UNIVERSIDADE FEDERAL DO MARANHÃO
CENTRO DE CIÊNCIA EXATAS E TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA DE ELETRICIDADE

MARCUS VINÍCIUS RIBEIRO DE CARVALHO

UMA ABORDAGEM BASEADA NA ENGENHARIA DIRIGIDA POR MODELOS PARA SUPORTAR MERGING DE BASE DE DADOS HETEROGÊNEAS

São Luís
2014

MARCUS VINÍCIUS RIBEIRO DE CARVALHO

UMA ABORDAGEM BASEADA NA ENGENHARIA DIRIGIDA POR MODELOS PARA SUPORTAR MERGING DE BASE DE DADOS HETEROGÊNEAS

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia de Eletricidade da Universidade Federal do Maranhão, para a obtenção do tï£¡tulo de mestre em Engenharia de Eletricidade - Área de Concentração: Ciência da Computação.

Orientador: Ph. D. Zair Abdelouahab

Co-Orientador: Dr.Denivaldo Lopes

São Luís

2014

Carvalho, Marcus Vinícius Ribeiro de

Uma abordagem baseada na engenharia dirigida por modelo para suportar *merging* de base de dados heterogêneas / Marcus Vinícius Ribeiro de Carvalho, 2014.

171 f.

Impresso por computador (fotocópia).

Orientador: Ph. D. Zair Abdelouahab.

Co-Orientador: Dr.Denivaldo Lopes.

Dissertação (Mestrado) – Universidade Federal do Maranhão, Programa de Pós-Graduação em Engenharia de Eletricidade , 2014.

1. Software - Engenharia. 2. Engenharia Dirigida por Modelos. I. Título.

CDU 004.41

UMA ABORDAGEM BASEADA NA ENGENHARIA DIRIGIDA POR MODELOS PARA SUPORTAR MERGING DE BASE DE DADOS HETEROGÊNEAS

Marcus Vinicius Ribeiro de Carvalho

Dissertação aprovada em 24 de fevereiro de 2014.

Prof. Zair Abdelouahab, Ph.D
(Orientador)

Prof. Denivaldo Cícero Pavão Lopes, Dr.
(Co-Orientador)

Profa. Eveline de Jesus Viana Sá, Dra.
(Membro da Banca Examinadora)

Prof. Marcos Didonet Del Fabro, Dr.
(Membro da Banca Examinadora)

Prof. Slimane Hammoudi, Dr.
(Membro da Banca Examinadora)

Prof. Samyr Beliche Vale, Dr.
(Membro da Banca Examinadora)

Dedico este trabalho a Luciana Coimbra Soares e a meus filhos Samuel Coimbra e Germano Coimbra.

RESUMO

A Engenharia Dirigida por Modelos (MDE) fornece suporte para o gerenciamento da complexidade de desenvolvimento, manutenção e evolução de software, através da criação e transformação de modelos. Esta abordagem pode ser utilizada em outros domínios também complexos como a integração de esquemas de base de dados. Neste trabalho de pesquisa, propomos uma metodologia para integrar *schema* de base de dados no contexto da MDE. Metamodelos para definição de *database model*, *database model matching*, *database model merging*, *integrated database model* são propostos com a finalidade de apoiar a metodologia. Um algoritmo para *database model matching* e um algoritmo para *database model merging* são apresentados. Apresentamos ainda, um protótipo que adapta e estende as ferramentas MT4MDE e SAMT4MDE a fim de demonstrar a implementação do *framework*, metodologia e algoritmos propostos. Um exemplo ilustrativo ajuda a melhor entender a metodologia apresentada, servindo para explicar os metamodelos e algoritmos propostos neste trabalho. Uma breve avaliação do *framework* e diretrizes futuras sobre este trabalho são apresentadas.

Palavras-chave: Engenharia dirigida por modelos, Correspondência de metamodelos e modelos, Integração de base de dados.

ABSTRACT

Model Driven Engineering (MDE) aims to make face to the development, maintenance and evolution of complex software systems, focusing in models and model transformations. This approach can be applied in other domains such as database schema integration. In this research work, we propose a framework to integrate database schema in the MDE context. Metamodels for defining database model, database model matching, database model merging, and integrated database model are proposed in order to support our framework. An algorithm for database model matching and an algorithm for database model merging are presented. We present also, a prototype that extends the MT4MDE and SAMT4MDE tools in order to demonstrate the implementation of our proposed framework, metodology, and algorithms. An illustrative example helps to understand our proposed framework.

Keywords: Model Driven Engineering, Schema Matching, Database Integration, Metamodel and Model Matching.

Agradecimentos

A Deus, por ter me dado forças e perseverança para conclusão do curso.
Ao meu orientador, professor Ph.D. Zair Abdelouahab, pela oportunidade concedida, pela orientação, pelos conselhos e pelo conhecimento repassado. Os mais sinceros agradecimentos.
Ao meu co-orientador, professor Dr. Denivaldo Lopes, pela oportunidade concedida, pela orientação, pela paciência, compreensão, pelo conhecimento repassado e pelo incentivo ao longo deste período. Meus sinceros agradecimentos.
Ao programa de Pós-Graduação em Engenharia de Eletricidade da Universidade Federal do Maranhão, pela oportunidade da realização do curso.
Aos integrantes e ex-integrantes do LESERC, em especial Pablo Matos, Steve Ataky, Jean Pablo, Sasha Nícolas, Gerson Lobato, Amanda Serra, Gustavo Dominices e Wesley Araujo que contribuíram direta ou indiretamente para realização desse trabalho.
Um especial agradecimento a amiga Jéssica Bassani pela amizade, pelas contribuições no trabalho, pelo o apoio no laboratório e incentivo.
Aos amigos Suzane Carvalho, Arikleyton Ferreira, Luiz Aurélio, Dhully Andrade, Ariel Soares e Gleison Medeiros pela ajuda nas disciplinas e companheirismo demonstrados.
Aos ex-companheiros de mestrado Vladimir Oliveira e Amélia Acácia pelo apoio ao iniciar o mestrado.
A Sra. Marília Futino pelo apoio na minha liberação do trabalho para cursar o mestrado.
A amiga Anaide Galiza pela ajuda e apoio que permitiram a minha ida para São Luís.
A minha esposa Luciana Coimbra pelo apoio durante a realização do curso. Sem esse apoio não seria possível concluir a pós-graduação.
Ao meu irmão Afonso Carvalho e sua esposa Sandriane Ferreira e ao meu irmão Jivago Carvalho e sua esposa Fernanda Amaral pela acolhida nesses dois anos de curso.
A todos que direta ou indiretamente contribuíram para a realização deste trabalho.

"O impossível é questão de tempo".

Alberto Saltiel

Lista de Figuras

Lista de Tabelas

Lista de Siglas

API Application Programming Interface.

ATL Atlas Transformation Language.

DBMS Database Management System.

DDL Data Definition Language.

DER Diagrama de Entidade e Relacionamento.

DML Data Manipulation Language.

EMF Eclipse Modeling Framework.

MDA Model-Driven Architecture.

MDE Model-Driven Engineering.

MT4MDE Mapping Tool for MDE.

OMG Object Management Group.

PIM Platform Independent Model.

PSM Platform Specific Model.

SAMT4MDE Semi-Automatic Matching Tool for MDE.

SID4MDE Semi-Automatic Integration Database for MDE.

SQL Structured Query Language.

SUS Sistema Único de Saúde.

UML Unified Modeling Language.

XML eXtensible Markup Language.

Sumário

1 INTRODUÇÃO

Neste capítulo, o contexto em que a dissertação foi desenvolvida será apresentado, expondo a problemática a ser tratada, bem como, a solução proposta e os objetivos. A metodologia empregada para a execução do trabalho de pesquisa também será mostrada, além da apresentação dos demais capítulos desta dissertação.

1.1 Contexto

Sistemas de informação estão presentes em organizações públicas ou privadas. É comum as organizações terem sistemas de informação que atendam setores distintos, como por exemplo, setor contábil, setor de venda e setor de recursos humanos. Todos esses sistemas geram um grande volume de informações que nem sempre são bem aproveitados pelos Gestores das Organizações.

Os Gestores precisam de uma visão unificada dos dados presentes em diversos sistemas, como os sistemas são normalmente desenvolvidos por equipes diferentes, ou ainda, quando são desenvolvidos pela mesma equipe estas esquecem de garantir uma integração de suas bases de dados. A falta de integração de bases de dados é um problema muito comum nas organizações e dificulta o aproveitamento de toda gama de dados gerada pelos sistemas de informação.

Para integrar bases de dados heterogêneas é necessário encontrar correspondências entre seus esquemas (*schema matching*). Correspondência desempenha um papel central em diversas aplicações, tais como a integração de dados orientado a *web*, *E-Commerce*, integração de esquema, evolução de esquema e migração, evolução de aplicativos, armazenamento de dados, *design* de banco de dados, criação de *web site* e de gestão e desenvolvimento baseado em componentes [55].

As bases de dados e aplicações estão cada vez mais complexas, aumentando assim, o esforço do especialista no domínio para fazer correspondências entre esquemas [55] visando sua integração. A integração de esquemas de base de dados vem sendo estudada por um longo tempo [5]. Ferramentas para auxiliar o trabalho de cor-

respondência vem sendo desenvolvidas como : Cupid [38], COMA++ [15], bem como, algoritmos foram propostos como o apresentado em [55].

Desenvolver, manter e evoluir aplicações tem se tornado uma atividade complexa. Novas plataformas de desenvolvimento e execução surgem, fazendo com que os sistemas tenham que estar preparados para suportar a evolução tecnológica. Paradigmas têm sido propostos para diminuir a complexidade de desenvolvimento de software, entre eles temos a Engenharia Dirigida a Modelo (ou Model Driven Engineering - MDE). A MDE é uma abordagem que conduz o processo de desenvolvimento de software na construção de modelos e transformação de modelos. A MDE fornece meios para adaptar novas tecnologias com sistemas legados e permite a integração entre diferentes tecnologias [58].

Outro aspecto a ser considerado é o acesso dos usuários a base de dados integrada. Uma tecnologia que tem ganhado espaço na área de Tecnologia da Informação(TI) é a Computação em Nuvem. Na computação em nuvem, os usuários acessam os dados, aplicações ou quaisquer outros serviços com a ajuda de um navegador, independentemente do dispositivo utilizado e localização do usuário [26]. A computação em nuvem fornece tecnologias que permitem o desenvolvimento de soluções que facilitam o acesso a uma visão única dos dados obtidos com a integração de base de dados.

1.2 Problemática

O problema a ser pesquisado neste trabalho é a criação de uma visão unificada de base de dados heterogêneas. Organizações possuem sistemas setoriais que geram um volume de dados que podem auxiliar a tomada de decisão de seus gestores. Mas para que informações possam ser recuperadas das bases de dados dos sistemas é necessário que as estruturadas dos esquemas das bases de dados sejam bem modeladas, facilitando assim, a recuperação de informações em comum contidas em duas ou mais bases de dados. Nem sempre é possível de forma fácil relacionar duas bases de dados heterogêneas, isto porque os sistemas nem sempre são desenvolvidos por uma mesma equipe. Cada equipe pode representar um domínio como um estrutura

de banco de dados diferente. Por exemplo, uma equipe desenvolve dois sistemas, mas não se preocupam em criar estruturas de base de dados que permitam a integração.

Para integrar bases de dados de sistemas de mesmo domínio é necessário encontrar elementos correspondentes entre os esquemas, isto é , *schema matching* [55]. Correspondência de esquemas é um passo crítico na integração de bases de dados heterogêneas [50]. Dentre as dificuldades encontradas para estabelecer correspondência entre esquemas estão conceitos do mundo real que são definidos de forma diferente ou em sua estrutura ou em seu nome. O problema de identificar elementos correspondentes entre dois esquemas é um desafio que vem sendo a tempos pesquisado [8] [50] [38] [40] [55]. O trabalho de encontrar correspondência entres dois esquemas de base de dados é uma tarefa que exige do especialista de domínio um conhecimento profundo sobre os esquemas de base de dados a serem correspondidos, além de ser um trabalho tedioso, demorado e propenso a erro [55].

Assim, mecanismos que auxiliem a tarefa de encontrar correspondências entre esquemas podem tornar o trabalho do especialista do domínio mais fácil. Para automatizar ou semiautomatizar o processo de correspondência de esquema, pesquisas buscam algoritmos que aumentam a qualidade dos elementos correspondidos como a apresentada em [8]. Bem como, a pesquisa por ferramentas que auxiliam a tarefa de corresponder dois esquemas, como em [38].

Novos abordagens também contribuem para auxiliar o processo de identificar correspondências entre esquemas. A abordagem de desenvolvimento de software *Model Driving Engineering*(MDE) pode ajudar a minimizar o trabalho do especialista de domínio na tarefa de correspondência de esquemas(*schema matching*). A *Model Driving Engineering* (MDE) é uma abordagem que conduz o processo de desenvolvimento de software na construção e transformação de modelos. Como um *Schema* é uma estrutura formal que representa um artefato de engenharia, como um *schema SQL*, *XML schema* [8], podemos considerar então que um *schema* é um modelo, e sendo um modelo, a MDE pode contribuir para minimizar a tarefa de correspondência entre esquemas, assegurando produtividade, portabilidade e interoperabilidade [28].

1.3 Solução proposta

A integração de base de dados envolve as tarefas de corresponder os esquemas de base de dados (*schema matching*) e realizar a fusão (*schema merging*) dessas bases de dados. Este trabalho de pesquisa propõe então, um *framework* que suporta o processo de identificação de correspondência de modelos de base de dados de forma semiautomática, além da construção de uma base de dados integrada através do *merge* (fusão) dos modelos de base de dados. O *framework* é construído a partir do trabalho de D. Lopes [34] que utiliza a abordagem MDE e gera regras de transformação de modelos em Atlas Transformation Language (ATL). O trabalho de D. Lopes é estendido para atender a integração de base de dados heterogêneas. Para implementar os conceitos da abordagem MDE, é utilizado o *Eclipse Modeling Framework - EMF* [62] que fornece os recursos de criação de metamodelos, modelos, definição de transformação de modelos e interface gráfica para interação com o especialista de domínio.

Portanto, inicialmente o especialista de domínio deve definir os modelos do esquema de base de dados fonte e alvo. A esses modelos são aplicadas algoritmos de *database model matching*, gerando um modelo de correspondência de modelo de base de dados, que deve ser validado pelo especialista do domínio. Ao modelo de correspondência de modelo de base de dados são aplicadas novas definições de transformação para gerar o *database merging model*. O *database merging model* define os elementos dos modelos de base de dados e do *database matching model* que deverão ser sobreposto, gerando assim, o modelo de base de dados integrado. Finalmente, uma definição de transformação toma como entrada o modelo de base de dados integrada para gerar o *SQL script* de criação da base de dados integrada.

A busca por correspondências entre os esquemas é baseada na comparação de *string* com o uso de diferentes métricas de similaridade de *strings*. Para minimizar o problema de valores nulos para atributos de tabelas fundidas entre dois esquemas de base de dados é proposto a *generalização de correspondência*, que será detalhada na seção 4.6.

O uso da abordagem MDE pode diminuir os esforços para realização do trabalho de integração de esquema de base de dados. Pois, na abordagem proposta, o especialista de domínio se ocupa em definir apenas os modelos de esquema de dados fonte e alvo, sendo os demais modelos gerados pela aplicação dos algoritmos de

database model matching e de *database model merging*. Assim, alterações realizadas nos esquemas de base de dados fonte e/ou alvo podem ser refletidas no esquema integrado aplicando a transformação de modelos até a obtenção do novo modelo de base de dados integrado.

A solução apresentada tem como foco a integração de base de dados, não fazendo parte do escopo do trabalho a integração de outros tipos de esquemas como ontologias e XML schema. Isto porque, a motivação para o trabalho de pesquisa foram os sistemas de gerenciamento do Sistema Único de Saúde (SUS), que na sua maioria utilizam banco de dados relacionais. Assim, é tratado apenas problema de integração de banco de dados relacionais.

1.4 Objetivos

O objetivo geral do trabalho de pesquisa é fornecer uma solução viável para o problema de fusão de bases de dados heterogêneas, buscando criar uma visão unificada das bases de dados de diversos sistemas.

1.4.1 Objetivos específicos

Os objetivos específicos são os seguintes:

- Construção de um *framework* para realizar integração de base de dados, fornecendo algoritmos de *schema matching* e *schema merging* dentro do contexto MDE;
- Construção de um protótipo do *framework* para integração de base de dados;
- Aplicar o *framework* para integração e evolução de base de dados no domínio dos software de gerenciamento do SUS, de modo que os profissionais de saúde tenham uma visão unificada dos diversos sistemas oferecidos pelo SUS, podendo assim, ter uma fonte de dados rica para suas pesquisas e tomada de decisões;
- Implantar a base de dados integrada em uma plataforma de computação em nuvem;
- Desenvolver uma aplicação de serviço web para permitir o acesso a base de dados integrada em uma plataforma de computação em nuvem.

1.5 Metodologia de pesquisa

A metodologia utilizada no trabalho de pesquisa pode ser listada como a seguir:

1. Pesquisa bibliográfica para coletar informações sobre o estado da arte dos tópicos a seguir:
   - Engenharia dirigida por modelos, incluindo linguagens de transformação, definição de transformação, especificação de correspondências, metamodelos e *matching* de metamodelos e modelos; [23] [28] [33];
   - Integração de base de dados ;
   - Abordagem de *schema matching* [55];
   - Estudar e utilizar algoritmos de *Matching* para suportar a criação de modelos de correspondência do esquema de base de dados [55];
   - Computação em Nuvem [60] [66] .
2. Proposta de construção de modelos para representar esquema de base de dados no contexto da MDE;
3. Uso do *Eclipse Modeling Framework Project* (EMF) [19] para a construção de modelos e metamodelos de esquema de base de dados, correspondência de esquema e esquema de base de dados integrado;
4. Uso do *Eclipse Modeling Framework Project* (EMF) para construção do *framework* de integração de base de dados como um *plug-in Eclipse*;
5. Proposta de um metamodelo da linguagem SQL para geração de *script* de base de dados;
6. O ambiente de programação *Eclipse* será utilizado durante o desenvolvimento do trabalho proposto [18].
7. Construção de um ambiente na plataforma de computação em nuvem para hospedar e disponibilizar acesso a base de dados integrada [20].

8. Estudar os sistemas de gerenciamento do Sistema Único de Saúde-SUS para identificar sistemas que deverão ter sua base de dados integrada [44].

9. Adaptação e extensão das ferramentas MT4MDE e SAMT4MDE para suportar a integração de base de dados através de *database model matching* e *database model merging*.

1.6 Apresentação da dissertação

A dissertação que descreve este trabalho de pesquisa está organizada em 7 (sete) capítulos, como segue:

O primeiro Capítulo trata do contexto tecnológico no qual todo o trabalho de pesquisa se insere. O capítulo também expõe a problemática que motivou a pesquisa, buscando o desenvolvimento de uma solução viável para integração de base de dados. Neste primeiro capítulo, o objetivo geral, os objetivos específicos e a metodologia de pesquisa são apresentados.

O segundo Capítulo, a fundamentação teórica que norteia este trabalho é apresentada. Os conceitos necessários para o entendimento da solução proposta neste trabalho são expostos, tais como: *Model-Driven Engineering* - MDE e as propostas de sua implementação *Model Driven Architecture* - MDA e *Eclipse Modeling Framework* - EMF são detalhados; conceitos sobre *Schema Matching* e *Schema Merging* são descritos, mostrando sua importância no processo de integração de base de dados; conceitos de Modelo Relacional e SQL são mostrados para que o leitor entenda como a abordagem MDE é aplicada na integração de base de dados; e finalizando o capítulo, o domínio de aplicação da solução proposta para integração de base de dados é apresentado.

O terceiro Capítulo faz uma breve explanação sobre o estado da arte da pesquisa, bem como, uma apresentação de trabalhos relacionados que tem como foco o *matching* de esquemas, não só de esquema de base de dados, mas também de XML *Schema*, Metamodelos e Ontologias.

O quarto Capítulo apresenta os metamodelos para suportar a integração de base de dados proposto para o *framework*, o algoritmo de *database model matching*,

responsável por criar o modelo de *database matching* e o algoritmo de *database model merging*, responsável por criar o modelo de *database merging*.

No quinto Capítulo, o protótipo para suportar integração de base de dados é apresentado. Uma visão dos componentes da arquitetura do *framework* é mostrada, além das interfaces de criação de modelos de esquema de base de dados, mapeamento dos modelos de esquema de base de dados, geração de *schema merging*, e, por fim , a visualização do *script* SQL do esquema integrado.

No sexto Capítulo, o *framework* proposto é avaliado através de um exemplo ilustrativo de sua aplicação. No exemplo, toda a metodologia para integrar bases de dados é executada para se obter um esquema de base de dados integrada.

Por fim, o sétimo Capítulo traz as conclusões da pesquisa, suas contribuições, as limitações e dificuldades enfrentadas, os resultados obtidos e os trabalhos futuros para aprimoramento da pesquisa iniciada.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta a base teórica necessária para o entendimento das tecnologias utilizadas no desenvolvimento do projeto de pequisa. Conceitos que fundamentam a solução são expostos como: a abordagem de desenvolvimento de software *Model-Driven Engineering (MDE)*; *Model-Driven Architecture (MDA)* [47], projeto MDE padronizado pela *Object Management Group (OMG)* [46]; o *Eclipse Modeling Framework (EMF)* [62] desenvolvido pelo grupo Eclipse; os conceitos e técnicas de correspondência entre esquemas (*schema matching*); os conceitos e técnicas de fusão de esquema (*schema merging*); conceitos sobre Modelo Relacional e Structured Query Language (SQL). Por fim, a aplicação da ferramenta de suporte a integração de modelos de base de dados no domínio do Sistema Único de Saúde (SUS) é apresentada.

2.1 *Model-Driven Engineering* (MDE)

O avanço tecnológico trouxe a necessidade que produtos tivessem em sua composição, de alguma forma, computadores e *software* [59]. O *software* então, torna-se essencial em atividades como controle de vôo, indústria automobilística, sistemas bancários, entre outros. Com isso, surge também o aumento da complexidade e da necessidade de qualidade no desenvolvimento de *software*, pois falhas no desenvolvimento e execução de softwares podem trazer prejuízos financeiros. Assim, desenvolver software se tornou mais complexo, exigindo qualidade e diminuição de tempo de desenvolvimento.

Para atender essa nova realidade uma série de técnicas ou paradigmas de desenvolvimento foram propostos, como: modelo em cascata, desenvolvimento evolucionário, transformação formal e reuso [59]. Buscando atender a qualidade de desenvolvimento de software surge o paradigma *MDE*. A MDE é uma proposta que busca gerenciar a complexidade de desenvolvimento de *software*, trazendo o modelo como foco principal do desenvolvimento [33]. Assim, o objetivo da MDE é elevar o nível de

abstração usando modelos como entidades de primeira classe e, consequentemente, os artefatos primários de desenvolvimento [30].

O modelo é o principal artefato de desenvolvimento no contexto MDE. Um modelo representa situações do mundo real, além de permitir obter visões diferentes de um mesmo sistema [11].

Para que modelos possam ser utilizados no processo de desenvolvimento de softwares, eles precisam ser concebidos em uma linguagem de modelagem bem definida, com sintaxe e semântica que permita expressar seus elementos e relações [36]. No contexto MDE, modelos são criados conforme um metamodelo. Um metamodelo especifica elementos que compõem um modelo, como estes elementos se relacionam e quais são as restrições impostas pelo modelo [11]. A seguir, duas propostas de abordagem de desenvolvimento baseado na MDE serão descritas.

2.1.1 *Model Driven Architecture* (MDA)

A Arquitetura Dirigida por Modelos (*Model-Driven Architeture*-MDA) da *Object Management Group*(OMG) é um exemplo de MDE. Na abordagem MDA, o ciclo de vida de desenvolvimento do software pode seguir o ciclo de vida de desenvolvimento de software tradicional, porém sua diferença está na criação de artefatos que são modelos formais que podem ser entendidos e processados pelo computador [28]. A OMG propõe a divisão da modelagem em quatro níveis de camada, descritas a se-

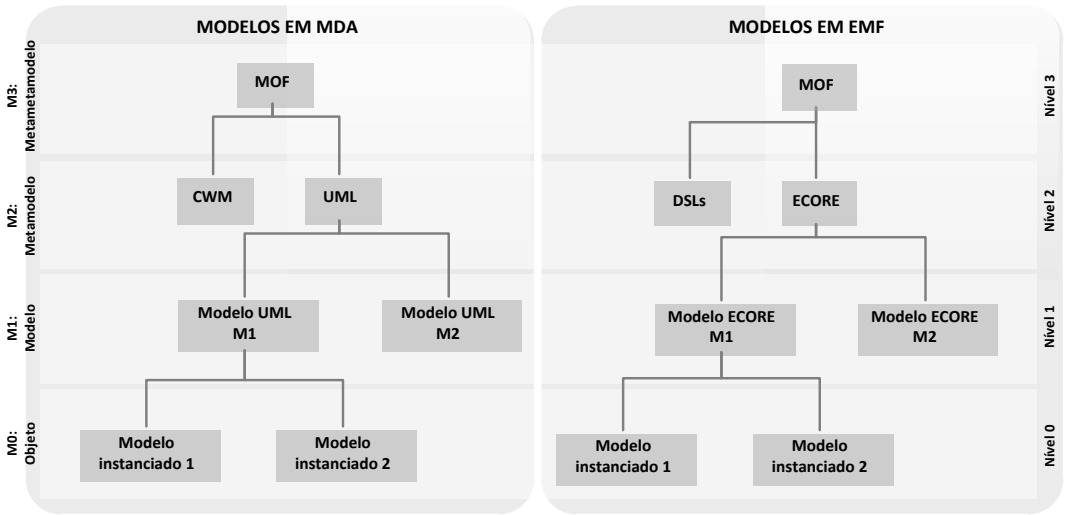


Figura 2.1: Arquitetura de quatro camadas [4]

guir(veja Figura 2.1) [28]:

- **M0 (objeto)**: camada que representa objetos do mundo real, isto é, são instâncias do domínio computacional de um plataforma final. Como exemplo de objeto desta camada podemos citar um aluno de um sistema acadêmico, onde este tem nome="João Batista" e curso="Ciência da Computação";
- **M1 (modelo)**: camada que representa modelos de conceitos do mundo real. A camada M0 é formada por instâncias de que estão conforme a camada M1, isto é, se em M1 especifica que uma classe aluno é composta por nome e curso, as instâncias de M0 terão objetos alunos com essas propriedades;
- **M2 (metamodelo)**: modelos que são criados nesta camada são chamados de metamodelos. Um metamodelo especifica como um determinado modelo deve ser criado, especificando quais elementos e relações poderão ser criadas no modelo. Modelos da camada M1 são instâncias que estão conforme especificação da camada M2. Como exemplo, podemos citar o metamodelo UML [48];
- **M3 (metametamodelo)**: nesta camada, os conceitos necessários para criação de metamodelos são definidos. Um metamodelo deve ser criado conforme o que é especificado em seu metametamodelo.

A Figura 2.1 também apresenta as camadas do EMF que possuem as mesmas funções das quatros camadas proposta pela OMG. No EMF o nível 3 representa os metametamodelos, o nível 2 tem-se a definição dos metamodelos conforme metametamodelos do nível 3, o nível 1 tem-se os modelos instanciados conforme metamodelos definidos no nível 2 e por fim tem-se o nível 0 com objetos instanciados conforme modelos do nível 1.

Para entender o processo de desenvolvimento no contexto MDA, alguns conceitos importantes que formam o *framework* MDA devem ser conhecidos(veja Figura 2.2). Dentre eles destacam-se:

- **Platform Independent Model (PIM)**: ao definir esse modelo o projetista cria um modelo de alto nível de abstração, sem se ater a qualquer aspecto tecnológico de implementação da solução. No PIM, o sistema é modelado em uma visão de melhor solução para o negócio;

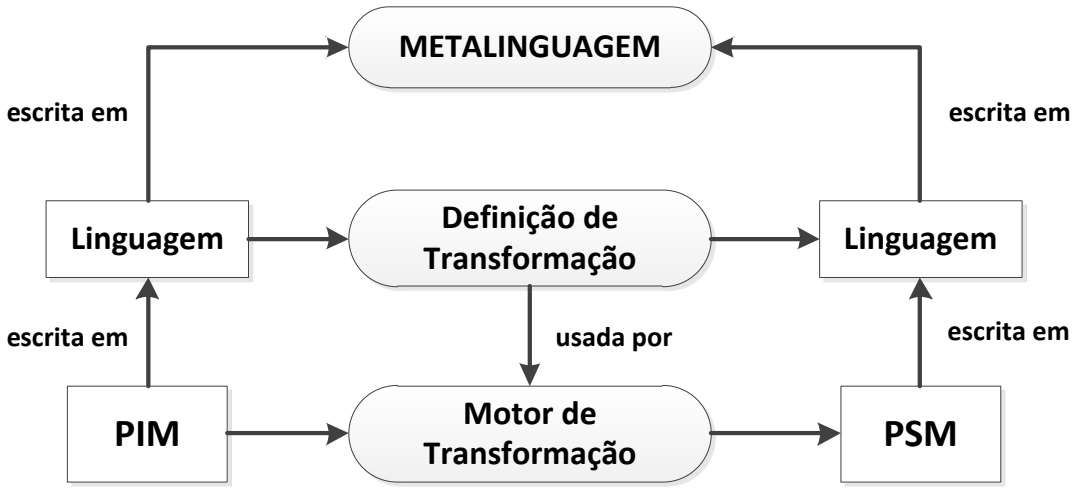


Figura 2.2: *Framework* básico MDA [4]

- **Platform Specific Model (PSM)**: ao definir esse modelo, o projetista descreve o sistema considerando os aspectos tecnológicos de sua implementação. O PSM é obtido a partir da execução de definições de transformação do modelo PIM;
- **Definição de Transformação**: permite definir como elementos de um modelo fonte será transformado em elementos de um modelo alvo. Com a definição de transformação criada, podemos então transformar qualquer modelo fonte para um modelo alvo;
- **Motor de Transformação**: definição de transformação são apenas especificações que declaram como gerar um modelo a partir de outro. Para que a transformação se realize é preciso que as definições de transformação sejam executadas por um motor de transformação, que recebe um modelo de entrada, executa as definições de transformação e gera como saída um outro modelo.

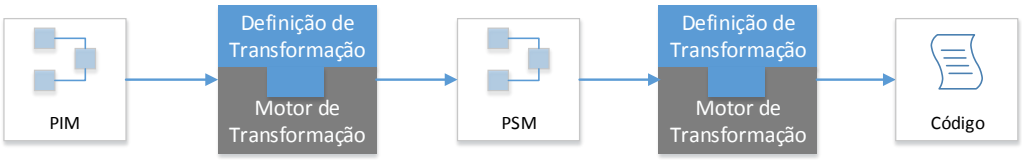


Figura 2.3: Processo de transformação na abordagem MDA [28]

O processo de transformação de modelos é essencial para a abordagem MDA, como mostra a Figura 2.3 um código é obtido após a execução de sucessivas

definições de transformação. Isto permite que alterações nos modelo PIM sejam refletidas no código, aplicando novamente as definições de transformação.

A MDA se propõe a fornecer um melhor gerenciamento dos problemas de produtividade, portabilidade, interoperabilidade presente no desenvolvimento de software [28], como apresentado a seguir:

- **Produtividade**: na abordagem MDA, os desenvolvedores concentram-se no trabalho de criação do modelo independente de plataforma, deixando aspectos da plataforma específica para a definição de transformação. Além disso, há menos código a ser escrito a nível de PSM e código, já que grande parte desse código é gerado pela transformação do PIM para PSM e do PSM para código;
- **Portabilidade**: um mesmo PIM pode ser transformado para plataformas diferentes;
- **Interoperabilidade**: a interoperabilidade é atingida com a criação de pontes entre PSM gerados a partir de um mesmo PIM. As pontes permitem que conceitos de uma plataforma sejam transformados em conceitos de outra plataforma.

A idéia básica da MDA é gerenciar o ciclo de desenvolvimento de *software* centrado na modelagem. Assim, a manutenção e evolução do software fica em alto nível de abstração, na criação e manutenção de modelos.

2.1.2 *Eclipse Modeling Framework* - EMF

O EMF foi utilizado para implementação do *framework* proposto neste trabalho de pesquisa. O EMF foi aplicado na construção dos metamodelos, na criação e visualização de modelos e na transformação de modelos. Como mostra a Figura 2.4, o EMF unifica o uso das tecnologias Java [27], *eXtensible Markup Language (XML)* [67] e *Unified Modeling Language (UML)* [65].

O EMF é um *framework* que facilita a modelagem e geração de código na construção de ferramentas ou aplicações tendo como base uma estrutura de modelo de dados [19]. Modelos conceituais podem ser definidos em UML, Java ou XML *Schema* [62]. O EMF é um esforço do *Eclipse Tools Project* para criação de aplicações

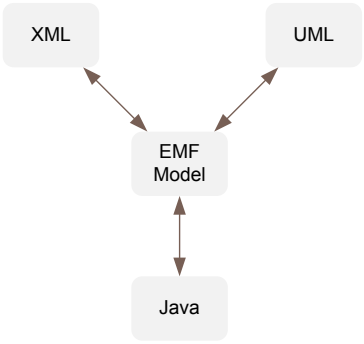


Figura 2.4: Tecnologias utilizadas pelo EMF [62]

dirigidas por modelos [12]. A seguir exemplificamos os possíveis formas de construção de modelos com o EMF.

Usando metamodelo *Ecore*

O EMF utiliza o metamodelo *Ecore* para construção de modelos (veja Figura 2.1). O metamodelo *Ecore* é um subconjunto simplificado do metamodelo UML [62]. A Figura 2.5 mostra um exemplo de um modelo definido com *Ecore*.

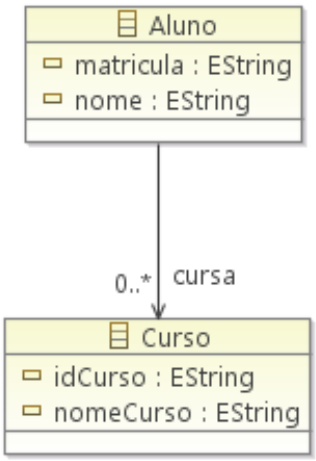


Figura 2.5: Fragmento de modelo Acadêmico definido em *Ecore*

Usando *Annotations*

O modelo conceitual também pode ser construindo utilizando @annotations Java. Na listagem de código descrito abaixo, a anotação @model define que a interface *Aluno* é uma classe do modelo Acadêmico, além , de definir os atributos que a com-

põe e seu relacionamento com a classe *Curso*, representada no código pela operação *getCursa*().

```
1 /**
2  * @model
3  * @generated
4  */
5 public interface Aluno extends EObject {
6    /**
7     * @model
8     * @generated
9     */
10   String getMatricula();
11   /**
12     *@model
13    * @generated
14    */
15   String getNome();
16   /**
17    * @model type="mmacedemico.Curso"
18    * @generated
19    */
20   EList getCursa();
21 } // Aluno
```

Usando XML *Schema*

O EMF fornece uma abordagem baseada em XML *Schema* para a definição de objetos de um modelo *Ecore*. O mapeamento XML para *Ecore* é composto por [62]:

- Um esquema é mapeado para um *EPackage*;
- Definição de tipos complexos são mapeados para *EClass*;
- Tipos simples são mapeados para um *EDataType*;
- Uma declaração de atributo ou uma declaração de elemento pode ser mapeada para *EAttribute*, se for do tipo *EDataType*, ou para *EReference* se for do tipo *EClass*;

A listagem abaixo demonstra a definição da classe *Aluno* do modelo da Figura 2.5.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <ecore:EPackage xmi:version="2.0"
3     xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.org/2001/↩
        XMLSchema-instance"
4     xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="mmacedemico"
5     nsURI="http://mmacedemico/1.0" nsPrefix="mmacedemico">
6   <eClassifiers xsi:type="ecore:EClass" name="Aluno">
7     <eStructuralFeatures xsi:type="ecore:EAttribute" name="matricula" ↩
        eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//↩
        EString"/>
8     <eStructuralFeatures xsi:type="ecore:EAttribute" name="nome" eType="↩
        ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
9     <eStructuralFeatures xsi:type="ecore:EReference" name="cursa" ↩
        upperBound="-1"
10        eType="#//Curso"/>
11  </ecore:EPackage>
```

Todos os metamodelos propostos neste trabalho foram construídos conforme metametamodelo *Ecore*. Para desenvolver o *framework* foi utilizado EMF *Generator Model* que permite a partir de um metamodelo *Ecore* gerar as interface e classes para manipulação de modelos *Ecore*.

2.2 Conceitos de *Schema Matching*

Sistemas informatizados em uma organização são fontes importantes de dados que podem ser utilizadas na tomada de decisão. Para permitir o aproveitamento desses dados produzidos é necessário que as bases de dados dos sistemas sejam integradas. Integrar base de dados envolve a busca de similaridade entre seus esquemas [55].

Schema matching é a tarefa de encontrar correspondências entre elementos de dois esquemas. *Correspondência* é a relação de um ou mais elementos de um esquema com um ou mais elementos de outro esquema [8]. Ferramentas como [17] [33] [38] [50]

auxiliam o trabalho de especialista de domínio em encontrar correspondência entre esquemas.

Em geral, há muita dificuldade em determinar o *matching*(correspondência) de forma automática [55]. Já que, muitos esquemas tem elementos com mesma semântica, porém suas estruturas não estão bem formatadas dificultando a criação de definição de correspondência dos esquemas. Assim, o que normalmente é retornado de um processo de *matching* são elementos candidatos a correspondência. De acordo com [55], *schema matching* pode ser caracterizado em *individual matching* e *combining matchers*. A Figura 2.6 mostra uma visão geral das abordagens para *schema matching*, a qual será descrita a seguir.

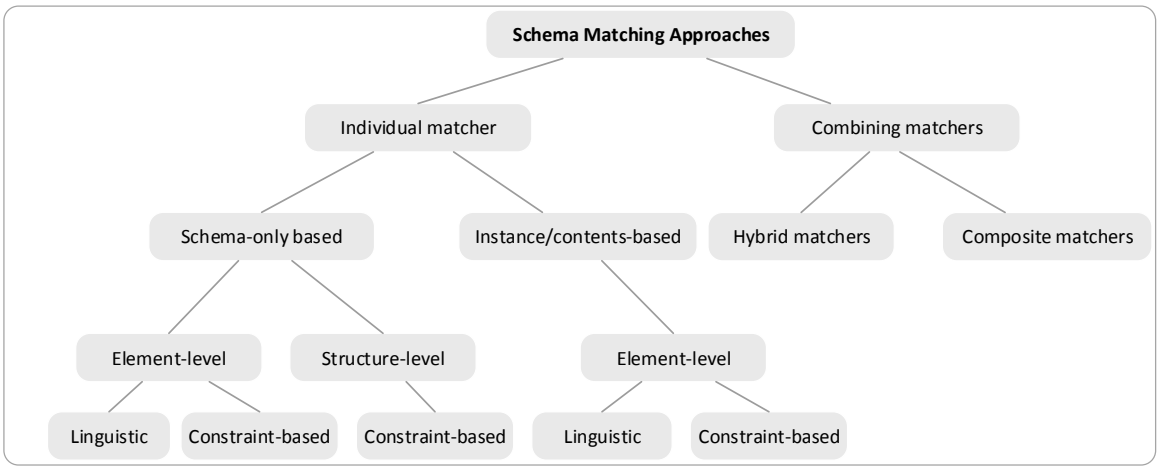


Figura 2.6: Visão geral de abordagens de *matching* (adaptado de [55])

2.2.1 Abordagem *individual matching*

A abordagem de *individual matching* tem como base a construção de mapeamento utilizando um único critério simples de correspondência. A abordagem de *individual matching* pode ser classificada em *Schema-only based*(baseada apenas em esquema) ou em *Instance/contents-based* (baseada em instância/conteúdo).

Schema-only based (baseado apenas nos esquemas)

O *Schema-only based* considera apenas informações a nível de esquema (tipos de dados, nome de elemento, descrição), não considerando instância de dados,

podendo ser classificado em *element-level* (nível de elemento) e *structure-level* (nível de estrutura).

Element-level matching (correspondência a nível de elemento) corresponde a um elemento do esquema fonte diretamente com outro elemento do esquema alvo, como por exemplo, encontrar correspondência entre atributos de uma XML *schema* ou colunas de um esquema relacional.

Structure-level matching (correspondência a nível de estrutura) considera uma combinação de correspondência de elementos que estão juntos em uma estrutura. Assim, para ser considerado correspondente todos elementos de um esquema fonte tem que ser correspondidos com todos elementos de um esquema alvo. Pode acontecer caso em que essa correspondência é parcial, isto é, nem todos elementos do dois esquemas tem correspondência. Em casos mais complexos, a correspondência considera a estrutura dos dois esquemas, onde um conceito representado por apenas um elemento do esquema fonte pode aparecer em um esquema alvo representado por mais de um elemento (veja Figura 2.7).

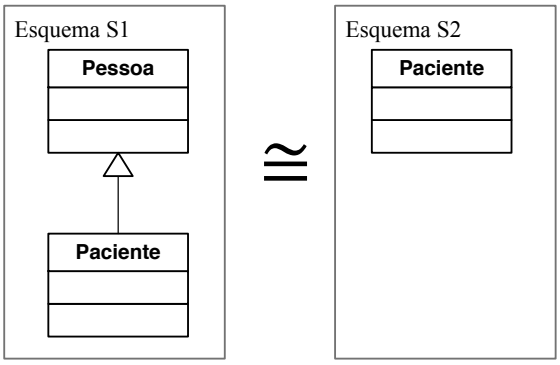


Figura 2.7: Um exemplo de caso complexo de *structure-level matching*

Outros aspectos que podem ser considerados no *matching* a nível de esquema são o *match cardinality* e a *linguistic approaches* [55]. O *match cardinality* determina, dentro de um mapeamento individual de elemento, se um ou mais elementos de um esquema fonte podem corresponder a um ou mais elementos de um esquema alvo. Essa cardinalidade pode ser a nível de elemento de 1:1, n:1 ou 1:n e a nível de estrutura n:1. A *linguistic approaches* busca correspondência entre elementos com uso de similaridade de textos e nomes, podendo ser distinguida em *name matching* e *description matching*. O *name matching* usa correspondência de nome de elementos do esquema determinando se são iguais ou similares. *Description matching* utiliza comentários, que

descrevem semanticamente os elementos de um esquema, para determinar se existe similaridade entre esquemas.

Além do *match cardinality* e *linguistic approaches* temos o critério de *match* baseado em *constraint*. Encontramos frequentemente definições de restrições em esquemas como tipos de dados permitido, faixa de valores permitidos, exclusividade, tipos de relacionamentos, entre outros. Assim, se dois esquemas contem restrições similares, é possível dizer que os dois esquemas possuem correspondências.

***Instance/contents-based* (baseado em instâncias/conteúdo)**

O uso da abordagem a nível de instância pode melhorar o trabalho de *matching*, especialmente em esquema com informações limitadas [55], como por exemplo esquema de dados semi-estruturado. A abordagem a nível de instância considera o conteúdo de elementos do esquema para determinar o *matching*.

As técnicas utilizadas para realização de *matching* na abordagem *schema-only based* podem ser aplicadas também para abordagem *instance/contents-based*. Uma técnica que pode ser utilizada nesta abordagem é a recuperação de informação através de palavras chaves que comumente surgem em um conteúdo. Em dados mais estruturados como elemento *string* e numéricos, pode ser aplicado caracterização baseada em restrições, onde verifica-se faixa de valores e padrões de caracteres para definir o *matching*.

2.2.2 Abordagem *combining matchers* (combinando correspondências)

A utilização de uma única abordagem de *matching* pode reduzir o número de elementos candidatos a serem correspondidos. Por isso, uma combinação das abordagens descritas anteriormente pode melhorar o resultado do *matching*. Isto pode ser feito de duas formas: *hybrid matcher* e *composite matchers*, que serão descrito a seguir.

***Hybrid matcher* (correspondências híbridas)**

O *hybrid matcher* combina o uso de abordagens de *matching* para determinar os elementos candidatos a correspondência. Por exemplo, o uso de uma abordagem a nível de elemento em conjunto com a abordagem a nível de estrutura pode fornecer melhores candidatos a *match*. Essa combinação de abordagem pode ser executada paralelamente ou em uma ordem fixa.

***Composite matcher* (correspondência por composição)**

O *composite matcher* combina o resultado de diferentes execuções de *matching*. Assim, o resultado de um *instance-level matchers* pode ser combinado com resultado de *schema-level matchers*. A execução da ordem de *matching* é flexível. O resultado de uma *matching* é logo aproveitado para a execução de um segunda tarefa de *matching*.

2.3 Conceitos de *Schema Merging*

Integrar dois esquemas requer primeiramente encontrar correspondências entre os dois esquemas (*schema matching* visto na seção anterior), para depois então, realizar o *merge* entre os dois modelos baseados em uma correspondência [53]. Assim, o *merge* de esquemas é uma etapa importante na integração de esquema.

O objetivo de *schema merging* é ter como produto final um esquema mediado com os elementos correspondentes entre dois esquemas, e também, os elementos que não foram correspondidos entre os dois esquemas [51]. Acontece que a tarefa de realizar o *merge* entre dois modelos não é uma tarefa tão simples. Em [51], dois aspectos que podem afetar o *merge* de esquemas são expostos:

1. Informações sobrepostas devem ter representação única no esquema mediado. Informação sobreposta significa que elementos correspondentes foram encontrados entre os esquemas é devem ter uma representação atômica no *merge*;

2. Informação específica dos esquemas podem ser repassadas para o esquema mediado ou por extensão de relações que representam a informação sobreposta ou

por adição de relação, dependendo se a informação específica do esquema fonte é ou não é estreitamente dependente dos elementos sobrepostos.

Portanto, a saída de uma operação de *merge* é um esquema que contém todos elementos não duplicados dos esquemas A e B, além da sobreposição dos elementos que são declarados redundantes no mapeamentos de correspondência Map_{AB} [52]. Para a execução da operação de *merge* são necessárias as seguintes entradas [53]:

1. Dois esquemas: A e B;
2. Um mapeamento, Map_{AB}, que define as correspondências entre A e B;
3. Uma designação opcional de qual modelo é o principal;
4. Substituições opcionais para o comportamento padrão do *merge*.

Em [52], requisitos também são propostos para um *merge* genérico, onde o esquema mediado tem que satisfazer os seguintes requisitos:

1. **Preservação de elementos**: cada elemento de A e B deve ter um correspondente no esquema medido;
2. **Preservação de igualdade**: elementos de entrada são mapeados para um único elemento no esquema mediado quando estão presentes no mapeamento Map_{AB};
3. **Preservação de relacionamento**: cada relacionamento de entrada está explicitamente ou implicitamente no esquema mediado;
4. **Preservação de similaridade**: Elementos que são declarados como sendo similares (mas não igual) ao outro em Map_{AB} mantém a sua identidade separada no esquema mediado e estão relacionados entre si por algum relacionamento.

A tarefa de criar uma visão única de dois esquemas, isto é, *schema merging*, requer a resolução de conflitos que são criados no momento em que elementos dos esquemas são sobrepostos. A Figura 2.8 mostra que Paciente.telefone $\cong$ (Pac.telefoneResidencial, Pac.telefoneTrabalho), exemplificando assim, um conflito que deve ser resolvido para que *Paciente* e *Pac* sofram *merge*. Os conflitos que surgem no processo de *merge* podem ser classificados como [53]:

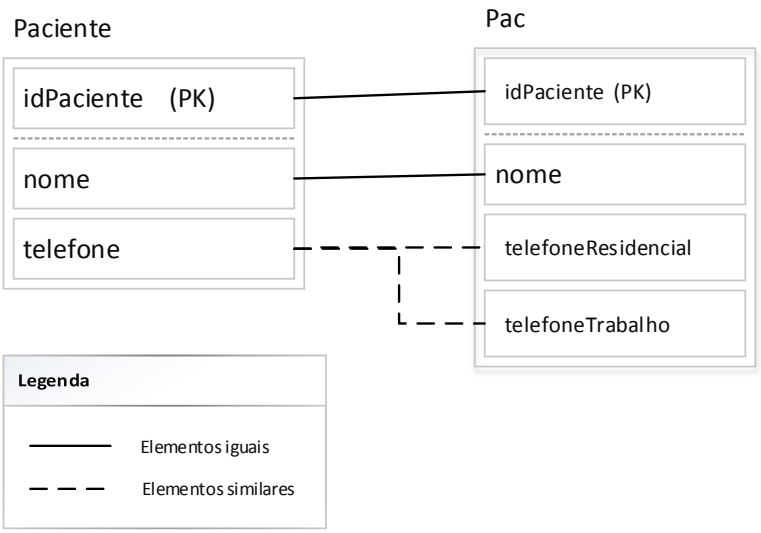


Figura 2.8: Um exemplo de correspondência de elementos iguais e similares

- **Conflito de representação**: este conflito ocorre quando a representação de um conceito do mundo real é feita de diferentes formas. Veja na Figura 2.8 como o conceito de telefone foi representado de forma diferente. No esquema *Paciente* telefone é representado por um único atributo (**Paciente.telefone**), enquanto que no esquema *Pac* telefone é representado por dois atributos (**Pac.telefoneResidencial** e **Pac.telefoneTrabalho**). Estes conflitos podem ser solucionados, quando possível, pela sobreposição de dos elementos conflitantes em um único elemento no esquema mediado. Ou por um mapeamento no esquema mediado, onde elementos conflitantes podem ser elementos filhos de um elemento no esquema mediado. Ou ainda, de uma forma mais complexa, onde os elementos similares poderiam ser representados por um expressão que os concatenassem.

- **Conflitos a nível de metamodelo**: estes conflitos acontecem quando há a violação de restrições impostas pelo metamodelo(por exemplo, SQL DDL). Por exemplo, em um esquema XML existe o conceito de sub-coluna, que não é implementado em um esquema relacional, assim, pode haver um conflito de restrição do esquema relacional por não apresentar este conceito. A resolução desde conflito pode ser conseguida com o uso de um operador que faça as adequações necessárias para que os elementos em conflitos atendam as restrições impostas pelo metamodelo.

- **Conflitos fundamentais**: ocorre quando o resultado no *merge* não é um modelo devido violação do metamodelo. Um exemplo deste conflito é quando dois elementos correspondidos possuem tipos diferentes. Por exemplo, em um endereço no esquema S_1 o atributo *numero* do endereço é do tipo *inteiro*, enquanto que no esquema S_2 o atributo *numero* do endereço é do tipo *string*, neste caso temos um conflito que deve ser resolvido.

Outros conflitos que devem ser considerados no processo de *merge* estão relacionados com os relacionamentos entre elementos, como por exemplo, conflitos de cardinalidade de relacionamentos entre elementos e conflitos de relacionamentos de especialização de elementos.

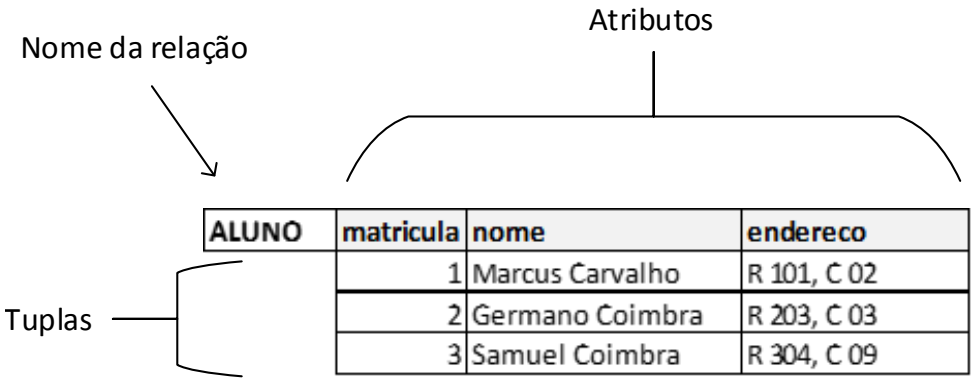
2.4 Modelo relacional

O trabalho de pesquisa proposto envolve a integração de base de dados relacional, assim será apresentado um visão dos principias conceitos sobre modelo de dados relacional, bem como sobre a Linguagem de Consulta Estruturada (***S**tructured **Q**uery **L**anguag*e - SQL).

O modelo de dados relacional foi proposto pelo matemático Ted Codd da IBM *Research* em 1970. O modelo tem fundamentação em conceitos de uma relação matemática (tabela de valores), tendo como base teórica a teoria de conjuntos e a lógica de predicados de primeira ordem [56].

O modelo relacional faz a representação de uma base de dados por uma coleção de *relações*. Assim, temos uma tabela de valores, onde temos as linhas representado entidades ou conceitos do mundo real. E as colunas representam valores que caracterizam as entidades presentes em cada linha. A seguir, conceitos importantes serão descritos para o entendimento do modelo relacional como *tupla*, *atributo*, *relação* entre outros.

- *Relação*: são conceitos/entidades que encontramos no mundo real e precisamos armazenar informações sobre eles. Uma relação é composta por um nome para individualizá-la, por um conjunto de tuplas composta por seus atributos. No contexto de banco de dados, as relações podem ser denominadas *tabelas*;



ALUNO	matricula	nome	endereco
	1	Marcus Carvalho	R 101, C 02
	2	Germano Coimbra	R 203, C 03
	3	Samuel Coimbra	R 304, C 09

Figura 2.9: Um exemplo de relação com tuplas e atributos [56]

- *Tuplas*: cada tupla em uma relação representa uma entidade do mundo real, como por exemplo *empregado*, *curso*, *aluno*, *vendas* entre outros. As tuplas são composta de atributos. No contexto de banco de dados, as tuplas são denominadas *registros*.
- *Atributos*: caracterizam uma entidade do mundo real. Os atributos representam propriedades que diferenciam entidades de tuplas. No contexto de banco de dados, os atributos são denominados colunas. Os atributos armazenam valores ou conjunto de valores, assim, no exemplo mostrado na Figura 2.9, o atributo *matricula* pode armazenar números inteiros;
- *Chave primária (PK - Primary Key)*: chave formada por um ou conjunto de atributos que identificam unicamente uma tupla de uma relação [63]. Na Figura 2.9, o atributo *matricula* identifica unicamente um aluno;
- *Chave estrangeira (FK - Foreign Key)*: relaciona um tupla de uma relação R1 com uma tupla de uma relação R2, isto é, a chave estrangeira permite criar relacionamento entre relações;

2.4.1 Linguagem de consulta estruturada - SQL

A linguagem SQL tem uma grande contribuição para a disseminação do uso de banco de dados relacionais [56]. Isto se deve ao SQL ter se tornado um padrão na manipulação de banco de dados, facilitando assim, a migração de aplicações.

Uma breve descrição dos principais conceitos que envolve o uso da linguagem SQL é mostrado a seguir.

A linguagem SQL está divida em instruções para *Data Definition Language (DDL)* e instruções para *Data Manipulation Language (DML)*. A SQL utiliza os termos tabela, registro e coluna para representar relação, tuplas e atributos respectivamente. A DDL é composta pelas seguintes instruções:

- CREATE: instrução que permite criar esquema de base de dados e tabelas;
- ALTER: instrução que permite alterar estrutura de tabelas;
- DROP: instrução que permite excluir estruturas de tabelas. Esta instrução deve ser executada com atenção, pois ao excluir um tabela conseqüentemente seus dados também são excluídos;

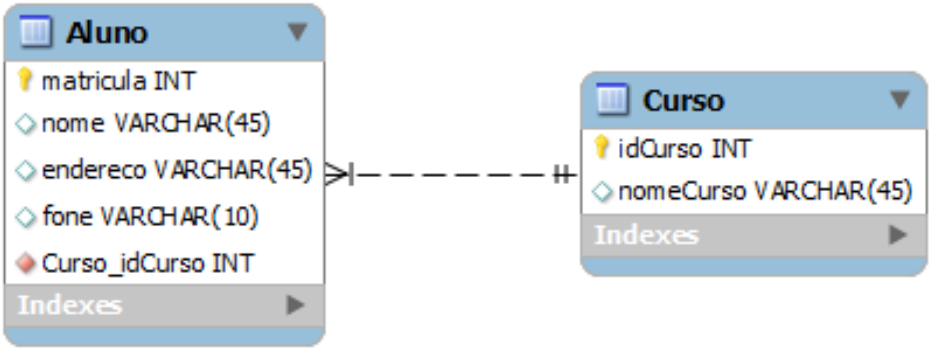


Figura 2.10: Diagrama ER do modelo acadêmico.

A Listagem 2.1 mostra exemplo do uso das instruções DDL para criação de um esquema nomeado de *Acadêmico*(veja Figura 2.10), que tem em sua composição uma tabela *Aluno*.

Listagem 2.1: Instruções DDL para criação de tabelas e colunas.

```
1 //------ Criando um esquema
2 CREATE SCHEMA 'Academico' ;
3 //------ Criando tabela
4 CREATE  TABLE IF NOT EXISTS 'academico'.'Curso' (
5   'idCurso' INT NOT NULL ,
6   'nomeCurso' VARCHAR(45) NULL ,
```

```
7   PRIMARY KEY ('idCurso') );
8 CREATE  TABLE 'Academico'.'Aluno' (
9   'matricula' INT NOT NULL ,
10  'nome' VARCHAR(45) NULL ,
11  'endereco' VARCHAR(45) NULL ,
12  'fone' VARCHAR(45) NULL ,
13  'Curso_idCurso' INT NOT NULL ,
14  PRIMARY KEY ('matricula'),
15  FOREIGN KEY ('Curso_idCurso' )  );
16 //------ Alterando uma tabela
17 ALTER TABLE 'academico'.'Aluno' ADD COLUMN 'Curso_idCurso' INT(11) NOT ↩
     NULL  AFTER 'fone';
18 //------ Excluindo tabela
19 DROP TABLE 'Academico'.'Aluno';
```

A DML fornece um conjunto de instruções que possibilita especialistas de banco de dados e aplicações realizar consultas e atualização de dados armazenado em um banco de dados. A seguir uma breve descrição de suas instruções é mostrada:

- INSERT: permite a inclusão de registros em uma tabela;
- UPDATE: permite a atualização de dados de um ou grupo de registros;
- DELETE: permite excluir um ou mais registros de uma tabela;
- SELECT: permite recuperar um conjunto de registros de uma ou mais tabelas.

A Listagem 2.2 mostra exemplo do uso das instruções DML para manipulação de tabelas do esquema *Acadêmico* (veja Figura 2.10), que tem em sua composição as tabelas *Aluno* e *Curso*.

Listagem 2.2: Instruções DML para manipulação de registros.

```
1 //----- Inserindo registros
2 INSERT INTO 'academico'.'aluno' ('matricula', 'nome', 'endereco', 'fone', ↩
    'Curso_idCurso') VALUES (1, 'Marcus Carvalho', 'R 102, C 03', '↩
    23456789', 1);
3 //----- Alterando o numero do telefone do aluno
4 UPDATE 'academico'.'aluno' SET 'fone'='12345678' WHERE 'matricula'='1';
5 //----- Excluindo aluno com a matricula '1'
```

```
6 DELETE FROM 'academico'.'aluno' WHERE 'matricula'='1';
7 //----- Recuperando alunos com nome de seus curso. SeleÃ§Ã£o em duas ↩
    tabelas.
8 SELECT matricula, nome, curso_idCurso, nomeCurso FROM 'academico'.'aluno',↩
    'academico'.'curso';
```

2.5 Síntese

Neste capítulo, apresentou-se uma visão geral dos principais conceitos e tecnologias envolvidas no desenvolvimento do trabalho de pesquisa. O objetivo foi situar o leitor sobre alguns conceitos importantes para entender a nossa proposta de *framework* para suportar a integração de base de dados.

Iniciou-se o capítulo fornecendo conceitos sobre MDE, contextualizando suas aplicações. A arquitetura MDA com suas características e seus benefícios foi descrita. Além de apresentar o EMF que é a abordagem MDE desenvolvida pelo projeto Eclipse. O EMF é a tecnologia base na construção do *framework* proposto.

Como o trabalho de pesquisa envolve integração de esquema de base de dados, fez-se necessário apresentar também conceitos de modelo relacional e da linguagem de manipulação e definição de base de dados SQL.

Os conceitos de *Schema Matching* e *Schema Merging* também foram tratados neste capítulo. Os dois conceitos são necessários para o processo de integração de base de dados, assim suas principais características foram expostas.

3 ESTADO DA ARTE

Este capítulo visa descrever as abordagens existentes na literatura para mapeamento e transformação de modelos no contexto MDE, incluindo pesquisas e ferramentas para estabelecer correspondências entre *schemas*. A correspondência entre modelos e esquemas de base de dados é o foco da pesquisa deste trabalho, assim, foram selecionados trabalhos que tratam ou de correspondência de modelos ou que correspondam esquemas de base de dados.

3.1 Especificação de correspondência e definição de transformação no contexto MDE

A *Model Driven Engineering* (MDE) propõe o uso de modelos como uma forma de aumentar a abstração de problemas complexos, diminuindo assim, o esforço no processo de desenvolvimento de *software* [1]. Neste contexto, o gerenciamento do desenvolvimento e da manutenção de sistemas de *software* depende de modelos que expressam diferentes visões de um sistema.

No contexto MDE, o desenvolvimento de *software* segue um processo baseado na construção de modelos e transformação destes modelos [33]. O processo de transformação de modelos envolve a tarefa de especificação de correspondência (mapping) e definição de transformação. D. Lopes em [31] propõe uma separação explícita entre especificação de correspondência (*mapping specification*) e definição de transformação. Especificação de correspondência define a lógica usada para estabelecer relações entre dois metamodelos. Enquanto que definição de transformação especifica regras para transformar elementos de um metamodelo fonte em elementos de um metamodelo alvo. Em [31], uma arquitetura para transformação de modelos é apresentada, como ilustra a Figura 3.1, sendo esta arquitetura constituída dos seguintes elementos:

- MMM (metametamodelo): Metametamodelos para construção de metamodelos, como por exemplo, *Meta Object Facility* (MOF) [49] ou Ecore [62];

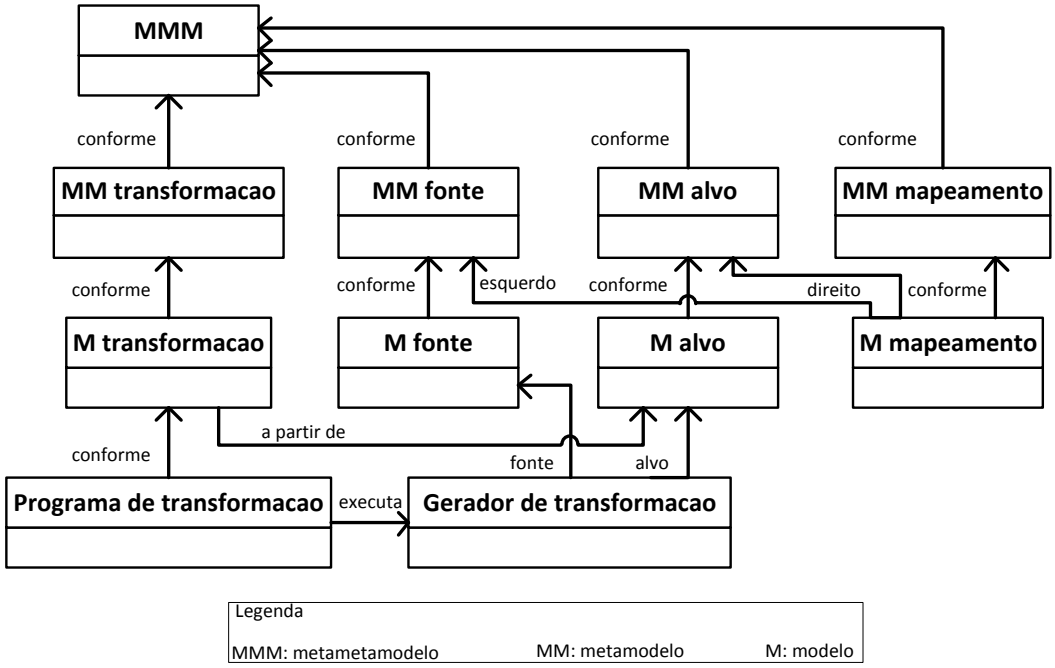


Figura 3.1: Uma proposta de arquitetura para transformação de modelos [32]

- MM fonte e MM Alvo (metamodelo fonte e metamodelo alvo): por exemplo, o metamodelo UML [65];
- M fonte e M Alvo (modelo fonte e modelo alvo): por exemplo, um modelo de sistema acadêmico definido na linguagem de modelagem UML que posteriormente deve ser transformado em modelo Java ou em um modelo C++;
- MM mapeamento (metamodelo de especificação de correspondência): utiliza linguagem para modelagem de correspondências entre os elementos de um metamodelo fonte e os elementos de um metamodelo alvo;
- M mapeamento (modelo de especificação de correspondência): instância (modelo) do metamodelo de mapeamento onde correspondências entre dois metamodelos estão contidas;
- MM transformação (metamodelo de transformação): formalismo que permite a criação precisa de transformações de um modelo fonte em um modelo alvo;
- M transformação (modelo de transformação): instância do metamodelo de transformação que descreve como elementos de um metamodelo fonte e transformado em elementos de um metamodelo alvo;

- Programa de transformação: um programa executável para realizar a transformação de um modelo em outro;

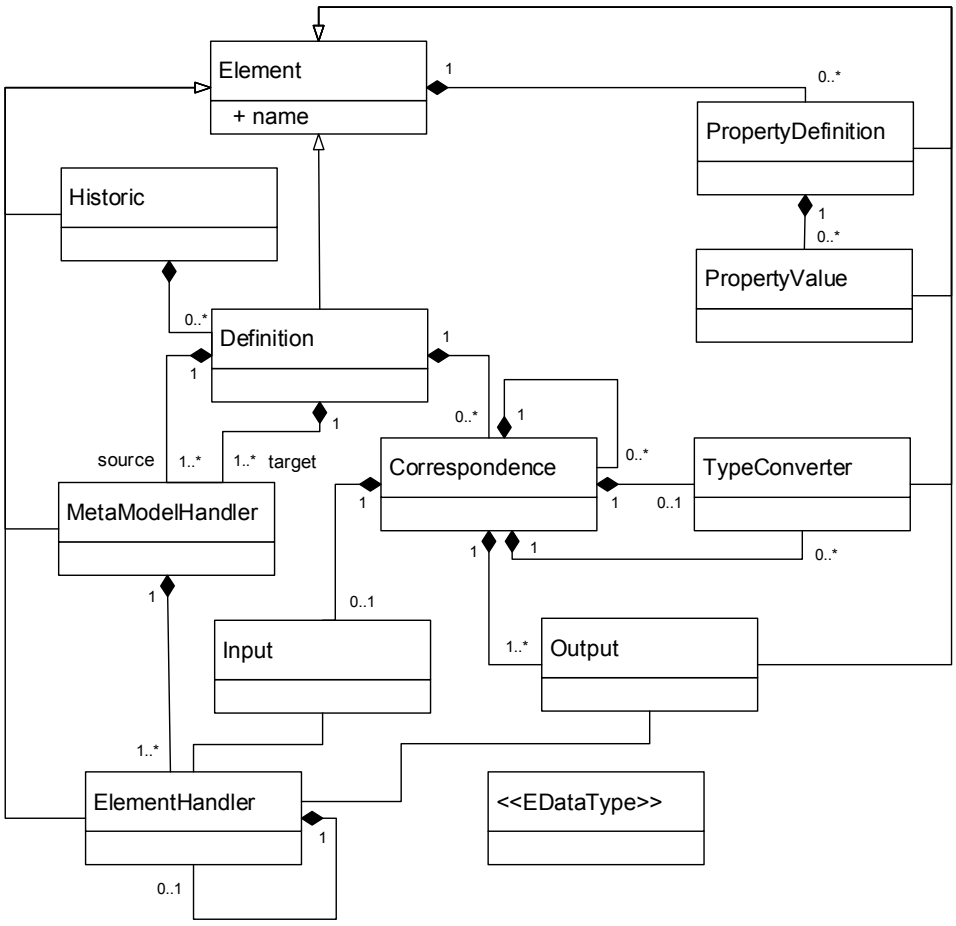


Figura 3.2: Metamodelo de especificação de correspondência proposto em [31](Fragmento)

Também é proposto por D. Lopes [31] um metamodelo de correspondência para suportar a especificação de correspondência (*mapping specification*) definida na arquitetura para transformação de modelos. Além disso, o metamodelo de correspondência contribui para:

- Manter um histórico de especificação de correspondência entre metamodelos;
- Identificar elementos equivalentes entre metamodelos (fonte e alvo);
- Navegar entre elementos do metamodelo fonte e alvo;
- Torna a especificação de correspondência a mais independente possível de linguagens de transformação.

A Figura 3.2 mostra o metamodelo de correspondência proposto em [31]. A seguir é descrito os principais elementos do metamodelo:

- *Element*: é uma generalização para outros elementos;
- *Historic*: mantém um histórico de especificação de correspondências;
- *Definition*: contém referencia para dois ou mais metamodelos(podendo ser um ou mais metamodelos *source* e um ou mais metamodelos *target*) e as correspondências entre estes metamodelos;
- *Correspondence*: define o interrelacionamento entre elementos de dois ou mais metamodelos. Pode ser relacionado um ou mais elemento *Input* com um ou mais elemento *Output*;
- *MetaModelHandler*: Permite a navegação em um metamodelo *source* e *target*;
- *ElementHandler*: permite a navegação nos elementos que estão sendo mapeados sem alterá-los.
- *Input*: identifica o elemento de entrada (metamodelo fonte) da especificação de correspondência;
- *Output*: identifica o elemento de saída (metamodelo alvo) da especificação de correspondência;

Para validar o metamodelo de especificação de correspondência D. Lopes [31] implementa a ferramenta *Mapping Modeling Tool* (MMT). A ferramenta é aplicada para gerar definição de transformação em ATL de um sistema de software definido na linguagem UML para WSDL. A ferramenta foi desenvolvida utilizando o projeto *Eclipse Modeling Framework* (EMF).

3.2 Trabalhos Relacionados

Nesta seção, trabalhos relacionados ao tema *schema matching* e *schema merging* são apresentados. Uma avaliação sobre algumas abordagens de *schema matching* e *schema merging* propostas na literatura são apresentadas.

3.2.1 Generic Schema Matching with Cupid

Madhavan et al. apresenta em [38] um protótipo para *schema matching* genérico denominado de CUPID. O CUPID adota uma abordagem híbrida de *match*, realizando correspondência de nome e correspondência estrutural. O protótipo faz correspondência de XML *schema* e esquema relacional. A solução de matching genérico tem as características a seguir [38]:

- Incluir *matching* automático baseado em lingüística, *schema matching* baseado em elemento e baseado em estrutura;
- É inclinado para similaridade atômica de elementos (ou seja, folhas), onde uma maior semântica do esquema é capturada;
- Explora a estrutura interna, mas procura minimizar erros induzidos pela variação dessa estrutura;
- Explora chaves, restrições referencias e visões;
- Cria *matchers* dependente de contexto de definições de tipos compartilhados usadas em grandes esquemas;
- O *matcher* de nomes utiliza fontes auxiliares como sinônimos e abreviações;
- Gera mapeamentos 1:1 ou 1:n.

O processo de *schema matching* do CUPID começa pela correspondência de nomes. O *matching* linguístico envolve três passos: normalização, onde os nomes são customizados em *tokens*; categorização, onde um grupo de elementos pode ser identificado por um conjunto de palavras chave; comparação, onde é calculada a similaridade linguística (*lsim*) para cada par de elementos a partir de categorias compatíveis.

O *matching* estrutural determina a similaridade estrutural de nós a partir de nomes e tipos de dados similares das folhas dos esquemas. Os esquemas são organizados em uma estrutura em árvore. Para cada par de elementos é calculado a similaridade de estrutura (*ssim*). Se dois elementos são muitos similares, ou seja, sua similaridade excede um alto limite, o *ssim* de cada par de folhas nas duas subárvores é aumentado. Caso contrário é diminuída.

O mapeamento de elementos dos esquemas é gerado utilizando o cálculo de similaridade linguística e estrutural. Os elementos com mais alto valor de *weighted similarity* (**wsim**) farão parte do mapeamentos. O *wsim* e uma média de *lsim* e *ssim*, onde wsim = wstruct * ssim + (1-wstruct) * lsim. A constante *wstruct* é um valor entre 0 e 1. Os mapeamentos criados pelo CUPID são mostrado pelo BizTalk Mapper [39] que são compilados em XSL *translation scripts*.

3.2.2 Tuned Schema Merging (TuSMe)

O trabalho apresentado por Jabeen et al. [25] propõe um algoritmo para o processo de *merging* no contexto Database Management System (DBMS). A proposta estabelece um equilíbrio entre os trabalhos apresentados em [51] [54]. Segundo o autor, a abordagem em [51] [54] pode expandir o *schema merging* horizontalmente ou verticalmente. Assim, TuSMe combina as duas abordagens criando um *schema merging* mais balanceado, isto é, equilibrando tanto horizontalmente (em termos de atributos) como verticalmente (em termos de número de relações) o *schema merging*.

O conceito proposto no trabalho é criar uma forte coesão dos atributos de uma relação individual do *global conceptual schema* (GCS). O cálculo da coesão interna de uma relação do GSC é definido por uma atribuição de peso. Este peso permite o ajuste da expansão horizontal do GSC. O peso de uma relação é baseado nos pesos calculados para cada atributo da relação, que é calculado com a razão do número total de fonte de dados no GCS e o número de fontes de dados a partir de onde um particular atributo está recebendo o valor. O exemplo a seguir demonstra a abordagem proposta:

Exemplo [51]: Dado os seguintes *schemas* fonte Go-Travel e Ok-Travel. Onde Go-Travel é composto das relações:

Go-flight(f-num,time,meal) //meal is of Boolean type

Go-price(f-num,date,price)

Go-airline(airline,phone)

Ok-Travel inclui uma única relação: *Ok-flight(f-num,date,time,price,nonstop) //nonstop is of Boolean type*

A correspondência entre *schemas* é expressada por consultas conjuntivas usando a notação *Datalog*, que é uma linguagem de consulta não procedural baseada na linguagem de programação lógica Prolog. Assim é obtida as seguintes consultas:

Flight(f-num,date,time,price):- Go-flight(f-num,time,meal), Go-price(f-num,date,price)

Flight(f-num,date,time,price):- Ok-flight(f-num,date,time,price,nonstop)

Considerando que uma possível relação no GSC seja ***M.Flight(f-num,date,time,price,meal,nonstop)*** o cálculo do peso para esta relação seria:

Cálculo do peso de cada atributo é dado por $W(a_i) = AH(a_i)/N_{M.R}$, onde $AH(a_i)$ é numero de fonte de dados que fornece um valor para um particular atributos a_i. $N_{M.R}$ é número total de membros da fonte de dados. O peso da relação M.R($a_1, a_2...a_m$) é a média de cada $W(a_i)$. Assim, $W(M.R) = \sum(W(a_i))/m_{M.R}$, onde $m_{M.R}$ é o número total de atributos em M.R. Aplicando $W(a_i)$ e $W(M.R)$ temos:

W(M.Flight.F-num) = $2/2 = 1$

W(M.Flight.date) = $2/2 = 1$

W(M.Flight.time) = $2/2 = 1$

W(M.Flight.price) = $2/2 = 1$

W(M.Flight.meal) = $1/2 = 0.5$

W(M.Flight.nostop) = $1/2 = 0.5$

Ww(M.Flight) = $(1+1+1+1+0.5+0.5)/6 = 5/6 = 0.83$

O valor de W(M.R) mais próximo de 1 significa que os atributos de M.R, possui um mapeamento de atributos que estão muito próximo da fonte de dados. O contrário, o valor mais próximo de zero, os atributos não estão relacionados com os atributos da fonte de dados. Os $W(a_i)$ com baixo peso podem ser removidos de M.R formando novas relações. O autor expõe que a divisão de M.R pode aumentar a expansão vertical e que tratará desta questão em trabalhos futuros.

3.2.3 Semi-automatic Model Integration using Matching Transformations and Weaving Models

O trabalho apresentado por Del Fabro e Valduriez [16] propõe uma nova solução para a construção de definição transformação de modelo de forma semi-automática. Através de *matching tranformation* que são transformações que produzem relacionamentos entre elementos de um conjunto de modelos de entrada. *Matching transformation* pode criar ou adaptar novas heurísticas para construir *weaving models*. *Weaving model* captura relacionamentos entre elementos de modelos [21].

Outra proposta é um novo metamodelo baseado em heurística que explora características internas de metamodelos de entrada para produzir um *weaving model*.

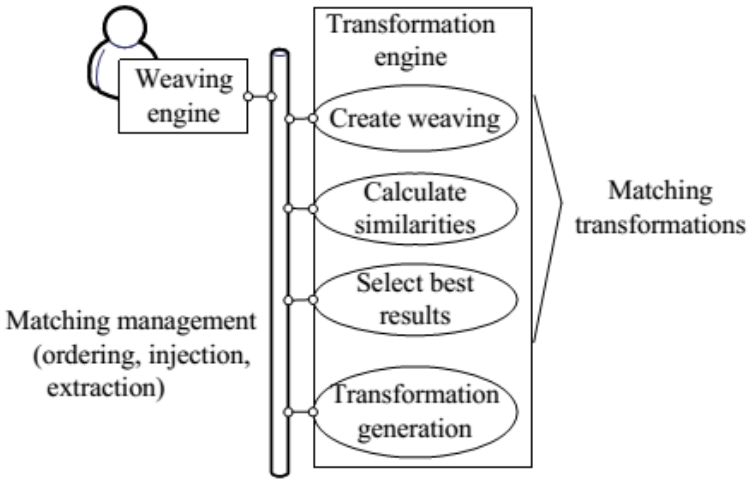


Figura 3.3: Arquitetura geral do *Semi-automátic Model Integration using Matching Transformations and Weaving Models* [16].

A Figura 3.3 mostra a arquitetura geral da proposta de Del Fabro e Valduriez, sendo composta dos seguintes módulos:

- *Weaving Engine*: fornece uma interface para construção e atualização de *weaving metamodels* e *weaving models*;
- *Transformation Engine*: permite a execução de transformação de modelos;
- *Matching Management*: gerencia a ordem de execução do *matching transformation* e sincroniza estas transformações com o *weaving engine*.

O trabalho utiliza o *weaving metamodel*, proposto em [16], para representar diferentes tipos de *links* que é retornado pelo *matching tranformation*. Cada tipo de *link* corresponde a uma padrão de transformação.

Operações de gerenciamento dos diferentes tipos de *matching tranformation* é proposto como os apresentados a seguir [16]:

- ***Creating Weaving Models*** que tem o objetivo de criar *weaving models*. O operador tem como entrada dois modelos e os transforma em um *weaving models*;
- ***AssignSimilarity*** que calcula a similaridade entre elementos dos modelos fonte e alvo, para cada *link* do *weaving models*. A similaridade é calculada baseada em métodos *element-to-element* e estrutural. Para o método *element-to-element* identificadores de elementos dos modelos fonte e alvo são comparados por similaridade de *string* e considerando um dicionário de sinônimo. A similaridade estrutural é calculada usando propriedades internas dos elementos do metamodelos como, tipos, cardinalidades, relacionamento entre elementos do modelos como restrições e árvore de herança;
- ***Select (Selecting Best Links)*** que seleciona somente links que satisfaçam um conjunto de condições. O operador possui um parâmetro <*condition*> que determina os critérios de seleção dos *links*. A seleção de *links* pode ser por filtragem e reescrita. O método de filtragem de *links* seleciona apenas os links como alto valor de similaridade para cada elemento do modelo fonte. O *weaving model* produzido pela filtragem de *links* contém um *link* para cada elemento do modelo fonte. O objetivo é encontrar um *link* para cada elemento do metamodelo fonte com elementos do metamodelo alvo. O método de reescrita de *links* utiliza padrões comuns como *nesting*, *inheritance*, *data conversions*, *concatenation*, *splitting* para transformar tipos simples de *links* em tipos complexos de *links*. Como por exemplo reescrever aninhamento entre elementos como o relacionamento de composição. Um exemplo desta reescrita de *links* seria classes e atributos ou tabelas e colunas.

Para a geração de transformação o trabalho propõe implementar transformações de alta ordem (higher-order transformations - HOT´s) que tem como objetivo traduzir extensões de *links* em *transformation rules* e *bindings* escritas em ATL [23].

3.2.4 Evolution of the COMA Match System

A ferramenta COMA para *schema matching* apresentada por Massmann et al. [41] tem como objetivo fornecer um sistema de *matching* genérico customizável. A ferramenta vem sendo desenvolvida a uma década, estando atualmente na versão denominada COMA 3.0.

Segundo o autor, a ferramenta apresenta pontos fortes e pontos a serem melhorados que são descritos a seguir [41]:

- **Pontos fortes**:
  - *Arquitetura multi-matcher*: a ferramenta usa estratégia de combinação de técnicas de match, além de reuso. Suporta match liguístico, estrutural e baseado em instância;
  - *Abordagem genérica*: a representação genérica de modelos como árvore acíclica direcionada permite aplicar matching a diversos tipos de esquemas e ontologias;
  - *Configuração padrão eficiente*: a configuração padrão oferecida é eficiente e sem necessidade de ajustes manuais. A configuração padrão utiliza *match* linguístico e estrutural e uma abordagem de combinação de *match*;
  - *Interface gráfica para usuário*: fornece interface que permite ao usuário importar e visualizar esquemas, configurar estratégia de match e corrigir mapeamentos;
  - *Personalização*: permite personalizar *matching* linguístico fornecendo dicionário de domínio, de sinônimos e abreviações;
  - *Estratégia de match avançado*: fornece o uso de reuso de match anteriores, matching de fragmentos dos esquemas;
  - *Repositório*: fornece o armazenamento de esquemas/ontologias importadas, além do matching realizados.
- **Pontos a melhorar**:
  - *Questões de escalabilidade*: a técnica de *matching* baseado no caminho leva a um problema de memória em *matching* de grandes esquemas. Isto ocorre

pelo armazenamento dos valores de semelhanças de pares de elementos. A aplicação de *match* de fragmento e *match* baseado em nó podem diminuir o problema, mas não é uma solução mais geral;

– *Esforço de configuração*: a configuração de parâmetros adequados para tarefa de *matching* ainda precisa de um esforço manual do especialista de domínio. O sistema deve ajudar a encontrar a melhor estratégia para o *match*;

– *Semântica limitada do mapeamento de match*: o mapeamento deve ir além de apenas simples correspondências entre elementos de esquemas/ontologias. Relacionamentos como *containment* ou *is-a* devem ser agregado a tarefa de *match*;

– *Acessibilidade limitada*: sistema é de uso *stand-alone*, não projetado para interagir com outros programas.

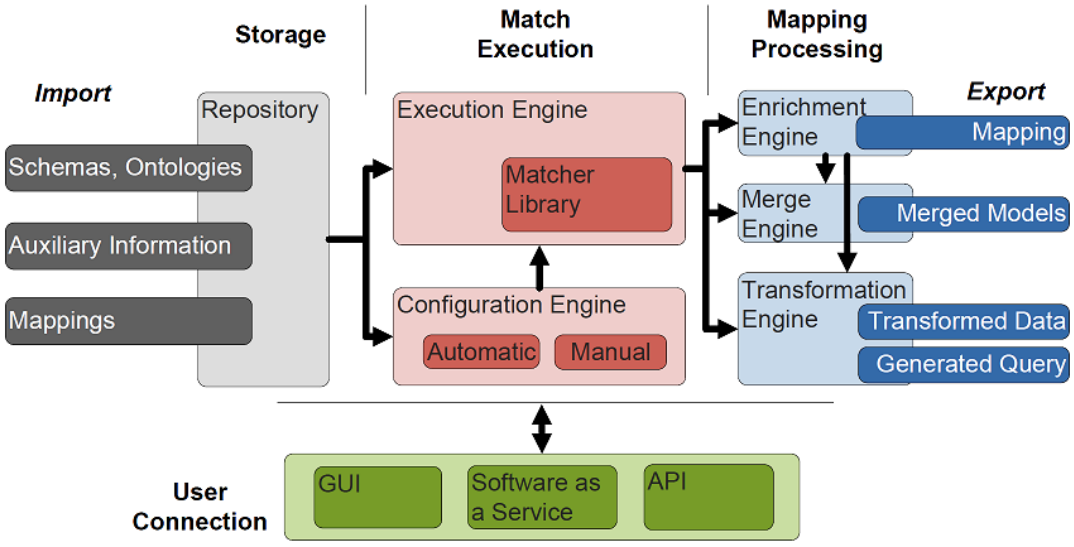


Figura 3.4: Arquitetura da ferramenta COMA 3.0 [41].

A ferramenta COMA 3.0 teve sua primeira versão em 2002, chamada de COMA (***co**mbined **ma**tching*). A Figura 3.4 mostra a atual arquitetura da ferramenta que é composta pelos os módulos que segue [41]:

- ***Configuration Engine*** que permite de forma automática (configuração padrão) ou manual determinar o fluxo de *schema matching*;
- ***User Interfaces***: uma interface para o usuário gerenciar o processo de match;

- ***Enrichment Engine*** : o processo de match identificar correspondências com relacionamento 1:1 entre elementos de esquemas/ontologias. Para tratar correspondências complexas são propostas funções de transformações de dados. Uma das funções proposta faz correspondência de um elemento fonte com um ou mais elementos alvos (Ex: *S1.name -> S2.firstName,S2.lastName*). O *Enrichment Engine* também suporta correspondência semântica entre ontologias como o relacionamento *is-a* entre elementos de ontologias.

- ***Merge and Transformation Engines*** : suporta *merging* de ontologias dirigida por *match*, como também geração de consultas para transformação de dado.

Resumidamente, o COMA 3.0 tem como estratégia de *match* baseada em *matching* dependente de contexto, *matching* baseado em fragmento e *matching* orientado a reuso. A ferramenta suporta tanto *matching* de *schemas* como de ontologias. A estratégia para *string match* inclui tarefas de processamento de entrada de string como remoção de *stop word* e resolução de abreviações e sinônimos.

3.2.5 Rondo: A Programming Platform for Generic Model Management

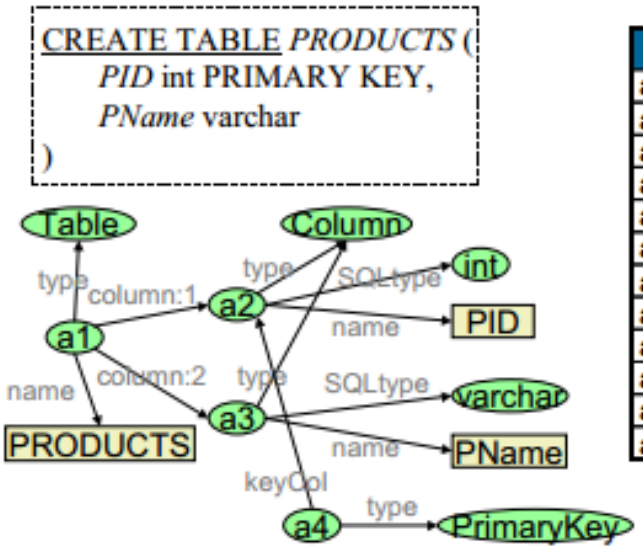
O trabalho apresentado por Melnik et al. [43] propõe uma ferramenta que implementa um conjunto de operadores que gerenciam modelos (modelos relacionais, esquemas XML, visões SQL), permitindo deste modo, a propagação de modificações no modelo. Segundo o autor a pesquisa apresenta as seguintes contribuições [43]:

- Propõe um estrutura conceitual para representação de modelos e mapeamentos. O mapeamento entre modelos é denominado de *morphisms* e a estrutura é denominada de *selector*;

- Define a semântica dos operadores-chave da gestão de modelo das estruturas conceituais, além de sugerir novos operadores genéricos;

- Apresenta um algoritmo para os operadores *Extract* e *Merge*.

O gerenciamento de modelos envolve a manipulação de artefatos como esquemas relacionais, esquemas XML, entre outros. O trabalho [43] propõe que estes



S	P	O	N
a1	type	Table	
a1	column	a2	1
a1	column	a3	2
a1	name	"PRODUCTS"	
a2	type	Column	
a2	SQLtype	int	
a2	name	"PID"	
a3	type	Column	
a3	SQLType	varchar	
a3	name	"PName"	
a4	type	PrimaryKey	
a4	keyCol	a2	

Figura 3.5: Exemplo de um *graph* representando uma relação [43].

artefatos sejam representados por um *directed labeled graphs*. Introduz também, os conceitos de *morphisms* e *selectors*. O *morphisms* consiste em definir correspondências n:m entre elementos de dois modelos (nó de dois grafos). *Selectors* representar o conjunto de elementos usados no modelos.

Na representação de modelo através de *directed labeled graphs*, os nós dos *graphs* identificam elementos dos modelos. Os elementos são identificados unicamente no modelo através de um identificador de objetos (OID). Um *directed labeled graphs* é composto por um conjunto de *edges* <*s*, *p*, *o*> onde *s* é o nó fonte, *p* é o rótulo de *edge* e *o* é o nó alvo.

Um *graph* pode representar uma relação M composta de 4 atributos M(S: OID, P: OID, O: OID ∪ LITERAL, N: Integer), onde N é um atributo opcional com função de ordenar e *S, P, O* que formam chaves únicas. A Figura 3.5 mostra um exemplo da representação de uma tabela relacional e os valores para os atributos *S, P, O* e *N*.

Para gerenciar os modelos representados nos *graphs*, o autor apresenta um conjunto de operadores. Inicialmente é demonstrados os operadores que o autor denomina como primitivos. Os operadores primitivos são apresentados na Figura 3.6, onde as definições são descritas em SQL.

O operador *Domain* retorna elementos da esquerda de um *morphism*. O *RestrictDomain* permite filtrar elemento de um *Domain*. O operador *Invert*, troca elementos da esquerda e direita de um *morphism*. O operador *Compose* cria um novo mapeamento

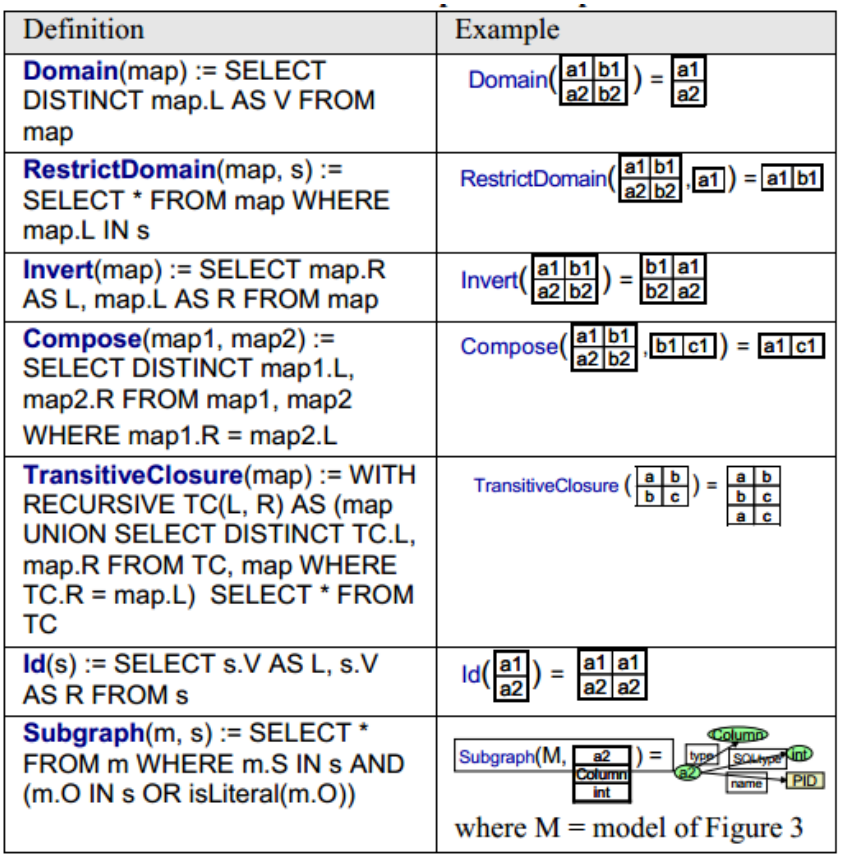


Figura 3.6: Operadores primitivos e suas representações [43].

a partir de outros dois mapeamentos. O operador *TransitiveClosure* implementa um *matching* transitivo, isto é, se A = B e B = C, então A = C. O operador *Id* cria um identificador de *morphism*. E o operador *Subgraph* retorna um fragmento de um *graph* a partir de um elemento. Além dos primitivos, são apresentados operadores derivados (Range, RestrictRange, Tranverse e Restrict), os operadores *Extract* e *Delete* e os operadores *Match* e *Merge*.

A implementação do operador *Match* usa o *algoritmo de Simalarity Flooting*(SF) [42]. O operador *Match* possui um atributo Sim que registra um valor de similaridade para cada par de elemento do *graph*. O operador *Match* foi implementado da seguinte forma:

operator Match(m1, m2, seed)

multimap = *SFjoin*(m1, m2, seed);

multimap = *Restrict*(multimap, m1, m2);

map = FilterBest(multimap);

return (map, multimap);

O operador *SFJoin* implementa o algoritmo SF. O algoritmo SF pode retornar um grande número de nós *m1* e *m2*, sendo necessário fazer um filtro com a função *FilterBest*. Também, é aplicada a função *Restrict* para restringir o resultado de *SFjoin*. O cálculo de similaridade entre literais de *m1* e *m2* é realizado pela função *NGranMatch*.

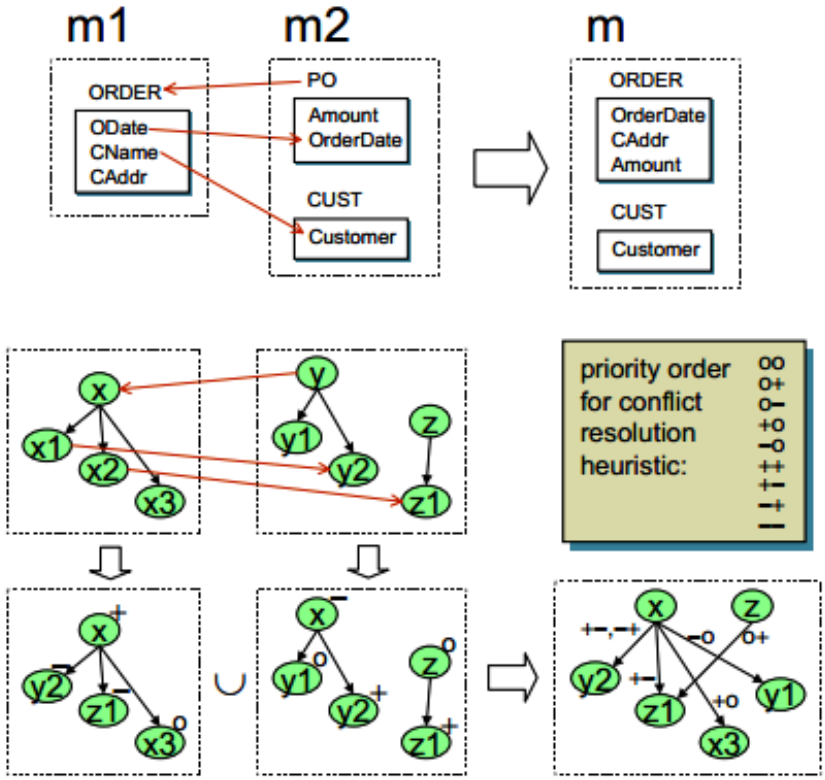


Figura 3.7: Exemplo aplicação do operador *Merge* [42].

Para o processo de *merging* o trabalho apresenta o operador *Merge*. O operador *Merge* é implementado pelo algoritmo denominado *GraphMerge*. O algoritmo define três etapas para a realização do *merging*, que são: renomear nó, união de *graph* e resolução de conflitos.

O topo da Figura 3.7 mostra um exemplo da aplicação do operador *Merge* aos modelos *m1* e *m2* retornando o modelo *m* com o merge. O *morphism map* é definido por setas, que determinam a preferência entre dois elementos do modelo. Elementos dos modelos fontes são descartados quando dois elementos dos modelos sofrem *merging*. A direção das setas pode ser alterada pelo especialista de domínio. O algoritmo do operador *Merge* segue os passos seguintes [43]:

1. Primeiro é feita a renomeação de nós alvo, como mostra a Figura 3.7, onde y, x1 e x2 foram renomeados para x, y2 e z1, respectivamente;

2. Segundo, acontece a união do *graph*, que faz o *merging* de dois conjuntos de *edges*;

3. Terceiro, acontece a resolução de conflitos. Nos conflitos da Figura 3.7 ou se elimina a *edge* entre *x* e *z1* ou então, se elimina a *edge* entre *z* e z1, criando o atributo *Customer* ou na relação CUST ou ORDER no esquema unido (merge). Esta escolha fica a cargo do especialista de domínio.

A Listagem 3.1 mostra o algoritmo *GraphMerge* de forma resumida [43]:

Listagem 3.1: Algoritmo *GraphMerge* [43].

```
1 AlgorithmGraphMerge(m1, m2, map)
2    M:= m1 U m2; L:= empty list; G:= empty graph
3    for each edge ein M do
4       rename nodes of eusing map; assign tag to e; append eto L;
5    end for
6    sort edges in Lby decreasing tag priority;
7    maxN:= SELECT max(M.N) FROM M;
8    while L not empty do
9       take edge e=<s, p, o, n> off top of L;
10      if tag(e) one of {"-o", "-+", "--"} then
11         n:= n+ maxN;
12         if o is literal then continue loop end if
13      end if
14      if exists e' = <s, p, o, n'> in G then
15         replace e' in G by <s, p, o, min{n, n'}>;
16      else if not conflictsWith(<s, p, o, n>, G)then
17            append <s, p, o, n>to G;
18         end if
19      end if
20   end while
21 return G
```

O *GraphMerge* pode ser aplicado aos diferentes tipos de modelos pela implementação da função conflictsWith() apropriada. O protótipo apresentado no trabalho implementa o algoritmo de *merging* para esquema relacional, esquema XML e visões SQL. O operador *Merge* proposto é implementado conforme Listagem 3.2 [43].

Listagem 3.2: Implementação operador *Merge* [43].

```
operator Merge(m1, m2, map){
   G = GraphMerge(m1, m2, map);
   s = SELECT L FROM map WHERE Dir="->" UNION SELECT R FROM map WHERE Dir="<-";
   m1G = RestrictDomain(map, All(m1) intersection s) + Id(All(m1) - s);
   m2G = RestrictDomain(map, All(m2) intersection s) + Id(All(m2) - s);
   m, mG)= Copy(G, All(G));
   return (m, m1G * Invert(mG), m2G * Invert(mG) );
}
```

O operador *Merge* retorna um *morphism* entre dois modelos de entrada, mas também, retorna os *morphisms* m1G e m2G. O algoritmo proposto para o *Merge* não cria novos elementos ou novas relações entre elementos existentes, isto é, não reorganiza os esquema para resolver conflitos.

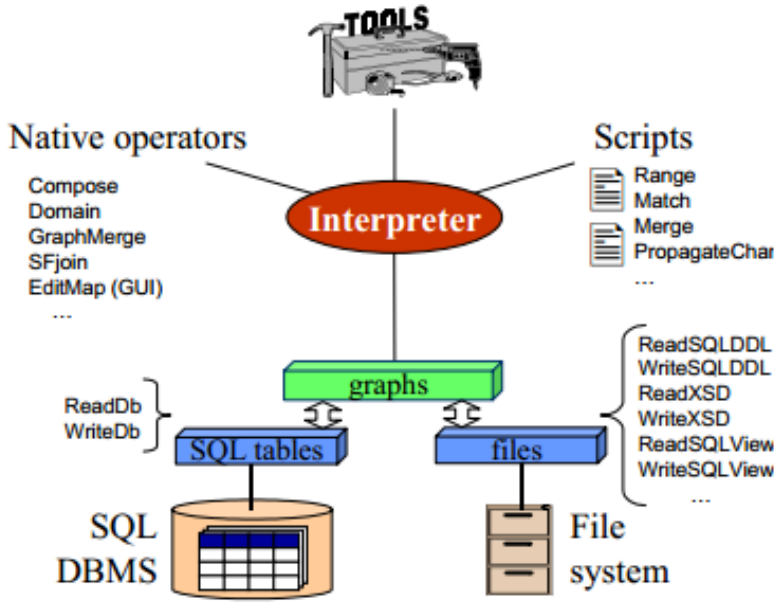


Figura 3.8: Arquitetura protótipo Rondo [42].

Para implementar os operadores de gerenciamento de modelos propostos no trabalho foi construído o protótipo Rondo [43]. A Figura 3.8 apresenta a arquitetura de Rondo [43].

O núcleo do protótipo é o *interpreter* que coordena o fluxo de execução dos operadores. Operadores como *Match* e *Merge* podem ser definidos por *scripts*. Operadores como o GraphMerge e o EditMap, que edita *morphisms*, são definidos por uma

implementação nativa. O protótipo suporta as linguagem de modelagem de *schema* SQL DDL, XML *schema*, RDF *schema* e SQL *views*.

3.3 Análise dos trabalhos relacionados

Nesta seção, os trabalhos relacionados são avaliados levando em considerações parâmetros que acredita-se serem relevantes para se fazer uma comparação com o trabalho de pesquisa proposto nesta dissertação. Os parâmetros utilizados na avaliação são os seguintes:

1. Utiliza o paradigma MDE para abordar o problema de *merge* de esquema;
2. Permite realizar o *matching* e/ou *merge* de diferente linguagens de esquema (XML, Relacional, Ontologia);
3. Permite *matching* e/ou *merge* de fragmento de esquema. O especialista de domínio pode realizar o *matching* e *merging* apenas de parte dos esquemas;
4. Uso de informações auxiliares para determinar a similaridade entre elementos dos esquemas;
5. Trata o surgimento de valores nulos quando esquemas relacionais são fundidos;
6. Permite o uso de diferentes algoritmos de *matching* e *merging*;
7. Gera o script do *merge* dos esquemas.

Analisando os conteúdo da Tabela 3.1, algumas considerações são necessárias, sendo descritas a seguir.

Del Fabro em [16] propõe a construção de definição de transformação de modelos de forma semi-automática. As definições de transformação são construídas a partir de um modelo *weaving* criado do *match* de modelos fonte e alvo. O artigo não descreve qual tratamento é dado para os atributos não correspondidos, quando classes do modelo fonte e alvo possuem apenas alguns atributos similares. Também não consta no artigo se é possível *merge* de fragmentos dos modelos.

O trabalho apresentado em [25] trata do problema do surgimento de valores nulos e crescimento horizontal da estrutura do esquema do *database* no processo de

Itens Avaliados	[16]	[25]	[41]	[43]	[38]
Utiliza o paradigma MDE para abordar o problema de *merge* de esquema	sim	não	não	não	não
Permite realizar o *match* e/ou *merge* de diferente linguagens de esquema(XML, Relacional, Ontologia)	sim	não	sim	sim	sim
Uso de informações auxiliares para determinar a similaridade entre elementos dos esquemas	sim	não	sim	não	sim
Permite *matching* e/ou *merge* de fragmento de esquema	não descrito no artigo	não descrito no artigo	sim	sim	não descrito no artigo
Trata o surgimento de valores nulos quando esquemas relacionais são fundidos	não descrito no artigo	sim	não descrito no artigo	não descrito no artigo	não descrito no artigo
Permite o uso de diferentes algoritmos de *matching* e *merging*	sim	não	sim	sim	não
Gera o *script* do *merge* dos esquemas	sim. Definições em ATL	não	não	sim	não

- *Semi-automatic Model Integration using Matching Transformations andWeaving Models* [16]
- *Tuned Schema Merging* (TuSMe) [25]
- *Evolution of the COMA Match System* [41]
- *Rondo: A Programming Platform for Generic Model Management* [43]
- *Generic Schema Matching with Cupid* [38]

Tabela 3.1: Análise dos Trabalhos relacionados

merge entre *database*. O trabalho não expõe a estratégia utilizada para realizar o *match* entre elementos de modelos relacionais. Também não é informado no artigo se foi desenvolvida uma ferramenta para implementar a abordagem proposta.

Em [43], grande parte dos critérios são atendidos pela abordagem. O tratamento de conflitos ocasionado pelo *merge* não são bem explicados no artigo. Existe apenas o uso de uma função que tem sua implementação adaptada dependendo do tipo de esquema que será correspondido.

COMA [41] é uma proposta de abordagem genérica de *match*. A abordagem faz *match* de esquema XML, esquema relacional e ontologias. COMA atende a maioria dos critérios apresentado na Tabela 3.1 e vem sendo desenvolvida a mais de 10 anos. Com uma interface que oferece bons recursos de edição e visualização de *match*. Além disso, oferece ao especialista a diversas possibilidades de ajustes para o processo de *match*.

CUPID apresenta uma proposta genérica para *matching* de XML *schema* e esquema relacional. O artigo não descreve se o CUPID também gera o *merge* dos esquemas, trata apenas do *schema matching*. O artigo não entra em detalhes de sua arquitetura e interface. Na há detalhes no artigo sobre a possibilidade de *match* de fragmentos dos esquemas.

3.4 Síntese

Este capítulo apresentou pesquisas acadêmicas desenvolvidas no campo do *matching* e *merging* de esquema. Esta pesquisa bibliográfica demonstrou as diversas abordagens que já foram propostas para tratar o problema de *matching* e *merging* de esquemas. Percebe-se que ao longo do tempo um grande esforço vem sendo realizado pelos pesquisadores para encontrar soluções que auxiliem o processo de encontrar correspondências entre esquema relacional, esquema XML, ontologias e metamodelos, possibilitando assim, um *merge* desses esquemas menos dispendioso.

Cinco trabalhos que tratam do *merge* e *match* de esquemas foram descritos. O trabalho [16] utiliza a abordagem MDE para criar um modelo unificados de dois outros modelos. Os demais trabalhos utilizou a abordagem de desenvolvimento tradicional de suas implementações de ferramentas para implementar suas propostas de *match* e *merge*.

Finalizando o capítulo, critérios de avaliação dos trabalhos foram sugeridos. Estes critérios foram exposto em um tabela, onde foi indicado quais critérios cada trabalho atendeu. Os critérios foram sugeridos considerando medidas que permitissem uma comparação com o trabalho desenvolvido nesta pesquisa.

4 UMA ABORDAGEM MDE PARA INTEGRAR MODELOS DE BASE DE DADOS

Neste capítulo, a nossa abordagem baseada em MDE para integrar esquemas de bases de dados será descrita, incluindo a metodologia, os metamodelos e os algoritmos. A metodologia a ser utilizada de guia para o desenvolvimento do *framework* para integrar esquema de base de dados é apresentada. Os metamodelos necessários para a tarefa de integração de esquema de dados são propostos, além de algoritmos de *matching* e *merging* para obtenção do modelo integrado de base de dados.

Kurtev et al. em [29] estuda as características de alguns espaços tecnológicos e seu uso para resolver problemas particulares no contexto da engenharia de *software*. Um espaço tecnológico é um ambiente de trabalho formado por conceitos, ferramentas, conhecimentos e habilidades para solução de um problema da engenharia de *software* [29]. Linguagem de programação, engenharia de ontologia e MDA são exemplos de espaços tecnológicos. Os espaços tecnológicos não são isolados, assim podem haver pontes entre um espaço tecnológico e outro. A Figura 4.1 ilustra as pontes entre alguns espaços tecnológicos. Estas pontes significam que determinadas operações podem ser realizadas de forma mais fácil em um espaço tecnológico e depois importado para outro [29].

A abordagem MDE para integrar bases de dados heterogêneas, pode ser baseada em pontes entre o espaço tecnológico DBMS e o espaço tecnológico MDE. Desta forma, as bases de dados a serem integradas são importadas para espaço tecnológico da MDE, integrados, e depois exportados para espaço tecnológico do DBMS. Sendo assim, algumas terminologias utilizadas no contexto de MDE e no contexto de DBMS devem ser esclarecidas, como:

- No contexto de banco de dados, tem-se linguagem de descrição de esquema. No contexto MDE, tem-se metamodelo;
- No contexto de banco de dados, tem-se esquema. No contexto MDE, tem-se modelo;

- No contexto de banco de dados ,tem-se *schema matching*. No contexto MDE, tem-se *model matching* ;
- No contexto de banco de dados, tem-se *schema merging*. No contexto MDE, tem-se *model merging*.

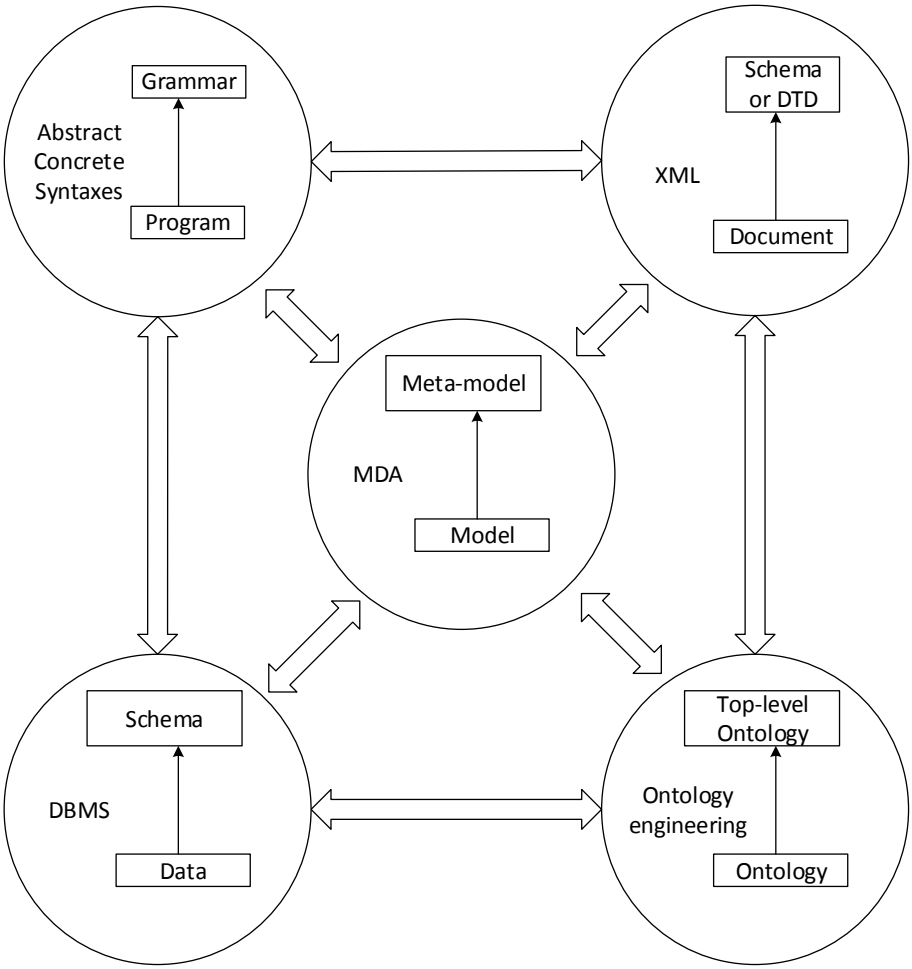


Figura 4.1: Pontes entre os espaços tecnológicos [29].

4.1 O *framework* SID (Semi-Automatic Integration Database for MDE - SID4MDE)

Nesta seção, o *framework* para suportar a integração de esquema de base de dados no contexto da MDE é apresentado. O *framework* SID tem o objetivo de auxiliar o especialista de domínio na tarefa de integrar esquemas de dados heterogêneos de

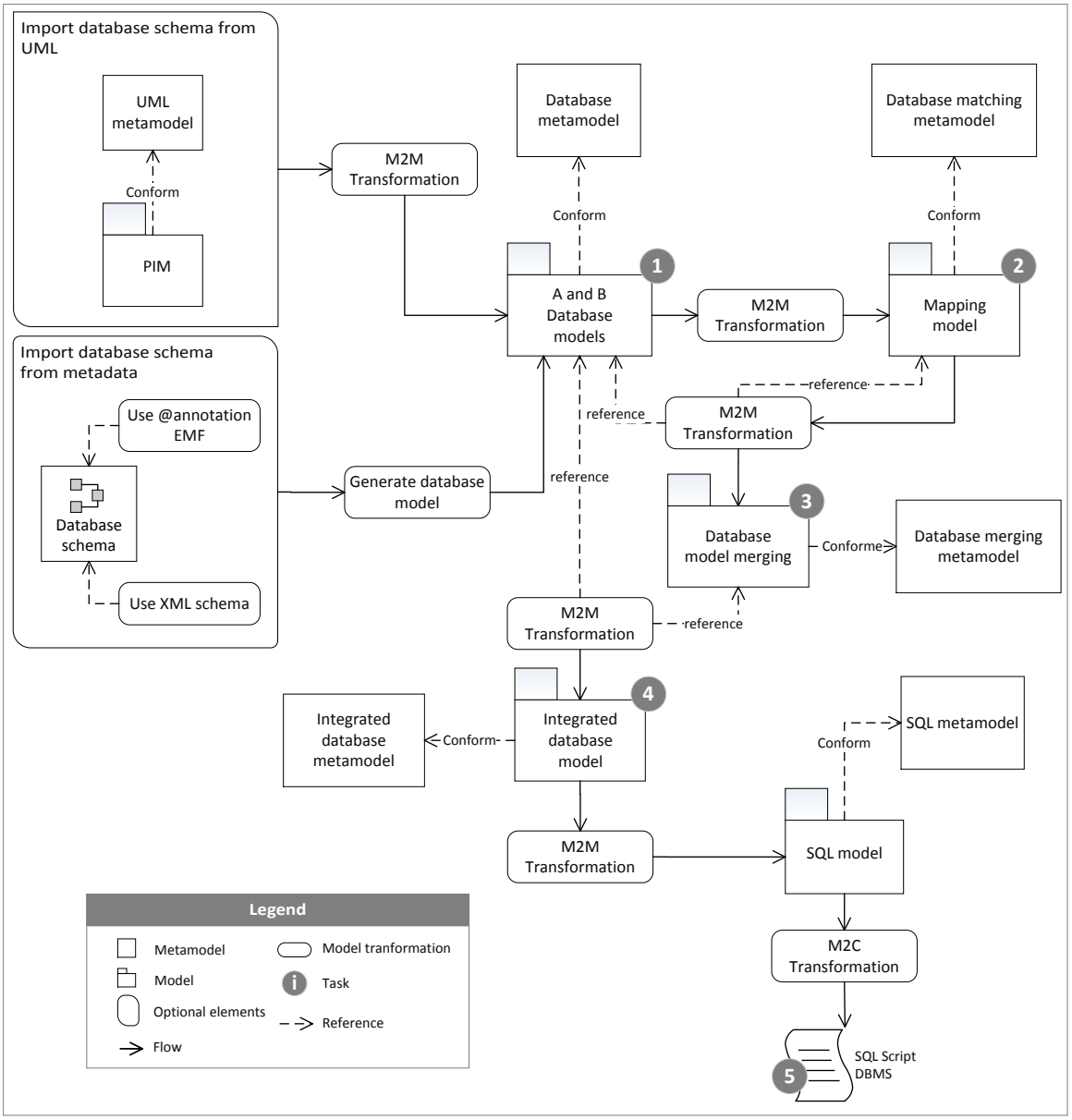


Figura 4.2: *Framework* para suportar a integração de base de dados.

forma semiautomática. Semiautomática significa dizer que há a participação humana no processo de integração, isto é, no processo de *matching*, o *framework* faz sugestões de possíveis elementos correspondentes entre dois esquemas e o especialista de domínio confirma as correspondências encontradas.

A abordagem proposta para se obter um modelo integrado de base de dados heterogêneas envolve cinco etapas. Observando a Figura 4.2, percebe-se que o produto final que se deseja alcançar é um *script* SQL com a estrutura do banco de dados integrado que contém a fusão (*merging*) dos modelos de base de dados recebidos

como entrada. Para isso, são necessárias execução das seguintes tarefas(veja Figura 4.2):

1. **Obter modelos dos esquemas de base de dados**: esses modelos são construídos conforme *database metamodel* e representam os esquemas de entrada a serem correspondidos;

2. **Obter *mapping model* entre os modelos de base de dados**: modelo construído conforme *database matching metamodel*, com a finalidade de registrar correspondências entre *database models*;

3. **Obter *database model merging***: modelo construído conforme *database merging metamodel*, com a finalidade de registrar elementos que irão compor o *integrated database model*, isto é, elementos que sofrerão ou não *merging*;

4. **Obter *integrated database model***: modelo construído conforme *integrated database metamodel*, com a finalidade de especificar que elementos farão parte do *script* SQL a ser gerado;

5. **Obter *scrip* SQL**: geração do *script* SQL pronto para ser executado em um SGBD que disponibilizará a estrutura da base de dados integrada.

A seguir, a metodologia para gerar todos os modelos necessários para construção do SQL *script* do modelo integrado é detalhada.

4.2 Metodologia para Geração de Modelo Integrado de Base de Dados

A metodologia proposta tem como objetivo guiar o processo de integração de esquema no contexto da MDE, definindo os metamodelos necessários para a tarefa de integração, bem como, quando os modelos são instanciados e transformados.

A metodologia envolve as fases de importar ou criar os modelos de esquema de base de dados, *schema matching* dos modelos de esquema de base de dados, *schema merging* dos modelos de esquema de base de dados, geração do modelo de esquema integrado e por fim, a geração do *script* SQL. Veja a Figura 4.3 que ilustra toda a metodologia.

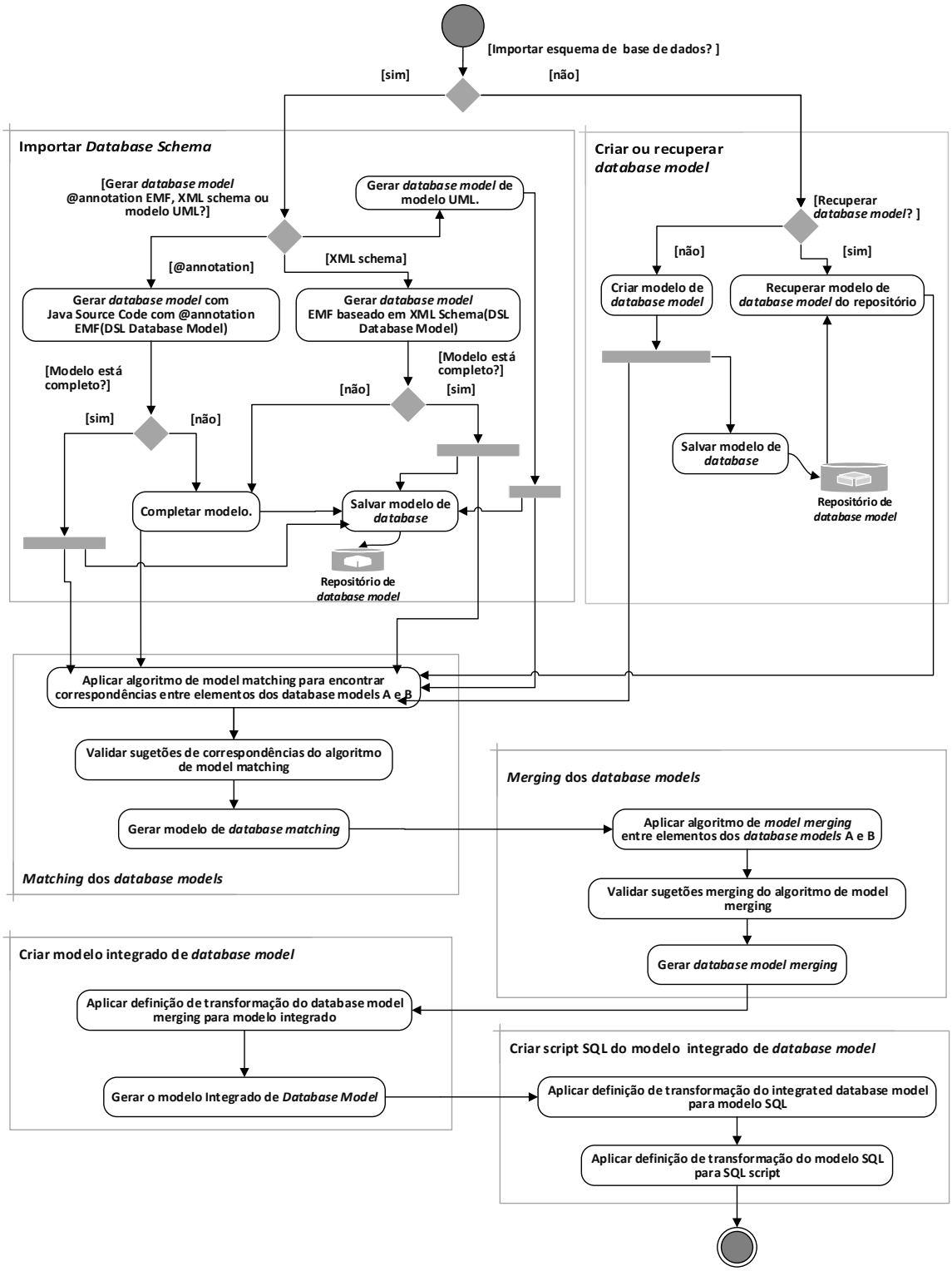


Figura 4.3: Metodologia para suportar integração de base de dados.

4.2.1 Importar esquema de base de dados

Nesta etapa, o especialista de domínio pode escolher entre três opções de obtenção do esquema de base de dados. Através da leitura de metadados de um banco de dados é possível construir o modelo de esquema de base de dados. Esta leitura pode ser feita pelo uso de dois recursos do *Eclipse Modeling Framework* - EMF, que são:

- Uso de código Java, através de @annotation para fazer a leitura dos metadados da base de dados, permitindo assim, que um modelo de esquema de base de dados seja criado conforme metamodelo de esquema de base de dados que é proposto neste trabalho;
- Uso de XML Schema, onde um esquema XML com informações de metadados do esquema de base de dados são carregados, possibilitando a definição de elementos do modelo de esquema de base de dados conforme metamodelo de esquema de base de dados.

Outra forma de obter os modelos de esquema de base de dados a serem integrados é com o uso de transformação de modelos. Modelos UML podem ser transformados em modelos de esquema de base dados através da aplicação de definição de transformação e de sua execução. O modelos importados são armazenados em um repositório de modelos para posterior uso no processo de integração de base de dados.

Caso a importação resulte em um modelo sem todos os elementos do esquema fonte importado. O especialista pode completar o modelo através de uma interface de manipulação de modelos.

4.2.2 Criar ou recuperar modelo de base de dados

O especialista de domínio pode criar seus *database models* diretamente no EMF. O EMF oferece o recurso de instanciar modelos conforme um metamodelo, assim, instância de um *database model* fonte podem ser definidos para uso no processo de integração. Modelos construídos dessa maneira ficam armazenados em um repositório de modelos para uso no futuro.

O uso do repositório de modelos permite que o especialista de domínio busque *database models* que foram salvos pelo processo de importação e pelo processo de

criação de modelos. Assim, o especialista seleciona, dentre os modelos existentes, os modelos que deseja fazer integração de seus esquemas.

4.2.3 *Matching* dos modelos de base de dados

Como visto na seção 2.2, *schema matching* é a tarefa de corresponder elementos de dois esquemas [55]. Assim, nesta etapa acontece todo o processo de busca de elementos correspondentes entre dois *database models*.

O objetivo desta fase é ter o *mapping model* com o mapeamento indicando que um elemento e_1 do modelo M_1 corresponde ao elemento e_2 do modelo M_2. Em seção posterior, será visto que um elemento de um modelo fonte pode corresponder a mais elementos de um modelo alvo. Para obter o *mapping model* é aplicado um algoritmo de *model matching* que busca por correspondências entre os *database models*. O algoritmo de *model matching* será apresentado na seção 4.5.

4.2.4 *Merging* dos modelos de base de dados

Para o processo de integração de base de dados, o operador *merging* implica em unificar os *database models* de forma que elementos que são correspondentes devam ser únicos no modelo integrado. Porém, alguns elementos que não são sobrepostos também devem ser repassados para o esquema integrado [51].

Assim, para criar o *database model merging* é preciso ter como entrada os *database models* e mais o *mapping model*. Desta forma, elementos presentes no *mapping model* serão unificados e elementos sem correspondência nos *database model* fonte e alvo serão repassados para o modelo integrado. O *database model merging* proposto nesta pesquisa registra apenas referências aos elementos dos *database model* fonte e alvo que sofrerão *merging*.

4.2.5 Criar modelo integrado de base de dados

O *integrated database model* é obtido com a transformação do *database model merging*. No *integrated database model*, diferentemente do *database model merging*, temos

exatamente a estrutura do esquema de base de dados que representa a integração dos *database models* fonte e alvo.

Esta estrutura é composta por todas as entidades, atributos e relacionamentos que farão parte do banco de dados escolhido para conter a base de dados integrada. Através de definições de transformação este modelo será transformado em um modelo específico de plataforma(PSM), no caso um *script* SQL.

4.2.6 Criar SQL *script* do modelo integrado de base de dados

Nesta etapa, é obtido o *script SQL* com a estrutura do banco de dados que define o esquema de base de dados em um DBMS específico. Neste ponto, há uma transformação de modelo para código, onde são aplicadas definições de transformação no modelo integrado de base de dados para gerar o *script* SQL de um banco de dados específico.

Desta forma, o *database integrated model* pode ser transformado para diversos banco de dados. Para isso, é necessário criar o metamodelo do banco de dados alvo e as regras de transformação do *database integrated model* para o modelo do banco de dados alvo.

4.3 Metamodelo propostos para integrar base de dados

Metamodelos apresentados nesta seção tem por objetivo auxilar a metodologia proposta. Metamodelos para *model matching*, *model merging*, *database model* e *integrated database model* são detalhados.

4.3.1 Metamodelo de base de dados

Para integrar esquemas de base de dados é preciso uma representação das estruturas dos esquemas das bases de dados que serão integradas. Assim, um metamodelo de base de dados é proposto para criar modelos que representam o *database* fonte e alvo. O *database metamodel* permite ao especialista de domínio definir as entidades,

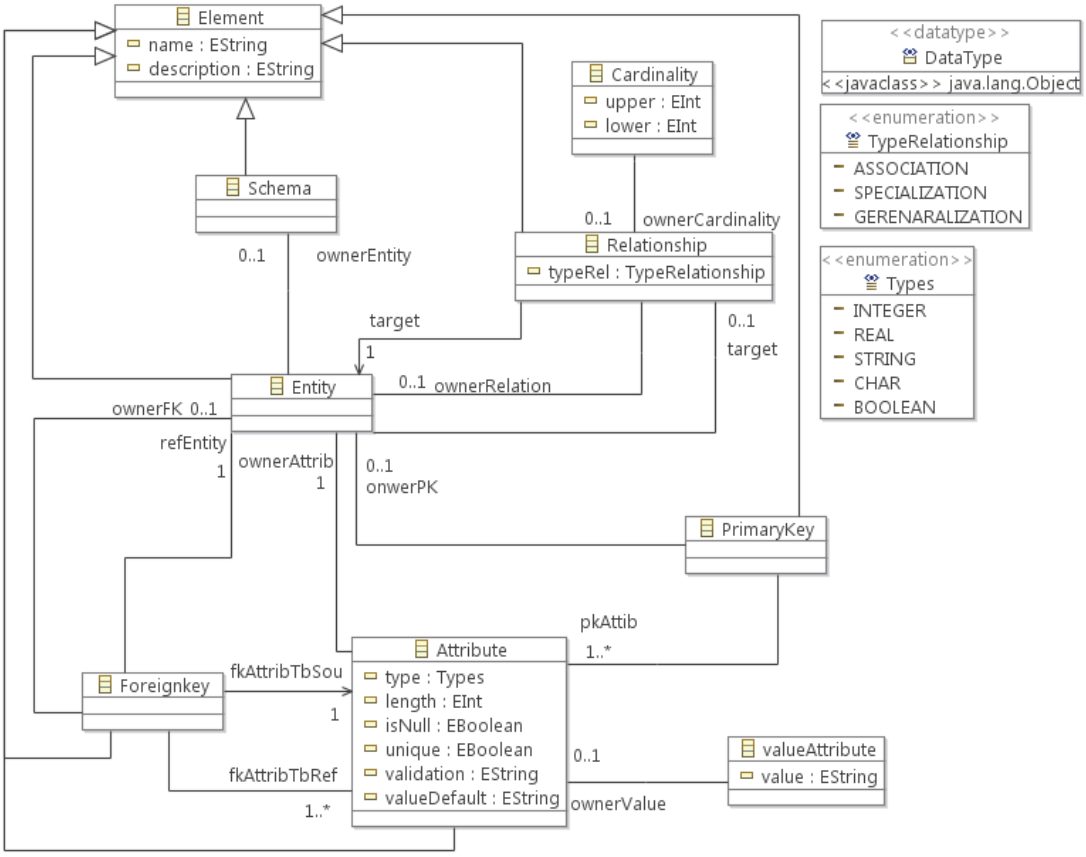


Figura 4.4: Metamodelo de definição de base de dados.

atributos de entidades e relacionamentos que estão presentes nos *database* fonte e alvo. A Figura 4.4 mostra o *database metamodel* que é composto pelos seguintes elementos:

- *Element*: é uma generalização para outros elementos do metamodelo, onde o atributo *name* identifica o nome do elemento e o atributo *description* contem uma descrição sobre o elemento;
- *Schema*: contém o nome do esquema do banco de dados relacional;
- *Entity*: contém as entidades do banco de dados relacional;
- *Attribute*: contém os atributos de cada entidade definida no banco de dado relacional, onde temos os atributos:
  - *type*: tipo de dado do atributo;

  - *length*: tamanho do tipo de dados. Por exemplo: se o *type* for inteiro, o valor para este atributo poderia ser 5, determinando assim que apenas números inteiros com 5 dígitos seriam aceitos;
  - *isNull*: booleano para determinar se o atributo pode ser nulo ou não;
  - *unique*: determina se o atributo é único;
  - *valitation*: permite armazenar instruções de validação do atributo;
  - *valueDefault*: contém um valor padrão para o atributo.

- *ValueAttribute*: contém valores do domínio de negócio. O valor do atributo poder ser usado para encontrar corresponderias entre atributos de duas entidades que não possuem correspondência por nome. Neste caso, o conteúdo dos atributos são comparados para busca de *match*;
- *ForeignKey*: contém as chaves estrangeiras de uma *Entity*;
- *PrimaryKey*: contém a chave primária de uma *Entity*;
- *RelationShip*: contém os relacionamentos entre *Entity*. O atributo *typeRel* determina o tipo do relacionamento entre as entidades;
- *Cardinality*: contém a cardinalidade de um relacionamento entre *Entity*. Os atributos *upper* e *lower* determinam quantos registros de uma entidade se relacionam com registros de outra entidade;
- *Types*: enumeração que contém os tipos de dados primitivos;
- *TypeRelationShip*: enumeração que contém os possíveis tipos de relacionamentos entre entidades da base de dados.

Como mostra a Figura 4.3, os *database models* criados pelo especialista de domínio ou importados, são armazenados em um repositório de *database models*. Assim, modelos podem ser recuperados para uso em uma tarefa de integração de modelos.

4.3.2 Metamodelo para mapeamento

O processo de *matching* de *database models* tem como retorno um mapeamento entre dois modelos. O mapeamento relaciona um elemento e_1 do modelo M_1

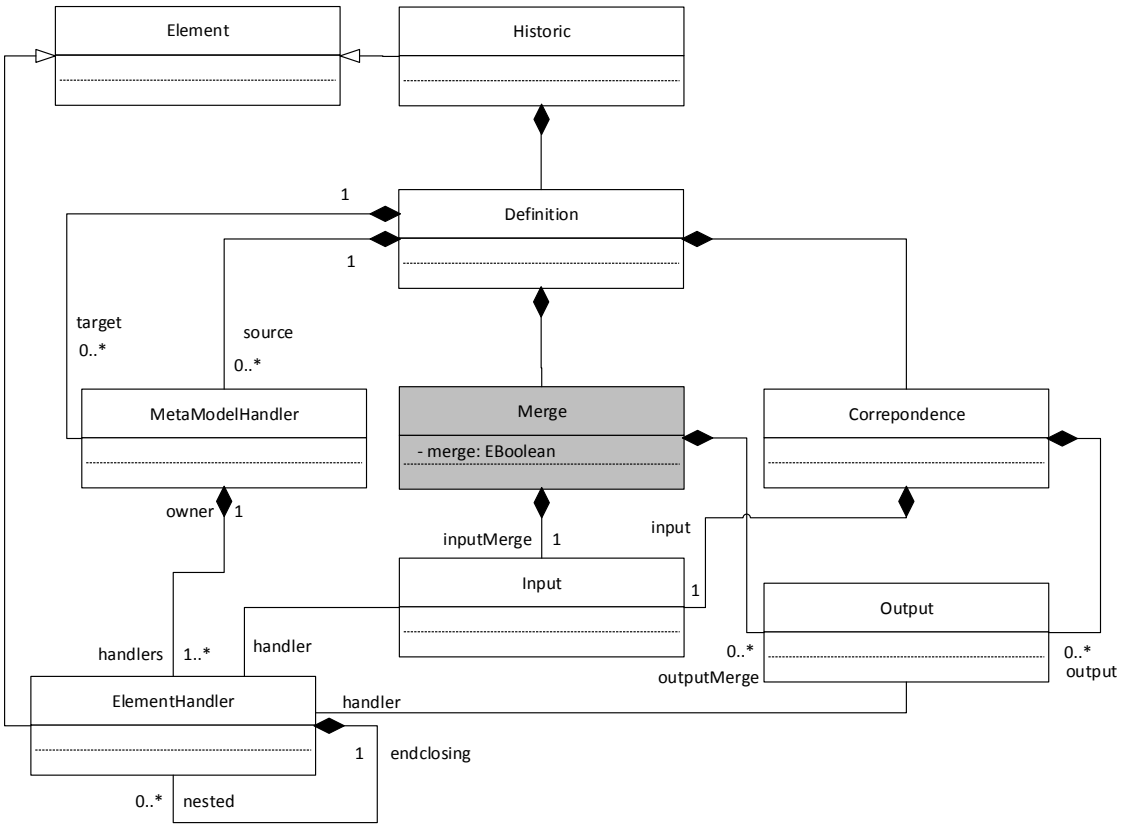


Figura 4.5: Fragmento do metamodelo de mapeamento (adaptado de [33]) .

com um ou mais elemento e_2 do modelo M_2, demonstrando assim, um relacionamento existente entre instâncias de dois modelos [9]. O mapeamento obtido entre modelos raramente é completo, isto porque, nem sempre é possível mapear todos os conceitos de um modelo com todos o conceitos de um outro [37]. No contexto MDE, o mapeamento entre metamodelos é um passo necessário para a transformação de modelos e tem sido tratado em muitos trabalhos de pesquisas como em [9] [17] [61] [33] [35].

Para registrar o mapeamento resultante do *database model matching* é utilizado o metamodelo de mapeamento proposto por D. Lopes [33]. O metamodelo de mapeamento proposto por D. Lopes foi adaptado para atender também o mapeamento que deve ser mantido no processo de *merging* de dois *database models*.

A Figura 4.5 demonstra o *mapping metamodel*. A classe em cor cinza foi acrescentada ao modelo para também manter o mapeamento resultante da aplicação do algoritmo de *database model merging*, o qual é apresentado na seção 4.6.

O elementos dos *database models* são armazenados no *mapping metamodel* nas classes *MetamodelHandler* e *ElementHandler*, permitindo assim, a manipulação destes

elementos para a tarefa de *model matching* e *model merging*. Todos os elementos do modelo fonte (*and-point source*) e alvo (*and-point target*) são registrados na classe *ElementHandler*, através do *and-point handlers*.

Os elementos que são correspondidos nos dois modelos são armazenados na classe *Correspondence*. As classes *Input* e *Output* registram o mapeamento dos elementos do modelo fonte e alvo, isto é, o elemento e_1 do modelo M_1 corresponde a qual elemento e_2 do modelo M_2.

Os elementos que sofreram *merge* são armazenados na classe *Merge*. Utilizando também, as classe *Input* e *Output* para registrar o mapeamento dos elementos do modelo fonte e alvo, isto é, o elemento e_1 do modelo M_1 que foi sobreposto ao elemento e_2 do modelo M_2.

O *mapping metamodel* é o metamodelo base para o *framework* que é proposto neste trabalho. Isto porque, o *framework* proposto é uma extensão das ferramentas *Mapping Tool for MDE (MT4MDE)* e *Semi-Automatic Matching Tool for MDE (SAMT4MDE)* desenvolvidas por D. Lopes [33] para gerar um mapeamento entre metamodelos e regras de definição de transformação em ATL [3].

4.3.3 Metamodelo para *database matching model*

O *database matching metamodel* é usado no processo de busca de elementos correspondentes entre os *database models*. Um modelo é instanciado conforme *database matching metamodel* registrando se entidades, atributos e relacionamentos são correspondentes. O *database matching metamodel* é composto pelos seguintes elementos:

- *Element*: é uma generalização para outros elementos do metamodelo, onde o atributo *name* identifica o nome do elemento e o atributo *validate* é um booleano que determina se elementos de dois *database models* são iguais, similares ou diferentes;
- *MatchDB*: contém o conjunto de elementos a serem correspondidos;
- *MEntity*: contém as entidades dos modelos a serem correspondidos. O atributo *eLeft* mantém a referência a entidade do *database model* da esquerda e o atributo *eRight* mantém a referência a entidade do *database model* da direita que são correspondidos;

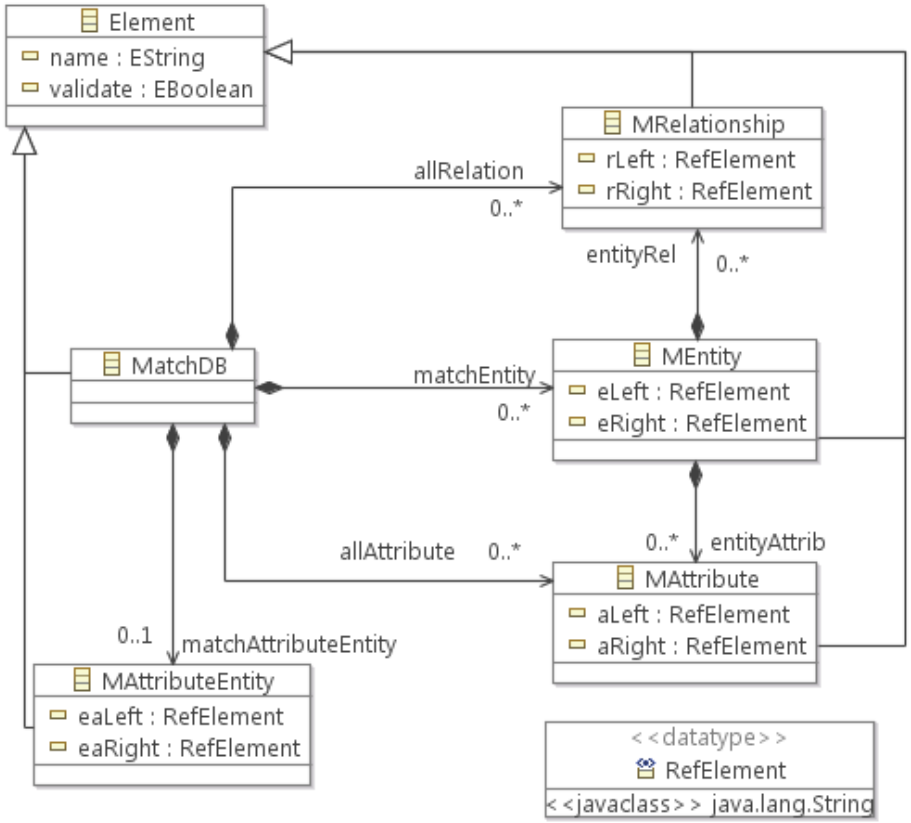


Figura 4.6: Metamodelo para *model matching*.

- *MAttribute*: contém os atributos dos modelos a serem correspondidos. O atributo *aLeft* mantém a referência ao atributo do *database model* da esquerda e o atributo *aRight* mantém a referência ao atributo do *database model* da direita que são correspondidos;

- *MRelationShip*: contém os relacionamentos dos modelos a serem correspondidos. O atributo *rLeft* mantém a referência ao relacionamento do *database model* da esquerda e o atributo *rRight* mantém a referência ao relacionamento do *database model* da direita que são correspondidos;

- *MAttributeEntity*: contém correspondência entre atributos e entidades. Muitas vezes em database model um atributo de uma entidade do esquema fonte corresponde a uma entidade do esquema alvo. Assim, o atributo *aeLeft* mantém a referência do atributo do *database model* da esquerda e o atributo *aeRight* mantém a referência a entidade do *database model* da direita que são correspondidos.

Modelos criados conforme *database matching metamodel* auxiliam o construção de mapeamentos entre *database models* fonte e alvo. Um mapeamento determina as relações entre elementos de dois modelos [31], isto é, o mapeamento diz que um elemento do modelo fonte corresponde a um ou mais elemento do esquema alvo. Assim, o *database matching metamodel* registra quais elementos são correspondentes para construir o *database matching model* que deve ser conforme *database matching metamodel*. Os elementos identificados como correspondentes no *database matching model* são inseridos no *mapping model* visto anteriormente.

4.3.4 Metamodelo para *database merging model*

Os modelos criados conforme *database merging metamodel* tem como objetivo representar a estrutura do *integrated database model*, especificando que entidades serão sobrepostas no processo de *merging*, e ainda, as entidades que não possuem correspondências e também devem está presente no *integrated database model*.

O *database matching metamodel* armazena apenas referências dos elementos dos *database models* fonte e alvo. Estas referências são utilizadas para localizar elementos no modelo do *database models* (fonte e alvo), que compõem o *integrated database model*. O *database merging metamodel* é composto pelas seguintes classes:

- *Element*: é uma generalização para outros elementos do metamodelo, onde o atributo *name* identifica o nome do elemento, o atributo *elemReference* que armazena a referencia ao elemento correspondido no *merging*, o atributo origem armazena *schema* de origem e o atributo *validate* indica se um elemento sofreu *merging*;

- *MergeS*: contém o nome do *merge* e os atributos *schemaLeft* armazena o caminho do esquema fonte e *schemaRigth* armazena o caminho do esquema alvo;

- *MeEntity*: contém as entidades do *database model*. O atributo *merge* indica se a entidade foi fundida e o atributo *generalized* indica se a entidade foi generalizada. O processo de generalização de entidade é visto na seção 4.6;

- *MeAttribute*: contém os atributos de entidade definida no *database model*. O atributo *merge* indica se o atributo foi fundido;

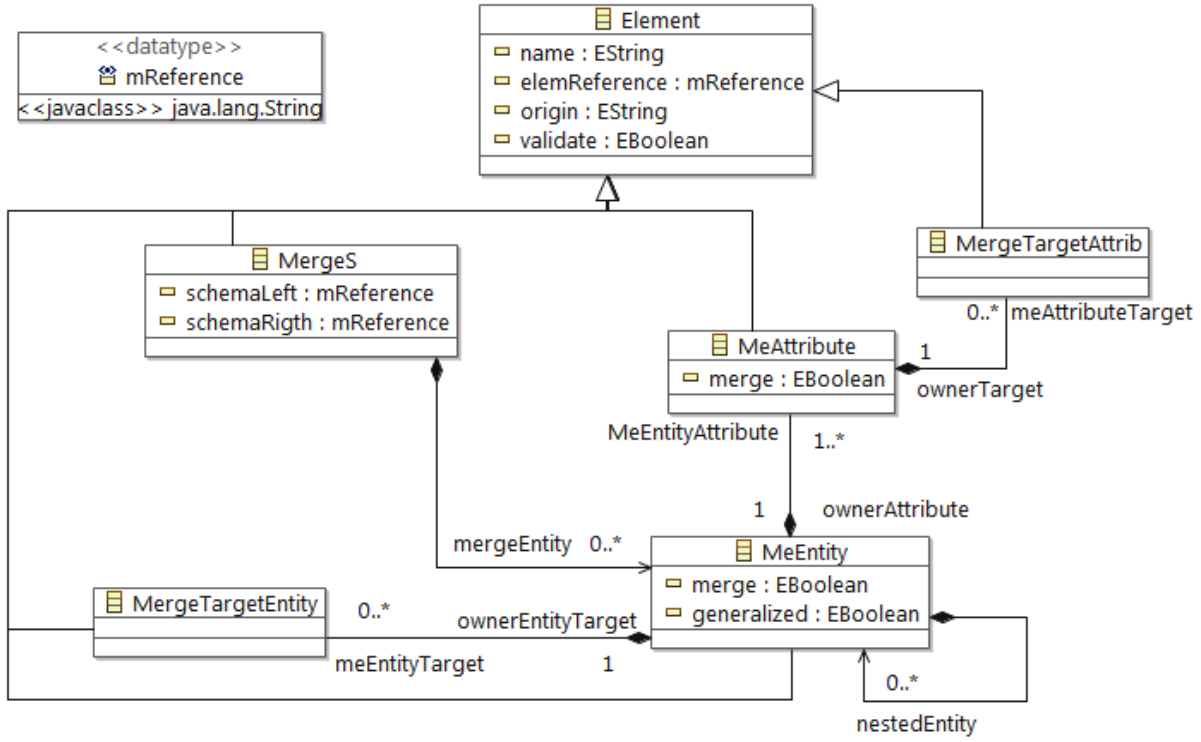


Figura 4.7: Metamodelo para *model merging*.

- *MergeTargetEntity*: registra a entidade correspondida no mapeamento, isto é, entidade do *database model* alvo que corresponde a entidade do *database model* fonte;
- *MergeTargetAttrib*: registra o atributo correspondido no mapeamento, isto é, atributo do *database model* alvo que corresponde ao atributo do *database model* fonte;

4.4 Metamodelo de dicionário de sinônimos de domínio

O *domain dictionary metamodel* tem o objetivo de armazenar lista de sinônimos de termos de domínios de *database models*. A lista de sinônimos é pesquisada para auxiliar o processo de indentificação de similaridade entre *strings*. O modelo instanciado conforme *domain dictionary metamodel* é utilizado no processo de *matching* para determinar se duas *strings* são similares.

O *domain dictionary metamodel* é criado pelo especialista de domínio através da interface de edição de modelos gerada pelo EMF. O *domain dictionary metamodel* é constituído das classes a seguir:

- *Element*: é uma generalização para outros elementos do metamodelo, onde o atributo *name* identifica o nome do elemento;
- *DomainSynonymous*: identifica o domínio dos sinônimos a serem armazenados;
- *Word*: armazena um termo raiz de um conjuntos de sinônimos de um domínio;
- *SynWord*: armazena sinônimos de um determinado termo raiz.

A Figura 4.8 ilustra o *domain dictionary metamodel* com suas classes e relacionamentos.

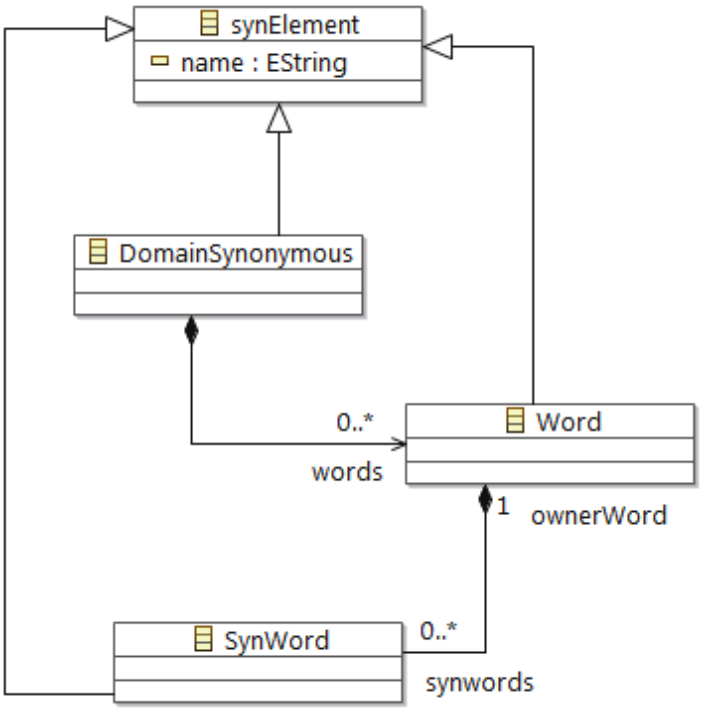


Figura 4.8: Metamodelo de dicionário de domínio.

4.5 Algoritmo para *database model matching*

Esta seção detalha o algoritmo que será aplicado para obter um modelo contendo elementos correspondidos entre dois *database models*. Para isso, um operador *Match* é proposto com a finalidade de implementar o algoritmo de *database model matching*.

Conforme visto na seção 2.2, onde foi descrita uma abordagem para *model matching*, o *database matching metamodel* foi criado para suportar tanto o *match* baseado

em esquema quanto o *match* baseado em instância. Assim, elementos serão avaliados levando em consideração a similaridade de *string*, bem como, a similaridade por valor atribuído a elementos dos *database models*.

O algoritmo de *database model matching* foi implementado nesta pesquisa através do operador *Match*. O operador *Match* retorna o mapeamento entre elementos de dois *database models*. Em [55] é definido mapeamento como um conjunto de elementos de correspondência que indica que determinados elementos do esquemas S_1 são mapeados para determinados elementos de um esquema S_2.

O operador *Match* foi definido baseado no trabalho de D. Lopes [33], que utiliza o operador para determinar correspondência entre metamodelos para gerar definição de transformação de modelo com *Atlas Transforation Language* - ATL [23]. O operador *Match* proposto nesta pesquisa foi adaptado para tratar correspondência de modelos ao invés de metamodelos como definido originalmente em [33]. O operador *Match*($M_1(S_1)/M_s, M_2(S_2)/M_s$) é responsável por retornar um mapeamento $C_{M_1 \to M_2} / M_c$ entre dois *database models*.

Dado um *database model* M_1 do sistema S_1, que está conforme *database model metamodel* M_s e um *database model* M_2 do sistema S_2, que está conforme *database model metamodel* M_s. O operador *Match* retorna o mapeamento $C_{M_1 \to M_2} / M_c$, que está conforme ao metamodelo M_c. O mapeamento $C_{M_1 \to M_2} / M_c$ corresponde a classe *MatchDB* do *database matching metamodel* apresentado na Figura 4.6.

O mapeamento $C_{M_1 \to M_2} / M_c$ pode ser definido como: $C_{M_1 \to M_2} / M_c \supseteq \{M_1 \cap M_2\}$, onde $\cap$ retorna elementos de M_1 e M_2 que são iguais ou similares. Elementos similares são elementos onde existe uma relação entre eles, mas essa relação não pode ser bem definida [6]. Por exemplo, um schema S_1 possui uma entidade *Paciente* com o atributo *name* para identificar o paciente. Enquanto que no schema S_2 também existe uma entidade *Paciente*, mas a identificação do paciente é composta de *primeiroNome* e *ultimoNome*. Assim, temos similaridade entre S_1.*Paciente.nome* $\cong$ (S_2.*primeiroNome*, S_2.*ultimoNome*).

Os conjuntos M_1, M_2 e $C_{M_1 \to M_2} / M_c$ podem ser definidos como:

- $M_1 = \{e_{1_i}, f_{1_j} \mid 0 < i \leq n \text{ and } 0 < j \leq m\}$, onde e_{1_i} são entidades, f_{1_j} são atributos. As variáveis n e m são a quantidade de entidades e atributos de M_1 respectivamente;

- $M_2 = \{e_{2_x}, f_{2_y} \mid 0 < x \leq n' \text{ and } 0 < y \leq m'\}$, onde e_{2_x} são entidades, f_{2_y} são atributos. As variáveis n' e m' são a quantidade de entidades e atributos de M_2 respectivamente;

- $C_{M_1 \to M_2} / M_c = \{c_1, c_2, ..., c_p\}$, que constitui o conjunto de elementos correspondentes de M_1 e M_2, onde $c_i = \{e_{1_i}, e_{2_x}\} \vee \{f_{1_j}, f_{2_y}\} \vee \{f_{1_j}, e_{2_x}\}$. Assim, c_i pode ser um *match* de entidade para entidades ou atributo para atributo ou então atributo para entidade.

Além da definição dos conjuntos M_1, M_2 e $C_{M_1 \to M_2} / M_c$ é necessário ainda, considerar alguns cenários que devem ser observados na busca por correspondência entre *database models*, como o grau de correspondência entre elementos dos esquemas. Com relação ao grau, a correspondência entre elementos dos modelos pode ter a seguinte classificação:

- ***Match* total**: todos os atributos do par de entidade (e_{1_i}, e_{2_x}) possuem correspondência;

- ***Match* parcial**: parte dos atributos do par de entidade (e_{1_i}, e_{2_x}) possuem correspondência.

O grau de correspondência entre elementos dos esquemas é uma medida importante para definir se entidades sofrerão *merging*, pois esta medida determina o quanto entidades são similares. O especialista de domínio deve configurar o valor limite para o grau de correspondência. Com relação aos cenários, a busca por correspondência pode considerar:

- **Cenário 1**: o par de entidades (e_{1_i}, e_{2_x}) correspondem em nome e atributos, podendo ser um *match* total ou parcial;

- **Cenário 2**: o par de entidades (e_{1_i}, e_{2_x}) correspondem apenas pelo nome das entidades, isto é, as entidades (e_{1_i}, e_{2_x}) possuem nomes similares, mas não possuem nenhum atributo correspondente. Neste cenário, as entidades não são similares;

- **Cenário 3**: o par de entidades (e_{1_i}, e_{2_x}) não possuem nomes correspondentes, mas todos seus atributos são similares. Neste cenário, as entidades são similares pela correspondência de seus atributos, podendo ser um *match* total ou parcial;

- **Cenário 4**: pode ter uma correspondência de atributo com entidade, isto é, pode haver a situação em que um ou mais atributos de uma entidade do esquema S_1 corresponda uma entidade do esquema S_2. A Figura 4.9 ilustra este cenário, onde **Pessoa.ocupacao** $\cong$ **Ocupacao**.

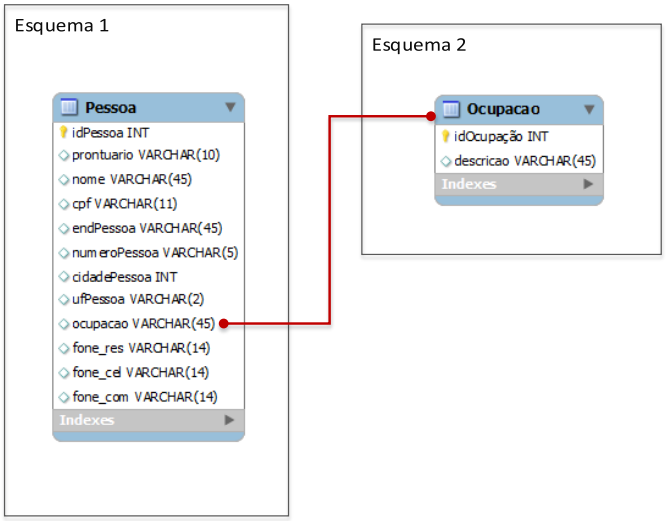


Figura 4.9: Exemplo de *matching* entre atributo e entidade.

Com os conjuntos M_1, M_2 e $C_{M_1 \to M_2} / M_c$ e com os cenários para busca de correspondência entre dois modelos definidos, o algoritmo de *database model matching* entre dois *database models* é apresentado e segue os seguintes passos:

1. Criar *MatchDB*: instanciar o *database matching metamodel* para iniciar uma correspondência;
2. Selecionar todas as entidades e_{1_i} de M_1 e e_{2_x} de M_2;
3. Para cada par $\{e_{1_i}, e_{2_x}\}$ aplicar a função *simString*$(e_{1_i}, e_{2_x}, < method >)$ que retorna se as entidades possuem nomes iguais ou similares;
   (a) Se retornar verdade, então,
      i. Para cada par $\{f_{1_i}, f_{2_x}\}$
         A. Verificar se atributos são similares (aplicar a função *simString*) calculando percentual de atributos iguais ou similares.
      ii. Caso percentual seja maior que um valor limite, incluir entidade em *MatchDB*;

(b) Retornando falso, verificar se entidades são iguais ou similares pelo seus atributos, calculando percentual de atributos iguais ou similares como no item (a) e incluindo em *MatchDB* quando for maior que valor limite;

(c) Entidade não sendo iguais ou similares nem pelo item (a) e nem pelo item (b) não são incluídas em *MatchDB*.

4. Comparar atributos com entidades. Para cada par $\{f_{1_j}, e_{2_x}\}$ aplicar a função *simAttributeToEntity*(f_{1_j}, e_{2_x}) que compara se um attribute de e_{1_i} é representado por uma entidade e_{2_x}. Retornando verdade, incluir f_{1_j} e e_{2_x} em *MatcDB*.

A função *simString*$(e_{1_i}, e_{2_x}, < method >)$ retorna o grau de similaridade entre duas strings. O valor retornado pode variar entre [0..1], onde o retorno sendo 1 as *strings* são consideradas iguais. Um valor limite deve se definido para que as *string* sejam consideradas similares. O parâmetro $< method >$ determina a métrica de similaridade de *string* aplicada. A função *simString*$(e_{1_i}, e_{2_x}, < method >)$ fornece a possibilidade conforme as seguintes métricas:

- *Levenshtein distance* [45]: considera o número mínimo de inserções, substituições e exclusões para tornar duas *string* iguais. Assim, quanto menor o número de operações para tornar uma string em outra, mais provável que elas sejam iguais;
- *N-Gram distance* [64]: leva em consideração a contagem de número de ocorrência de *q-grams* (partes de uma string) entre duas strings. Quanto maior a quantidade de *q-grams* comuns, mais provável que as *string* sejam iguais;
- *Jaro Winkler* [13]: considera o número de caracteres comuns e suas posições para determinar a igualdade de duas *string*.

Foi utilizada a *Application Programming Interface (API)* Java Apache Lucene, disponibilizada pela *Apache Software Foundation* [2], com a implementação das métricas de similaridade de *string*. A função *simString* usa também um dicionário de sinônimos para determinar similaridade de *string*. Entre os metamodelos propostos neste trabalho, está o metamodelo de dicionário de domínio. Este metamodelo tem a finalidade de criar um modelo, conforme metamodelo dicionário de domínio (veja seção 4.4), com um dicionário específico de um domínio.

4.6 Algoritmo para *database model merging*

Seguindo a metodologia de integração de base de dados, a próxima etapa após o *matching* entre os elementos dos *database model* é o *merging* dos *database models*. A operação de *merging* tem como finalidade combinar dois modelos em um único modelo [43]. Portanto, é proposto o operador *Merge* que tem como entrada dois *database models* e fornece como saída um *database merging model*, que está conforme *database merging metamodel* (veja Figura 4.7), com uma visão unificada dos *database models* recebidos.

O operador *Merge* foi construído observando os requisitos de preservação de elementos, preservação de igualdade, preservação de relacionamento e preservação de similaridade que foram propostos em [52] e descritos na seção 2.3.

Antes de o algoritmo de *database model merging* ser detalhado, para um melhor entendimento, é necessário fornecer a fundamentação proposta para o operador *Merge*. O operador $Merge(M_1(S_1)/M_s, M_2(S_2)/M_s, C_{M_1 \to M_2}/M_c)$ é definido como a seguir:

- Os elementos $M_1(S_1)/M_s$, $M_2(S_2)/M_s$ e $C_{M_1 \to M_2}/M_c$ foram definidos na seção 4.5;
- O $Merge(M_1(S_1), M_2(S_2), C_{M_1 \to M_2}) = Me_{\left[\left(M_1 - C_{M_1 \to M_2}\right) \cup \left(M_1 - C_{M_1 \to M_2}\right) \cup C_{M_1 \to M_2}\right]} / M_{M_e}$, onde:
  - Me/M_{M_e}, M_e é o modelo do *merge* entre M_1, M_2 e $C_{M_1 \to M_2}$ criado conforme metamodelo M_{M_e};
  - $M_1 - C_{M_1 \to M_2} = \{m_{1_i} | m_{1_i} \in M_1 \wedge m_{1_i} \notin C_{M_1 \to M_2}\}$;
  - $M_2 - C_{M_1 \to M_2} = \left\{m_{2_j} | m_{2_j} \in M_2 \wedge m_{2_j} \notin C_{M_1 \to M_2}\right\}$.

Com base na fundamentação apresentada, o operador ***Merge*** retorna um modelo que contém os elementos dos *database model*s M_1 e M_2 que não são sobrepostos, isto é, não sofreram *matching*, além dos elementos que estão contidos no *mapping model* de M_1 e M_2, que devem ser representados por um único elemento no *database model merging*.

Baseado em [33] também foi definido uma proposta de fundamentação da função de transformação dos *database model* (M_1 e M_2), do modelo de mapeamento ($C_{M_1 \to M_2}$) e do modelo de *merge* (M_e). A função de transformação $Transf(M_1, M_2,$

$C_{M_1 \to M_2}$, M_e)$\to$ $M_i(S_1, S2)/M_I$ retorna um modelo do *database* integrado(M_i), que está conforme *integrated database model metamodel* (M_I), de dois *database models*.

O modelo integrado (M_i) construído pela função de transformação $Transf(M_1, M_2, C_{M_1 \to M_2}, M_e)$ servirá de modelo base para a geração do *script* SQL com a estrutura da base de dados integrada a ser implantada em um SGBD.

Como visto na fundamentação do operador *Merge*, um dos elementos de entrada do *Merge* é um modelo com o mapeamento de elementos correspondentes entres os dois *database models*. O modelo de mapeamento é importante nesta etapa, pois o grau de correspondência entre entidades do *database model* define se o *merge* entre entidade fonte e alvo será realizado.

Uma classificação de *matching* foi exposta na seção 4.5, onde o processo de *matching* pode retornar um *match total* ou *parcial* entre entidades de dois *database models*. O algoritmo de *database model matching* proposto trata esta classificação de *matching* e também o que denominamos de *no match*, isto é, entidades não duplicadas nos dois *database models*. A seguir é descrita a proposta de tratamento de cada uma destas correspondências:

- No caso de entidades apresentarem um *no match*, para cada entidade sem correspondência deve ser criada uma entidade correspondente no *database model merging*;
- No caso de duas entidades apresentarem um *match total*, uma única entidade deve ser criada no *database model merging*, representando assim, a fusão da entidade do *database model* fonte com a entidade do *database model* alvo. Esta fusão deve observar que todos os relacionamentos das as entidades fonte e alvo devem ser preservados;
- O caso mais complexo para ser tratado no *merging* de *database models* é quando se tem *match parcial* entre duas entidades. Isto porque, quando surge esta situação sabe-se que as entidades possuem alguma correspondência, porem necessita-se de uma medida que determine o seu grau. No algoritmo de *database model merging* o grau de correspondência entre entidades será determinante para a decisão de *merge* de entidades. Assim, para que as entidades sofram *merge* os seguintes cenários devem ser avaliados:

- **Cenário 1 (*high match*)**: quando o grau de correspondência for igual ou maior a um valor limite, deve ser criada uma entidade no *database model merging* representando a fusão entre as entidades. O valor limite neste caso representa um alto grau de correspondência, onde poucos atributos das duas entidades não foram correspondidos. Está solução traz a necessidade da criação de um atributo identificador da origem de instância de atributos. Isto é, qual entidade é fonte do registro que irá popular a tabela que representa as entidades que sofreram *merge*;
- **Cenário 2 (*low match*)**: quando o grau de correspondência for menor ou igual a um valor limite, entidades distintas devem ser criadas no *database model merging*. O valor limite para este caso não é o mesmo do item anterior. O valor limite neste caso representa um grau insignificante de correspondência, tão baixo que inviabiliza o *merge* das entidades;
- **Cenário 3 (*middle match*)**: neste cenário a quantidade de elementos correspondentes entre duas entidades não chega a ser tão baixo para se enquadrar no cenário 2 (dois), e nem tão alto para se enquadrar no cenário 1 (um). Assim, a solução proposta para este cenário é o que denominamos de *generalização de correspondência*. Isto é, as entidades fonte e alvo são redefinidas e uma nova entidade é criada, ficando suas composições como segue(veja Figura 4.10):
  - O nome da nova entidade é composto pelos os nomes da entidade fonte e alvo, além de ser acrescentado um atributo para identificar a qual entidade generalizada suas instâncias pertencem;
  - Os atributos da nova entidade, serão os atributos correspondidos entre entidade fonte e alvo. Sendo escolhidos os atributos da entidade fonte para compor a nova entidade;
  - A entidades fonte e alvo são redefinidas ficando apenas com os atributos não correspondidos, tendo como chave primária a chave primária da nova entidade criada.

Além dos cenários descritos anteriormente, outros aspectos são tratados pelo algoritmo de *database model merging* como:

- Conflitos de tipos de atributos, como por exemplo, um elemento fonte é do tipos caracter e o elemento alvo é do tipo *string*. Neste caso, o elemento sobreposto será definido como *string*. Da mesma forma, acontecerá com conflitos de tipo inteiro e real, onde o elemento sobreposto será tipo real;

- Todos os relacionamentos das entidades fonte e alvo devem ser preservados quando for aplicado a *generalização de correspondência*.

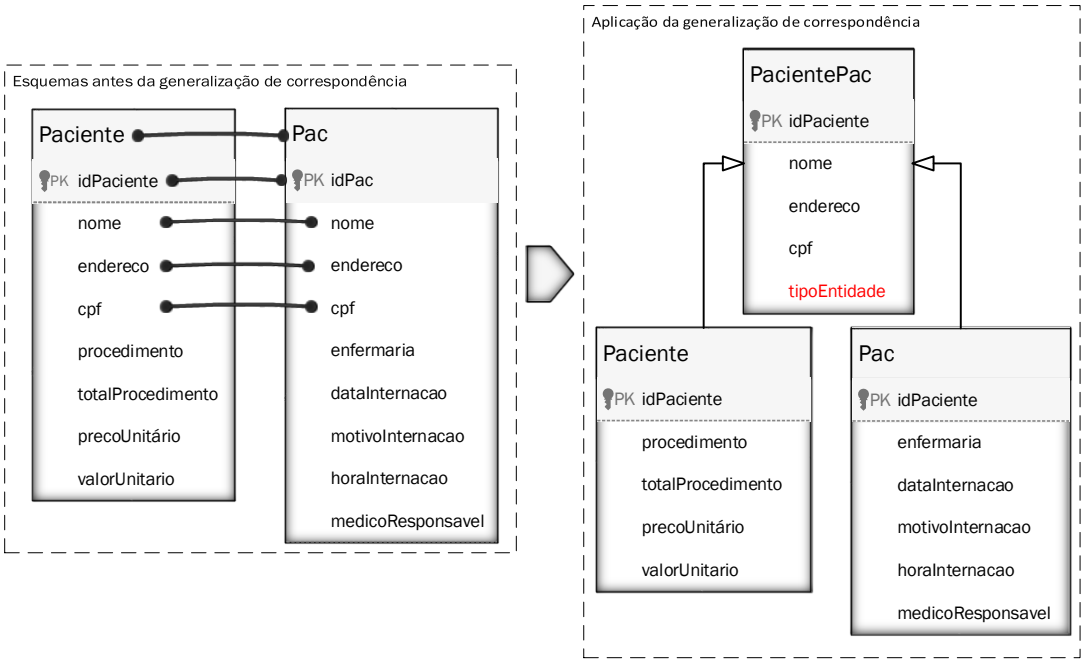


Figura 4.10: Exemplo da aplicação da generalização de correspondência entre dois esquemas.

Com a fundamentação teórica e os cenários a serem tratados pelo operador *Merge* definidos, o algoritmo de *database model merging* que instancia o *database model merging* conforme *database merging metamodel* (veja Figura 4.7) será detalhado como segue:

1. **Criar *MergeS***: cria o conjunto de entidades que comporão o *database model merging*;

2. **Selecionar entidades *no match***: criar um conjunto E de elementos a_{1_i} de M_1 e a_{2_j} de M_2 que não estão presentes no mapeamento $C_{M_1 \rightarrow M_2}$;

3. **Para cada** $\{a_{1_i}/M_1, a_{2_j}/M_2\} \in E$:

(a) Incluir uma nova entidade *MEntity* no *MergeS* com seus atributos e relacionamento idênticos a entidade de origem.

4. **Selecionar entidades classificadas como *match total***: criar um conjunto T com entidades a_{1_i}/M_1 que estão presentes no mapeamento $C_{M_1 \to M_2}$. O *database model merging* tem sua construção baseada nos elementos existentes no *database model* M_1, isto é, no modelo fonte;

5. **Para cada** $a_{1_i}/M_1 \in T$:

   (a) Incluir uma nova entidade *MeEntity* no *MergeS* com seus atributos e relacionamento idênticos a entidade de origem;

   (b) Criar uma ou mais classes *MergeTargetEntity*. As classes *MergeTargetEntity* representam entidades a_{2_j}/M_2 que correspondem a a_{1_i}/M_1. Assim, *MergeTargetEntity* tem a função de registrar o mapeamento do *Merge* entre a_{1_i}/M_1 e a_{2_j}/M_2;

   (c) Criar uma ou mais classes *MergeTargetAttrib*. As classes *MergeTargetAttrib* representam *MeAttribute* f_{2_j}/a_{2_j} que correspondem a f_{1_i}/a_{1_i}. Assim, *MergeTargetAttrib* tem a função de registrar o mapeamento do *Merge* entre f_{1_i}/M_1 e f_{2_j}/M_2;

   (d) Criar uma ou mais classes *MergeTargetRel*. As classes *MergeTargetRel* representam *Relationship* r_{2_j}/a_{2_j} que correspondem a r_{1_i}/a_{1_i}. Assim, *MergeTargetRel* tem a função de registrar o mapeamento do *Merge* entre r_{1_i}/M_1 e r_{2_j}/M_2.

6. **Selecionar entidades classificadas como *match parcial***: criar um conjunto Υ com entidades a_{1_i}/M_1 que estão presentes no mapeamento $C_{M_1 \to M_2}$, mas não possuem todos seus atributos correspondidos;

7. **Para cada** $a_{1_i}/M_1 \in \Upsilon$:

   (a) Se grau de similaridade for *low match*, então

      i. Incluir uma nova entidade *MeEntity* no *MergeS* para a_{1_i} e para a_{2_j} com seus atributos e relacionamento idênticos as entidades de origem;

   (b) Se o grau de similaridade for *high match*, então

i. Incluir uma nova entidade *MeEntity* no *MergeS* com seus atributos e relacionamento idênticos a entidade de origem. Incluir também, um *MeAttribute* com *name* = *"typeEntity"* para identificação da origem de tuplas;

ii. Executar as mesmas ações dos ítens 5.(a), 5.(b), 5.(c) e 5.(d);

(c) Se o grau de similaridade for *average match*, então (aplicação da generalização de correspondência)

i. Criar uma nova entidade *MeEntity* com os atributos *MeAttribute* f_{1_i}/a_{1_i} que correspondem aos atributos de f_{2_j}/a_{2_j}. Incluir também, um *MeAttribute* com *name* = *"typeEntity"* para identificação da origem de tuplas;

ii. Incluir novas entidades *MeEntity* no *MergeS* para a_{1_i} e para a_{2_j} com somente os atributos f_{1_i}/a_{1_i} e f_{2_j}/a_{2_j} que não são correspondidos. *MeEntity* que representará a_{1_i} e a_{2_j} devem ter a mesma chave primária da entidade *MeEntity* criada no ítem anterior (i);

iii. Executar as mesmas ações dos ítens 5.(a), 5.(b), 5.(c) e 5.(d).

Após a construção do *database merging model*, um mapeamento também é criado no *mapping model* que é conforme *mapping metamodel* (veja Figura 4.5). Elementos da classe *Merge* são criados no *mapping model* para manter informações de quais elementos dos *database models* fonte e alvo foram sobrepostos, além de elementos não correspondentes entre os *database models* fonte e alvo. Elementos da classe *Merge* do *mapping model* são recuperados para instanciar o modelo integrado de base de dados que será transformado em um SQL*script*.

4.7 Síntese

Este capítulo apresentou a abordagem para o desenvolvimento de um *framework* para integrar *database model* no contexto da MDE. Para atender a abordagem foram apresentados metamodelos, algoritmos e uma metodologia.

Assim, o capítulo apresentou uma visão geral do *framework* e logo em seguida, a metodologia necessária para obter um esquema de base de dados integrada de dois *database model* de entrada foi apresentado. O capítulo segue com a apresentação

do *database metamodel* (Figura 4.4) que deve ser usado pelo projetista de base de dados para definir os esquemas de base de dados que serão integrados.

Também foi visto, o *mapping metamodel* que foi proposto por D. Lopes [33]. O *mapping metamodel* foi adaptado e estendido para mapear *database models*. Além do *mapping metamodel*, foram definidos o *database merging metamodel*, o *database matching metamodel* e os algoritmos de *database model matching* e de *database model merging*.

5 IMPLEMETAÇÃO DO PROTÓTIPO SID4MDE: *Framework* para suportar *Database Model Merging*

Neste capítulo, um protótipo para implementar o *framework* para suportar *Database Model Merging* é descrito. O *framework* foi implementado com o EMF *(Eclipse Modeling Framework)*. Além do EMF, ATL foi utilizada para construção de definições de transformações de *database model* para *script* SQL. Os metamodelos para suportar o *database model merging* foram definidos na linguagem de metamodelagem Ecore do EMF. Utilizou-se a ferramenta *GenModel* do EMF, para gerar os *plug-ins* em *Eclipse* para criação e edição dos modelos utilizados no trabalho.

5.1 Protótipo da ferramenta para *Database Model* Merging

O protótipo tem sua construção baseada nas ferramentas MT4MDE (*Mapping Tool for Model Driven Engineereing*) e SAMT4MDE (*Semi-Automatic Matching Tool for MDE*) que foram desenvolvidas por D. Lopes [33]. As ferramentas MT4MDE e SAMT4MDE tem como objetivo criar uma correspondência (*matching*) entre metamodelos Ecore e gerar regras de transformação em *ATL*. A implementação do protótipo seguiu a metodologia apresentada na seção 4.2 sendo descrita a seguir.

5.1.1 Arquitetura do *Framework* para suportar *Database Model Merging*

Para que as ferramentas MT4MDE e SAMT4MDE suportassem o *merging* de *database model*, foram necessárias modificações em suas arquiteturas. Assim, o algoritmo de correspondência de elementos, o metamodelo de mapeamento de correspon-

dência e classes de manipulação de modelos foram adaptadas. Além disso, um metamodelo para o *merging* de *database model*, juntamente com os metamodelos *database model integrated*, algoritmo de *database merging* e algoritmo para integração de *database model* foram propostos.

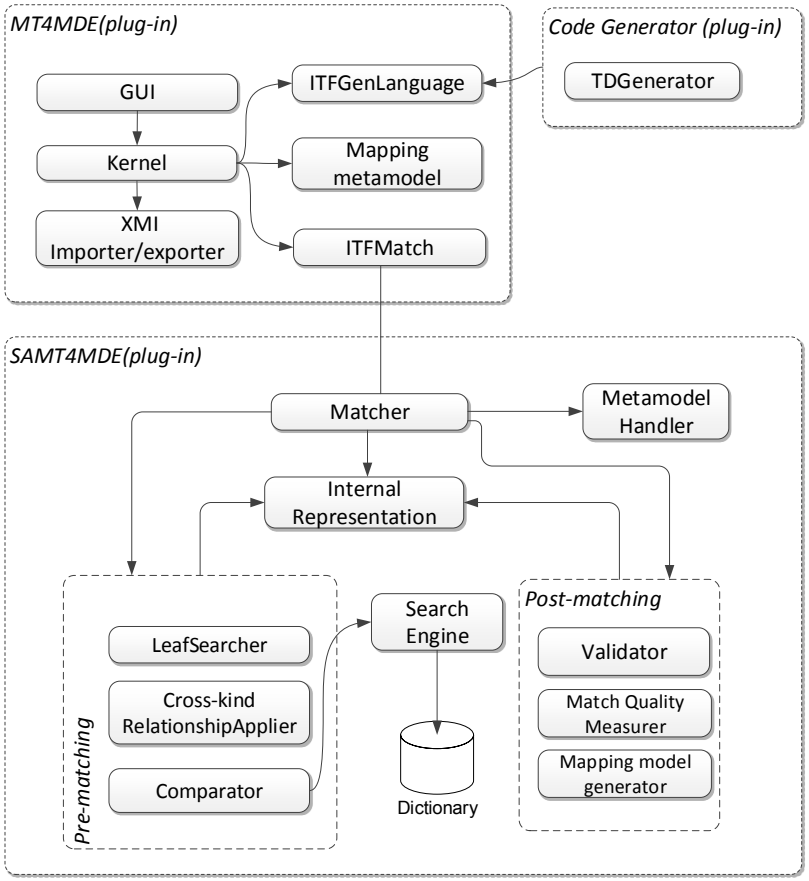


Figura 5.1: Arquitetura das ferramentas MT4MDE e SAMT4MDE [33]

A Figura 5.1 mostra a arquitetura original das ferramentas MT4MDE e SAMT4MDE. Primeiramente, a arquitetura da ferramenta MT4MDE é descrita, depois a arquitetura da ferramenta SAMT4MDE é apresentada .

MT4MDE (*Mapping Tool for Model Driven Engineereing*)

MT4MDE é composto pelos blocos funcionais descritos a seguir:

- GUI: uma interface gráfica para interação do especialista de domínio com a ferramenta;
- *Kernel*: núcleo de funcionalidades básicas da ferramenta MT4MDE;

- *Importer/Exporter*: módulo que traduz metamodelo no formato XMI para o formato Ecore;
- *Mapping Metamodel*: correspondências entre metamodelos são armazenas em modelos conforme *mapping metamodel*;
- *ITFMatch*: interface que deve ser implementada com o algoritmo de *Matcher*. Deste modo, a ferramenta sua extensão com a implementação de diferentes algoritmos de *Matcher*;
- *ITFGenLanguage*: interface que deve ser implementado com o algoritmo para geração de definições de transformações entre metamodelos;

A ferramenta MT4MDE tem como objetivo criar e editar correspondências entre elementos de metamodelos e a partir delas gerar definições de transformações em ATL. Para a ferramenta proposta neste trabalho, o objetivo final é gerar SQL *script* com a estrutura de um *database* integrado, isto é, a ferramenta recebe como entrada dois *database models* e gera como saída um SQL *script* com uma estrutura unificada dos dois modelos de entrada.

SAMT4MDE (*Semi-Automatic Matching Tool for MDE*)

SAMT4MDE é composto pelos blocos funcionais descritos a seguir:

- *Matcher*: implementação da interface *ITFMatch* com o algoritmo que coordena a busca de correspondências entre elementos de metamodelos;
- *Internal Representation*: representação mais adequada para identificar correspondências entre elementos dos metamodelos. Em [33], um metamodelo para representar *matching* entre elementos de dois metamodelos é apresentado.. Neste trabalho, para esta mesma função foi definido o *database matching metamodel*(veja Figura 4.6);
- *Metamodel Handler*: tem a função de permitir a navegação entre elementos dos metamodelos. Os metamodelos fonte e alvo são importados para o *Metamodel Handler* disponibilizando os elementos para busca de correspondências;
- *Search Engine*: realiza a busca de sinônimos para determinar similaridade entre elementos do modelo;

- *Dictionary*: base de dados de sinônimos utilizados pelo *search engine*;
- *Validator*: interface com o especialista de domínio para validação das correspondências encontradas de forma semiautomática;
- *Mapping model generator*: constrói um modelo de correspondências, conforme *mapping metamodel*, a partir da validação das correspondências encontradas pelo especialista de domínio;
- *Match Quality Measurer*: avalia as correspondências encontradas determinando uma medida de qualidade.

SAMT4MDE tem como objetivo complementar a ferramenta MT4MDE, acrescentando a busca de correspondências entre elementos de metamodelos de forma semiautomática [35]. A busca de correspondência é definida como semiautomática porque exige a participação humana para confirmação das correspondências encontradas. Da mesma forma, a ferramenta proposta neste trabalho busca correspondências de forma semiautomática, porém, utilizando um algoritmo de *matching* direcionado a correspondência de modelos de *database*.

Portanto, a adequação e extensão das ferramentas MT4MDE e SAMT4MDE incluem alterações nos modelos utilizados como entrada e alterações no algoritmo de *matching* definidos originalmente. Bem como, a inclusão de novos metamodelos e do algoritmo para geração de um *integrated database model*.

5.1.2 Arquiteturas SAMT4MDE e MT4MDE adaptadas para suportar *Database Model Merging*

As ferramentas SAMT4MDE e MT4MDE foram construídas para trabalhar com correspondência de metamodelos. Contudo, para serem aplicadas a correspondências de *database model* foram necessárias alterações para permitir a correspondência entre modelos. A Figura 5.2 mostra elementos que foram mantidos e as adaptações propostas nas arquiteturas das ferramentas para atender os requisitos de *database model merging*. A seguir é descrita as adequações feitas na arquitetura original das ferramentas SAMT4MDE e MT4MDE:

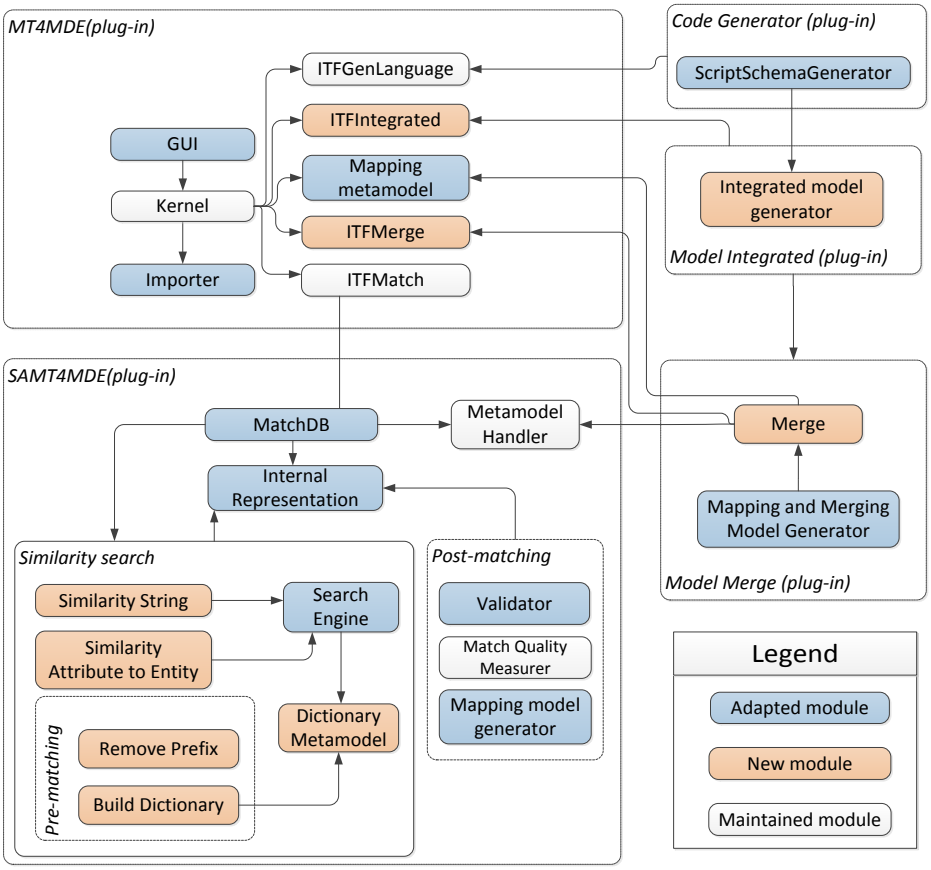


Figura 5.2: Arquitetura das Ferramentas MT4MDE e SAMT4MDE adaptadas para *Database Model Merging*

- *Importer*: diferente da proposta original, este módulo é responsável pela importação de *database schemas* de fontes como SQL *script*, ou ainda, diretamente de metadados do *DBMS*;

- *Mapping metamodel*: o *mapping metamodel*(veja Fig. 4.5) foi alterado para suportar também o registro de mapeamento criados pelo algoritmo de *merging*;

- *MatchDB*: um metamodelo de *database model matching* foi proposto para substituir o metamodelo de *matching* original. Como visto na seção 4.3 este metamodelo define quais elementos dos modelos fonte e alvo são similares. O módulo *MatchDB*, que implementa a interface *ITFMatch*, utiliza o metamodelo *database model matching* para a criação de um modelo que registra os elementos correspondentes entre dois *database models*;

- *ITFMerge*: interface que deve ser implementada para criação do algoritmo de *merging*. Deste modo, a ferramenta permite que diferentes algoritmos de *database model merging* sejam utilizados;

- *Merge*: o *database merging metamodel* (veja Fig. 4.7), que não existia na arquitetura original, foi proposto . O módulo *Merge* implementa a interface *ITFMerge*, que constrói um modelo conforme *database merging metamodel*. O *database merging model* é a *Internal Representation* utilizada no processo de busca de similaridade entre elementos de dois *database models*;

- *Mapping and Merging Model Generator*: responsável por construir um *mapping model* conforme *mapping metamodel*. Este modelo é construído a partir da validação, pelo especialista de domínio, do *schema merging model*;

- *Integrated Model Generator*: o módulo *Integrated Model Generator* constrói um *database model* , contendo toda estrutura necessária para criação do SQL *script* . Sua construção é baseada no *mapping model* criado pelo módulo *mapping e merging model generator*;

- *Script Schema Generator*: implementa a interface *ITFGenLanguage* gerando o SQL *script* do *database* integrado;

- *Similarity String*: função que busca similaridade entre nomes de elementos de *database model*. A similaridade é estabelecida através de métricas de similaridade de *string* (Jairo Winkler, NGram Distance e Levenshtein Distance). A função também faz consulta a um modelo de dicionário de sinônimos que está conforme *dictionary metamodel* para definir similaridade entre elementos de *database model*;

- *Similarity Attribute to Entity*: função que busca identificar se existem atributos de um *database model* fonte que é similar a entidades de outro *database model* alvo;

- *Remove prefix*: função que identifica a existência de prefixos nos atributos de entidades de *database model* fonte e alvo, isto é, faz um limpeza nos elementos antes de passarem pelo processo de *matching*;

- *Build Dictionary*: responsável pela construção do *dictionary domain model* conforme *dictionary domain metamodel*. Este dicionário de domínio é alimentado pelo

especialista no domínio para auxiliar o processo de busca de similaridade entre nomes de elementos dos modelos.

A Figura 5.3 mostra o diagrama de classe do *framework* com suas principais classes. A classe *MappingTreeViewer* cria uma interface que fornece acesso aos recursos de execução do *matching*, *merging* e integração de modelos, além de exibir os modelos fonte e alvo e oferecer mecanismos de edição do mapeamento entre eles. As classes *MatchAction*, *MergeAction*, *IntegratedModelAction* interagem com classes que implementam os metamodelos que dão suporte ao *framework*. A classe *Match* implementa a interface *ITFMatchEngine*, sendo responsável por instanciar o *mapping model* com elementos correspondentes entres os modelos repassados ao método *init (Model Ma, Model Mb)*. Da mesma forma, a classe *Merge* instancia o *database merging model* através da implementação da interface *ITFMergeEngine* e a classe *IntegratedModel* instancia o *integrated database model* através da implementação da interface *ITFIntegratedModelEngine*.

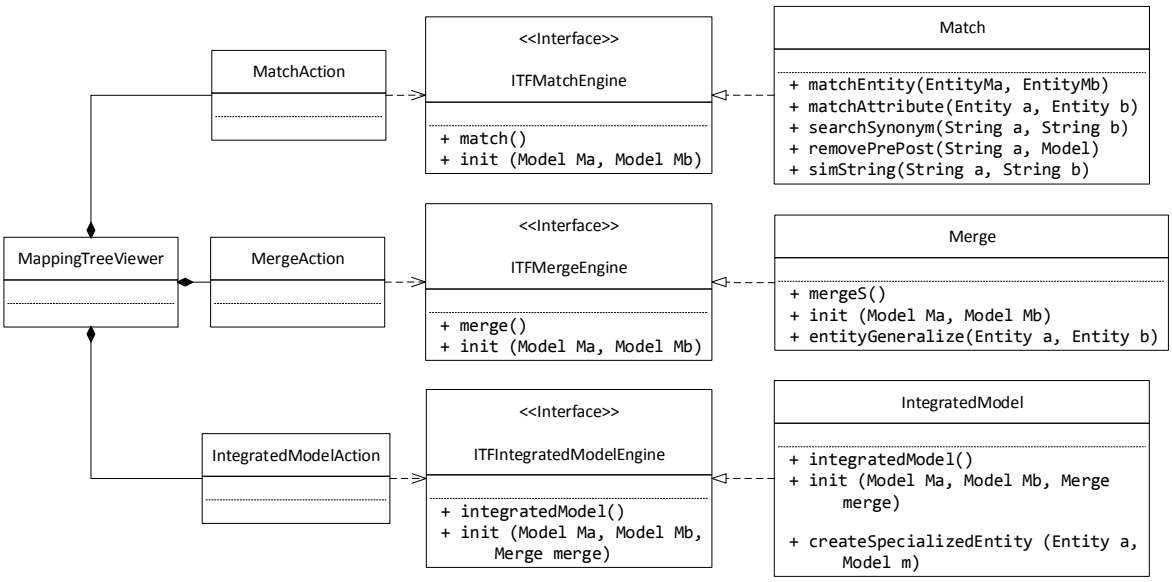


Figura 5.3: Principais classes do *framework* SID4MDE.

Com estas modificações, as ferramentas MT4MDE e SAMT4MDE se ajustam a tarefa *match* e *merge*, isto é, dado dois *database models* como entrada, a ferramenta retorna como saída um mapeamento com elementos correspondentes entre os modelos, retorna ainda, um modelo unificado dos modelos de entrada (*database integrated model*). A seção seguinte demonstra a implementação do protótipo do *framework* seguindo a metodologia proposta na seção 4.2.

5.1.3 Implementação do protótipo

A extensão das ferramentas MT4MDE e SAMT4MDE foi implementada com o projeto *Eclipse Modeling Framework*-EMF [19]. Como visto na seção 2.1.2, o EMF fornece um *framework* que possibilita a criação de *software* seguindo a abordagem MDE. Desse modo, os metamodelos propostos para suportar a ferramenta foram criados conforme o metamodelo Ecore fornecido pelo EMF. Bem como, para o desenvolvimento dos algoritmos de *matching* e *merging* foram usados classes Java fornecidas pelo EMF para instanciar e manipular modelos Ecore. A única transformação de modelo que não utilizou EMF para a definições de transformações foi a utilizada na transformação do *database integrated model* para SQL *script model*, onde, foi utilizado a ATL [3]. A seguir a implementação do protótipo é detalhada obedecendo o fluxo proposto na metodologia para integração de *database* apresentada neste trabalho.

Obter *database model*

Conforme a metodologia proposta na seção 4.2, o processo de *database model merging* começa com a obtenção dos esquemas dos *databases* a serem integrados. Isto pode ser feito através da importação dos esquemas dos *databases* ou então o especialista de domínio pode construir o *database model* diretamente na ferramenta, ou ainda, recuperar um *database model* do repositório de modelos criados ou importados em um momento anterior.

Importar o *database model*

Esta etapa tem o objetivo de facilitar a criação do *database model*, através de sua importação diretamente de um DBMS. O especialista deve indicar qual o nome do *database*, bem como, seu usuário e senha de acesso ao banco de dados. O produto desta etapa é ter um *database model* fonte e alvo que serão usados como modelos de entrada no processo de *matching*, *merging* e integração dos modelos.

Criar o *database model* diretamente na ferramenta

O EMF oferece um ambiente para edição de modelos conforme a linguagem de modelagem Ecore, denominado de *EMF Generator Model*. A partir do metamodelo de esquema de base de dados (**schemaDataBase.ecore**), um modelo gerador é criado com classes que permitem instanciar modelos conforme o metamodelo **schemaData-**

Base.ecore. A Figura 5.4 ilustra o modelo **schemaDataBase.genmodel** criado a partir do **schemaDataBase.ecore**.

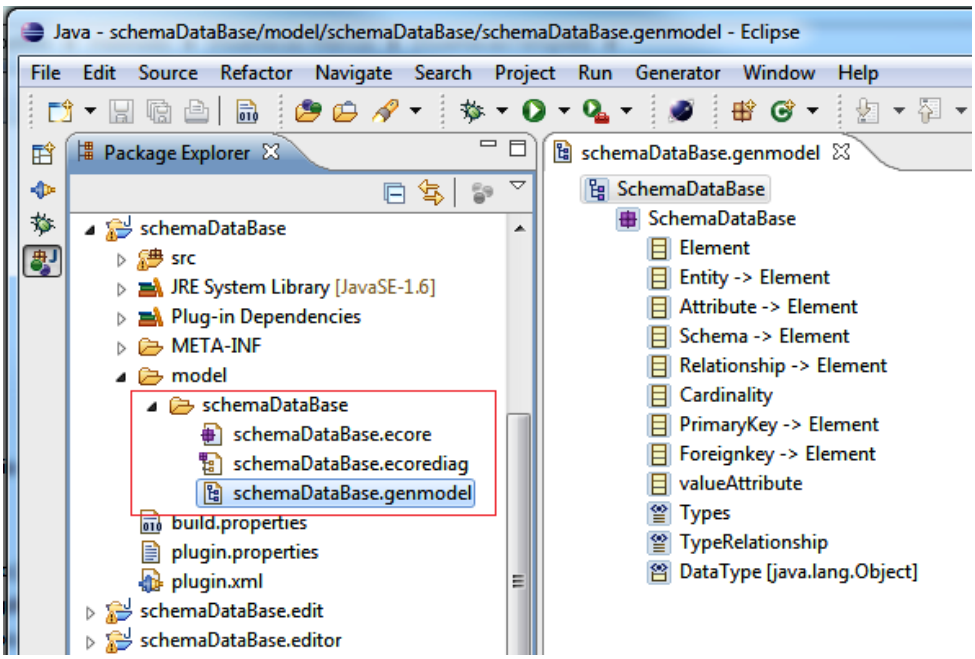
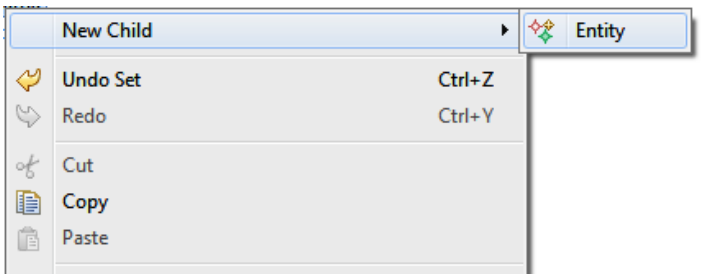


Figura 5.4: Modelo **schemaDataBase.genmodel** construído a partir **schemaDataBase.ecore**.

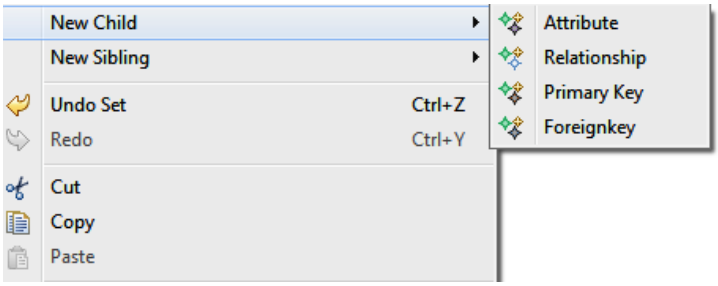
A instância criada pelo *EMF Generator Model* permite ao especialista de domínio criar um *database model* definindo informações como nome esquema (*Schema*), entidades (*Entity*), atributos (*Attribute*), chave primária (*PrimaryKey*), chave estrangeira (*ForeignKey*). A Figura 5.5(a) ilustra o uso do ambiente gerado pelo *EMF Generator Model* para definir uma nova entidade. A Figura 5.5(b) ilustra como o especialista pode definir atributos, chave primária e chave estrangeira para uma entidade do *database model*. Os *database model* criados são armazenados em um repositório de modelos possibilitando o seu uso em novos processos de integração, como definido na metodologia para integrar *database model*.

Gerar correspondência entre *database models* (gerar um *mapping model*)

Os *database models* criados ou importados na etapa anterior, são agora utilizados pelo algoritmo de *matching*, apresentado na seção 4.5, para a construção de um *mapping model* (veja Fig. 4.5) com o mapeamento de elementos correspondentes entre modelos fonte e alvo. Um mapeamento demonstra o relacionamento entre elementos de dois modelos [7]. O relacionamento pode ser de igualdade ou similaridade.



(a) Criar um nova entidade no *database model*



(b) Criar atributos, chave primária e estrangeira do *database model*

Figura 5.5: Ambiente gerado pelo *EMF Genertor Model* para manipulação de modelos.

A Figura 5.6 mostra a interface do protótipo para execução da correspondência entre *database models*. O especialista deve escolher o algoritmo de *database model matching* que será aplicado na tarefa de correspondência, bem como, qual a métrica de similaridade de *string* que deverá ser usada pela função *simString*. A função *simString* determina se nomes de elementos dos *database models* são iguais ou similares.

A Listagem 5.1 mostra a implementação da função *simString*. Nas linhas 148 e 147 são retirados os caracteres traço e *underline* das strings recebidas pelos parâmetros *a* e *b*. Primeiro, é feita um consulta ao dicionário de sinônimos (linha 152), caso as *strings* não sejam sinônimas, então, é realizada a busca de similaridade indicada pelo parâmetro *method* (linha 156).

Listagem 5.1: Função para busca de similaridade de *string*.

```
147 private double simString(String a, String b, int method) {
148    String nameA = a.replaceAll("[_-]","").toLowerCase();
149    String nameB = b.replaceAll("[_-]","").toLowerCase();
150    double max=0; double vsimJairo = 0; double vsimLeven = 0;
151    double vsimGran  = 0;
152    if (searchSynonym(nameA, nameB) == 0){
153       max= 1;
154       return max;
```

```
155     }
156     switch (method) {
157     case 1:
158        JaroWinklerDistance sim = new JaroWinklerDistance();
159        vsimJairo = sim.getDistance(nameA, nameB);
160        max = vsimJairo;
161        break;
162     case 2:
163        NGramDistance ngram = new NGramDistance();
164        vsimGran  = ngram.getDistance(nameA, nameB);
165        max = vsimGran;
166        break;
167     case 3:
168        LevensteinDistance simL = new LevensteinDistance();
169        vsimLeven = simL.getDistance(nameA, nameB);
170        max = vsimLeven;
171        break;
172     }
173     return max;
174 }
```

A execução do algoritmo de *database model matching* escolhido retorna um conjunto de correspondências candidatas. Este conjunto de elementos correspondidos devem ser confirmados pelo especialista de domínio. A Figura 5.7 mostra a interface de confirmação das correspondências candidatas. Nela o especialista seleciona quais correspondências são verdadeiras. As correspondências não selecionadas são consideradas como resultados falso positivo, isto é, correspondências indicada pelo algoritmo de *database model matching* como verdadeiras, mas que na realidade não deveriam ser candidatas. A validação das correspondências indica valores que são considerados no cálculo da medida de qualidade de *match*. Outra possibilidade fornecida para o especialista é a escolha de criar um mapeamento apenas de fragmento dos *database model* de entrada. Para isto, basta que seja selecionado apenas os fragmentos que se deseja ter um mapeamento na interface de validação de *match* (Figura 5.7). Deve-se lembrar, que a seleção de fragmento altera a medida de qualidade de *match*, isto porque, a função responsável pelo cálculo sempre leva em consideração todas as correspondências encontradas.

Outro aspecto que deve ser ressaltado é que a construção de um modelo de mapeamento pode ser o resultado de execuções de algoritmos de *match* diferentes. Como visto na Figura 5.6, o especialista pode fazer combinações de qual algoritmo aplicar, e ainda, qual similaridade de métrica utilizar. Desde modo, a cada execução

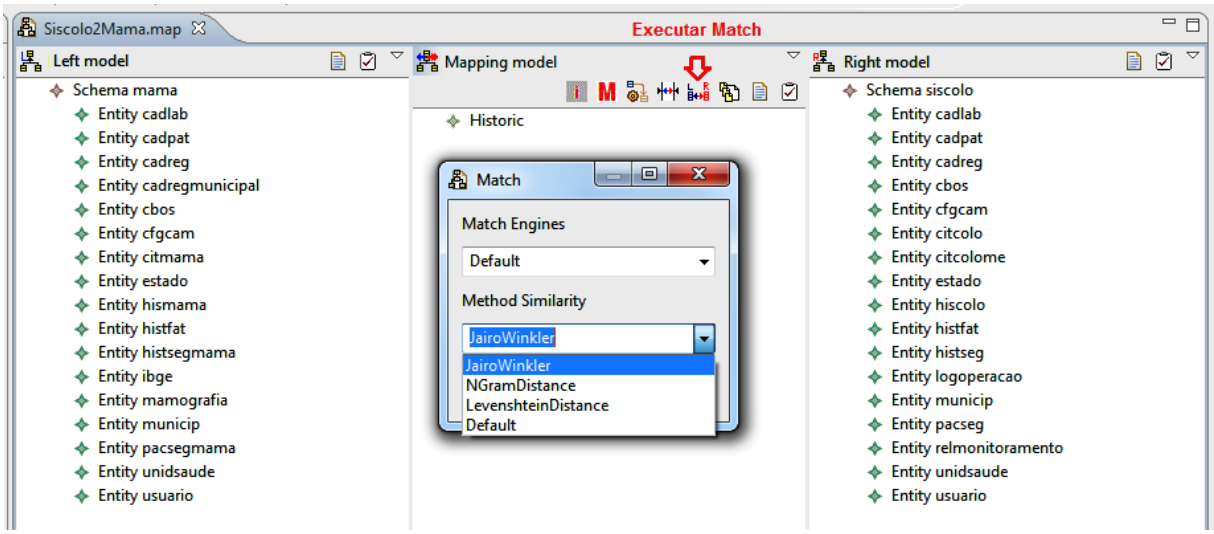


Figura 5.6: Interface de execução de *match*.

é retornada uma lista candidata de elementos correspondentes, que pode trazer correspondências que não foram encontradas em uma outra execução. Deste modo, o especialista constrói o modelo de mapeamento através de interações de execuções de *matching*.

O *database model matching* instanciado pelo algoritmo de *database model matching* e validado pelo especialista de domínio é o modelo de entrada para construção modelo de mapeamento (*mapping model*). Este modelo de mapeamento indica com qual elemento do *database model* alvo um elemento do *database model* fonte se relaciona.

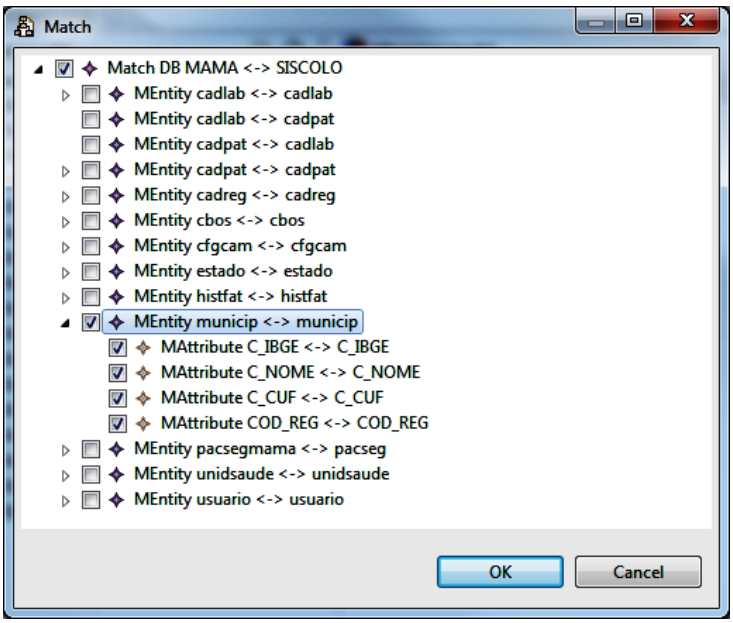


Figura 5.7: Interface de validação de *match*.

A Figura 5.8 mostra a interface que é disponibilizada para o especialista após a construção do modelo de mapeamento. A esquerda é exibido o *database model fonte*, a direita o *database model* alvo e ao centro o mapeamento entre os dois modelos. O mapeamento pode ser manipulado pelo especialista para a inclusão de mapeamentos não identificados pelo algoritmo de *database model matching*, bem como, para a exclusão de um mapeamento indesejado.

O modelo de mapeamento é um modelo conforme *mapping metamodel* e tem como raiz uma definição de mapeamento (*Definition mama2siscolo*). A definição de mapeamento é composta por elementos como: *Model Handler*, que contém elementos do *database model* fonte e alvo; *Correspondence*, que contém elementos correspondidos entre modelos fonte e alvo. Os elementos *Input* e *Output* indicam o elemento do modelo fonte (Input) e do modelo alvo (Output) que são correspondidos; *Merge*, que contém o mapeamentos de elementos fonte e alvo resultante da execução do *merge*. O mapeamento do *merge*, indicando quais elementos sofrerão ou não fusão no momento da construção do modelo integrado.

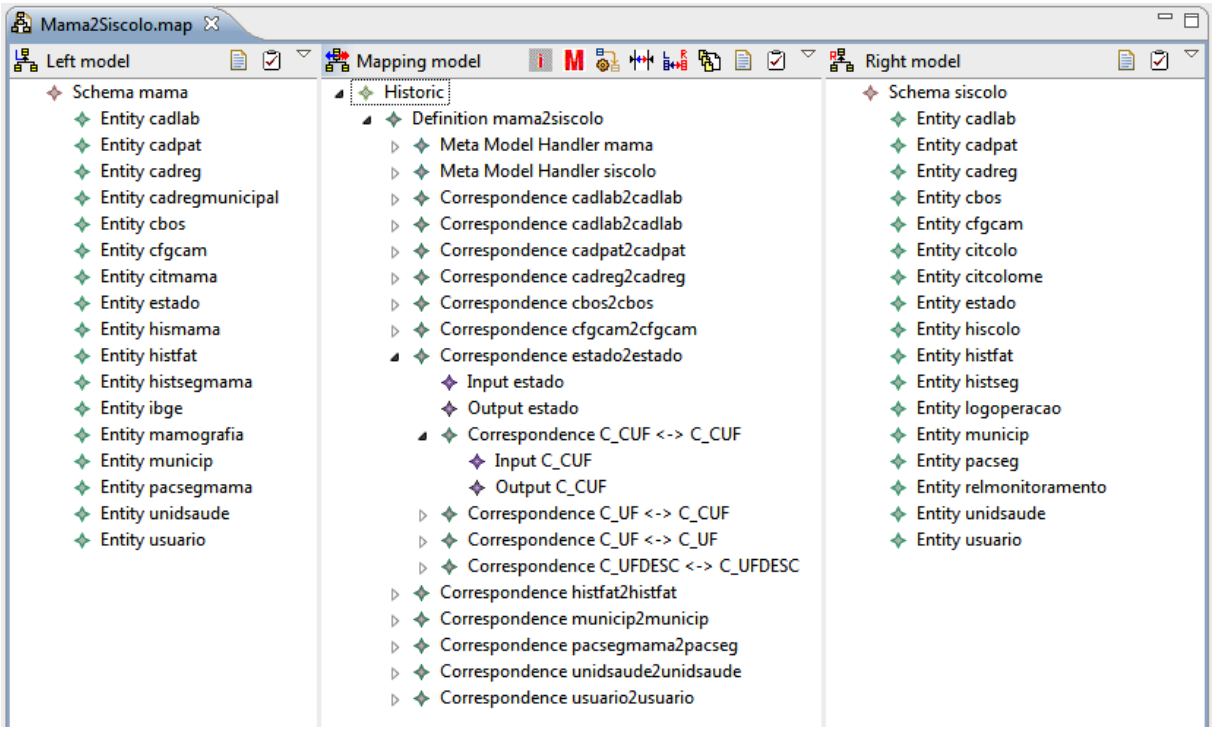


Figura 5.8: Interface para edição, exclusão e inclusão de correspondência.

Gerar o *merge* entre *database models*

O processo de *merging* tem como entrada a tupla (*database model fonte, database model alvo, mapping model*). O objetivo desta etapa é criar um modelo que especifica quais elementos farão parte do modelo integrado. Isto é, quais entidades dos *database models* fonte e alvos serão sobrepostas, e também, quais entidades que não sofrerão *merge*. Em [51] cinco requisitos que devem ser seguidos são propostos para a construção de um esquema integrado. Um desses requisitos é a preservação de extensão de *overlap*, que diz que elementos não sobrepostos também devem fazer parte do esquema integrado. Por isso, o modelo de *merging* proposto é composto de elementos sobrepostos dos *database models* e dos elementos fonte e alvo sem correspondência.

Esta etapa segue o algoritmo de *merging* apresentado na seção 4.6. Como visto na seção 4.6, a fusão de entidades dos *database models* estão baseadas no grau de correspondência (*no match, high match, low match e middle match*) entre atributos de entidades dos *database models* fonte e alvo. Assim, deve ser estabelecido um valor limite para o *high match*, *low match* e *middle match*. Este valor determina o percentual que deve ser atingido para que as entidades dos *database models* fonte e alvo sofram *merge*. Como ilustra a 5.9 o especialista de domínio escolhe qual algoritmo de *database model merging* será aplicado, bem como, os percentual que deve ser considerado para os limites de *high match* e *low match* de forma que retorne um mapeamento de *merge* adequado para os *database models* de entrada.

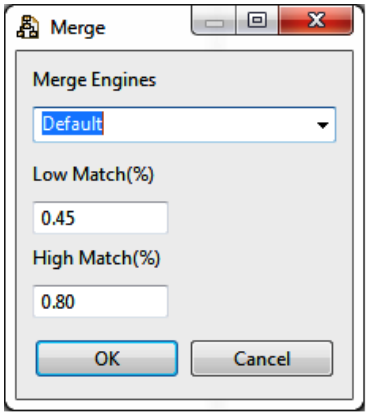


Figura 5.9: Interface de execução e configuração do *merge*.

O algoritmo de *database model merging* propõe a construção dos conjuntos *noMatch* (conjunto E), *total match* (conjunto T) e *partial match* (conjunto Υ), que agrupam as entidades pelo grau de correspondências entre seus atributos.

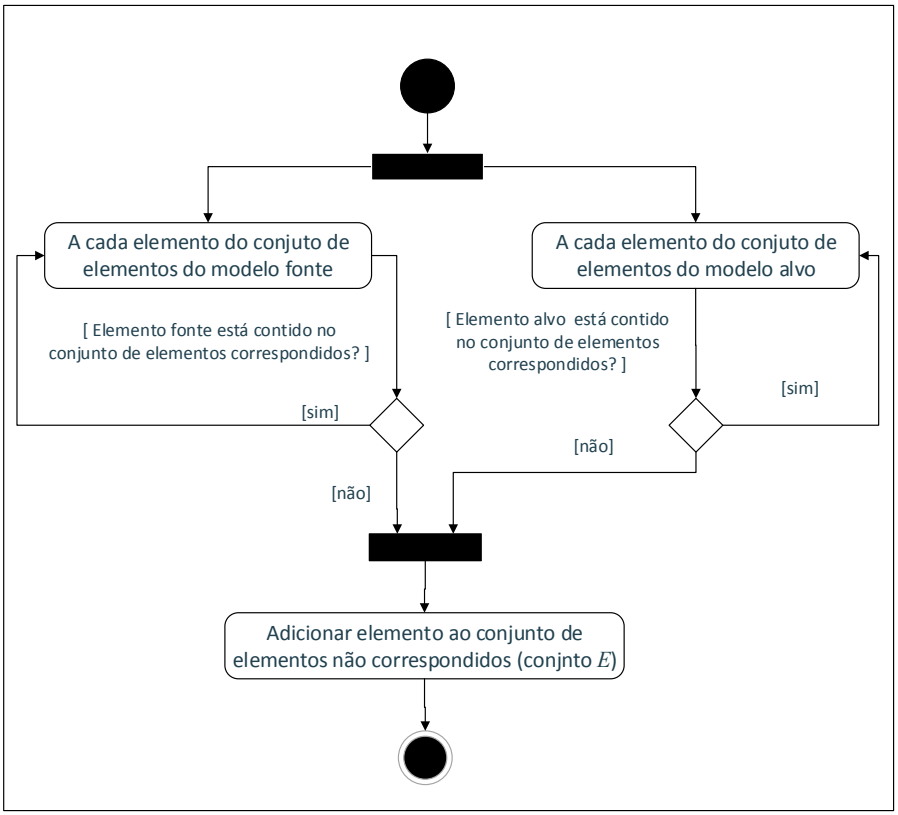


Figura 5.10: Gerar conjunto E com elementos não correspondidos.

A Figura 5.10 mostra o diagrama de atividades para o processo que gera a lista de elementos não correspondidos entre dois *database model*. A Listagem 5.2 mostra a construção do conjunto de entidades dos *database model* fonte e alvo que não possuem correspondências (conjunto E). A classe *ElementHandler* armazena os elementos do *database model*. Para cada elemento dos *database models* fonte e alvo é criado um elemento *ElementHandler* no modelo de mapeamento. Isto permite a navegação entre todos elementos dos *database models* fonte e alvo. A lista *elementsMatch* armazena todos os elementos que possuem correspondências, assim, quando um elemento não está contido nesta lista, ele é incluindo na lista *allNoMatch*(lista dos elementos não correspondidos), como mostra as linhas 149 e 150 do código a baixo. Para cada entidade pertencente ao conjunto E é criada uma entidade no *schema merging model*.

Listagem 5.2: Fragmento de código que constrói lista do elementos *noMatch*

```
147 for (Iterator<ElementHandler> iNoMatch = elHandlerSource.iterator();iNoMatch.hasNext();) {
148    ElementHandler elementHandlers = iNoMatch.next();
149    if (!elementsMatch.contains(elementHandlers)) {
150       allNoMatch.add(elementHandlers);
```

```
151     }
152 }
```

O processo de obter a lista de entidades com correspondência total é ilustrado pelo diagrama de atividades apresentado na Figura 5.11.

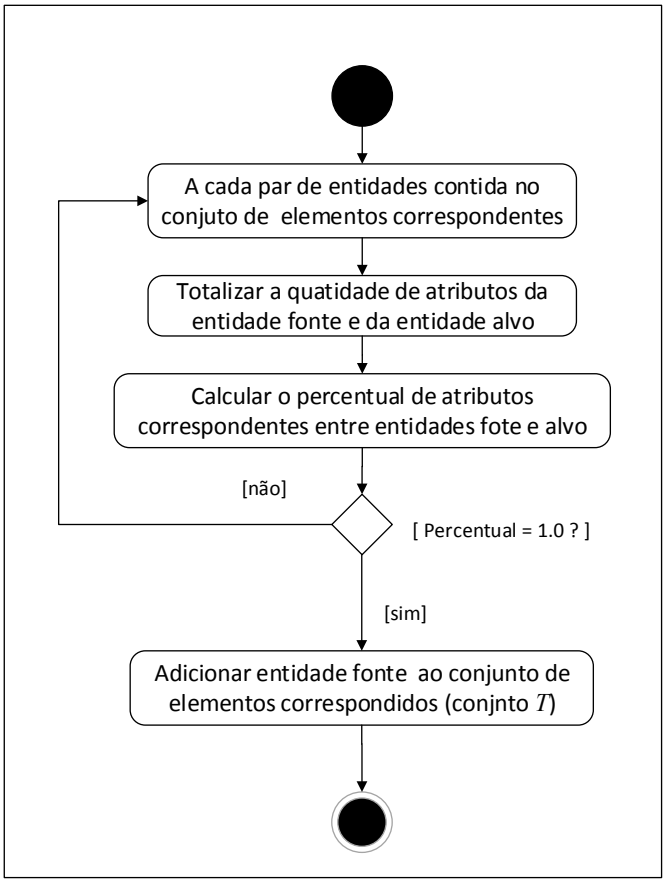


Figura 5.11: Gerar conjunto T com entidades com todos atributos correspondidos.

A Listagem 5.3 mostra a construção do conjunto de entidades dos *database models* fonte e alvo que possuem todos seus atributos correspondentes, *total match* (conjunto T). A lista *correspondences* (linha 245) contém entidades que possuem atributos correspondentes. Esta lista é percorrida verificando se todos os atributos da entidade fonte correspondem a todos os atributos da entidade alvo. As variáveis *totalHandlerInput* e *totalHandlerOutput* guardam quantos atributos as entidades fonte e alvo possuem respectivamente. A variável *totalCorresp* possui a quantidade de atributos correspondidos entre entidade fonte e alvo. A variável *perc* guarda o percentual de correspondência dos atributos das entidades fonte e alvo. Caso esse percentual resulte 1, significa

que todos seus atributos são correspondentes, sendo então, incluída uma única entidade no *schema merging model* representando o *merge* de entidades fonte e alvo.

Listagem 5.3: Fragmento de código que constrói lista do elementos *total match*

```
245 for (Iterator<Correspondence> irules = correspondences.iterator(); irules.hasNext();) {
246    Correspondence elCorresp = irules.next();
247    float totalHandlerInput = elCorresp.getInput().getHandler().getNested().size();
248    Output output = (Output) elCorresp.getOutput().get(0);
249    float totalHandlerOutput =output.getHandler().getNested().size();
250    float totalCorresp = elCorresp.getNested().size();
251    float totalHandler = totalHandlerInput+totalHandlerOutput;
252    float perc = (2*totalCorresp) / totalHandler;
253    //total match
254    if (perc == 1) {
255       MeEntity meEntity = createEntity(elCorresp);
256       mergeS.getMergeEntity().add(meEntity);
257    }
258 }
```

O conjunto Υ agrupa entidades com atributos correspondidos parcialmente (*partil match*). Como descrito na seção 4.6, sua construção é baseada na baixa (*low match*), média (*middle match*) e alta (*high match*) correspondência entre os atributos de entidades fonte e alvo. Esta classificação de correspondência parcial entre atributos determina se as entidades fontes e alvo serão fundidas ou não. Assim, entidades com *low match* não serão fundidas, entidades com *high match* serão fundidas em uma única entidades e entidades *middle match* sofrerão o processo de *generalização de correspondência*, como foi apresentado na seção 4.6.

A Listagem 5.4 mostra como entidades com *middle match* são generalizadas. A função *createEntity* (linha 480) cria uma nova entidade apenas com os atributos correspondidos entre entidades fonte e alvo. A esta nova entidade é incluída um atributo para identificar a origem de tuplas, já que as entidades fonte e alvo foram sobrepostas. Isto é necessário, porque esta entidade será populada com tuplas originadas das entidades fonte e alvo, necessitando assim, identificar a sua origem. Na *generalização de correspondências*, além da entidade criada com os atributos correspondentes, existe também, a criação de novas entidades para manter os atributos das entidades fonte e alvo não são correspondidos. A função *entitySpecialized* tem o objetivo de criar estas novas entidades(linha 494).

Listagem 5.4: Fragmento de código que constrói entidade generalizada

```
478 //middle match
479 //Criar nova entidade com match de atributos das entidades fonte e alvo
480 MeEntity newEntidade = createEntity(elCorresp);
481 String entitySource =capitalized(eInput);;
482 String entityTarget =capitalized(eOutput);
483 String nomeNewEntidade = entitySource+"."+entityTarget;
484 newEntidade.setName(nomeNewEntidade);
485 newEntidade.setMerge(true);
486
487 //incluir atributo identificador de origem de tuplas
488 MeAttribute originTuplas = schemaMergingFactory.createMeAttribute();
489 originTuplas.setName("originTupla");
490 originTuplas.setMerge(false);
491 originTuplas.setOwnerAttribute(newEntidade);
492
493 //criar entidade especializada
494 entitySpecialized(source,target,newEntidade);
495
496 mergeS.getMergeEntity().add(newEntidade);
```

A Listagem 5.5 mostra a implementação da função *entitySpecialized*, que tem como entrada as entidades fonte (*ElementHandler source*) e alvo (*ElementHandler target*), e também, um parâmetro para a nova entidade que representa a generalização (*newEntidade*). Como mostra as linhas de 482 e 495, uma busca por atributos das entidades fonte e alvo que não estão presentes na lista de elementos correspondentes (*elementsMatch*) é realizada. Caso os atributos não estejam contidos nesta lista, então é criada uma nova entidade composta pelos atributos não correspondidos. Portanto, a função *entitySpecialized* cria duas entidades, uma entidade com os atributos do database model fonte não correspondidos e uma entidade como os atributos do database model alvo não correspondidos. Estas duas entidades são incluídas no *database merging model* com filhas da entidade generalizada. O *database merging model* gerado, deve ser validado pelo especialista de domínio.

Listagem 5.5: Fragmento de código que constrói entidade generalizada

```
478 protected void entitySpecialized(ElementHandler source, ElementHandler target, MeEntity ↩
        newEntidade){
479
480     MeEntity newEntityS = schemaMergingFactory.createMeEntity();
481     newEntityS.setName(capitalized(objDef.getSource().getName()+"."+capitalized(source.↩
            getName())));
```

```
482    for(Iterator<ElementHandler> iHandler = source.getNested().iterator();iHandler.hasNext()↩
           ;){
483       ElementHandler handlerSource = iHandler.next();
484       MeAttribute newAttrib = schemaMergingFactory.createMeAttribute();
485       if(!elementsMatch.contains(handlerSource)){
486          newAttrib.setName(handlerSource.getName());
487          newAttrib.setOwnerAttribute(newEntityS);
488       }
489    }
490    newEntidade.getNestedEntity().add(newEntityS);
491
492    boolean hasAttrib=false;
493    MeEntity newEntityT = schemaMergingFactory.createMeEntity();
494    newEntityT.setName(capitalized(target.getOwner().getName())+"."+capitalized(target.↩
           getName()));
495    for(Iterator<ElementHandler> iHandler = target.getNested().iterator();iHandler.hasNext()↩
           ;){
496       ElementHandler handlerTarget = iHandler.next();
497       MeAttribute newAttrib = schemaMergingFactory.createMeAttribute();
498       if(!elementsMatch.contains(handlerTarget)){
499          newAttrib.setName(handlerTarget.getName());
500          newAttrib.setOwnerAttribute(newEntityT);
501          hasAttrib = true;
502       }
503    }
504    if (hasAttrib){
505       newEntidade.getNestedEntity().add(newEntityT);
506    }
507 }
```

A Figura 5.12 mostra a interface para que o especialista confirme o *database merging model* sugerido pelo algoritmo de *database model merging*. A entidade *estado* destacada pelo retângulo vermelho sofreu o processo de fusão. O *Merge Target Entity* indica a entidade do database alvo que foi sobreposta com a entidade do *database model* fonte.

A Figura 5.13 mostra a generalização de entidades. Os atributos marcados com pontos azuis são os elementos sobrepostos, enquanto que os atributos marcados com pontos verdes são elementos sem correspondências entres as entidades fonte e alvo. Para os atributos marcados com pontos verdes, novas entidades foram criadas apenas com os atributos sem correspondências entre *database model* fonte e alvo.

Como já mencionado, o *database merging model* guarda apenas referências aos elementos do database fonte e alvo. A estrutura real do *dabase model* integrado

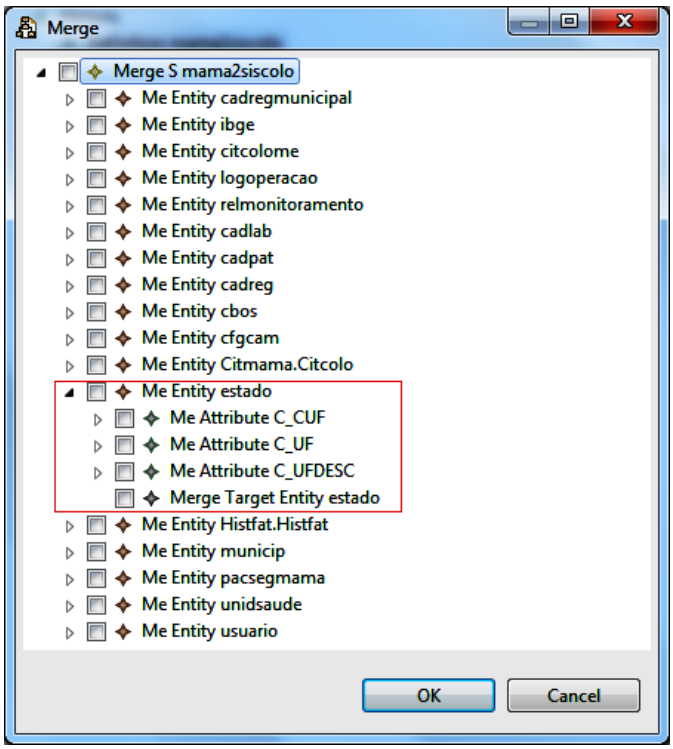


Figura 5.12: Interface de confirmação de *merging* e a fusão das entidades *estado* dos modelos fonte e alvo.

é criada quando for feita a transformação do *database merging model* para o *database integreted model*.

Gerar o *database integrated model*

Esta etapa trata da construção de um modelo integrado. A construção de um modelo integrado é necessário, porque o *database merging model* representa apenas um mapeamento dos *database model* fonte e alvo. Assim, para a construção de um modelo que possa ser transformado em um SQL script é necessário a inclusão de novos elementos.

A Figura 5.14 mostra o *database integrated metamodel* que permite a construção de um modelo adequado para ser transformado em um SQL script. O *database integrated metamodel* é semelhante ao *database metamodel*. As diferenças importantes entre eles são: a classe *Definition* que registra informações de data da primeira e última versão e nomes dos *database schemas* fonte e alvo; a classe *IdbEntity*, que neste metamodelo está preparada para suportar a *generalização de correspondência* proposta no algoritmo de *database model merging*. O relacionamento *superEntity* indica que uma en-

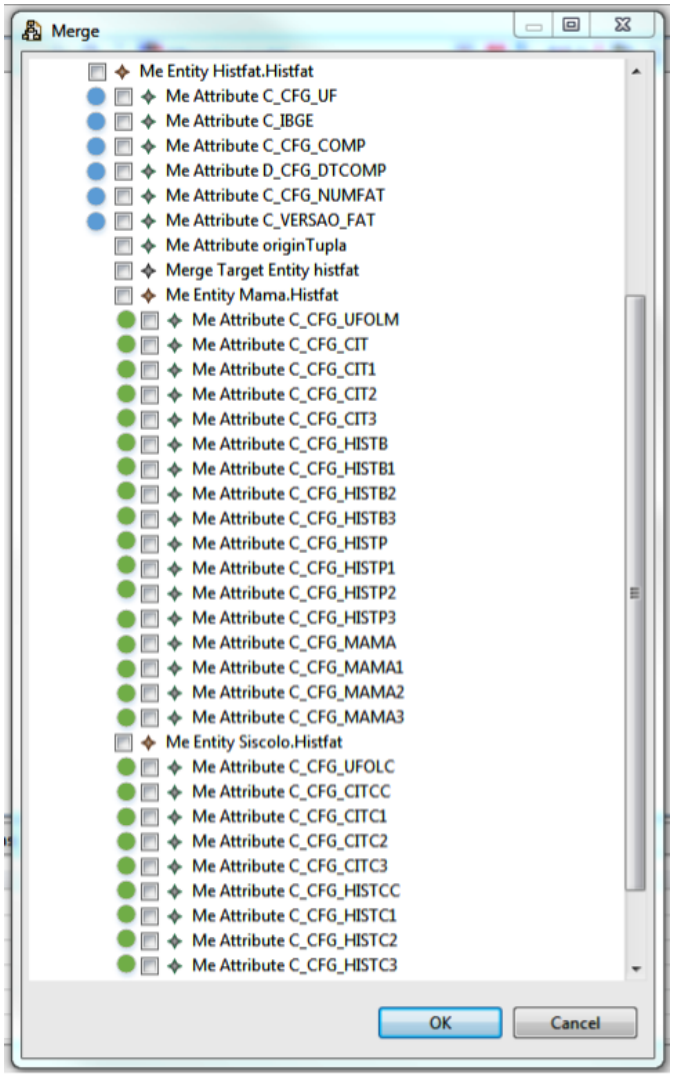


Figura 5.13: Interface de confirmação de *merging*.

tidade se relaciona com uma ou mais outras entidades, como proposto na *generalização de correspondência*.

O modelo integrado é obtido através do processo de transformação dos *database model* fonte e alvo e do *database merging model*. O *database merging model* serve de base para a sua criação, pois, possui o mapeamento indicando quais elementos (entidades, atributos) dos *database models* (fonte ou alvo) devem ser sobrepostos. Indica também, os elementos (entidades, atributos) dos *database models* (fonte ou alvo) que não possuem correspondência. A construção do *database integrated model* foi realizado da seguinte forma:

- Carregar o *database models* (fonte e alvo) e o *database merging model*;

- Para cada entidade presente no *database merging model*, criar uma entidade no *dadabase integrated model* com uma nova chave primária. O nome da chave primária segue o seguinte formato: ID+ NOME ENTIDADE+K, isto é, a chave inicia com os caracteres ID, depois o nome da entidade e por fim o caractere K;

- Após a criação de todas entidades com suas chaves primárias, são criados as chaves estrangeiras de cada entidade.

A Listagem 5.6 mostra um trecho do código de criação de chave primária para uma entidade do *database integrated model*. A linha 107 apresenta a atribuição do nome da chave primária conforme definida anteriormente.

Listagem 5.6: Fragmento de código que constrói chaves primárias em entidades do *database merging model*

```
105 //Criar chave primária
106 IdbPrimaryKey primaryKey = idbFactory.createIdbPrimaryKey();
107 primaryKey.setName("id" + idbEntity.getName()+"K");
108 primaryKey.setOwnerKey(idbEntity);
```

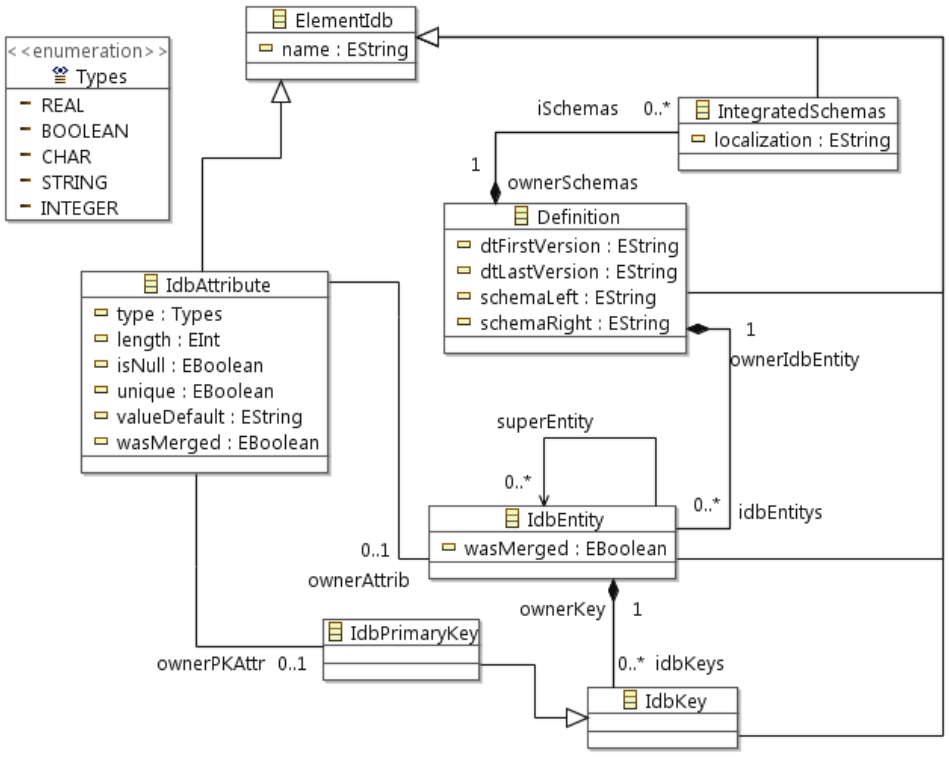
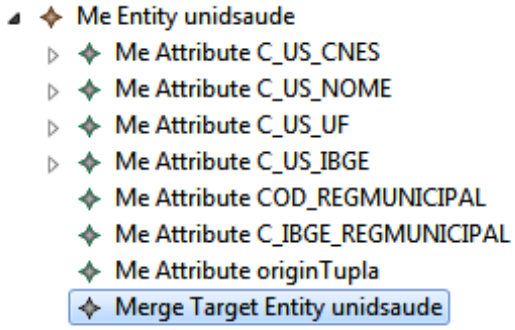
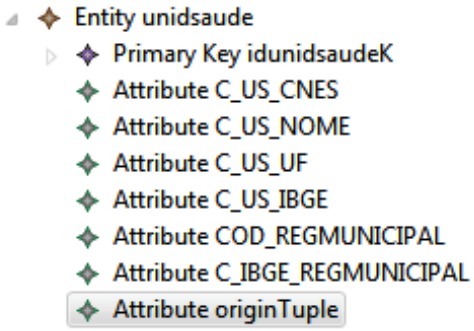


Figura 5.14: Metamodelo de integração de *database*.



(a) *Database merge model* da entidade *unidsaude*



(b) *Database integrated model* da entidade *unidsaude*.

Figura 5.15: Transformação do *database merging model* para *database integrated model*.

A Figura 5.15(a) ilustra um fragmento de um *database merge model*, onde tem-se uma entidade que foi sobrepostas a outra. A entidade do modelo fonte teve uma correspondência parcial com a entidade do modelo alvo, percebe-se isto pois, a entidade possui um atributo identificador de origem de tuplas (*origintuple*). A Figura 5.15(b) mostra um fragmento de um *database integrated model* já com a entidade indicada como sobreposta pelo *database merge model*.

Diferentemente do *database merging model*, o *database integrated model* já tem toda a sua estrutura pronta para ser transformado em um SQL *script*. A Figura 5.16 mostra as propriedades de um atributo de uma entidade definida em um *database integrated model*. Como pode ser visto na Figura 5.16, atributos como tamanho (*Length*) e tipo (*Type*) já possuem valores. Os valores de propriedades são os mesmos definidos no *database model* fonte e alvo.

Property	Value
Destine	
Is Null	false
Length	7
Name	C_US_CNES
Type	STRING
Unique	false
Value Default	
Was Merged	false

Figura 5.16: Propriedades de um atributo de uma entidade de um *database integrated model*.

Gerar o SQL *script*

Esta é a última etapa da metodologia apresentada na seção 4.2. O *database integrated model* sofre transformação para um modelo SQL, que está conforme o metamodelo SQL. As especificações de correspondência entre *database integrated metamodel* e SQL *metamodel* são definidas em ATL. Da mesma forma, definição de transformação escritas em ATL são aplicadas ao modelo SQL obtendo um SQL script.

Assim, o objetivo de obter um SQL *script* contendo o *merging* de *database model* fonte e alvo é atingido. O SQL *script* obtido pode agora ser executado em um DBMS. A Listagem 5.7 mostra um trecho de código em ATL que define o elemento coluna de uma tabela. As especificações de correspondências e definições de transformações escritas em ATL utilizadas na implementação do *framework* serão vistas no Capítulo 6.

Listagem 5.7: Fragmento de código em ATL que constrói coluna de uma tabela.

```
10 helper context SQL!Column def : toString() : String =
11    '\n'+self.name+' '
12    +if self.length > 0 then
13       self.type + '('+self.length+')'
14     else
15       self.type
16     endif
17    +',';
```

5.2 Síntese

Neste capítulo, a implementação do *framework* para suportar *database model merging* foi apresentada. A implementação levou em consideração a arquitetura proposta para o *framework*, bem como, a metodologia proposta neste trabalho.

Algoritmos de *database model matching* e *database model merging* foram implementados estendendo as interfaces de *Match* e *Merge* proposta na arquitetura do *framework*. A implementação de algoritmos e construção de metamodelos foram definidos em EMF, bem como, as transformações de modelos.

Assim, a partir dos *database model* fonte e alvo , transformação de modelos implementadas em EMF geraram os *database matching model*, *mapping model*, *database merging model* e *database integrated model*. Por fim, o *database integrated model* foi transformado em um *SQL script* através de especificações de correspondência de metamodelos e definições de transformações escritas em ATL.

6 APLICAÇÃO DO SID4MDE AOS SISTEMAS DO SUS

Neste capítulo, a aplicação do protótipo do *framework* de suporte a *database model merging* no domínio da saúde é apresentada. Uma breve explicação sobre os sistemas do Sistema Único de Saúde (SUS) escolhidos para a aplicação do protótipo é apresentada. E logo em seguida, guiado pela metodologia proposta, os *database* esquemas dos sistemas são integrados através do protótipo do *framework*. O objetivo é avaliar as funcionalidades do *framework* proposto, bem como, seus algoritmos e metamodelos.

6.1 Sistemas de Gerenciamento do SUS

A motivação para desenvolver uma ferramenta que auxilia o processo de integração de esquema de base de dados surgiu observando a dificuldade de profissionais de saúde em colher dados dos sistemas de gerenciamento do Sistema Único de Saúde - SUS para suas pesquisas sobre agravos da população brasileira. O SUS é um sistema público de saúde que abrange atendimento ambulatorial, internação hospitalar, transplante de órgão e distribuição de medicamentos entre outras atividades criado em 1998, pela Constituição Federal Brasileira [44].

O SUS disponibiliza sistemas para registros epidemiológicos, hospitalares, ambulatoriais e financeiros. Essa variedade de sistemas produz uma quantidade de informações importantes para a tomada de decisão dos gestores de saúde. Contudo, muitos desses sistemas não são integrados, fragmentando assim, o acesso à informação. Para exemplificar o problema da falta de integração dos sistemas do SUS, se quisermos fazer o cruzamento de informações financeiras e epidemiológicas de um paciente que foi internado com dengue em um hospital temos que consultar dois sistemas, o financeiro (SIH/SUS) e o epidemiológico (SINAN) [14]. Não existe um identificador único que permita a identificação de um paciente nas bases de dados de todos os sistemas do SUS. Uma tentativa de criar esse identificador é a implantação do Car-

tão Nacional de Saúde, que pretende identificar unicamente um usuário. Porém, nem todos os sistemas estão preparados para o seu uso, além de haver falhas em sua implantação, como por exemplo um usuário do SUS possuir mais de um Cartão Nacional de Saúde.

No caso dos sistemas do SUS, acreditamos que criar mecanismo que auxilie a integração de suas bases de dados pode minimizar o problema de fragmentação da informação, facilitando o trabalho de cruzamento de dados dos profissionais de saúde, agilizando assim, as ações para tratar agravos da população. Portanto, a pesquisa científica proposta neste trabalho envolve o estudo do problema de integração de base de dados com a abordagem de desenvolvimento MDE e tem como domínio de aplicação os sistemas de informação utilizados para controle e gerenciamento do Sistema Único de Saúde-SUS.

Os sistemas SISMAMA e SISCOLO são um exemplo da falta de integração dos sistemas. O SISCOLO e o SISMAMA são sistemas utilizados no âmbito do SUS para cadastrar exames citopatológicos e histopatológicos do colo do útero e mama, e também mamografia [57].

Os dois sistemas foram desenvolvidos para tratar o mesmo problema, monitorar ações de detecção precoce do câncer, porém, suas bases de dados são distintas. As estruturas de base de dados dos sistemas são muito parecidas, o que nos permite deduzir que um único sistema poderia ter sido desenvolvido. A existência dos dois sistemas cria uma dificuldade aos profissional para cruzar informações de pacientes que apresentem câncer de útero e mama.

A Figura 6.1 mostra o Diagrama de Entidade e Relacionamento (DER) do sistema SISCOLO. O sistema SISCOLO possui 17 tabelas com um total de 481 colunas. A estrutura da base de dados não está bem normalizada, algumas tabelas tem mais de 80 colunas, além disso, é frequente a mesma informação presente em mais de uma tabela. Isto pode ser visto nas tabelas *citcolo* e *citcolome*, onde informações de pacientes como nome e nome da mãe aparecem nas duas tabelas.

Como citado, o sistema SISMAMA tem sua estrutura parecida com a do sistema SISCOLO. Algumas tabelas inclusive são iguais nas duas bases de dados. Assim, os mesmos problemas de repetição de informação e tabelas com um grande número de colunas são presentes no sistema SISMAMA. A Figura 6.2 mostra o DER do sistema

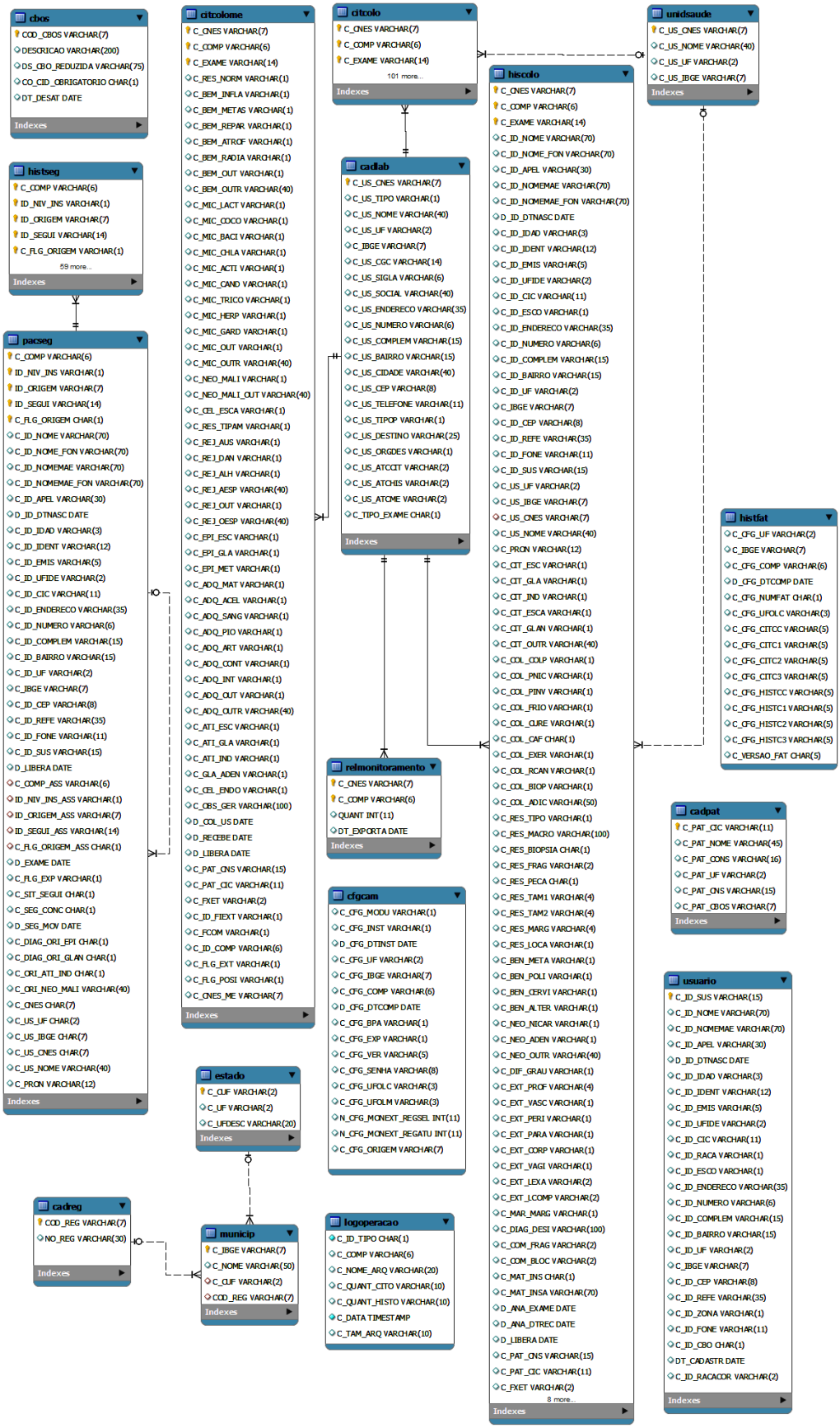


Figura 6.1: Diagrama de Entidade Relacionamento do SISCOLO.

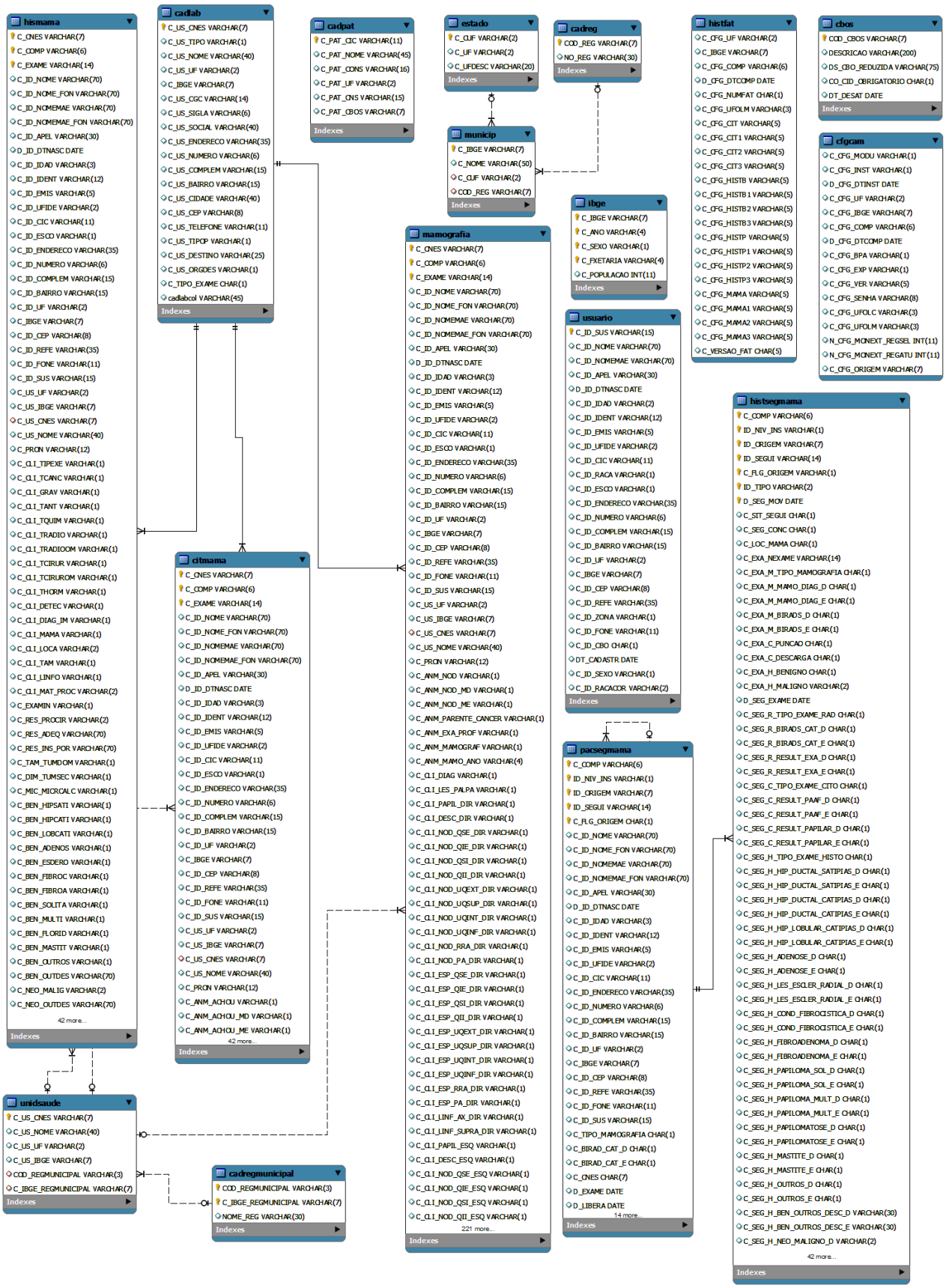


Figura 6.2: Diagrama de Entidade Relacionamento do SISMAMA.

SISMAMA. O sistema SISMAMA é composto de 17 tabelas com um total de 741 colunas. Na realização de correspondências reais entre os elementos dos sistemas foram encontradas 248 correspondências.

O diagrama de entidade e relacionamento mostrado na Figura 6.1 foi gerado no MySql Workbench através do recurso de engenharia reversa da base de dados do sistema SISCOLO. O mesmo foi feito com para o diagrama de entidade e relacionamento do sistema SISMAMA.

A esses dois sistemas foi aplicado o protótipo do *framework* proposto para gerar um *database integrated model* das bases de dados dos sistemas SISCOLO e SISMAMA. O processo de geração de um modelo integrado seguiu a metodologia proposta no trabalho, como descrito nas seções a seguir:

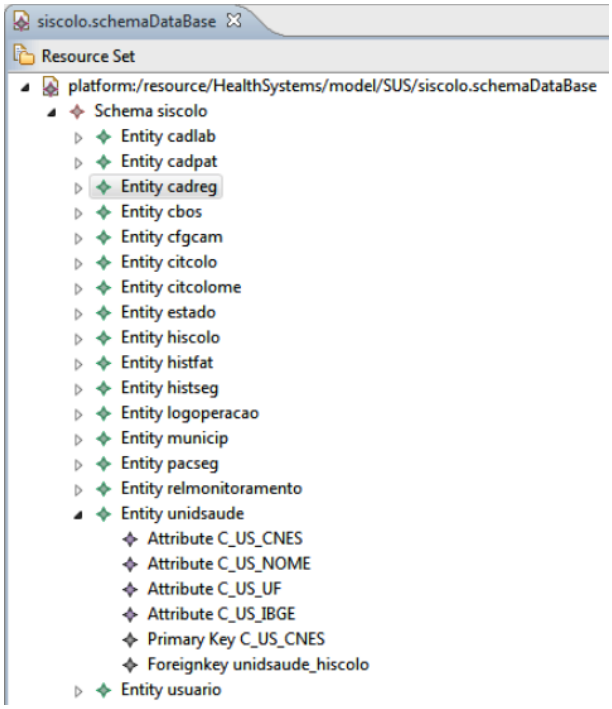
Importar os esquemas de base de dados dos sistemas

Os sistemas possuem banco de dados relacional *Firebird* [22]. Assim, o *database model* dos sistemas foram instanciados a partir da recuperação de informações de metadados diretamente de seus banco de dados. O pacote *java.sql.DatabaseMetaData* disponibiliza métodos que permitem acessar um esquema de base de dados, podendo assim, recuperar informações como nome de tabelas e suas colunas, além de chaves primárias e estrangeiras.

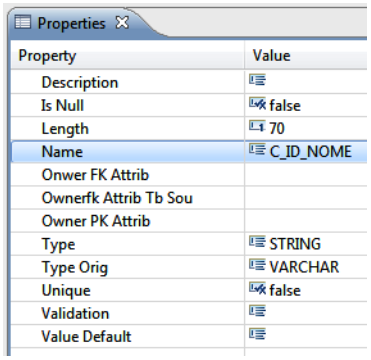
A Figura 6.3(a) mostra os *database models* dos sistemas SISCOLO e a 6.3(c) do sistema SISMAMA após a importação dos metadados dos esquemas de suas bases de dados. Como pode ser observado nestas figuras, informações de nome de tabelas, bem como, chave primária e estrangeiras foram recuperadas. A Figura 6.3(b) mostra as propriedades de um atributo, como *Length*,*Type* e *Type Orig*. Estas propriedades foram atribuídas a partir da recuperação de metadados. A propriedade *Type Orig* registrar o tipo da coluna na base de dados de origem.

Criar *Mapping Model* dos *Databese Model* SISCOLO e SISMAMA

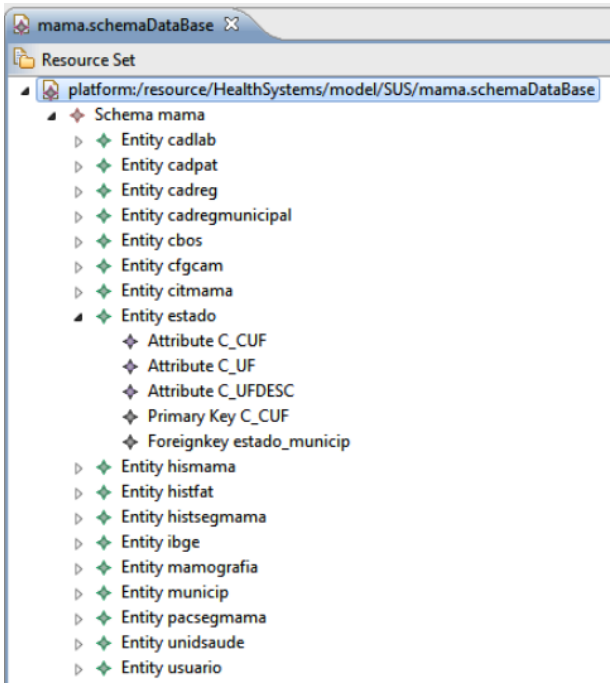
Com os *database model* dos sistemas instanciados, a tarefa de encontrar elementos correspondentes entre os dois modelos pode ser executada. O especialista de domínio pode escolher qual o algoritmo de *database model matching* quer utilizar, como



(a) *Database model* do SISCOLO



(b) Propriedade de um atributo do SISMAMA.



(c) *Database model* do SISMAMA.

Figura 6.3: *Database model* instanciado a partir de metadados.

também, qual métrica de similaridade de *string* será aplicada para determinar se duas *strings* são similares.

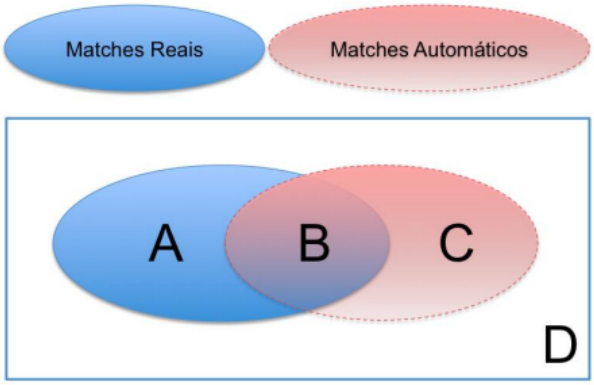


Figura 6.4: Conjuntos de *matches* para cálculo da *Precision* e *Recall* [10].

A medida de qualidade de correspondência (*Match Quality Measures*) [24] é utilizada para determinar a qualidade de algoritmo de *database model matching* proposto. Para determinar a qualidade do *matching* são considerados o conjunto de correspondências reais encontradas pelo especialista de domínio entre dois modelos e o conjunto de correspondências encontradas pelo algoritmo de *database model matching*.

As medidas de *Precision* e de *Recall* são utilizadas para obter a qualidade do algoritmo de *database model matching*. O cálculo destas duas medidas é baseado nos seguintes conjuntos, como ilustra a Figura 6.4 e como descrito a seguir [24]:

- Conjunto dos falsos negativos (A): são correspondências necessárias, mas que não foram encontradas pelo algoritmo de *database model matching*;
- Conjunto dos verdadeiros positivos (B): são correspondências necessárias que foram encontradas pelo algoritmo de *database model matching*;
- Conjunto dos falsos positivos (C): são correspondências falsas propostas como verdadeiras pelo algoritmo de *database model matching*;
- Conjunto dos verdadeiros negativos (D): são correspondências falsas descartadas pelo algoritmo de *database model matching*.

Com os conjuntos A, B, C e D definidos, a *Precision* e o *Recall* é obtido aplicando as seguintes expressões [24]:

- ***Precision*** = $\frac{|B|}{|B|+|C|}$, representa a parte de correspondências reais dentre todas as correspondências retornadas;
- ***Recall*** = $\frac{|B|}{|A|+|B|}$, representa a parte real de correspondências que foi retornadas.

Nenhuma das medidas sozinhas pode determinar a qualidade do algoritmo de *database model matching*. Isto porque, a medida que o algoritmo de *database model matching* é ajustado para melhorar o *Recall*, a *Precision* é diminuída, o inverso também é verdadeiro.

Assim, para medir a qualidade de *matching* é necessário combinar as duas medidas de qualidade. As seguintes medidas podem ser usadas para esse fim, que são [24]:

- ***F-Measure*** = $2 * \frac{Precision * Recall}{Precision + Recall}$, que reflete a média harmônica entre *Precision* e *Recall*;
- ***Overall*** = $Recall * \left(2 - \frac{1}{Precision}\right)$, reflete o esforço *post-match* para incluir falsos negativos e remover falsos positivos.

O *Overall* pode ter um valor negativo quando o número de *falso positivos* exceder o número de *verdadeiros positivos* [24].

Um valor limite deve ser estabelecido para identificar elementos verdadeiros positivos (conjunto B). Deste modo, elementos com o grau de similaridade maior que este limite são considerados *verdadeiros positivos* e elementos com valores inferiores são considerados *falsos positivos* (conjunto C).

As medidas de qualidade de correspondências, foram aplicadas aos Sistemas SISCOLO e SISMAMA obtendo-se os resultados apresentados na tabela 6.1.

Tabela 6.1: Medida de qualidade do algoritmo de *database model matching*

	Precision	*Recall*	*F_Mesuare*	*Overall*
Combinação das Métricas	0,72	0,76	0,74	0,47
Jaro Winkler	0,04	0,98	0,09	- 18,41
Ngram	0,66	0,58	0,62	0,29
Levenshtein	0,75	0,75	0,75	0,51

O algoritmo de *database model matching* proposto permite a escolha da métrica de similaridade de *string* que será utilizada no processo de *matching*. Assim, a Tabela 6.1 mostra os resultados da aplicação de cada umas das métricas.

A métrica *Jaro Winkler* obteve o pior resultado como mostra a Tabela 6.1. Este resultado ruim deve-se ao fato dos *database models* apresentarem muitos atributos com nomes similares, mas que não representam correspondências verdadeiras (*falso positivos*). Assim, tem-se um alto valor para o *Recall* e um valor baixo para *Precision*. Valor de *Precision* < 0.5 implica em um *Overall* negativo [24].

A similaridade de *string Levenshtein* obteve o melhor resultado com valores equivalentes para todas as medidas de similaridade, implicando em um menor esforço para inserção de *verdadeiros positivos* e remoção de *falsos positivos*.

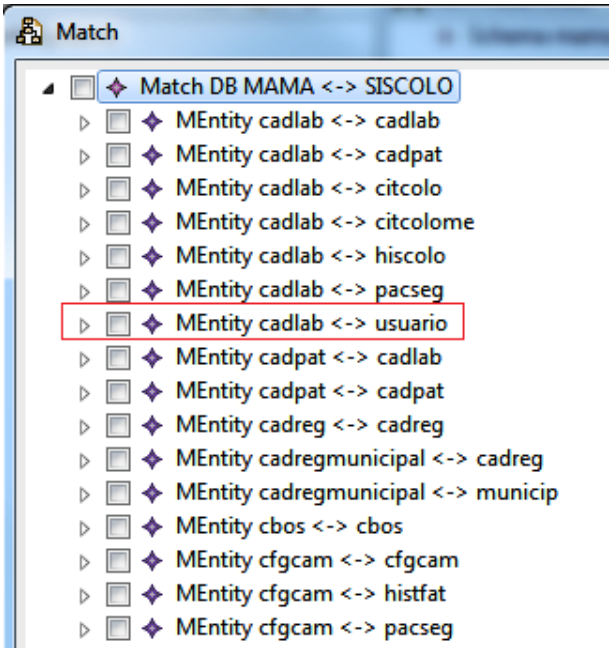
A combinação das métricas também obteve um bom resultado. A combinação das métricas faz a correspondência pela aplicação das três métricas da seguinte forma:

- Aplicando a métrica de *Jaro Winkler* é obtido um conjunto A com os elementos correspondentes dos *database model* fonte e alvo. Da mesma forma, é criado um conjunto B pela aplicação da métrica *Ngram* e conjunto C pela aplicação da métrica *Levenshtein*;
- O conjunto da combinação das métricas (conjunto D) é obtido como a seguir: D = (A ∩ B) ∪ (A ∩ C) ∪ (B ∩ C).

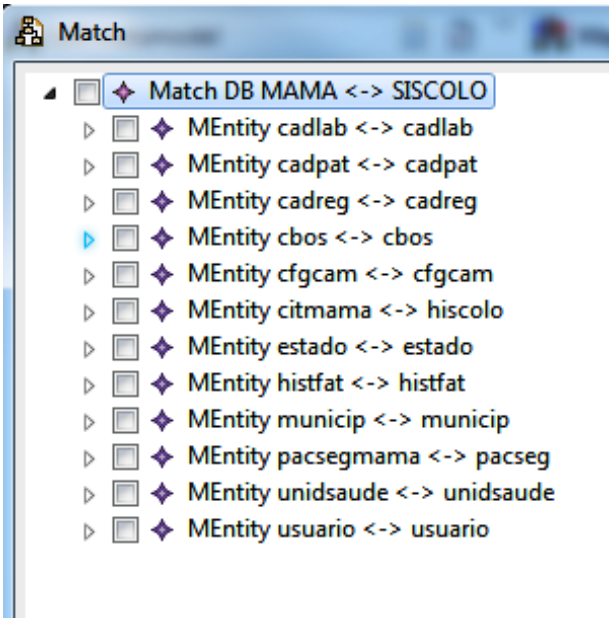
A Figura 6.5(a) mostra o resultado do algoritmo de *database model matching* utilizando a métrica *Jaro Winkler* e a Figura 6.5(b) o resultado utilizando a métrica de *Levenshtein*. Percebe-se que a métrica de *Jaro Winkler* tem um retorno correspondências *falsos positivas* maior que a métrica de *Levenshtein*, como por exemplo, o *match* entre *cadlab* e *usuário* que é encontrado na métrica *Jaro Winkler* e não encontrado na métrica *Levenshtein*.

O *matching* entre *cadlab* e *usuário* foi retornado porque o grau de correspondência entre seus atributos foi maior que o limite determinado para definir se duas entidades são similares pela similaridade de seus atributos.

Os valores limites para determinar se uma *string* é similar a outra e para determinar se duas entidades são similares por correspondências de seus atributos foram de 0.7 e 0.8, respectivamente. Além disso, foram removidos prefixos e posfixos dos nomes de entidades e atributos.



(a) Resultado do matching usando métrica *Jaro Winkler*.



(b) Resultado do matching usando métrica *Levenshtein*.

Figura 6.5: Retorno de algoritmo de *database model matching*.

A Figura 6.6 mostra o protótipo após a confirmação do especialista de domínio das correspondências sugeridas pelo algoritmo de *database model matching* dos sistemas SISCOLO e SISMAMA. A esquerda tem-se o *database model* SISMAMA, a direita o *database model* SISCOLO e ao centro o mapeamento entre eles com a correspondências confirmadas pelo especialista de domínio. A Figura 6.6 mostra também,

o mapeamento entre as entidades *histfat* dos dois modelos, onde é mostrado que nem todos os atributos das duas entidades foram correspondidos.

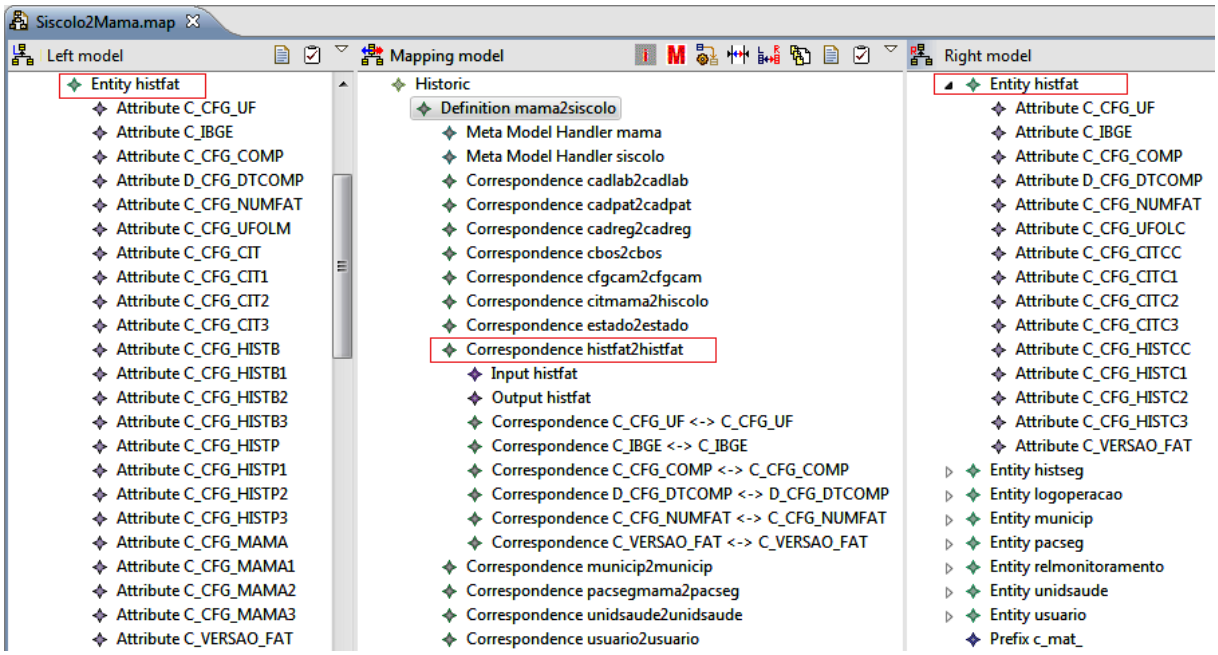


Figura 6.6: Protótipo com mapeamento dos *database models* SISCOLO e SISMAMA.

A possibilidade de ajustar o processo de *matching* com a definição de limites de similaridades e métricas que podem ser usadas, traz uma flexibilidade para especialista de domínio, pois, cada ajuste pode retornar correspondências que não foram encontradas em outros. Deste modo, o mapeamento entre *database models* pode ser conseguido através de sucessivas execuções do algoritmo de *database model matching*.

Criar *database model merging* dos *database models* SISCOLO e SISMAMA

Como visto anteriormente, para executar a tarefa de *merge* são necessários as seguintes entradas: dois *database models* (SISCOLO e SISMAMA) e um mapeamento entre os *database models*. O mapeamento entre os modelos SISCOLO e SISMAMA foi obtido pelo algoritmo de *database model matching*.

O *merging* entre os *database models* leva em consideração o grau de correspondência entre as entidades dos *database models*. Assim, o especialista de domínio deve determinar o valor limite para definir uma baixa correspondência (*low match*) e um valor limite para definir uma alta correspondência (*high match*), como mostra a Fi-

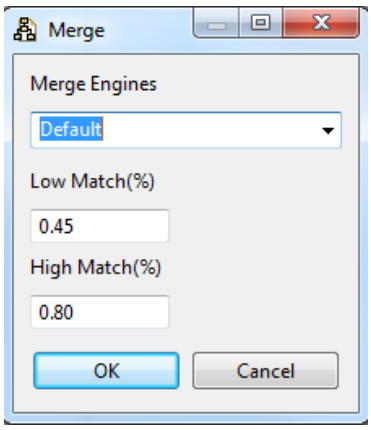


Figura 6.7: Configuração de execução do algoritmo de *database model merging*.

gura 6.7. Para gerar o *database model merging* dos sistemas SISCOLO e SISMAMA, os valores limites atribuídos para o *low match* e *high match* foram de 0.3 e 0.8, respectivamente.

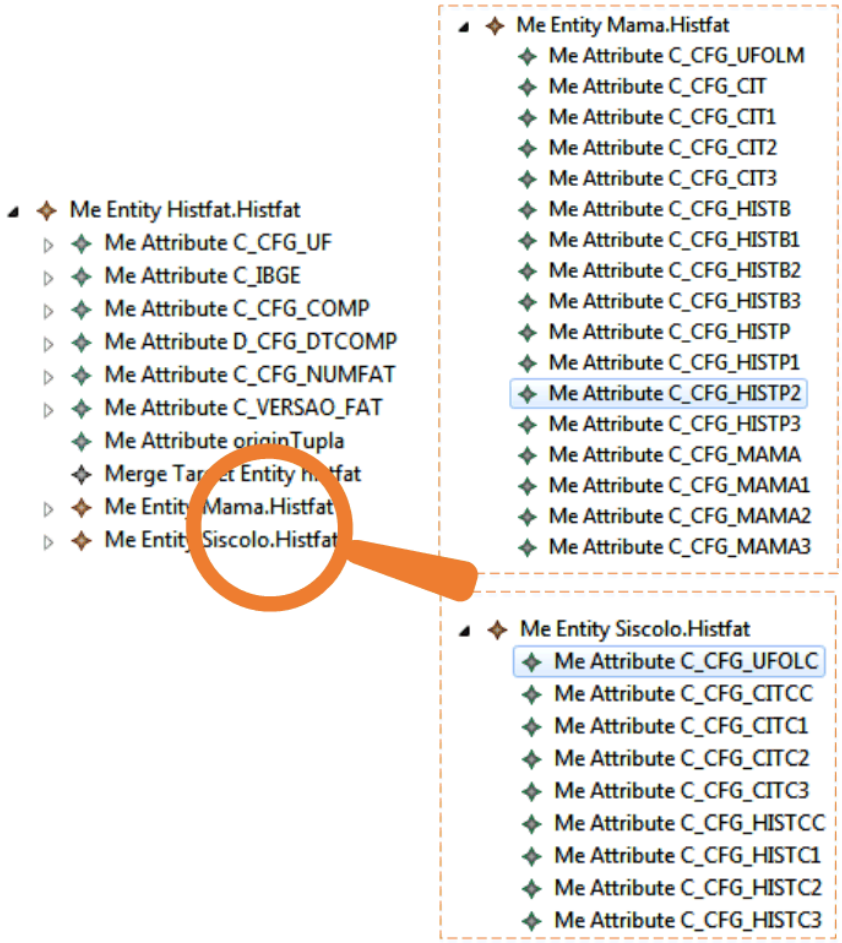


Figura 6.8: Fragmento do *database merging model* dos sistemas SISCOLO e SISMAMA com aplicação da *Generalização de Correspondência*.

A Figura 6.8 mostra um fragmento do *database model merging* dos sistema SISCOLO e SISMAMA, onde foi aplicado a *generalização de correspondência*. A aplicação *generalização de correspondência* gerou três novas entidades: *Histfat.Histfat*, que fundiu os atributos correspondidos entre as entidades *Histfat* de SISCOLO e SISMAMA. A esta entidade foi incluído um novo atributo (*originTupla*) com a finalidade de identificar a origem das tuplas no momento em que registros das entidades de origem forem armazenados nesta entidade; *Mama.Histfat* e *Siscolo.Histfat*, que mantém os atributos não correspondidos.

O *merging* das entidades *Histfat* dos dois *database models* sem a aplicação da *generalização de correspondência* geraria uma nova entidade que teria muitos atributos com valores nulos no momento em que ocorresse a população da entidade. Já com a *generalização de correspondência* atributos comuns populam a entidade *Histfat.Histfat* e atributos não correspondidos populam ou *Mama.Histfat*, ou *Siscolo.Histfat*, dependendo da origem dos registros. Desta forma, os possíveis valores nulos são evitados.

O *database merging model* mantém apenas referências as entidades e atributos dos *database models*. Isto é, este modelo serve de guia para a construção do *database integrated model*.

Criar *database integrated model* dos *database models* SISCOLO e SISMAMA

O *database integrated model* é construído a partir da transformação dos *database models* SISCOLO e SISMAMA e do *database merging model*. Como visto anteriormente, o *database merging model* possui apenas referências aos elementos dos *database models* SISCOLO e SISMAMA. Informações como tipo do atributo, chave primária e secundária são obtidos a partir dos *database models* e incluídos no *database integrated model*.

A Figura 6.9 mostra um fragmento do *database integrated model* gerado pelo protótipo. Pode-se observar as entidades *HistFat.HistFat*, *Mama.Histfat* e *Siscolo.Histfat* que foram criadas conforme definidas no *database merging model*. Diferente do *database merging model*, estas entidades no *database integrated model* possuem agora informações como chave primária e propriedades dos atributos.

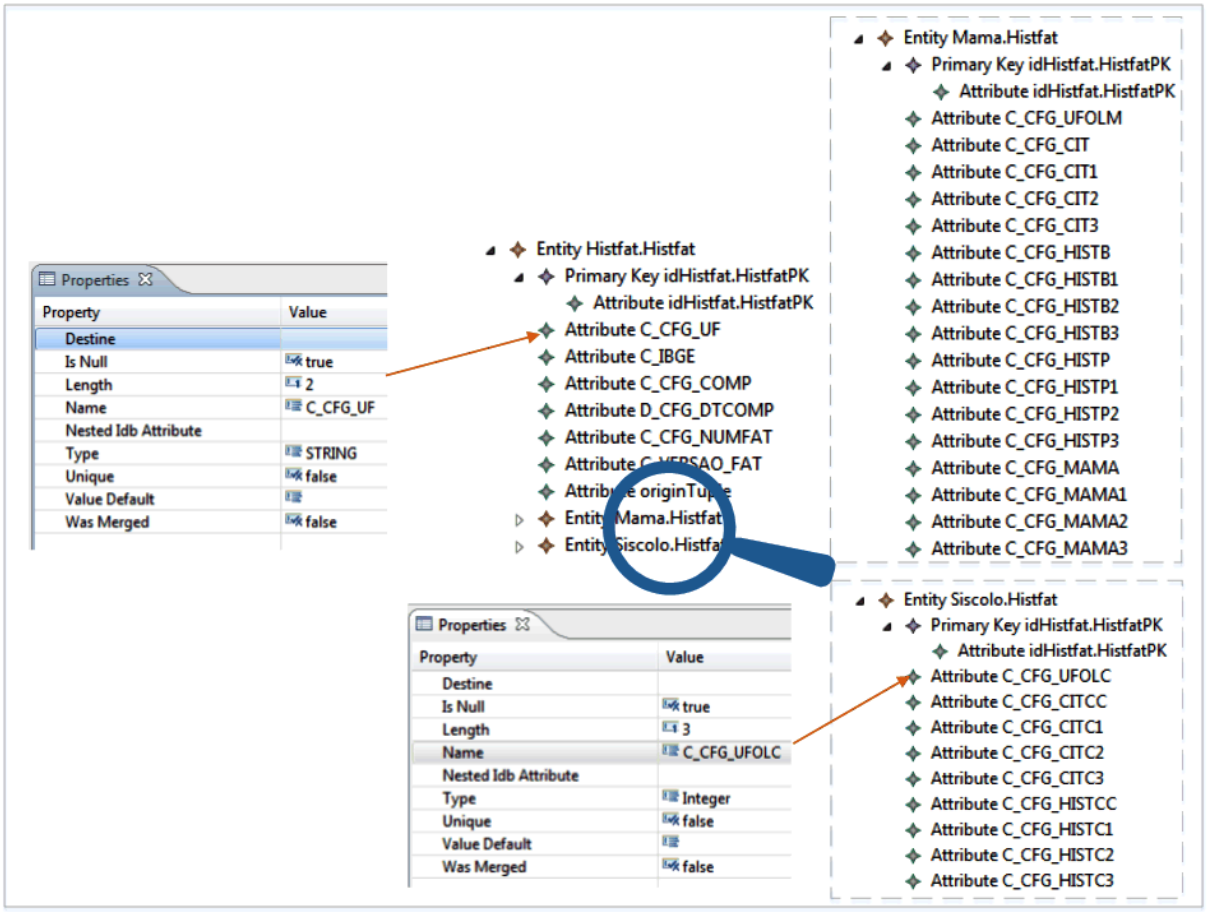


Figura 6.9: Fragmento do *database integrated model* dos sistemas SISCOLO e SISMAMA com aplicação da *Generalização de Correspondência*.

Portanto, o *database integrated model* é composto de todas as informações necessárias para ser transformado em um SQL *script* que representa um *database* com o *merging* dos *database models* SISCOLO e SISMAMA.

Criar SQL *script* do *database integrated model* dos *database models* SISCOLO e SISMAMA

Esta é a etapa final da metodologia proposta neste trabalho. Nesta etapa, o *database integrated model* é transformado em um modelo SQL, que é conforme meta-modelo SQL. As definições de transformações de modelos nesta etapa são escritas em ATL.

A Listagem 6.1 mostra as definições de transformação de elementos do *database integrated model* para um modelo SQL, onde tem-se as transformação de entidade

para tabela, atributo para colunas e chave primária de entidade para chave primária de tabela.

Listagem 6.1: Fragmento de definições de transformação escrita em ATL para transformar *database integrated model* em modelo SQL.

```
1 rule Entity2Table{
2    from entity: Integrado!IdbEntity
3    to table: SQL!Table (
4       name <- entity.name
5    )
6 }
7 rule Attrib2Column{
8    from attrib: Integrado!IdbAttribute
9    to column: SQL!Column(
10       name <- attrib.name,
11       table <- attrib.getOwner(),
12       type <- attrib.getType(),
13       length <- attrib.length
14    )
15 }
16 rule IdbPrimaryKey2PrimaryKey{
17    from pkIdb:Integrado!IdbPrimaryKey
18    to  pk: SQL!PrimaryKey(
19       name <- pkIdb.name,
20       table <- pkIdb.ownerKey
21    )
22 }
```

As definições de transformações da Listagem 6.1 gerar o arquivo *integrado4Sql.idb2sql* com o modelo SQL dos *database* dos sistemas SISCOLO e SISMAMA integrado. Agora, o próximo passo é transformar este modelo em um SQL *script*. Para isso, novas definições de transformações em ATL são escritas para transformar um modelo SQL em um SQL *script*. A Listagem 6.2 ilustra as definições de transformação em ATL que geram o SQL *script* do modelo SQL gerado a partir do *database integrated model*.

Listagem 6.2: Definições de transformação para transformar modelo SQL em SQL *script*.

```
1 library x_SqlToCode;
2
3 helper context SQL!Table def : toString() : String =
4    'DROP TABLE IF EXISTS ' + self.name + ';\n\n' +
5    'CREATE TABLE ' + self.name + ' (' +
```

```
6     self.elements -> iterate(i; acc : String = '' | acc + i.toString() + ' ' )+
7     '\n);';
8
9  helper context SQL!ForeignKey def: getNameAttribFk: String =
10    self.column -> collect(col | col.name);
11
12 helper context SQL!Column def : toString() : String =
13    '\n'+self.name+' '
14    +if self.length > 0 then
15       self.type + '('+self.length+')'
16     else
17       self.type
18     endif
19    +',';
20
21 helper context SQL!PrimaryKey def : toString() : String =
22    '\nPRIMARY KEY('+self.name+ ')';
23
24 helper context SQL!ForeignKey def : toString() : String =
25    ',\nCONSTRAINT `'+self.name+'` FOREIGN KEY (`' +
26        self.getNameAttribFk+ ') REFERENCES ' +self.tabelaReferencia +' ('+self.↩
            getNameAttribFk+')';
```

Por fim, tem-se o SQL *script* gerado com a estrutura do *database* integrado dos sistemas SISCOLO e SISMAMA. A Figura 6.10 mostra a lista de arquivos que foram gerados pela execução das definições de transformação do modelo SQL em SQL *script*. Pode-se observar que para cada entidade do *database integrated model* foi gerado um arquivo com o SQL *script* que cria sua respectiva tabela em um *dabatabase* MySql.

A Figura 6.11, mostra o SQL *script* gerado da entidade município do *database integrated model*.

6.2 Avaliação dos Resultados

A obtenção de um SQL *script* do *merging* de dois *database schemas* utilizando o protótipo do *Framework* para Suportar a Integração de *Database* foi realizado com intuito de demonstrar a viabilidade de uso dos metamodelos, metodologia e algoritmos propostos nesta pesquisa.

Apesar da necessidade de ajustes inicias para gerar o *merging* dos *database models* SISCOLO e SISMAMA, como por exemplo, a definição de prefixos e posfixo dos dois modelos, o ajuste de valores de limite de similaridade de *string* e de limite para

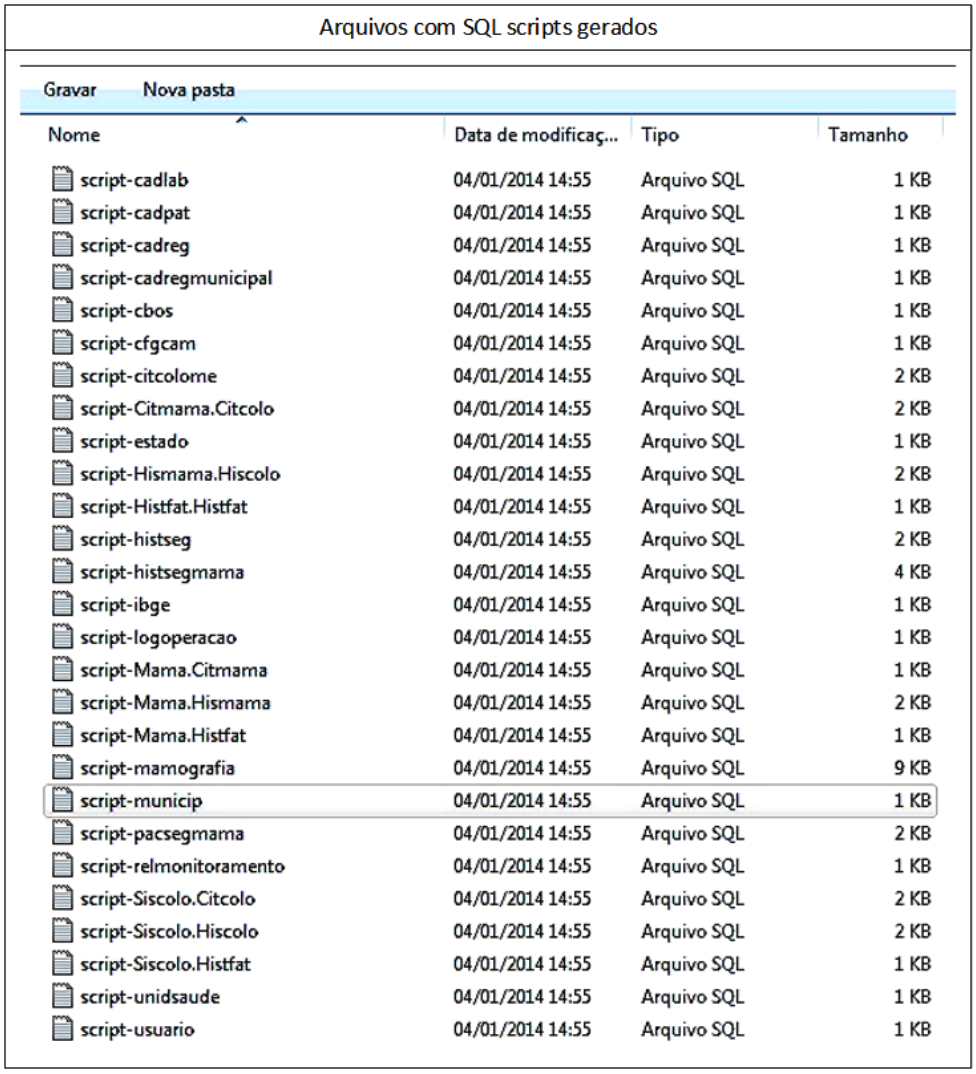


Figura 6.10: Arquivos com SQL *script* gerados pela transformação do modelo SQL.

definir se duas entidades são similares. O processo de *merging* dos modelos com o uso do protótipo se mostrou menos dispendioso que se a tarefa fosse realizada de forma manual. Conclui-se isto, devido aos seguintes aspectos:

- *Recurso oferecido pelo protótipo de importar os database schema diretamente do database*. Este recurso evitar que o especialista de domínio tenha que criar manualmente os *database models* que serão integrados;
- *O recurso de sugestão de elementos correspondentes entres os database models*. A busca por elementos correspondentes entre dois modelos é a tarefa mais trabalhosa no processo de se obter um modelos unificado. A possibilidade de ajustes no processo de *matching*, permite ao especialista combinar estratégias de *match* para encontra elementos correspondentes.;

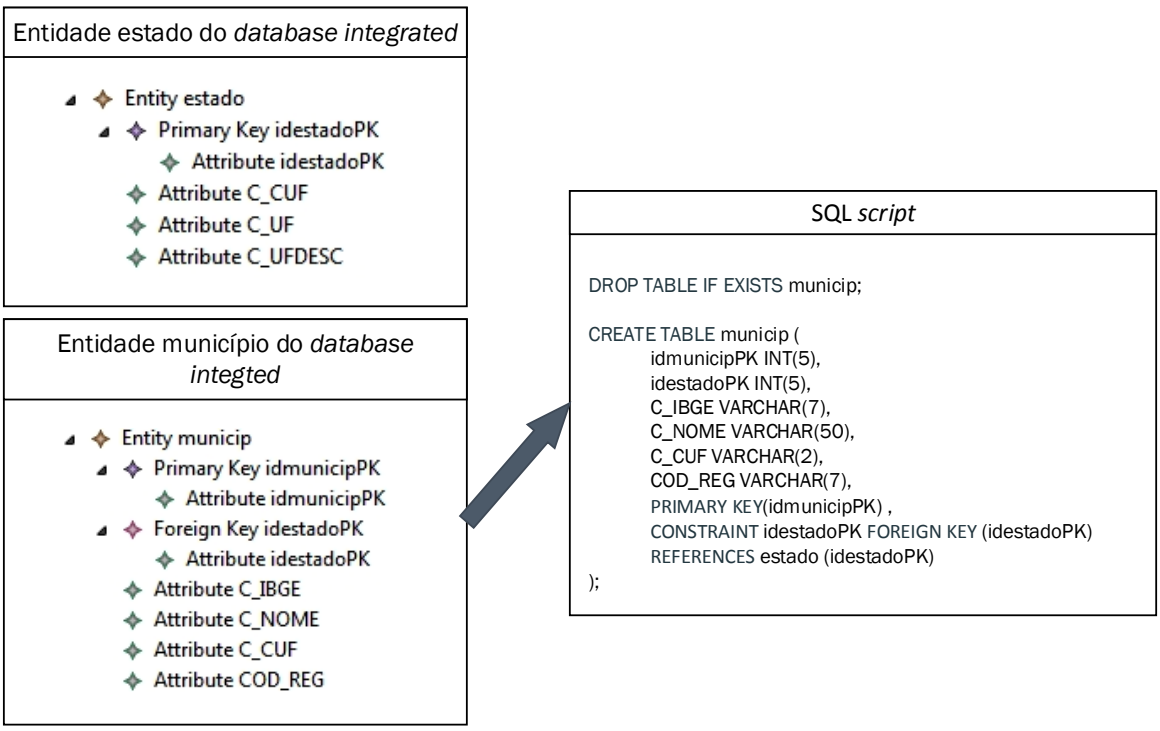


Figura 6.11: Resultado da transformação do *database integrated model* em SQL *script*.

- *O recurso de gerar o merge apenas de algumas entidades do database models*. O protótipo fornece a opção de escolha de quais entidades devem compor o *database model merging*. Assim, a fusão de apenas algumas entidades é possível;
- *Finalmente, o recurso de gerar o SQL script*. Isto evita que o especialista escreva instruções SQL para construir um *database schema*.

Outro ponto a ser mencionado, é que o *database integrated metamodel* permite a criação de um *database integrated model* que pode ser transformado em outras linguagens além do SQL *script*, como por exemplo, a linguagem XML. Para isso, basta estender a interface *ITFLanguage* com o metamodelo específico.

A aplicação do protótipo a um ambiente real, mostrou também a necessidade de melhoramento no algoritmo de *database model matching*. Alguns ajustes de parâmetros resultaram em um *Recall* como muitos *falsos positivos*, o que demandou um pouco mais de trabalho na tarefa de validação das correspondências sugeridas pelo algoritmo de *database model matching*.

Para avaliar a *Generalização de Correspondências* proposta neste trabalho, as tabelas *Histfat.Histfat*, *Siscolo.Histfat* e *Sismama.Histfat* do *database* integrado foram po-

puladas com um conjunto fictícios de registros. As tabelas HISTFAT dos sistemas SISCOLO e SISMAMA também foram populadas com os mesmo conjuntos de registros.

Com as tabelas populadas, foram aplicadas algumas consultas SQL as tabelas SISCOLO e SISMAMA. Estas mesmas consultas foram aplicadas a tabela *Histfat* fundida no *database* integrado, para avaliar se os registros resultantes da consulta foram os mesmo das tabelas de origem.

A Figura 6.12(a) e a Figura 6.12(b) mostram os registros que foram inseridos para a avaliação nas tabelas *HISTFAT* dos sistemas SISCOLO e SISMAMA. Algumas colunas das tabelas foram suprimidas devido ao problema de falta de espaço para sua exibição neste documento.

C_CFG_UF	C_IBGE	C_CFG_COMP	D_CFG_DTCOMP	C_CFG_NUMFAT	C_CFG_UFOLC	C_CFG_CITCC	C_CFG_CITC1	C_CFG_CITC2	C_CFG_CITC3	C_CFG_HISTCC	C_VERSAO_FAT
1	1	201109	01/09/2012	1	1	1	1	1	1	1	1
1	2	201210	02/10/2012	2	2	2	2	2	2	2	1
2	3	201210	03/10/2012	3	3	3	3	3	3	3	1
1	4	201210	04/10/2012	4	4	4	4	4	4	4	1
1	6	201210	06/10/2012	6	6	6	6	6	6	6	1

(a) Tabela HISTFAT do sistema SISCOLO..

C_CFG_UF	C_IBGE	C_CFG_COMP	D_CFG_DTCOMP	C_CFG_NUMFAT	C_CFG_UFOLM	C_CFG_CIT	C_CFG_CIT1	C_CFG_CIT2	C_VERSAO_FAT
1	1	201301	01/10/2013	1	1	1	1	1	1
2	2	201301	02/10/2013	2	2	2	2	2	2
2	3	201301	03/10/2013	3	3	3	3	3	3

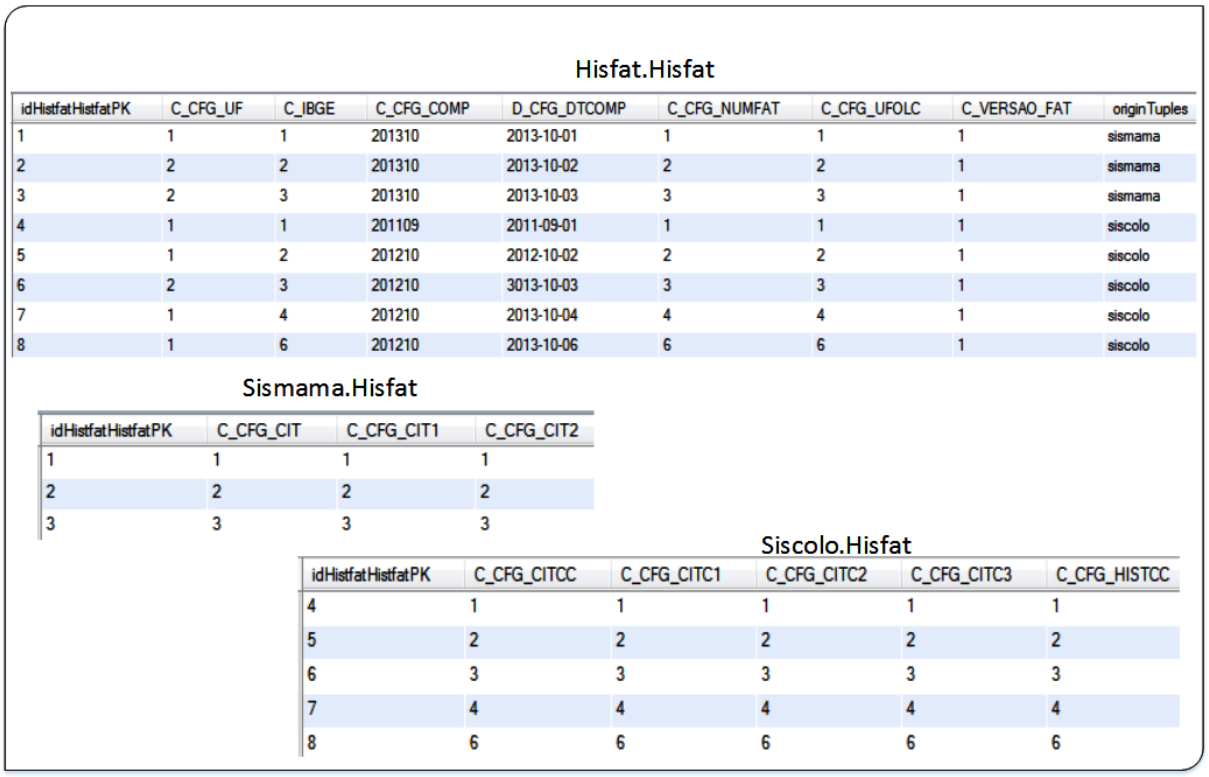
(b) Tabela HISTFAT do sistema SISMAMA.

Figura 6.12: Tabelas populadas dos sistemas SISCOLO e SISMAMA.

A Figura 6.13 mostra como o *merging* da tabela *Histfat* foi estruturado no *database* integrado. A estrutura corresponde exatamente ao que foi proposto pelo algoritmo de *database model merging*. Para avaliar o *merging* das tabelas *Histfat*, as consultas SQL foram executadas para as tabelas de origem (*Histfat* de SISMAMA e SISCOLO), bem como, para as tabelas do *database* integrado e seus resultados foram comparados.

O *Firebird* é o banco de dados original dos dois sistemas [57]. Para esta avaliação, os esquemas dos bancos de dados do SISCOLO e SISMAMA foram importados para o MYSQL e as consultas SQL foram executada no MYSQL *WorkBench*. A Figura 6.12(a) e Figura 6.12(b) mostram os registros incluídos a cada tabela para uso nesta avaliação.

A consulta realizada na tabela HISTFAT (SISMAMA) tem a finalidade de encontrar os registros que possuem C_CFG_UF=2. A Figura 6.14 mostra o resultado da consulta e os registros que foram retornados. O objetivo é obter o mesmos registros



Hisfat.Hisfat

idHistfatHistfatPK	C_CFG_UF	C_IBGE	C_CFG_COMP	D_CFG_DTCOMP	C_CFG_NUMFAT	C_CFG_UFOLC	C_VERSAO_FAT	originTuples
1	1	1	201310	2013-10-01	1	1	1	sismama
2	2	2	201310	2013-10-02	2	2	1	sismama
3	2	3	201310	2013-10-03	3	3	1	sismama
4	1	1	201109	2011-09-01	1	1	1	siscolo
5	1	2	201210	2012-10-02	2	2	1	siscolo
6	2	3	201210	3013-10-03	3	3	1	siscolo
7	1	4	201210	2013-10-04	4	4	1	siscolo
8	1	6	201210	2013-10-06	6	6	1	siscolo

Sismama.Hisfat

idHistfatHistfatPK	C_CFG_CIT	C_CFG_CIT1	C_CFG_CIT2
1	1	1	1
2	2	2	2
3	3	3	3

Siscolo.Hisfat

idHistfatHistfatPK	C_CFG_CITCC	C_CFG_CITC1	C_CFG_CITC2	C_CFG_CITC3	C_CFG_HISTCC
4	1	1	1	1	1
5	2	2	2	2	2
6	3	3	3	3	3
7	4	4	4	4	4
8	6	6	6	6	6

Figura 6.13: Tabelas do *database* integrado dos sistemas SISCOLO e SISMAMA com aplicação da *Generalização de Correspondência*.

C_CFG_UF	C_IBGE	C_CFG_COMP	D_CFG_DTCOMP	C_CFG_NUMFAT	C_CFG_UFOLM	C_CFG_CIT	C_CFG_CIT1	C_CFG_CIT2	C_VERSAO_FAT
02	0000002	201301	2013-10-02	2	002	00002	00002	00002	00002
02	0000003	201301	2013-10-03	3	003	00003	00003	00003	00003

```
1 SELECT    `histfat`.`C_CFG_UF`,     `histfat`.`C_IBGE`,
2       `histfat`.`C_CFG_COMP`,   `histfat`.`D_CFG_DTCOMP`,
3       `histfat`.`C_CFG_NUMFAT`,`histfat`.`C_CFG_UFOLM`,
4       `histfat`.`C_CFG_CIT`,    `histfat`.`C_CFG_CIT1`,
5       `histfat`.`C_CFG_CIT2`,   `histfat`.`C_VERSAO_FAT`
6 FROM `mama`.`histfat`
7 WHERE C_CFG_UF = '02';
```

Figura 6.14: Resultado da consulta SQL a tabela HISTFAT do sistema SISMAMA.

na consulta feita ao *database* integrado. A Figura 6.15 mostra o retorno da consulta anterior, que agora foi realizada na tabela SISMAMA.HISTFAT do *database* integrado. Os mesmos registros retornados na consulta a tabela HISTFAT do sistema SISMAMA, foram retornados na consulta realizada no *database* integrado.

Umas das características que deve ser observada para que um esquema mediado seja considerado bem formatado é que consultas realizadas em um *database* in-

idHistfatHistfatPK	C_CFG_UF	C_IBGE	C_CFG_COMP	D_CFG_DTCOMP	C_CFG_NUMFAT	C_CFG_UFOLC	C_CFG_CIT	C_CFG_CIT1	C_CFG_CIT2	C_VERSAO_FAT
2	2	2	201310	2013-10-02	2	2	2	2	2	1
3	2	3	201310	2013-10-03	3	3	3	3	3	1

```
1 SELECT `mamahistfat`.`idHistfatHistfatPK`, `histfathistfat`.`C_CFG_UF`, `histfathistfat`.`↩
    C_IBGE`,`histfathistfat`.`C_CFG_COMP`, `histfathistfat`.`D_CFG_DTCOMP`, `histfathistfat↩
    `.`C_CFG_NUMFAT`, `histfathistfat`.`C_CFG_UFOLC`,`mamahistfat`.`C_CFG_CIT`, `mamahistfat↩
    `.`C_CFG_CIT1`, `mamahistfat`.`C_CFG_CIT2`, `histfathistfat`.`C_VERSAO_FAT`
2 FROM histfathistfat JOIN  mamahistfat
3 ON `histfathistfat`.`idHistfatHistfatPK` = `mamahistfat`.`idHistfatHistfatPK`
4 where `histfathistfat`.`C_CFG_UF`='2';
```

Figura 6.15: Resultado da consulta SQL a tabela SISMAMA.HISTFAT do *database* integrado.

tegrado deve retornar os mesmo registros de consultas realizadas em um database fonte [51]. Portanto, a proposta da *Generalização de Correspondência* se mostrou viável para o *merge* de tabelas com correspondência parcial de colunas, já que, a consulta realizada no *database* integrado, obteve os mesmos registros da consulta realizada na *database* fonte.

Além disso, a *Generalização de Correspondência* evitou que o *merging* das tabelas HISTFAT de SISCOLO e SISMAMA apresentasse uma quantidade de registros com colunas nulas na inserção de registros.

6.3 Síntese

Neste capítulo, a aplicação do protótipo do *Framework* para Suportar *Database Model Merging* foi apresentada. Com a aplicação do protótipo a uma situação real, constatou-se a viabilidade dos metamodelos, algoritmos e metodologia proposta nesta pesquisa.

A avaliação do protótipo for realizada com os *databases* dos sistema SISCOLO e SISMAMA, que tem a finalidade de acompanhar resultados de exames de câncer de colo do útero e mama de pacientes atendidos pelo Sistema Único de Saúde.

Os modelos gerados a cada etapa da metodologia proposta foram apresentados e avaliados. Além disso, consultas SQL foram realizadas ao *database* integrado gerado a partir dos *database* dos sistemas SISCOLO e SISMAMA, avaliando a viabilidade do uso do protótipo em uma situação real.

7 CONCLUSÕES E TRABALHOS FUTUROS

Este capítulo apresenta os objetivos alcançados com a pesquisa e suas limitações. Além disso, contribuições científica e sociais e os trabalhos futuros são apresentados.

7.1 Conclusões do trabalho

Este trabalho apresentou uma proposta de solução de *merging* de *database schemas* através da abordagem MDE. Para esta tarefa, um *framework* foi proposto e uma metodologia foi apresentada para auxiliar o seu desenvolvimento e aplicação.

Um metamodelo para instanciar *database model* foi desenvolvido com a finalidade de trazer para o espaço tecnológico da MDE esquemas de base de dados relacional, permitindo assim, a sua manipulação no contexto MDE.

Outros metamodelos apresentados neste trabalho foram: o *mapping metamodel* pra instanciar modelos com o mapeamento entre elementos de *database models*; o *database merging model*, que instancia um modelo com as entidades que sofreram *merging*; o *database integrated metamodel* que tem como finalidade criar um modelo com todas as informações necessárias para ser transformado em um SQL *script*.

Além dos metamodelos, o algoritmo de *database model matching* que instancia o *mapping model* e o algoritmo de *database model merging* que constrói um modelo com o *merging* de *database models* foram apresentados.

Um protótipo foi construído para a avaliação do *framework*, juntamente com seus metamodelos e algoritmos. O protótipo foi aplicado a sistemas reais para validar a proposta de *database schema merging* no contexto MDE.

7.2 Objetivos alcançados

O principal objetivo proposto do trabalho de pesquisa é a criação de um *framework*, baseado na MDE, que apresente uma solução viável para obter um *database* integrado de *database schema* heterogêneos.

Para alcançar este objetivo alguns objetivos específicos foram propostos, como visto na seção 1.4.1. Os seguintes objetivos foram alcançados no decorrer do trabalho de pesquisa:

- Implementar um *framework* que suporte o *merging* de *database* heterogêneos utilizando a abordagem MDE. Este objetivo especifico foi alcançado com a definição de metamodelos e algoritmos de *matching* e *merging* que suportam o *merging* de *database models*. Além disso, foi construído um protótipo para avaliação do *framework*;

- Aplicar o *framework* no domínio dos sistemas do SUS. O protótipo desenvolvido foi aplicado aos sistemas SISCOLO e SISMAMA que fazem parte dos sistemas de gerenciamento dos serviços de saúde fornecidos pelo SUS, apresentando um resultado satisfatório no *merge* dos *database* dos dois sistemas.

- Implantar a base de dados integrada em uma plataforma de computação em nuvem. Os SQL *scripts* gerados pelo SID foram executados em uma nuvem *Eucalyptus* com uma maquina virtual Ubuntu com o MySql instalado.

A aplicação do protótipo do *framework* proposto ao ambiente real demonstrou uma diminuição de tempo necessário para integrar dois *database*. No exemplo ilustrativo algumas tabelas tinham quase 100 colunas. Comparar estas tabelas de forma manual certamente exigiria do especialista de domínio um tempo maior. Com o protótipo este tempo foi reduzido, permitindo ainda, que o especialista valide a correspondência sugerida. Assim, acredita-se que o *framework* alcançou o objetivo de ser uma ferramenta que auxilia o especialista de domínio no processo de *database merging*.

A possibilidade de ajuste de parâmetros na tarefa de *match* e *merge* fornece ao especialista de domínio o recurso de adequar o algoritmo de *database model matching* de acordo com os modelos que serão fundidos. Modelos mais heterogêneos podem ter

uma configuração do algoritmo diferente da configuração destinada a modelos menos heterogêneos.

A *generalização de correspondência* proposta para minimizar o surgimento de valores nulos de atributos quando duas entidades são fundidas se mostrou viável como visto na seção 6.2. Esta proposta prepara a estrutura da base de dados integrada para receber os registros das bases de dados fonte e alvo tratando o problema de atributos nulos na fusão de tabelas similares.

Os critérios que foram utilizados para avaliar os trabalhos relacionados na seção 3.3 serão retomados agora para uma avaliação da abordagem para suportar *database merging* no contexto MDE descrita nesta dissertação. O *framework* proposto atende os seguintes critérios:

1. *Utiliza o paradigma MDE para abordar o problema de merge de esquema*. A proposta apresentada nesta dissertação esta centrada na Engenharia Dirigida por Modelos. Todo o trabalho de construção do *framework* levou em consideração os conceitos abordados pela metodologia de desenvolvimento de *software* da MDE;

2. *Permite realizar o match e/ou merge de diferente linguagens de esquema(XML, Relacional, Ontologia)*. A motivação do trabalho de pesquisa foi procurar uma solução para integrar *database* de sistemas fornecidos pelo Sistema Único de Saúde - SUS. Por isso, os estudos foram concentrados apenas em propor uma solução para integração esquemas relacionais. Assim, o *framework* não trata do problema de *merge* de esquema XML e Ontologias;

3. *Uso de informações auxiliares para determinar a similaridade entre elementos dos esquemas*. O *framework* propõe o uso de dicionário de domínio, bem como, o uso de supressão de prefixo e sufixo para melhorá a precisão do processo de *match*;

4. *Permite match e/ou merge de fragmento de esquema*. O recurso de validar as sugestões de *match* e *merge*, permite que o especialista de domínio escolha como será composto o modelo de mapeamento de correspondência e o modelo de *merge*. Assim, caso seja de interesse do especialista apenas fragmentos dos *database models* podem ser integrados;

5. *Trata o surgimento de valores nulos quando esquemas relacionais são fundidos*. O trabalho propões a *Generalização de Correspondência* para tratar este problema. A

Generalização de Correspondência se mostrou uma solução viável para evitar atributos com valores nulos quando entidades sofrem *merge*, isto foi demonstrado na aplicação do *framework* a *database* de aplicações do SUS.

6. *Permite o uso de diferentes algoritmos de match e merge*. A arquitetura do *framework* proposta está preparada para aplicar diferentes algoritmos de *matching* e *merging*, bastando para isso desenvolver novas classes que estendam a interfaces *ITFMatch* e *ITFMerge*. O mesmo pode ser feito com a classe que gerar o modelo integrado, basta o especialista estender a interface *ITFIntegrated*;

7. *Gera o script do merge dos esquemas*. O produto final do *framework* é um SQL *script* que pode ser executado em um DBMS, criando assim, a estrutura de um *database* que permite uma visão unificada de *database* heterogêneos.

A Tabela 7.1 mostra os critérios atendidos pelo *framework* apresentado neste trabalho de pequisa.

7.3 Limitações

Embora o trabalho de pesquisa tenha alcançado seu objetivo principal, alguns pontos apresentam limitações podendo ser melhorados ou ampliados. Estes pontos são:

- Atualmente, os *database models* podem ser intanciados manualmente ou importando o esquema da base de dados diretamente do DBMS. Outros mecanismos de obtenção do *database model* podem ser incorporados ao *framework* como: importar de um SQL *script*, ou de um XML *schema* ou de um Diagrama UML;

- O *framework* realiza as transformações de modelos usando EMF. Outras linguagens de transformações de modelos poderiam ser utilizadas para esta tarefa como a ATL;

- O *database model* instanciado de forma manual utiliza o editor de modelos padrão do EMF. Este editor permite a edição de modelos em forma de árvore. Um ambiente com elementos gráficos tornaria a tarefa de construção do *database models* mais intuitiva;

Itens Avaliados	SID	[16]	[25]	[41]	[43]	[38]
Utiliza o paradigma MDE para abordar o problema de *merge* de esquema	sim	sim	não	não	não	não
Permite realizar o *match* e/ou *merge* de diferente linguagens de esquema(XML, Relacional, Ontologia)	não	sim	não	sim	sim	sim
Uso de informações auxiliares para determinar a similaridade entre elementos dos esquemas	sim	sim	não	sim	não	sim
Permite *matching* e/ou *merge* de fragmento de esquema	sim	não descrito no artigo	não descrito no artigo	sim	sim	não descrito no artigo
Trata o surgimento de valores nulos quando esquemas relacionais são fundidos	sim	não descrito no artigo	sim	não descrito no artigo	não descrito no artigo	não descrito no artigo
Permite o uso de diferentes algoritmos de *matching* e *merging*	sim	sim	não	sim	sim	não
Gera o *script* do *merge* dos esquemas	sim	sim. Definições em ATL	não	não	sim	não

- Uma Abordagem Baseada na Engenharia Dirigida por Modelos para Suportar *Merging* de Modelos de Bases de Dados (SID4MDE)
- *Semi-automatic Model Integration using Matching Transformations and Weaving Models* [16]
- *Tuned Schema Merging* (TuSMe) [25]
- *Evolution of the COMA Match System* [41]
- *Rondo: A Programming Plataform for Generic Model Management* [43]
- *Generic Schema Matching with Cupid* [38]

Tabela 7.1: Comparação do trabalho de pesquisa com os trabalhos relacionados.

- Como já mencionado, determinadas configurações no processo de *matching* produz um alto *Recall*, tornando a tarefa de validação de sugestões do algoritmo de *database model matching* mais demorada. Assim, uma melhora do algoritmo de *database model matching* pode tornar mais preciso os *merge* entre *database models*;
- O modelo de dicionário de domínio utilizado para encontrar similaridade de *string*, também é construído de forma manual pelo especialista de domínio. Este dicionário poderia ser instanciado através de um mecanismo de recuperação de informação em documentos sobre os sistemas que participam do processo de *matching*.

7.4 Contribuições

Nesta seção, contribuições do trabalho de pesquisa no campo científico e no campo social são apresentados.

7.4.1 Científicas

Acredita-se que a pesquisa obteve as seguintes contribuições científicas:

- Aplicação e avaliação da abordagem MDE no problema que envolve o *merging* de *database* heterogêneos;
- Proposta de algoritmos de *matching* e *merging* para integrar *database models*;
- Desenvolvimento de um protótipo que implementa o *framework* proposto neste trabalho de pesquisa. O protótipo foi construído com o projeto *Eclipse Modeling Framework* - EMF, contribuindo assim, para a disseminação do conhecimento do uso de ferramentas que utilizam a abordagem de desenvolvimento de *software* dirigido por modelos;
- Propostas de metamodelo de *database*, metamodelo de *matching*, metamodelo de *merging* e metamodelo de *database integrated* que dão suporte ao processo de *merge* de *database models* no contexto MDE;
- Publicação do artigo *A Framework Based on Model Driven Engineering to Support Schema Merging in Database Systems* sobre o trabalho de pesquisa descrito nesta dissertação na *International Joint Conferences on Computer, Information, and Systems Sciences, and Engineering* - CISSE 2013.

7.4.2 Sociais

A motivação para o desenvolvimento da pesquisa surgiu de uma necessidade vivenciada no órgão municipal gestor da saúde de Teresina (PI). Muitos profissionais de saúde do município estão constantemente buscando informações nos sistemas fornecidos pelo SUS para planejamento das ações de saúde no município. Acontece que o cruzamento de informações entre os sistemas de saúde fornecidos pelos SUS

sempre foi muito difícil, isto porque, os sistemas não possuem suas bases de dados adequadas para uma integração.

Isto exposto, acredita-se que a principal contribuição da pesquisa é fornecer uma ferramenta que auxilia o especialista de domínio a integrar base de dados dos sistemas dos SUS, dando assim, uma visão unificada de informações de agravo da população do município. Isto irá ajudar aos profissionais e gestores de saúde acompanhar ocorrências de agravos nos munícipes, agilizando a tomada de decisões e ações de saúde de devem ser aplicadas em casos de ocorrência de um agravo.

7.5 Trabalhos futuros

Nesta seção, os trabalhos futuros e melhoramento que podem surgir a partir desta pesquisa são apresentados . Os trabalhos futuros que podem ser derivados desta pesquisa são os seguintes:

- Aperfeiçoar o *plug-in* do eclipse desenvolvido com os recursos: um ambiente para edição de modelos de forma gráfica; incluir um *wizard* para importação de esquemas de base de dados; criar um ambiente de configuração dos processo de *matching* e *merging*;
- Melhorar o algoritmo de *database model matching* com novas técnicas de correspondências, como *match* por vizinhança [61];
- Complementar o *framework* com uma abordagem MDE para popular, com os registros dos *databases* fonte e alvo, o *database integrated model* gerado pelo *framework*. Esta abordagem teria como objetivo a criação de um SQL *script* ou rotina que populasse o *database* integrado evitando a duplicação de registros ;
- Incluir a opção de usar ATL para realizar a transformação de modelos. Atualmente a ATL foi utilizada apenas na transformação do *database integrated model* para o modelo SQL e do modelo SQL para o SQL *script*;

Referências Bibliográficas

[1] ALOUINI, W., GUEDHAMI, O., HAMMOUDI, S., GAMMOUDI, M., , D., AND ESSAIT, E. S. D. S. D. Semi-automatic generation of transformation rules in model driven engineering: The challenge and first steps. *International Journal of Software Engineering and Its Applications 5*, 1 (2011), 73–88.

[2] APACHE. Apache software foundation, 2013. Disponível em http://www.apache.org/. Acessado em 01/07/2013.

[3] ATL. Atlas transformation language. Disponível em http://www.eclipse.org/atl/. Acessado em 10/2013.

[4] BASSANI, J. Uma abordagem baseada em engenharia dirigida por modelos para suportar o teste de sistemas de software na plataforma de computação em nuvem. Master's thesis, Universidade Federal do Maranhão, 2012.

[5] BATINI, C., LENZERINI, M., AND NAVATHE, S. B. A comparative analysis of methodologies for database schema integration. *ACM Comput. Surv. 18*, 4 (Dec. 1986), 323–364.

[6] BERNSTEIN, P. A. Applying model management to classical meta data problems. In *CIDR* (2003).

[7] BERNSTEIN, P. A., GREEN, T. J., MELNIK, S., AND NASH, A. Implementing mapping composition. In *VLDB* (2006), pp. 55–66.

[8] BERNSTEIN, P. A., MADHAVAN, J., AND RAHM, E. Generic schema matching, ten years later. *PVLDB 4*, 11 (2011), 695–701.

[9] BERNSTEIN, P. A., AND MELNIK, S. Model management 2.0: manipulating richer mappings. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 2007), SIGMOD '07, ACM, pp. 1–12.

[10] BEZERRA, E. D. C. Composição dinâmica de serviços web semânticos utilizando abordagens da engenharia dirigida por modelos. Master's thesis, Universidade Federal do Maranhão, 2011.

[11] BÉZIVIN, J. Model driven engineering: An emerging technical space. In *Generative and Transformational Techniques in Software Engineering*, R. Lammel, J. Saraiva, and J. Visser, Eds., vol. 4143 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2006, pp. 36–64.

[12] BUDINSKY, F. *Eclipse modeling framework: a developer's guide*. The eclipse series. Addison-Wesley, 2004.

[13] COHEN, W. W., RAVIKUMAR, P. D., FIENBERG, S. E., ET AL. A comparison of string distance metrics for name-matching tasks. In *IIWeb* (2003), vol. 2003, pp. 73–78.

[14] DATASUS. Departamento de informática do sus, 2013. Disponível em http://www2.datasus.gov.br/. Acesso em 23/01/2013.

[15] DAVID AUMUELLER, HONG-HAI DO, S. M., AND RAHM, E. Schema and ontology matching with coma++. In *IN SIGMOD CONFERENCE* (2005), ACM Press, pp. 906–908.

[16] DEL FABRO, M. D., AND VALDURIEZ, P. Semi-automatic model integration using matching transformations and weaving models. In *Proceedings of the 2007 ACM symposium on Applied computing* (New York, NY, USA, 2007), SAC '07, ACM, pp. 963–970.

[17] DIDONET, M., FABRO, D., BÉZIVIN, J., AND VALDURIEZ, P. Weaving models with the eclipse amw plugin. In *In Eclipse Modeling Symposium, Eclipse Summit Europe* (2006).

[18] ECLIPSE, 2013. Eclipse IDE. Disponível em http://www.eclipse.org/. Acessado em 07/2013.

[19] EMF. Eclipse modeling framework project, 2013. Eclipse Modeling Framework Project (EMF). Available at http://www.eclipse.org/modeling/emf/. Acessed at 05/05/2013.

[20] EUCALYPTUS, S. *Eucalyptus Usage Guide 3.0.2*. Eucalyptus Systems, 05 2012.

[21] FABRO, M. D. D., BÉZIVIN, J., JOUAULT, F., AND VALDURIEZ, P. Applying generic model management to data mapping. In *Proceedings of the Journées Bases de Données Avancées (BDA05)* (2005).

[22] FIREBIRD. Relational database firebird. Disponível em http://www.firebirdsql.org/en/server-packages/. Acesso em 10/12/2013.

[23] GROUP, A., LINA, AND INRIA. *Atlas Transformation Language - ATL User Manual version 0.7*, February 2006.

[24] HONG-HAI, MELNIK, S., AND ERHARD, R. Comparison of schema matching evaluations. In *Web, Web-Services, and Database Systems*, vol. 2593 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2003, pp. 221–237.

[25] JABEEN, G., AND MASOOD, N. Tuned schema merging (tusme). In *Software, Knowledge Information, Industrial Management and Applications (SKIMA), 2011 5th International Conference on* (2011), pp. 1–7.

[26] JADEJA, Y., AND MODI, K. Cloud computing - concepts, architecture and challenges. In *Computing, Electronics and Electrical Technologies (ICCEET), 2012 International Conference on* (2012), pp. 877–880.

[27] JAVA. Plataforma java. Disponível em: http://www.oracle.com/technetwork/java/. Acessado em 01/05/2013.

[28] KLEPPE, A., WARMER, J., AND BAST, W. *MDA Explained: The Model Driven Architecture: Practice and Promise*, 1st ed. Addison-Wesley, August 2003.

[29] KURTEV, I., BÉZIVIN, J., AND AKŞIT, M. Technological spaces: An initial appraisal. In *International Conference on Cooperative Information Systems (CoopIS), DOA'2002 Federated Conferences, Industrial Track, Irvine, USA* (October 2002), pp. 1–6.

[30] LAFI, L., FEKI, J., AND HAMMOUDI, S. Metamodel matching techniques evaluation and benchmarking. 1–6.

[31] LOPES, D. *Study and Applications of the MDA Approach in Web Service Platforms*. Ph.D. thesis (written in French), University of Nantes, 2005.

[32] LOPES, D., HAMMOUDI, H., BÉZIVIN, J., AND JOUAULT, F. Generating transformation definition from mapping specification: Application to web service platform. *The 17th Conference on Advanced Information Systems Engineering (CAiSE'05)*, LNCS 3520 (June 2005), 309–325.

[33] LOPES, D., HAMMOUDI, S., AND ABDELOUAHAB, Z. Schema Matching in the Context of Model Driven Engineering: From Theory to Practice. *Proceedings of the International Conference on Systems, Computing Sciences and Software Engineering (SCSS 2005)* (December 2005).

[34] LOPES, D., HAMMOUDI, S., BÉZIVIN, J., AND JOUAULT, F. Mapping Specification in MDA: From Theory to Practice. *First International Conference INTEROP-ESA'2005 Interoperability of Enterprise Software and Applications* (February 2005).

[35] LOPES, D., HAMMOUDI, S., DE SOUZA, J., AND BONTEMPO, A. Metamodel Matching: experiments and comparison. *In IEEE International Conference on Software Engineering Advances (ICSEA 06)* (2006).

[36] LOPES, D. C. P. *Introdução a Engenharia Dirigida por Modelos.* I Escola Regional de Computação Ceará Maranhão Piauí (ERCEMAPI), 2007.

[37] MADHAVAN, J., BERNSTEIN, P. A., DOMINGOS, P., AND HALEVY, A. Y. Representing and reasoning about mappings between domain models. In *Eighteenth national conference on Artificial intelligence* (Menlo Park, CA, USA, 2002), American Association for Artificial Intelligence, pp. 80–86.

[38] MADHAVAN, J., BERNSTEIN, P. A., AND RAHM, E. Generic schema matching with cupid. In *Proceedings of the 27th International Conference on Very Large Data Bases* (San Francisco, CA, USA, 2001), VLDB '01, Morgan Kaufmann Publishers Inc., pp. 49–58.

[39] MAPPER, M. B. Biztalk mapper. Disponível em: http://msdn.microsoft.com/. Acessado 01/12/2013.

[40] MARTIN NECASKY, JAKUB KLÍMEK, J. M., AND MLYNKOVÁ, I. Evolution and change management of xml-based systems. *Journal of Systems and Software 85*, 3 (2012), 683 – 707. <ce:title>Novel approaches in the design and implementation of systems/software architecture</ce:title>.

[41] MASSMANN, S., RAUNICH, S., AUMÜLLER, D., ARNOLD, P., AND RAHM, E. Evolution of the COMA Match System. In *The Sixth International Workshop on Ontology Matching* (Oct. 2011).

[42] MELNIK, S., GARCIA-MOLINA, H., AND RAHM, E. Similarity flooding: A versatile graph matching algorithm and its application to schema matching. In *Proceedings of the 18th International Conference on Data Engineering* (Washington, DC, USA, 2002), ICDE '02, IEEE Computer Society, pp. 117–.

[43] MELNIK, S., RAHM, E., AND BERNSTEIN, P. A. Rondo: a programming plataform for generic model management. 193–204.

[44] MINISTÉRIO, S. Ministério da saúde, 2013. Disponível em: http://portal.saude.gov.br/portal/saude/. Acesso em 23/06/2013.

[45] NAVARRO, G. A guided tour to approximate string matching. *ACM Comput. Surv. 33*, 1 (Mar. 2001), 31–88.

[46] OMG. Object management group. Object Management Group. Disponível em `http://www.omg.org/`, Acesso em 17-05-2013.

[47] OMG. *MDA Guide Version 1.0.1, Document Number: omg/2003-06-01*. OMG, June 2003.

[48] OMG. Uml version 2.3 specification, omg documents formal/2010-05-03, 2010.

[49] OMG. Meta object facility (mof) core specification, 2013.

[50] PEUKERT, E., EBERIUS, J., AND RAHM, E. Amc - a framework for modelling and comparing matching systems as matching processes. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on* (2011), pp. 1304–1307.

[51] POTTINGER, R., AND BERNSTEIN, P. A. Schema merging and mapping creation for relational sources. In *Proceedings of the 11th international conference on Extending database technology: Advances in database technology* (New York, NY, USA, 2008), EDBT '08, ACM, pp. 73–84.

[52] POTTINGER, R., AND BERNSTEIN, P. A. Associativity and commutativity in generic merge. 254–272.

[53] POTTINGER, R. A., AND BERNSTEIN, P. A. Merging models based on given correspondences. 862–873.

[54] RADWAN, A., POPA, L., STANOI, I. R., AND YOUNIS, A. Top-k generation of integrated schemas based on directed and weighted correspondences. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data* (New York, NY, USA, 2009), SIGMOD '09, ACM, pp. 641–654.

[55] RAHM, E., AND BERNSTEIN, P. A. A survey of approaches to automatic schema matching. *The VLDB Journal 10*, 4 (Dec. 2001), 334–350.

[56] RAMEZ ELMASRI, S. B. N. *Sistemas de Banco de Dados*, 2 ed. Pearson Addison Wesley, 2002.

[57] SAÚDE, M. D. *Sistema de informção do controle do câncer de mama (SISMAMA) e do câncer do colo do útero (SISCOLO): manual gerencial*. MINISTÉRIO DA SAÚDE - INSTITUTO NACIONAL DE CÂNCER (INCA), 2010.

[58] SCHMIDT, D. Guest editor's introduction: Model-driven engineering. *Computer 39*, 2 (feb. 2006), 25 – 31.

[59] SOMMERVILLE, I. *Software Engineering*, 8st ed. Pearson, 2007.

[60] SOSINSKY, B. *Cloud Computing Bible*. Wiley Publishing, 2011.

[61] SOUSA, JOSÉ, J., LOPES, D., CLARO, D., AND ABDELOUAHAB, Z. Step forward in semi-automatic metamodel matching: Algorithms and tool. *11th International Conference on Enterprise Information Systems LNBIP* (2009).

[62] STEINBERG, D., BUDINSKY, F., PATERNOSTRO, M., AND MERKS, E. *EMF: Eclipse Modeling Framework*, 2rd ed. Addison-Wesley Professional, 2008.

[63] THOMSON, L. W. L. *Tutorial MySQL - Uma Introdução Objetiva aos Fundamentos do Banco de Dados Mysql*, 1 ed. Ciencia Moderna, 2004.

[64] UKKONEN, E. Approximate string matching with q-grams and maximal matches. Tech. rep., 1991.

[65] UML. Unified modeling language. Disponível em:http://www.omg.org/spec/UML/. Acesso em 10/5/2013.

[66] VELTE, A. T., VELTE, T. J., AND ESLENPETER, R. *Computação em Nuvem. Uma abordagem prática*. Alta Books, 2010.

[67] W3C. extensible markup language. Disponível em:http://www.w3.org/standards/xml/. Acesso em 10/5/2013.

8 Anexos

8.1 Anexo A - Implementação do operador *Match*

```
1  %%
2   % <copyright>
3   % Copyright (c) 2004, Denivaldo LOPES
4   % All rights reserved. This program and the accompanying materials
5   % were designed, created and developped by Denivaldo LOPES (denivaldo_lopes@hotmail.com)
6   % The extension and/or modification of this program and the material must make reference
7   % to them in the following way:
8   %    % This program and the acompanying materials are based on the plug-in for
9   %      Mapping designed, created and developped
10  %      by Denivaldo LOPES (denivaldo_lopes@hotmail.com).
11  %
12  % @Modified by Marcus Vinicius Carvalho, 2013.
13  % </copyright>
14  %
15  %%/
16 package match.engine;
17
18 import java.math.BigDecimal;
19 import java.math.RoundingMode;
20 import java.util.ArrayList;
21 import java.util.Collection;
22 import java.util.Iterator;
23
24 import javax.smartcardio.ATR;
25 import javax.swing.JWindow;
26
27 import mapping.presentation.ITFMatchEngine;
28 import mapping.presentation.MappingEditorPlugin;
29 import match.support.SynDictionary;
30
31 import org.apache.lucene.search.spell.JaroWinklerDistance;
32 import org.apache.lucene.search.spell.LevensteinDistance;
33 import org.apache.lucene.search.spell.NGramDistance;
34 import org.eclipse.emf.common.util.BasicEList;
35 import org.eclipse.emf.common.util.EList;
36 import org.eclipse.emf.ecore.EClass;
37
38 import schemaDataBase.Attribute;
39 import schemaDataBase.Entity;
```

```
40 import schemaDataBase.Postfix;
41 import schemaDataBase.Prefix;
42 import schemaDataBase.Schema;
43 import schemaMatchDB.MAttribute;
44 import schemaMatchDB.MEntity;
45 import schemaMatchDB.MatchDB;
46 import schemaMatchDB.SchemaMatchDBFactory;
47 import schemaMatchDB.SchemaMatchDBPackage;
48 import schemamerging.MeAttribute;
49 import schemamerging.MeEntity;
50 import dicsyn.DicsynFactory;
51 import dicsyn.DicsynPackage;
52 import dicsyn.DomainSynonymous;
53 import dicsyn.SynWord;
54 import dicsyn.Word;
55
56 /%%
57  * @author Denivaldo
58  * @Modified by Marcus Vinicius Carvalho
59  * TODO To change the template for this generated type comment go to Window -
60  * Preferences - Java - Code Style - Code Templates
61  */
62 public class Match implements ITFMatchEngine {
63     float globalAverage = 0.2f;
64     private Schema schemaMa;
65     private Schema schemaMb;
66     protected SchemaMatchDBPackage matchPackage = SchemaMatchDBPackage.eINSTANCE;
67     protected DicsynPackage dicSynPackage = DicsynPackage.eINSTANCE;
68     protected DicsynFactory dicSynFactory = dicSynPackage.getDicsynFactory();
69     protected SynDictionary dicSyn;
70     protected DomainSynonymous domainSynonymous = dicSynFactory.createDomainSynonymous();
71     int method = 0;
72     int functionMethod = 0;
73     float limite = 0;
74     float limiteAttribPerc = 0;

75     protected SchemaMatchDBFactory matchFactory = matchPackage.getSchemaMatchDBFactory();
76     String fileName;
77     MatchDB matchJWtoNG = matchFactory.createMatchDB();
78    MatchDB matchJWtoLS = matchFactory.createMatchDB();
79    MatchDB matchNGtoLS = matchFactory.createMatchDB();
80    MatchDB lsMatchDB = (MatchDB) matchFactory.createMatchDB();
81    MatchDB ngMatchDB = (MatchDB) matchFactory.createMatchDB();
82    MatchDB jwMatchDB = (MatchDB) matchFactory.createMatchDB();
83     public Match() {
84
85     }
86     public void init(Schema schemaMa, Schema schemaMb, int method, DomainSynonymous ↩
           domainSynonymous) throws Exception {
87         this.schemaMa = schemaMa;
88         this.schemaMb = schemaMb;
```

```
89          this.method   = method;
90          this.limite = (float) 0.7;
91          this.limiteAttribPerc = (float) 0.8;
92          this.domainSynonymous = domainSynonymous;
93          this.dicSyn = new SynDictionary(domainSynonymous);
94      }
95
96      @Override
97      public MatchDB match() throws Exception {
98              MatchDB matchDB = (MatchDB) matchFactory.createMatchDB();
99              matchDB.setName(schemaMa.getName().toUpperCase() + " <-> " + schemaMb.getName().↩
                    toUpperCase());
100             BasicEList<Entity> listEntityMa = new BasicEList<Entity>();
101             BasicEList<Entity> listEntityMb = new BasicEList<Entity>();
102             BasicEList<Attribute> listAttributeMa = new BasicEList<Attribute>();
103             BasicEList<Attribute> listAttributeMb = new BasicEList<Attribute>();
104
105             %% carregar  Ma
106
107             for (Iterator<Entity> iterMa = schemaMa.getEntityDatabase().iterator();iterMa.↩
                    hasNext();){
108              Entity entityMa = (Entity) iterMa.next();
109                 listEntityMa.add(entityMa);
110                    for (Iterator iterAtt = entityMa.getEntityAttribute().iterator();iterAtt.↩
                           hasNext();){
111                        schemaDataBase.Attribute attributeMa = (schemaDataBase.Attribute) ↩
                               iterAtt.next();
112                        listAttributeMa.add(attributeMa);
113                    }
114
115             }
116             %%carregar Mb
117
118             for (Iterator iterMb = schemaMb.getEntityDatabase().iterator();iterMb.hasNext();)↩
                    {
119
120              Entity entityMb = (Entity) iterMb.next();
121                 listEntityMb.add(entityMb);
122
123                 for (Iterator iterAtt = entityMb.getEntityAttribute().iterator();iterAtt.↩
                        hasNext();){
124                    schemaDataBase.Attribute attributeMb = (schemaDataBase.Attribute) iterAtt.↩
                           next();
125                    listAttributeMb.add(attributeMb);
126                }
127             }
128             if(method==0){
129              MatchDB mixMatchDB = (MatchDB) matchFactory.createMatchDB();
130              int amountFoundMatch=0;
131
132              functionMethod =1;
```

```
133             matchEntity(listEntityMa, listEntityMb, jwMatchDB);
134             BasicEList<MEntity> jwList = new BasicEList<MEntity>();
135             jwList.addAll(jwMatchDB.getMatchEntity());
136
137             functionMethod =2;
138             matchEntity(listEntityMa, listEntityMb, ngMatchDB);
139             BasicEList<MEntity> ngList = new BasicEList<MEntity>();
140             ngList.addAll(ngMatchDB.getMatchEntity());
141
142             functionMethod =3;
143             matchEntity(listEntityMa, listEntityMb, lsMatchDB);
144             BasicEList<MEntity> lsList = new BasicEList<MEntity>();
145             lsList.addAll(lsMatchDB.getMatchEntity());
146
147             BasicEList<MEntity> matchDBTempAtoC = searchSimilarEntities(lsList,jwList);
148             BasicEList<MEntity> matchDBTempBtoC = searchSimilarEntities(ngList,lsList);
149             BasicEList<MEntity> matchDBTempAtoB = searchSimilarEntities(jwList,ngList);
150
151             BasicEList<MEntity> matchDBList = matchDBTempAtoC;
152
153             matchDBList.addAll(mergingEntitySet(matchDBList, matchDBTempAtoB));
154             matchDBList.addAll(mergingEntitySet(matchDBList, matchDBTempBtoC));
155
156             matchDB.getMatchEntity().addAll(matchDBList);
157            }else{
158             functionMethod = method;
159             matchEntity(listEntityMa, listEntityMb, matchDB);
160            }
161            return matchDB;
162    }
163
164    protected BasicEList<MEntity> searchSimilarEntities(BasicEList<MEntity> listA, BasicEList↩
         <MEntity> listB){
165       BasicEList<MEntity> matchAtoB = new BasicEList<MEntity>();
166       for(int iTempA = 0;iTempA<listA.size();iTempA++){
167          MEntity entityTempA = listA.get(iTempA);
168          for(Iterator<MEntity> iTempB  = listB.iterator();iTempB.hasNext();){
169             MEntity entityTempB = iTempB.next();
170             if(entityTempA.getName().equalsIgnoreCase(entityTempB.getName())){
171                matchAtoB.add(entityTempA);
172             }
173          }
174       }
175       return matchAtoB;
176    }
177
178    protected BasicEList<MEntity> mergingEntitySet(BasicEList<MEntity> listA, BasicEList<↩
         MEntity> listB){
179
180       BasicEList<MEntity> matchAtoB = new BasicEList<MEntity>();
181       BasicEList<MEntity> lA = new BasicEList<MEntity>();
```

```
182       BasicEList<MEntity> lB = new BasicEList<MEntity>();
183       lA.addAll(listA);
184       lB.addAll(listB);
185
186       for(int iTempA = 0;iTempA<lA.size();iTempA++){
187          MEntity entityTempA = lA.get(iTempA);
188
189          for(Iterator<MEntity> iTempB  = lB.iterator();iTempB.hasNext();){
190             MEntity entityTempB = iTempB.next();
191
192             if(entityTempA.getName().equalsIgnoreCase(entityTempB.getName())){
193
194                listB.remove(entityTempB);
195             }
196          }
197       }
198       return listB;
199    }
200
201    protected void createEntityComb(MEntity entity, MEntity mEntityJW){
202       entity.setELeft(mEntityJW.getELeft());
203       entity.setERight(mEntityJW.getERight());
204       entity.setName(mEntityJW.getName());
205       entity.setValidate(mEntityJW.isValidate());
206       for(Iterator<MAttribute> iAtt = mEntityJW.getEntityAttrib().iterator();iAtt.hasNext()↩
              ;){
207          MAttribute mAttribute = iAtt.next();
208          MAttribute attribute = matchFactory.createMAttribute();
209          String aLeftJW = mAttribute.getALeft();
210          String aRightJW = mAttribute.getARight();
211          for(Iterator<MAttribute> iAttribJW = mEntityJW.getEntityAttrib().iterator();↩
                 iAttribJW.hasNext();){
212             MAttribute attributeNG = iAttribJW.next();
213             String aLeftNG = attributeNG.getALeft();
214             String aRightNG = attributeNG.getARight();
215
216             if(aLeftJW.equalsIgnoreCase(aLeftNG) && aRightJW.equalsIgnoreCase(aRightNG)){
217                attribute.setALeft(aLeftJW);
218                attribute.setARight(aRightJW);
219                attribute.setName(mAttribute.getName());
220                attribute.setValidate(mAttribute.isValidate());
221                entity.getEntityAttrib().add(attribute);
222
223             }
224          }
225       }
226    }
227
228     private void matchEntity(BasicEList lEntityMa, BasicEList lEntityMb, MatchDB matchDB) {
229
230        for (Iterator iterEntityMa = lEntityMa.iterator(); iterEntityMa.hasNext();) {
```

```
231
232             Entity entityA = (Entity) iterEntityMa.next();
233
234             for (Iterator iterEntityMb = lEntityMb.iterator(); iterEntityMb.hasNext();) {
235
236                Entity  entityB= (Entity) iterEntityMb.next();
237
238                MEntity mEntity = instaciarEntity(entityA, entityB);
239                double simAttribVal = 0;
240
241                String nameEntityA = removePrePostfix(entityA.getName(), schemaMa);
242                String nameEntityB = removePrePostfix(entityB.getName(), schemaMb);
243                double simEntityName = simString(nameEntityA, nameEntityB, functionMethod);
244                if(simEntityName >=limite){
245                   simAttribVal = matchAttributes(entityA, entityB,mEntity);
246                   if(simAttribVal>=0){
247                      matchDB.getMatchEntity().add(mEntity);
248                   }
249                }else{
250                   simAttribVal = matchAttributes(entityA, entityB,mEntity);
251                   if(simAttribVal>limiteAttribPerc){
252                      matchDB.getMatchEntity().add(mEntity);
253                   }
254                }
255             }
256          }
257       }
258
259       private MEntity instaciarEntity(Entity entityA,Entity entityB){
260
261         MEntity mEnt = (MEntity) matchFactory.create((EClass) matchPackage.getEClassifier("↩
               MEntity"));
262         mEnt.setELeft(entityA.getName());
263         mEnt.setERight(entityB.getName());
264         return mEnt;
265       }
266
267       private double matchAttributes(Entity entityA, Entity entityB, MEntity mEntity) {
268
269         double simAttribName = -1;
270         int totAttSim = 0;
271         double percSim=0.0;
272
273           for (Iterator<schemaDataBase.Attribute> iterAttrA = entityA.getEntityAttribute().↩
               iterator(); iterAttrA.hasNext();) {
274             schemaDataBase.Attribute attrA = (schemaDataBase.Attribute) iterAttrA.next();
275
276             int atributosIguais = 0;
277             for (Iterator iterAttrB = entityB.getEntityAttribute().iterator(); iterAttrB.↩
                   hasNext();) {
278
```

```
279                 schemaDataBase.Attribute attrB = (schemaDataBase.Attribute) iterAttrB.next();
280                 String nameAttribA = removePrePostfix(attrA.getName(), schemaMa);
281                 String nameAttribB = removePrePostfix(attrB.getName(), schemaMb);
282                 simAttribName = simString(nameAttribA, nameAttribB, functionMethod);
283
284                 if (simAttribName == 1 ) {
285                    totAttSim++;
286                    atributosIguais++;
287                    MAttribute mAttr = matchFactory.createMAttribute();
288                    mAttr.setALeft(attrA.getName());
289                    mAttr.setARight(attrB.getName());
290                    mAttr.setName(attrA.getName() + " <-> "+ attrB.getName());
291                    mEntity.getEntityAttrib().add(mAttr);
292                 } else {
293                    if (simAttribName >= limite ) {
294
295                       totAttSim++;
296                       atributosIguais++;
297
298                       MAttribute mAttr = matchFactory.createMAttribute();
299                       mAttr.setALeft(attrA.getName());
300                       mAttr.setARight(attrB.getName());
301                       mAttr.setName(attrA.getName() + " <-> "    + attrB.getName());
302
303                       mEntity.getEntityAttrib().add(mAttr);
304                    }
305                 }
306
307              }
308        }
309           float totalAttrib = (float) entityA.getEntityAttribute().size()+entityB.↩
                  getEntityAttribute().size();
310           float totalSim = 2*(float)totAttSim;
311           percSim = totalSim/totalAttrib;
312
313           return percSim;
314
315     }
316
317     %% busca por sinonimo
318     private int searchSynonym(String a, String b) {
319         String lowerCaseA = a.toLowerCase();
320         String lowerCaseB = b.toLowerCase();
321         Word synonyms = null;
322         try {
323          synonyms = (Word) dicSyn.findSynonymsIgnoreCase(a);
324          if(synonyms!=null){
325             for (Iterator iter = synonyms.getSynwords().iterator(); iter.hasNext();) {
326                SynWord foundterm = (SynWord) iter.next();
327                String lowerCaseSynon = foundterm.getName().toLowerCase();
328                if (lowerCaseSynon.equals(lowerCaseB))
```

```
329                 return 0;
330
331             if (lowerCaseSynon.indexOf(lowerCaseB) >= 0)
332                 return 1;
333
334             if (lowerCaseB.indexOf(lowerCaseSynon) >= 0)
335                 return 1;
336
337           }
338         }
339       } catch (Exception e) {
340           MappingEditorPlugin.INSTANCE.log(e);
341       }
342       return -1;
343   }
344
345   private String removePrePostfix(String a, Schema schema){
346
347     for(Iterator<Prefix> iPrefix = schema.getPrefixes().iterator();iPrefix.hasNext();){
348        Prefix prefix = iPrefix.next();
349        a = a.toLowerCase().replaceFirst(prefix.getPrefix().toLowerCase(), "");
350     }
351     for(Iterator<Postfix> iPostfix = schema.getPostfixes().iterator();iPostfix.hasNext();)↩
          {
352        Postfix postfix = iPostfix.next();
353        a = a.toLowerCase().replaceFirst(postfix.getPostfix().toLowerCase(), "");
354     }
355     return a;
356
357   }
358
359   %% similaridade de string
360   private double simString(String a, String b, int method) {
361       String nameA = a.replaceAll("-","").toLowerCase();
362       String nameB = b.replaceAll("-","").toLowerCase();
363
364       double vsimEntity=0;
365
366       System.out.println(nameA+" <-> "+nameB);
367
368       double max=0;
369       double vsimJairo = 0;
370       double vsimLeven = 0;
371       double vsimGran  = 0;
372       if (searchSynonym(nameA, nameB) == 0){
373        max= 1;
374       }
375       switch (method) {
376     case 1:
377        JaroWinklerDistance sim = new JaroWinklerDistance();
378        vsimJairo = sim.getDistance(nameA, nameB);
```

```
379             max = vsimJairo;
380             break;
381         case 2:
382             NGramDistance ngram = new NGramDistance();
383             vsimGran  = ngram.getDistance(nameA, nameB);
384             max = vsimGran;
385             break;
386         case 3:
387 
388             LevensteinDistance simL = new LevensteinDistance();
389             vsimLeven = simL.getDistance(nameA, nameB);
390             max = vsimLeven;
391             break;
392         }
393 
394         return max;
395     }
396 }
```

8.2 Anexo B - Implementação do operador *Merge*

```
1  package merge.engine;
2  
3  import java.util.Iterator;
4  import mapping.Correspondence;
5  import mapping.Definition;
6  import mapping.ElementHandler;
7  import mapping.Historic;
8  import mapping.Input;
9  import mapping.MetaModelHandler;
10 import mapping.Output;
11 import mapping.presentation.ITFMergeEngine;
12 import org.eclipse.emf.common.util.BasicEList;
13 import schemaDataBase.Schema;
14 import schemaDataBase.SchemaDataBaseFactory;
15 import schemamerging.MeAttribute;
16 import schemamerging.MeEntity;
17 import schemamerging.MergeS;
18 import schemamerging.MergeTargetAttrib;
19 import schemamerging.MergeTargetEntity;
20 import schemamerging.SchemamergingFactory;
21 import schemamerging.SchemamergingPackage;
22 
23 public class Merge implements ITFMergeEngine {
24    protected Historic mapHistoric;
25    protected Schema schemaDatabaseSorce;
26    protected Schema schemaDatabaseTarget;
```

```
27    protected Input inputs;
28    private Schema schemaMa;
29    private Schema schemaMb;
30    protected BasicEList<Output> outputs;
31    protected BasicEList<Definition> definitions;
32    protected BasicEList<Correspondence> correspondences;
33    protected BasicEList<ElementHandler> allElementHandlers;
34    protected BasicEList<MetaModelHandler> metaModelHandlers;
35    protected BasicEList<ElementHandler> elementsMatch = new BasicEList<ElementHandler>();
36    protected SchemamergingPackage schemamergingPackage = SchemamergingPackage.eINSTANCE;
37    protected SchemaDataBaseFactory schemaDataBaseFactory = SchemaDataBaseFactory.eINSTANCE;
38    protected SchemamergingFactory schemaMergingFactory = schemamergingPackage
39          .getSchemamergingFactory();
40    protected Definition objDef;
41    protected float weihtLow;
42    protected float weihtHight;
43    @Override
44    public MergeS mergeS() throws Exception {
45       MergeS mergeS = schemaMergingFactory.createMergeS();
46       objDef = (Definition) mapHistoric.getDefinition().get(0);
47       schemaDatabaseSorce = schemaMa;
48       schemaDatabaseTarget = schemaMb;
49       mergeS.setName(objDef.getName());
50       mergeS.setSchemaLeft(objDef.getSource().getMetaModelname());
51       mergeS.setSchemaRigth(objDef.getTarget().getMetaModelname());
52       String mapeamento = "src/model/" + mergeS.getName();
53        %% ----- criando lista de elementos correspondidos
54       correspondences = (BasicEList<Correspondence>) objDef.getRules();
55       for (Iterator<Correspondence> irules = correspondences.iterator(); irules
56             .hasNext();) {
57          Correspondence correspRules = irules.next();
58          Input inputRules = correspRules.getInput();
59          ElementHandler elementHandlerIn = inputRules.getHandler();
60          elementsMatch.add(elementHandlerIn);
61          for (Iterator<Output> ioutputRules = correspRules.getOutput()
62                .iterator(); ioutputRules.hasNext();) {
63             Output outputRules = ioutputRules.next();
64             ElementHandler elementHandlerOut = outputRules.getHandler();
65             elementsMatch.add(elementHandlerOut);
66          }
67          for (Iterator<Correspondence> inested = correspRules.getNested()
68                .iterator(); inested.hasNext();) {
69             Correspondence correspNested = inested.next();
70             Input inputNested = correspNested.getInput();
71             ElementHandler elHandlerIn = inputNested.getHandler();
72             elementsMatch.add(elHandlerIn);
73             for (Iterator<Output> ioutputNested = correspNested.getOutput()
74                   .iterator(); ioutputNested.hasNext();) {
75                Output outputNested = ioutputNested.next();
76                ElementHandler elementHandlerOut = outputNested
77                      .getHandler();
```

```
78              elementsMatch.add(elementHandlerOut);
79            }
80          }
81        }
82        %% -------------Fim lista de elementos correspondidos
83        %% Inicio lista de todos elementos dos modelos esquema de base de dados e noMatch
84        BasicEList<ElementHandler> allNoMatch = new BasicEList<ElementHandler>();
85        %% ------------ source
86        allElementHandlers = new BasicEList<ElementHandler>();
87        BasicEList<ElementHandler> elHandlerSource = (BasicEList<ElementHandler>) objDef
88              .getSource().getHandlers();
89        allElementHandlers.addAll(elHandlerSource);
90        %% Lista do elementos noMatch (Conjunto E do algoritmo de merging)
91        for (Iterator<ElementHandler> iNoMatch = elHandlerSource.iterator(); iNoMatch.hasNext↩
              ();) {
92           ElementHandler elementHandlers = iNoMatch.next();
93           if (!elementsMatch.contains(elementHandlers)) {
94              allNoMatch.add(elementHandlers);
95           }
96        }
97        for (Iterator<ElementHandler> ihandlersSource = objDef.getSource()
98              .getHandlers().iterator(); ihandlersSource.hasNext();) {
99           ElementHandler elementHandlers = ihandlersSource.next();
100          elHandlerSource = (BasicEList<ElementHandler>) elementHandlers
101                .getNested();
102          allElementHandlers.addAll(elHandlerSource);
103       }
104       %% ----------- target
105       BasicEList<ElementHandler> elHandlerTarget = (BasicEList<ElementHandler>) objDef
106             .getTarget().getHandlers();
107       allElementHandlers.addAll(elHandlerTarget);
108       %% ---- Lista do elementos noMatch
109       for (Iterator<ElementHandler> iNoMatch = elHandlerTarget.iterator(); iNoMatch
110             .hasNext();) {
111          ElementHandler elementHandlers = iNoMatch.next();
112          if (!elementsMatch.contains(elementHandlers)) {
113             allNoMatch.add(elementHandlers);
114          }
115       }
116       for (Iterator<ElementHandler> ihandlersTarget = objDef.getTarget()
117             .getHandlers().iterator(); ihandlersTarget.hasNext();) {
118          ElementHandler elementHandlers = ihandlersTarget.next();
119          elHandlerTarget = (BasicEList<ElementHandler>) elementHandlers
120                .getNested();
121          allElementHandlers.addAll(elHandlerTarget);
122             }
123       %%--- Fim de lista de todos elementos dos modelos esquema de base de dados e noMatch
124       %% ---- Inicio - Incluir entidades nao correspondentes nos esquemas - noMatch
125       for (Iterator<ElementHandler> iAllNoMatch = allNoMatch.iterator(); iAllNoMatch
126             .hasNext();) {
127          ElementHandler noMatchHandler = iAllNoMatch.next();
```

```
128         MeEntity entityNoMatch = schemaMergingFactory.createMeEntity();
129         entityNoMatch.setMerge(false);
130         entityNoMatch.setName(noMatchHandler.getName());
131         entityNoMatch.setElemReference(noMatchHandler.getName());
132         entityNoMatch.setOrigin(noMatchHandler.getOwner().getName());
133         %% inserir atributos da entidade
134         for (Iterator<ElementHandler> iNestedHendler = noMatchHandler
135               .getNested().iterator(); iNestedHendler.hasNext();) {
136            ElementHandler attribHandler = iNestedHendler.next();
137            MeAttribute attribEntityNoMatch = schemaMergingFactory
138                  .createMeAttribute();
139            attribEntityNoMatch.setElemReference(attribHandler.getName());
140            attribEntityNoMatch.setName(attribHandler.getName());
141            attribEntityNoMatch.setOrigin(attribHandler.getEnclosing()
142                  .getName());
143            attribEntityNoMatch.setOwnerAttribute(entityNoMatch);
144         }
145         mergeS.getMergeEntity().add(entityNoMatch);
146      }
147       %%  Fim entidades nao duplicadas nos esquemas
148       %%  Inserir no modelo schema merging entidades com match total e parcial
149      for (Iterator<Correspondence> irules = correspondences.iterator(); irules.hasNext();) ↩
          {
150         Correspondence elCorresp = irules.next();
151         float totalHandlerInput = elCorresp.getInput().getHandler().getNested().size();
152         Output output = (Output) elCorresp.getOutput().get(0);
153         float totalHandlerOutput =output.getHandler().getNested().size();
154         float totalCorresp = elCorresp.getNested().size();
155         float totalHandler = totalHandlerInput+totalHandlerOutput;
156         float dif = totalHandlerInput - totalCorresp;
157         float perc = (2*totalCorresp) / totalHandler;
158         %% total match
159         if (perc == 1) {
160            MeEntity meEntity = createEntity(elCorresp);
161            mergeS.getMergeEntity().add(meEntity);
162         } else {
163            String eInput = elCorresp.getInput().getName().toLowerCase();
164            ElementHandler source = elCorresp.getInput().getHandler();
165            %% baixo grau de match
166            if (perc <= weihtLow) {
167               for (Iterator<Output> outRules = elCorresp.getOutput().iterator(); outRules.↩
                   hasNext();) {
168                  Output outCorresp = outRules.next();
169                  ElementHandler target = outCorresp.getHandler();
170                  String eOutput = outCorresp.getName().toLowerCase();
171                  target.getOwner().getName();
172                  %% -- entidades low match, mas com nomes iguais (generalizar)
173                  if (eInput.equals(eOutput)) {
174                     MeEntity newEntidade = createEntity(elCorresp);
175                     String entitySource =capitalized(eInput);;
176                     String entityTarget =capitalized(eOutput);
```

```
177                     String nomeNewEntidade = entitySource+"."+entityTarget;
178                     newEntidade.setName(nomeNewEntidade);
179                     newEntidade.setMerge(true);
180                     newEntidade.setGeneralized(true);
181                     %% -- incluir atributo identificador de origem de tuplas
182                     MeAttribute originTuplas = schemaMergingFactory.createMeAttribute();
183                     originTuplas.setName("originTupla");
184                     originTuplas.setMerge(false);
185                     originTuplas.setOwnerAttribute(newEntidade);
186                     %% -- criar entidade especializada
187                     entityGeneralize(source, target, newEntidade);
188                     newEntidade.setMerge(true);
189                     mergeS.getMergeEntity().add(newEntidade);
190                 }
191             }
192         %% parcial match
193         }else{
194             %% -- alto grau de match
195             if(perc >= weihtHight){
196                 for (Iterator<Output> outRules = elCorresp.getOutput().iterator(); ↩
                        outRules.hasNext();) {
197                     Output outCorresp = outRules.next();
198                     ElementHandler target = outCorresp.getHandler();
199                     String eOutput = outCorresp.getName().toLowerCase();
200                     target.getOwner().getName();
201                     MeEntity newEntidade = createEntity(elCorresp);
202                     newEntidade.setMerge(true);
203                     && -- incluir atributo nao correspondido entre as entidades
204                     if(totalHandlerInput>totalHandlerOutput){
205                         for(Iterator<ElementHandler> iHandler = source.getNested().iterator↩
                                ();iHandler.hasNext();){
206                             ElementHandler handler = iHandler.next();
207                             if(!elementsMatch.contains(handler)){
208                                 MeAttribute meAttribute = schemaMergingFactory.↩
                                        createMeAttribute();
209                                 meAttribute.setName(handler.getName());
210                                 meAttribute.setElemReference(handler.getPath());
211                                 meAttribute.setMerge(false);
212                                 meAttribute.setOrigin(eInput);
213                                 meAttribute.setOwnerAttribute(newEntidade);
214                             }
215                         }
216                     }
217                     %% -- incluir atributo identificador de origem de tuplas
218                     MeAttribute origenEntidade = schemaMergingFactory.createMeAttribute();
219                     origenEntidade.setName("originTupla");
220                     origenEntidade.setMerge(false);
221                     origenEntidade.setOwnerAttribute(newEntidade);
222                     mergeS.getMergeEntity().add(newEntidade);
223                 }
224             }else{
```

```
225                  %% -- medio match
226                  for (Iterator<Output> outRules = elCorresp.getOutput().iterator(); ↩
                         outRules.hasNext();) {
227                     Output outCorresp = outRules.next();
228                     ElementHandler target = outCorresp.getHandler();
229                     String eOutput = outCorresp.getName().toLowerCase();
230                     target.getOwner().getName();
231                     %% -- entidades com nomes iguais, mas com alguns atributos nao ↩
                            correspondidos (entidade generalizada)
232                     if (eInput.equals(eOutput)) {
233                        MeEntity newEntidade = createEntity(elCorresp);
234                        String entitySource =capitalized(eInput);;
235                        String entityTarget =capitalized(eOutput);
236                        String nomeNewEntidade = entitySource+"."+entityTarget;
237                        newEntidade.setName(nomeNewEntidade);
238                        newEntidade.setMerge(true);
239                        newEntidade.setGeneralized(true);
240                        %% -- incluir atributo identificador de origem de tuplas
241                        MeAttribute originTuplas = schemaMergingFactory.createMeAttribute();
242                        originTuplas.setName("originTupla");
243                        originTuplas.setMerge(false);
244                        originTuplas.setOwnerAttribute(newEntidade);
245                        %% -- criar entidade especializada
246                        entityGeneralize(source,target,newEntidade);
247                        mergeS.getMergeEntity().add(newEntidade);
248                     }else{
249                        MeEntity newEntidade = createEntity(elCorresp);
250                        String entitySource =capitalized(eInput);;
251                        String entityTarget =capitalized(eOutput);
252                        String nomeNewEntidade = entitySource+"."+entityTarget;
253                        newEntidade.setName(nomeNewEntidade);
254                        newEntidade.setMerge(true);
255                        newEntidade.setGeneralized(true);
256                        %% -- incluir atributo identificador de origem de tuplas
257                        MeAttribute originTuplas = schemaMergingFactory.createMeAttribute();
258                        originTuplas.setName("originTupla");
259                        originTuplas.setMerge(false);
260                        originTuplas.setOwnerAttribute(newEntidade);
261                        %% -- criar entidade especializada
262                        entityGeneralize(source,target,newEntidade);
263                        newEntidade.setMerge(true);
264                        mergeS.getMergeEntity().add(newEntidade);
265                     }
266                  }
267               }
268            }
269         }
270      }
271      SaveMerge salvar = new SaveMerge();
272      salvar.saveSchemaMerge(mergeS, mapeamento + ".schemamerging");
273      return mergeS;
```

```
274    }
275    @Override
276    public void init(Schema schemaMa, Schema schemaMb, Historic historic, float wL, float wH)↩
           throws Exception {
277       this.schemaMa = schemaMa;
278       this.schemaMb = schemaMb;
279       this.mapHistoric = historic;
280       this.weihtLow  = wL;
281       this.weihtHight= wH;
282    }
283    protected String capitalized(String string){
284       boolean upper = true;
285       String s = string;
286       s = s.substring(0, 1).toUpperCase() + s.substring(1);
287       return s;
288    }
289    protected MeEntity createEntity(Correspondence elCorresp){
290       MeEntity newEntity = schemaMergingFactory.createMeEntity();
291       newEntity.setMerge(true);
292       newEntity.setName(elCorresp.getInput().getName());
293       newEntity.setElemReference(elCorresp.getInput().getName());
294       newEntity.setOrigin(elCorresp.getInput().getHandler().getOwner().getName());
295       for (Iterator<Output> outRules = elCorresp.getOutput().iterator(); outRules.hasNext()↩
              ;) {
296          Output outCorresp = outRules.next();
297          MergeTargetEntity targetEntity = schemaMergingFactory.createMergeTargetEntity();
298          targetEntity.setName(outCorresp.getName());
299          targetEntity.setElemReference(outCorresp.getHandler().getName());
300          targetEntity.setOrigin(outCorresp.getHandler().getOwner().getName());
301          targetEntity.setOwnerEntityTarget(newEntity);
302       }
303       for (Iterator<Correspondence> iNesteds = elCorresp.getNested().iterator(); iNesteds.↩
              hasNext();) {
304          Correspondence nestedCorresp = iNesteds.next();
305          MeAttribute attribEntityAllMatch = schemaMergingFactory.createMeAttribute();
306    attribEntityAllMatch.setElemReference(nestedCorresp.getInput().getHandler().getName());
307          attribEntityAllMatch.setName(nestedCorresp.getInput().getName());
308          attribEntityAllMatch.setOrigin(elCorresp.getName());
309          attribEntityAllMatch.setOwnerAttribute(newEntity);
310          attribEntityAllMatch.setMerge(false);
311          for (Iterator<Output> iOutputs = nestedCorresp.getOutput()
312                .iterator(); iOutputs.hasNext();) {
313             Output outCorresp = iOutputs.next();
314             MergeTargetAttrib targetAttrib = schemaMergingFactory
315                   .createMergeTargetAttrib();
316             targetAttrib.setName(outCorresp.getName());
317             targetAttrib.setOrigin(outCorresp.getName());
318             targetAttrib.setOwnerTarget(attribEntityAllMatch);
319             attribEntityAllMatch.setMerge(true);
320          }
321       }
```

```
322         return newEntity ;
323     }
324     protected MeEntity entityGeneralize(ElementHandler source, ElementHandler target, ↩
            MeEntity newEntidade){
325         int qtAttribSource = source.getNested().size();
326         int qtAttribTarget = target.getNested().size();
327         %% -- criar entidade especializada
328         MeEntity newEntityS = schemaMergingFactory.createMeEntity();
329         newEntityS.setGeneralized(true);    newEntityS.setName(capitalized(objDef.getSource().↩
                getName()+"."+capitalized(source.getName())));
330         newEntityS.setElemReference(source.getPath());
331         newEntityS.setOrigin(source.getOwner().getName());
332         for(Iterator<ElementHandler> iHandler = source.getNested().iterator();iHandler.hasNext↩
                ();){
333             ElementHandler handlerSource = iHandler.next();
334             MeAttribute newAttrib = schemaMergingFactory.createMeAttribute();
335             if(!elementsMatch.contains(handlerSource)){
336                 newAttrib.setName(handlerSource.getName());
337                 newAttrib.setOwnerAttribute(newEntityS);
338             }
339         }
340         newEntidade.getNestedEntity().add(newEntityS);
341         boolean hasAttrib=false;
342         MeEntity newEntityT = schemaMergingFactory.createMeEntity();
343         newEntityT.setGeneralized(true);    newEntityT.setName(capitalized(target.getOwner().↩
                getName())+"."+capitalized(target.getName()));
344         newEntityT.setElemReference(target.getPath());
345         newEntityT.setOrigin(target.getOwner().getName());
346         for(Iterator<ElementHandler> iHandler = target.getNested().iterator();iHandler.hasNext↩
                ();){
347             ElementHandler handlerTarget = iHandler.next();
348             MeAttribute newAttrib = schemaMergingFactory.createMeAttribute();
349             if(!elementsMatch.contains(handlerTarget)){
350                 newAttrib.setName(handlerTarget.getName());
351                 newAttrib.setOwnerAttribute(newEntityT);
352                 hasAttrib = true;
353             }
354         }
355         if (hasAttrib){
356         newEntidade.getNestedEntity().add(newEntityT);
357         }
358         return null;
359     }
360 }
```

8.3 Anexo C - Definição de tranformação de *Database Integrated Model* para Modelo SQL

Listagem 8.1: Definições de trasnformações do *database integrated model* para modelo SQL

```
1  -- @path SQL=/IntegradoParaSQL/metamodels/mmSQL.ecore
2  -- @path Integrado=/IntegradoParaSQL/metamodels/idb.ecore
3
4  module integrado2sql;
5  create OUT : SQL from IN : Integrado;
6
7  helper context Integrado!IdbAttribute def: isPK(): Boolean =
8     if not self.ownerPKAttr.oclIsUndefined() then true
9     else false
10    endif;
11 helper context Integrado!IdbAttribute def: isFK(): Boolean =
12    if not self.ownerFKAttrib.oclIsUndefined() then true
13    else false
14    endif;
15 helper context Integrado!IdbAttribute def: isK(): Boolean =
16    if not self.ownerFKAttrib.oclIsUndefined() then true
17    else if not self.ownerFKAttrib.oclIsUndefined() then true
18    else false
19    endif endif;
20
21 helper context Integrado!IdbAttribute def: getOwner(): SQL!Element =
22    if not self.ownerFKAttrib.oclIsUndefined() then
23       self.ownerFKAttrib.ownerKey
24    else if not self.ownerPKAttr.oclIsUndefined() then
25       self.ownerPKAttr.ownerKey
26       else
27          self.ownerAttrib
28       endif
29    endif;
30
31 helper context Integrado!IdbAttribute def: getType(): String = --SQL!Types ='VARCHAR';
32    if self.type = #STRING then 'VARCHAR'
33    else if self.type = #CHAR then 'VARCHAR'
34    else if self.type = #BOOLEAN then 'BOOLEAN'
35    else if self.type = #INTEGER then 'INT'
36    else if self.type = #REAL then 'REAL'
37    else 'INT'
38    endif endif endif endif endif;
39
40 rule Entity2Table{
41    from entity: Integrado!IdbEntity
42    to table: SQL!Table (
43       name <- entity.name
44
45    )
46 }
47
48 rule Attrib2Column{
49    from attrib: Integrado!IdbAttribute
```

```
50    to column: SQL!Column(
51       name <- attrib.name,
52       table <- attrib.getOwner(),
53       type <- attrib.getType(),
54       length <- attrib.length
55    )
56 }
57
58 rule Definition2Database{
59    from defin: Integrado!Definition
60    to   db: SQL!Database(
61       name <- defin.name,
62       tables <- defin.idbEntitys
63    )
64 }
65
66 rule IdbPrimaryKey2PrimaryKey{
67    from pkIdb:Integrado!IdbPrimaryKey
68    to  pk: SQL!PrimaryKey(
69       name <- pkIdb.name,
70       table <- pkIdb.ownerKey
71    )
72 }
73 rule IdbForeignKey2ForeignKey{
74    from fkIdb: Integrado!IdbForeignKey
75    to  fk: SQL!ForeignKey(
76       name <- fkIdb.name,
77       column <- fkIdb.fkAttrib,
78       table <- fkIdb.ownerKey,
79       tabelaReferencia <- fkIdb.RefereceEntity
80    )
81 }
```

Listagem 8.2: *Query* de criação dos aquivos SQL *scritpt*

```
1    query sql2mysql = SQL!Table.allInstances()->collect(x | x.toString(). writeTo(
2    'C:/Temp/SQL/script-'+ x.name+'.sql'));
3
4 uses x_SqlToCode;
```

Listagem 8.3: *Library* usada pela Query da listagem anterior.

```
1 library x_SqlToCode;
2
3 helper context SQL!Table def : toString() : String =
4    'DROP TABLE IF EXISTS ' + self.name + ';\n\n' +
5    'CREATE TABLE ' + self.name + ' (' +
```

```
6     self.elements -> iterate(i; acc : String = '' | acc + i.toString() + ' ' )+
7     '\n);';
8
9  helper context SQL!ForeignKey def: getNameAttribFk: String =
10    self.column -> collect(col | col.name);
11
12 helper context SQL!Column def : toString() : String =
13    '\n'+self.name+' '
14    +if self.length > 0 then
15       self.type + '('+self.length+')'
16     else
17       self.type
18     endif
19    +',';
20
21 helper context SQL!PrimaryKey def : toString() : String =
22    '\nPRIMARY KEY('+self.name+ ')';
23
24 helper context SQL!ForeignKey def : toString() : String =
25    ',\nCONSTRAINT `'+self.name+'` FOREIGN KEY (`' +
26        self.getNameAttribFk+ ') REFERENCES ' +self.tabelaReferencia +' ('+self.↩
            getNameAttribFk+')';
```