação da função de mapeamento em MOFM2T, e dispara outros autômatos encadeados como sub-máquinas conforme os comandos que encontra. A adaptatividade se faz necessária apenas no modo de interpretação, e não no modo de análise sintática, para a execução dos laços *for*. Enquanto as iterações não acabam, é mantida uma transição do estado do fim do laço para o estado de início do mesmo que dispara o reposicionamento do ponteiro do analisador léxico para o primeiro símbolo presente dentro do laço *for*. Assim que as iterações acabam, esta transição é removida e o processamento da cadeia de entrada prossegue daquele ponto.

O Apêndice B apresenta os autômatos de pilha estruturados adaptativos que foram usados no projeto do analisador sintático e interpretador.

Como resultado da execução do motor, produz-se a saída na forma de código-fonte de acordo com a função de mapeamento, metamodelo e modelos processados.

6.2.6 **Interpretador de sentenças em lógica de primeira ordem**

Para implementar a checagem de conformidade de modelos, foi necessário construir um interpretador de sentenças em lógica de primeira ordem para avaliar restrições. O Quadro 9 apresentou as palavras que substituem os símbolos quantificadores e outros. Conforme a seção 4.6, predicados são interpretados como metaclasses e funções como valores atribuídos a propriedades nos *slots* do modelo. O interpretador construído é baseado no autômato denominado *FOL* (que denota *First Order Logic*), apresentado no Apêndice B, e implementa esses conceitos. Entretanto, algumas adaptações sintáticas foram feitas para que se aproveitasse o autômato avaliador de expressões *EXPRESSION* construído para o interpretador MOFM2T. Essas adaptações são três:

- *Predicado(a)* é denotado como *a is 'Predicado'*. Essa expressão resulta em 1 (verdadeiro) caso a instância de nome *a* possua a metaclasse de nome *Predicado* como tipo, ou 0 (falso) caso contrário;
- *Predicado*(a)* é denotado como *a is_direct 'Predicado'*. Essa expressão resulta em 1 (verdadeiro) caso a instância de nome *a* possua a metaclasse de nome *Predicado* como tipo base, ou 0 (falso) caso contrário;
- *Metaclasse.Propriedade(a)* é denotado como *a.Propriedade*, utilizando portanto uma notação mais similar àquelas das linguagens orientadas a objeto tradicionais. Não é necessário especificar o nome da metaclasse na expressão pois o interpretador consegue obter os tipos de *a* em tempo de execução. A expressão retorna o valor atribuído à propriedade *Propriedade* para a instância *a* através do *slot* correspondente.

Com essas adaptações sintáticas nas expressões em lógica de primeira ordem, reduziu-se o trabalho de implementação ao se aproveitar o resolvedor de expressões MOFM2T, sem a necessidade de criar autômatos com os mesmos objetivos, porém com sintaxe reconhecida diferente.

Esse autômato resolvedor de expressões usado tanto pelo interpretador MOFM2T quanto pelo interpretador de lógica de primeira ordem é capaz de trabalhar com

números e strings como constantes, bem como operações básicas de adição, subtração, multiplicação, divisão, potência, resto, e-lógico, ou-lógico, ou-exclusivo, implicação, concatenação, comparações e checagem de tipos, além de avaliação de valores de propriedades de instâncias.

6.3 EXEMPLOS DE APLICAÇÃO

Para ilustrar a utilização da ferramenta SBMMTool, são fornecidos exemplos de duas classes de aplicações distintas. O Apêndice C apresenta um exemplo de aplicação para programas adaptativos. Ou seja, mostra a SBMMTool sendo aplicada como ferramenta CASE para desenvolvimento de programas adaptativos de acordo com a abordagem MDE. O Apêndice D, por sua vez, apresenta um exemplo de aplicação para aplicativos de negócio. Trata-se de uma categoria de sistemas com mais variedade de requisitos, permitindo testar a aplicação do formalismo também nesse ambiente através da SBMMTool.

Após ter apresentado o formalismo SBMM e algumas aplicações teóricas e práticas, que serviram como forma de validação de sua real utilidade, parte-se para as considerações finais deste trabalho, onde são apresentados os resultados, contribuições, conclusão e ideias de trabalhos futuros.

7 CONSIDERAÇÕES FINAIS

7.1 RESULTADOS E DISCUSSÃO

Este trabalho propôs e apresentou um formalismo não-metacircular para metamodelagem denominado SBMM, baseado em conjuntos, relações e funções como conceitos primitivos principais. Apresentaram-se definições e explicações para metamodelos e sua notação gráfica, modelos, restrições sobre modelos, relação de conformidade, modelos de marcação, operação de mesclagem de metamodelos, funções de mapeamento modelo-para-modelo, funções de mapeamento modelo-para-código e funções para construção incremental de modelos.

O formalismo proposto pode então ser comparado aos outros formalismos estudados, por meio de sua introdução nos quadros comparativos apresentados originalmente no Capítulo 3, conforme Quadro 11, Quadro 12 e Quadro 13.

Em relação ao MOF, o SBMM apresenta a vantagem de ser especificado formalmente, e não apenas em descrições baseadas em linguagem natural. Embora não apresente duas das capacidades do MOF, conforme mostra o Quadro 12, junta capacidades de outros formalismos e introduz algumas capacidades que nenhum dos outros apresenta, ao menos explicitamente, tais como modelos de marcação e modelos executáveis. O recurso de mesclagem de metamodelos, já existente no MOF, passou a ser formalizado no contexto do SBMM.

Quadro 11 – Comparativo de formalismos e linguagens segundo especificação, incluindo o SBMM

Característica	MOF	KM3	Alanen	NEREUS	Jackson	SBMM
Metacircular	Sim	Sim	Não	Não	Sim	Não
Natureza predominante da especificação	Linguagem natural	Formal	Formal	Linguagem natural	Formal	Formal
Principais conceitos primitivos subjacentes usados na especificação	Meta-classes	Multigrafos direcionados e funções	Conjuntos, relações e funções	Meta-classes	Grafos direcionados e funções	Conjuntos, relações e funções

Fonte: autor

Quadro 12 – Comparativo de formalismos e linguagens sobre recursos, incluindo o SBMM

Característica	MOF	KM3	Alanen	NEREUS	Jackson	SBMM
Suporte a operações nas metaclasses	Sim	Não	Não	Sim	Não	Não
Suporte à definição de enumerações e tipos primitivos	Sim	Não*	Não**	Não*	Sim	Sim
Suporte à capacidade de reflexão do MOF	Sim	Não	Não	Não	Não	Não
Suporte à capacidade de identificação do MOF	Sim	Não	Não	Sim	Não	Sim
Suporte à capacidade de extensão do MOF ou modelos de marcação	Sim	Não	Não	Não	Não	Sim
Suporte à mesclagem e importação de metamodelos	Sim	Não	Não***	Somente importação	Não	Sim
Suporte à transformação de modelos	Sim	Sim	Não	Sim****	Não*****	Sim
Descreve modelos executáveis	Não	Não	Não	Não	Não	Sim
Descreve sincronização de modelos	Não	Não	Não	Não	Não	Sim

* Utiliza como recurso implícito, mas não especifica definições.
** Não, mas sugere que é possível.
*** Apenas assume implicitamente esta possibilidade.
**** Menciona QVT como uma possibilidade para transformar modelos, mas não especifica sua interação com a linguagem apresentada.
***** Refere-se à transformação como operações no contexto de um mesmo modelo para editá-lo, mas não no contexto da MDE entre modelos de diferentes tipos.

Fonte: autor

Quadro 13 – Comparativo de formalismos sobre sintaxe concreta e utilização, incluindo o SBMM

Característica	MOF	KM3	Alanen	NEREUS	Jackson	SBMM
Linguagem ou notação de referência para metamodelagem	Notação gráfica própria similar a classes UML	Conjuntos, relações, sentenças em LPO e linguagem própria	Conjuntos, relações e sentenças em LPO	Própria	FORMULA	Conjuntos, relações, sentenças em LPO e notação gráfica
Linguagem ou notação de referência para modelagem	UML	Conjuntos, relações, sentenças em LPO e linguagem própria	Conjuntos, relações e sentenças em LPO	Não especifica	FORMULA	Conjuntos, relações e sentenças em LPO e notação gráfica
Linguagem ou notação de referência para restrições sobre modelos	OCL	Sentenças em LPO e linguagem própria	LPO*	Própria	FORMULA	LPO
Linguagem ou notação de referência para transformações	QVT e MOFM2T	ATL	Não especifica	QVT	Não especifica	QVT e MOFM2T
Mecanismo de referência para inferência e raciocínio (*reasoning*)	Não especifica	LPO	LPO	Não especifica	FORMULA	LPO
Ferramentas de referência	Diversas. Ex: Papyrus e Acceleo	Compilador Prolog, Eclipse	Não especifica	Não especifica	FORMULA	SBMMTool

* Não especifica mecanismo explícito de inserir restrições específicas sobre modelos nos metamodelos, mas como usa LPO para definir restrições comuns e invariantes para todos os modelos, então se subentende que pode ser usada LPO para isso.

Fonte: autor

O formalismo apresentado também foi usado para definir precisamente os conceitos de modelos executáveis, inclusive adaptativos, e suas semânticas de execução. Uma conceituação desse tipo não foi encontrada na literatura, considerando a revisão sistemática apresentada por Szvetits e Zdun (2013), sendo usualmente o funcionamento dos modelos executáveis explicados em linguagem natural e diagramas. Como outro exemplo de aplicação teórica, o SBMM foi utilizado para definir as ideias de Xiong et al. (2007) sobre sincronização de modelos, apresentada em seu trabalho com base em definições próprias com foco em transformação de modelos, mas desconectadas de outros conceitos da MDE.

A ferramenta SBMMTool foi criada como prova de conceito, com o objetivo de avaliar se o formalismo pode realmente ser aplicado na MDE, não sendo apenas um conjunto de definições e expressões distantes de aplicação prática. A construção da ferramenta por si só também não seria suficiente, pois a MDE advoga a geração automática de software executável a partir de modelos de alto nível como artefatos de primeira classe.

Para testar a aplicação do formalismo através da ferramenta, foram realizados dois estudos de caso para diferentes classes de aplicações: o primeiro para programas adaptativos (Apêndice C) e o segundo para aplicativos de negócio (Apêndice D). O primeiro caso representa um ambiente mais restritivo, enquanto segundo caso possui maior variedade de requisitos. Com a introdução e representação adequada de novos metamodelos (que compuseram DSMLs especializadas) e funções de mapeamento, pôde-se gerar o código-fonte com alto grau de funcionalidades prontas, ainda com a possibilidade de ser alterado manualmente por um programador para maiores refinamentos sem que uma nova transformação automática os sobreponha, desde que respeitadas certas regras.

7.2 CONTRIBUIÇÕES

Como contribuições desta pesquisa, pode-se citar:

1. Criação de um formalismo de metamodelagem e modelagem que pode ser usado como teoria subjacente para descrever conceitos e fenômenos da MDE. Reúne conceitos de outros formalismos da literatura (p. ex., relação de conformidade, funções para construção incremental de modelos) e preenche lacunas não abordadas por eles (p. ex., operação de mesclagem entre metamodelos e modelos de marcação), apresentando-se como uma formulação original e ampliando o conhecimento da área;
2. Separação clara entre a notação gráfica dos metamodelos (sintaxe concreta) e sua sintaxe abstrata, com um mapeamento entre ambos a partir de equações, não contando apenas com a linguagem natural. Em alguns formalismos, como o próprio MOF, a notação gráfica se confunde com os próprios objetos do metamodelo. Aqui, separa-se plenamente as definições

algébricas sobre os objetos do metamodelo (camada M2) de sua notação gráfica, que é dada pela associação de símbolos visuais a um grafo que pode ser derivado a partir de qualquer metamodelo conforme detalhado na seção 4.4;

3. Modelagem algébrica de transformação de modelos, tanto de modelo-para-modelo quanto de modelo-para-código, incluindo explicitamente o conceito de refinamento de modelos, de forma independente da linguagem ou método utilizado para descrever as transformações específicas. Não se observou a abordagem do conceito de refinamento de modelos nos formalismos apresentados no Capítulo 3;
4. Estabelecimento de definições formais sobre modelos executáveis e modelos executáveis adaptativos, usualmente descritos na literatura através de linguagem natural e figuras. A adaptatividade de modelos executáveis parece ter sido estudada no contexto da MDE primeiramente, segundo os próprios autores, por Cariou, Le Goaer e Barbier (2012), Cariou et al. (2013) e Pierre et al. (2014), sendo a introdução de definições formais desse tipo uma contribuição original. Canovas e Cugnasca (2016), adicionalmente, apresentam um mapeamento de modelos adaptativos para dispositivos adaptativos guiados por regras, contextualizando os primeiros dentro da teoria geral de adaptatividade. Desse modo, além de se ampliar os conhecimentos da área, é feita uma conexão com a formulação teórica geral de dispositivos adaptativos de Neto (2001);
5. Proposta de inserção do operador # no MOFM2T, com o objetivo de obter o valor do operando na versão anterior do modelo, permitindo calcular diferenças entre versões e gerar código de acordo;
6. Enquanto se pretende que um ambiente de desnevolvimento MDA possa ser atingido com uma cadeia de ferramentas (MELLOR et al., 2004), mostrou-se que também é possível utilizar os principais conceitos da MDE em uma única ferramenta de forma bastante simples, apenas com cinco funcionalidades gerais. Isso poderia permitir a alavancagem da utilização da MDE como prática mais geral nos processos de desenvolvimento de software, incluindo a metamodelos e funções de mapeamento como artefatos do dia-a-dia do desenvolvedor, tais como já são os modelos e código-fonte.

Como contribuições para aplicações específicas, pode-se citar:

7. Criação de um metamodelo e função de mapeamento para programas adaptativos, permitindo utilizar a SBMMTool como ferramenta CASE para essa classe de aplicações e aplicar a abordagem MDE, inclusive gerando código-fonte em linguagem BADAL a partir dos modelos. Tal aplicação continua o trabalho de pesquisa de Silva (2010), gerando mais um avanço e contribuição original de ordem prática para a área;
8. Criação de cinco metamodelos e funções de mapeamento para, em conjunto, gerar automaticamente partes consideráveis de aplicativos de negócio. Embora a DSML estabelecida por esses metamodelos ainda careça de comparações com outras tais como ActionGUI (ACTIONGUI, 2016) e WebML (WEBML, 2013), foi possível utilizá-la de modo simples para gerar programas atendendo diversos requisitos dessa classe de aplicação. Canovas e Cugnasca (2015b) apresentam um exemplo de aplicação no contexto do Laboratório de Automação Agrícola da Escola Politécnica, relacionado ao WebBee, um sistema de informação que organiza conhecimento e facilita o compartilhamento de informações sobre abelhas.

7.3 CONCLUSÃO

A MDE é considerada uma das mais recentes mudanças de paradigma da Engenharia de Software (KOLOVOS et al., 2013). Sua abordagem propõe trazer dois benefícios principais: o aumento do nível de abstração em relação à programação tradicional e a possibilidade de aproveitar os mesmos modelos de alto nível para gerar ou executar o sistema em várias plataformas.

Uma das áreas de pesquisa da MDE é a fundamentação e formalização de seus conceitos. O MOF, padrão do OMG para a MDA, é considerado subformalizado (JACKSON; LEVENDOVSZKY; BALASUBRAMANIAN, 2013) e sofre outras críticas do ponto de vista de definições, tais como aquelas vistas na subseção 3.2.3. Adicionalmente, alguns autores consideram que a ausência de formalização dificulta

a adoção prática e ampla da MDE (RUTLE et al., 2012). De fato, a MDE ainda não atingiu sua disseminação na indústria de software (MOHAGHEGHI et al., 2013), embora seus benefícios já tenham sido comprovados por diversos estudos (KARNA; TOLVANEN; KELLY, 2009) (PAPOTTI et al., 2013) e sua adoção venha crescendo (CUADRADO; IZQUIERDO; MOLINA, 2014).

Este trabalho de pesquisa buscou oferecer contribuições no campo da formalização de conceitos da MDE, oferecendo uma proposta de teoria para descrever conceitos, operações, propriedades e fenômenos em geral. Utilizaram-se ideias de outros formalismos da literatura para unificar conceitos e preencher lacunas, trazendo também algumas originalidades tais como aquelas descritas na seção 7.2. Com isso, acredita-se ter oferecido um avanço no campo teórico da MDE, incluindo capacidades de aplicação.

7.4 TRABALHOS FUTUROS

Como sugestões de continuidade desta pesquisa e trabalhos futuros, pode-se citar:

1. Incluir no formalismo a capacidade de um metamodelo conter outro metamodelo, representando a hierarquia de pacotes que existe no MOF;
2. Carecterísticas de associações do MOF, tais como *ordered*, composição e agregação não existem diretamente no SBMM, mas podem ser obtidas por construções específicas. Por exemplo, para obter uma associação ordenada, pode-se criar uma metaclasse intermediária que registre a ordem de cada instância associada através de uma propriedade inteira *Order*. Pode-se criar um catálogo de soluções de metamodelagem no SBMM, estilo padrões de projeto (*design patterns*), que indique como obter recursos presentes diretamente no MOF que não foram contemplados para simplificar o formalismo;
3. Na ferramenta SBMMTool, permitir a definição de valores permitidos de enumerações a partir de expressões. Até a conclusão deste trabalho, é necessário especificar cada valor permitido;

4. Na ferramenta SBMMTool, incluir um editor de notação gráfica para os elementos dos metamodelos, de modo que os diagramas dos modelos possam apresentar notações próprias e não uma única notação geral;
5. Na ferramenta SBMMTool, incluir um editor de semântica de execução de modelos e motor de execução para modelos executáveis, tais como conceituados na seção 5.2;
6. Estudo de processos de desenvolvimento de software incluindo fases de metamodelagem iterativa;
7. Incluir no formalismo alguma maneira de especificar operações de mesclagem de modelos (e não apenas metamodelos). Essas dependem da semântica atribuída ao metamodelo;
8. Formalização de mesclagem de semânticas de execução quando metamodelos executáveis são mesclados;
9. Estudo comparativo entre a SBMMTool e outras ferramentas para a MDE apoiadas em outros formalismos, tais como os *plug-ins* Papyrus e Acceleo para a IDE Eclipse, que são apoiadas no MOF;
10. A UML é a referência principal e amplamente difundida de linguagem de modelagem de software. Neste trabalho, um metamodelo para o diagrama de classes da UML foi apresentado como exemplo, com simplificações. Idem para casos de uso. A tentativa de expressar todos os tipos de modelos que a UML prevê através de metamodelos SBMM pode ajudar a identificar lacunas e carências do mesmo, permitindo desenvolvimentos adicionais no formalismo ou sobre ele;
11. Prever o conceito de ações específicas de domínio nos modelos executáveis ou nas semânticas de execução.

REFERÊNCIAS

ACTION GUI TEAM. ActionGUI Web Page**.** Disponível em: <http://www.actiongui.org>. Acesso em: 07 fev. 2016.

ALANEN, M.; PORRES, I. A Metamodeling Language Supporting Subset and Union Properties. **Software & Systems Modeling**, v. 7, n. 1, p. 103-124, 2008.

ALY, M.; CHARFI, A.; WU, D.; MEZINI, M. Understanding multilayered applications for building extensions. In: WORKSHOP ON COMPREHENSION OF COMPLEX SYSTEMS, 1, 2013, Fukuoka, Japão. **Proceedings of the 1st workshop on Comprehension of complex systems**. Nova York, EUA: ACM, 2013, p. 1-6.

AMBLER, S. **The Object Primer**: Agile Modeling-Driven Development with UML 2.0. 3rd ed. New York: Cambridge University Press, 2004.

AMBLER, S. The Design of a Robust Persistence Layer For Relational Databases. 2005. Disponível em <http://www.ambysoft.com/downloads/persistenceLayer.pdf>. Acesso em: 07 fev. 2016.

AMBLER, S. Mapping Objects to Relational Databases: O/R Mapping In Detail. 2006. Disponível em: <http://www.agiledata.org/essays/mappingObjects.html>. Acesso em: 07 fev. 2016.

AMBLER, S. Agile Modeling (AM) Home Page. 2014. Disponível em: <http://www.agilemodeling.com>. Acesso em: 07 fev. 2016.

BAAR, T. Metamodels without Metacircularities. **L'Objet**, v. 9, n. 4, p. 95-114, 2003.

BECK, K. Embracing changes with extreme programming. **IEEE Computer**, v. 32, n. 10, p. 70-77, 1999.

BECK, K.; BEEDLE, M.; VAN BENNEKUM, A.; COCKBURN, A.; CUNNINGHAM, W.; FOWLER, M.; GRENNING, J.; HIGHSMITH, J.; HUNT, A.; JEFFRIES, R.; KERN, J.; MARICK, B.; MARTIN, R.; MELLOR, S.; SCHWABER, K.; SUTHERLAND, J.; THOMAS, D. Manifesto for Agile Software Development. 2001. Disponível em: <http://agilemanifesto.org>. Acesso em: 07 fev. 2016.

BENCOMO, N.; FRANCE, R. B.; CHENG, B. H. C.; AβMANN, U. **Models@run.time**: Foundations, Applications and Roadmaps. Springer International Publishing, 2014.

BÉZIVIN, J.; GERBÉ, O. Towards a precise definition of the OMG/MDA framework. In: ANNUAL INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING, 16, 2001, San Diego, EUA. **Anais**. IEEE, 2001, p. 273-280.

BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. **The Unified Modeling Language User Guide.** Addison-Wesley, 1999.

BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. **The Unified Software Development Process**. Addison-Wesley, 1999.

BORONAT, A; MESEGUER, J. An algebraic semantics for MOF. In: INTERNATIONAL CONFERENCE ON FUNDAMENTAL APPROACHES TO SOFTWARE ENGINEERING, 11, 2008, Budapeste, Hungria. **Lecture Notes in Computer Science vol. 4961**. Springer Berlin Heidelberg, 2008, p. 377-391.

BOULANGER, F.; HARDEBOLLE, C. Simulation of Multi-Formalism Models with ModHel'X. In: INTERNATIONAL CONFERENCE ON SOFTWARE TESTING, VERIFICATION AND VALIDATION, 1, 2008, Lillehammer, Noruega. **Anais**. IEEE, 2008, p. 318-327.

BRETON, E.; BÉZIVIN, J. Towards an Understanding of Model Executability. In: INTERNATIONAL CONFERENCE ON FORMAL ONTOLOGY IN INFORMATION SYSTEMS, 1, 2001. **Anais.** ACM, 2001, p. 70-80.

CABOT, J. Relationship between MDA, MDD and MDE. 2009. Disponível em: <http://modeling-languages.com/relationship-between-mdamdd-and-mde>. Acesso em: 22 abr. 2016.

CANOVAS, S. R. M.; CUGNASCA, C. E. Um metamodelo para programas adaptativos utilizando SBMM. In: WORKSHOP DE TECNOLOGIA ADAPTATIVA, 8, 2014, São Paulo. **Memórias do VIII Workshop de Tecnologia Adaptativa – WTA 2014.** São Paulo: USP, 2014, p. 27-34. Disponível em: <http://lta.poli.usp.br/lta/publicacoes/artigos/2014/memorias-do-wta-2014>. Acesso em: 07 fev. 2016.

CANOVAS, S. R. M.; CUGNASCA, C. E. Em direção ao desenvolvimento de programas adaptativos utilizando MDE**.** In: WORKSHOP DE TECNOLOGIA ADAPTATIVA, 9, 2015, São Paulo. **Memórias do IX Workshop de Tecnologia Adaptativa – WTA 2015.** São Paulo: USP, 2015, p. 29-39. Disponível em: <http://lta.poli.usp.br/lta/publicacoes/artigos/2015/memorias-do-wta-2015>. Acesso em: 07 fev. 2016.

CANOVAS, S. R. M.; CUGNASCA, C. E. Applying Model Driven Engineering to Develop a Bee Information System. In: THE EFITA | WCCA | CIGR CONFERENCE, 10, 2015, Poznan, Polônia. **Anais**, p. 103-114. Disponível em: <https://www.dropbox.com/s/lus0snxu491tg65/EFITA2015-proceedings-abstracts.pdf?dl=0>. Acesso em: 07 fev. 2016.

CANOVAS, S. R. M.; CUGNASCA, C. E. Um mapeamento de modelos adaptativos para dispositivos adaptativos guiados por regras. In: WORKSHOP DE TECNOLOGIA ADAPTATIVA, 10, 2016, São Paulo. **Memórias do X Workshop de Tecnologia Adaptativa – WTA 2016.** São Paulo: USP, 2016, p. 86-97. Disponível em: <http://lta.poli.usp.br/lta/publicacoes/artigos/2016/memorias-do-wta-2016>. Acesso em: 07 fev. 2016.

CARDELLI, L.; WEGNER, P. On Understanding Types, Data Abstraction, and Polymorphism. **Computing Surveys**, v. 17, n. 4, p. 471-522, 1985.

CARIOU, E.; BALLAGNY, C.; FEUGAS, A.; BARBIER, F. Contracts for Model Execution Verification. In: EUROPEAN CONFERENCE ON MODELING FOUNDATIONS AND APPLICATIONS, 7, 2011, Birmingham, Reino Unido. **Lecture Notes in Computer Science vol. 6698.** Springer Berlin Heidelberg, 2011, p. 3-18.

CARIOU, E.; LE GOAER, O.; BARBIER, F. Model Execution Adaptation? In: INTERNATIONAL WORKSHOP ON MODELS@RUN.TIME, 7, 2012, Innsbruck, Austria. **Proceedings of the 7th workshop on Models@run.time**. Nova York, EUA: ACM, 2012, p. 60-65.

CARIOU, E.; LE GOAER, O.; BARBIER, F.; PIERRE, S. Characterization of Adaptable Interpreted-DSML: In: EUROPEAN CONFERENCE ON MODELING FOUNDATIONS AND APPLICATIONS, 9, 2013, Montpellier, França. **Lecture Notes in Computer Science vol. 7949.** Springer Berlin Heidelberg, 2013, p. 37-53.

CLARK, T.; EVANS, A.; KENT, S. The metamodeling language calculus: foundation semantics for UML. In: INTERNATIONAL CONFERENCE ON FUNDAMENTAL APPROACHES TO SOFTWARE ENGINEERING, 4, 2001, Gênova, Itália. **Lecture Notes in Computer Science vol. 2029**. Springer Berlin Heidelberg, 2001, p. 17-31.

COMBEMALE, B.; CRÉGUT, X.; PANTEL, M. A Design Pattern to Build Executable DSMLs and Associated V&V Tools. In: ASIA-PACIFIC SOFTWARE ENGINEERING CONFERENCE, 19, 2012, Hong Kong. **Anais**. IEEE, 2008, p. 282-287.

COMBEMALE, B.; HARDEBOLLE, C.; CHRISTOPHE, J.; BOULANGER, F.; BAUDRY, B. Bridging the Chasm between Executable Metamodeling and Models of Computation. In: INTERNATIONAL CONFERENCE ON SOFTWARE LANGUAGE ENGINEERING, 5, 2012, Dresden, Alemanha. **Lecture Notes in Computer Science vol. 7745.** Springer Berlin Heidelberg, 2012, p. 184-203.

CUADRADO, J. S.; IZQUIERDO, J. L. C.; MOLINA; J. G. Applying model-driven engineering in small software enterprises**. Science of Computer Programming**, n. 89, p. 176-198, 2014.

FAVRE, J. M. Towards a basic theory to model model driven engineering. In: WORKSHOP IN SOFTWARE MODEL ENGINEERING, 3, 2004, Lisboa, Portugal. **Anais**, 2004, p. 262-271.

FAVRE, L. A Formal Foundation for Metamodeling**.** In: ADA-EUROPE INTERNATIONAL CONFERENCE ON RELIABLE SOFTWARE TECHNOLOGIES, 14, 2009, Brest, França. **Lecture Notes in Computer Science vol. 5570.** Springer Berlin Heidelberg, 2009, p. 177-191.

FEYNMAN, R. EBNF: a Notation to Describe Syntax. 2012. Disponível em <https://www.ics.uci.edu/~pattis/ICS-33/lectures/ebnf.pdf>. Acesso em: 23 abr. 2016.

FONDEMENT, F.; MULLER, A.; THIRY, L.; WITTMANN, B.; FORESTIER, G. Big Metamodels are Evil. In: MODEL-DRIVEN ENGINEERING LANGUAGES AND SYSTEMS, 16, 2013, Miami, EUA. **Lecture Notes in Computer Science vol. 8107.** Springer Berlin Heidelberg, 2013, p. 138-153.

FREE PASCAL TEAM. Free Pascal Web Page. 2010. Disponível em: <http://www.freepascal.org>. Acesso em: 07 fev. 2016.

FREE PASCAL TEAM. Lazarus Homepage. 2015. Disponível em: <http://www.lazarus.freepascal.org>. Acesso em: 07 fev. 2016.

GESSENHARTER, D.; RAUSCHER, M. Code Generation for UML 2 Activity Diagrams. In: EUROPEAN CONFERENCE ON MODELING FOUNDATIONS AND APPLICATIONS, 7, 2011, Birmingham, Reino Unido. **Lecture Notes in Computer Science vol. 6698.** Springer Berlin Heidelberg, 2011, p. 205-220.

GRAHAN, I. **Business Rules Management and Service Oriented Architecture.** Chichester: Wiley, 2006.

JAAKSI, A. Developing Mobile Browsers in a Product Line**. IEEE Software**, v. 19, n. 4, pp. 73-80, 2002.

JACKSON, E.; LEVENDOVSZKY, T.; BALASUBRAMANIAN, D. Automatically reasoning about metamodeling**. Software & Systems Modeling**, v. 14, n. 1, p. 271-285, 2013.

JÉZÉQUEL, J. M.; COMBEMALE, B.; DERRIEN, S.; GUY, C.; RAJOPADHYE, S. Bridging the chasm between MDE and the world of compilation**. Software & Systems Modeling**, v. 11, n. 4, p. 581-597, 2012.

JOUAULT, F.; KURTEV, I. Transforming models with ATL. In: INTERNATIONAL CONFERENCE ON MODEL DRIVEN ENGINEERING LANGUAGES AND SYSTEMS, 8, 2005, Montego Bay, Jamaica. **Lecture Notes in Computer Science vol. 3844.** Springer Berlin Heidelberg, 2005, p. 128-138.

JOUAULT, F.; BÉZIVIN, J. KM3: A DSL for Metamodel Specification. In: INTERNATIONAL CONFERENCE ON FORMAL METHODS FOR OPEN OBJECT-BASED DISTRIBUTED SYSTEMS, 8, 2006, Bologna, Itália. **Lecture Notes in Computer Science vol. 4037.** Springer Berlin Heidelberg, 2006, p. 171-185.

KARNA, J.; TOLVANEN, J. P.; KELLY, S. Evaluating the Use of Domain-Specific Modeling in Practice. In: WORKSHOP ON DOMAIN-SPECIFIC MODELING, 9, 2009, Orlando, EUA. **Proceedings of the 9th Workshop on Domain-Specific Modeling (DSM'09)**. Helsinki: Helsinki School of Economics, 2009, p. 14-20.

KITCHENHAM, B. **DESMET**: A Method for Evaluating Software Engineering Methods and Tools. Keele, Reino Unido: Department of Computer Science, University of Keele, Keele, UK, Agosto 1996. (Technical Report TR96-09).

KÖBLER, F.; MILEWSKI, A.; EGAN, R.; ZHANG, S.; TREMAINE, M. **Methodological Diversity in Global Software Engineering Research.** 2008. (Technical Report). Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.539.4352&rep=rep1&type=pdf>. Acesso em: 07 fev. 2016.

KÖNIGS, A. Model Transformation with triple graph grammars. In: MODEL TRANSFORMATIONS IN PRACTICE SATELLITE WORKSHOP OF MODELS, 8, 2005, Montego Bay, Jamaica. **Lecture Notes in Computer Science vol. 3844.** Springer Berlin Heidelberg, 2005, p. 166.

KOLOVOS, D. S.; ROSE, L. M.; MATRAGKAS, N.; PAIGE, R. F.; GUERRA, E.; CUADRADO, J. S.; DE LARA, J.; RATH, I.; VARRÓ, D.; TISI, M.; CABOT, J. A Research Roadmap Towards Achieving Scalability in Model Driven Engineering. In: WORKSHOP ON SCALABILITY IN MODEL DRIVEN ENGINEERING, 1, 2013, Budapeste, Hungria. **Proceedings of the Workshop on Scalability in Model Driven Engineering**. ACM, 2013, p. 2.

LACHTERMACHER, L.; SILVEIRA, D. S.; PAES, R. B.; LUCENA, C. J. P. **Transformando o Diagrama de Atividade em uma Rede de Petri.** 2008. 27 p. Monografia - Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2008.

MARCHETTI; G. A. **Um método de transformação de modelos UML para a inclusão de componentes de frameworks com o uso de planejador lógico.** 2012. 80 p. Dissertação (Mestrado) - Escola Politécnica, Universidade de São Paulo, São Paulo, 2012.

MARKOVIC, S.; BAAR, T. Semantics of OCL specified with QVT. **Software & Systems Modeling**, v. 7, n. 4, p. 399-422, 2008.

MAXIMILIEN, E. M.; CAMPOS, P. Facts, trends and challenges in modern software development. **International Journal of Agile and Extreme Software Development**, v. 1, n. 1, p. 1-5, 2012.

MELLOR, S. J.; BALCER, M. J. **Executable UML**: A Foundation For Model-Driven Architecture. Addison-Wesley, 2002.

MELLOR, S.; SCOTT, K.; UHL, A.; WEISE, D. **MDA Distilled**: Principles of Model-Driven Architecture. Boston: Pearson Education Inc., 2004.

MOHAGHEGHI, P.; FERNANDEZ, M.A.; MARTELL, J.A., FRITZSCHE, M., GILANI, W. MDE adoption in industry: challenges and success criteria. In: INTERNATIONAL CONFERENCE ON MODEL DRIVEN ENGINEERING LANGUAGES AND SYSTEMS, 11, 2008, Toulouse, França. **Lecture Notes in Computer Science vol. 5421.** Springer Berlin Heidelberg, 2008, p. 54-59.

MOHAGHEGHI, P.; GILANI, W.; STEFANESCU, A.; FERNANDEZ, M. A. An empirical study of the state of the practice and acceptance of model-driven engineering in four industrial cases. **Empirical Software Engineering**, v. 18, n. 1, p. 89-116, 2013.

MULLER, P. A.; FLEUREY, F.; JÉZÉQUEL, J. M. Weaving Executability into Object-Oriented Meta-Languages. In: INTERNATIONAL CONFERENCE ON MODEL DRIVEN ENGINEERING LANGUAGES AND SYSTEMS, 8, 2005, Montego Bay, Jamaica. **Lecture Notes in Computer Science vol. 3713.** Springer Berlin Heidelberg, 2005, p. 264-278.

NETO, J. J. Adaptive Rule-Driven Devices - General Formulation and Case Study. In: IMPLEMENTATION AND APPLICATION OF AUTOMATA INTERNATIONAL CONFERENCE, 6, 2001, Pretória, África do Sul. **Lecture Notes in Computer Science vol. 2494.** Springer Berlin Heidelberg, 2001, p. 234-250.

OBJECT MANAGEMENT GROUP. MDA Guide Version 1.0.1. 2003. Disponível em: <http://www.omg.org/cgi-bin/doc?omg/03-06-01>. Acesso em: 07 fev. 2016.

OBJECT MANAGEMENT GROUP. MOF Model to Text Transformation Language. 2008. Disponível em: <http://www.omg.org/spec/MOFM2T/1.0/PDF>. Acesso em: 07 fev. 2016.

OBJECT MANAGAMENT GROUP. Model Driven Architecture Executive Overview. 2014. Disponível em: <http://www.omg.org/mda/executive_overview.htm>. Acesso em: 07 fev. 2016.

OBJECT MANAGEMENT GROUP. Model Driven Architecture FAQ. 2014. Disponível em: <http://www.omg.org/mda/faq_mda.htm>. Acesso em: 07 fev. 2016.

OBJECT MANAGEMENT GROUP. Meta Object Facility (MOF) Core Specification. 2015. Disponível em: <http://www.omg.org/spec/MOF/2.5/>. Acesso em: 07 fev. 2016.

OBJECT MANAGEMENT GROUP. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. 2015. Disponível em: <http://www.omg.org/spec/QVT>. Acesso em: 07 fev. 2016.

OBJECT MANAGEMENT GROUP. OMG Unified Modeling Language. 2015. Disponível em: <http://www.omg.org/spec/UML/2.5>. Acesso em: 28 fev. 2016.

OBJECT MANAGEMENT GROUP. Object Management Group Web Site. 2016. Disponível em: <http://www.omg.org>. Acesso em: 07 fev. 2016.

OBJECT MANAGEMENT GROUP. OMG's MetaObject Facility Site. 2016. Disponível em: <http://www.omg.org/mof>. Acesso em: 07 fev. 2016.

OBJECT MANAGEMENT GROUP. Semantics of a Foundational Subset for Executable UML Models (fUML), v.1.2.1. 2016. Disponível em: <http://www.omg.org/spec/FUML/1.2.1/>. Acesso em: 28 fev. 2016.

PADILLA, L. A. N. **Transformation of Business Process Models**: A Case Study. 2014. 119 p. Dissertação (Mestrado) - Faculdade de Engenharia, Universidade do Porto, Porto, Portugal, 2014.

PAPOTTI, P.; PRADO, A. F.; SOUZA, W. L.; CIRILO, C. E.; PIRES, L. F. A Quantitative Analysis of Model-Driven Code Generation through Software Experimentation. In: INTERNATIONAL CONFERENCE ON ADVANCED INFORMATION SYSTEMS ENGINEERING, 25, 2013, Valência, Espanha. **Lecture Notes in Computer Science vol. 7908.** Springer Berlin Heidelberg, 2013, p. 321-337.

PASTOR, O.; MOLINA, J.C. **Model-driven architecture in practice**: a software production environment based on conceptual modeling. New York: Springer-Verlag, 2010.

PELEGRINI, E. J. **Códigos Adaptativos e Linguagem para Programação Adaptativa**: Conceitos e Tecnologia. 2009. Dissertação (Mestrado) - Escola Politécnica, Universidade de São Paulo, São Paulo, 2009.

PIERRE, S.; CARIOU, E.; LE GOAER, O.; BARBIER, F. A Family-based Framework for i-DSML Adaptation. In: EUROPEAN CONFERENCE ON MODELING FOUNDATIONS AND APPLICATIONS, 10, 2014, York, Reino Unido. **Lecture Notes in Computer Science vol. 8569.** Springer Berlin Heidelberg, 2014, p. 164-169.

QUINTERO, J.; RUCINQUE, P.; ANAYA, R.; PIEDRAHITA, G. How to face the top MDE adoption problems**.** In: CONFERENCIA LATINOAMERICANA EM INFORMATICA, 38, 2012, Medellin. **Proceedings.** IEEE, 2012, p. 1-10.

RAMOS, M. V.; NETO, J. J.; VEGA, I.S. **Linguagens Formais**: Teoria, Modelagem e Implementação. Porto Alegre: Bookman, 2009.

RODRIGUES; G. A. **Um Simulador para Modelos Descritos em UML.** 2009. 93 p. Dissertação (Mestrado) - Escola Politécnica, Universidade de São Paulo, São Paulo, 2009.

RUSSEL, S.; NORVIG, P. **Inteligência Artificial**: Tradução da Segunda Edição. 2nd ed. Rio de Janeiro: Elsevier, 2004.

RUTLE, A.; ROSSINI, A.; LAMO; Y.; WOLTER; U. A formal approach to the specification and transformation of constraints in MDE. **The Journal of Logic and Algebraic Programming**, v. 81, n. 4, p. 422-457, 2012.

SIPSER, M. **Introdução à Teoria da Computação.** São Paulo: Thomson Learning, 2007.

SHAH, Z. I.; IBRAHIM, R. The Design of Android Metadata based on Reverse Engineering using UML. In: INTERNATIONAL CONFERENCE ON ADVANCED DATA AND INFORMATION ENGINEERING, 1, 2013, Kuala Lumpur, Malásia. **Proceedings of the 1st International Conference on Advanced Data and Information Engineering.** Springer Singapore, 2014, p. 579-586.

SILVA, D. D. **Um Método de Transformação de Modelos em UML e OCL para código Java e SQL.** 2005. 176 p. Tese (Doutorado) - Escola Politécnica, Universidade de São Paulo, São Paulo, 2005.

SILVA, S. R. B. **Software Adaptativo**: Método de Projeto, Representação Gráfica e Implementação de Linguagem de Programação. 2010. 113 p. Dissertação (Mestrado) - Escola Politécnica, Universidade de São Paulo, São Paulo, 2010.

SIQUEIRA, F. L. **Transformação de um Modelo de Empresa em um Modelo de Casos de Uso Seguindo os Conceitos de Engenharia Dirigida por Modelos.** 2011. 274 p. Tese (Doutorado) - Escola Politécnica, Universidade de São Paulo, São Paulo, 2011.

SZVETITS, M.; ZDUN; U. Systematic literature review of the objectives, techniques, kinds, and architectures of models at runtime. **Software & Systems Modeling**, p. 1-39, 2013.

SZYPERSKY, C. A. Import is not Inheritance Why we need both: Modules and Classes. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, 1992, Utrecht, Holanda. **Anais**. Springer Berlin Heidelberg, 1992, p. 19-32.

VARRÓ, D.; PATARICZA, A. VPM: a visual, precise and multilevel metamodeling framework for describing mathematical domains and UML. **Software & Systems Modeling**, v. 2, n. 3, p. 187-210, 2003.

WORLD WIDE WEB CONSORTIUM (W3C). XSL Transformations (XSLT). 1999. Disponível em: <http://www.w3.org/TR/xslt>. Acesso em: 07 fev. 2016.

WAGNER, F.; SCHMUKI, R.; WAGNER, T.; WOLSTENHOLME, P. **Modeling Software with Finite State Machines**: A Pratical Approach. Boca Raton: Auerbach Publications, 2006.

WEBML TEAM. WebML Web Page. 2013. Disponível em: <http://www.webml.org>. Acesso em: 07 fev. 2016.

WIRTH, N. A brief history of software engineering. **IEEE Annals of the History of Computing**, v. 1, n. 3, p. 32-39, 2008**.**

WU, Y.; HERNANDEZ, F.; ORTEGA, F.; CLARKE, P. J.; FRANCE, R. Measuring the Effort for Creating and Using Domain Specific Models. In: WORKSHOP ON DOMAIN-SPECIFIC MODELING, 10, 2010, Reno/Tahoe, EUA. **Proceedings of the 10th Workshop on Domain-Specific Modeling (DSM'10)**. Helsinki: Aaalto School of Economics, 2010, p. 14.

XIONG, Y.; LIU, D.; HU, Z.; ZHAO, H.; TAKEICHI, M.; MEI, H. Towards automatic model synchronization from model transformation. In: IEEE/ACM INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING, 22, 2007, Atlanta, EUA. **Proceedings of the twenty-second IEEE/ACM international conference on automated software engineering**. ACM, 2007, pp. 164-173.

YANG, D.; LIU, M.; WANG, S. Object-oriented methodology meets MDA software paradigm. In: IEEE INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING AND SERVICE SCIENCE, 3, 2012, Pequim, China. **Anais.** IEEE, 2012, p. 208-211.

YANG, D.; LIU, M.; LI, B. Modeling Business for MDA Software Paradigm. **International Journal of Database Theory and Application**, v. 7, n. 4, p. 155-168, 2014.

ZAN, T.; PACHECO, H.; HU, Z. Writing bidirectional model transformations as intentional updates. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 36, 2014, Hyderabad, Índia. **Companion Proceedings of the 36th International Conference on Software Engineering.** Nova York: ACM, 2014, p. 488-491.

APÊNDICE A – Métodos e Linguagens de Transformação com o SBMM

Este apêndice apresenta um resumo sobre três técnicas ou linguagens de transformação de modelos, ou seja, formas de descrever funções de mapeamento. As duas primeiras se referem a transformações do tipo modelo-para-modelo: gramáticas de grafo triplas e *Query / View / Transformation* (QVT). A terceira delas é o *MOF MODEL TO TEXT TRANSFORMATION* (MOFM2T), aplicando-se a transformações do tipo model-para-código. Como essas técnicas e linguagens já existiam antes, não tendo sido concebidas especificamente para o SBMM, então apresenta-se também considerações sobre sua aplicação com o formalismo aqui proposto.

A.1 GRAMÁTICAS DE GRAFO TRIPLAS

Um método para descrever funções de mapeamento do tipo modelo-para-modelo são as gramáticas de grafo triplas (KÖNIGS, 2005). Gramáticas de grafo proveem um formalismo útil para descrever manipulações estruturais em dados multidimensionais. Gramáticas de grafo triplas consistem em um tipo especial que serve como uma técnica para definir correspondência entre dois metamodelos de uma maneira declarativa.

A ideia básica é que modelos podem ser enxergados como grafos que evoluem através da aplicação de regras. O grafo que declara essas correspondências e regras pode ser separado em três subgrafos, justificando o nome da técnica. Dois desses subgrafos correspondem aos metamodelos origem e alvo dos modelos que se deseja transformar, enquanto o terceiro serve como um metamodelo que define elementos que mantêm registro das correspondências entre modelos dos outros dois.

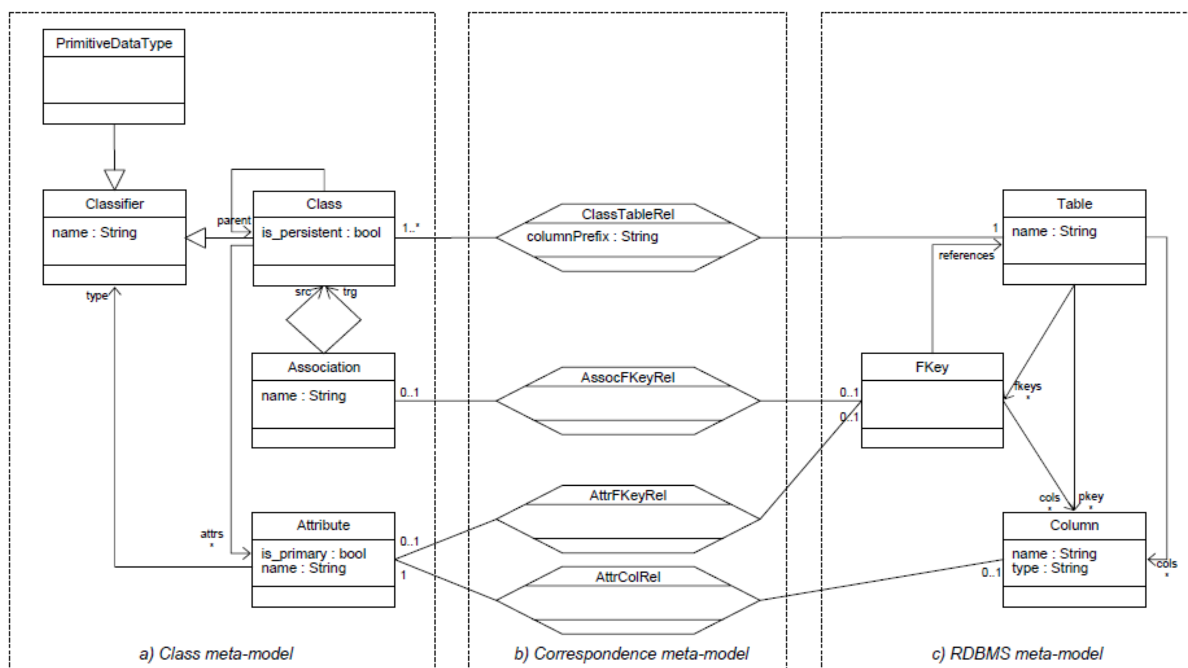
O exemplo da Figura 39 ilustra, à esquerda, um metamodelo simplificado para diagramas de classes UML. O metamodelo está apresentado na notação gráfica do MOF, já que a técnica apresentada por Königs (2005) é baseada em metamodelos MOF. À direita, é apresentado um metamodelo simplificado para bancos de dados relacionais, que consiste em tabelas que possuem colunas e chaves estrangeiras. Adicionalmente, cada tabela contém um conjunto de colunas que constituem suas

chaves primárias. O grafo central apresenta o metamodelo que determina as correspondências entre elementos de diagrama de classes e de bancos de dados relacionais. O nó *ClassTableRel* estabelece que um elemento do tipo *Class* corresponde a um elemento do tipo *Table*. Por outro lado, um elemento do tipo *Table* corresponde a pelo menos um do tipo *Class*. O nó *AttrColRef* liga um *Attribute* com no máximo um *Column*, enquanto cada *Column* está ligado a exatamente um *Attribute* e até no máximo um *FKey*. No outro sentido, um *FKey* e no máximo um *Attribute*.

Uma gramática de grafo tripla provê um conjunto de regras, as quais descrevem como os modelos evoluem simultaneamente. A Figura 40 mostra dois exemplos dessas regras. A primeira delas significa que toda vez que um novo elemento do tipo *Class* (*c*) é adicionado no primeiro modelo, então um novo elemento do tipo *Table* (*t*) deve ser inserido no outro modelo. Adicionalmente, uma ligação de correspondência do tipo *ClassTableRel* é instanciada, servindo para manter rastreabilidade entre os elementos relacionados *c* e *t*. A restrição determina que esta regra é apenas aplicável se o valor da propriedade *is_primary* de *c* é igual a *true*. Além disso, o nome de *c* deve combinar com o nome de *t*. Finalmente, *low* descreve a prioridade desta regra. A prioridade de uma regra é usada pelo mecanismo de aplicação da regra para determinar a ordem de aplicação.

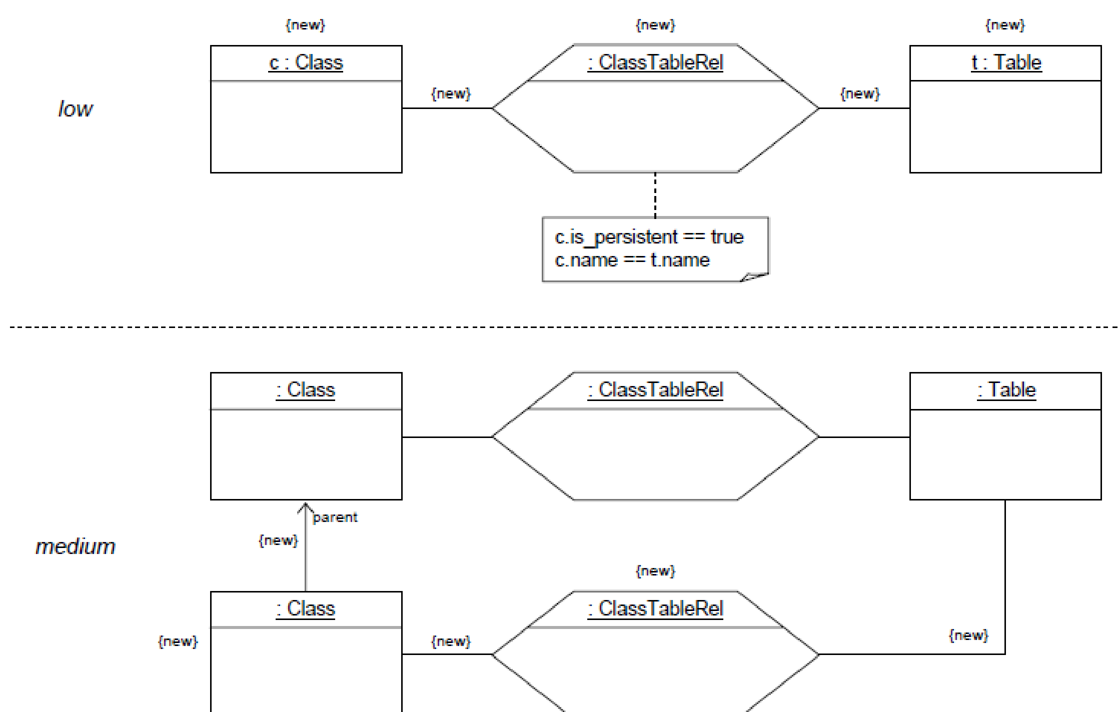
A segunda regra da Figura 40 indica que sempre que um novo elemento do tipo *Class* (*c*) é inserido como subclasse de outro elemento de mesmo tipo, então *c* será associada ao elemento do tipo *Table* já existente por uma nova ligação de correspondência do tipo *ClassTableRel*. O elemento tipo *Table* pré-existente já deve ter sido ligado ao elemento tipo *Class* correspondente à superclasse antes. A restrição indica que esta regra só é aplicável se *c* é persistente. A regra tem prioridade *medium*.

Figura 39 – Exemplo de gramática de grafo tripla



Fonte: Königs (2005)

Figura 40 – Exemplo de regra de gramática de grafo tripla



Fonte: Königs (2005)

Não é objetivo deste trabalho apresentar todos os detalhes sobre a técnica das gramáticas de grafo triplas. Mais detalhes podem ser consultados em Königs (2005). Do ponto de vista de sua aplicação a modelos escritos em SBMM, os nós dos grafos correspondentes aos metamodelos origem e alvo são as metaclasses. As arestas ou são generalizações ou propriedades de metaclasses cujos tipos alvos são outras metaclasses. Königs (2005) também apresenta um algoritmo para aplicação de regras, que permitiria dar base à construção de um motor de transformação do tipo modelo-para-modelo em uma ferramenta para MDE.

A.2 QUERY / VIEW / TRANSFORMATION (QVT)

Query / View / Transformation (QVT) é uma linguagem de transformação de modelos estabelecida como padrão OMG (OMG, 2015b). Foi projetada para transformar modelos conformes a metamodelos baseados em MOF. Seu objetivo é transformar modelos de um determinado metamodelo para modelos de outro metamodelo (ou eventualmente do mesmo), tais como converter um diagrama de classes UML em um diagrama Entidade-Relacionamento de um banco de dados relacional. Vai de encontro à definição de função de mapeamento do tipo modelo-para-modelo apresentada pela eq. (46). Na verdade, o QVT é composto por três linguagens: duas de natureza declarativa e uma de natureza imperativa. A parte declarativa é composta pelas linguagens *Relations* e *Core*, que formam um arcabouço para a semântica de execução da parte imperativa, a qual é representada pela linguagem *Operational Mappings*.

A camada *Relations* suporta a identificação de padrões complexos de objetos e criação de novas instâncias através de *templates*. Basicamente, uma transformação é definida através de um conjunto de relações definidas sobre diferentes metamodelos (eventualmente do mesmo). Cada relação estabelece como as instâncias de um modelo devem combinar (*match*) com instâncias de outro modelo, e isso é feito através de *templates*. O interpretador QVT então, quando acionado, varre o modelo de origem buscando por esses padrões e procura pelas combinações necessárias no modelo alvo. Caso essas combinações não sejam encontradas, o modelo alvo é alterado através de criação de novas instâncias, exclusões ou modificações de forma a garantir que a relação passe a ser válida.

Ligações de rastreabilidade entre os elementos dos modelos envolvidos em uma transformação são criadas implicitamente como um modo de documentar os passos da execução da transformação. Além da execução propriamente dita, uma relação pode ser utilizada apenas para garantir que o outro modelo está de acordo com o primeiro (LACHTERMACHER et al., 2008).

A camada *Core* é composta por um metamodelo e uma pequena linguagem definida pela utilização mínima de extensões sobre EMOF e OCL. As classes de rastreabilidade são explicitamente definidas através de metamodelos MOF. Esta camada é responsável por casar padrões de acordo com um conjunto de variáveis e condições. Ela pode ser diretamente implementada ou utilizada apenas como referência para a semântica de *Relations* (LACHTERMACHER et al., 2008). Em outras palavras, *Core* seria uma linguagem de mais baixo nível que *Relations*, com a mesma capacidade, mas consequentemente requerendo mais código para se obter o mesmo resultado. Ou seja, elas carregam a mesma semântica, porém em dois níveis de abstração diferentes. A especificação do QVT (OMG, 2015b) provê um mapeamento *Relations* para *Core*.

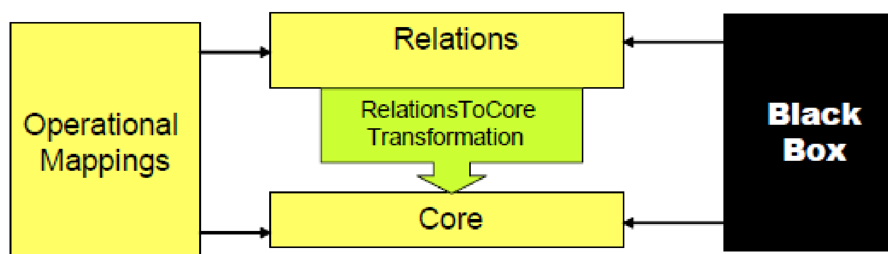
Uma analogia com a arquitetura da linguagem Java pode ser feita (OMG, 2015b). *Core* seria como o *byte code* e sua semântica seria como a especificação do comportamento da máquina virtual Java. A linguagem *Relations* tem o papel da linguagem Java em si, e o mapeamento padrão *Relations* para *Core* seria como a especificação de um compilador Java que produz *byte code*.

Em adição à parte declarativa, o QVT estabelece dois mecanismos para invocar implementações imperativas de transformação a partir de *Relations* e *Core*: uma linguagem padrão denominada *Operational Mappings* e também implementações caixa-preta de operações MOF, ou *Black-box MOF Operations*, em linguagens externas ao QVT. Esses mecanismos podem ser usados quando é difícil prover uma especificação puramente declarativa com Relations e Core.

A Figura 41 ilustra as relações entre as linguagens ou camadas do QVT.

Nesta subseção são apresentados alguns aspectos da linguagem *Relations* através de exemplos, bem como considerações sobre sua aplicação para modelos baseados em SBMM.

Figura 41 – Relações entre as linguagens ou camadas do QVT



Fonte: OMG (2015b)

Em *Relations*, uma função de mapeamento entre modelos candidatos é especificada por um conjunto de relações que precisam ser verdadeiras. Um modelo candidato é qualquer um que possua relação de conformidade com o metamodelo previsto pela função de mapeamento. Dentro da definição da função de mapeamento, os modelos candidatos são argumentos nomeados para serem referenciados pelas expressões permitidas pela linguagem. Por exemplo, a declaração de função de mapeamento do Quadro 14 estabelece dois modelos de entrada, um correspondente ao metamodelo *SimpleUML* e outro correspondente ao metamodelo *SimpleRDBMS*.

Quadro 14 – Exemplo de declaração de função de mapeamento em QVT

```
1  transformation umlRdbms (uml: SimpleUML, rdbms: SimpleRDBMS)
```

Fonte: OMG (2015b)

O metamodelo *SimpleUML* representa um metamodelo simplificado do diagrama de classes da UML, enquanto *SimpleRDBMS* representa um metamodelo simplificado de bancos de dados relacionais.

Os modelos *uml* e *rdbms* são os modelos candidatos. Uma transformação pode ser invocada tanto para checar a consistência entre os dois modelos como para modificar um dos modelos para garantir a consistência com o outro. Quando invocada no modo de modificação, deve ser especificada uma direção particular através da seleção de um dos modelos candidatos como alvo. O modelo alvo pode estar vazio ou pode conter instâncias pré-existentes que devem ser devidamente consideradas pela transformação. Essa capacidade está de acordo com a modelagem das funções de mapeamento do tipo modelo-para-modelo apresentada na eq. (48), em que a versão anterior do modelo alvo também é entrada para a

função de mapeamento de modo a preservar, na medida do possível, as alterações feitas manualmente pelo desenvolvedor.

A execução da transformação procede primeiramente pela checagem das relações que são verdadeiras e, para aquelas que não o são, prossegue-se com a tentativa de torná-las assim através da criação, exclusão ou modificação de instâncias no modelo alvo.

Uma relação declara restrições que precisam ser satisfeitas pelas instâncias dos modelos candidatos, e é definida por dois ou mais domínios. Um domínio é uma variável que pode ser casada com um elemento de um modelo de acordo com um padrão (*pattern*). Este, por sua vez, pode ser visto como um conjunto de variáveis e um conjunto de restrições que determinam um *template* para instâncias e suas propriedades que precisam ser localizadas, modificadas ou criadas em um modelo candidato para satisfazer a relação. Casamento de padrões e criação de objetos usando esses padrões são, portanto, a base de funcionamento para um interpretador QVT. No exemplo do Quadro 15, dois domínios são declarados em uma relação para combinar elementos dos modelos *uml* e *rdbms*, respectivamente. Cada domínio especifica um simples padrão: um pacote (*package*) com um nome e um banco relacional também com um nome, estando ambas as propriedades *name* relacionadas pela mesma variável *pn*, o que implica que devem ter o mesmo valor.

Quadro 15 – Exemplo de relação em QVT

```
1 relation PackageToSchema
2    /* mapeia cada package em um schema */
3    {
4       domain uml p:Package {name = pn}
5       domain rdbms s:Schema {name = pn}
6    }
```

Fonte: OMG (2015b)

A cláusula *when* especifica condições sob as quais a relação precisa ser válida. No próximo exemplo, a relação *ClassToTable* só precisa ser válida quando *PackageToSchema* também é válida entre o pacote contendo a classe e o banco de dados relacional contendo a tabela. A cláusula *where* especifica condições que devem ser satisfeitas por todos os elementos que participam da relação, e podem

restringir quaisquer variáveis na relação e seus domínios. Portanto, sempre que a relação *ClassToTable* é válida, então *AttributeToColumn* também deve ser válida, e o processador QVT deve garanti-la também. O exemplo do Quadro 16 utiliza essas cláusulas.

Quadro 16 – Exemplo de relação em QVT com cláusulas when e where

```
1  relation ClassToTable
2     /* mapeia cada classe persistente em uma tabela */
3     {
4        domain uml c:Class {
5           namespace = p:Package {},
6           kind = 'Persistent',
7           name = cn
8        }
9        domain rdbms t:Table {
10          schema = s:Schema {},
11          name = cn,
12          column = cl:Column {
13             name = cn + '_tid',
14             type = 'NUMBER'},
15          primaryKey = k:PrimaryKey {
16             name = cn+'_pk',
17             column = cl }
18       }
19       when {
20          PackageToSchema(p, s);
21       }
22       where {
23          AttributeToColumn(c, t);
24       }
25    }
```

Fonte: OMG (2015b)

Uma transformação pode conter dois tipos de relações: nível topo (*top-level*) e nível não-topo (*non-top-level*). A execução da transformação requer que todas as relações nível topo sejam válidas, enquanto as não-topo são requeridas apenas se invocadas direta ou transitivamente pela cláusula *where* de outra relação. Uma relação nível

topo possui a palavra-chave *top* para distingui-la sintaticamente. No exemplo do Quadro 17, *PackageToSchema* e *ClassToTable* são relações nível topo, enquanto *AttributeToColumn* é não-topo.

Quadro 17 – Exemplo de relações topo e não-topo

```
1  transformation umlRdbms (uml: SimpleUML, rdbms: SimpleRDBMS)
2     {
3        top relation PackageToSchema {…}
4        top relation ClassToTable {…}
5        relation AttributeToColumn {…}
6     }
```

Fonte: OMG (2015b)

Cada domínio pode ser marcado com as palavras-chave *checkonly* (apenas checagem) ou *enforced* (garantido ou forçado). Quando o domínio correspondente ao modelo alvo está marcado com *checkonly*, então ele é apenas checado para verificar se existe uma combinação válida que satisfaz a relação. Quando a transformação é executada em direção ao modelo alvo cujo domínio está marcado como *enforce*, então o modelo alvo é modificado para satisfazer a relação quando ela não é verdadeira. No exemplo do Quadro 18, o domínio do modelo *uml* está marcado com *checkonly* e o domínio do modelo *rdbms* com *enforce*.

Quadro 18 – Exemplo de domínios apenas checagem e forçados

```
1  relation PackageToSchema
2     /* mapeia cada package a um schema */
3     {
4        checkonly domain uml p:Package {name = pn}
5        enforce domain rdbms s:Schema {name = pn}
6     }
```

Fonte: OMG (2015b)

Se o modelo alvo na execução da transformação for *uml* e existir um banco de dados relacional em *rdbms* para o qual não há um pacote correspondente de mesmo

nome em *uml*, então simplesmente é reportada uma inconsistência. Um pacote não é criado, pois o modelo *uml* não tem a palavra-chave *enforce*. Entretanto, se o modelo alvo for *rdbms*, então para cada pacote no modelo *uml* a relação primeiro verifica se existe um banco de dados relacional com o mesmo nome em *rdbms*. Se não existir, então um novo banco relacional é criado neste modelo com o mesmo nome do pacote. Em contrapartida, se a transformação for executada com o mesmo modelo alvo e existir um banco de dados relacional em rdbms que não tenha um pacote de mesmo nome em *uml*, então esse banco relacional seria excluído para assim garantir consistência com *uml*.

A combinação dos padrões associados aos domínios é um dos pontos centrais do QVT. Para entender melhor as expressões que formam esses *templates*, considere-se novamente o exemplo da relação *ClassToTable*. Uma das expressões de *template*, associada ao domínio do modelo *uml*, é apresentada no Quadro 19.

Quadro 19 – Relação ClassToTable

```
1  c:Class
2     {
3        namespace = p:Package {},
4        kind = 'Persistent',
5        name = cn
6     }
```

Fonte: OMG (2015b)

Uma combinação de expressão de *template* resulta em uma associação (*binding*) de instâncias do modelo candidato com variáveis declaradas no domínio. Uma combinação pode ser realizada em um contexto em que algumas variáveis já podem ter sido associadas a elementos de modelo (p. ex., resultante de uma avaliação da cláusula *when*). Neste caso, apenas variáveis ainda livres podem ser associadas a outros elementos.

Por exemplo, no caso acima, a combinação de padrões vai associar todas as variáveis na expressão de *template* (*c*, *p* e *cn*), iniciando pela variável raiz do domínio *c* do tipo (metaclasse) *Class*. Neste exemplo, a variável *p* já teria uma associação resultante da avaliação da expressão *PackageToSchema(p,s)* na cláusula *when*. A combinação procede pela filtragem de todas as instâncias do tipo

Class no modelo *uml*, eliminando qualquer um que não possua o mesmo valor para suas propriedades conforme especificado pela expressão de *template*. Neste exemplo, qualquer instância de *Class* cujo valor para a propriedade *kind* não seja *Persistent* será eliminada.

Para as propriedades que são comparadas com variáveis tais como em *name = cn*, dois casos podem acontecer. Se a variável *cn* já possuir uma associação (*binding*) de valor, então qualquer classe que não tenha o mesmo valor para sua propriedade *name* é eliminada. Se a variável *cn* ainda estiver livre, como no exemplo, então ela receberá uma associação ao valor da propriedade *name* para todas as classes que não foram eliminadas no filtro devido a diferenças em outra comparação de propriedades. O valor de *cn* será ou usado em outro domínio, ou pode receber restrições adicionais nas expressões da cláusula *where*.

A combinação então prossegue para propriedades cujos valores são comparados a expressões de *template* aninhadas. Por exemplo, o padrão *namespace = p:Package{}* vai combinar apenas aquelas classes cuja propriedade *namespace* possuir uma referência não nula para um pacote. Ao mesmo tempo, a variável *p* será associada para se referir ao pacote. Entretanto, neste exemplo, *p* já possui associação proveniente da cláusula *when*, e por isso o padrão vai apenas combinar com aquelas classes cuja propriedade *namespace* referenciar o mesmo pacote que já está associado a *p*. Um número arbitrário de aninhamentos em expressões de *template* é permitido. As combinações e associações de variáveis prosseguem recursivamente até que exista um conjunto de tuplas de valores correspondentes às variáveis do domínio e sua expressão de *template*. Por exemplo, as três variáveis *c*, *p* e *cn* formam uma tripla e cada combinação válida resulta em um único trio de valores representando a associação (*binding*).

Maiores detalhes sobre a linguagem *Relations*, seu funcionamento e até mesmo diretrizes para implementação de um interpretador podem ser encontradas em OMG (2015b). Esta mesma especificação também detalha as linguagens *Core* e *Operational Mappings*.

Observa-se uma similaridade grande entre os conceitos fundamentais de *Relations* e as gramáticas de grafo triplas. Na verdade, o metamodelo de correspondência (vide exemplo da Figura 39) corresponde a uma descrição de função de mapeamento

(*transformation*) de *Relations*, sendo o formalismo base para a concepção desta linguagem.

Embora tenha sido projetada para utilização entre modelos baseados em metamodelos MOF, o QVT poderia ser usado para modelos baseados em SBMM pois existe uma correspondência direta entre os conceitos que aparecem. Os tipos de modelos são os metamodelos, e os tipos de variáveis são as metaclasses ou enumerações. Constantes e expressões com literais do QVT podem ser mapeadas diretamente em valores permitidos de enumerações.

Como apenas um modelo candidato pode ser escolhido como alvo para a execução de uma transformação, então um interpretador QVT *Relations* é um bom candidato para implementar o motor te_model_to_model da eq. (50) e, consequentemente, o universo de descritores U_D é composto pelo conjunto de cadeias pertencentes à linguagem gerada pela gramática do QVT *Relations* apresentada em sua especificação (OMG, 2015b). Sendo assim, uma ferramenta MDE baseada no formalismo SBMM pode utilizar ou embutir um interpretador QVT *Relations* ou *Core* para executar transformações do tipo modelo-para-modelo.

A.3 MOF MODEL TO TEXT TRANSFORMATION (MOFM2T)

MOF Model To Text Transformation Language (MOFM2T) é um padrão do OMG (OMG, 2008) que provê uma linguagem para traduzir um modelo para artefatos baseados em texto, tais como código-fonte, especificações de instalação, relatórios, documentos, etc. Essencialmente, esse padrão permite transformar um modelo em uma representação textual linearizada, ou seja, uma cadeia de caracteres. Vai de encontro à definição de função de mapeamento do tipo modelo-para-código apresentada pela eq. (47).

O padrão MOFM2T se enquadra na abordagem de transformação de modelos baseada em arquétipo, pois consiste em *templates* de texto com marcadores para extrair dados dos modelos. Esses marcadores são basicamente expressões especificadas sobre entidades do metamodelo (em geral, metaclasses e propriedades) com *queries* servindo de mecanismo primário para selecionar e extrair valores do modelo. Esses valores são então convertidos em fragmentos de texto

usando uma linguagem de expressões aumentada por uma biblioteca de manipulação de *strings*.

Nesta subseção são apresentados os conceitos básicos sobre o MOFM2T e também considerações sobre sua aplicação no contexto do SBMM.

O exemplo do Quadro 20 gera uma definição de classe Java para uma classe UML.

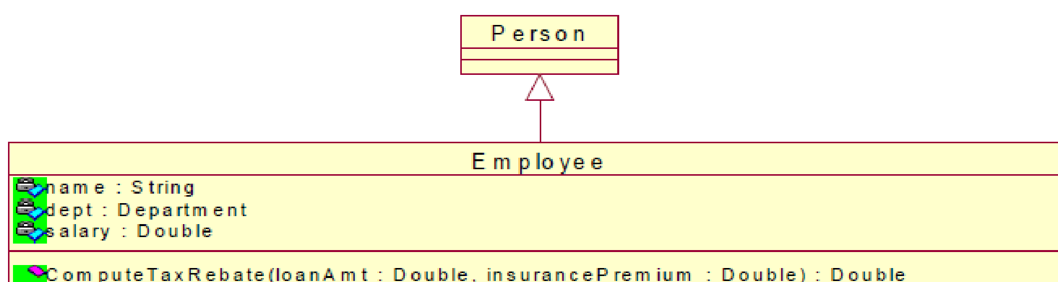
Considere-se o diagrama de classes UML da Figura 42. O texto apresentado no Quadro 21 é gerado para a classe *Employee* de acordo com o *template* apresentado no Quadro 20.

Quadro 20 – Exemplo de template gerador de classes Java em MOFM2T

```
1  [template public classToJava(c: Class)]
2  class [c.name/]
3  {
4     // Constructor
5     [c.name/]
6     {
7     }
8  }
9  [/template]
```

Fonte: OMG (2008)

Figura 42 – Exemplo de diagrama de classes UML para conversão em código-fonte Java



Fonte: OMG (2008)

Quadro 21 – Exemplo de classe Java gerada automaticamente

```
1  class Employee
2  {
3     // Constructor
4     Employee()
5     {
6     }
7  }
```

Fonte: OMG (2008)

A saída produzida pela execução da transformação preserva a ordem de caracteres, identação, espaços em branco e outras características do *template*.

Um *template* pode chamar outros *templates*, o que equivale à colocação do texto produzido pelo *template* chamado no local do marcador correspondente à chamada. O exemplo do Quadro 22 apresenta a chamada de *attributeToJava*.

Quadro 22 – Exemplo de chamada de template em MOFM2T

```
1  [template public classToJava(c : Class)]
2  class [c.name/]
3  {
4     // Attribute declarations
5     [attributeToJava(c.attributes)/]
6  
7     // Constructor
8     [c.name/]()
9     {
10    }
11 }
12 [/template]
13 
14 [template public attributeToJava(a : Attribute)]
15 [a.type.name/] [a.name/];
16 [/template]
```

Fonte: OMG (2008)

O *template classToJava* chama o *template attributeToJava* para cada atributo da classe. Isso ocorre porque a propriedade *attributes* da metaclasse *Class* tem multiplicidade maior que 1. Nesses casos, a chamada ao *template* ocorre n vezes conforme os n valores presentes no *slot* da instância correspondente à propriedade. A saída do Quadro 23 é produzida, considerando ainda a classe *Employee* da Figura 42.

Quadro 23 – Exemplo de classe Java gerada automaticamente com chamada de template

```
1  class Employee
2  {
3     // Attribute declarations
4     String name;
5     Department dept;
6     Double salary;
7  
8     // Constructor
9     Employee()
10    {
11    }
12 }
```

Fonte: OMG (2008)

Uma alternativa em relação à chamada do *template* para gerar o código relativo à declaração dos atributos é a utilização de um bloco *for*. Esse tipo de bloco é capaz de iterar sobre a coleção de valores presentes no *slot* de uma propriedade com multiplicidade maior que 1. Em certos casos, pode facilitar a leitura e o entendimento do *template*. O exemplo do Quadro 24 gera a mesma saída que o *template* apresentado no Quadro 22.

O bloco *for* declara uma variável de laço *a* do tipo *Attribute*, que itera sobre todas as instâncias presentes no *slot* correspondente à propriedade *attributes* da instância *c*.

Um *template* pode conter uma condição que decide se ele pode ser chamado, a qual é denominada guarda. Por exemplo, no Quadro 25 *classToJava* é chamado apenas se a classe passada como argumento é concreta. Essa condição está definida após o ponto de interrogação da linha 1.

Quadro 24 – Exemplo de utilização do laço for em MOFM2T

```
1  [template public classToJava(c : Class)]
2  class [c.name/]
3  {
4     // Attribute declarations
5     [for(a : Attribute | c.attributes)]
6     [a.type.name/] [a.name/];
7     [/for]
8  
9     // Constructor
10    [c.name/]()
11    {
12    }
13 }
14 [/template]
```

Fonte: OMG (2008)

Quadro 25 – Exemplo de guarda em MOFM2T

```
1  [template public classToJava(c : Class) ? (c.isAbstract = false)]
2  class [c.name/]
3  {
4     // Attribute declarations
5     [attributeToJava(c.attribute)/]
6  
7     // Constructor
8     [c.name/]()
9     {
10    }
11 }
12 [/template]
```

Fonte: OMG (2008)

Um dos mais importantes recursos do MOFM2T é a capacidade de definir trechos de texto protegidos. Esses trechos são preservados e não sobrescritos por execuções subsequentes da transformação. Dessa forma, o desenvolvedor pode inserir código manualmente nas áreas protegidas com a garantia de que uma próxima execução da transformação não vai sobrescrevê-las. Por exemplo, o *template* do Quadro 26

determina uma área protegida chamada *_user_code* dentro da implementação do método construtor, precedida pelo identificador da classe no modelo. A área protegida é delimitada pela abertura e fechamento do marcador *protected*.

Quadro 26 – Exemplo de área protegida em template MOFM2T

```
1  [template public classToJava(c : Class)]
2  class [c.name/]
3  {
4     // Constructor
5     [c.name/]()
6     {
7     [protected(c.id() + '_user_code')]
8     ; user code
9     [/protected]
10    }
11 }
12 [/template]
```

Fonte: OMG (2008)

O nome da área protegida serve para gerar, no texto de saída, delimitadores que claramente a identificam para posterior recuperação do conteúdo inserido pelo desenvolvedor. Uma vez que esses delimitadores são específicos da linguagem alvo, correspondendo a comentários para que não afetem a compilação do programa, eles não são definidos pelo padrão MOFM2T. Uma ferramenta que implementa um motor de transformação do tipo modelo-para-código baseado neste padrão é responsável por produzir os delimitadores corretos para a linguagem alvo desejada.

Esse recurso de definir áreas protegidas torna implícito que a saída gerada na execução anterior da transformação, com as modificações do desenvolvedor, também é uma entrada para a execução da transformação. Isso está de acordo com a definição do motor te_model_to_code apresentado pela eq. (51).

O MOFM2T oferece a capacidade para os *templates* direcionarem sua saída para arquivos através do bloco *file*. Esse bloco especifica o arquivo para o qual o texto gerado deve ser enviado. Possui três parâmetros: uma *Uniform Resource Indicator*

(URI) que denota o nome do arquivo, um valor booleano que indica se o arquivo deve ser aberto no modo de continuidade de gravação, ou *append* (verdadeiro), ou sobrescrito (falso), e um identificador único opcional, o qual possibilita que a ferramenta que implementa a transformação encontre um arquivo que foi gerado em uma sessão prévia mesmo quando o elemento do modelo foi renomeado (e o mesmo foi usado na URI que determina o nome do arquivo). O Quadro 27 apresenta um exemplo.

O *template* do Quadro 27, além de produzir a saída principal em um arquivo com o nome da classe seguido da extensão .java (definido na linha 2), também acumula em um arquivo de registro *(log*) indicações das classes processadas (linha 3).

Quadro 27 – Exemplo de bloco file em MOFM2T

```
1  [template public classToJava(c : Class)]
2  [file (‘file:\\’+c.name+’.java’, false, c.id + ‘impl’)]
3  [file('log.log', true)]processing [class.name/][/file]
4  class [c.name/]
5  {
6     // Constructor
7     [c.name/]()
8     {
9     }
10 }
11 [/file]
12 [/template]
```

Fonte: OMG (2008)

O MOFM2T apresenta diversos outros recursos. Dentre eles, destaca-se a possibilidade de definir *queries*, através do marcador *query*, que servem para obter elementos ou conjuntos de elementos específicos do modelo de entrada, tais como todas as operações das superclasses abstratas de uma classe. Também se destaca o bloco *if*, que permite gerar um texto se determinada condição é satisfeita. Não é objetivo deste trabalho apresentar uma especificação completa do MOFM2T. Maiores detalhes podem ser encontrados em OMG (2008).

Especificações grandes de transformações podem ser estruturadas em módulos. Um módulo consiste em um conjunto de *templates* e *queries* e possui uma parte pública e outra privada. A parte pública expõe *templates* e *queries* que podem ser chamados por outros módulos. Um módulo pode apresentar uma dependência de importação de outros módulos. Isso permite que o módulo que faz a importação possa chamar *templates* e *queries* públicos dos módulos importados.

A possibilidade de um mesmo *template* MOFM2T direcionar saída para mais de um arquivo implica em uma modificação na definição do motor te_model_to_code apresentado pela eq. (51). Ao invés de a função produzir uma única cadeia de saída, ela passa a ser capaz de produzir uma tupla de cadeias. O domínio da função também deve ser alterado para permitir uma tupla de cadeias de entrada, e não uma só. Com isso, essa definição passa a ser uma descrição mais precisa de um motor MOFM2T geral, se desejado.

Um interpretador MOFM2T é um bom candidato para implementar o motor te_model_to_code e, consequentemente, o universo de descritores U_D consiste no universo de *templates* possíveis, que são as cadeias pertencentes à linguagem definida pela gramática do MOFM2T apresentada em sua especificação (OMG, 2008). Sendo assim, uma ferramenta MDE baseada no formalismo SBMM pode utilizar ou embarcar um interpretador MOFM2T para executar transformações do tipo modelo-para-código. Caso múltiplos arquivos de saída precisem ser gerados, o que é comum em projetos de software, múltiplos *templates* e múltiplas chamadas podem ser utilizadas.

APÊNDICE B – Autômatos dos Analisadores Sintáticos e Interpretadores

Os analisadores sintáticos e interpretadores MOFM2T e de sentenças em lógica de primeira ordem da ferramenta SBMMTool foram construídos com base em autômatos de pilha estruturados adaptativos (RAMOS; NETO; VEGA, 2009). Por simplificação, os autômatos do analisador léxico são omitidos aqui, já que muitos dos *tokens* reconhecidos pelo mesmo são triviais e comuns nas linguagens de programação. Esses *tokens* são apresentados no Quadro 28.

Quadro 28 – Tokens do analisador léxico MOFM2T e LPO

Token	Descrição	Exemplos ou Símbolos
Identifier	Identificador	Nome de instância, metaclasse, etc.
Integer	Constante de número inteiro	0, 1, 242, 320, 192
Double	Constante de número em ponto flutuante	0.5, 83.92
String	Constante string com aspas simples	‘Cadeia de texto’
BracketOpen	Abre colchetes	[
BracketClose	Fecha colchetes	]
BracketOpenWithSlash	Abre colchetes com barra	[/
BracketCloseWithSlash	Fecha colchetes com barra	/]
ParenthesisOpen	Abre parêntesis	(
ParenthesisClose	Fecha parêntesis	)
BraceOpen	Abre chaves	{
BraceClose	Fecha chaves	}
Colon	Dois-pontos	:
Comma	Vírgula	,
Pipe	Pipe	|
Dot	Ponto	.
NumberSymbol	Sustenido	#
BinaryExprOperator	Operador binário	=, <, <=, >, >=, <>, is, is_direct, or, xor, and, implies, +, -, *, /, %, ^
UnaryExprOperator	Operator unário	not
Reserved_If	Palavra reservada if	if
Reserved_For	Palavra reservada for	for
Reserved_Value	Palavra reservada value	value
Reserved_File	Palavra reservada file	file
Reserved_Template	Palavra reservada template	template
Reserved_Public	Palavra reservada public	public
Reserved_Private	Palavra reservada private	private
Reserved_Top	Palavra reservada top	top
Reserved_Call	Palavra reservada call	call
Reserved_Protected	Palavra reservada protected	protected
Reserved_Asc	Palavra reservada asc	asc
Reserved_Desc	Palavra reservada desc	desc
Reserved_ForAll	Palavra reservada forall	forall
Reserved_Exists	Palavra reservada exists	exists
OutputText	Texto para a cadeia de saída	program MyProgram, begin

Fonte: autor

Os autômatos da camada sintática, por sua vez, podem ser executados em modo de análise sintática pura ou de interpretação. No segundo caso, além de efetuar a análise sintática, as transições dos autômatos já processam os efeitos desejados para cada elemento da linguagem conforme a semântica do MOFM2T. É nesse caso que a adaptatividade foi utilizada para resolver problemas de iterações. Por exemplo, em um laço do tipo `[for(...)]conteúdo[/for]`, processa-se o trecho `conteúdo` quantas vezes forem necessárias para iterar sobre os elementos definidos nos parâmetros do laço. Enquanto isso, uma transição que parte do fim do laço em direção o início do laço, encaminhando para a próxima iteração, existe no autômato. Sempre que essa transição é executada, o ponteiro de leitura do analisador léxico também é reposicionado de volta ao início do trecho `conteúdo`. Quando o laço termina, a camada adaptativa do interpretador remove essa transição e adiciona outra transição para a continuidade do processamento após o laço `for`.

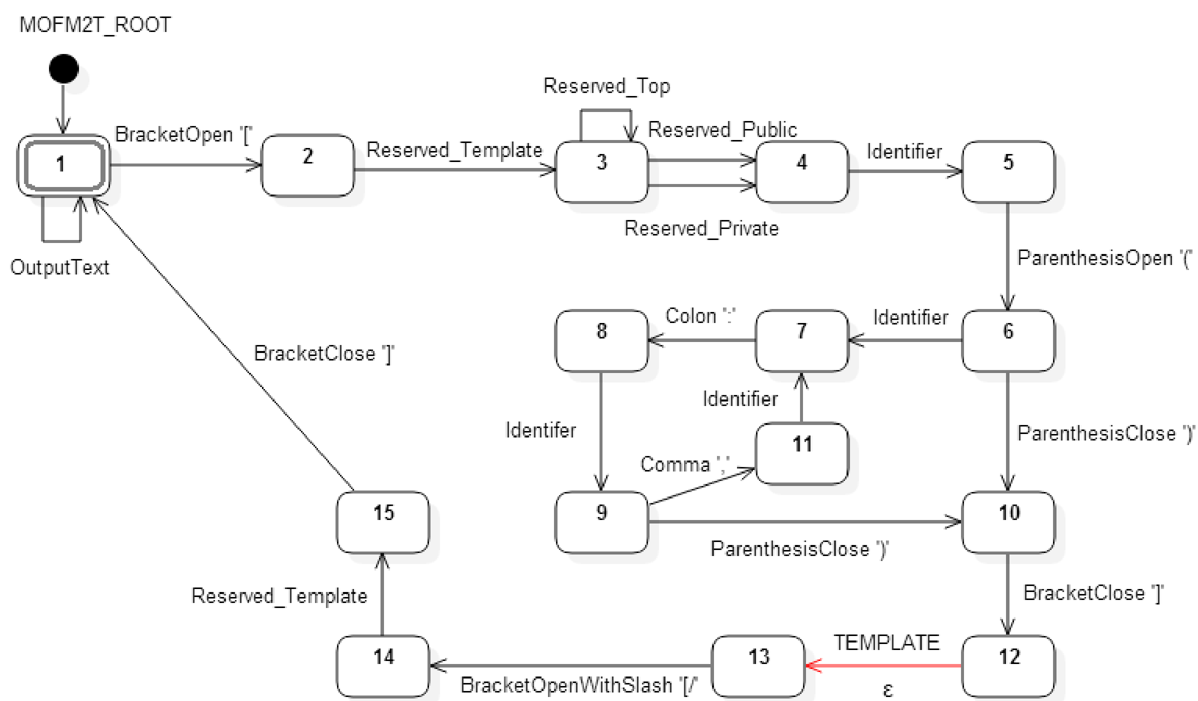
No caso do interpretador LPO, o mesmo raciocínio adaptativo para processamento dos quantificadores ∀ e ∃, ou `forall` e `exists`, foi utilizado, já que o processamento de expressões com os mesmos precisa avaliar se todas ou alguma das instâncias do modelo respeita alguma condição.

A seguir, são apresentados os autômatos utilizados pelo analisador sintático e interpretador MOFM2T e, posteriormente, para o analisador sintático e interpretador LPO. Setas vermelhas representam chamadas de submáquinas. O símbolo ε denota a cadeia vazia, ou seja, uma transição rotulada com este símbolo ocorre sem o consumo de *token*. Setas azuis representam transições adaptativas, que podem ser colocadas ou removidas pela camada adaptativa do interpretador conforme necessidade para processamento da próxima iteração em um laço ou saída do laço.

A máquina inicial é *MOFM2T_ROOT*, ou seja, o início do processamento de uma função de mapeamento escrita em MOFM2T inicia no estado inicial desse autômato.

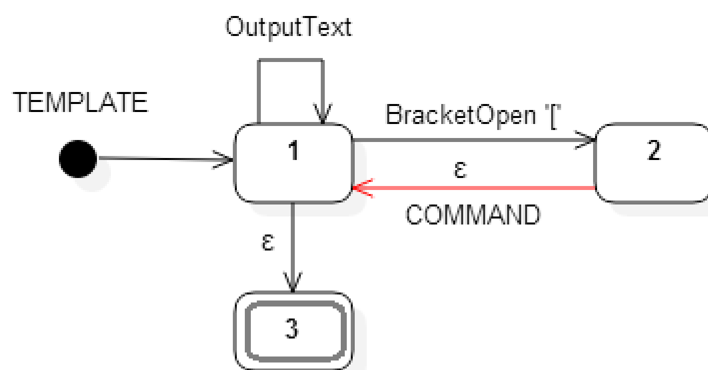
O autômato *EXPRESSION*, responsável por avaliar expressões, não diferencia sintaticamente tipos diferentes na mesma expressão. Se uma expressão inválida ocorrer (p. ex., divisão de um inteiro por uma *string*), o interpretador reporta um erro em tempo de execução. Tipos booleanos não foram implementados para fins de simplificação. Como na linguagem C, inteiros são utilizados como booleanos. Uma expressão lógica resulta no inteiro 0 para falso e 1 para verdadeiro.

Figura 43 – Autômato MOFM2T_ROOT



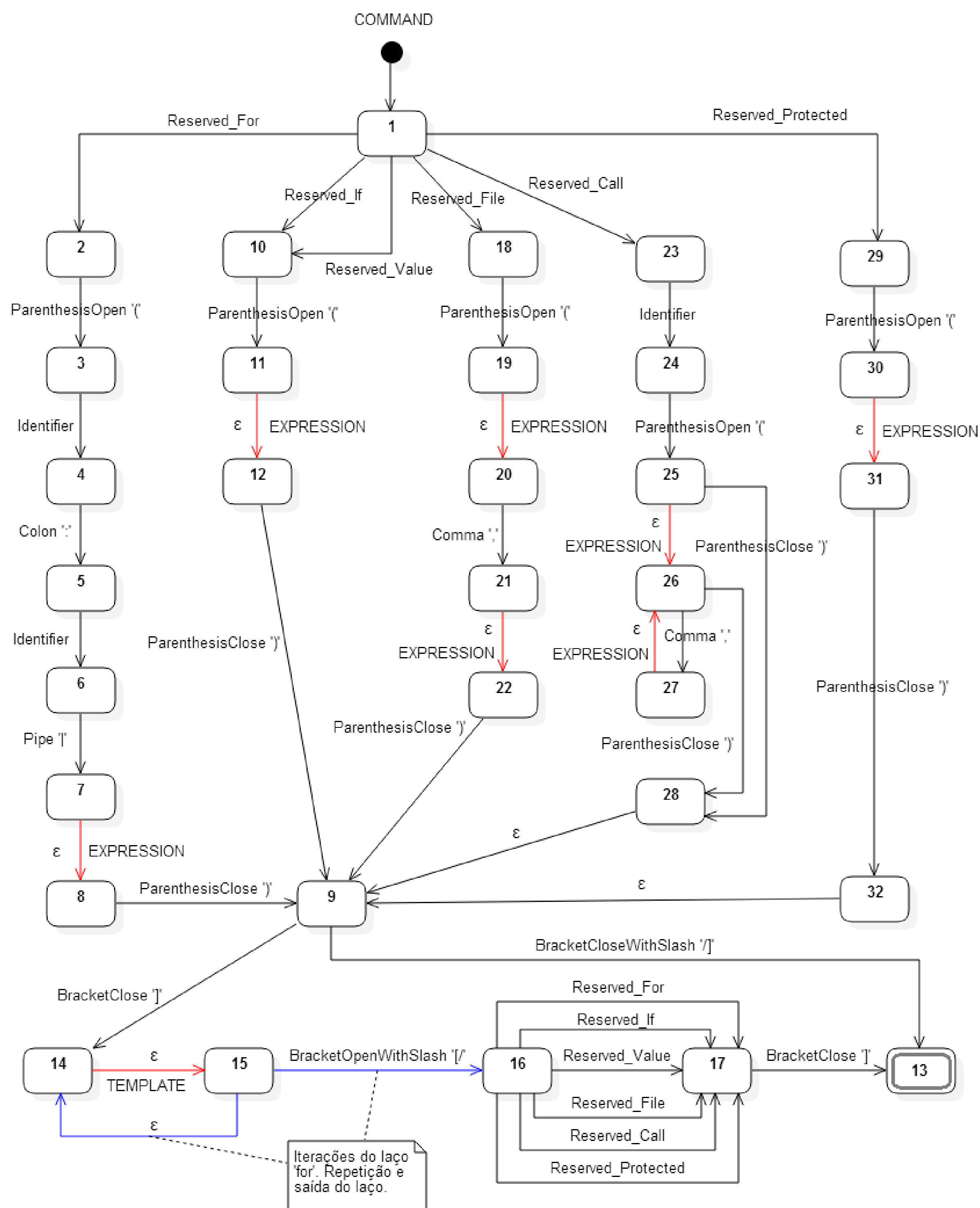
Fonte: autor

Figura 44 – Autômato TEMPLATE



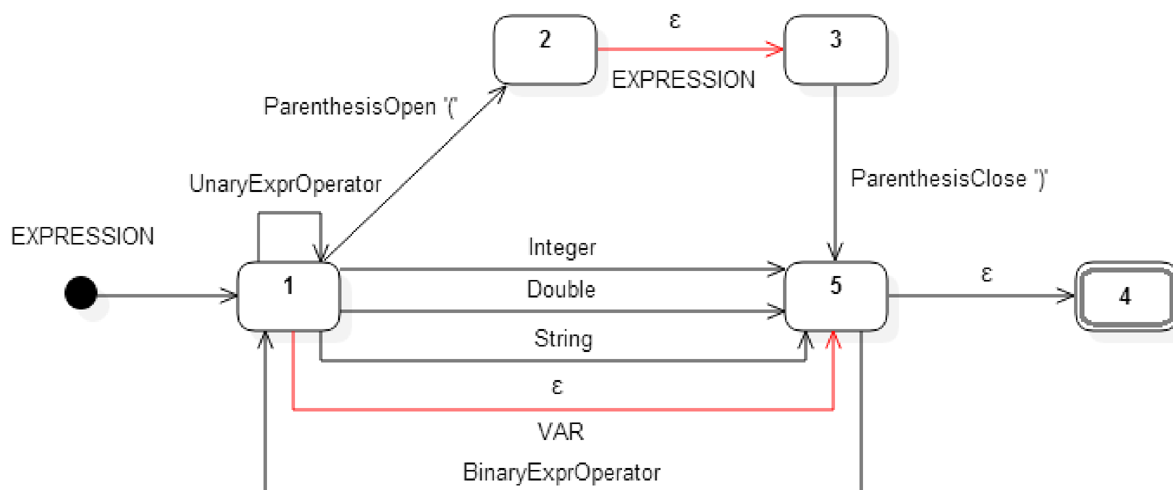
Fonte: autor

Figura 45 – Autômato COMMAND



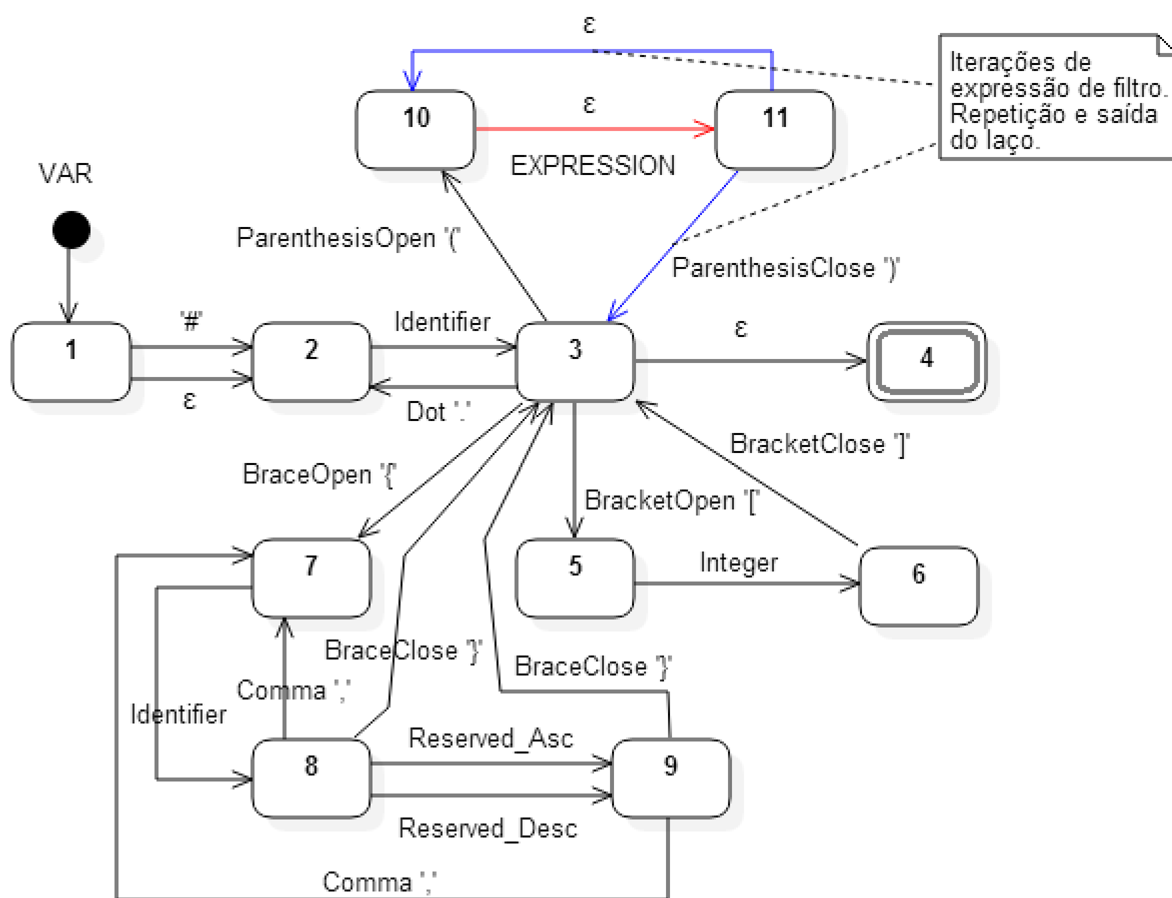
Fonte: autor

Figura 46 – Autômato EXPRESSION



Fonte: autor

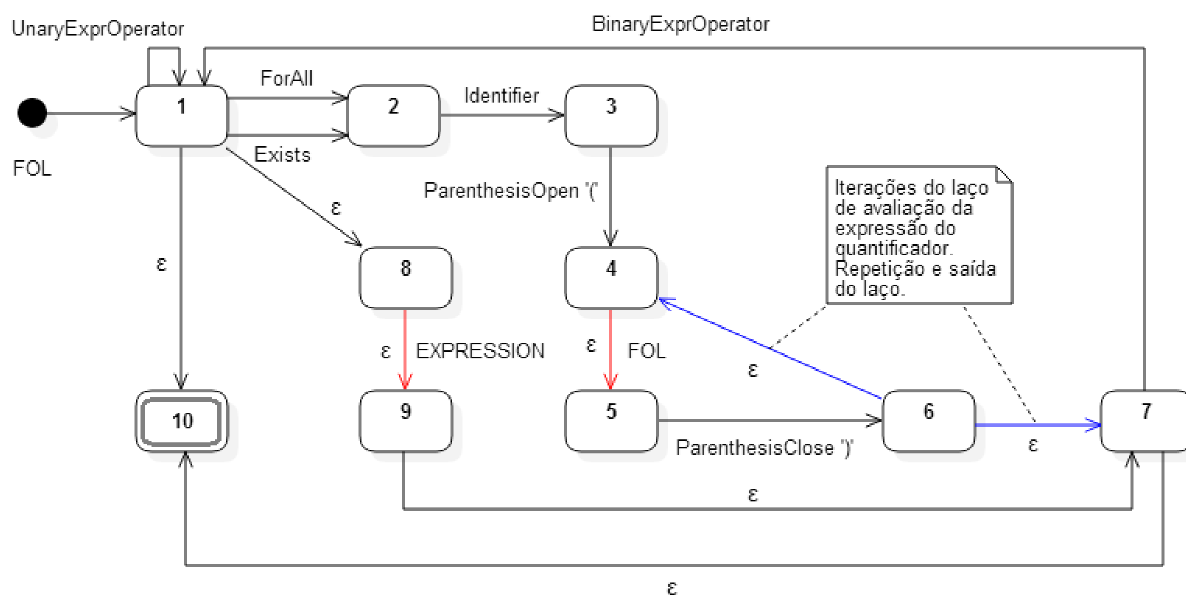
Figura 47 – Autômato VAR



Fonte: autor

No caso do analisador sintático e interpretador de LPO, a máquina inicial é *FOL*, ou seja, o início do processamento de uma sentença LPO inicia no estado inicial desse autômato, apresentado na Figura 48. Conforme comentado anteriormente, ele faz uso do autômato *EXPRESSION* como submáquina.

Figura 48 – Autômato FOL



Fonte: autor

APÊNDICE C – Exemplo de Aplicação para Programas Adaptativos

O objetivo deste apêndice é apresentar um exemplo de aplicação do formalismo SBMM e da ferramenta SBMMTool para o desenvolvimento de programas adaptativos utilizando MDE.

Um dispositivo adaptativo é composto por um dispositivo não-adaptativo subjacente e um mecanismo formado por funções adaptativas capaz de alterar o conjunto de regras que define seu comportamento (NETO, 2001). Essa camada adaptativa confere ao dispositivo capacidade de automodificação, em que as alterações nas regras de comportamento são disparadas em função da configuração corrente do dispositivo e dos estímulos recebidos. Essas alterações se caracterizam pela substituição, inserção ou remoção de regras.

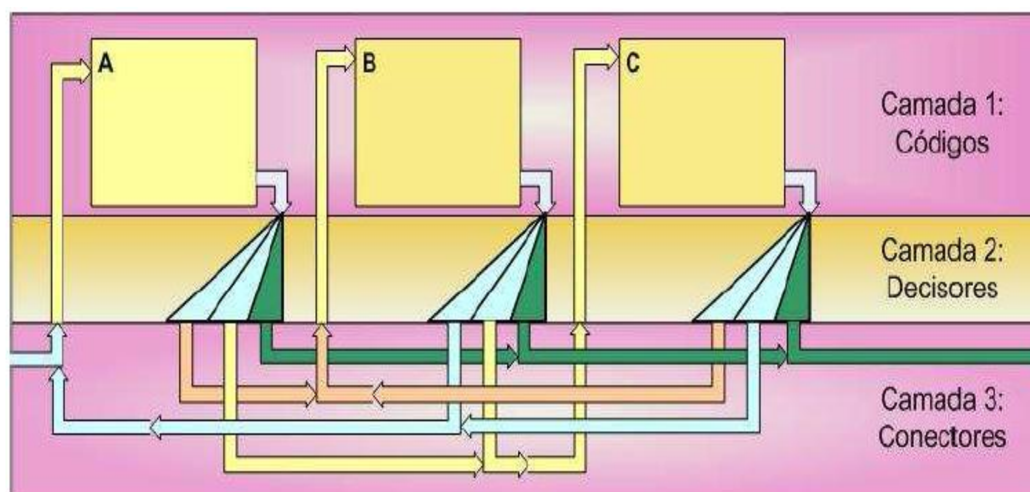
Um programa adaptativo pode ser entendido como uma especificação de uma sequência automodificável de instruções, que representa um código dinamicamente alterável (SILVA, 2010). Podem ser considerados dispositivos adaptativos cujo dispositivo não-adaptativo subjacente é um programa estático, a ser detalhado mais adiante.

Em um programa adaptativo, as ações adaptativas podem inserir ou remover linhas de código, antes ou depois de processar um estímulo.

Basic Adaptive Language (BADAL) é uma linguagem de programação adaptativa de alto nível proposta por Silva (2010). Uma linguagem adaptativa deve prover instruções explícitas para alteração do código-fonte em tempo de execução, e assim o faz a BADAL. O compilador BADAL apresentado por Silva (2010) gera código para o ambiente de execução desenvolvido em Pelegrini (2009), que é uma máquina virtual com características específicas que possibilita que um programa realize automodificações em seu código em tempo de execução.

Primeiramente, é apresentada uma notação para o dispositivo subjacente não-adaptativo do programa adaptativo, isto é, o programa estático escrito em uma linguagem de alto nível hospedeira. A Figura 49 (SILVA, 2010) apresenta a arquitetura de um programa projetado como um dispositivo guiado por regras, onde é possível verificar uma camada de código formada por blocos básicos escritos em linguagem hospedeira (camada 1).

Figura 49 – Arquitetura de um programa não-adaptativo na forma de um dispositivo guiado por regras



Fonte: Silva (2010)

A camada 3 provê conexões que ligam a saída de um bloco básico, após passar por um decisor, à entrada de outro bloco (ou do mesmo). O valor de saída do bloco recém executado é verificado por seu decisor, na camada 2, que encaminha a execução do programa para o bloco conectado à saída respectiva.

Define-se um bloco básico como uma parcela do programa expressa na forma de uma sequência de comandos da linguagem hospedeira, que deve ser descrito de modo que apresente uma só entrada e uma só saída. O valor de saída deve exprimir uma condição referente ao resultado de sua execução.

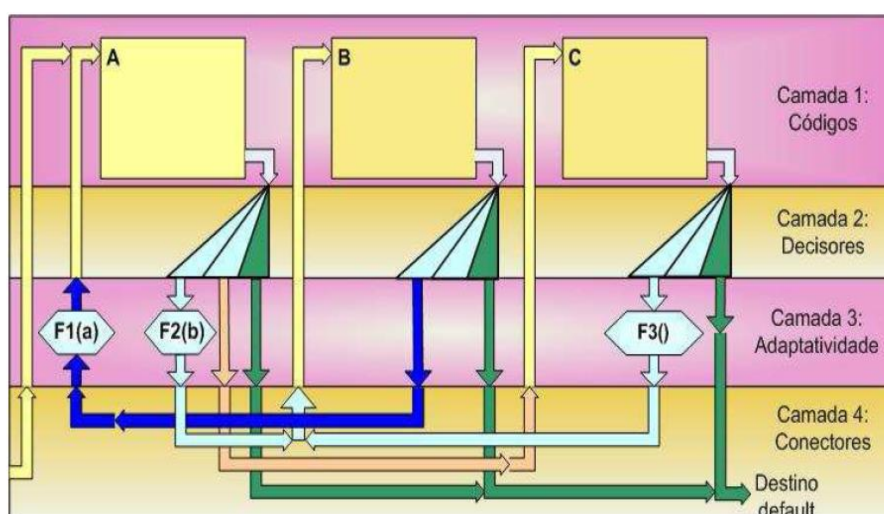
Dessa forma, os programas adaptativos a serem elaborados pelo programador contêm, como elementos construtivos iniciais, blocos básicos escritos puramente na linguagem hospedeira (SILVA, 2010). Para assegurar a coerência estrutural dos programas assim construídos, é preciso que o programador projete adequadamente as conexões e decisores, e que o compilador faça as validações necessárias.

Cabe ao programador definir os possíveis valores de saída de cada bloco básico. Caso se obtenha um valor de saída não especificado nos decisores, BADAL determina que a cláusula *OTHERWISE*, obrigatória em todas as conexões, garanta que sempre haverá algum destino para o fluxo do programa após o término da execução de um bloco básico.

Uma entrada de bloco básico pode receber mais de uma conexão, conforme exemplo da Figura 49. Por outro lado, cada valor de saída de um bloco básico deve estar associado a uma única conexão, garantindo que o próximo bloco a executar seja obtido deterministicamente.

Até aqui foi definido o dispositivo não-adaptativo subjacente. A camada adaptativa é introduzida entre a camada de decisores e de conectores. Ela se responsabiliza pela capacidade de alteração do programa em tempo de execução, correspondendo a funções adaptativas. Suas chamadas ficam atreladas às conexões condicionais estabelecidas entre os blocos básicos. A Figura 50 apresenta a arquitetura atualizada com a camada adaptativa.

Figura 50 – Arquitetura de um programa adaptativo



Fonte: Silva (2010)

Na nova camada 3 aparecem todas as funções adaptativas cuja utilização esteja prevista na lógica do programa, e essas são associadas aos conectores da camada 4.

As funções adaptativas devem ser especificadas em algum lugar no corpo do programa adaptativo. A declaração de uma função adaptativa se resume a indicar as ações de modificação do programa adaptativo a serem efetuadas em tempo de execução nas ocasiões em que a função for ativada.

BADAL determina que as funções adaptativas se restrinjam a executar ações de inserção ou de remoção, tanto de blocos básicos como de conexões entre eles (SILVA, 2010). As partes do metamodelo que formalizam as funções adaptativas devem refletir esse aspecto.

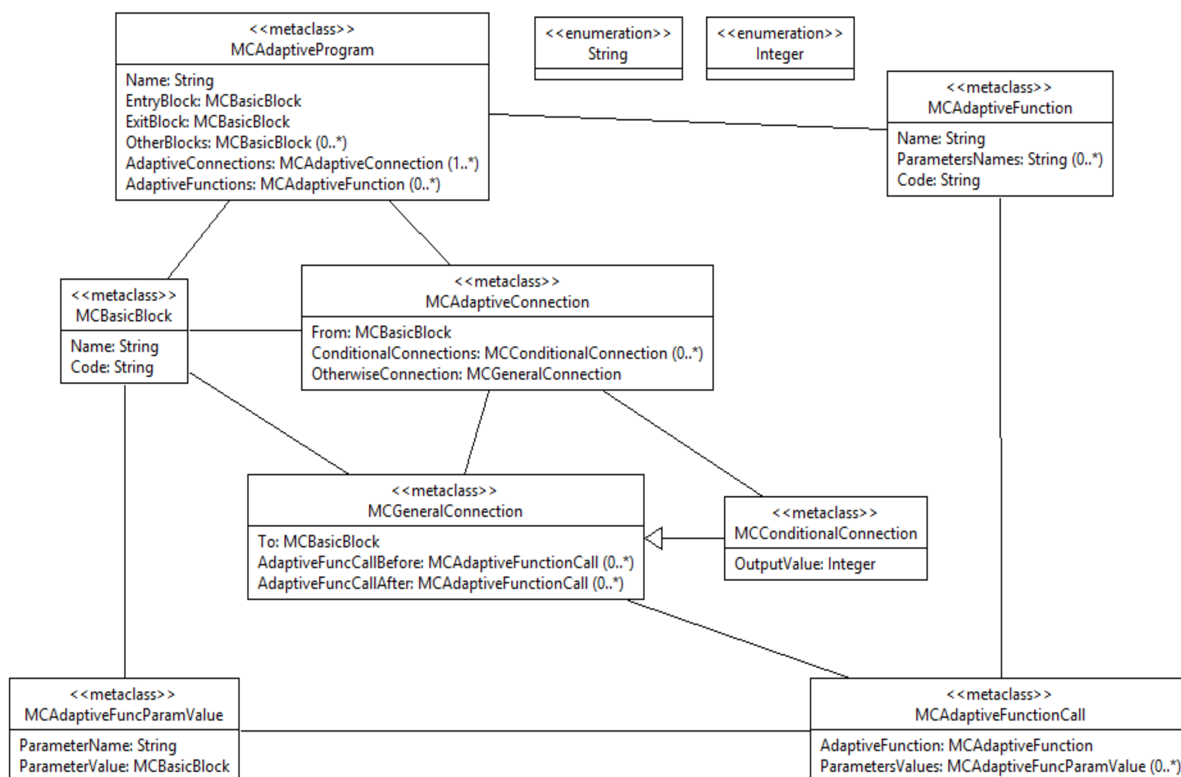
A referência a um bloco básico deve ser feita por nome, enquanto a referência a uma conexão deve especificar o respectivo bloco básico de origem, bem como o valor de saída desse bloco básico que o seleciona (SILVA, 2010).

Baseando-se na arquitetura descrita acima, Canovas e Cugnasca (2014) estabelecem um metamodelo descrito em SBMM para programas adaptativos. O metamodelo reflete os conceitos apresentados e suas relações. Um refinamento desse metamodelo incluindo restrições é descrito por Canovas e Cugnasca (2015a).

É importante, neste ponto, observar que os programas adaptativos aqui apresentados e os modelos executáveis adaptativos, apresentados na seção 5.2, são artefatos distintos, embora ambos adotem o conceito da automodificação em tempo de execução. Os modelos de programas adaptativos discutidos neste apêndice têm por objetivo representar programas adaptativos em mais alto nível de abstração, cujo código-fonte na linguagem BADAL pode ser gerado por uma transformação do tipo modelo-para-código. Esses modelos em si não são adaptativos, ao menos enquanto nem mesmo uma semântica de execução é atribuída. O artefato executável e adaptativo é o programa BADAL resultante da transformação. Já os modelos executáveis adaptativos, tais como descritos na seção 5.2, são executáveis diretamente e possuem capacidade de automodificação.

O metamodelo que define a DSML para programas adaptativos, desenvolvido como estudo de caso neste apêndice, está representado na Figura 51, em notação gráfica, e descrito em seguida, em notação de conjuntos, iniciando pela eq. (87). O detalhamento e interpretação de cada restrição podem ser encontrados em Canovas e Cugnasca (2015a).

Figura 51 – Metamodelo de uma DSML para representação de programas adaptativos



Fonte: autor

$$MM_{PA} = (\text{“AdaptiveProgramMetamodel”}, C_{PA}, \Gamma_{PA}, E_{PA}, R_{PA}, descriptor_{c_PA}, descriptor_{e_PA}) \quad (87)$$

Em que:

- $C_{PA} = \{c_0, c_1, c_2, c_3, c_4, c_5, c_6, c_7\}$
  - $c_0 =^{c_PA}$ (“MCAdaptiveProgram”, P_0)
    - $P_0 = \{p_{01}, p_{02}, p_{03}, p_{04}, p_{05}, p_{06}\}$
      - p_{01} = (“Name”, e_1, 1..1)
      - p_{02} = (“EntryBlock”, c_1, 1..1)
      - p_{03} = (“ExitBlock”, c_1, 1..1)
      - p_{04} = (“OtherBlocks”, c_1, 0..*)
      - p_{05} = (“AdaptiveConnections”, c_2, 1..*)
      - p_{06} = (“AdaptiveFunctions”, c_7, 0..*)

- $c_1 =^{c_PA}$ (“MCBasicBlock”, P_1)
  - $P_1 = \{p_{11}, p_{12}\}$
    - p_{11} = (“Name”, e_1, 1..1)
    - p_{12} = (“Code”, e_1, 1..1)
- $c_2 =^{c_PA}$ (“MCAdaptiveConnection”, P_2)
  - $P_2 = \{p_{21}, p_{22}, p_{23}\}$
    - p_{21} = (“From”, c_1, 1..1)
    - p_{22} = (“ConditionalConnections”, c_3, 0..*)
    - p_{23} = (“OtherwiseConnection”, c_4, 1..1)
- $c_3 =^{c_PA}$ (“MCConditionalConnection”, P_3)
  - $P_3 = \{p_{31}\}$
    - p_{31} = (“OutputValue”, e_2, 1..1)
- $c_4 =^{c_PA}$ (“MCGeneralConnection”, P_4)
  - $P_4 = \{p_{41}, p_{42}, p_{43}\}$
    - p_{41} = (“To”, c_1, 1..1)
    - p_{42} = (“AdaptiveFuncCallBefore”, c_5, 0..1)
    - p_{43} = (“AdaptiveFuncCallAfter”, c_5, 0..1)
- $c_5 =^{c_PA}$ (“MCAdaptiveFunctionCall”, P_5)
  - $P_5 = \{p_{51}, p_{52}\}$
    - p_{51} = (“AdaptiveFunction”, c_7, 1..1)
    - p_{52} = (“ParametersValues”, c_6, 0..*)
- $c_6 =^{c_PA}$ (“MCAdaptiveFuncParamValue”, P_6)
  - $P_6 = \{p_{61}, p_{62}\}$
    - p_{61} = (“ParameterName”, e_1, 1..1)
    - p_{62} = (“ParameterValue”, c_1, 1..1)
- $c_7 =^{c_PA}$ (“MCAdaptiveFunction”, P_7)
  - $P_7 = \{p_{71}, p_{72}, p_{73}\}$
    - p_{71} = (“Name”, e_1, 1..1)
    - p_{72} = (“ParametersNames”, e_1, 0..*)
    - p_{73} = (“Code”, e_1, 1..1)

- $\Gamma_{PA} = \{ (c_3, c_4) \}$
- $E_{PA} = \{e_1, e_2\}$
  - $e_1 =^{e_PA}$ (“String”, Σ^*)

- $e_2 =^{e_PA}$ ("Integer", $\mathbb{Z}$)

- $R_{PA} = \{r_1, r_2, r_3, r_4, r_5, r_6, r_7, r_8, r_9, r_{10}, r_{11}, r_{12}, r_{13}\}$
  - r_1: $\forall o$ | MCBasicBlock(o) $\Rightarrow \neg$ (MCBasicBlock.Name (o) = ε)
  - r_2: $\forall o_1, o_2$ | MCBasicBlock (o_1) $\wedge$ MCBasicBlock (o_2) $\wedge$ (MCBasicBlock.Name(o_1) = MCBasicBlock. Name(o_2)) $\Rightarrow o_1 = o_2$
  - r_3: $\forall a, o_1, o_2$ | MCAdaptiveConnection(a) $\wedge$ MCConditionalConnection(o_1) $\wedge$ MCConditionalConnection(o_2) $\wedge$ ($o_1 \in$ MCAdaptiveConnection.ConditionalConnections(a)) $\wedge$ ($o_2 \in$ MCAdaptiveConnection. ConditionalConnections(a)) $\wedge$ (MCConditionalConnection.OutputValue(o_1) = MCConditionalConnection.OutputValue(o_2)) $\Rightarrow o_1 = o_2$
  - r_4: $\forall a, o_1, o_2$ | MCAdaptiveFunctionCall(a) $\wedge$ MCAdaptiveFuncParamValue(o_1) $\wedge$ MCAdaptiveFuncParamValue(o_2) $\wedge$ ($o_1 \in$ MCAdaptiveFunctionCall.ParametersValues(a)) $\wedge$ ($o_2 \in$ MCAdaptiveFunctionCall. ParametersValues(a)) $\wedge$ (MCAdaptiveFuncParamValue.ParameterName(o_1) = MCAdaptiveFuncParamValue. ParameterName (o_2)) $\Rightarrow o_1 = o_2$
  - r_5: $\forall o$ | MCAdaptiveFunction(o) $\Rightarrow \neg$ (MCAdaptiveFunction.Name (o) = ε)
  - r_6: $\forall o_1, o_2$ | MCAdaptiveFunction(o_1) $\wedge$ MCAdaptiveFunction(o_2) $\wedge$ (MCAdaptiveFunction.Name(o_1) = MCAdaptiveFunction. Name(o_2)) $\Rightarrow$ $o_1 = o_2$
  - r_7: $\forall a, o$ | MCAdaptiveFunction(a) $\wedge$ String(o) $\wedge$ ($o \in$ MCAdaptiveFunction.ParametersNames(a)) $\Rightarrow \neg(o = \varepsilon)$
  - r_8: $\forall a, o_1, o_2$ | MCAdaptiveFunction(a) $\wedge$ String(o_1) $\wedge$ String(o_2) $\wedge$ ($o_1 \in$ MCAdaptiveFunction.ParametersNames(a)) $\wedge$ ($o_2 \in$ MCAdaptiveFunction.ParametersNames(a)) $\wedge$ (String.Value(o_1) = String.Value(o_2))) $\Rightarrow o_1 = o_2$
  - r_9: $\forall a, o, f$ | MCAdaptiveFunctionCall(a) $\wedge$ MCAdaptiveFuncParamValue(o) $\wedge$ ($o \in$ MCAdaptiveFunctionCall.ParametersValues(a)) $\wedge$ MCAdaptiveFunction(f) $\wedge$

$(\text{MCAdaptiveFunctionCall.AdaptiveFunction}(a) = f) \Rightarrow$

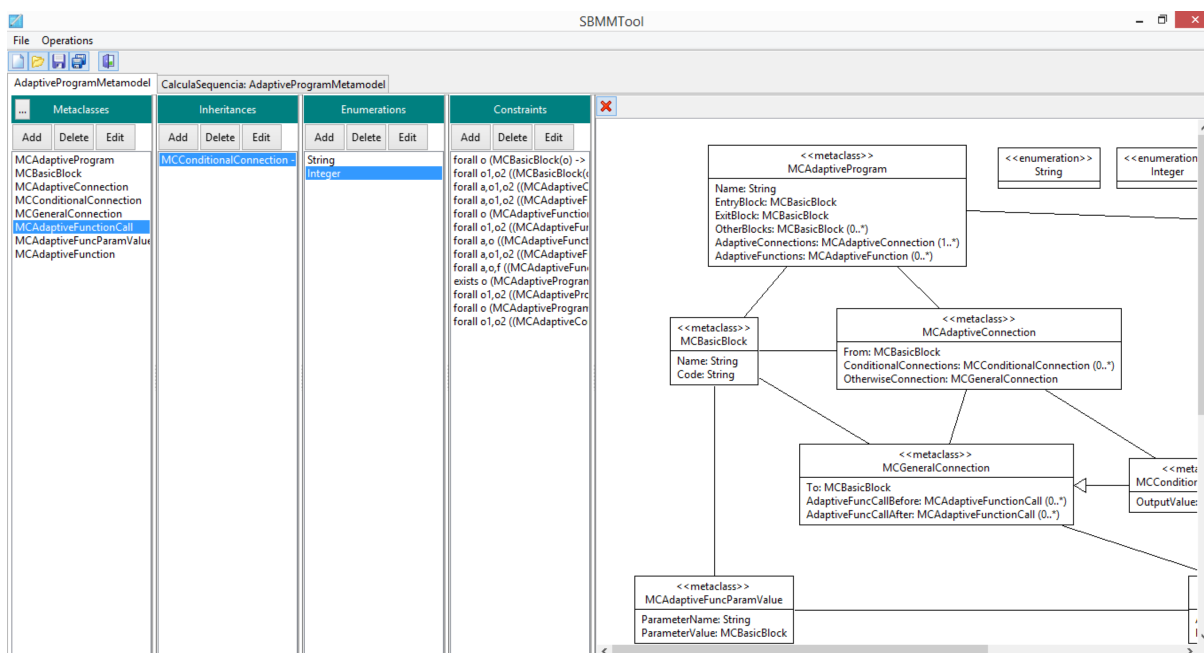
$\text{MCAdaptiveFuncParamValue.ParameterName}(o) \in$

$\text{MCAdaptiveFunction.ParametersNames}(f)$

- r_{10}: $\exists o \mid \text{MCAdaptiveProgram}(o)$
- r_{11}: $\forall o_1, o_2 \mid \text{MCAdaptiveProgram}(o_1) \wedge \text{MCAdaptiveProgram}(o_2) \Rightarrow o_1 = o_2$
- r_{12}: $\forall o \mid \text{MCAdaptiveProgram}\ (o) \Rightarrow \neg\ (\text{MCAdaptiveProgram.Name}\ (o) = \varepsilon)$
- r_{13}: $\forall o_1, o_2 \mid \text{MCAdaptiveConnection}(o_1) \wedge \text{MCAdaptiveConnection}(o_2) \wedge (\text{MCAdaptiveConnection.From}(o_1) = \text{MCAdaptiveConnection.From}(o_2)) \Rightarrow o_1 = o_2$

Esse metamodelo foi introduzido na ferramenta SBMMTool, conforme apresentado na Figura 52.

Figura 52 – Tela da ferramenta SBMMTool com o metamodelo AdaptiveProgramMetamodel



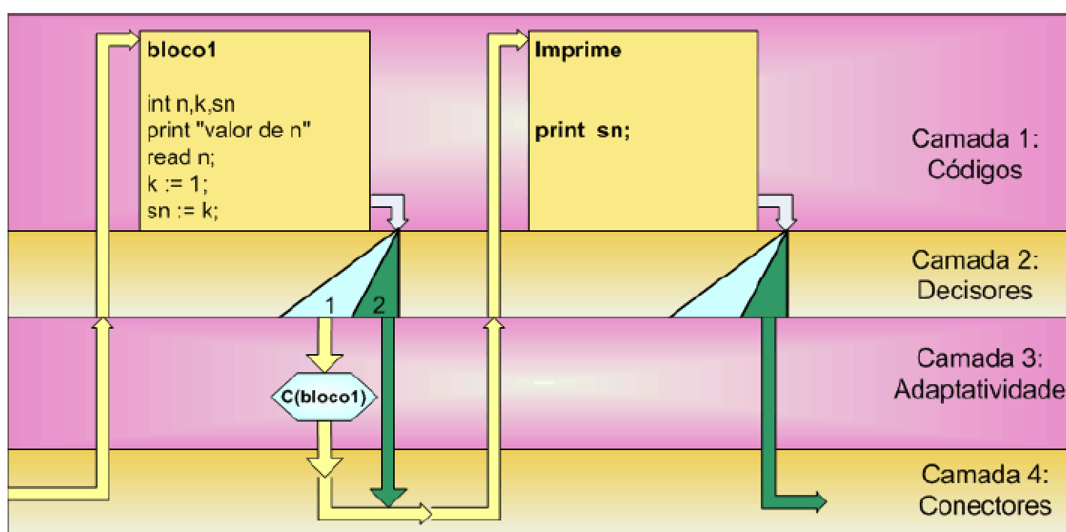
Fonte: autor

Uma vez que o metamodelo *AdaptiveProgramMetamodel* foi carregado na SBMMTool, é possível criar modelos de programas adaptativos. A ferramenta é responsável por aplicar todas as restrições do conjunto R_{PA} sobre o modelo em edição, garantindo a geração de modelos conformes.

A Figura 53 apresenta o modelo do programa conforme representação gráfica apresentada em Silva (2010), enquanto a Figura 54 ilustra a tela de edição do modelo na ferramenta SBMMTool.

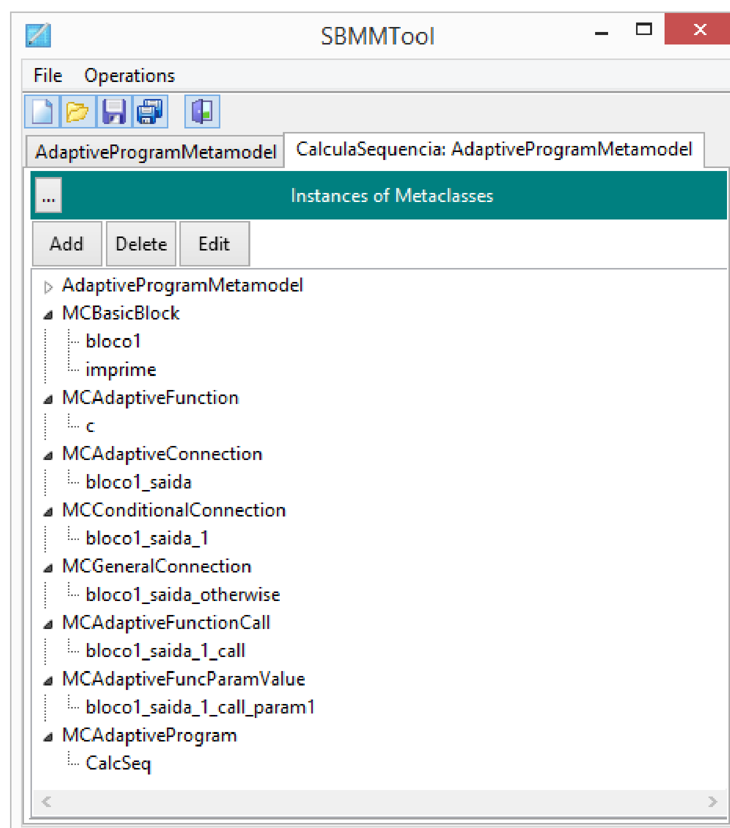
Esse programa calcula o n-ésimo termo da sequência matemática S(n) = S(n-1) + 2n para n ≥ 2, sendo S(1) = 1. A estrutura do programa consiste em dois blocos: *bloco1* e *imprime*. O *bloco1* é responsável por solicitar ao usuário o valor de n. Uma função adaptativa *c* é colocada em uma conexão condicional na saída de *bloco1* e insere código no programa para processar mais uma iteração da sequência quando ainda necessário. O bloco *imprime* é responsável por apresentar o resultado final ao usuário.

Figura 53 – Modelo de programa adaptativo exemplo



Fonte: Silva (2010)

Figura 54 – Modelo de programa adaptativo exemplo editado na ferramenta SBMMTool



Fonte: autor

Como cada bloco básico deve gerar um valor de saída (exceto o último bloco a ser processado), é necessário então atribuir a *bloco1* um valor de saída e um decisor. O decisor direcionará a execução de acordo com o valor de saída. Quando o valor de entrada digitado pelo usuário (variável *n*) for 1, já se sabe que S(1) = 1 e, portanto, o resultado é apenas impresso sem necessidade de cálculo. Para os termos a partir do segundo, a variável *k* de *bloco1* guarda o número de ordem do termo cujo valor foi computado, o qual é armazenado na variável *sn*. A comparação de *k* com *n* permite saber se o termo requerido pelo usuário já foi encontrado, condição que está implementada no decisor. Por convenção, o valor de saída será 1 se o n-ésimo termo ainda não foi atingido e 2 caso contrário. Esse valor é testado pelo decisor para determinar qual o próximo bloco básico a ser processado. O código-fonte original de *bloco1* e *imprime* pode ser conferido no Quadro 29.

Quadro 29 – Código-fonte de bloco1 e imprime

```
1  code bloco1: <
2     int n,k,sn;
3     print "valor de n";
4     read n;
5     k := 1;
6     sn := k;
7     if n>k then bloco1 := 1 else bloco1 := 2;
8  >;
9
10 code imprime: <
11    print sn;
12 >;
```

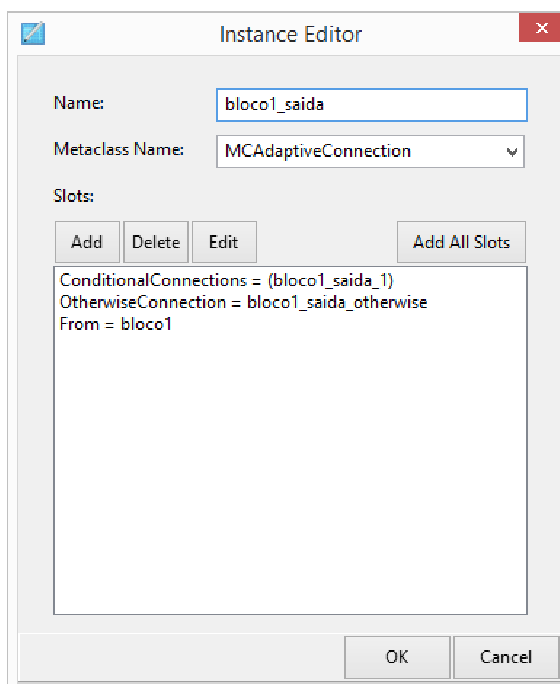
Fonte: Silva (2010)

O bloco *imprime* não necessita de um decisor, pois corresponde ao bloco de saída do programa, ou seja, à propriedade *ExitBlock* da única instância de programa adaptativo permitida no modelo, isto é, da metaclasse *MCAdaptiveProgram*.

De acordo com a Figura 53, o valor de saída 1 de *bloco1* indica que o valor informado pelo usuário é maior que 1 e, portanto, é necessário computar os termos seguintes da sequência até o n-ésimo. Sendo assim, a conexão que o interliga ao bloco *imprime* chama a função adaptativa *c*, que calculará o n-ésimo termo antes de processar o bloco *imprime*.

A montagem da conexão, juntamente com os respectivos valores de decisão e a chamada da função adaptativa para a saída 1, estão presentes no modelo do programa conforme a Figura 54, mais precisamente de *bloco1_saida* até *bloco1_saida_1_call_param1*. A Figura 55 apresenta outra tela da ferramenta com detalhes do preenchimento dos valores das propriedades para a instância do modelo *bloco1_saida*.

Figura 55 – Edição da instância bloco1_saida na ferramenta SBMMTool



Fonte: Canovas e Cugnasca (2015a)

A função adaptativa *c*, por sua vez, é responsável por instanciar um novo bloco básico, não presente no programa estático subjacente original. Trata-se de uma ação adaptativa de inserção de código em tempo de execução, caracterizando a adaptatividade do programa. Também é instanciada uma conexão desse novo bloco básico para *imprime*, passando por um decisor igual ao de *bloco1*, que novamente aciona a função adaptativa *c* para a saída igual a 1. A condição para a saída desse novo bloco instanciado resultar em 2 é que o valor de *k* atinja o valor de *n*, provocando então a parada do laço de repetição montado através de sucessivas instanciações de blocos básicos pela função adaptativa *c*. O código-fonte dessa função adaptativa em BADAL pode ser conferido no Quadro 30.

A função de mapeamento do Quadro 31 foi inserida na ferramenta SBMMTool para transformar modelos conformes ao metamodelo *AdaptiveProgramMetamodel* para código-fonte em linguagem BADAL. Com isso, a SBMMTool passa a funcionar como uma ferramenta CASE para o desenvolvimento de programas adaptativos. Através dessa função de mapeamento, a ferramenta não só é capaz de permitir a edição de modelos de programas adaptativos, mas também de gerar código BADAL a partir dos mesmos.

Quadro 30 – Código-fonte da função adaptativa c

```
1  adaptive function c(i) [generators g] {
2     insert [ code g: <
3        k := k + 1;
4        sn := sn + 2 * k;
5        if n>k then g := 1 else g := 2;
6        >;
7     ];
8     insert [ connection from g (case 1: to imprime perform c(g)
   before otherwise to imprime)
9     ];
10 }
```

Fonte: Silva (2010)

Quadro 31 – Função de mapeamento de modelos de programas adaptativos para linguagem BADAL

```
1  [template top public Main()]
2  [if(MCAdaptiveProgram[0])]
3  [file(MCAdaptiveProgram[0].Name+'.txt',0)]ADAPTIVE MAIN \[NAME =
   [value(MCAdaptiveProgram[0].Name) /], ENTRY =
   [if(MCAdaptiveProgram[0].EntryBlock)][value(MCAdaptiveProgram[0]
   .EntryBlock.Name) /][/if], EXIT =
   [if(MCAdaptiveProgram[0].ExitBlock)][value(MCAdaptiveProgram[0].
   ExitBlock.Name) /][/if] ] IS
4
5  [if(MCAdaptiveProgram[0].EntryBlock)]CODE
   [value(MCAdaptiveProgram[0].EntryBlock.Name) /]:
   <[value(MCAdaptiveProgram[0].EntryBlock.Code) /]>;[/if]
6  [if(MCAdaptiveProgram[0].ExitBlock)]CODE
   [value(MCAdaptiveProgram[0].ExitBlock.Name) /]:
   <[value(MCAdaptiveProgram[0].ExitBlock.Code) /]>;[/if]
7  [for(b:MCBasicBlock|MCAdaptiveProgram[0].OtherBlocks)]CODE
   [value(b.Name) /]: <[value(b.Code) /]>;[/for]
8  [for(ac:MCAdaptiveConnection|MCAdaptiveProgram[0].AdaptiveConnec
   tions)]
9  CONNECTION FROM [value(ac.From.Name) /] (
10 [for(cc:MCConditionalConnection|ac.ConditionalConnections)]
   [if(position<>0)],[/if]CASE [value(cc.OutputValue) /]: TO
   [value(cc.To.Name) /][if(cc.AdaptiveFuncCallBefore)] PERFORM
   [value(cc.AdaptiveFuncCallBefore.AdaptiveFunction.Name)
   /]([for(ccp:MCAdaptiveFuncParamValue|cc.AdaptiveFuncCallBefore.P
   arametersValues)][if(position<>0)],[/if][value(ccp.ParameterValu
   e.Name) /][/for]) BEFORE [/if][if(cc.AdaptiveFuncCallAfter)]
   PERFORM [value(cc.AdaptiveFuncCallAfter.AdaptiveFunction.Name)
   /]([for(ccp:MCAdaptiveFuncParamValue|cc.AdaptiveFuncCallAfter.Pa
   rametersValues)][if(position<>0)],[/if][value(ccp.ParameterValue
   .Name) /][/for]) AFTER[/if]
```

```
11  [/for]   OTHERWISE TO [value(ac.OtherwiseConnection.To.Name) /]
    [if(ac.OtherwiseConnection.AdaptiveFuncCallBefore)] PERFORM
    [value(ac.OtherwiseConnection.AdaptiveFuncCallBefore.AdaptiveFun
    ction.Name)
    /]([for(ccp:MCAdaptiveFuncParamValue|ac.OtherwiseConnection.Adap
    tiveFuncCallBefore.ParametersValues)][if(position<>0)],[/if][val
    ue(ccp.ParameterValue.Name) /][/for]) BEFORE
    [/if][if(ac.OtherwiseConnection.AdaptiveFuncCallAfter)] PERFORM
    [value(ac.OtherwiseConnection.AdaptiveFuncCallAfter.AdaptiveFunc
    tion.Name)
    /]([for(ccp:MCAdaptiveFuncParamValue|ac.OtherwiseConnection.Adap
    tiveFuncCallAfter.ParametersValues)][if(position<>0)],[/if][valu
    e(ccp.ParameterValue.Name) /][/for]) AFTER[/if]
12  )
13  [/for]
14  [for(af:MCAdaptiveFunction|MCAdaptiveProgram[0].AdaptiveFunction
    s)]
15  ADAPTIVE FUNCTION [value(af.Name)
    /]([for(afp:String|af.ParametersNames)][if(position<>0)],[/if][v
    alue(afp) /][/for])
16  {[value(af.Code) /]
17  }
18  [/for]
19  END MAIN.
20  [/file]
21  [/if]
22  [/template]
```

Fonte: autor

No entanto, um modelo descrito no metamodelo proposto será capaz de gerar código adaptativo apenas no que diz respeito à estrutura dos blocos e conexões. A implementação dos blocos e funções adaptativas em si foram modeladas como propriedades *string*, ou seja, a SBMMTool ou outra ferramenta que permita o programador editar um modelo conforme ao metamodelo apresentado considera que essas implementações são texto livre, devendo o programador preencher código na linguagem hospedeira e linguagem adaptativa de interesse. Futuras extensões do metamodelo proposto podem considerar detalhes dos aspectos de implementação ao menos das funções adaptativas, permitindo que o modelo contenha informações completas sobre sua implementação sem depender de nenhuma linguagem específica.

APÊNDICE D – Exemplo de Aplicação para Aplicativos de Negócio

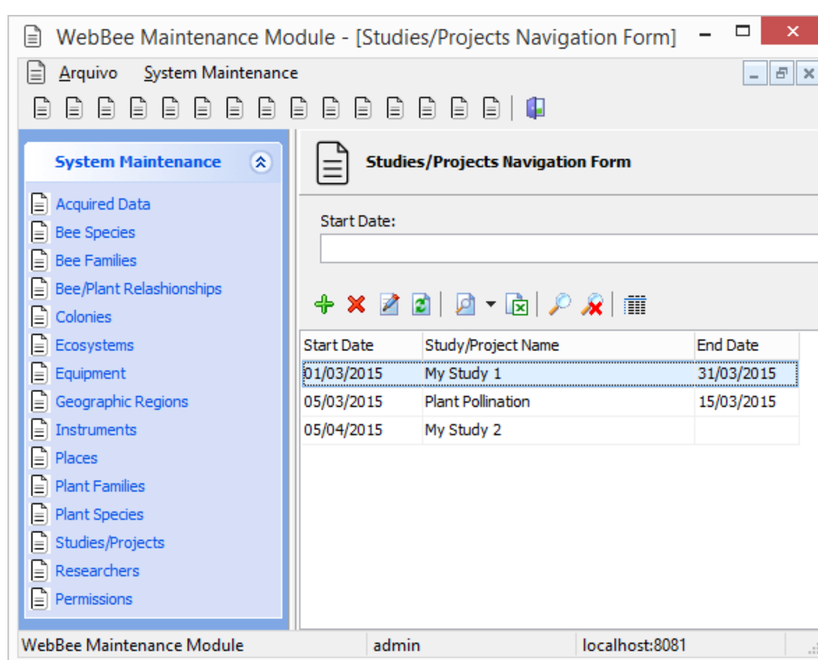
O objetivo deste apêndice é apresentar um exemplo de aplicação do formalismo SBMM e da ferramenta SBMMTool para o desenvolvimento de aplicativos de negócio utilizando MDE.

Aplicativos de negócio são utilizados para suportar a execução de processos de negócio tais como processamento de pedidos de venda ou gerenciamento de recursos humanos (ALY et al., 2013). Tipicamente têm aplicação em ambiente corporativo e são consistidos de várias camadas lógicas (interface de usuário, banco de dados, lógica de negócio, etc.), apresentando características e requisitos comuns. Dentre esses elementos comuns, podem-se citar os seguintes (CANOVAS; CUGNASCA, 2015b):

- **Objetos persistentes:** Um objeto de negócio é uma instância de uma classe que modela um conceito de determinado domínio. Um objeto persistente é um objeto (p. ex., representando um produto em um aplicativo de vendas) que é salvo em algum tipo de armazenamento permanente (p. ex., um banco de dados relacional) para posterior recuperação. Objetos persistentes são de suma importância para aplicativos de negócio, caso contrário a aplicação perderia todos os dados inseridos pelo usuário quando fosse encerrada. Classes persistentes são os tipos dos objetos persistentes. Apesar de a persistência de objetos ser um requisito importante, as linguagens de programação orientadas a objeto (p. ex., Java, C#) em geral não possuem mecanismos nativos para persistir objetos. Uma camada de persistência pode ser incluída na aplicação, usualmente na forma de bibliotecas ou *frameworks*, sendo responsável por armazenar permanentemente, recuperar e excluir objetos em um mecanismo de persistência, comumente um banco de dados relacional (AMBLER, 2005);
- **Interfaces de navegação de objetos:** Esse tipo de interface de usuário permite a navegação por objetos persistentes de determinado tipo, por exemplo em listas ou tabelas apresentando os principais atributos de cada objeto divididos em páginas. As interfaces de navegação de objetos também

permitem a seleção e execução de operações sobre os mesmos. As cinco operações básicas são: criação, recuperação, atualização, exclusão e busca. A Figura 56 apresenta um exemplo de interface de navegação de objetos de um aplicativo *desktop* Windows. Observe que o usuário pode navegar entre os descritores de objetos, representados por linhas da grade, e aplicar alguma operação sobre eles conforme a barra de ferramentas;

Figura 56 – Exemplo de interface de navegação de objetos



Fonte: Canovas e Cugnasca (2015b)

- **Interfaces de edição de objeto:** Objetos persistentes carregam dados conforme os atributos definidos em sua classe. Os usuários de aplicativos de negócio frequentemente precisam editar valores de atributos em objetos pré-existentes ou em objetos recém-criados (p. ex., editar o endereço em um objeto representando um cliente). Esse tipo de interface serve para atender esse requisito. A Figura 57 apresenta um exemplo de interface de edição de objeto. Enquanto na interface de navegação apresentam-se vários objetos de determinado tipo, a interface de edição permite editar os atributos de um objeto especificamente;

Figura 57 – Exemplo de interface de edição de objeto

Fonte: Canovas e Cugnasca (2015b)

- **Importação e exportação de dados:** Aplicativos de negócio quase nunca estão sozinhos no contexto de uma organização. Eles precisam de algum modo interagir com outras aplicações, internas ou externas à empresa. Importação e exportação de dados baseada em troca de arquivos (p. ex., Excel, XML, etc.) é um meio simples e efetivo para se obter algum nível de integração;
- **Relatórios:** Relatórios agregam dados existentes, carregados por objetos persistentes, e podem incluir informações calculadas (somas, médias, máximos, mínimos, etc.). Eles também podem incluir gráficos e imagens, não apenas texto. A Figura 58 apresenta o exemplo de um relatório em forma de listagem de objetos para impressão;
- **Aplicações *client*:** Considerando a variedade de plataformas computacionais existentes, é desejável que um aplicativo de negócio proveja mais de uma modalidade de acesso para seus usuários, tais como *web*, *desktop* ou aplicativo móvel. Uma importante arquitetura na área dos aplicativos de negócio é a arquitetura de três camadas, na qual a aplicação é dividida em

apresentação, negócio e dados. A camada de apresentação é responsável por prover interfaces de usuário e, portanto, interação com os mesmos. Se utilizada essa arquitetura, uma única camada de negócio (na qual está implementada a lógica principal da aplicação) pode atender requisições de mais de uma aplicação *client*, cada uma com uma modalidade ou finalidade. Por exemplo, em um sistema de gerenciamento de um depósito, uma certa aplicação *client* pode ser executada no setor de expedição, com interface e acesso a funcionalidades adequadas para esse departamento, enquanto outra aplicação *client* pode ser executada no ambiente administrativo da empresa, provendo acesso a funções e relatórios financeiros;

Figura 58 – Exemplo de relatório de aplicativo de negócio

Studies/Projects Navigation Form

Studies/Projects

3 registro(s)

Start Date	Study/Project Name	End Date
01/03/2015	My Study 1	31/03/2015
05/03/2015	Plant Pollination	15/03/2015
05/04/2015	My Study 2	

Fechar Imprimir Página 1 de 1 Zoom: 100% Imprimir Págs.:

Fonte: autor

- **Processos de negócio:** Um processo de negócio é uma coleção de atividades estruturadas e relacionadas em um contexto de negócio. A ordem nas quais as tarefas são executadas e as condições sob as quais são feitas são partes fundamentais de qualquer processo de negócio (GRAHAN, 2006). Um exemplo é o processo de efetuar uma venda. Uma ordem prescrita de

atividades que compõem a venda é necessária (p. ex., pedido, aprovação, pagamento e entrega). Paralelismo de atividades também é permitido. Aplicativos de negócio utilizam objetos de negócio e regras de negócio para executar esse tipo de processo;

- **Permissões de acesso a usuários:** É desejável que aplicativos de negócio permitam que um usuário administrador atribua permissões de acesso para outros usuários. Em outras palavras, alguns recursos devem estar escondidos ou inacessíveis para alguns deles, mas acessíveis para outros.

Para aplicar MDE no desenvolvimento de aplicativos de negócio, é necessário inicialmente estabelecer metamodelos adequados para a construção dos modelos das aplicações desejadas. A UML é o padrão de fato para modelagem de sistemas de software (GESSENHARTER; RAUSCHER, 2011). Por outro lado, ela não é completa. Por exemplo, aplicativos de negócio incluem interfaces de usuário, mas a UML não inclui um meio padrão para modelá-las. É necessário, portanto, estendê-la ou utilizar DSMLs de modelagem complementares. Em outras palavras, a UML estabelece uma coleção de metamodelos que formam um importante núcleo conceitual para modelagem, mas é necessário ir além da UML para suceder (PASTOR; MOLINA, 2010).

Geradores de código baseados em modelos UML usualmente são capazes de gerar código para diagramas de classes ou diagramas de atividades, ainda que descartando informações provenientes de diagramas de caso de uso, de sequência e outros. Isso não significa necessariamente que essas ferramentas devam ser “culpadas” porque não cobrem todos os aspectos da UML. Informações descartadas na geração de código não têm utilidade apenas sob a perspectiva da intenção dessas ferramentas (FONDEMENT et al., 2013). Portanto, ao definir funções de mapeamento para transformação automática de modelos, elas não necessariamente têm que ser capazes de manipular cada elemento e recurso definido nos metamodelos da UML. Definir um subconjunto útil e adequado da UML, conforme os objetivos de utilização, é recomendado no contexto de um processo MDE. De fato, esforços como xUML ou a fUML buscam definir um subconjunto da UML 2 e especificar fundações de semântica para sua execução (OMG, 2016c).

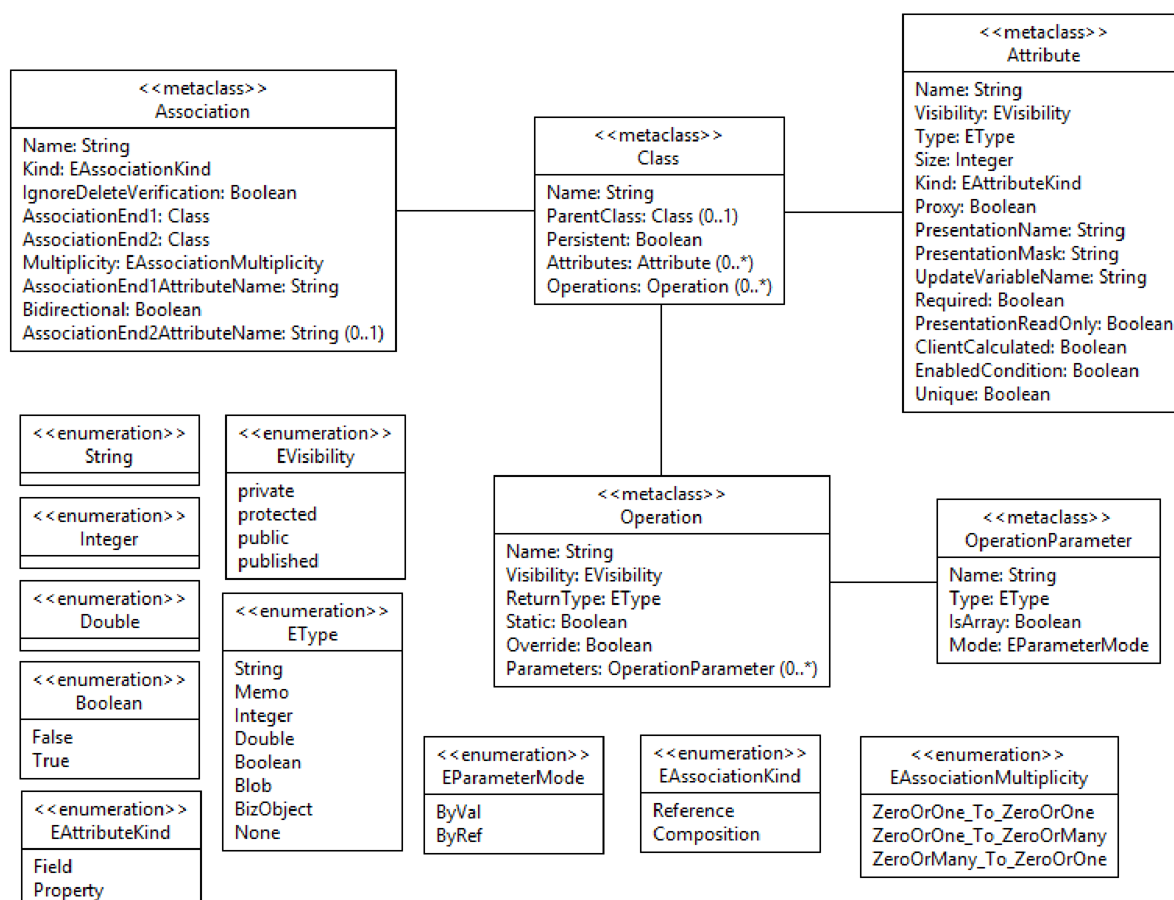
Uma vez que se deseja metamodelos que suportem os modelos com nível adequado de abstração sobre os domínios do problema e solução, e não apenas modelos que funcionam apenas como representação diferente do código-fonte em sua forma final, deve-se buscar metamodelos que se referem diretamente aos elementos comuns identificados para o tipo de aplicação desejada. Por exemplo, para modelar interfaces de navegação de objetos, é desejável referir-se aos campos visíveis ao usuário e às operações permitidas na navegação, e não a como a mudança de página deve ser processada ou a como a área de visualização deve ser renderizada. Nesse nível de abstração, também não é desejável ter que modelar detalhes sobre como é realizada a criação, exclusão e recuperação de objetos persistentes.

A seguir, são definidos metamodelos apresentados em Canovas e Cugnasca (2015b) que, em conjunto, formam uma pequena DSML que cobrem alguns elementos comuns de aplicativos de negócio identificados anteriormente.

- **Metamodelo de classes:** Este metamodelo basicamente consiste no diagrama de classes da UML, com adaptações de simplificação. Foi criado para expressar abstrações de objetos do mundo real (concretos ou abstratos) e é a principal base da tecnologia orientada a objetos. Descreve a estrutura estática da aplicação e possui suporte avançado de ferramentas. Os elementos predominantes dos modelos estruturais são classes e relacionamentos, mais frequentemente generalizações e associações (GESSENHARTER; RAUSCHER, 2011). Eles são refletidos como metaclasses no metamodelo. Classes possuem atributos e operações. Para esse propósito, apenas um subconjunto dos elementos principais da UML 2 foi tomado. Algumas modificações foram feitas para aplicar mais restrições visando à geração automática de código-fonte, isto é, a especificação de funções de mapeamento modelo-para-código. Isso acontece porque quanto mais graus de liberdade são permitidos, mais complexas as funções de mapeamento devem ser para capturar e tratar todas as possibilidades. A Figura 59 apresenta esse metamodelo em notação gráfica. Por convenção semântica desse metamodelo, assume-se que as associações são sempre navegáveis do primeiro terminal (*association-end-1*) para o segundo terminal

(*association-end-2*). Se a propriedade *Bidirectional* da metaclasse *Association* for verdadeira, então também se assume que a associação é navegável do segundo terminal para o primeiro terminal. *Multiplicity* é uma propriedade de *Association*. Difere do metamodelo padrão da UML em alguns aspectos. Primeiro, como pode ser visto na enumeração *EAssociationMultiplicity*, apenas três combinações de multiplicidade são permitidas para as propriedades nos terminais das associações. Combinações arbitrárias não são permitidas devido a intenção de simplificar a geração de código automática. Segundo, atributos de tipos primitivos são supostos terem sempre multiplicidade 1. Não há um meio de definir sua multiplicidade neste metamodelo, o que simplifica a geração de código. Se o modelador, usuário do metamodelo, deseja um *array* de inteiros como propriedade de uma classe, por exemplo, ele pode resolver o problema criando uma segunda classe com uma propriedade inteira, e então criar uma associação entre a classe original e a nova. Uma vez que a multiplicidade pode ser definida para essa associação, o valor *0..1 – 0..** (zero-ou-um para muitos) pode ser usado porque, de acordo com sua semântica, contém um-para-muitos. Além disso, uma classe pode conter operações e essas podem prever parâmetros. Um parâmetro neste metamodelo pode ser passado por valor ou por referência. Assim como atributos, parâmetros podem ser de um tipo primitivo (nesse caso, *arrays* de uma dimensão são permitidos) ou ter uma classe como tipo. Nesse caso, atribui-se a semântica usual da orientação a objetos: um objeto cujo tipo base é uma subclasse da classe prevista como parâmetro também pode ser passado na chamada da operação;

Figura 59 – Metamodelo de classes de uma DSML para aplicativos de negócio

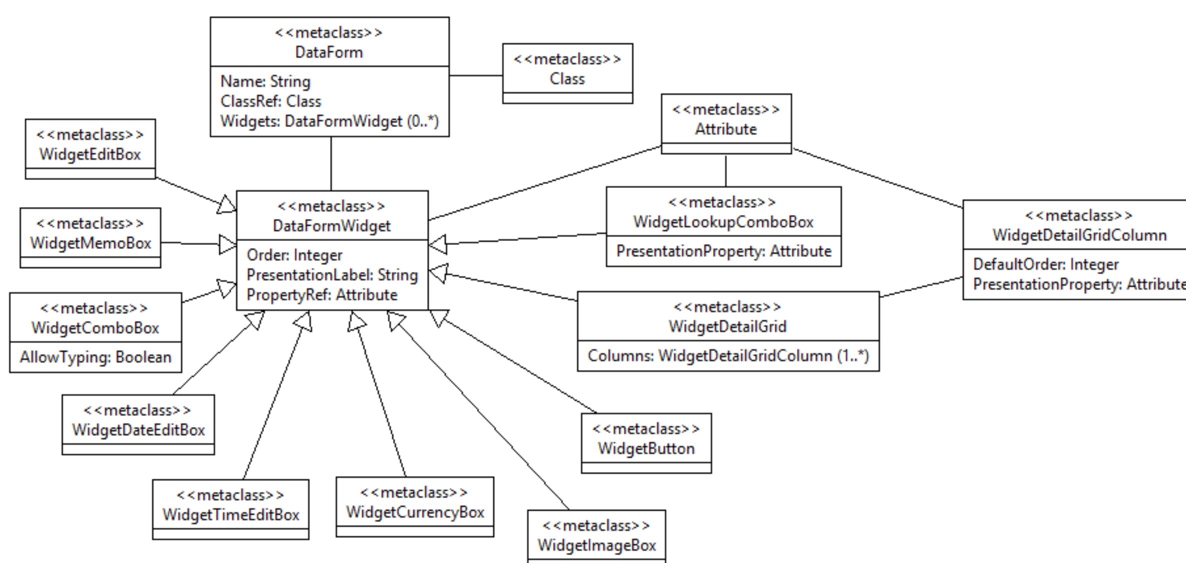


Fonte: Canovas e Cugnasca (2015b)

- **Metamodelo de interfaces de edição de objeto:** Este metamodelo provê elementos para modelar interfaces de usuário de edição de objeto. Cada interface desse tipo corresponde no modelo a uma instância da metaclasse *DataForm*. Independentemente de qualquer implementação concreta (*web*, *desktop*, móvel, etc.), preferencialmente gerada por uma função de mapeamento, um *DataForm* em execução permite o usuário editar as propriedades de um objeto de negócio. O metamodelo de interfaces de edição de objeto está apresentado na Figura 60. Um *DataForm* possui *widgets*. *Widgets* são componentes gráficos como caixas de edição de texto e caixas de checagem (marcado/desmarcado) que funcionam como editores de valores de propriedades. Funções de mapeamento tomam as instâncias de *DataForm* para gerar implementações concretas em uma linguagem alvo. Uma função de mapeamento que produz código em Visual Basic para o

sistema operacional Windows, por exemplo, mapearia cada instância de *WidgetEditBox* para um controle *TextBox* da linguagem. Outra função de mapeamento para *web* mapearia a mesma instância em um trecho de código HTML do tipo *<input type = ‘text’...*>;

Figura 60 – Metamodelo de interfaces de edição de objeto de uma DSML para aplicativos de negócio

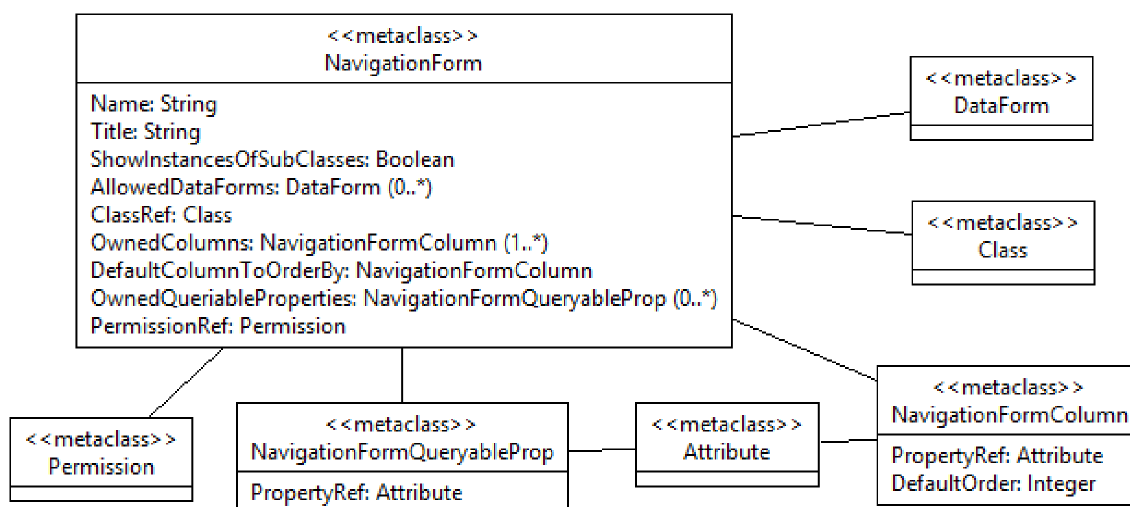


Fonte: Canovas e Cugnasca (2015b)

- **Metamodelo de interfaces de navegação de objetos:** Este metamodelo provê elementos para modelar interfaces de usuário de navegação de objetos e está representado na Figura 61. Por simplicidade para fins desse exemplo, assume-se que uma interface desse tipo, instância de *NavigationForm*, mostra objetos persistentes em uma lista ou uma grade com colunas. Cada interface de navegação está associada a uma classe, que determina quais objetos persistentes devem ser exibidos. Objetos de suas subclasses também devem ser apresentados se a propriedade *ShowInstancesOfSubClasses* for verdadeira. As células dessa grade contêm valores das propriedades dos objetos persistentes, e são representadas por instâncias das submetaclasses de *NavigationFormColumn*. Cada coluna é associada a uma propriedade e especifica um rótulo de apresentação, determinado pelo valor de *PresentationLabel*, o qual pode ser mapeado ao título da coluna a ser exibido

para o usuário. Em geral, deseja-se especificar um rótulo de apresentação mais amigável que o nome da propriedade no modelo. A metaclasse *NavigationForm* está associada à metaclasse *DataForm*, proveniente do metamodelo de interfaces de edição de objetos, com multiplicidade um-para-muitos. As instâncias de *DataForm* de fato associadas no modelo da aplicação correspondem às interfaces de edição de objetos permitidas quando o usuário tenta editar ou criar um objeto persistente. Usualmente, um único *DataForm* é suficiente, mas se a interface de navegação de objetos também permite subclasses (p. ex., uma interface de navegação de clientes onde a classe *Cliente* possui subclasses representando tipos especiais de clientes), então mais de um *DataForm* especializado pode ser necessário. A metaclasse *Permission* permite instanciar no modelo objetos que modelam informações de privilégio de usuário. Cada instância de *NavigationForm* deve estar associada a uma instância dessa metaclasse, que provém do metamodelo de permissões de usuário;

Figura 61 – Metamodelo de interfaces de navegação de objetos de uma DSML para aplicativos de negócio

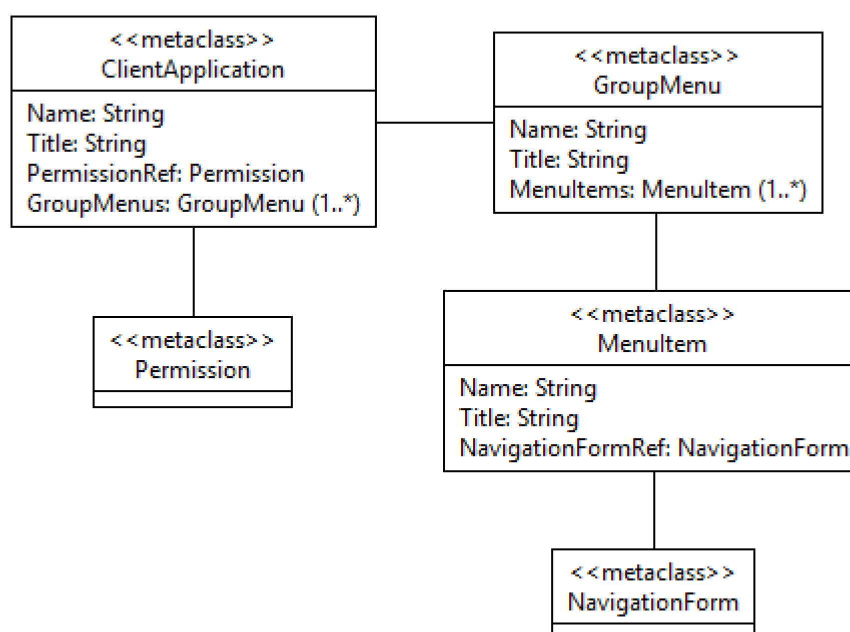


Fonte: Canovas e Cugnasca (2015b)

- **Metamodelo de aplicações *client***: Este metamodelo está apresentado na Figura 62 e provê elementos para modelar aspectos básicos de aplicações *client*, que são, para propósitos deste exemplo, compostas por uma hierarquia

de dois níveis de menus. Cada item de menu, instância de *MenuItem*, deve estar associado a uma única instância de *NavigationForm*. Define-se a semântica de que a interface de navegação de objetos correspondente é apresentada ao usuário quando o mesmo acessa o menu correspondente, e então ele pode navegar pelos objetos persistentes, utilizando as funcionalidades de criação, edição e exclusão de acordo com suas permissões de acesso;

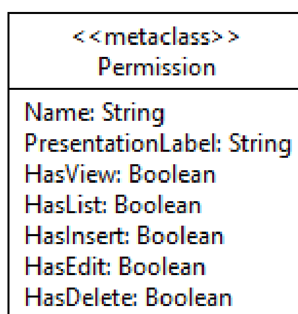
Figura 62 – Metamodelo de aplicações client de uma DSML para aplicativos de negócio



Fonte: Canovas e Cugnasca (2015b)

- **Metamodelo de permissões de usuário:** Este metamodelo provê um elemento para modelar privilégios de usuário, e está apresentado na Figura 63. Cada instância da metaclasse *Permission* representa uma configuração de permissão. Essa metaclasse está associada a *NavigationForm* com multiplicidade 1, conforme Figura 61, o que significa que uma implementação concreta conhece qual configuração de permissão deve ser checada em tempo de execução antes de permitir ou não o acesso à operação específica tal como criação ou exclusão de objeto.

Figura 63 – Metamodelo de permissões de usuário de uma DSML para aplicativos de negócio



Fonte: Canovas e Cugnasca (2015b)

Com os metamodelos definidos, resta estabelecer funções de mapeamento em alguma linguagem alvo para que seja possível a geração automática de código-fonte em pelo menos uma linguagem alvo. As funções de mapeamento devem ser escritas levando-se em consideração a semântica pretendida pelos metamodelos.

Para os propósitos deste exemplo, escolheu-se uma arquitetura e uma linguagem de programação alvo para apresentar algumas funções de mapeamento. Algumas dessas escolhas são arbitrárias. Outro desenvolvedor pode preferir outras opções conforme suas necessidades. Nesse caso, as funções de mapeamento apresentadas podem ser utilizadas como base para adaptação ou novas podem ser criadas.

Adotou-se a arquitetura de três camadas, mencionada anteriormente, a qual determina que o aplicativo de negócio seja dividido em três camadas principais: apresentação, negócio e dados.

As funções de mapeamento criadas nesse exemplo de aplicação geram código para esta arquitetura, e o código gerado trabalha sobre um conjunto de bibliotecas pré-existentes de suporte que compõem um *framework*, não sendo parte do código gerado.

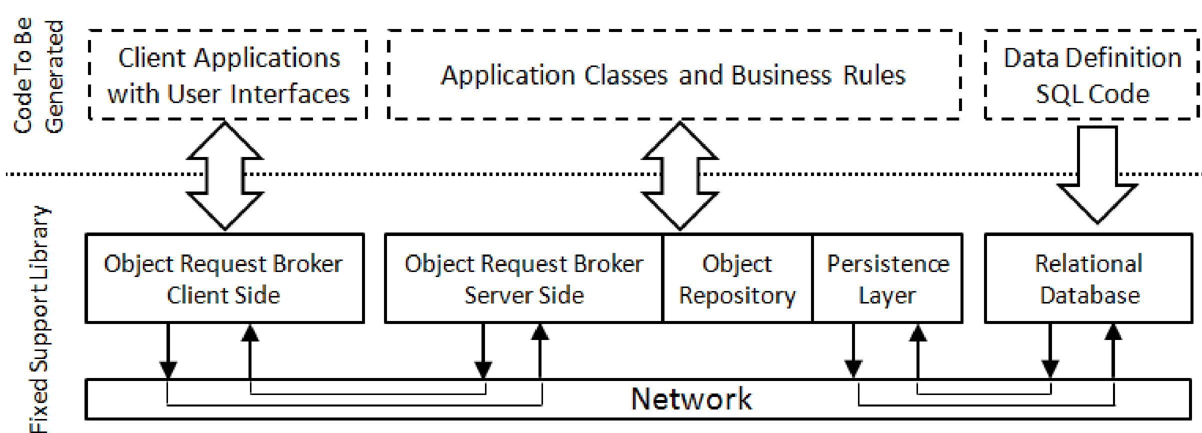
Esse *framework* provê um *middleware* que possibilita a troca de dados entre a camada de apresentação (cliente) e a camada de negócio (servidor), permitindo acesso aos objetos de negócio e, portanto, funcionando como um *Object Request Broker* (ORB). Uma camada de persistência de objetos, ou *Object Persistence Layer* (OPL) também está disponível no *framework* para dar suporte ao código gerado para a camada de negócio poder utilizar persistência de objetos em um banco de dados

relacional de forma transparente. A Figura 64 ilustra essa arquitetura, destacando em retângulos pontilhados o código que deve ser gerado pelas funções de mapeamento a partir dos modelos.

O ambiente de desenvolvimento Lazarus (FPT, 2015) foi usado em conjunto com o compilador Free Pascal Compiler (FPT, 2010) para criar a biblioteca de suporte identificada na Figura 64.

Basicamente, o servidor oferece seis operações sobre objetos de negócio: criação (instanciação), recuperação, busca, atualização, exclusão e chamada de método. A implementação real das classes e as instâncias dos objetos são localizadas no lado do servidor, ou seja, na camada de negócio. O lado cliente recebe cópias de dados dos objetos, isto é, os valores de suas propriedades, em operações de criação, recuperação ou busca. Quando um método de um objeto é invocado por qualquer aplicação *client*, o lado cliente do ORB é responsável por traduzir em uma chamada de procedimento remoto, suportada pelo *middleware*. O método é então executado no lado do servidor, e o resultado devolvido ao cliente como parte do processo de chamada remota.

Figura 64 – Metamodelo de permissões de usuário de uma DSML para aplicativos de negócio



Fonte: autor

As funções de mapeamento foram escritas em MOFM2T. Apesar de este exemplo gerar código apenas para a plataforma Windows na modalidade *desktop*, com a

linguagem alvo sendo o Pascal, outras funções de mapeamento também podem ser escritas para outras linguagens e plataformas alvo usando a ferramenta SBMMTool.

As funções de mapeamento deste exemplo estão divididas em três blocos:

- **Código SQL para definição de dados:** Cada classe persistente no modelo é mapeada em uma tabela com o mesmo nome. Apenas propriedades públicas nas classes persistentes são consideradas para serem mapeadas em campos. Na primeira execução, a função de mapeamento gera declarações do tipo *CREATE TABLE*. Nas próximas execuções, caso classes correspondentes a tabelas já existentes tenham sido alteradas, são geradas declarações do tipo *ALTER TABLE* de modo a preservar os dados existentes, que seriam excluídos em caso de recriação das tabelas;
- **Classes e regras de negócio:** Toda classe no modelo é mapeada em uma declaração de classe na linguagem alvo. Classes não persistentes são derivadas de *TBusinessObject*, enquanto classes persistentes estendem *TPersistentObject*. Propriedades públicas no modelo são mapeadas em propriedades no código-fonte que contam atributos privados correspondentes para armazenar valores. Devido a uma característica da linguagem alvo, propriedades públicas são colocadas na seção *published* pois o mecanismo de reflexão do compilador Free Pascal só funciona para este tipo de visibilidade. O metamodelo de classes utilizado neste exemplo não oferece um meio para implementar operações de classes diretamente no modelo. Uma extensão futura do trabalho poderia incluir, por exemplo, metaclasses da UML relacionadas a ações, tais como os elementos do diagrama de atividades. Outra forma seria prever propriedades para armazenar código em *Action Language for Foundational UML* (ALF) no modelo, traduzindo-o de algum modo em código da linguagem alvo. Neste caso, cada operação que existe no modelo da aplicação é mapeada em uma declaração de método vazia com um bloco protegido de conteúdo, permitindo o programador implementar cada operação diretamente na linguagem alvo sem perder o código gerado em próximas execuções de transformações;

- **Aplicações *client* e interfaces de usuário:** Duas funções de mapeamento são responsáveis por gerar interfaces de edição de objetos. Duas funções foram necessárias pois o ambiente de desenvolvimento Lazarus armazena código em um arquivo .pas e informações dos *widgets* em um arquivo .lfm. O código gerado contém instruções para requisitar à camada cliente do ORB por um objeto específico quando a interface é aberta. Após receber os dados do objeto, os *widgets* são populados. Quando o usuário pressiona o botão de confirmação que faz parte da interface, validações de campos requeridos são feitas, de acordo com o valor do campo *Required* das instâncias de *Property*, e dados atualizados do objeto são enviados de volta para o servidor por uma outra chamada ao cliente do ORB. De modo similar, interfaces de navegação de objetos são geradas por outras duas funções de mapeamento. Cada tela de navegação contém um botão de busca que abre outra janela (parte da biblioteca de suporte mostrada na Figura 64), a qual permite que o usuário monte expressões de busca. Por exemplo, o usuário pode usar esse botão em uma interface de navegação de ordens de venda para localizar apenas as ordens de um cliente específico ou em determinado intervalo de datas. Ao selecionar um objeto e acessar o botão de edição, a interface de edição de objeto correspondente é apresentada. Outras funções de mapeamento foram criadas para gerar aplicações *client*. Elas geram código para aplicações Windows *desktop* com funções de *login* e um conjunto de menus para abrir as interfaces de navegação presentes no modelo.

O Quadro 32 reproduz a função de mapeamento do bloco de classes e regras de negócio. Outros detalhes podem se encontrados em Canovas e Cugnasca (2015b), que também apresenta um exemplo de construção de um aplicativo de negócio do contexto do Laboratório de Automação Agrícola da Escola Politécnica, o WebBee, utilizando como base os metamodelos aqui apresentados.

Quadro 32 – Função de mapeamento do bloco de classes e regras de negócio

```
1  [template public ModelTypeToLanguageType(t: EType, IsArray:
   Boolean)][if(not (IsArray = 'True'))][if (t =
   'String')]string[/if][if (t = 'Memo')]string[/if][if (t =
   'Integer')]Integer[/if][if (t = 'Double')]Double[/if][if (t =
   'Boolean')]Boolean[/if][if (t = 'Blob')]TP2SBFBlob[/if][if (t =
   'BizObject')]TP2SBFAbsBizObject[/if][/if][if(IsArray =
   'True')][if (t = 'String')]TStringDynArray[/if][if (t =
   'Memo')]TStringDynArray[/if][if (t =
   'Integer')]TIntegerDynArray[/if][if (t =
   'Double')]TDoubleDynArray[/if][if (t =
   'Boolean')]TBooleanDynArray[/if][if (t =
   'Blob')]TP2SBFBlobDynArray[/if][if (t =
   'BizObject')]TP2SBFAbsBizObjDynArray[/if][/if][/template]
2
3  [template public AttributeFieldSignature(a:
   Attribute)]F[value(a.Name) /]: [call
   ModelTypeToLanguageType(a.Type, 'False')/][/template]
4
5  [template public AttributeSignature(a: Attribute)][if(a.Kind =
   'Property')]property [/if][value(a.Name) /]: [call
   ModelTypeToLanguageType(a.Type, 'False')/][if(a.Kind =
   'Property')] read F[value(a.Name) /] write F[value(a.Name)
   /][/if][/template]
6
7  [template public OperationSignature(o: Operation, ServerSide:
   Boolean, ClassName: String)][if(o.Static = 'True')]class
   [/if][if((o.ReturnType = 'None') or (o.ReturnType =
   ''))]procedure[/if][if(not ((o.ReturnType = 'None') or
   (o.ReturnType = '')))]function[/if]
   [if(ClassName<>'')][value(ClassName) /].[/if][value(o.Name)
   /]([if(ServerSide = 'True')]AConnectionData:
   TP2SBFConnectionData[/if][for(p:OperationParameter|o.Parameters
   )][if((position > 0) or (ServerSide = 'True'))];
   [/if][value(p.Name) /]: [call
   ModelTypeToLanguageType(p.Type,p.IsArray) /][/for])[if(not
   ((o.ReturnType = 'None') or (o.ReturnType = '')))]: [call
   ModelTypeToLanguageType(o.ReturnType,'False')
   /][/if][/template]
8
9  [template public ParentClassName(c:
   Class)][if(c.ParentClass)][value(c.ParentClass[0].Name)
   /][/if][if(not (c.ParentClass))][if(c.Persistent =
   'True')]TP2SBFAbsPerBizObj[/if][if(not (c.Persistent =
   'True'))]TP2SBFAbsBizObj[/if][/if][/template]
10
11 [template public AttributeFieldList(c: Class, v:
   EVisibility)][for(a:Attribute|c.Attributes)][if((a.Visibility =
   v) and (a.Kind = 'Property'))]      [call
   AttributeFieldSignature(a) /];
12 [/if][/for][/template]
13
```

```
14  [template public AttributeList(c: Class, v:
    EVisibility)][for(a:Attribute|c.Attributes)][if(a.Visibility =
    v)]      [call AttributeSignature(a) /];
15  [/if][/for][/template]
16
17  [template public OperationList(c: Class, v: EVisibility,
    ServerSide:
    Boolean)][for(o:Operation|c.Operations)][if(o.Visibility = v)]
    [call OperationSignature(o,ServerSide,'False') /]
    stdcall;[if(o.Override = 'True')] override;[/if]
18  [/if][/for][/template]
19
20  [template public AssociationEnd1AttributeType(a:
    Association)][if(a.Kind = 'Reference')][if(a.Multiplicity =
    'ZeroOrMany_To_ZeroOrOne')]TP2SBFBizObjReference[/if][/if][if(a
    .Kind = 'Composition')][if(a.Multiplicity =
    'ZeroOrOne_To_ZeroOrMany')]TP2SBFRelationship1N[/if][if(a.Multi
    plicity =
    'ZeroOrOne_To_ZeroOrOne')][value(a.AssociationEnd2.Name)
    /][/if][/if][/template]
21
22  [template public AssociationEnd2AttributeType(a:
    Association)][if((a.Multiplicity = 'ZeroOrOne_To_ZeroOrOne') or
    (a.Multiplicity =
    'ZeroOrOne_To_ZeroOrMany'))]TP2SBFBizObjReference[/if][/templat
    e]
23
24  [template public AssociationEnd1Signature(a: Association,
    IsProperty: Boolean)][if(IsProperty = 'True')]property
    [/if][if(not (IsProperty =
    'True'))]F[/if][value(a.AssociationEnd1AttributeName) /]: [call
    AssociationEnd1AttributeType(a) /][if(IsProperty = 'True')]
    read F[value(a.AssociationEnd1AttributeName) /] write
    F[value(a.AssociationEnd1AttributeName) /][/if][/template]
25
26  [template public AssociationEnd2Signature(a: Association,
    IsProperty: Boolean)][if(IsProperty = 'True')]property
    [/if][if(not (IsProperty =
    'True'))]F[/if][value(a.AssociationEnd2AttributeName) /]: [call
    AssociationEnd2AttributeType(a) /][if(IsProperty = 'True')]
    read F[value(a.AssociationEnd2AttributeName) /] write
    F[value(a.AssociationEnd2AttributeName) /][/if][/template]
27
28  [template public AssociationAttributeList(c: Class, IsProperty:
    Boolean)][for(a:Association|Association)][if(a.AssociationEnd1.
    Name = c.Name)]      [call
    AssociationEnd1Signature(a,IsProperty) /];
29  [/if][/for]
30  [for(a:Association|Association)][if((a.AssociationEnd2.Name =
    c.Name) and (a.Bidirectional = 'True'))]      [call
    AssociationEnd2Signature(a,IsProperty) /];
31  [/if][/for][/template]
```

```
32
33 [template public AssociationEnd1AttributeCreate(a:
   Association)]F[value(a.AssociationEnd1AttributeName)
   /]:=[if(a.Kind = 'Reference')][if(a.Multiplicity =
   'ZeroOrMany_To_ZeroOrOne')]TP2SBFBizObjReference.Create[/if][/i
   f][if(a.Kind = 'Composition')][if(a.Multiplicity =
   'ZeroOrOne_To_ZeroOrMany')]TP2SBFRelationship1N.Create(Self,[va
   lue(a.AssociationEnd1.Name) /],[value(a.AssociationEnd2.Name)
   /],'[value(a.name) /]',[if(a.Bidirectional =
   'True')]True,'[value(a.AssociationEnd2AttributeName)
   /]'[/if][if(not (a.Bidirectional =
   'True'))]False,''[/if])[/if][if(a.Multiplicity =
   'ZeroOrOne_To_ZeroOrOne')][value(a.AssociationEnd2.Name)
   /].Create(....)[/if][/if][/template]
34
35 [template public AssociationEnd2AttributeCreate(a:
   Association)]F[value(a.AssociationEnd2AttributeName)
   /]:=[if((a.Multiplicity = 'ZeroOrOne_To_ZeroOrOne') or
   (a.Multiplicity =
   'ZeroOrOne_To_ZeroOrMany'))]TP2SBFBizObjReference.Create[/if][/
   template]
36
37 [template public AssociationAttributeCreateList(c:
   Class)][for(a:Association|Association)][if(a.AssociationEnd1.Na
   me = c.Name)]   [call AssociationEnd1AttributeCreate(a) /];
38 [/if][/for]
39 [for(a:Association|Association)][if((a.AssociationEnd2.Name =
   c.Name) and (a.Bidirectional = 'True'))]   [call
   AssociationEnd2AttributeCreate(a) /];
40 [/if][/for][/template]
41
42 [template public AssociationEnd1AttributeDestroy(a:
   Association)]F[value(a.AssociationEnd1AttributeName)
   /].[if(a.Kind = 'Reference')][if(a.Multiplicity =
   'ZeroOrMany_To_ZeroOrOne')]Free[/if][/if][if(a.Kind =
   'Composition')][if(a.Multiplicity =
   'ZeroOrOne_To_ZeroOrMany')]Free[/if][if(a.Multiplicity =
   'ZeroOrOne_To_ZeroOrOne')][value(a.AssociationEnd2.Name)
   /]Free[/if][/if][/template]
43
44 [template public AssociationEnd2AttributeDestroy(a:
   Association)]F[value(a.AssociationEnd2AttributeName)
   /].[if((a.Multiplicity = 'ZeroOrOne_To_ZeroOrOne') or
   (a.Multiplicity =
   'ZeroOrOne_To_ZeroOrMany'))]Free[/if][/template]
45
46 [template public AssociationAttributeDestroyList(c:
   Class)][for(a:Association|Association)][if(a.AssociationEnd1.Na
   me = c.Name)]   [call AssociationEnd1AttributeDestroy(a) /];
47 [/if][/for]
48 [for(a:Association|Association)][if((a.AssociationEnd2.Name =
   c.Name) and (a.Bidirectional = 'True'))]   [call
```

```
   AssociationEnd2AttributeDestroy(a) /];
49 [/if][/for][/template]
50
51 [template public ReferenceTriggerSignature(a: Association,
   IsProperty: Boolean)][if(IsProperty = 'True')]property
   [/if][if(not (IsProperty = 'True'))]F[/if]DT[value(a.Name) /]:
   TP2SBFReferenceTrigger[if(IsProperty = 'True')] read
   FDT[value(a.Name) /] write FDT[value(a.Name) /][/if][/template]
52
53 [template public ReferenceTriggerList(c: Class, IsProperty:
   Boolean)][for(a:Association|Association)][if(((a.AssociationEnd
   1.Name = c.Name) and (a.Kind = 'Reference')) or
   ((a.AssociationEnd2.Name = c.Name) and (a.Bidirectional =
   'True')))]      [call ReferenceTriggerSignature(a,IsProperty)
   /];
54 [/if][/for][/template]
55
56 [template public ReferenceTriggerCreateForAssociationEnd1(a:
   Association)]FDT[value(a.Name)
   /]:=TP2SBFReferenceTrigger.Create(Self,[value(a.AssociationEnd2
   .Name) /],[value(a.AssociationEnd1.Name) /],'[value(a.Name)
   /]','',[value(a.AssociationEnd1AttributeName) /])[/template]
57
58 [template public ReferenceTriggerCreateForAssociationEnd2(a:
   Association)]FDT[value(a.Name)
   /]:=TP2SBFReferenceTrigger.Create(Self,[value(a.AssociationEnd1
   .Name) /],[value(a.AssociationEnd2.Name) /],'[value(a.Name)
   /]','',[value(a.AssociationEnd2AttributeName) /])[/template]
59
60 [template public ReferenceTriggerCreateList(c:
   Class)][for(a:Association|Association)][if((a.AssociationEnd1.N
   ame = c.Name) and (a.Kind = 'Reference'))]   [call
   ReferenceTriggerCreateForAssociationEnd1(a) /];
61 [/if][if((a.AssociationEnd2.Name = c.Name) and (a.Bidirectional
   = 'True'))]   [call ReferenceTriggerCreateForAssociationEnd2(a)
   /];
62 [/if][/for][/template]
63
64 [template public ReferenceTriggerDestroy(a:
   Association)]FDT[value(a.Name) /].Free[/template]
65
66 [template public ReferenceTriggerDestroyList(c:
   Class)][for(a:Association|Association)][if(((a.AssociationEnd1.
   Name = c.Name) and (a.Kind = 'Reference')) or
   ((a.AssociationEnd2.Name = c.Name) and (a.Bidirectional =
   'True')))]   [call ReferenceTriggerDestroy(a) /];
67 [/if][/for][/template]
68
69 [template public DeleteVerificationSignature(a: Association,
   IsProperty: Boolean)][if(IsProperty = 'True')]property
   [/if][if(not (IsProperty = 'True'))]F[/if]DV[value(a.Name) /]:
```

```
   TP2SBFDeleteVerification[if(IsProperty = 'True')] read
   FDT[value(a.Name) /] write FDT[value(a.Name) /][/if][/template]
70
71 [template public DeleteVerificationList(c: Class, IsProperty:
   Boolean)][for(a:Association|Association)][if((a.AssociationEnd2
   .Name = c.Name) or ((a.AssociationEnd1.Name = c.Name) and
   (a.Kind = 'Reference') and (a.Bidirectional = 'True')))]
   [call DeleteVerificationSignature(a,IsProperty) /];
72 [/if][/for][/template]
73
74 [template public DeleteVerificationCreateForAssociationEnd1(a:
   Association)]FDV[value(a.Name)
   /]:=TP2SBFDeleteVerification.Create(Self,[value(a.AssociationEn
   d2.Name) /],[value(a.AssociationEnd1.Name) /],'[value(a.Name)
   /]',nil,'[value(a.AssociationEnd1AttributeName) /]')[/template]
75
76 [template public DeleteVerificationCreateForAssociationEnd2(a:
   Association)]FDV[value(a.Name)
   /]:=TP2SBFDeleteVerification.Create(Self,[value(a.AssociationEn
   d1.Name) /],[value(a.AssociationEnd2.Name) /],'[value(a.Name)
   /]',nil,'[value(a.AssociationEnd2AttributeName) /]')[/template]
77
78 [template public DeleteVerificationCreateList(c:
   Class)][for(a:Association|Association)][if((a.AssociationEnd2.N
   ame = c.Name))]   [call
   DeleteVerificationCreateForAssociationEnd1(a) /];
79 [/if][if((a.AssociationEnd1.Name = c.Name) and (a.Kind =
   'Reference') and (a.Bidirectional = 'True'))]   [call
   DeleteVerificationCreateForAssociationEnd2(a) /];
80 [/if][/for][/template]
81
82 [template public DeleteVerificationDestroy(a:
   Association)]FDV[value(a.Name) /].Free[/template]
83
84 [template public DeleteVerificationDestroyList(c:
   Class)][for(a:Association|Association)][if((a.AssociationEnd2.N
   ame = c.Name))]   [call DeleteVerificationDestroy(a) /];
85 [/if][if((a.AssociationEnd1.Name = c.Name) and (a.Kind =
   'Reference') and (a.Bidirectional = 'True'))]   [call
   DeleteVerificationDestroy(a) /];
86 [/if][/for][/template]
87
88 [template public AssociationEnd1AttributeRegister(a:
   Association)][if(a.Kind = 'Reference')][if(a.Multiplicity =
   'ZeroOrMany_To_ZeroOrOne')]gP2SBFClassRegistry.RegisterObjectPr
   operty([value(a.AssociationEnd1.Name)
   /],'[value(a.AssociationEnd1AttributeName)
   /]',[value(a.AssociationEnd2.Name) /],'[value(a.Name)
   /]')[/if][/if][if(a.Kind = 'Composition')][if(a.Multiplicity =
   'ZeroOrOne_To_ZeroOrMany')]gP2SBFClassRegistry.Register1NObject
   Property([value(a.AssociationEnd1.Name)
   /],'[value(a.AssociationEnd1AttributeName)
```

```
   /]',[value(a.AssociationEnd2.Name) /],'[value(a.Name)
   /]')[/if][if(a.Multiplicity =
   'ZeroOrOne_To_ZeroOrOne')]gP2SBFClassRegistry.RegisterObjectPro
   perty(....)[/if][/if][/template]
89
90 [template public AssociationEnd2AttributeRegister(a:
   Association)][if((a.Multiplicity = 'ZeroOrOne_To_ZeroOrOne') or
   (a.Multiplicity =
   'ZeroOrOne_To_ZeroOrMany'))]gP2SBFClassRegistry.RegisterObjectP
   roperty([value(a.AssociationEnd2.Name)
   /],'[value(a.AssociationEnd2AttributeName)
   /]',[value(a.AssociationEnd1.Name) /],'[value(a.Name)
   /]')[/if][/template]
91
92 [template public AssociationAttributeRegisterList(c:
   Class)][for(a:Association|Association)][if(a.AssociationEnd1.Na
   me = c.Name)]   [call AssociationEnd1AttributeRegister(a) /];
93 [/if][/for]
94 [for(a:Association|Association)][if((a.AssociationEnd2.Name =
   c.Name) and (a.Bidirectional = 'True'))]   [call
   AssociationEnd2AttributeRegister(a) /];
95 [/if][/for][/template]
96
97 [template top public BusinessModel_ServerSide()]
98 [file('SourceCode\\Server\\uModel.pas',0)]unit uModel;
99
100 interface
101
102 uses Types, SyncObjs,
103      uP2SBFAbsModelTypes,uP2SBFAbsModel,uP2SBFClassRegistry,
    uP2SBFObjRepos,
104      uP2SBFOrbServerTypes,uP2SBFBizEventManager,uP2SBFUtils,
    uP2SBFSystemModel,
105      uP2SBFOrbServerUIManager,uP2SBFParams,
    uP2SBFLicenseChecker,uP2SBFOrbServer,
106 [protected('UNIT_HEADER_USER')]     Classes, SysUtils;
107 [/protected]
108 type
109
110 // Forward declarations
111 [for(c:Class|Class)][value(c.Name) /] = class;
112 [/for]
113
114 [for(c:Class|Class)]I[value(c.Name) /] = interface(IInvokable)
115 [call OperationList(c,'published','True') /]
116 end;
117
```

```
118 [value(c.Name) /] = class([call ParentClassName(c)
    /],I[value(c.Name) /])
119    private
120 [call AttributeList(c,'private') /][call
    OperationList(c,'private','True') /]
121    protected
122 [call ReferenceTriggerList(c,'False') /]
123 [call DeleteVerificationList(c,'False') /]
124 [call AssociationAttributeList(c,'False') /]
125 [call AttributeFieldList(c,'public') /][call
    AttributeFieldList(c,'published') /][call
    AttributeList(c,'protected') /][call
    OperationList(c,'protected','True') /]
126       procedure CreateAttributes(AConnectionData:
    TP2SBFConnectionData); override;
127       procedure CreateAttributesForRetrieve; override;
128       procedure DestroyAttributes(AConnectionData:
    TP2SBFConnectionData); override;
129       procedure BeforeDelete(AConnectionData:
    TP2SBFConnectionData); override;
130       procedure AfterDelete(AConnectionData:
    TP2SBFConnectionData); override;
131       function  CustomCanDelete(AConnectionData:
    TP2SBFConnectionData; var AReason: string): Boolean; override;
132       procedure BeforeSave(AConnectionData:
    TP2SBFConnectionData; ANew: Boolean); override;
133       procedure AfterSave(AConnectionData:
    TP2SBFConnectionData; ANew: Boolean; AChangedState:
    TP2SBFChangedState); override;
134    public
135       constructor Create(AConnectionData:
    TP2SBFConnectionData); override;
136 [call AttributeList(c,'public') /][call
    OperationList(c,'public','True') /]
137    Published
138 [call ReferenceTriggerList(c,'True') /]
139 [call DeleteVerificationList(c,'True') /]
140 [call AssociationAttributeList(c,'True') /]
141 [call AttributeList(c,'published') /][call
    OperationList(c,'published','True') /]
142 end;
143
144 [/for]
145
146 implementation
147
148 [protected('IMPLEMENTATION_HEADER_USER')][/protected]
149
```

```
150 [for(c:Class|Class)]
151 //*************************************************************
    ***************
152 //* [value(c.Name) /].Create
153 //*************************************************************
    ***************
154 constructor [value(c.Name) /].Create(AConnectionData:
    TP2SBFConnectionData);[protected('CREATE_'+c.Name)]begin
155    inherited Create(AConnectionData);
156    // User implementation
157 end;
158 [/protected]
159 
160 //*************************************************************
    ***************
161 //* [value(c.Name) /].CreateAttributes
162 //*************************************************************
    ***************
163 procedure [value(c.Name) /].CreateAttributes(AConnectionData:
    TP2SBFConnectionData);
164 begin
165    inherited CreateAttributes(AConnectionData);
166 [call AssociationAttributeCreateList(c) /]
167 [call ReferenceTriggerCreateList(c) /]
168 [call DeleteVerificationCreateList(c) /]
169 end;
170 
171 //*************************************************************
    ***************
172 //* [value(c.Name) /].CreateAttributesForRetrieve
173 //*************************************************************
    ***************
174 procedure [value(c.Name) /].CreateAttributesForRetrieve;
175 begin
176    inherited CreateAttributesForRetrieve;
177 [call AssociationAttributeCreateList(c) /]
178 [call ReferenceTriggerCreateList(c) /]
179 [call DeleteVerificationCreateList(c) /]
180 end;
181 
182 //*************************************************************
    ***************
183 //* [value(c.Name) /].DestroyAttributes
184 //*************************************************************
    ***************
185 procedure [value(c.Name) /].DestroyAttributes(AConnectionData:
```

```
    TP2SBFConnectionData);
186 begin
187 [call AssociationAttributeDestroyList(c) /]
188 [call ReferenceTriggerDestroyList(c) /]
189 [call DeleteVerificationDestroyList(c) /]
190    inherited DestroyAttributes(AConnectionData);
191 end;
192
193 //************************************************************
    ***************
194 //* [value(c.Name) /].BeforeDelete
195 //************************************************************
    ***************
196 procedure [value(c.Name) /].BeforeDelete(AConnectionData:
    TP2SBFConnectionData);[protected('BEFOREDELETE_'+c.Name)]begin
197    inherited BeforeDelete(AConnectionData);
198    // User implementation
199 end;
200 [/protected]
201
202 //************************************************************
    ***************
203 //* [value(c.Name) /].AfterDelete
204 //************************************************************
    ***************
205 procedure [value(c.Name) /].AfterDelete(AConnectionData:
    TP2SBFConnectionData);[protected('AFTERDELETE_'+c.Name)]begin
206    inherited AfterDelete(AConnectionData);
207    // User implementation
208 end;
209 [/protected]
210
211 //************************************************************
    ***************
212 //* [value(c.Name) /].CustomCanDelete
213 //************************************************************
    ***************
214 function  [value(c.Name) /].CustomCanDelete(AConnectionData:
    TP2SBFConnectionData; var AReason: string):
    Boolean;[protected('CUSTOMCANDELETE_'+c.Name)]begin
215    Result := inherited
    CustomCanDelete(AConnectionData,AReason);
216    // User implementation
217 end;
218 [/protected]
219
```

```
220 //*************************************************************
    **************
221 //* [value(c.Name) /].BeforeSave
222 //*************************************************************
    **************
223 procedure [value(c.Name) /].BeforeSave(AConnectionData:
    TP2SBFConnectionData; ANew:
    Boolean);[protected('BEFORESAVE_'+c.Name)]begin
224    inherited BeforeSave(AConnectionData,ANew);
225    // User implementation
226 end;
227 [/protected]
228
229 //*************************************************************
    **************
230 //* [value(c.Name) /].AfterSave
231 //*************************************************************
    **************
232 procedure [value(c.Name) /].AfterSave(AConnectionData:
    TP2SBFConnectionData; ANew: Boolean; AChangedState:
    TP2SBFChangedState);[protected('AFTERSAVE_'+c.Name)]begin
233    inherited AfterSave(AConnectionData,ANew,AChangedState);
234    // User implementation
235 end;
236 [/protected]
237
238 [for(o:Operation|c.Operations)]
239 //*************************************************************
    **************
240 //* [value(c.Name) /].[value(o.Name) /]
241 //*************************************************************
    **************
242 [call OperationSignature(o,'True','True') /]
    stdcall;[protected('USEROPERATION_'+c.Name+'_'+o.Name)]begin
243    // User implementation
244 end;
245 [/protected][/for]
246 [/for]
247
248 procedure Init1;
249 begin
250    // Register Classes, Interfaces and Proxy Properties
251 [for(c:Class|Class)]
    gP2SBFClassRegistry.RegisterClass([value(c.Name)
    /],TypeInfo(I[value(c.Name) /]),'',False);
252 [for(a:Attribute|c.Attributes)][if(a.Proxy = 'True')]
    gP2SBFClassRegistry.RegisterProxyProperty([value(c.Name)
```

```
    /],'[value(a.Name) /]');
253 [/if][/for][/for]
254 end;
255 
256 procedure Init2;
257 begin
258    // Register Object Properties (Reference and Part)
259 [for(c:Class|Class)][call AssociationAttributeRegisterList(c)
    /][/for]
260 end;
261 
262 initialization
263    Init1;
264    Init2;
265 
266    gP2SBFOrbServer.CheckMaxConnections:=False;
267 
268 end.
269 [/file]
270 [/template]
```

Fonte: autor